Problem description

With the trend of increasingly parallel processor architectures, we aim to investigate how these can be utilized in database queries. The task is to study existing techniques for performing database queries using multi-core processors, development of new algorithms, and experimental evaluation.

Assignment given: 15 January 2013
Supervisor: Kjetil Nørvåg

TDT4900 - Datateknikk og informasjonsvitenskap, masteroppgave

Spring 2013

Database Operations on Multi-Core Processors

Author:
Stian Liknes

Supervisor:
Kjetil Nørvåg

Abstract

The focus of this thesis is on investigating efficient database algorithms and methods for modern multi-core processors in main memory environments. We describe central features of modern processors in a historic perspective before presenting a number of general design goals that should be considered when optimizing relational operators for multi-core architectures. Then, we introduce the skyline operator and related algorithms, including two recent algorithms optimized for multi-core processors. Furthermore, we develop a novel skyline algorithm using an angle-based partitioning scheme originally developed for parallel and distributed database management systems. Finally, we perform a number of experiments in order to evaluate and compare current shared-memory skyline algorithms.

Our experiments reveals some interesting results. Despite of having an expensive pre-processing step, the angle-based algorithm is able to outperform current best-performers for multi-core skyline computation. In fact, we are able to outperform competing algorithms by a factor of 5 or more for anti-correlated datasets with moderate to large cardinalities. Included algorithms exhibit similar performance characteristics for independent datasets, while the more basic algorithms excel at processing correlated datasets. We observe similar performance for two small real-life datasets. Whereas, the angle-based algorithm is more efficient for a work-intensive real-life dataset containing more than 2M 5-dimensional tuples.

Based on our results we propose that database research targeted at shared-memory systems is focused not only on basic algorithms but also more sophisticated techniques proven effective for parallel and distributed database management systems. Additionally, we emphasize that modern processors have very fast inter-thread communication mechanisms that can be exploited to achieve parallel speedup also for synchronization-heavy algorithms.

Sammendrag

Fokuset i denne oppgaven er å forske på effektive algoritmer og metoder rettet mot moderne flerkjernearkitekturer i en databasesammenheng. Vi beskriver sentrale aspekter ved moderne prosessorer i et historisk perspektiv, før vi presenterer en rekke generelle konstruksjonsmål for relasjonsoperatorer i flerkjernesystemer. Deretter beskriver vi skyline-operatoren med relaterte algoritmer og utvikler en ny skyline-algoritme som bruker en vinkelbasert partisjoneringsmetode nylig publisert i sammenheng med parallelle og distribuerte databasesystemer. Avslutningsvis gjør vi en rekke eksperimenter for å evaluere og sammenligne skyline-algoritmer kjørt på moderne prosessorer.

Til tross for et tidskrevende preprosesseringstrinn, er den vinkelbaserte algoritmen i stand til å utkonkurrere de meste effektive skyline-algoritmene optimalisert for flerkjerneprosessorer. Faktisk er vi i stand til å utkonkurrere den beste algoritmen med en faktor på fem eller mer for anti-korrelerte datasett med moderate til store kardinaliteter. Algoritmene oppnår lignende ytelseskarakteristikker for datasett med uavhengig distribusjon, mens de mer grunnleggende algoritmene utmerker seg ved behandling av korrelerte datasett. Samtlige algoritmer oppnår noenlunde lik ytelse for to små ikke-syntetiske datasett, mens den vinkelbaserte algoritmen er mer effektiv for et arbeidskrevende ikke-syntetisk datasett.

Basert våre resultater, foreslår vi at databaseforskning relatert til flerkjernesystemer ikke bare fokuseres på grunnleggende algoritmer, men også på mer omfattende teknikker fra parallelle og distribuerte databasesystemer. I tillegg understreker vi at moderne prosessorer er veldig effektive på kommunikasjon mellom parallelle tråder, og dermed bedre egnet til parallell utførelse av synkroniseringstunge algoritmer enn mer tradisjonelle parallelle og distribuerte systemer.

Contents

Chapter 1

Introduction

Algorithms used in databases management systems (DBMS) have traditionally focused on external factors like I/O-operations. Disk access is an order of magnitude slower than memory access and minimizing I/O-access is a necessity working on datasets exceeding main memory capacity. However, a rapid increase in main memory capacity are making memory based algorithms more relevant.

There are two main areas for memory based algorithms in database systems: main memory database management systems (MMDB) and disk resident database management systems (DRDB) with very large caches. MMDBs keep all data in memory and thereby completely avoid the I/O bottleneck, consequently traditional algorithms can not be regarded as optimal for these systems without close examination. Main memory bandwidth and compute capacity are becoming dominant factors in algorithm performance. In addition, DRDBs with large caches can complete many operations without intermediate I/O-access. This leads to a need for algorithms that use memory bandwidth efficiently and are capable of optimal cache patterns without making compromises to minimize I/O access.

For a long time, processor manufacturers increased compute power by increasing the operating frequency and placing transistors closer together. Unfortunately, this is no longer an option, because increasing operating frequency further achieves diminishing performance gains compared to associated power requirements. The so-called power wall has been reached, and processor manufacturers have been forced to find other ways to utilize chip-transistors. Nevertheless, Moore's law is still valid, the number of transistors per chip continue to increase. To better utilize these resources manufacturers have started increasing the number of cores per processor. This is an efficient method of increasing compute power without exponentially increasing the processors power consumption.

Another trend is that processors are providing greater capacity for data parallelism in the form of single instruction multiple data (SIMD) processing. SIMD allows one operation to be performed for multiple inputs without additional CPU cycles, e.g. multiply four values at the price of one. For instance both Intel and AMD are supporting streaming SIMD extensions 4 (SSE4), providing many opportunities for data parallelism. Currently, general purpose processors have limited SIMD support, SSE4 supports signed multiplication of no more than four 32-bit integers in one operation. However, a speedup of four in an inner loop, with practically no increase in work are definitely worth pursuing. By combining task- and data parallelism, one can achieve significant performance gains in suitable algorithms.

In [43], Stonebraker compares the three primary parallel architectures: shared-memory (SM), shared-disk (SM), and shared-nothing (SN). In SM systems, all processors (or cores) share a common central memory. Each processor in a SM system has private memory, while all processors share a collection of one or more external disks. For SN systems, nothing is shared, this is the case of distributed systems and have been the primary focus in database management systems (DMBS). Stonebraker argued that SM systems did not scale to a large number of processors, hence they were less interesting than the other systems. He also observed that SD systems does not excel in any area compared to the other two, and concluded that SN systems is the primary target for DBMS. Not only did SN show the best characteristics for scaling, it also matched the current marketplace interest, distributed database management systems. At the time (1985), this was the obvious contender for further research. However, with recent trends in processor- and memory development, SM systems are becoming more prominent. Multi-core processors are highly-popular SM systems, and developers must resort to SM programming to utilize the increasing parallel compute capacity.

The increased interest in SM programming has led to an number of frameworks [30] to help programmers gradually parallelize existing algorithms, and to develop completely new methods. These frameworks address some of the issues mentioned by Stonebraker in [43], for instance, frameworks provide practical solutions for concurrency control and management of hot spots. OpenMP is a popular choice for incrementally adding parallelism to an algorithm, this framework allows the developer to tag parallel regions as parallel using compiler directives. Critical sections can be tagged as critical to provide a simple concurrency control.

Memory based databases are also obtaining increased interest. Commercial products like IBM solidDB and Oracle TimesTen are examples of high-performance relational MMDBs in use today. By managing data in memory, and optimizing data structures and access algorithms accordingly, database operations execute with maximum efficiency, achieving dramatic gains in responsiveness and throughput [33].

For the join operator, Blanas et al. [4] evaluate hash based algorithms [6] in a shared-memory context and find that algorithms are most efficient with no pre-partitioning. This somewhat surprising, because the partitioning phase improves cache locality for subsequent phases [42], thereby increasing algorithm performance in a sequential system. Similar results are reported in [37], where they achieve impressive performance using a linear partitioning scheme instead of exploiting well-known geometric properties for multi-core [1] skyline computation. However, the compute-intensive skyline algorithm has not yet been tested with a sophisticated partitioning technique

[1]We use the terms multi-core and shared-memory interchangeably to describe a multi-core shared-memory system

in the multi-core context.

In this thesis we explore the multi-core landscape in a database context by developing and evaluating a novel algorithm for the skyline operator. Furthermore, we perform several experiments in order to evaluate and compare the current best-performing skyline algorithms optimized for multi-core architectures. Based on our results, we perform the first comprehensive comparison between current multi-core skyline algorithms, revealing some interesting characteristics along the way. We address the following research questions:

RQ1 How can we efficiently exploit multi-core architectures when implementing database operators? Are there cases where this is impossible?

RQ2 How can we determine if an algorithm or an operator is viable for multi-core optimizations?

RQ3 Is it reasonable to regress into more basic algorithms in order to exploit the increasing parallel compute power in modern processors?

RQ4 Can pre-processing techniques from parallel and distributed systems be efficient in a shared-memory context where inter-thread communication is an order of magnitude less expensive?

The thesis is organized as follows. First, modern processor architectures are described in a historical perspective, and a number of design goals for multi-core algorithms in a database context are presented. Second, the skyline operator is introduced with associated sequential and parallel algorithms. Third, we develop an efficient algorithm for skyline computation based our design goals. Fourth, we compare our algorithm to state-of-the-art skyline algorithms developed for multi-core architectures using a variety of experiments, and discuss the implication of the results. Finally, we present our conclusions, and suggest a direction for further research.

Chapter 2

Modern processor architectures

In this chapter, we introduce modern processor architectures and suggest a number of design goals that should be considered when developing database algorithms for multi-core processors.

2.1 von Neumann architecture

Modern processors are increasingly complex constructions, and come in many forms. Vendors like Intel, Sun and ARM have focused on different markets and optimized their architectures for a range of uses (desktop, mobile, server, and so on). However, most of the architectures use a similar architecture based on the von Neumann architecture [52] shown in Figure 2.1.

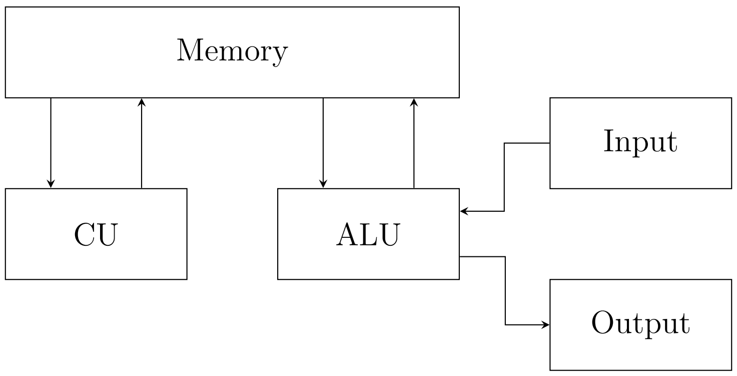


Figure 2.1: von Neumann architecture

The von Neumann architecture consists of a control unit and an arithmetic logic unit (ALU). The control unit (CU) fetches the code of all the instructions in the program and directs the operation of the ALU. Meanwhile, the ALU does the actual calculation. The input and output units are allows a person to interact with the machine using registers. Program and data is stored in the memory unit. Due to the fact that the architecture is sequential in nature, and has a well known data transfer bottleneck, it is not used in practice without modifications. It is described here as a basis for the more complex architectures.

To overcome the von Neumann bottleneck, the shared data and instruction bus has been replaced by multiple buses (in a simple architecture there are typically three buses: instruction, data, and control). By using multiple buses, instructions and data can be fetched from memory simultaneously, resulting in a significant performance boost.

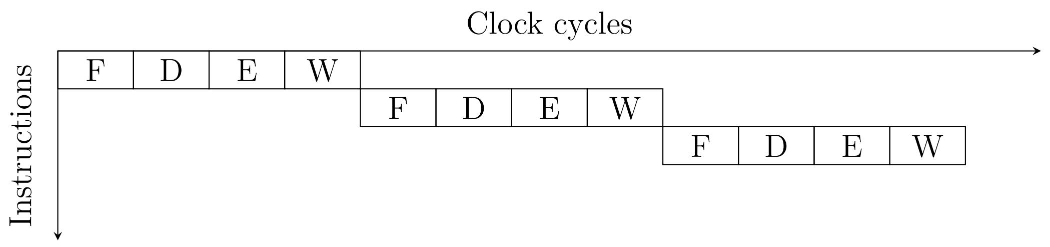


Figure 2.2: Instruction flow of a sequential processor. The letters F, D, E, W indicates fetch, decode, execute, writeback respectively

2.2 Cache structures

In order to decrease latency and contention while accessing data, modern processors use deep cache hierarchies. Both instructions and data are cached close to the processing unit so that frequently used data can be read without slow memory operations. Figure 2.5 shows a processor with two level caching, L1 is smaller and faster than L2, which is smaller and faster than main memory. Processors use both temporal and spatial locality to determine which values should be placed in cache.

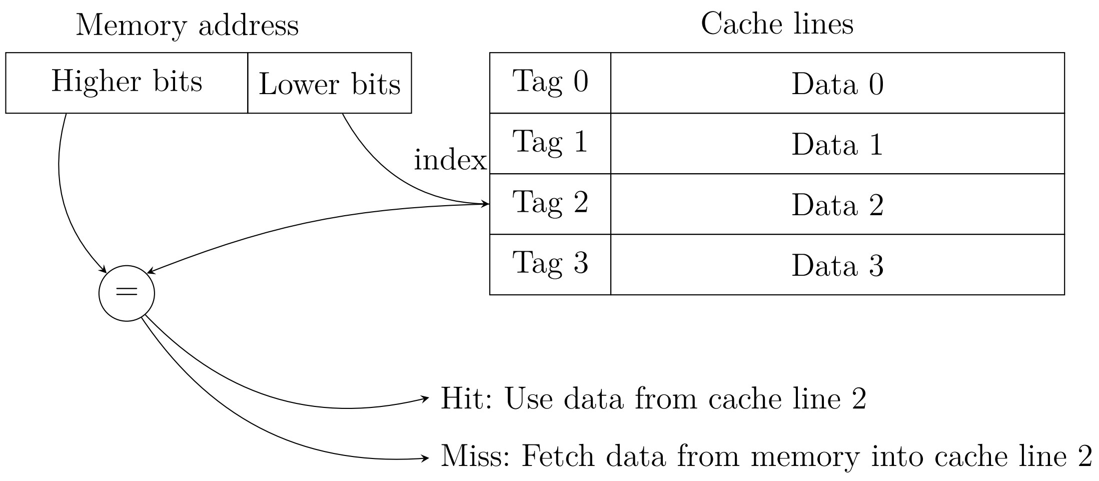


Figure 2.3: Cache lookup. Line is first found using the lower bits as an index into cache, higher parts of memory address is subsequently compared with tag to determine if a match is found

Data is transferred from main memory to cache in blocks (i.e. cache lines) to ensures spatial locality, values close to each other are fetched at the same time and put in the same location (even if they have not been requested yet), see Figure 2.3. Temporal locality is ensured by keeping recently addressed data in cache. Typically a cache allows data from a particular memory address to be placed at a handful of predetermined cache lines, allowing very fast lookups. A potential problem with this technique is cache conflicts: If a program repeatedly access two memory locations which happen to map to the same cache line, the cache must keep storing and loading from main memory and the cache hierarchy actually decrease overall performance. To avoid this problem, most processors use set-associative caches, where each memory location can map to a number of different cache lines. However, cache conflicts are still possible, therefore one needs to consider locality and association when developing performance critical code.

2.3 Instruction-level parallelism

An issue with the von Neumann model that it is using an inherently sequential execution model. Instructions are executed one after another as depicted in Figure 2.2. By closer examination, one will notice that each instruction has at least four steps, fetch, decode, execute, and writeback. Modern processors use pipelined execution to perform each step in parallel as depicted in Figure 2.4. This method is referred to as instruction-level parallelism (ILP) and has been widely used since the 1970s. Deeper pipelines can be constructed by further subdividing instructions and use so-called superpipelining. However, subdividing instructions has a cost, therefore it is uncommon to have very deep pipelines. Because the execute step actually consists of a number of different operations such as floating-point and integer calculation, performance can also be increased using superscalar pipelining. That is, executing multiple instructions in parallel, each in its own functional unit. Today, virtually every processor is superpipelined-superscalared.

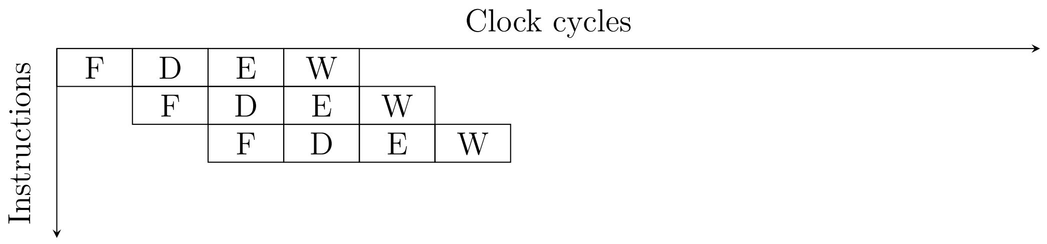


Figure 2.4: Instruction flow of a pipelined processor. The letters F, D, E, W indicates fetch, decode, execute, writeback respectively

ILP is a great advance from the vanilla von Neumann model, but it is has some issues. The main limiting factors for ILP is instruction dependencies and branches. Instructions that depend on each other can not be executed in parallel, and will not experience any performance gain from pipelining. For branching instructions, the processor has no way of knowing which operation will be next and can not load the relevant instruction into the pipeline before the branch has been resolved. To overcome the branching issue, processors typically use some form of branch prediction algorithm to load the most likely instruction. This instruction is executed in the pipeline, and if the prediction was correct the processor can continue as normally. If the prediction was wrong, however, the processor have just wasted a bunch of cycles and has to go back to the branch and perform the correct instruction. An approach to avoid the branching problem is to use predicated instructions like conditional move. Predicated instructions are executed as normal, but will only commit itself if condition is true. In any case, ILP will have unused cycles when one instruction is waiting for a branch or dependency. This is somewhat alleviated using out-of-order execution (OOE; instructions are reordered to allow for improved ILP processing), but OOE is an expensive solution and only provides a limited speedup.

2.4 Task parallelism

Simultaneous multithreading (SMT; or Hyper-Threading) have been introduced to further increase available resources. If additional instructions are not available from the current thread, instructions from other (independent) threads can be placed in the pipeline and execute in parallel. To achieve this, processors typically present one physical processor core as two or more logical processors to the operating system. SMT is relatively cheap to implement and, in some cases, introduce an significant performance gain. A weakness of this technique is that multiple threads executed on the same core may cause contention for some resources (like the bus and cache structures), this can limit performance gain, or in the worst case reduce overall performance.

Multi-core processors are similar to SMT in that multiple threads can run in parallel. However, in multi-core processors, the entire core is duplicated, including registers and cache, see Figure 2.5. This solves the contention problem in many cases, but shared components are not eliminated entirely and some contention can still occur (e.g. shared-memory bus and lower level cache). To allow cores to communicate, lower level cache is normally shared, while keeping higher level cache private. There are many variations of this scheme, some processors may have completely independent cores, not sharing any cache. Processors can also combine SMT and multi-core in creative ways, like the AMDs Bulldozer design where the processor includes multiple independent cores for integer operations, and shared units for floating-point operations.

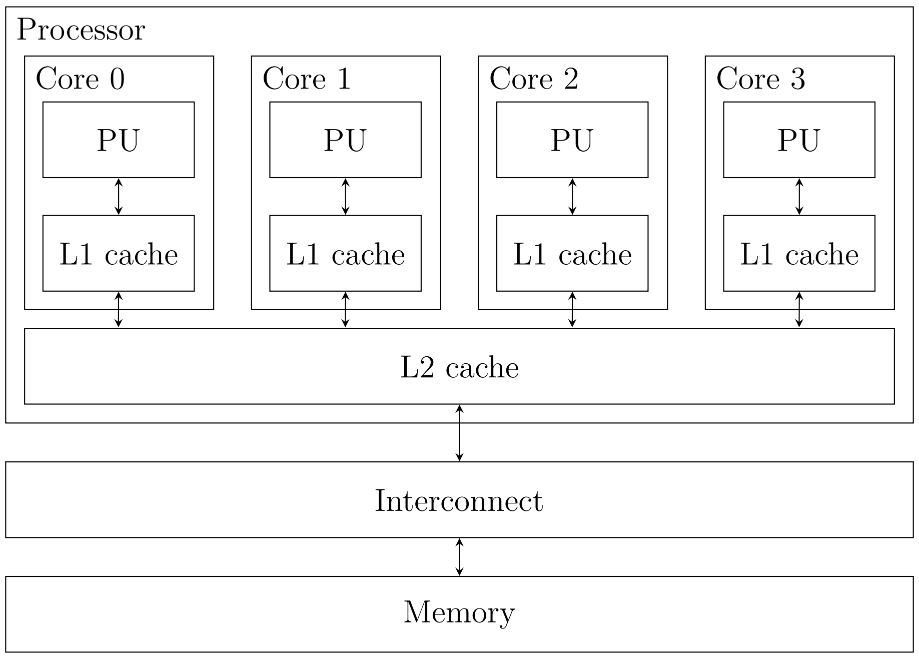


Figure 2.5: Multi-core processor with shared L2 cache and uniform memory access. PU is short for processing unit, including fetch, decode, execute and writeback components

2.5 Data parallelism

Techniques to exploit parallel potential mentioned thus far have focused mainly on task parallelism, that is to execute different tasks in parallel. Another approach is to look for ways to perform the same operation on multiple values in parallel to induce data parallelism. This idea was extensively used in the old supercomputers, with special purpose processors. It is commonly called vector processing, or single instruction, multiple data (SIMD) and have been increasingly integrated into modern processors. SIMD operations are available for a subset of the operations supported by a processor and, in most cases, has to be explicitly programmed. The end result is that multiple values processed using SIMD can be performed the same number of CPU cycles as a single value without SIMD, see Figure 2.6.

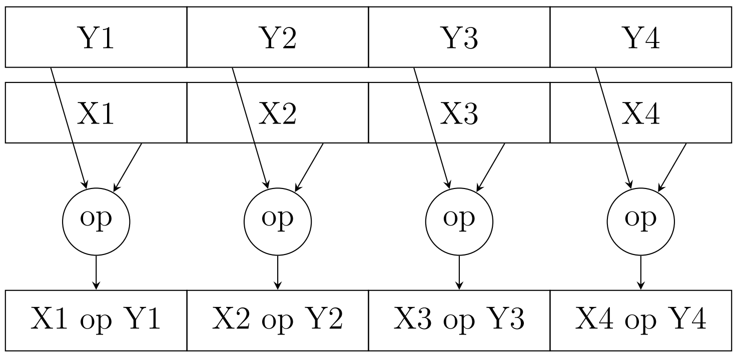


Figure 2.6: SIMD operator to process four values in one cycle

SIMD operations are implemented based on observations like the fact that four 8 bit additions can be performed using a modified 32 bit add. Both AMD and Intel provides SIMD support in their processors which they call 3DNow! and streaming SIMD extension (SSE) [22] respectively.

2.6 Shared memory systems

Multi-core processors are typically configured as a shared-memory system: a collection of independent cores connected to a memory system via an interconnect network. There are two principal types of shared-memory systems: uniform memory access (UMA), and non-uniform memory access (NUMA) [34].

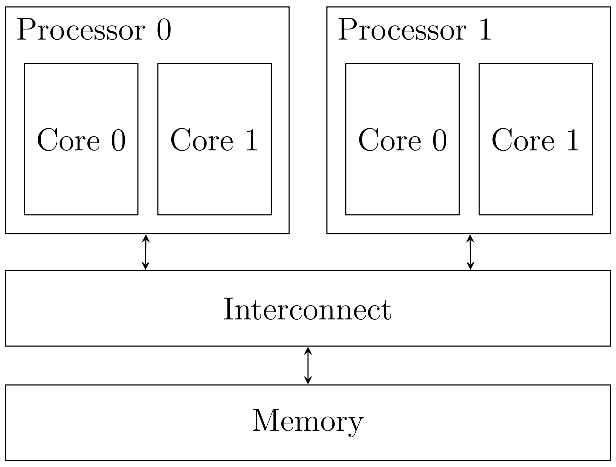


Figure 2.7: UMA architecture with two processors

In UMA systems all processors are connected directly to main memory, see Figure 2.7. Each core can access data from all of the other cores directly. Access to all memory locations are the same for each core. UMA systems main advantage in relation to NUMA systems is their simplicity.

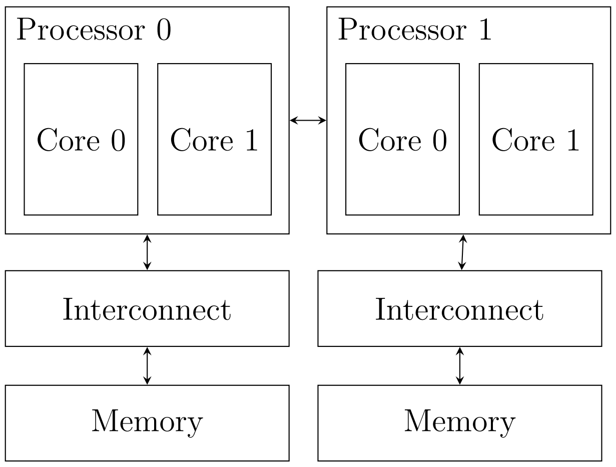


Figure 2.8: NUMA architecture with two processors

In NUMA systems each processor is directly connected to a block of main memory, and the processors can reach each others data through special hardware built into the processors, see Figure 2.8. A memory location that a processor is directly connected to can be accessed faster than a memory location that must be accessed through another chip [34]. Advantages of the increased complexity is that the system can address a larger memory space, and directly memory access typically is faster than in UMA systems. It is common to use UMA for the cores in one processor, and NUMA between processors using a fast bus.

It is also common for multi-core processors to include a limited capacity for data parallelism. Typically, there are a set of float-point and integer operations that can be executed simultaneously using specialized data structures as explained in Section 2.5

2.6.1 False sharing

False sharing is a performance-degrading usage pattern that can occur in systems with coherent caches. If a thread periodically access data that will never be altered by any other thread, but that data shares a cache block with data that is altered by other threads, the caching protocol may force the first thread to reload the whole cache line [40]. In other words, the first thread will bear the caching overhead required by true shared access of a resource, despite of being the only thread that actually modifies the cached data.

There are several techniques for avoiding false sharing. A simple approach is to pad data structures so that each instance is exactly one cache line. When two threads work on different padded data structures, data are placed in different cache lines, thus avoiding false sharing. Another technique is to keep a thread private copy of the data being worked on. In most divide-and-conquer based algorithms, the latter occurs automatically by data partitioning.

2.7 Algorithm design

Shared-nothing is by far the dominating architecture in parallel and distributed database management systems today [38, 57, 51, 46]. However, recent advances in processor technology are leading to a new generation of algorithms based on shared-everything in-memory processing [1, 54, 25].

Multi-core processors are most efficient for CPU-intensive tasks that require little synchronization. However, multi-core processors are also efficient for inter-thread communication [31]. Specifically, Meneghin et al. shows that fine grained parallelism, with unit of work in the order of a few microseconds, can be implemented efficiently on modern architectures. This indicates that algorithms should be able to utilize shared data structures for CPU-intensive tasks. In order to develop efficient algorithms optimized for modern processor architectures, we propose the following design goals:

Paralleism To utilize the potential of multiple cores, it is essential that algorithms are designed with a high degree of parallelism. Independent tasks should be identified and executed in parallel wherever possible. In the terms of Amdahl's law [2, 21], we would like to maximize the parallel part of the algorithm so that we are be able to utilize an great number of parallel cores.

Scalability Algorithms should scale up to a large number of threads [26]. It should not be necessary to modify algorithms to support an increased number of cores.

Communication Communication between different threads of execution will generally reduce the parallel portion of an algorithm due to synchronization costs and should be limited. However, in some cases shared data structures can be used to reduce memory requirements or to avoid expensive merging phases. Shared data structures should not be discarded without carefully considering the performance impact of an alternative solution.

Memory Memory bandwidth, volume, and locality should always be considered when designing an algorithm [19]. Generally, one should limit bandwidth and volume as much as possible, avoid random reading/writing, and place related data close together to achieve good data locality. Additionally, threads should be designed to avoid cache-conflicts. Specifically, threads should avoid writing to memory areas close to memory areas written by other threads. It is often an advantage to keep a private copy of the data being worked on in each thread in order to avoid false sharing.

Fairness Workloads should be evenly distributed among available threads. In a worst-case scenario, one thread is assigned all of the work, making it impossible to exploit any parallel compute power.

Note that we did not include IO-operations in the proposed design goals. This is because we target in-memory computations, where all data is placed exclusively in main memory, making IO-operations irrelevant. This is consistent with trends in shared-memory database research, and is useful for evaluating algorithms targeted at systems with an increasingly large main memory capacity.

Chapter 3

The skyline operator

Consider a database containing price and locality information about hotels. If a user wants to find the cheapest hotel that is closest to the city centre, the DBMS cannot always return one simple answer. The price typically increase as distance decrease, leaving the user with multiple possibilities that are equally ”good” as illustrated in Figure 3.1. No matter how the user weigh his personal references towards price or distance, the best hotel will be placed in the skyline (the dashed line in Figure 3.1). Specifically, the best hotel will be as good or better than all other hotels in all dimensions.

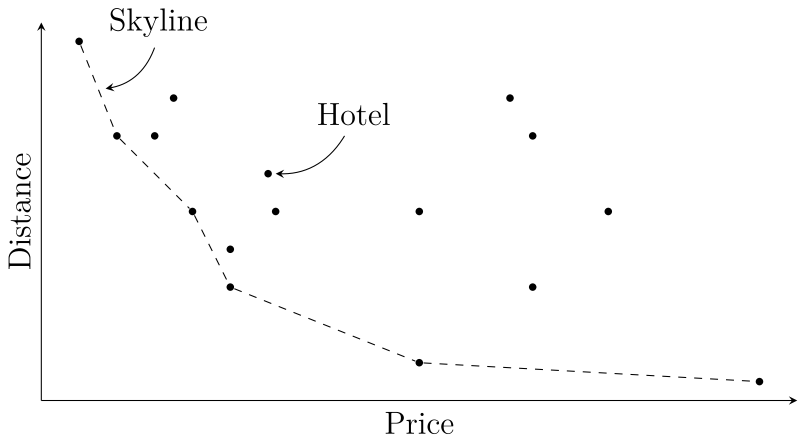


Figure 3.1: Skyline of Hotels, searching for minimal price and distance. Each point indicates a hotel while the dashed line traverse all hotels contained in the skyline

The skyline operator selects all interesting tuples in an input relation, i.e. tuples which are not dominated by any other tuple [5]. A tuple dominates another tuple if it is as good in all dimensions, and better in at least one. The skyline of a relation D is formally described in Definition 3, while the relationship between tuples are described in Definitions 1 and 2. The skyline operator is also known as the maximal vector problem [5, 18, 17], however, this name is rarely used in a database context.

Definition 1. A tuple p dominates another tuple q if p is as good or better than

p in all dimensions and better than q in at least one dimension. We write $p \prec q$ to mean that tuple p dominates tuple q, and $p \nprec q$ to mean that tuple p does not dominate tuple q

Definition 2. A tuple p is incomparable to another tuple q if $q \nprec q$ and $q \nprec p$. We use $p \prec\succ q$ to mean that p is incomparable to q

Definition 3. For a relation D, a tuple $p \in D$ that is not dominated by any other tuples $q \in D$ is considered as a skyline tuple. The skyline of D consists of all skyline tuples in D

In subsequent sections, we describe a number of algorithms for skyline computation. All of these algorithms exploit the fact that, when comparing two tuples p and q, there are three possible outcomes:

1. $p \prec q$
2. $p \succ q$
3. $p \prec\succ q$

In case 1, we know that q is not part of the skyline because it is dominated by p, therefore q can be discarded. Whereas, in case 2 we know that p is not part of the skyline and p can be discarded. In case 3, p and q are incomparable, therefore we need to keep p and q as potential skyline tuples until we can be sure they are a part of the skyline, or that they are dominated by some other tuple and can be discarded.

3.1 Sequential algorithms

This section describes common sequential algorithms for skyline computation. We focus mainly on algorithms related to parallel methods evaluated in this thesis. However, we also include important state-of-the-art sequential algorithms to provide an overview of the subject.

3.1.1 Block-nested-loops

The block-nested-loops algorithm repeatedly reads the set of input tuples, keeping a window of incomparable tuples in main memory. When a tuple p is read from the input, p is compared to all tuples of the window. Based on this comparison, p is either discarded, placed in the window or into a temporary file that will be considered in the next iteration [5]. The following cases can occur:

1. Some window tuple dominates p. In this case, p cannot be part of the skyline and is discarded
2. One or more window tuples $q_1, \ldots, q_n$ are dominated by p. In this case, $q_1, \ldots, q_n$ cannot be part of the skyline and are discarded. Furthermore, p is inserted into the window

3. All tuples in the window are incomparable with p. If there is room in the window, p inserted into the window. Otherwise, p, is written to a temporary file to be further processed in the next iteration

At the end of each iteration, tuples that have been compared to all tuples written to the temporary file are guaranteed to be part of the skyline and can be output. Other tuples of the window can be output if they are not eliminated during the next iteration. Specifically, when we read a tuple from the temporary file, all tuples inserted to the window before this tuple was written to the temporary file is part of the skyline. In order to keep track of the order, each tuple is marked by a timestamp at the time they are inserted to the window or into the temporary file. See Algorithm 1 for a detailed description.

Algorithm 1 BNL

```
Require: $D$ is the input relation,
Ensure: $R$ contains the skyline of $D$
  $W \leftarrow \{\}$                                  ▷ Limited window in memory
  $T \leftarrow \{\}$                                  ▷ Temporary file
  $A \leftarrow D$
  $countIn \leftarrow 0$
  $countOut \leftarrow 0$
  for all $p \in A$ do
      $p.timestamp \leftarrow countOut$
      $W \leftarrow W \cup \{p\}$
      for all $q \in W$ do                             ▷ Discard non-skyline tuples
          if $p \succ q$ then
              $W \leftarrow W - \{p\}$
              break
          else if $p \prec q$ then
              $W \leftarrow W - \{q\}$
      if $W$ is full then                              ▷ Temporarily save incomparable tuples
          $T = T \cup \{p\}$
          $countOut \leftarrow countOut + 1$
      if $p$ is last tuple of $D$ then                 ▷ Load next temporary file
          $A \leftarrow T$
          $T \leftarrow \{\}$
          $countIn \leftarrow 0$
          $countOut \leftarrow 0$
      for all $r \in W$ do                             ▷ Save results
          if $r.timestamp = countIn$ then
              $W \leftarrow W - \{r\}$
              $R \leftarrow R \cup \{r\}$
      $countIn \leftarrow countIn + 1$
  $R \leftarrow R \cup W$
```

Börnzsönyi et al. suggests two variants of the BNL algorithm in order to more quickly eliminate non-skyline tuples. The first one maintains the window as a self-organizing list, where dominating tuples are moved to the beginning of the list. The idea is that dominating tuples have a greater pruning power than other tuples and are more likely to be able to eliminate subsequently read tuples. When a new tuple is read, it is compared to the dominating tuples first, making it more likely for an early elimination. Based on the same principle, the second variant works by replacing tuples in the window by the most dominating set, thereby ensuring early eliminating of new tuples read. These strategies are particularly effective for skewed datasets, where a few tuples dominate large portions of the dataset.

Algorithms of the BNL class are likely the most prominent algorithms for computing skylines. There have been published several algorithms based on the same principle [10, 41, 55], and the basic operation of collecting maxima during a single scan of the input data can be found at the core of many state-of-the-art skyline algorithms [14, 36, 3].

3.1.2 Divide-and-conquer

The basic divide-and-conquer (D&Q) algorithm [5, 27, 13] subsequently divides the input into partitions until each partition contains only one (or a few) tuples. When this is done, the partitions are merged to compute the overall skyline. The merging step removes tuples dominated by other tuples. Input is divided by calculating the median m in some dimension and placing all tuples better than m in the left partition, and tuples worse than m in the right partition. There are different strategies for partitioning the input, however, most can be categorized as some form of grid partitioning as shown in Figure 3.2. A neat feature of this arrangement is that some partitions can be discarded without processing the nodes within, e.g. the upper right partition in Figure 3.2 can be discarded because it is guaranteed to be dominated by all tuples in the lower left partition. Algorithm 2 describes the basic D&Q algorithm.

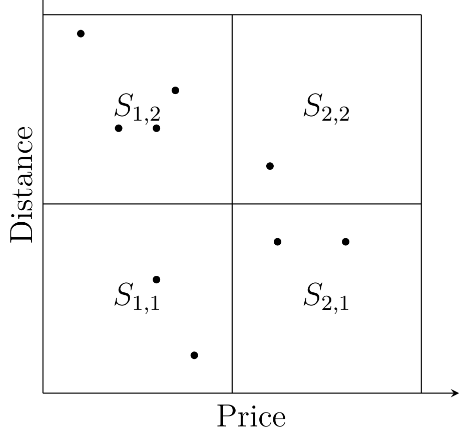


Figure 3.2: Grid partitioning

Note that the basic variation achieves poor performance unless the entire computation can be done in main memory. The reason being that the input is read, partitioned, written to disk, reread to be partitioned again, and so on several times until a partition fits into main memory. This is alleviated using m-way partitioning in such a way that every partition is expected to fit into main memory. Instead of the median, α-quantiles are computed in order to determine partition boundaries.

Algorithm 2 BasicDQ

```
Require: $D$ is a $d$-dimensional relation
Ensure: $R$ contains the skyline of $D$
  function SKYLINE($M, d$)
      if $|M| = 1$ then
          return $M$
      $p \leftarrow median(M, d)$
      $M_1, M_2 \leftarrow$PARTITION($M, d, p$)
      $S_1 \leftarrow$SKYLINE($M_1, d$)
      $S_2 \leftarrow$SKYLINE($M_2, d$)
      return $S_1 \cup$MERGE($S_1, S_2, d$)
  function PARTITION($M, d, p$)
      $S_1 \leftarrow \{p \in M | p_d < p\}$
      $S_2 \leftarrow \{q \in M | q_d \geqslant p\}$
      return $S_1, S_2$
  function MERGE($S_1, S_2, d$)
      if $S_1 = \{p\}$ then                                  ▻ Trivial case
          return $\{p \in S_2 | p \nprec q\}$
      else if $S_2 = \{q\}$ then
          $R \leftarrow S_2$
          for all $p \in S_1$ do
              if $p \prec q$ then
                  return $\{\}$
          return $S_2$
      else if $d = 2$ then                                   ▻ Low dimension
          $min \leftarrow minimum(S_1, d - 1)$
          return $\{q \in S_2 | q_1 < min\}$
      else                                                   ▻ General case
          $p \leftarrow median(S_1, d - 1)$
          $S_{1,1}, S_{1,2} \leftarrow$PARTITION($S_1, d - 1, p$)
          $S_{2,1}, S_{2,1} \leftarrow$PARTITION($S_2, d - 1, p$)
          $R_1 \leftarrow$MERGE($S_{1,1}, S_{2,1}, d$)
          $R_2 \leftarrow$MERGE($S_{1,2}, S_{2,2}, d$)
          $R_3 \leftarrow$MERGE($S_{1,1}, R_2, d - 1$)
          return $R_1 \cup R_3$
  $R \leftarrow$SKYLINE($D, d$)
```

3.1.3 SSkyline

SSkyline, also known as Best [47], is an algorithm for skyline computation where the entire input relation is assumed to be placed in main memory [37]. In contrast to BNL and other external algorithms, SSkyline does not need to consider temporary files and disk delays. Instead, it is optimized to exhibit good memory access patterns, and is considered a highly efficient cache conscious in-memory algorithm for small to moderate input relations.

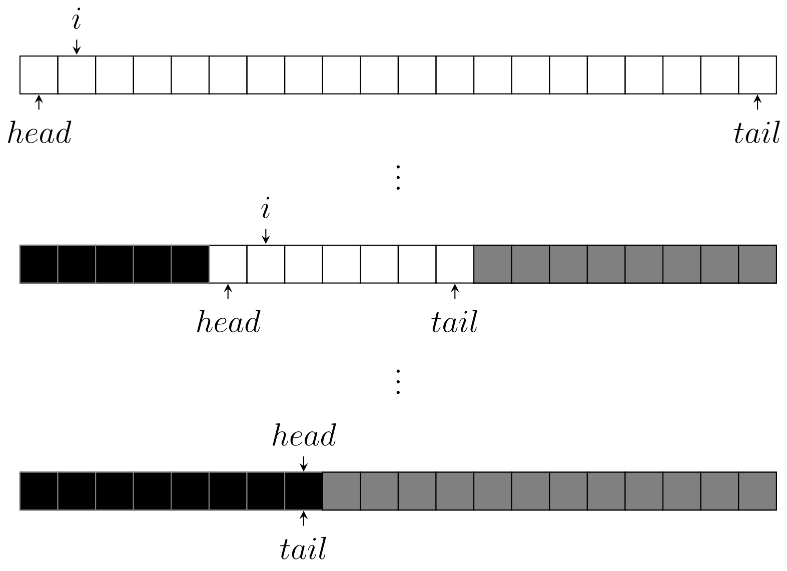


Figure 3.3: SSkyline example. Black boxes are part of the skyline, white boxes are undetermined, and gray boxes are dominated (i.e. gray boxes are not part of the skyline)

SSkyline takes an input relation D containing n tuples as input, and returns the skyline of D. The skyline is computed using two nested loops and three indices: *head*, *tail*, and i. Intuitively the inner loop searches for the next skyline tuple, while the outer loop repeats the inner loop until all skyline tuples have been found. Figure 3.3 shows an example run of SSkyline. In the first iteration *head* points to the first tuple, i to the second, and *tail* to the last. Confirmed skyline tuples are placed left of *head* and colored black, confirmed non-skyline tuples are placed to the right of *tail* and marked with gray, while tuples in-between are still under consideration and will at some point be pointed to by *head* if they are part of the skyline. Each iteration of the outer loop confirms one skyline tuple by moving *head* to the right, and the inner loop may discard many non-skyline tuples by moving *tail* to the left. When *head* = *tail*, the algorithm terminates and returns all skyline tuples. At this point, *head* and *tail* points to the last skyline tuple, while i has been discarded. Algorithm 3 describes SSkyline in detail.

Algorithm 3 SSkyline

```
Require: $D$ is the input relation, $n$ is number of tuples in $D$
Ensure: $R$ contains the skyline of $D$
  $head \leftarrow 1$
  $tail \leftarrow n$
  while $head < tail$ do
      $i \leftarrow head + 1$
      while $i \leqslant tail$ do
          if $D_{head} \prec D_i$ then
              $D_i \leftarrow D_{tail}$
              $tail \leftarrow tail - 1$
          else if $D_i \prec D_{head}$ then
              $D_{head} \leftarrow D_i$
              $D_i \leftarrow D_{tail}$
              $tail \leftarrow tail - 1$
              $i \leftarrow head + 1$
          else
              $i \leftarrow i + 1$
      $head \leftarrow head + 1$
  $R \leftarrow D_1, \ldots, D_{head}$
```

3.1.4 Index based

There are a number of skyline algorithms that exploit index structures in some way [45, 35, 5, 36]. For instance, branch-and-bound is the best performing disk-based algorithm. Index based methods are not central to this thesis, however, we include a superficial explanation of the nearest-neighbor (NN) and branch-and-bound (BBS) algorithms for the sake of completeness.

Nearest-neighbor algorithm and branch-and-bound skyline algorithm use indexing (R-trees) in order to eliminate dominance tests for a block of tuples at once. NN use the results of nearest neighbor search to partition the input data recursively [5]. NN was proven to outperform other algorithms in [5], however, it has some shortcomings that was addressed in [36]. Specifically, NN lacked a duplicate elimination, introduced multiple node visits and had large space requirements. BBS is a state-of-the-art skyline algorithm based on NN. BBS is IO-optimal, meaning it visits only the nodes that may contain skyline points, and it does not access the same node twice, and can be considered the successor of NN.

3.2 Parallel algorithms

In this section we present parallel skyline algorithms, including traditional algorithms targeted at parallel and distributed systems, and recent algorithms developed for multi-core architectures.

3.2.1 Parallel divide-and-conquer

The parallel divide-and-conquer algorithm consists of three phases: partition, local skyline computation, and merge. The partition phase divides tuples into distinct sets and can be performed using a number of different techniques. In the local skyline phase, each thread independently computes the local skyline for its designated partition D_i. Finally, local skylines are merged to produce the global skyline R. Algorithm 4 shows an overview of parallel D&Q.

Algorithm 4 PDQSkyline

```
Require: $D$ is the input relation, $N$ is number of partitions
Ensure: $R$ is the skyline of $D$
  $D_1, \ldots, D_N \leftarrow$ Partition($D$,$N$)
  $S_1, \ldots, S_N \leftarrow$ LocalSkyline($D_i, \ldots, D_N$)
  $R \leftarrow$ Merge($S_1, \ldots, S_N$)
```

In a shared-nothing architecture, there typically exists one central server, called coordinator, which is responsible for a set of N servers. In a basic parallel D&Q skyline algorithm, a coordinator delegates the local skyline computation to N servers by sequentially partitioning the data based on some criteria. After local skylines have been computed, each server sends the result to the coordinator, which performs a sequential merge.

Parallel D&Q is based on the fact that, for any decomposition of the set D into subsets $D_1, \ldots D_N$, the global skyline is equal to the skyline of the union of all local skylines [11]: $Skyline(D) = Skyline(Skyline(D_1) \cup \ldots \cup Skyline(D_N))$. This means that we can use any sequential skyline algorithm to calculate the local skylines. Additionally we can use the same sequential skyline algorithm to merge the results.

In [11] they propose a parallel D&Q algorithm use random partitioning to ensure that each server get a similar workload. For local skyline computation they use BBS. In the merge phase, they use all-to-all communication to provide a more efficient elimination of non-skyline tuples. We refer to the original article for a detailed explanation.

There are a wide range of partitioning techniques available for skyline computation. For instance, one can use grid or angular partitioning to exploit geometric properties of the dataset, random partitioning can be used to achieve fairness in uniform data distributions, or one can simply divide data based on its location in memory without any consideration for skyline-specific traits or heuristics. In subsequent sections we describe selected partitioning techniques that are related to work done in this thesis.

Linear partitioning

In a linear partitioning strategy, data is distributed without no geometric considerations. The input set is simply divided into the n equal sized parts $D_1 =$

$[d_1, \ldots, d_{Card(D)/n}], \ldots, D_n = [d_{Card(D)/n(n-1)}, \ldots, d_{Card(D)}]$. This strategy is illustrated in Figure 3.4.

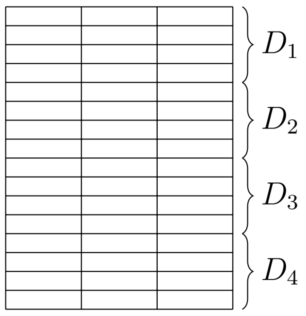
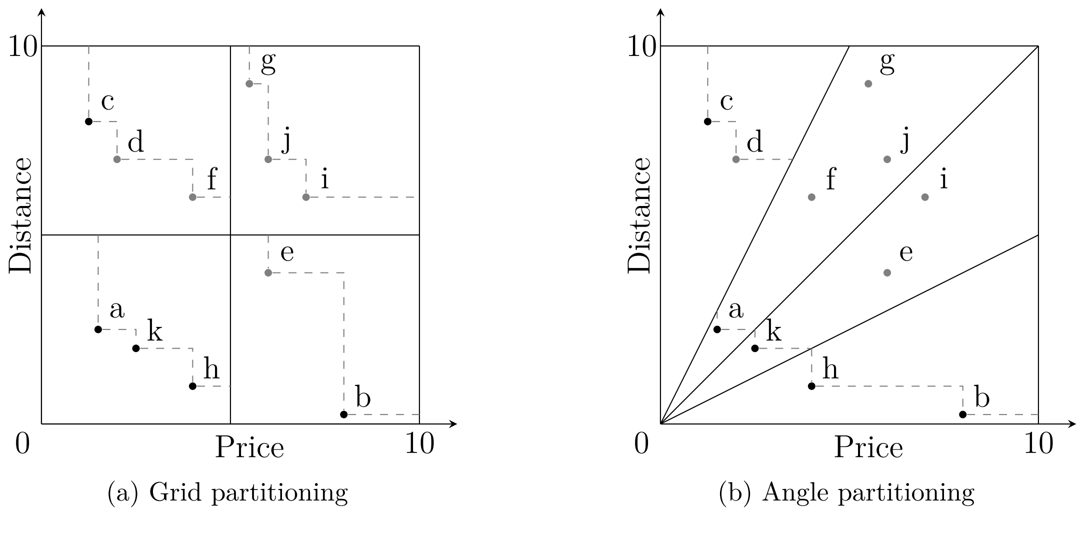


Figure 3.4: Linear partitioning strategy into 4 partitions for a 3-dimensional dataset

Linear partitioning is easy to implement, distributes tuples fairly, and can be performed very efficiently. In some cases, data can be processed without explicitly creating separate partitions, thus saving memory volume and bandwidth. However, it does not exploit operator-specific features such as geometric properties.

Angle-based space partitioning

Angle-based partitioning is an efficient geometric partitioning technique. It is a big step from linear to angle-based partitioning, and one may expect random- or grid partitioning to be intermediate steps. However, all space partitioning techniques incur similar memory bandwidth and locality costs. Specifically, each point needs to be mapped to its designated partition by inspecting its value, which translates into consecutive read operations and random write operations. We therefore choose to go directly for the most efficient (and complex) partitioning technique for skyline computation.



(a) Grid partitioning

(b) Angle partitioning

Figure 3.5: Partitioning example using 4 partitions

The motivation behind angle-based partitioning is best explained by an example from [48]. We use the classical hotel distance-price scenario where one would like to find hotels that have the lowest possible price and distance. Figure 3.5 shows all possible hotels as points, where black points are included in the global skyline, and local skylines are dashed. The global skyline consists of 5 points: c, a, k, h, and b. Using a grid-based partitioning, the local skyline computation phase will return 11 points as potential candidates, all which need to be processed in the merging phase. In contrast, the angle-based method return only 6 points to the merge phase reducing workload by 45% compared to the grid-based technique. Indeed, the angle-based technique generate a set of local skylines that closely resemble the end result, differing only by a single point.

Another feature that is worth mentioning is the pruning power of the average local data point. Looking at 3.5b), one can see that points a and k dominate all other points in their partition, significantly reducing the work required for local skyline computation. We refer to [48] for a detailed analysis on pruning power.

In order to compute partitioning bounds and distribute points, an angle-based partitioning technique maps the cartesian coordinate space into a hyperspherical space, and partitions the data space based on angular coordinates into N partitions.

$$\begin{aligned} r &= \sqrt{x_n^2 + x_{n-1}^2 + \ldots + x_1^2} \\ tan(\phi_1) &= \frac{\sqrt{x_n^2 + x_{n-1}^2 + \ldots + x_2^2}}{x_1} \\ &\vdots \\ tan(\phi_{d-2}) &= \frac{\sqrt{x_n^2 + x_{n-1}^2}}{x_{n-2}} \\ tan(\phi_{d-1}) &= \frac{x_n}{x_n - 1} \end{aligned} \tag{3.1}$$

Coordinates are mapped to the hyperspherical space using Equation 3.1, resulting in $d-1$ angular dimensions for a d-dimensional dataset. Subsequently, the angular coordinates are used to divide the space into N partitions using a grid partitioning technique. This leads to a partitioning where all points that have similar angular coordinates fall into the same partition. In effect, each local skyline computation will get an increasingly correlated dataset as the number of partitions increase. The end-result is that local skylines are of a small cardinality and that most skyline algorithms perform well on them.

Given the number of partitions N and a d-dimensional data space D, angle-based partitioning assigns to each partition a part of the data space D_i, defined by Equation 3.2. Note that we assume that points are non-negative in all dimensions.

$$\begin{aligned} D_i &= [\phi_1^{i-1}, \phi_1^i] \times \ldots \times [\phi_{d-1}^{i-1}, \phi_{d-1}^i] \\ \phi_j^0 &= 0 && 1 \leqslant j \leqslant d, \quad 1 \leqslant i \leqslant N \\ \phi_j^N &= \frac{\pi}{2} \end{aligned} \tag{3.2}$$

Vlachou et al. suggests two methods of defining partitioning bounds: equi-volume and dynamic. In the original article, this technique is used in a disk-based shared-nothing parallel DBMS context. However, we focus on multi-core shared-memory systems, and will use angle partitioning as an integral part of our experiments. Consequently, we explain each method as if the entire relation, and all partitions can be stored in main memory. Angle-based partitioning algorithms presented in following sections are based on textual explanations in [48], and source code received from the authors.

Equi-volume partitioning

Equi-volume partitioning aims to derive the grid boundaries in such a way that the points are equally distributed to partitions, assuming an independent data distribution. With an independent data distribution, a partitioning scheme that generates partitions of equal volume will ensure an approximately equal number of points in each partition. If V is the volume of the d-dimensional space and N the available number of cores, the volume of each partition should be $\frac{V}{N}$.

The equi-volume strategy is described by Algorithm 5. Partitioning bounds are represented by a collection of N $(d-1)$-dimensional structures that includes the lower and higher bounds for each angle. Similar to [48], bounds are calculated by dividing the data space independently by each angle creating approximately equal volume for each partition. In order to divide the data space into N equi-volume parts based on one angular dimension ϕ, a binary search on the interval $[0, \ldots, \frac{\pi}{2}]$ is performed. In each step the volume of the current partition is evaluated and compared to the target volume of $\frac{V}{d-1}$ When the volume is sufficiently close, bounds are accepted, and bounds for the next angle are calculated.

Algorithm 5 ComputeBounds

```
Require: $d$ is the number of dimensions in the input relation, $N$ is the total number
    of partitions, $N_0 \ldots N_{d-1}$ is the number of partitions per dimension,
Ensure: $R$ is the bounds for each angle
  for all $j \leftarrow 1$ to $d-1$ do                          ▷ each angular dimension
      $\phi_j^0 = 0$
      $\phi_j^N = \frac{\pi}{2}$
      for all $i \leftarrow 1$ to $N_j$ do                          ▷ each partition
          $V_i \leftarrow calcVolume(\phi_j^{i-1}, \phi_j^i, d)$
          while $|V_i - \frac{V}{N}| > threshold(V)$ do                ▷ binary search
              if $V_i < \frac{V}{p}$ then
                  $\phi_j^i \leftarrow \phi_j^i + \frac{\phi_j^i - \phi_j^{i-1}}{2}$
              else
                  $\phi_j^i \leftarrow \phi_j^i - \frac{\phi_j^i - \phi_j^{i-1}}{2}$
              $V_i \leftarrow calcVolume(\phi_j^{i-1}, \phi_j^i, d)$
  $R \leftarrow \forall_{i,j}(\phi_j^i | 0 \leqslant i \leqslant N, 0 \leqslant j < d)$      ▷ all calculated boundaries
```

We use Algorithm 6 in order to determine the number of partitions possible per dimension. For the simplest case, each dimension incur an equal number of splits, resulting in a symmetric partitioning where each dimension is split into $\leftarrow \lfloor \sqrt[d-1]{N} \rfloor$ parts. If this is impossible, due to non-cubic partition count we resort to an asymmetric strategy as described in the while loop of 6. Number of partitions in each dimension is increased progressively until the total partition is equal to N.

Algorithm 6 CountPartitions

```
Require: $N$ is the total number of partitions, $d$ is the number of dimensions
Ensure: $R$ contains the number of partitions for each angle
  $N_1, \ldots, N_{d-1} \leftarrow \lfloor \sqrt[d-1]{p} \rfloor$
  $achieved \leftarrow (N_1)^{d-1}$
  $change \leftarrow$ True
  while $achieved < N \wedge change$ do                          ▷ asymmetric partitioning
      $change \leftarrow$ False
      for all $i \leftarrow 1$ to $d-1$ do
          if $\frac{achieved}{N_i} * (N_i + 1) \leqslant N$ then
              $N_i \leftarrow N_i + 1$          ▷ increase partition count for angular dimension $i$
              $achieved \leftarrow achieved + 1$
              $change \leftarrow$ True
  $R \leftarrow N_i, \ldots, N_{d-1}$
```

When the boundary values have been computed, all points are distributed by Algorithm 7. This algorithm give each partition a unique identifier. A point is allocated to its designated partition by looping through all partitions and checking if the point is confined in the pre-computed bounds for all angular dimensions.

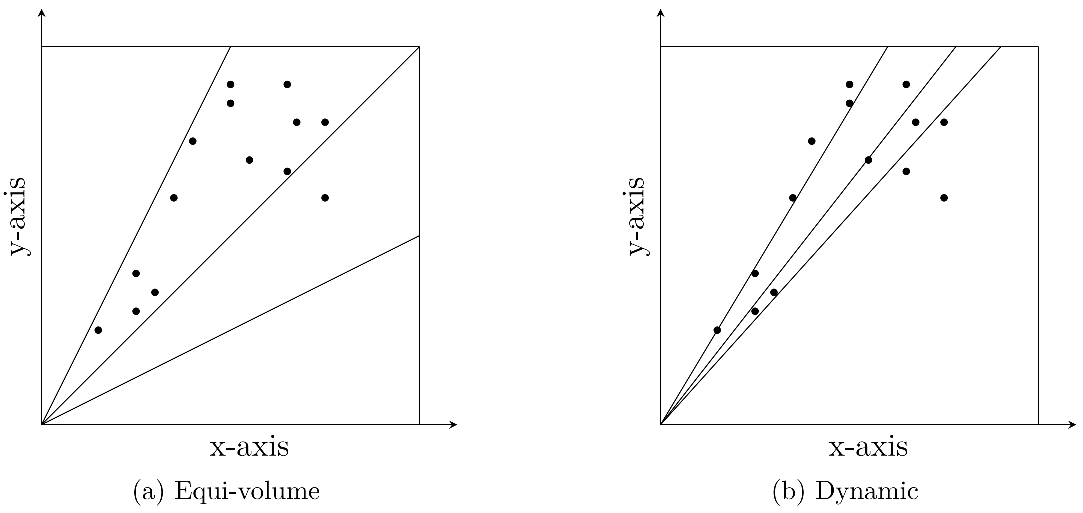
Algorithm 7 MapPointToPartition

```
Require: $p$ is a point of $d$ dimensions, $N$ is number of partitions, $\ldots, \phi_j^i, \ldots$ is the
   partitioning bounds
Ensure: $R$ is the partition id for $p$
  $p_1, \ldots, p_{d-1} \leftarrow$ angular coordinates for p
  for all $i \leftarrow 1 \ldots N$ do                                    ▷ each partition
      NextPartition:
      for all $j \leftarrow 1$ to $d-1$ do                        ▷ each angular dimension
          if $\neg(\phi_j^{i-1} < p_j \leqslant \phi_j^i)$ then
              goto NextPartition
      $R = j$
```

Dynamic partitioning

A weakness of static schemes like equi-volume partitioning is that non-uniform data distributions may be unevenly distributed. In the worst-case, all work may be distributed to one single thread, prohibiting the algorithm for utilizing any parallel resources. Dynamic partitioning alleviate this problem by splitting the space according to the data distribution.

Figure 3.6 shows an example of a non-uniform distribution where equi-volume partitioning is particularly inefficient. In this case one partition holds 92% of the input data, while the other three are more or less empty, clearly not a fair distribution. In contrast, the dynamic technique are able to distribute the same dataset quite well, resulting in the following distribution: 29%, 21%, 29%, 21% (clockwise).



(a) Equi-volume (b) Dynamic

Figure 3.6: Equi-volume versus dynamic partitioning in a non-uniform dataset

Dynamic partitioning works by progressively splitting the data space into smaller partitions as shown in Algorithm 8. Starting with one partition, Algorithm 8 distributes points until the maximum number of tuples per partition is reached for the current partition. At this point, the partition is split in angular dimension 1, and points are redistributed into two partitions based on the new boundaries. To ensure an even distribution among the new partitions, the split is made at the mean (or median) value in angular dimension 1 of all points in the partition being split. Subsequent splitting is done similarly. The angular split dimension is chosen in a round robin fashion.

Note that we use Algorithm 7 when mapping coordinates. However, partitioning boundaries has to be specified in tuples $[low, high]$ as they are no longer ordered in such a way that we can use a simple list. To make this clear, we use a slightly different boundary specification where θ indicates bound low, and ϕ indicates boundary high. Additionally, we use Φ and Θ to indicate all low-, and high boundary values respectively.

Algorithm 8 DynamicPart

```
Require: $D$ is the input relation of $d$ dimensions, $N$ is the number of partitions,
  $n_{max}$ is maximum partition size
Ensure: $D_1, \dots D_p$ is $N$ disjoint partitions of $D$
  $N_{achieved} \leftarrow 1$
  $\theta^1_{1,\dots,d} \leftarrow 0$
  $\phi^1_{1,\dots,d} \leftarrow \frac{\pi}{2}$
  $j \leftarrow 1$
  for all $p \in D$ do                                              ▷ distribute points
      $i \leftarrow MapPointToPartition(p, N_{achieved}, \Theta, \Phi)$
      $D_i \leftarrow D_i \cup \{p\}$
      if $Card(D_i) > n_{max}$ then                                 ▷ split partition
          $N_{achieved} \leftarrow N_{achieved} + 1$
          $\theta^{N_{achieved}}_{1,\dots,d} \leftarrow \theta^i_{1,\dots,d}$
          $\phi^{N_{achieved}}_{1,\dots,d} \leftarrow \phi^i_{1,\dots,d}$
          $\theta^{N_{achieved}}_j \leftarrow \overline{p^1_j, \dots, p^{Card(D_i)}_j}$       ▷ mean value in angular dimension $d$
          $\phi^i_j \leftarrow \theta^{N_{achieved}}_j$
          $j \leftarrow j + 1$
          for all $q \in D_i$ do                                    ▷ redistribute points
              $D_i \leftarrow D_i - \{q\}$
              $k \leftarrow MapPointToPartition(p, N_{achieved}, \Theta, \Phi)$
              $D_k \leftarrow D_k \cup \{p\}$
```

3.2.2 PSkyline

The PSkyline algorithm, is a D&Q based algorithm optimized for multi-core processors. In contrast to most existing divide-and-conquer algorithms for skyline computation that divide into partitions based on geometric properties, PSkyline simply divides D linearly into smaller blocks using the partitioning scheme described in Section 3.2.1. Im et al. evaluate PSkyline and parallel versions of BBS, SFS, and SSkyline on a sixteen core machine in [37]. In their experiments, PSkyline consistently had the best utilization of multiple cores, however, algorithms were evaluated based on parallel speedup only, and were not compared in terms of performance.

Algorithm 4 describes the general structure of PSkyline. First, the input relation is D is split into N partitions sequentially. Second, the local skyline is computed for each partition in parallel using Algorithm 3. Finally, local skylines are merged in parallel using Algorithm 9 repeatedly until all local skylines have been merged.

Algorithm 9 PMerge

```
Require: $S_1$ is a local skyline, $S_2$ is another local skyline
Ensure: $R$ is skyline of $S_1 \cup S_2$
 1: $T_1 \leftarrow S_1$
 2: $T_2 \leftarrow \{\}$
 3: function F(y)
 4:     for all $x \in T_1$ do
 5:         if $y \prec x$ then
 6:             $T_1 \leftarrow T_1 - \{x\}$
 7:         else if $x \prec y$ then
 8:             return
 9:     $T_2 \leftarrow T_2 \cup \{y\}$
10: parallel for $y \in S_2$ do
11:     F($y$)
12: $R \leftarrow T_1 \cup T_2$
```

Figure 3.7 shows two possible merging strategies: balanced and unbalanced. The balanced strategy are able to utilize more coarse-grained parallelism than the unbalanced strategy [28]. However, as pointed out in [37], this strategy will be increasingly sequential as number of local skylines decrease (and the size of each local skyline increase). In other words, relying solely on a balanced merge strategy is not sufficient to efficiently utilize parallel compute power when its needed the most.

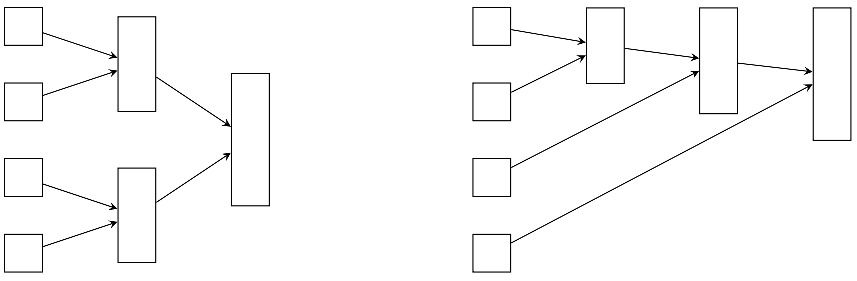

(a) Balanced (b) Unbalanced

Figure 3.7: Parallel merge strategies

To avoid the problems associated with coarse-grained parallelism, Algorithm 9 use a more fine-grained parallelism by allowing threads to fetch tuples from S_2 in a round robin fashion until skylines S_1 and S_2 have been merged. In a distributed or parallel system this would have been very inefficient due to expensive communication through a slow interconnect. However, in a shared-memory system, where communication costs are relatively small [31], the algorithm is able to utilize a great degree of parallelism with minimal overhead by working on the same data structure. PSkyline use the unbalanced merge strategy illustrated in Figure 3.7b), which is easier to implement in most shared-memory frameworks [12, 39, 7, 53] than the balanced strategy .

3.2.3 ParallelBNL

In [41], Selke et al. suggests a parallel version of BNL using a shared linked list for the skyline window. This is a straightforward approach, where a sequential algorithm is parallelized without big modifications. However, there are some issues related to concurrent modification of the shared list.

Three variants of list synchronization are suggested in the article: continuous, lazy, and lock-free. With continuous locking, each thread will acquire a lock on the next node before processing, lazy locking [20] only locks nodes that should be modified (or deleted), while lock-free are an optimistic approach that does not use any locking. Lazy and lock-free variants need to verify that iterations have been correctly performed and restart iterations that fail. ParallelBNL algorithms with each of the mentioned synchronization techniques are tested and the lazy locking scheme is shown to be most efficient. When we write ParallelBNL later in this thesis, we refer to parallel BNL using the lazy locking scheme, defined in Algorithm 10.

Each thread in ParallelBNL continuously process points fetched from the input relation. Fetching is done in a round robin fashion so that each thread receives similar work loads. Synchronization is only needed when the shared list needs to be modified. ParalellBNL uses a binary flag *deleted* to guarantee that no adjacent nodes are modified concurrently, which might otherwise result in violating pointer integrity.

Algorithm 10 ParallelBNL

```
Require: $D$ is an input relation of $d$ dimensions $N$ is available number of cores
Ensure: $R$ is skyline of $D$
  function BNLTHREAD(head,tail)
      NEXTRECORD:
      while $a \leftarrow D.next()$ do
          TRAVERSELIST:
          $pred \leftarrow head$
          $curr \leftarrow pred.next$
          while $curr \neq tail$ do
              if $curr.item \prec a$ then                          ▷ discard a
                  goto NEXTRECORD
              else if $a \prec curr.item$ then               ▷ try to discard curr
                  lock $pred$
                  lock $curr$
                  if VALIDATE$(pred, curr)$ then
                      $curr.deleted \leftarrow$ True
                      $pred.next \leftarrow curr.next$
                      ulock $curr$
                      ulock $pred$
                  else                          ▷ list has been modified, restart
                      ulock $curr$
                      ulock $pred$
                      goto TRAVERSELIST
                  $curr \leftarrow curr.next$
              else
                  $pred \leftarrow curr$
                  $curr \leftarrow curr.next$
          lock $pred$                                   ▷ try to append a
          if VALIDATE$(pred, curr)$ then
              $new \leftarrow Node(a)$
              $pred.next \leftarrow new$
              $new.next \leftarrow tail$
              ulock $pred$
          else                                  ▷ list has been modified, restart
              ulock $pred$
              goto TRAVERSELIST
  function VALIDATE(pred, curr)
      return $\neg pred.deleted \wedge \neg curr.deleted \wedge pred.next = curr$
  $head \leftarrow Node()$
  $tail \leftarrow Node()$
  parallel for $i \in 1, \ldots, N$ do
      BNLTHREAD$(head, tail)$
  $R \leftarrow head$
```

Chapter 4

APSkyline

In this chapter we develop an efficient skyline algorithm optimized for multi-core architectures. The algorithm is specifically designed in order to answer RQ3, whether an elaborate pre-processing phase can be efficient also in a multi-core context.

4.1 Algorithm overview

APSkyline is a D&Q based algorithm that use the structure defined in Algorithm 4. It consists of three phases: partition, local skyline computation, and merge. The partition phase divides tuples into distinct sets using an angle based partitioning scheme [48, 24]. The local skyline computation phase computes the local skyline for each partition using Algorithm 3. And, the merge phase combines local skylines to produce final result using the parallel merging strategy defined in Algorithm 9.

The basic idea is to add a state-of-the art partitioning scheme to the currently best performing multi-core skyline algorithm, PSkyline, in order to improve its capability to efficiently handle anti-correlated datasets.

4.2 Partition phase

The partition phase is responsible for dividing data evenly among threads and is an essential contributor overall performance. In general, partitioning should ensure that we achieve a decent degree of parallelism, good scalability, and fair work scheduling in subsequent phases. Additionally, we consider skyline-specific criteria like size of intermediate and equi-sized local skylines [48].

In contrast to PSkyline, we would like to exploit geometric properties in the dataset even though it may require additional pre-processing costs. We therefore choose the angle-based partitioning scheme published in [48], which has been shown to be the most efficient partitioning scheme currently known for skyline computation.

Vlachou et al. propose two schemes: equi-volume and dynamic in [48]. We will use equi-volume as defined in 3.2.1 with minor modifications to allow for parallelism in the partitioning phase. Because the dynamic scheme requires a substantial amount of memory move operations, it will be quite expensive in a multi-core context. We therefore suggest to use a sample-dynamic partitioning technique that will significantly reduce the costs associated with a dynamic scheme.

Because of the high computational costs associated with angle partitioning, it is essential that we utilize parallel compute power not only in phases that compute the actual skyline, but also in the partitioning phase. We propose straight-forward techniques for parallelizing equi-volume and sample-dynamic partitioning schemes in Section 4.2.3. Additionally, we propose a hybrid partitioning technique that utilize geometric properties in combination with random partitioning in order to prioritize a fair workload.

4.2.1 Sample-dynamic partitioning

In a dynamic partitioning scheme, each partition split induce an expensive redistribution cost. Specifically, each time a partition is split $\frac{n_{max}}{2}$ or more tuples needs to be moved from one memory location to another. This is reasonable in a parallel or distributed environment where IO-operations (including communication between nodes) are the dominating factor. However, in a shared-memory system such a method consumes a significant amount of the overall runtime. For instance, if we were to split the dataset into four partitions each containing at most $n_{max} = V/N$ tuples, a dynamic partitioning scheme need to move $\frac{3}{2} * n_{max} = \frac{3}{2} * \frac{V}{4} = 0.375V = 37.5\%$ of the dataset to achieve a perfectly fair data distribution. In the linear and equi-volume schemes this cost will not occur. Therefore, we propose a sample-based scheme where we use a smaller portion of the data in order to determine the partitioning boundaries before actual partitioning is performed.

We use Equation 4.1 to calculate the cost of splitting. Obviously this is a simplification and will not be completely accurate in all cases. Nevertheless, it provides some value when comparing the efficiency of dynamic and sample-dynamic partitioning.

$$C_{split} = \frac{N-1}{2} * n_{max} \tag{4.1}$$

In the sample-dynamic partitioning scheme, a configurable percentage s of the dataset is used to pre-compute the partitioning boundaries. In the simplest case, one can simply use a small n_{max} so that partitioning boundaries are defined at an early point. To increase the likelihood of picking significant sample points, we propose to choose samples in a uniform matter, as illustrated by Figure 4.1. In general, one can choose a number of different strategies, like random picking or choosing points based on domain-specific knowledge.

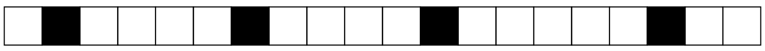


Figure 4.1: Sample points

To compare dynamic and sample-dynamic partitioning, we assume that dynamic partitioning use $n_{max} = \frac{V}{2}$ and sample-dynamic use $n_{max} = \frac{V_s}{2}$, where V_s is the sample size. By replacing n_{max} Equation 4.1 with respective values, we get the following costs estimations as number of threads increase to infinity:

$$C_{dynamic} = \lim_{N \to \infty} \frac{N-1}{2} * \frac{V}{N} = 0.5V$$
$$C_{sample} = \lim_{N \to \infty} \frac{N-1}{2} * \frac{V_s}{N} = 0.5V_s$$

We observe that the additional cost of dynamic partitioning will never be more than 50% of the dataset, while the additional cost of sample-dynamic partitioning will never be more than 50% of the sample size. This indicates that a sample-dynamic scheme can reduce partitioning costs significantly if configured correctly. In order to achieve a fair distribution, one must find a compromise between accuracy and speed. We propose a sample size of 1-10%, which requires a modest splitting cost of 0.5% – 5%, greatly reducing the cost of dynamic partitioning.

4.2.2 Geometric-random partitioning

For distributions where neither equi-volume nor sample-dynamic partitioning are able to achieve a fair workload APSkyline will not be able to efficiently utilize parallel compute power. For example, datasets that include many equal rows will in most cases cause a skewed workload, and cannot be fairly distributed by geometric partitioning alone. To handle such cases we suggest a hybrid approach of angle- and random partitioning.

The idea is to modify geometric partitioning schemes like equi-volume and sample-dynamic by limiting the size of each partition. When we reach the predefined limit for a partition, overflow points are placed in some other partition using one of the following strategies:

1. Place overflow points in a random partition
2. Place overflow points in the partition with most available space
3. Place overflow points in the first seen non-full partition
4. Use a progressively smaller subset of the angular dimensions to place overflow points in partitions similar to their original placement

This strategy can be performed during the point distribution or as a subsequent step where points are redistributed as needed. The former avoids moving points in memory, thus saving memory bandwidth. However, it may cause random partitioning to interfere with the geometric scheme. Specifically, points that are randomly distributed may fill partitions that otherwise could have held geometrically distributed points, resulting in a more expensive merge phase. For the latter, geometric distribution first priority. Because all points are distributed geometrically in a first step, overflow points will not interfere with the original partitioning scheme. Obviously, this comes at a price, namely memory bandwidth in the re-distribution step.

Depending on costs associated with sub-optimal partitioning, it is likely that both methods are viable for selected data distributions. Nevertheless, we believe that that avoiding a re-distribution step can be advantageous in a multi-core context in order to save memory bandwidth in the partition phase.

When placing overflow points, alternative 1 is low-cost and ensures that all partitions achieve an approximately equal workload. Additionally, we distribute the possible interference to all partitions as opposed to alternative 3, where a few partitions risk being filled by random points at an early stage. Alternative 4 has the advantage of keeping some geometric properties also for overflow points, however, for skewed datasets we risk placing overflow points in the original partition, voiding our effort to induce a fair workload. If partitioning is done sequentially, we would most likely choose alternative 1 or 4 (with some modifications to ensure that points are indeed fairly distributed), however, when utilizing parallelism in in the partitioning phase, alternative 1 will ensure that threads to an increasing degree can write to different partitions, reducing synchronization costs.

Algorithm 11 GeometricRandomPointDistributor

```
Require: $D_i$ is partition $i$ of input relation $D$, $N$ is requested number of partitions,
    $p$ is the point being distributed
Ensure: $p$ is distributed to a non-full partition for the majority of cases
    $i \leftarrow MapPointToPartition(\ldots)$
    if $D_i$ is full then                                    ▷ Add to random partition
        $j \leftarrow rand(1, N)$
        $D_j \leftarrow D_j \cup \{p\}$
    else
        $D_i \leftarrow D_i \cup \{p\}$
```

Algorithm 11 use alternative 1 during the point distribution phase to prioritize a fair workload over a strictly geometric partitioning scheme.

4.2.3 Parallelism in the partition phase

In contrast to algorithms directed at parallel and distributed systems, a shared-memory algorithm needs to have a highly optimized partitioning phase. Blanas et al. shows that, for hash join, the partitioning costs were sufficiently high compared to overall runtime that a simple algorithm with no partitioning outperformed a more sophisticated algorithm with a partitioning phase [4]. Therefore we suggest to utilize the capability of low-cost communication to induce parallelism in this phase.

Algorithm 12 shows the parallel partitioning algorithm that are repeatedly executed by each thread. We use N threads in order to partition a relation into N partitions. Obviously, we need some way of determining which points each thread should distribute. This can be done in a round robin fashion, or using a linear partitioning scheme to define read boundaries without physically partitioning points. In our implementation we use the linear scheme. To split data into two partitions, thread 1 process tuples $[1, \ldots, n/2]$ and thread 2 process tuples $[n/2, \ldots n]$. Both threads place tuples into multiple shared collections (partitions). In the ideal case, thread 1 only writes to partition 1, while thread 2 writes to partition 2, avoiding the need for any synchronization. In practice, some collisions will occur and threads will sometimes have to wait for locks. However, the number of partitions increase with the number of threads, making synchronization gradually more fine-grained, reducing the likeliness of collisions. This property should allow the algorithm to scale well with an increasing degree of parallelism, thereby making it a good match for future architectures.

Algorithm 12 PointDistributor

```
Require: $D_i$ is partition $i$ of input relation $D$, $p$ is the point being distributed
Ensure: $p$ is distributed to the correct partition according to partitioning scheme
  $i \leftarrow MapPointToPartition(\ldots)$
  lock $D_i$
  $D_i \leftarrow D_i \cup \{p\}$
  ulock $D_i$
```

In the linear and equi-volume partitioning schemes this strategy can be used directly. The boundaries are pre-determined, threads can therefore easily distribute data with a low amount of synchronization. For the sample-dynamic partitioning scheme, partition boundaries will change before the requested partitioning count has been reached. In this case we suggest a two-step process, where partitioning boundaries are calculated sequentially using 1% - 10% of the input relation before parallel partitioning is used as described by Algorithm 12. We could use a similar strategy for the dynamic scheme, but that would only allow parallel computation for a small portion if the relation. Instead, we propose the modified version described by Algorithm 13.

Algorithm 13 DynamicPointDistributor

```
Require: $D_i$ is partition $i$ of input relation $D$, $p$ is the point being distributed
Ensure: $p$ is distributed to the correct partition according to partitioning scheme
  $V_{1,\ldots,N}^{before} \leftarrow V_{1,\ldots,N}$
  $i \leftarrow MapPointToPartition(\ldots)$
  lock $D_i$
  if $V_i = V_i^{before}$ then                    ▷ partition is unchanged, safe to add point
    $D_i \leftarrow D_i \cup \{p\}$
    if $D_i$ is full then
      $next \leftarrow getNextPartitionId()$          ▷ atomic fetch-and-increment
      split $D_i$ into $D_i, D_{next}$
      $V_i \leftarrow V_i + 1$
      $V_{next} \leftarrow 1$                          ▷ initially equal to 0
  ulock $D_i$
```

In Algorithm 13, we use $V_{1,\ldots,N}$ to store the current version of each partition. When a partition is changed (split), we increment the version number by one using atomic operations. It is worth noting that splitting a partition will not affect any of the remaining partitions, therefore threads working on other partitions can continue as usual. If a point is mapped to partition D_i by thread 1, and D_i split by thread 2 before thread 1 gets a lock, thread 1 will detect that the version has changed and restart. This ensures that the algorithm does not add points to incorrect partitions even if boundaries are concurrently modified. Partitions with version number 0 is not considered in the mapping algorithm, thereby avoiding the usage of incomplete boundary values.

4.3 Local skyline computation phase

In [48], BBS and SFS are used to compute the local skylines, however, in a multi-core context it may be beneficial to use simpler algorithms optimized for in memory execution. In our experiments, we use the SSkyline algorithm introduced in 3.1. In the APSkyline algorithm, each thread executes an instance of Algorithm 3 using its private partition as input to compute the local skyline independently.

4.4 Merge phase

When the local skylines have been computed, Algorithm 9 is executed in a for loop until all partitions $S_1, \ldots, S_N$ has been merged into the global skyline. Note that Algorithm 9 exploit parallel compute power to a great degree by utilizing efficient inter-thread communication mechanisms in modern processor architectures [37].

4.5 Adaptive algorithm

The cost of skyline computation is highly dependent on input data characteristics. Specifically, input data that produce a large skyline can be very compute-intensive, while input data that produce a small skyline can be processed quite fast using simple algorithms. However, an angular partitioning technique requires the costly operation of transforming every tuple into the hyperspherical space, regardless of the input data. For correlated datasets, APSkyline is likely to spend more time partitioning than it would take a simpler algorithm to produce a result, whereas, APSkyline may excel for anti-correlated input data. In order to achieve efficient skyline computation for a wider range of data input, we present an adaptive algorithm that use skyline cardinality estimation to select the most efficient method for each case.

Algorithm 14 describes an adaptive algorithm. It takes the dataset D, and a set of algorithms A with their optimal range of computation C. By computing an estimate of the skyline size and comparing to ranges in C, the best algorithm is selected. For datasets where an optimal algorithm is unknown, we use the default algorithm $A_{default}$.

Algorithm 14 AdaptiveSkyline

```
Require: $D$ is the input relation, $A$ is a set of skyline algorithms where $A_{default}$ is
   the default choice, $C$ contains the optimal skyline size ranges for algorithms $A$
Ensure: $R$ is the skyline of $D$
  $e \leftarrow$ estimate of $card(D)$
  for all $c \in C$ do
      if $c.min < e \leqslant c.max$ then
          $R \leftarrow A_c(D)$
          return
  $R \leftarrow A_{default}(D)$                              ▷ No optimal algorithm found
```

Cost estimation for the skyline operator is addressed in [56, 8, 16]. In order to estimate skyline size, Chaudhuri et al. suggest a sample based method based on sampling, log sampling (LS). They modify statistical methods for independent data distributions in order to be able to estimate size of other data distributions. Specifically, they observe that skyline size of independent data distributions increase logarithmically with the input cardinality. For anti-correlated distributions, the skyline size will be larger than this value, whereas for a correlated dataset it will be smaller. This observation is used to generate relatively accurate measures of skyline size with low computational costs. However, as pointed out in [56], computing an estimate for non-independent distributions using statistical methods designed for independent distributions can in some cases lead so large estimation errors. Zhang et al. propose a kernel based (KB) method in order to estimate skyline cardinality based on a small sample from the data set. KB use more sound statistical methods, thereby achieving high accuracy for a variety of real and synthetic datasets, even where LS fails.

In an adaptive algorithm running in a shared-memory environment, it is essential that the cardinality estimate that can be computed efficiently. If we spend a significant time estimating, performance gains will be lost for lest cost-intensive datasets. The kernel based method proposed in [56] is the most accurate method, however it requires a substantial amount of numeric calculations. Therefore, it may be reasonable to use a more basic method, like the simpler log-sampling method proposed in [8] in this case. An adaptive method can simply default to an algorithm efficient in the computation of low to moderate skyline cardinalities and use APSkyline for the most compute-intensive cases. As long as we are able to detect the most compute-intensive cases efficiently, an adaptive algorithm should perform well for most datasets.

We need some way to determine the range of each algorithm. This can be done manually based on a set of empirical experiments. However, this would require a substantial amount of manual labour in order to map each algorithm to its optimal skyline range. Instead, we suggest to use data sets representing a wide variety of distributions to calibrate the algorithm automatically when it is installed. Another advantage of this approach is that the algorithm can adapt to different DBMSs by automatically choosing the method best suited for a certain dataset in its current execution environment [15].

Algorithm 15 Calibrate

```
Require: $D$ is the input relation, $A$ is a set of skyline algorithms, $D$ is datasets
  used to estimate the optimal range for algorithms, $C$ maps skyline ranges to each
  data set
Ensure: $R$ specifies the most efficient algorithm for each skyline cardinality range
  for all $C_i \in C$ do
      $best \leftarrow \infty$
      $algorithm \leftarrow$ Null
      for all $A_j \in A$ do
          $startTimer()$
          for all $D_k \in D | j \in C_i.datasets$ do
              $A_j(D_k)$
          $t \leftarrow endTimer()$
          if $t < best$ then
              $algorithm \leftarrow A_j$
              $best \leftarrow t$
      $C_i.algorithm \leftarrow algorithm$
```

Algorithm 15 estimates the best algorithm for each case based on runtime measurements executed on datasets representative for each skyline cardinality range. We are now able to map each range to the best-performing algorithm simply by defining a standard set of datasets that can be used to evaluate current, and future algorithms. Our method can be improved by measuring additional parameters, like memory used. We use runtime in because it is an objective measurement on performance that give us good indications as to which algorithm is most efficient for each case. To ensure that mappings are sufficiently accurate it may be appropriate to run algorithms multiple times and use average runtime measurements.

Chapter 5

Experiments and results

In this chapter, we study the performance of the alternative skyline algorithms presented in Chapter 3 and 4. For our experiments, we use synthetic and real-life datasets. Furthermore, we vary the number of threads available, and input cardinality and dimensionality.

5.1 Experiment setup

All our experiments are carried out on a single node of a cluster running Debian Linux 7.0. The node used is equipped with two Intel Xeon X5650 2.67GHz six-core processors, thus providing a total of 12 physical cores at each node. Our algorithms are implemented using the Java programming language version 1.6 running on the OpenJDK (IcedTea6) runtime environment. We include an overview of the test environment in Table 5.1.

Processors	2x Intel Xeon X5650 @ 2.67 GHz
Cores per processor	6
Contexts per core	2
Cache size, sharing	12MB L3, shared
Memory	128GB
Operating system	Debian 7.0 (wheezy/sid)
Kernel	3.2.0-39-generic
Java runtime	Java version 1.6.0_27 running on OpenJDK (IcedTea6 1.12.5)

Table 5.1: Platform characteristics

The Intel Xeon X5650 processor has 6 physical cores which can run up to 12 hardware threads using hyper-threading technology. Each core is equipped with private L1 and L2 caches, and all cores on one die share the bigger L3 cache, which is last level cache (LLC) for this architecture.

If not stated otherwise, each experiment is executed using values displayed in bold in Table 5.2. A tuple has d attributes of type single-precision float. The values of the d float values of a tuple are generated randomly in the range of $[0, 1)$ for synthetic data distributions. Input cardinality and dimensionality for synthetic distributions are included in Table 5.2, whereas real-life datasets are described in Section 5.1.2.

Parameter	**Values**
Data distribution	Independent, **Anti-correlated**, Correlated
Dataset	**Synthetic**, NBA, Household, Zillow5D, Zillow6D
Input cardinality	50k, 100k, 500k, 1M, **5M**, 10M, 15M
Input dimensionality	2, 3, 4, **5**
Thread count	1, 2, 4, 6, 12, **24**, 32, 64, 128, 256, 512, 1024

Table 5.2: Test parameters

Each experiment is executed ten times and we use the median values when reporting results. Before taking any measurements, we perform a dry-run of the algorithm being tested. After each execution we verify the algorithm output by comparing all tuples to the output computed by a sequential execution of BNL. Additionally, we measure variance, minimum, and maximum values in order to ensure that results are sufficiently accurate. For synthetic datasets, new input is generated for each of the ten executions. In cases where we achieve unexpected results we repeat affected experiments to rule out external factors.

$$S = T_1/T_p \tag{5.1}$$

In order to calculate parallel speedup [44], and to compare algorithms in general, we use Equation 5.1. When comparing sequential to parallel performance, S is the speedup, T_1 is runtime for a sequential execution, and T_p is runtime for a parallel execution using p threads. When comparing two algorithms, regardless of parallel speedup, T_1 refers to the runtime of one algorithm, while T_p refers to the runtime of another algorithm. Specifically, when we state that one algorithm A outperforms algorithm B by a factor of X, T_1 is replaced by the runtime of algorithm B, whereas T_p is replaced by the runtime of algorithm A, and the factor X is equal to S in Equation 5.1.

5.1.1 Algorithms

We perform experiments on current state-of-the-art multi-core algorithms, in addition to three variations of our newly developed APSkyline algorithm. Specifically, we implement the following variants:

ParallelBNL Parallel version of BNL described in Section 3.2.3

PSkyline Parallel D&Q-based algorithm described in Section 3.2.2

APSEquiVolume Parallel D&Q-based algorithm using the angle-based partitioning scheme as explained in Section 3.2.1

APSSampleDynamic Parallel D&Q-based algorithm using the sample-dynamic partitioning scheme described in Section 4.2.1

APSSampleDynamic+ Parallel D&Q-based algorithm using the sample-dynamic partitioning scheme described in Section 4.2.1 in combination the geometric-random modification explained in Section 4.2.2. We set the limit for each partition to be $\frac{card(D)}{N} * 0.1$ and place overflow points into random partitions. Overflow handling is done during the initial distribution, avoiding extra memory bandwidth costs

Due to time restrictions, we do not implement the adaptive algorithm, however experiments will still give indications of its applicability by testing how different algorithms respond to different data distributions.

Note that when we often refer to APSEquiVolume, APSSampleDynamic, and APSSampleDynamic+ collectively as APSklyine. When all variations of APSkyline achieve similar performance characteristics we refer to APSkyline as one algorithm. Whereas, we refer to each variation individually when characteristics differ.

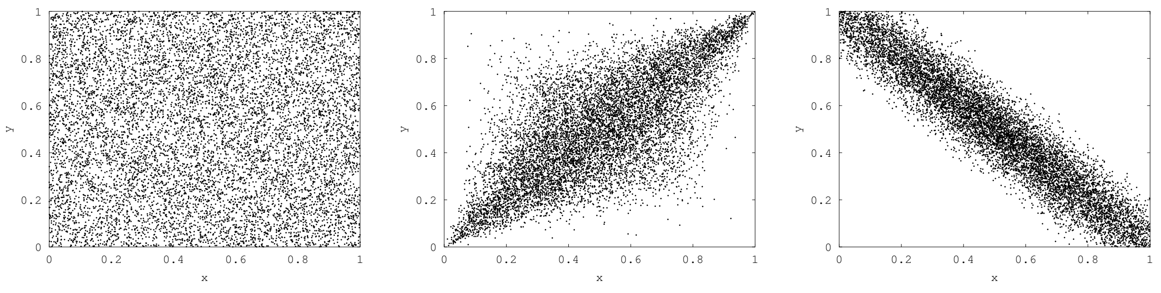
ParallelBNL and PSkyline implementations are based on source code published by Selke et al. in [41]. However, we removed some some testing code and abstractions in order to improve performance. For all cases our implementations performs equally good, or better than original implementations. We also examined the code used by Park et al. in [37] to ensure a fair comparison.

Relations are represented as encapsulated low-level floating-point arrays of one dimension. We use a class named PointSource to encapsulate input relations, and partitions. Each time a point is read from a PointSource instance, it is copied into a separate float array so that it can be worked on independently.

5.1.2 Input data

The skyline operator is very sensitive to correlations among attributes [8]. When the attributes have perfect positive correlation, the skyline is a single tuple. Whereas, for a perfectly anti-correlated dataset, the skyline is the whole table. In general, it can be anywhere in between. We therefore perform experiments with a variety of real-life and synthetic datasets.

For the synthetic datasets, we study three different data distributions that differ in the way values are generated. Figure 5.1 shows a visual representation of each distribution in two dimensions.



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

Figure 5.1: Data distributions in 2 dimensions with 10k points

The skyline is fairly small for correlated data distributions, increases sharply for anti-correlated data distributions, and is somewhere in-between for independent data distributions [5]. Synthetic datasets are generated as follows:

- For independent distributions, all attribute values are generated independently using a uniform distribution
- For correlated distributions, points which are good in one dimension are also good in the other dimensions
- For anti-correlated, points which are good in one dimension are bad in one or all of the other dimensions

We study four real-life datasets that differ in cardinality, dimensionality, and data distribution as explained below. Note that we do not explicitly state the correlation for any of these datasets. However, a casual analysis using the Pearson product-moment correlation coefficient indicates positive correlation of varying degree for all of our real-life datasets.

- Household is a 6-dimensional dataset containing approximately 130k entries, where each entry records the percentage of an annual income spent on six types of expenditures. All values are in the range of $[0, 10000)$
- NBA is a 5-dimensional dataset containing approximately 17k entries, where each entry records performance statistics for a NBA player. All values are in the range of $[0, 10000)$
- Zillow5D is a 5-dimensional dataset containing more than 2M entries about real estate in the United States. Each entry includes number of bedrooms and bathrooms, living area, lot area, and year built. All values are in the range of $[0, 661371480)$
- Zillow6D is a 6-dimensional dataset containing more than 2M entries about real estate in the United States. Each entry includes number of bedrooms and bathrooms, living area, lot area, year built, and tax value. All values are in the range of $[0, 661371480)$

Zillow datasets are also used in [48], however, we use different attributes in our experiments. Specifically, Vlachou et al. use a 5-dimensional subset of Zillow6D which includes tax value, whereas we exclude tax value from the Zillow5D dataset to test different aspects of the algorithms, namely a more cost-intensive real-life distribution. Nevertheless, we recognize the importance of balancing tax value against lot area and other aspects of real estate by including all attributes in the Zillow6D dataset.

5.2 Effect of programming language

Database systems are commonly implemented in low-level compiled languages like C or C++ and it would be reasonable to implement experimental algorithms in the same (or very similar) languages to produce realistic results. Nevertheless, higher-level languages like Java provide some convenient features that can make algorithm implementation less error-prone and allow the code to be closer to the algorithm specification. For instance, automatic garbage collection makes it much easier to implement some concurrent data structures, like a lazy linked list, where deletions can be made simply by removing a reference as opposed to removing the reference, queuing it for deletion and finally delete the node after ensuring that all threads have released the resource.

If the performance characteristics are sufficiently similar, it would be a preferable to use Java. This will most likely avoid hours of debugging, and it allows for a fair comparison to the results published in [41], which are written in Java and publicly available.

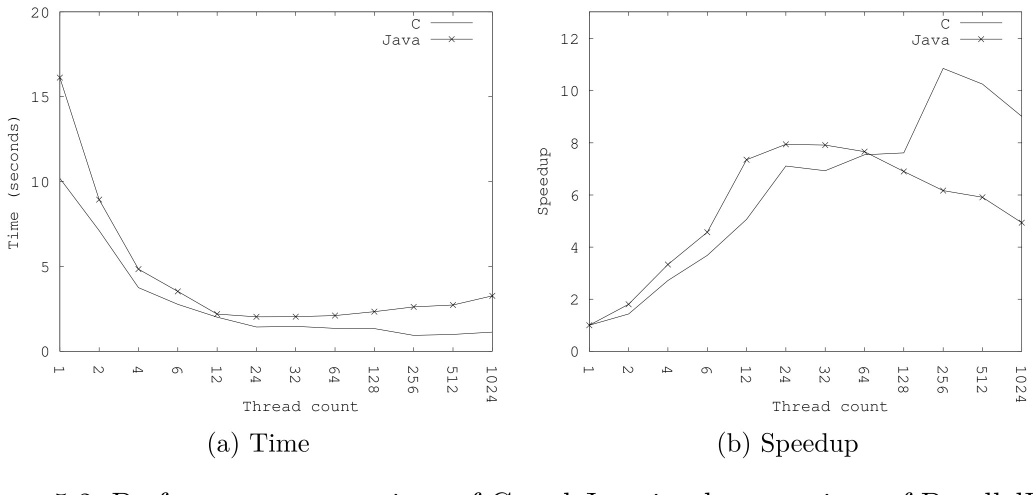


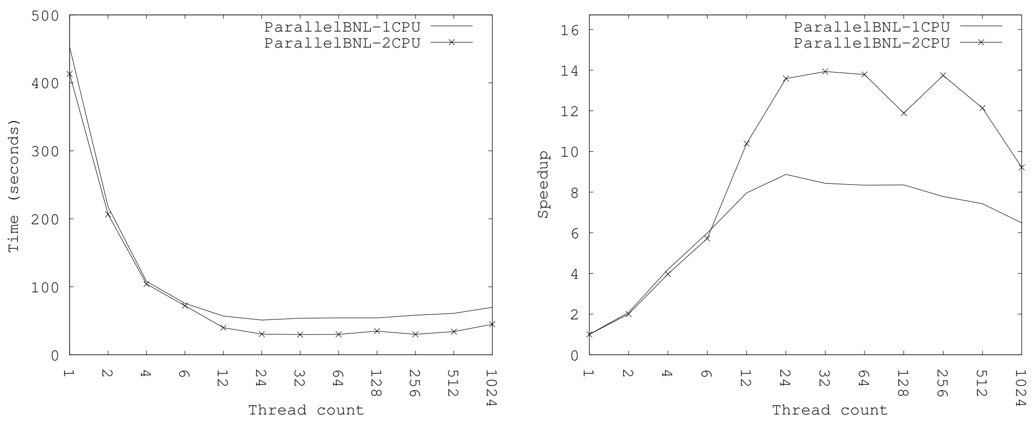
Figure 5.2: Performance comparison of C and Java implementations of ParallelBNL

To determine the feasibility of writing experiments in Java, a microbenchmark was run. The benchmark compares two equivalent implementations of ParallelBNL-LazyList as described in [41]. Figure 5.2 shows that both implementations have similar performance characteristics, and more or less the same speedup when thread count is increased. Based on this result, it seems reasonable to implement our experiments in Java.

5.3 Effect of inter-CPU communication

In this experiment we examine the effects of inter- vs. intra-CPU communication. The test environment consists of two CPUs connected by a fast interconnect, however, it is unlikely that this interconnect is as efficient as communication between cores on one chip. We compare runtime and speedup for ParallelBNL running on one CPU exclusively to the same algorithm running on both CPUs.

Because ParallelBNL use a shared list to store the results, it requires a substantial amount of inter-thread communication. Therefore it is a good algorithm for testing the effect of CPU communication.



(a) Runtime (b) Speedup

Figure 5.3: Performance comparison between ParallelBNL executed with one and two CPUs enabled

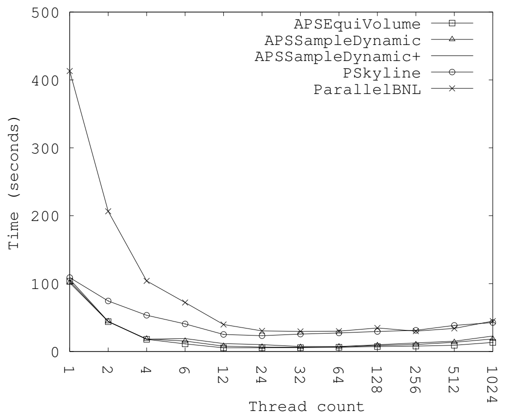
Figure 5.3 shows the results. The speedup graph shows the same speedup that for 6 threads or less, which is to be expected because each CPU has exactly 6 cores. When thread count is increased to 12 we see a big difference between one and two CPUs. Specifically, ParallelBNL-1CPU achieves only a modest increase in speedup, which can be attributed Hyper-Threading, while ParallelBNL-2CPU achieves a speedup close to 11. If the algorithm were run on one CPU with 12 cores, we would expect both algorithms to experience a speedup of exactly 12 when 12 threads were used. However, it is evident that inter-CPU communication comes at a cost. Using two identical CPUs is at most 1.75 times faster than using one CPU in our experiments. That is, we lose about 25% parallel speedup compared to one CPU with the same number of cores.

5.4 Effect of thread count

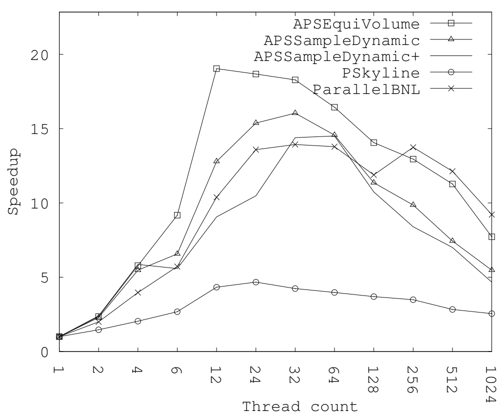
In this experiment, we examine the effect of using multiple cores on the speed of ParallelBNL, PSkyline, and APSkyline. We go from 1 to 1024 threads. It is expected to reach peak performance at 24 threads, which is the maximum number of hardware threads available (12 physical cores + hyper-threading). As number of threads increase beyond 24, we expect that performance will gradually decrease due to increased synchronization costs without additional parallel compute power.

We test the worst-case scenario of skyline computation using an anti-correlated dataset. The skyline of anti-correlated datasets generally have a high cardinality, requiring algorithms to handle large local- and global skylines.

ParallelBNL is expected to be inefficient because skyline tuples are stored in a large shared linked list that need to be iterated through for every input tuple. PSkyline should be able to better utilize parallel compute power with independent local skyline computations. However, because PSkyline does not exploit geometric properties of the dataset in order to limit local skylines, it will most likely do more work than necessary in the merge phase. APSkyline has the same advantages as PSkyline in addition to a strategic partitioning scheme that should limit the size of local skylines, thereby reducing merge costs. This experiment will give insight into RQ4 by investigating if the expensive partitioning phase in APSkyline is able to leverage skyline computation in such a way that APSkyline is given a real performance advantage compared to the trivial scheme used in PSkyline.



(a) Runtime



(b) Speedup

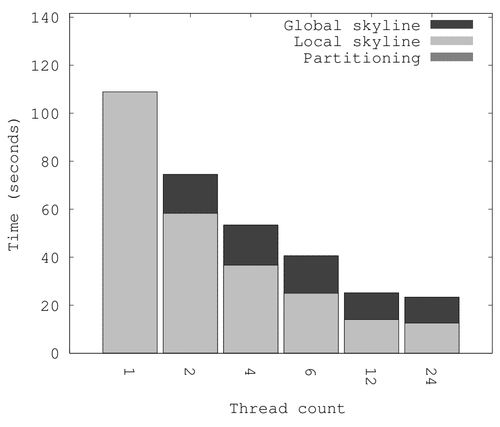
Figure 5.4: Comparison of algorithms running with a different number of threads

Figure 5.4 shows the results. Speedup for each algorithm is relative to the same algorithm run with one thread, not to a common reference point. This was done so that results can be easily compared related articles [37, 41].

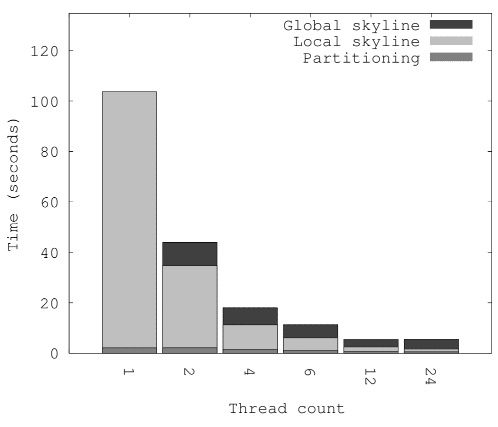
For a low thread count ParallelBNL is inefficient compared to the D&Q based algorithms. This is most likely due to the fact that D&Q based algorithms use SSkyline for for local skyline computation, which, in contrast to ParallelBNL, use an array for storing results. The linked list used in ParallelBNL is not as memory efficient as an array structure for sequential computations.

ParallelBNL has a good speedup as thread count is increased. This is in line with results presented in [41], and shows that a basic algorithm can be quite effective when parallel compute power and low-cost synchronization constructs are available. Using all cores , ParallelBNL shows a performance very close to PSkyline.

PSkyline shows a modest speedup compared to the other algorithms. Surprisingly, independent local skyline computations is not sufficient to beat ParallelBNL to any great degree, even for a worst-case scenario. Figure 5.5 shows the segmented runtime for PSkyline. Local skyline computation shows diminishing performance gains as available parallel compute power increase. This supports our expectations regarding the lack of geometric partitioning. As number of threads increase, PSkyline will produce an increasing amount of local skyline tuples that are discarded in the merge phase.



(a) PSkyline



(b) APSEquiVolume

Figure 5.5: Segmented runtime for PSkyline and APSEquiVolume

APSkyline clearly outperforms other algorithms in this experiment. When all cores are in use (24 threads), APSkyline is **4.2 times faster than PSkyline** and **5.2 times faster than ParallelBNL**. Figure 5.5 shows that the partitioning technique used in APSkyline is more expensive than the one used in PSkyline. Nevertheless, time spent partitioning is more than compensated for in subsequent phases. In Figure 5.4 we can see that APSkyline achieves super-linear speedup for up to 12 threads. Obviously, super-linear speedup cannot be explained parallelism alone, we therefore attribute positive results to a combination of an increased parallel compute power, an increase in high-level cache (each core contributes with its private cache), and smaller input cardinalities for SSkyline. That is, SSkyline does not receive its optimal input cardinality for low thread counts, therefore perceived speedup cannot be attributed parallel compute power alone. APSkyline is able to utilize parallel compute power in every phase as shown by Figure 5.5. Unsurprisingly, there is no significant performance differences between variations of APSkyline. For an anti-correlated all proposed APSkyline variations achieve are able to distribute data fairly.

In summary, we observe that ParallelBNL is very inefficient for a low thread count compared to D&Q-based algorithms. However, as the number of available threads increase, ParallelBNL shows performance characteristics comparable to PSkyline. Furthermore, all variations of APSkyline achieve great speedup and outperforms other algorithms in a significant degree.

5.5 Effect of data dimensionality

Unlike selections where adding a new selection can only decrease the cardinality, adding a new dimension can increase the skyline cardinality, indeed up to the size of the entire relation [8]. This means that an increase in dimensionality will not only increase the input volume in terms of an additional column, it will also increase the number of rows that need to be produced by the operator. It is likely that a small increase in dimensionality will create profound effects on algorithm performance.

To test the effect of different dimensionality, algorithms are run with input dimensions ranging from 2 to 5. We expect APSkyline to be increasingly efficient compared to other algorithms as the input dimensionality increases. For a low dimensionality, the data volume may not be large enough to take advantage of the more expensive pre-processing (partitioning phase) used in APSkyline, giving the other algorithms an advantage.

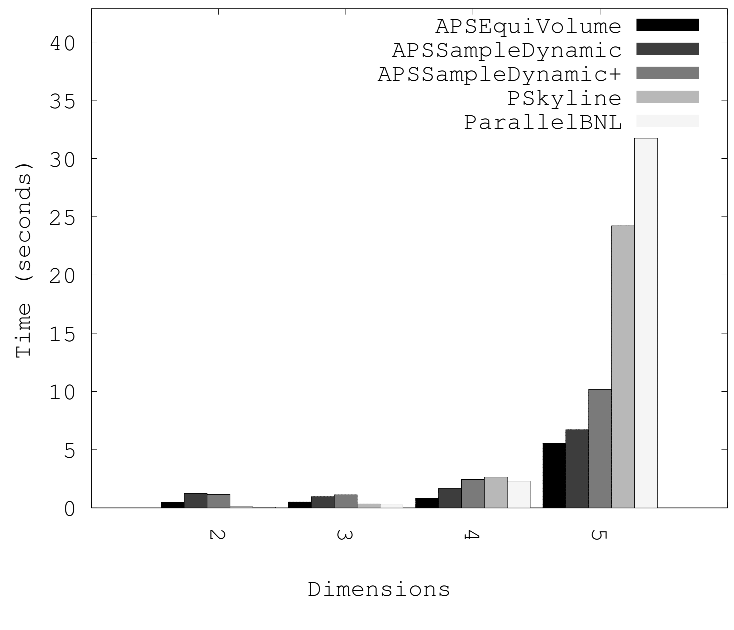
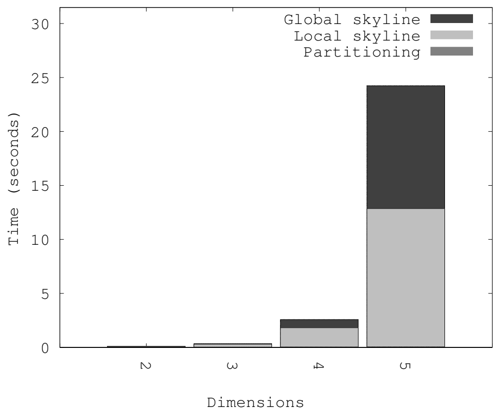


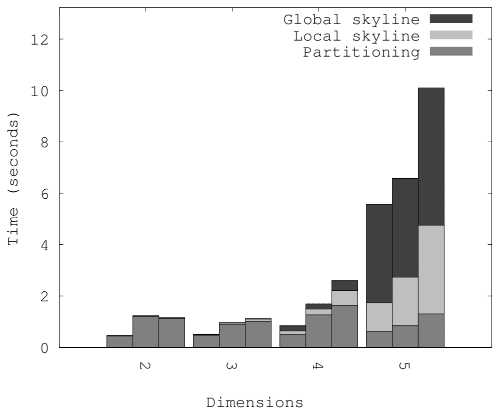
Figure 5.6: Comparison of algorithms running with varying dimensionality

Figure 5.6 shows the results and confirms our expectations. For a dimensionality of 3 or less, ParallelBNL is the most efficient algorithm, while APSkylineSample-Dynamic+ is the least efficient. For a dimensionality of 4, APSEquiVolume and

APSSampleDynamic outperform other algorithms by a small margin. With 5 dimensions or more, all variations of APSkyline significantly outperform other algorithms. In contrast to earlier algorithms, APSkyline scale well with dimensionality, and we expect the same pattern to continue as dimensionality is further increased.



(a) PSkyline



(b) APSkyline

Figure 5.7: Segmented runtime for PSkyline and APSkyline. For APSkyline, results from equi-volume, sample-dynamic, and geometric-random partitioning are shown for each dimensions with equi-volume as the leftmost, and geometric-random as the rightmost bar

Interestingly, the equi-volume partitioning scheme is faster than sample-dynamic and geometric-random schemes. By investigating each phase separately (see Figure 5.7), we observe a performance difference in all phases. This is most likely due to sub-optimal partitioning caused by the greedy partitioning technique used in both of the dynamic partitioning schemes. That is, when we specify partitioning boundaries based on only 1% of the input set, the algorithm may choose a subset that does not accurately represent the dataset as a whole. By inducing random partitioning we make matters worse, causing each partition not only to have sub-optimal partitioning boundaries, but also a number of tuples that are distributed with no consideration for geometric properties. Nevertheless, all variations of APSkyline achieve superior performance compared to PSkyline and ParallelBNL for sufficiently big datasets, and it is likely that sample-dynamic and geometric-random schemes may be more effective for real-life datasets.

In conclusion, ParallelBNL and PSkyline are more efficient for datasets with low dimensionality, while APSkyline are more efficient for datasets with high dimensionality. Furthermore, performance differences increase as dimensionality increase, causing APSkyline to be significantly more efficient than competing algorithms for a 5-dimensional dataset.

5.6 Effect of data cardinality

In this experiment we examine how algorithms scale with an increasing cardinality. Obviously, we expect the runtime for all algorithms to increase with the cardinality. However, we also expect different algorithms to respond differently to cardinality changes. Specifically, we expect ParallelBNL to be most efficient for low cardinalities with PSkyline as a close runner up. Furthermore, we expect APSkyline to be most efficient for high cardinalities.

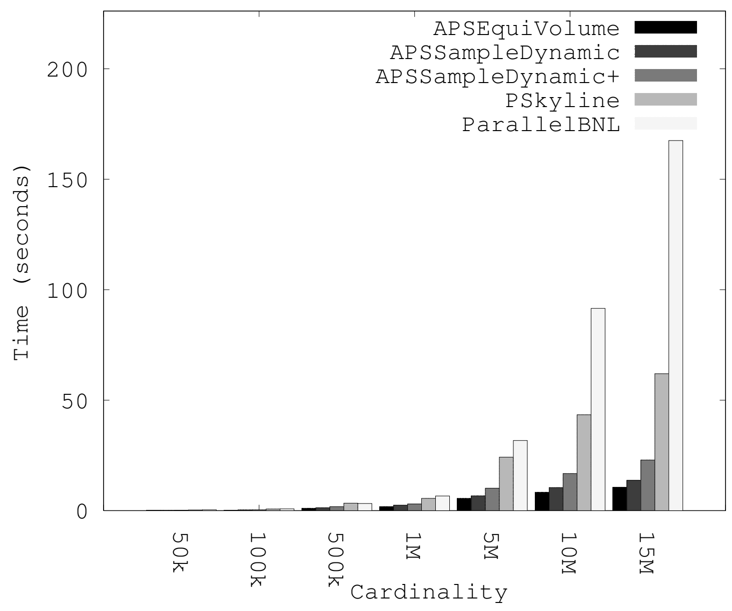


Figure 5.8: Comparison of algorithms running with varying cardinality

Card	APS-EV	APS-SD	APS-SD+	PSkyline	ParallelBNL
50k	162ms	168ms	198ms	309ms	388ms
100k	222ms	352ms	382ms	757ms	803ms
500k	1.00s	1.33s	1.81s	3.39s	3.24s
1M	1.85s	2.51s	3.02s	5.55s	6.65s
5M	5.57s	6.71s	10.18s	24.22s	31.75s
10M	8.30s	10.46s	16.78s	43.40s	91.55s
15M	10.61s	13.74s	22.92s	61.97s	167.5s

Table 5.3: Test results for cardinality experiment. APS-EV is short for APSEquiVolume, APS-SD for APSSampleDynamic, and APS-SD+ for APSSampleDynamic+

To expose details hard to notice in the graph, we supplement with tabular representations. Test results are presented in Table 5.3 and Figure 5.8.

In contrast to our expectations, we observe that APSkyline achieve the best runtime for all input sizes. Table 5.4 shows that small datasets have a large percentage of skyline tuples. This means that, even with small datasets, skyline algorithms have to process a relatively large number of skyline tuples, which will give an advantage to D&Q-based algorithms. APSkylineEquiVolume, the most efficient APSkyline variant for this experiment, are significantly more efficient than competing algorithms for big cardinalities, and outperform ParallelBNL and PSkyline by factors of **15.8** and **5.9**, respectively.

Cardinality	**Skyline size**
50k	18.25%
100k	12.48%
500k	5.35%
1M	3.64%
5M	1.31%
10M	0.82%
15M	0.62%

Table 5.4: Test results for cardinality experiment

Figure 5.9 shows the time used per skyline tuple by each algorithm. It is evident that ParallelBNL does not handle big cardinalities well. Time used per skyline tuple should ideally be unchanged (or decreasing) as cardinality increase. However, since the percentage of skyline tuples are substantially reduced with a bigger cardinality, it is acceptable with a modest increase in processing time per skyline tuple. In this regard, APSkyline is quite successful. Processing time per tuple increase very slowly compared to the other two algorithms, and it is likely that we will see the same behaviour as the cardinality increase further. This is a great example of the applicability of an angle-based partitioning scheme in parallel environments. We attribute APSkylines success to its ability to eliminate non-skyline tuples early. The same reason can be used to explain why the geometric-random partitioning scheme, favoring fairness over geometric heuristics, is somewhat less efficient than the other schemes.

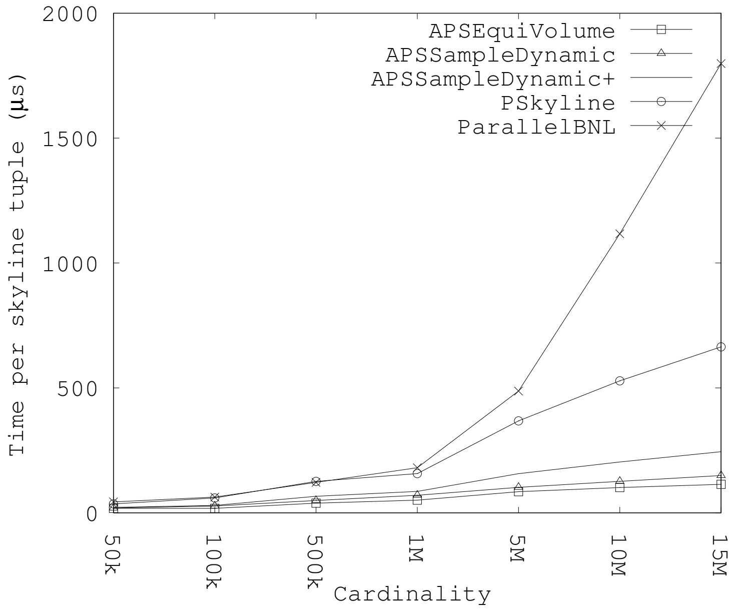


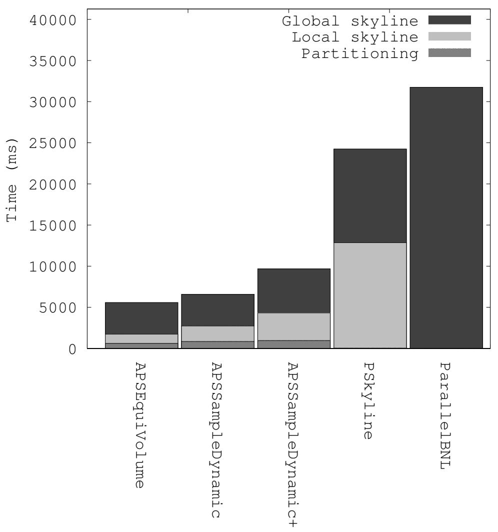
Figure 5.9: Runtime per tuple with increasing cardinality

In summary, ParallelBNL achieves acceptable performance for low cardinalities, but scales poorly for cardinalities greater than 1M, where processing time per tuple increases steeply. PSkyline exhibit the similar behavior as ParallelBNL as we reach cardinalities of 1M or higher, however, not same degree. All variants of APSkyline scales well with increasing cardinality, and are most efficient for all datasets in this experiment.

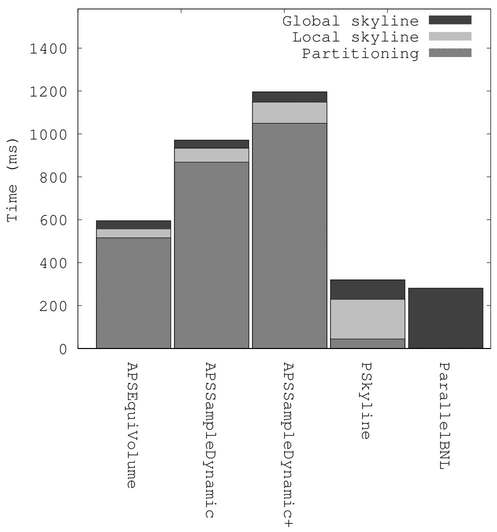
5.7 Effect of data distribution

In this experiment we compare algorithms in terms of data distribution. Börzsöny et al. states that the skyline is fairly small for correlated input, whereas the skyline size for anti-correlated input increases sharply. The size of the skyline for independent input is somewhere in between [5]. We execute each algorithm with three synthetic datasets with correlated, independent, and anti-correlated distributions. Additionally, we execute each algorithm for the three real-life datasets NBA, Household, and Zillow.

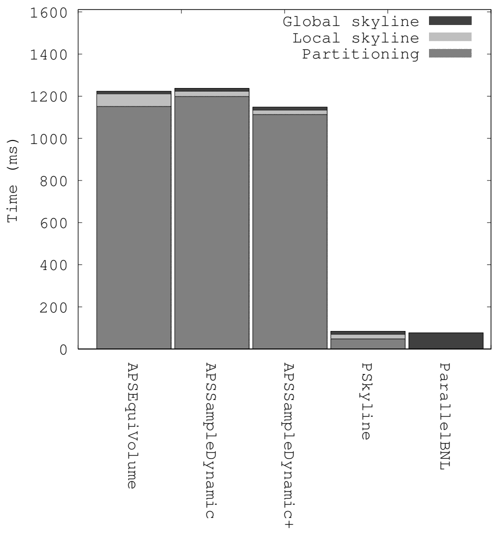
We expect ParallelBNL and PSkyline to be best for the correlated and to some degree for the independent datasets. For the anti-correlated dataset, we expect APSkyline to be most efficient. Skyline computation for correlated datasets are in general a simple problem, requiring little work, therefore simple algorithms with less partitioning costs may have an advantage. As stated in [48], the anti-correlated dataset is most interesting, since the skyline operator aims to balance contradicting criteria [32, 48]. For the real-life datasets, we expect algorithms to have similar performance. However, they are quite small, which can give an advantage to ParallelBNL and PSkyline.



(a) Anti-correlated



(b) Independent



(c) Correlated

Figure 5.10: Segmented runtime for synthetic datasets

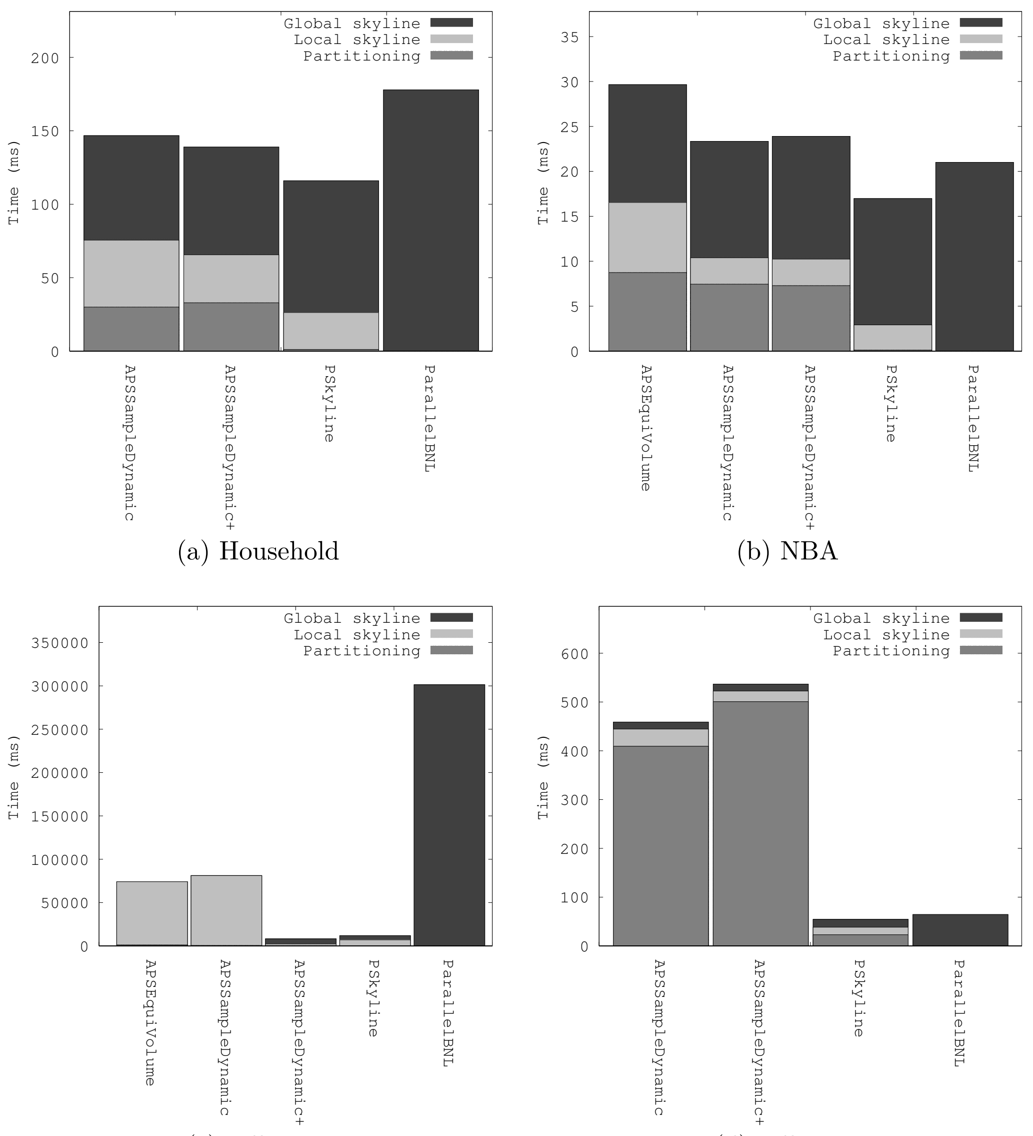
Figure 5.10 shows the results for synthetic datasets. We present the segmented runtime for each algorithm so that the reader can easily see how each phase responds to the different data distributions. Test results match well with our expectations. ParallelBNL and PSkyline are clearly most efficient for the correlated dataset, while APSkyline is superior for the anti-correlated dataset. For the independent dataset algorithms achieve more similar performance characteristics, however, PSkyline and ParallelBNL are noticeably faster than APSkyline.

Angle-based partitioning is particularly efficient for the anti-correlated dataset, where APSkyline achieves the best performance. We observe equi-volume partitioning is most efficient in this case. In an anti-correlated dataset, points will be more or less perfectly distributed by an equi-volume scheme, ensuring fair workloads and early elimination of skyline tuples. Our sample-dynamic scheme use only 1% of the input to generate partitioning boundaries, and may end up with a sub-optimal distribution. The same is true for the geometric-random strategy, in which overflow points may interfere with the angle-based partitioning scheme, further degenerating the angle-based partitioning strategy.

Interestingly, the sample-dynamic partitioning scheme is notably less efficient than the equi-volume partitioning scheme for an independent data distribution. Most of the difference can be accounted for in the partitioning phase, where the dynamic strategy is more expensive. Nevertheless, we also observe that local skyline computation is faster when using equi-volume partitioning. We attribute this to the fact that the sample-dynamic scheme use only a few sampling points to determine the partitioning boundaries, which give a sub-optimal division of labour if the sample points is insufficient to represent the dataset as a whole.

For the correlated dataset ParallelBNL and PSkyline are vastly more efficient than APSkyline variants, which spend most of the time partitioning. For cases where the skyline is small compared to input cardinality, basic algorithms can be very efficient. The reason being that ParallelBNL is able to keep its list of potential skyline tuples small, resulting in fast dominance testing, and that PSkyline are able to produce a small local skylines that can be merged efficiently.

Figure 5.11 shows test results for the real-life datasets. We include only the sample-dynamic partitioning schemes for APSkyline for datasets of dimensionality greater than five because of implementation problems for equi-volume partitioning. This should not be seen as a weakness of the partitioning strategy in itself, but rather an unfortunate implementation detail that we were unable to solve before presenting our results.



(a) Household (b) NBA

(c) Zillow5D (d) Zillow6D

Figure 5.11: Segmented runtime for real-life datasets

PSkyline achieves the best performance for NBA and Household. For these sets D&Q based algorithms show similar performance in the local- and global (merge) skyline computation phases. However, APSkyline spend too much time partitioning and are therefore less efficient than PSkyline. In the NBA dataset, the sample-dynamic partitioning schemes outperforms the equi-volume partitioning scheme. This can be attributed the fact that the dynamic strategy is designed to respond to different data distributions in such a way that work is more fairly distributed. In a real-life dataset, a (sample-)dynamic partitioning scheme will necessarily be more robust than a fixed scheme that does not adapt to its input.

For the Zillow5D dataset we see some interesting results. ParallelBNL is outperformed in an order of magnitude by all other algorithms. In fact, the best-performing algorithm, APSSampleDynamic+ is 36 times faster than ParallelBNL for this case.

Additionally, we observe that PSkyline and APSSampleDynamic+ is spends significantly less time in the local skyline computation phase than APSEquiVolume and APSSampleDynamic.

We attribute ParallelBNLs performance decline to a high percentage of skyline tuples, for a large dataset. About 4% of the overall dataset is part of the skyline. However, as this is the case also for an anti-correlated dataset, and ParallelBNL achieves better performance relative to other algorithms with synthetic datasets, it is unlikely that skyline size is the only cause. The dominant performance factor in ParallelBNL is number of tuples that need to be stored in the linked list, and the duration in which they need to be stored. If, for some reason, this list is filled with many points at an early stage due to a particularly hard data distribution, ParallelBNL will achieve sub-optimal performance. We believe that this is the case for the Zillow5D distribution.

The reason for APSEquiVolume and APSSampleDynamic spending so much time in the local skyline computation phase when processing the Zillow dataset is lack of a fair work distribution. We observed that most points were placed in only a few partitions, causing the majority of threads being idle. PSkyline does not have this problem, as it will always divide data fairly without the use of any heuristics. In this case PSkyline performs very well despite of using a trivial partitioning technique. Nevertheless, APSSampleDynamic+ is able to outperform PSkyline with a factor of 1.4 using an angle-based partition technique in combination with random partitioning.

Another interesting observation is the significant performance gap between Zillow5D and Zillow6D. The only difference between these two datasets is one dimension, namely tax value. However, considering that tax value for a house necessarily is bound to have some form of correlation with nearly all aspects of the house, it is hardly surprising that results are affected. In fact, Zillow5D contains a massive skyline comprised of 91152 tuples, whereas Zillow6D contains only 719 skyline tuples, less than 1 percent of the former. Furthermore, we see distinct similarities between the correlated dataset and Zillow6D. APSkyline spends the majority of the runtime partitioning, while ParallelBNL and PSkyline take advantage of the simple dataset to achieve superior performance.

In summary, algorithms based on angle-partitioning were most efficient for the time consuming datasets, while ParallelBNL and PSkyline excelled for simpler cases. PSkyline performed slightly better than APSkyline for two small real-life datasets and was quite efficient for a work-intensive real-life dataset. Nevertheless APSSampleDynamic+ was able to reduce runtime by approximately 30% compared to PSkyline for the most work-intensive real-life dataset.

5.8 Implications

These results imply that the increasing parallel compute power in modern processors can have a significant impact on performance in DBMSs. Not only for independent queries running in parallel, but also internally in relational operators.

A simple basic-nested-loop [41] was able to efficiently utilize parallel compute power for synthetic datasets and moderately small real-life datasets. When all threads were in use, ParallelBNL achieved performance comparable to PSkyline, even though ParallelBNL performed significantly worse for low thread counts. This indicates that, basic algorithms can be very efficient if they are able to utilize parallel compute power to a sufficient degree.

The D&Q-based skyline algorithm presented in [37] achieved competitive performance characteristics for wide variety of input, and was significantly more efficient than ParallelBNL and APSkyline variations that exclusively used geometric properties in partitioning. However, a variant of APSkyline using geometric-random partitioning to prioritize a fair workload was most efficient for this case. This illustrates the importance of fairness in parallel algorithms. Furthermore it shows that, even though an basic algorithm may exhibit great speedup when parallelized, there is no guarantee that it will be able to compete with more sophisticated algorithms for work-intensive input data.

We observed that an expensive pre-processing step was very effective for skyline computation in a multi-core environment. Specifically, we were in some cases able to significantly outperform competing algorithms by adapting a novel angle-based partitioning technique originally developed for parallel and distributed systems. We therefore emphasize the importance of considering techniques known to be efficient for traditional DBMSs also in a multi-core context, even though they may be compute-intensive.

Each of the tested algorithms excelled for some situations, however, the D&Q-based algorithms were more robust for a real-life input, and in general more stable than ParallelBNL for a variety of datasets. Additionally, APSkyline was significantly more efficient than competing algorithms for work-intensive datasets. In skyline computation, it is reasonable to use heuristics to choose between a number of algorithms in order to maximize performance for all distributions. Therefore, it is important identify characteristics that can be used to determine which algorithm is best for a given dataset, extending the work done in [8, 16, 23, 29] by also considering multi-core algorithms in shared-memory environments.

Chapter 6

Conclusions and future work

Recently multi-core processors have become wide-spread, and it is likely that this this architecture will continue to thrive in the future. Even laptops are equipped with processors of four cores or more. This have led to an increasing interest in shared-memory programming and parallel algorithms. Efficient algorithms and data structures running on shared-memory systems with multi-core processors are needed to utilize the increased compute power.

In this thesis we explored the multi-core landscape in a database context by investigating existing methods and algorithms, developing a novel multi-core skyline algorithm, and by conducting various experiments. In order to identify and explain important design criteria for multi-core algorithms we introduced modern processor architectures in a historical perspective. Additionally, we described the skyline operator, including associated state-of-the art sequential and parallel algorithms. Finally, we conducted various experiments to evaluate how our proposed skyline algorithm performed compared to the current state-of-the-art multi-core algorithms, and discussed the implications of reported results.

6.1 Conclusions

We started this thesis by asking four research questions related to database operators on multi-core architectures. In this section we discuss and answer each question based on observations made in research, development, and experimentation.

RQ1 How can we efficiently exploit multi-core architectures when implementing database operators? Are there cases where this is impossible?

In Section 2.7 we developed a number of design goals for shared-memory algorithms. In short, to efficiently exploit multi-core architectures it is essential that algorithms are designed with a high degree of parallelism and scalability. Algorithms should minimize synchronization costs and memory usage, however, efficient inter-thread communication can in some cases be used to improve performance [31] in a shared-memory system. When inter-thread communication in use, there is a risk of false sharing, which can cause significant performance degradation. Consequently, developers must take special care to avoid memory access patterns causing false sharing, as described in Section 2.6.1. Finally, algorithms should be designed to achieve a fair workload among parallel threads, a parallel algorithm is only as fast as its slowest thread.

RQ2 How can we determine if an algorithm or an operator is viable for multi-core optimizations?

It is widely known that multi-core processors are ideally used for CPU-intensive tasks like skyline computation. Nevertheless, by exploiting low inter-thread communication costs it is also possible to achieve performance gains for less CPU-intensive operators like top-k and join [1, 4]. By analyzing algorithms, looking for independent operations and other characteristics presented in Section 2.7, it should be relatively easy to determine applicability for multi-core optimizations.

RQ3 Is it reasonable to regress into more basic algorithms in order to exploit the increasing parallel compute power in modern processors?

Research done by Selke et al. in [41] suggests that the answer is yes, it may be reasonable parallelize basic algorithms in order to outperform more elaborate algorithms that is inherently sequential or require a substantial amount of pre-processing. Our results indicated that, for CPU-intensive operators like the skyline operator, parallelizing the most basic algorithms may not be sufficient. However, we also observed that basic algorithm did excel for some inputs. Based on our observations, we conclude that regressing to more basic algorithms is reasonable in some cases, but not as a general strategy

RQ4 Can pre-processing techniques from parallel and distributed systems be efficient in a shared-memory context where inter-thread communication is an order of magnitude less expensive?

By adapting the partitioning scheme first published in [48] into a shared-memory system we illustrated that pre-processing techniques directed at parallel and distributed systems can indeed be efficient in a shared-memory environment. We were able to consistently outperform state-of-the-art shared-memory algorithms for skyline computation with anti-correlated data input. For correlated and independent datasets our algorithm spent a considerable part of the runtime in the partition phase, and was not as efficient as more basic algorithms. Nevertheless, observations indicate that such pre-processing techniques have a use also in multi-core systems.

6.2 Future work

There is still much research that can be done for skyline computation, and in general, related to effective utilization of modern processors in DBMSs. It is likely that related operators like skyband [36, 9] and skyline cube [55] can be adapted into multi-core environments using techniques similar to the ones in APSkyline. Additionally, we believe that our research can be used in the process of implementing other CPU-intensive operators like reverse top-k [49, 50] on shared-memory systems.

In order to utilize data parallelism in APSkyline, it should be possible to use SIMD operations when computing angular coordinates by performing multiple multiplications simultaneously (during a single point transformation). Additionally, we see a potential for processing multiple points at the same time utilizing SIMD additions. This requires blurring of the boundaries between points to some degree, however, such details can easily be abstracted into a functions and data structures. If APSkyline are able to more efficiently calculate angular coordinates, the algorithm may be relevant also for less cost-intensive datasets.

The adaptive algorithm suggested in Section 4.5 can be implemented to create a more robust skyline algorithm for shared-memory environments. As a variation, it would be interesting to investigate the possibility of processing many small sample sets using SSkyline in parallel, then combine sample cardinalities to estimate the global skyline cardinality. This may well be inaccurate, however, by choosing small sample sizes it can be done very efficiently by utilizing parallel compute power.

In this thesis, all experiments were executed on the Intel Nehalem architecture. It possible that one would achieve different results by executing experiments on another architecture, like the AMD Bulldozer. The AMD Bulldozer is equipped with a greater number of cores, grouped pairwise into modules. Each pair sharing components like the floating-point unit and early pipeline stages. Because our experiments make heavy use of floating-point operations in order to calculate the skyline, such an architecture may achieve sub-optimal performance.

Bibliography

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB)*, volume 5, pages 1064–1075. VLDB Endowment, 2012.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference (SJCC)*, pages 483–485. ACM, 1967.

[3] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems (TODS)*, 33(4):31, 2008.

[4] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 37–48. ACM, 2011.

[5] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering (ICDE)*, pages 421–430. IEEE, 2001.

[6] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the 10th International Conference on Very Large Data Bases (VLDB)*, pages 323–333. VLDB Endowment, 1984.

[7] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity analysis of the upc language. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 254. IEEE, 2004.

[8] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *Proceedings of the 22th International Conference on Data Engineering (ICDE)*, pages 64–64. IEEE, 2006.

[9] L. Chen, J. Zhao, Q. Huang, and L. H. Yang. Effective space usage estimation for sliding-window skybands. *Mathematical Problems in Engineering*, 2010, 2010.

[10] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 717–719. IEEE, 2003.

[11] A. Cosgaya-Lozano, A. Rau-Chaplin, and N. Zeh. Parallel computation of skyline queries. In *Proceedings of High Performance Computing Systems and Applications (HPCS)*, pages 12–12. IEEE, 2007.

[12] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *Computational Science & Engineering*, 5(1):46–55, 1998.

[13] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational geometry*. Springer, 1997.

[14] P.-K. Eng, B. C. Ooi, and K.-L. Tan. Indexing for progressive skyline computation. *Data & Knowledge Engineering*, 46(2):169–201, 2003.

[15] M. Frigo and S. G. Johnson. Fftw: An adaptive software architecture for the fft. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 3, pages 1381–1384. IEEE, 1998.

[16] P. Godfrey. Skyline cardinality for relational processing. In *Foundations of Information and Knowledge Systems (FoIKS)*, pages 78–97. Springer, 2004.

[17] P. Godfrey, R. Shipley, and J. Gryz. Maximal vector computation in large data sets. In *Proceedings of the 31th International Conference on Very Large Data Bases (VLDB)*, pages 229–240. VLDB Endowment, 2005.

[18] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. volume 16, pages 5–28. Springer-Verlag New York, Inc., 2007.

[19] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *ACM SIGPLAN Notices*, volume 28, pages 177–186. ACM, 1993.

[20] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems*, pages 3–16. Springer, 2006.

[21] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[22] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*

[23] M. Karnsteclt, J. Muller, and K.-U. Sattler. Cost-aware skyline queries in structured overlays. In *Proceedings of the 23th International Conference on Data Engineering (ICDE)*, pages 285–288. IEEE, 2007.

[24] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 85–96, 2011.

[25] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-

core cpus. In *Proceedings of the 38th International Conference on Very Large Data Bases (VLDB)*, volume 5, pages 61–72. VLDB Endowment, 2011.

[26] V. Kumar and A. Gupta. Analysis of scalability of parallel algorithms and architectures: a survey. In *Proceedings of the 5th international conference on Supercomputing*, pages 396–405. ACM, 1991.

[27] H.-T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[28] C. E. Leiserson and H. Prokop. A minicourse on multithreaded programming.

[29] Y. Lu, J. Zhao, L. Chen, B. Cui, and D. Yang. Effective skyline cardinality estimation on data streams. In *Proceedings of the 19th International Conference on Database and Expert Systems Applications (DEXA)*, pages 241–254. Springer, 2008.

[30] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Frameworks for multi-core architectures: a comprehensive evaluation using 2d/3d image registration. In *Proceedings of the 24th international conference on Architecture of computing systems (ARCS)*, pages 62–73, Berlin, Heidelberg, 2011. Springer-Verlag.

[31] M. Meneghin, D. Pasetto, H. Franke, F. Petrini, and J. Xenidis. Performance evaluation of inter-thread communication mechanisms on multicore/-multithreaded architectures. In *Proceedings of 21st International ACM Symposium on High-Performance Parallel and Distributed Computing (HDPC)*, 2012.

[32] M. Morse, J. M. Patel, and H. Jagadish. Efficient skyline computation over low-cardinality domains. In *Proceedings of the 33th International Conference on Very Large Data Bases (VLDB)*, pages 267–278. VLDB Endowment, 2007.

[33] *Oracle TimesTen In-Memory Architectural Database Overview*, 2012.

[34] P. Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann, 2011.

[35] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 467–478. ACM, 2003.

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

[37] S. Park, T. Kim, J. Park, J. Kim, and H. Im. Parallel skyline computation on multicore architectures. In *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, pages 760–771. IEEE, 2009.

[38] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 61–72. ACM, 2012.

[39] K. H. Randall. *Cilk: Efficient Multithreaded Computing.* PhD thesis, Massachusetts Institute of Technology, 1998.

[40] M. Scott and W. Bolosky. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, page 57. Usenix Assoc, 1993.

[41] J. Selke, C. Lofi, and W.-T. Balke. Highly scalable multiprocessing algorithms for preference-based database retrieval. In *Proceedings of the 15th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 246–260. Springer, 2010.

[42] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, pages 510–510. VLDB Endowment, 1994.

[43] M. Stonebraker. The case for shared nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

[44] X.-H. Sun and L. M. Ni. Another view on parallel speedup. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 324–333. IEEE, 1990.

[45] K.-L. Tan, P.-K. Eng, B. C. Ooi, et al. Efficient progressive skyline computation. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, pages 301–310, 2001.

[46] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–12. ACM, 2012.

[47] R. Torlone and P. Ciaccia. Finding the best when it's a matter of preference. In *Proceedings of the 10th Italian National Conference on Advanced Database Systems (SEBD)*, 2002.

[48] A. Vlachou, C. Doulkeridis, and Y. Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 227–238. ACM, 2008.

[49] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *Proceedings of the 26th International Conference on Data Engineering (ICDE)*, pages 365–376. IEEE, 2010.

[50] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis. Branch-and-bound algorithm for reverse top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD).* ACM, 2013.

[51] A. Vlachou, C. Doulkeridis, K. Nørvåg, and M. Vazirgiannis. On efficient top-k query processing in highly distributed environments. In *Proceedings of the*

ACM SIGMOD International Conference on Management of Data (SIGMOD), pages 753–764. ACM, 2008.

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

[53] T. Willhalm and N. Popovici. Putting intel® threading building blocks to work. In *Proceedings of the 1st international workshop on Multicore software engineering (IWMSE)*, pages 3–4. ACM, 2008.

[54] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *Proceedings of the 7th International Workshop on Data Management on New Hardware (DaMoN)*, pages 1–9. ACM, 2011.

[55] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proceedings of the 31th International Conference on Very Large Data Bases (VLDB)*, pages 241–252. VLDB Endowment, 2005.

[56] Z. Zhang, Y. Yang, R. Cai, D. Papadias, and A. Tung. Kernel-based skyline cardinality estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 509–522. ACM, 2009.

[57] J. Zhou, N. Bruno, and W. Lin. Advanced partitioning techniques for massively distributed computation. In *Proceedings of the 2012 international conference on Management of Data*, pages 13–24. ACM, 2012.