Template Skycube Algorithms for Heterogeneous Parallelism on Multicore and GPU Architectures

Kenneth S. Bøgh
Aarhus University, Denmark
ksb@cs.au.dk

Sean Chester
NTNU, Norway
sean.chester@idi.ntnu.no

Darius Šidlauskas
EPFL, Switzerland
darius.sidlauskas@epfl.ch

Ira Assent
Aarhus University, Denmark
ira@cs.au.dk

ABSTRACT

Multicore CPUs and cheap co-processors such as GPUs create opportunities for vastly accelerating database queries. However, given the differences in their threading models, expected granularities of parallelism, and memory subsystems, effectively utilising all cores with all co-processors for an intensive query is very difficult. This paper introduces a novel templating methodology to create portable, yet architecture-aware, algorithms. We apply this methodology on the very compute-intensive task of calculating the *skycube*, a materialisation of exponentially many skyline query results, which finds applications in data exploration and multi-criteria decision making. We define three parallel templates, two that leverage insights from previous skycube research and a third that exploits a novel point-based paradigm to expose more data parallelism. An experimental study shows that, relative to the state-of-the-art that does not parallelise well due to its memory and cache requirements, our algorithms provide an order of magnitude improvement on either architecture and proportionately improve as more GPUs are added.

1. INTRODUCTION

Modern, affordable, highly-parallel co-processors such as general purpose graphics processing units (GPUs) may vastly accelerate database servers; how to incorporate GPUs into query processing is thus a very active area of research [6]. However, the prominent approach of scheduling queries to either the CPU or the GPU misses the opportunity to exploit *all* parts of a heterogeneous database server for really expensive data management tasks. We aim for a "cross-device parallelism" in which all devices (i.e., multiple CPU chips and multiple co-processors) contribute proportionately to the result. If a) each GPU can produce results as efficiently as the CPU cores by minimising excess instructions and b) the CPU cores reach comparable throughput to the GPUs, then, combined, one can obtain a significant speed-up over either architecture. Developing non-naive algorithms that fully utilise both multicore and GPU architectures is quite challenging, however, given their different threading models and mechanisms for hiding latencies (see Section 2.3).

SIGMOD'17, May 14 - 19, 2017, Raleigh, NC, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3035962

	Flight Numbers	Price	Duration	Arrival
f_0	860→485→4759	\$ 120	17 hr	12.20
f_1	1264→661	\$ 148	12 hr	09.00
f_2	860→3655	\$ 169	13 hr	08.20
f_3	1260→659	\$ 186	3 hr	21.25
f_4	1258→659	\$ 196	5 hr	21.25

Table 1: Five selected flights from point A to B. Flights f_0-f_3 offer appealing trade-offs, but f_4 is not in the skyline: it is more expensive, longer, and no earlier to arrive than f_3.

Our approach is hardware-aware in the hot spots: we first define abstract parallel *template* algorithms (Section 4) that define a high-level hardware-oblivious strategy for exposing parallelism. Each template specifies *hooks* where the primary parallel processing occurs. We then define concrete *specialisations* for each architecture (CPUs in Section 5; GPUs in Section 6) that implement the hooks. The specialisations permit designing for specific hardware differences, such as the limited state available to each GPU thread and the necessity for cache-consciousness on the CPU. As a result, our algorithms can run on a specific architecture by using just that specialisation, or we can distribute the parallel processing tasks across all physical CPU cores and GPU devices by using both architecture-specific specialisations.

We apply this approach to the NP-hard task [31] of computing a skycube [30, 41], which is the materialisation of $2^d - 1$ skyline query results. Briefly (see Section 2.2), the skyline operator [5, 11] selects from a dataset only those tuples that express some appealing trade-off of attributes. For example, to choose a cheap, quick, early-arriving flight from the options in Table 1, a user need not see flight f_4: it arrives at the same time as f_3, but takes longer and costs more. Tuples that are no better, relative to some distinct competitor tuple, on any single attribute (such as f_4 relative to f_3) are said to be *dominated* and removed by the skyline operator, thereby reducing the decision space for the user.

Unfortunately, skylines lose selectivity as the number of attributes increases. Many of the points may be included because they have good values on attributes of no interest to a particular user. For a business traveller unconcerned by a price range of only \$ 70, flight f_0 in Table 1 is clearly unappealing: it is dominated by (both slower and later than) f_1 and f_2 with respect to duration and arrival time. Thus, one often projects the data into a relevant subset of the attributes (i.e., *subspace*) to compute the skyline. The skycube is the materialised skyline in all possible projections.

Although a compute-intensive task, the only parallel skycube algorithm is a distributed version [36] of the older "bottom-up" Orion algorithm [34]. Yet, skycube materialisation is suited to shared-memory parallelism: datasets are typically memory-resident on a single device, and data points are repeatedly compared to each other in different subspaces; so, comparison results should be reused [22,24,31] and communicated between cores. Moreover, the state-of-the-art sequential algorithm, QSkycube [22,24] is compute-bound. However, as QSkycube is a very memory-intensive, pointer-based, tree-traversal algorithm, a CPU parallel version does not remain compute-bound: cores start competing for limited L3 cache. A GPU version is difficult to envision.

We introduce three parallel skycube templates with accompanying CPU and GPU specialisations. Our experimental evaluation (Section 7) analyses the templates, both at a high level (i.e., execution time) and an architectural level (i.e., hardware counters). When deployed cross-device on our entire heterogeneous ecosystem (2 CPU sockets and 3 Nvidia GPUs from 2 generations), our template that exposes the most data parallelism accelerates skycube construction by $> 150\times$. Notably, the generality of our template design promotes extension to other heterogeneous systems.

Contributions and outline This paper introduces a novel template-specialisation methodology to target distinct shared-memory architectures, which is applied to the NP-Hard [31] problem of computing skycubes [30,41]. After introducing background material (Section 2) and detailing related work (Section 3), but before concluding (Section 8), we:

- Outline three templates for exposing shared-memory parallelism in skycube computation (Section 4);
- Give architecture-specific template specialisations for multicore CPUs (Section 5) and GPUs (Section 6); and
- Evaluate the relative advantages of the parallel strategies with high- and hardware-level metrics (Section 7).

2. BACKGROUND AND PRELIMINARIES

This section introduces notation (Section 2.1), formally defines skycubes (Section 2.2), and provides background context on multicore and GPU parallelism (Section 2.3).

2.1 Notation: point sets and bitmasks

The standard notation for skycubes (c.f., [22,24,31]) is to denote the dataset as a set P of n points[1] over a set $\mathcal{D}$ of d dimensions. *Subspace skylines* (defined in Section 2.2) are computed for each subspace projection, specified as a subset of $\mathcal{D}$. We, too, denote the dataset as a set, P, using array notation for points: $p = (p[d-1], \ldots, p[0])$. However, not all of our algorithms are structured as iterations over the subsets of $\mathcal{D}$; so, we instead adopt a bitmask notation for subspaces that fits better for all of our algorithms.

We represent each subspace by a bitmask δ in which the i'th bit is set iff the subspace includes the i'th dimension. For example, the business traveller viewing Table 1 is interested in the two-dimensional subspace $\delta = 3$, corresponding to {Duration, Arrival}. $|\delta|$ denotes the number of active dimensions in subspace δ (i.e., the number of bits set). Generally speaking, δ' is a subspace of δ iff $(\delta \mathbin{\&} \delta') = \delta'$ (i.e., all bits set in δ' are also set in δ), necessitating that $|\delta'| \leq |\delta|$.

Following [2,8,21,23,42,43] and for the sake of point-based partitioning (described in Section 2.2), we also denote with a bitmask the per-dimension relationship between two points. Bit i of $\mathcal{B}_{p \oplus q}$ is set iff $p[i] \oplus q[i]$. For example, given f_0 and f_1 in Table 1, $\mathcal{B}_{f_0 \leq f_1} = 100$, $\mathcal{B}_{f_1 \leq f_0} = 011$, and $\mathcal{B}_{f_0 = f_1} = 000$.

Finally, following [4], we overload $\mathcal{B}$ to also denote a per-subspace membership relation; e.g., $\mathcal{B}_{f_1 \in \mathcal{S}} = 1110100$ denotes that flight f_1 is in subspace skylines for $\delta \in \{7, 6, 5, 3\}$. (Note that $\mathcal{B}$ is shifted right by 1, since $\delta = 0$, the empty subspace, will not be used.)

2.2 Skycubes and subspace skylines

As a skycube is a set of subspace skylines and a skyline is a set of non-dominated points, we begin by defining *dominance*. A point q is dominated by a distinct point p if there is no dimension i on which $q[i]$ is better than $p[i]$.Then point p is clearly better than point q. In terms of our notation, and applied to subspace δ, p dominates q iff all bits of δ are set in $\mathcal{B}_{p \leq q}$[2] but some bit of δ is not set in $\mathcal{B}_{p=q}$:

Definition 1 (Subspace Dominance [5]). Given points p, q and a subspace $0 < \delta < 2^d$, we say p *dominates* q in δ, denoted $p \prec_\delta q$, iff $(\mathcal{B}_{p=q} \mathbin{\&} \delta) \neq \delta$ and $(\mathcal{B}_{p \leq q} \mathbin{\&} \delta) = \delta$. Moreover, we say p *strictly dominates* q in δ, denoted $p \prec\!\!\prec_\delta q$, iff $p \prec_\delta q$ and $(\mathcal{B}_{p=q} \mathbin{\&} \delta) = 0$ (i.e., $(\mathcal{B}_{p<q} \mathbin{\&} \delta) = \delta$).

We verify that $f_1 \prec f_0$ in subspace $\delta = 011$, since $\mathcal{B}_{f_1 \leq f_0} \mathbin{\&} 011 = 011$ and $f_1[i] \neq f_0[i], \forall i$. Moreover, f_3 strictly dominates f_4 in subspace $\delta = 110$ (i.e., $f_3 \prec\!\!\prec_6 f_4$) but merely dominates f_4 in $\delta = 111$ (i.e., $f_3 \prec_7 f_4$, but $f_3 \not\prec\!\!\prec_7 f_4$).

Subspace and *extended* skylines are then defined as the set of points that are not subspace or strictly dominated:

Definition 2 (Subspace Skyline [5,30]). Given P and $0 < \delta < 2^d$, the *skyline* of P in δ, denoted $\mathcal{S}_\delta(P)$, is the subset: $\mathcal{S}_\delta(P) = \{q \in P : \nexists p \in P, p \prec_\delta q\}$. Similarly, the *extended skyline* [37] of P in δ is: $\mathcal{S}^+_\delta(P) = \{q \in P : \nexists p \in P, p \prec\!\!\prec_\delta q\}$.

For example, the skyline of Table 1 in subspace $\delta = 3$ is $\mathcal{S}_3(P) = \{f_1, f_2, f_3\}$, since f_0 is dominated in $\delta = 3$ by f_1 and f_2 and f_4 is dominated by f_3. The extended skyline, $\mathcal{S}^+_3(P)$ also includes f_4, because although $f_3 \prec_{011} f_4$, they share a common arrival time. Subspace skylines are more selective than the skyline. The extended skyline, unlike the skyline, of δ has the key property of necessarily containing the (extended) skyline of all subspaces of δ [37]. In other words, the extended skyline can be used as a smaller input than P for computing skylines and skycubes.

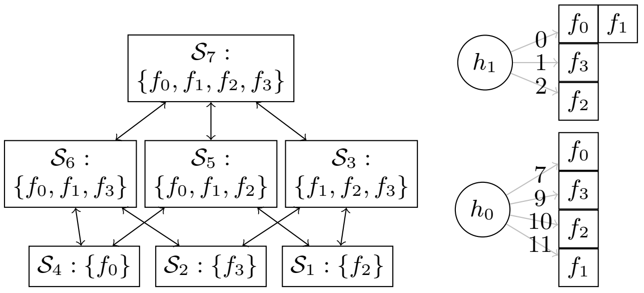
The *skycube* is then the materialisation of the subspace skyline for all subspace projections:

Definition 3 (Skycube [30,41]). The *skycube* of P is a materialised map from each subspace $0 < \delta < 2^d$ onto $\mathcal{S}_\delta(P)$.

Skycube representations Figure 1 illustrates the skycube for the flight data (Table 1) under two separate representations, the lattice and the HashCube [4]. The lattice (Figure 1a) is a common data cube data structure and is

[1]Points implicitly have ids, so can be otherwise indistinct.

[2]WLOG, we assume that smaller values are better.


$\mathcal{S}_7 : \{f_0, f_1, f_2, f_3\}$
$\mathcal{S}_6 : \{f_0, f_1, f_3\}$
$\mathcal{S}_5 : \{f_0, f_1, f_2\}$
$\mathcal{S}_3 : \{f_1, f_2, f_3\}$
$\mathcal{S}_4 : \{f_0\}$
$\mathcal{S}_2 : \{f_3\}$
$\mathcal{S}_1 : \{f_2\}$
h_1
0
1
2
f_0
f_1
f_3
f_2
h_0
7
9
10
11
f_0
f_3
f_2
f_1

(a) Lattice (b) HashCube [4]

Figure 1: Skycube representations, constructed from Table 1

used in all existing skycube algorithms. For each subspace δ, a flat array stores the point ids for $\mathcal{S}_\delta(P)$. The subspaces may be ordered as an array or, as in Figure 1a, with pointers between each subspace and its immediate superspaces. The drawback of the lattice is its redundancy: here, each id is stored 4 times for 7 subspaces. Skycube-specific compression techniques [18, 34, 39, 40] can be applied to a fully-constructed lattice to reduce its redundancy.

The HashCube [4] (Figure 1b; reviewed in Appendix B.1) stores each point p based on $\mathcal{B}_{p \notin \mathcal{S}}$. The bitmask is split into 32-bit "words" that are hashed independently; thus, a point id is stored once per 32 subspaces. The relevant difference is that the HashCube is defined with respect to each point p (its bitmask, $\mathcal{B}_{p \notin \mathcal{S}}$), whereas the lattice is defined with respect to each subspace δ (its skyline, $\mathcal{S}_\delta(P)$).

Point-based partitioning Recent shared-memory skyline algorithms [2, 8, 21, 23, 42, 43] avoid explicit point-to-point comparisons using transitivity with respect to a common "pivot" point. Appendix B.2 reviews this. Briefly, a quad tree partitioning of P (e.g., in Figure 3) naturally provides "pivot" points: two nodes in the tree have common ancestors which serve as pivots. The two nodes can be compared explicitly dimension-by-dimension, i.e., an exact *dominance test* (DT), or by simply comparing their respective relationships to their common ancestors, i.e., a *mask test* (MT). A DT loads up to $|\delta|$ `floats` for each point, whereas an MT only requires one, lessening the burden on the memory subsystem and improving cache hit ratios. However, a DT is required, anyway, if an MT is inconclusive.

2.3 Architecture considerations

When a memory load is issued, a latency is incurred until the load completes. Given the speed of compute resources relative to memory, latencies are inevitable in many applications where memory loads cannot be easily predicted, particularly as more cores compete for a fixed amount of shared cache. Latencies starve compute throughput because operands are not available in registers when instructions are ready, so both CPUs and GPUs have mechanisms to "hide" them. Designing algorithms with these mechanisms in mind is necessary to achieve parallel scalability and full hardware utilisation. Correspondingly, we briefly review some of these mechanisms and other pertinent architecture specifics.

Utilising cache A primary way to reduce latencies is to read from lower levels of cache. The CPU has a deep, multi-level cache hierarchy and sophisticated prefetchers to load data speculatively into cache before it is needed. The L1 and L2 caches are local to each core, whereas L3 is local to a socket; so, threads compete for and/or cooperate in L3 cache. The large globally-shared main memory incurs non-uniform access times per core (i.e., NUMA), since not all cores reside on the same socket. While the best way to reduce the number of latencies is to eliminate unnecessary loads, using smaller data structures can reduce the high-latency traffic that crosses the intersocket link and the competition for L3 cache. Ensuring that algorithms are predictable increases the success rate of the prefetchers.

Each GPU, on the other hand, has one uniform, globally-shared main memory (called *global memory*) with very high bandwidth and very high latency (connected to the CPU via PCIe3). Data can be loaded from global memory through three separate L1 caches, two of which are faster because they only permit read-only data; the other is directly addressable. The L2 cache is very small relative to the CPU and there is no L3 cache. Thus, it is particularly important to fully utilise every cache line using coalesced reads; i.e., consecutive loads should access physically consecutive values on the same cache line. Coalesced reads can improve memory throughput eight-fold (for 4-byte values on a 32-byte cache line) since only one eighth of the loads goes to global memory. Data structures with linear layouts that are traversed sequentially lead to coalesced reads.

Threading model GPUs and CPUs have different threading models. CPU threads are autonomous, fast (3.0+ GHz), and complex with out-of-order execution, branch prediction, and several layers of data prefetching. High independence among instructions (i.e., *instruction-level parallelism*, ILP) gives a CPU core flexibility in each cycle to find instructions with operands available in registers. The CPU also hides latencies with hyper-threading, which alternates the context of two threads on each core so that the ratio between memory and compute speeds is halved. Algorithms with easy-to-predict conditional statements have better branch prediction, and therefore throughput. AVX2-enabled CPUs have 8-wide SIMD (single-instruction-multiple-data) lanes; so, a register can concurrently apply the same operation to eight 32-bit values (i.e., leverage *data-level parallelism*, DLP).

GPU threads are comparatively slow ($\approx$ 1.1 GHz) and step-locked in batches of 32 (called *warps*) that always execute the same instruction. The threads in a warp should avoid branch divergence: disagreement on conditional statements serialises execution, with some threads evaluating one side of the condition while the others wait before the roles are reversed. They have a limited (configurable) amount of state: a small, fixed number of registers and shared L1/texture cache is available. Using too much state per thread permits fewer concurrent threads, sacrificing throughput. The GPU hides (very long) latencies by switching thousands of contexts, depending on which have instructions with operands ready; so, one should expose substantial DLP to create far more contexts than there are physical resources.

3. RELATED WORK

GPUs have been used to accelerate query processing in both industry [27, 28] and academia [6, 15, 16, 32, 33]. However, these works mainly focus on scheduling query operators to either the CPU or the GPU, while only two [15, 32] consider cross-device parallelism. OmniDB [15] partitions data horizontally and invokes hardware-oblivious OpenCL operator implementations on each partition. Pirk et al. [32] use the GPU to filter the result on the most significant bits of each data value and the CPU to refine the result using the entire data value. Our work, on the other hand, proposes maximising the concurrent utilisation of both architectures by sharing data structures and parallel tasks between the devices and defining hardware-conscious specialisations.

Skycubes and lattice-traversal algorithms were introduced independently by Yuan et al. [41] and Pei et al. [30], whose insights were later combined [31]. The *bottom-up* strategy [41] begins with low-dimensional subspaces and exploits that $\mathcal{S}_{\delta'} \subseteq \mathcal{S}_{\delta}, \forall \delta' \subset \delta$ in a breadth-first traversal. So, only points not in some such δ' must be explicitly verified with DTs. The *top-down* strategy [30,41] begins with high-dimensional subspaces and, making accommodations for duplicate values, uses parent cuboids as input for their direct successors. The TDS algorithm [41] reuses partial results of a divide & conquer algorithm whereas the Skyey algorithm [30] reuses sort computations from parent cuboids. Both showed top-down to be substantially faster than bottom-up. The Orion algorithm [34] traverses bottom-up anyway to compress the lattice by only recording each point p in the lowest-dimensional subspaces in which it is not dominated. Veloso et al. [36] give a distributed version of Orion. Kailasam et al. [19] transform the values in each dimension into unary ranks to which they apply binary operators to derive the skycube. QSkycube [22, 24] improves TDS by replacing the divide and conquer algorithm with the more efficient BSkyTree point-based partitioning method [21,23].

Tao et al. [35] introduce the SUBSKY method to answer ad-hoc subspace skyline queries without materialisation. It iterates a B-tree that organises points by their L_∞ distance to an assigned anchor point, using the property that points cannot be dominated by those with larger distances. The algorithm does not perform well for $d > 5$ [18]. The SKYPEER algorithm [37] handles ad-hoc subspace skyline queries over peer-to-peer networks by communicating the *extended skyline* (Definition 2) between peers. Rather than full materialisation, Jin et al. [18] construct a subspace skyline index based on maximum partial dominating spaces.

The lattice can be too large for skycube construction [24]. To compress it, the *closed skycube* [34] splits subspaces into equivalence classes within which points are not duplicated. The *compressed skycube* [39, 40] defines *minimal subspaces* (MSs) and builds a bipartite membership graph from points to MSs to remove redundancy, but only scales to $d \leq 6$ [4,18]. The former requires an inefficient bottom-up strategy; the latter, like the MSIndex [18], cannot be employed until the entire lattice is known (so the memory consumption remains). Skylists [41] compress cuboids in a depth-first manner while the HashCube [4] compresses them breadth-first.

There are no single-node parallel skycube algorithms, but skylines can be computed on FPGAs [38], GPUs [1, 2, 10], and multicore CPUs [3, 8, 9, 12, 13, 17, 20, 26, 29]. For the CPU, PSkyline [17, 29] is a naive divide-and-conquer algorithm that distributes the data to each core, independently computes a local skyline, and then merges the results. APSkyline [26], APS [20], and PPPS [20] extend this approach with better data partitioning; the former two are based on spherical coordinates and the latter, hyperplane projections. VSkyline [9] orthogonally proposes using SIMD registers to accelerate DTs, which are available on modern multicore machines. Scalagon [12,13] constructs a partial order of data values and traverses the resultant partial order lattice to produce the skyline, which is effective when the number of distinct values for each attribute is low. Hybrid [8] applies tiling and point-based partitioning. For the GPU, SkyAlign [2] applies a statically-defined quad tree approach, whereas the `GNL` [10] and `GGS` [1] algorithms focus on throughput with the latter first sorting the data. A detailed study [3], subsequent to this work, evaluates the performance characteristics of SkyAlign and `GGS` when ported to multicore platforms.

The point-based partitioning skyline algorithms [2, 8, 21, 23, 42, 43] vary primarily in how they define their trees. OSP [43] is recursive and uses a random skyline point as a pivot for each sub-partition. BSkytree [21, 23] is also recursive, but selects as pivot the skyline point with the smallest scaled L_1 distance from the origin. VMPSP [42] and Hybrid [8] are recursive, using the median of each dimension to construct virtual pivot points, with the latter building a two-level tree in tiled batches to support multi-threading. The SkyAlign [2] algorithm also uses medians and quartiles, but defined globally (rather than recursively) to create more predictable tree traversals that minimise branch divergence.

4. PARALLEL SKYCUBE TEMPLATES

This section describes our template methodology (Section 4.1) and then presents three template algorithms that each expose parallelism in a unique way (Sections 4.2-4.3).

4.1 Introduction to template algorithms

Template methodology Shared-memory *data parallelism* distributes a dataset horizontally across cores; then each core efficiently computes a function over every datum in its subset, perhaps using shared data structures to reduce computations or memory loads. As the number of cores scales up, each core processes a smaller subset, thereby finishing sooner. Shared-memory *task parallelism* distributes a parallel task across cores. As the number of cores scales up, so too does the number of concurrent tasks. We aim to exploit all available co-processors in the system—to fully utilize the data and task parallelism that can be exposed by a modern heterogeneous machine. The principal challenge of this heterogeneous parallelism is the apparent paradox in constructing an algorithm that is sufficiently general to work on both GPUs and CPUs yet exploits specific architectural considerations that are vital to maximising performance.

Our objective is to use only the general parallel strategy to define static, read-only, shared data structures that are communicated across sockets and PCIe connections (i.e., between devices or NUMA nodes), whereas state (i.e., thread-local data structures) depend specifically on architecture. Similarly, we aspire that architecture influences the control flow within the parallel work task, but the parallel strategy defines the overall control flow of the algorithm in an architecture-oblivious manner. We define our template algorithms to precisely and procedurally describe the shared structures and the overall control flow, but declaratively describe the work to be parallelised.[3] A declarative *hook* defines what function should be applied to which data, but not how that function should be implemented. The template is therefore *abstract* until the sub-algorithms are defined (i.e.,

[3]Our templates resemble the *Template Method* design pattern [14], except that our templates remain abstract and we specialise towards architectures rather than subclasses.

Algorithm 1 Single-thread-single-cuboid (STSC)

```
Input: A set of points $\mathcal{P}$
Output: The skycube of $\mathcal{P}$ as a lattice, $\mathcal{L}$
1: $\mathcal{L}, \mathcal{L}^+ \leftarrow$ empty lattices (vector of vectors)
2: $\mathcal{L}[2^d-1] \leftarrow \mathcal{S}_{2^d-1}(P)$ and
   $\mathcal{L}^+[2^d-1] \leftarrow \mathcal{S}^+_{2^d-1}(P) \setminus \mathcal{S}_{2^d-1}(P)$ in parallel
3: for lattice level $l = d-1, \ldots, 1$ do
4:     for all subspaces $\delta$, with $|\delta| = l$ in parallel do
5:         parent $\leftarrow \arg\min_{\delta':|\delta'|=l+1} |\mathcal{L}[\delta']| + |\mathcal{L}^+[\delta']|$
6:         $\mathcal{L}[\delta] \leftarrow \mathcal{S}_\delta(\mathcal{L}[\text{parent}] \bigcup \mathcal{L}^+[\text{parent}])$ and
           $\mathcal{L}^+[\delta] \leftarrow \mathcal{S}^+_\delta(\mathcal{L}[\text{parent}] \bigcup \mathcal{L}^+[\text{parent}]) \setminus$
                   $\mathcal{S}_\delta(\mathcal{L}[\text{parent}] \bigcup \mathcal{L}[\text{parent}])$
7: return $\mathcal{L}$
```

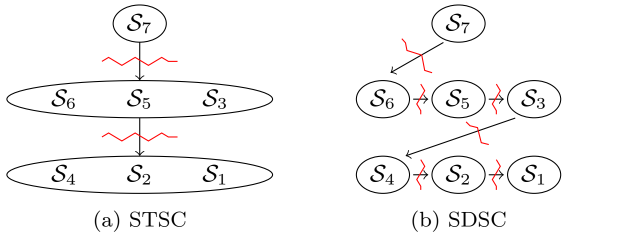

$\mathcal{S}_7$
$\mathcal{S}_6$ $\mathcal{S}_5$ $\mathcal{S}_3$
$\mathcal{S}_4$ $\mathcal{S}_2$ $\mathcal{S}_1$
(a) STSC
$\mathcal{S}_7$
$\mathcal{S}_6$ $\mathcal{S}_5$ $\mathcal{S}_3$
$\mathcal{S}_4$ $\mathcal{S}_2$ $\mathcal{S}_1$
(b) SDSC

Figure 2: Control flow of the lattice-traversal-based templates. Zigzags represent synchronisation points and circles represent the concurrent tasks. STSC (a) computes concurrent single-threaded cuboids; SDSC (b) serially computes each cuboid with a parallel algorithm on a single device.

specialised) for (a) concrete architecture(s), which specifies the thread-local data structures and control flow. In this way, we produce a shared-memory algorithm that uses multiple architectures, but for which the majority of processing time is spent in architecture-specific *specialisations*.

Whereas general, portable algorithms may lack the architecture awareness necessary for performance and architecture-specific algorithms may interface poorly with each other in a heterogeneous environment, we design directly for the heterogeneity without forgoing architecture awareness.

Skycube template overview We present three varied approaches to parallel skycube construction. The first two (Section 4.2) define *task-parallel* hooks based on the insights of the lattice-traversal-based approaches that characterise most existing skycube literature [22, 24, 30, 34, 36, 41]. The third (Section 4.3) adopts a novel paradigm: The algorithm iterates data points rather than subspaces, as in [19], but computes for each point p the bitmask $\mathcal{B}_{p \in \mathcal{S}}$. This produces *data-parallel* hooks that expose significantly more parallelism at the expense of more thread-local state (although not the prohibitive amount required in [19]).

4.2 Lattice-traversal-based templates

The state-of-the-art (sequential) approach in literature [22, 24, 30, 41] constructs skycubes by iterating the lattice top-down, i.e., in the order $\langle \mathcal{S}_7, \mathcal{S}_6, \mathcal{S}_5, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_2, \mathcal{S}_1 \rangle$ in Figure 1a. Compared to traversing the subspaces bottom-up, which requires scanning the full dataset $2^d - 1$ times, the extended skylines of immediate superspaces can be used as a reduced input set for each cuboid computation; e.g., $\mathcal{S}_4$ can be produced from just the three points in $\mathcal{S}^+_6$ or $\mathcal{S}^+_5$. Moreover, data structures that are built to produce one cuboid, say $\mathcal{S}_6$, can be re-used in subsequent cuboids, say $\mathcal{S}_4$ or $\mathcal{S}_2$.

We expose task parallelism in the top-down lattice traversal: One can compute multiple cuboids concurrently or one can use a parallel algorithm for each individual cuboid computation. The former we call *single-thread-single-cuboid* (Section 4.2.1) since each cuboid is computed single-threadedly; the latter we call *single-device-single-cuboid* (Section 4.2.2).

4.2.1 Single-thread-single-cuboid (STSC)

Parallel strategy Figure 2a illustrates the control flow of the top-down lattice traversal template, STSC, and Algorithm 1 outlines the pseudocode. The general idea is to exploit the natural independence among cuboids on the same level of the lattice. Each cuboid computation is an atomic parallel task, and an entire level of the lattice (indicated by the circles and Line 3) is launched concurrently. For each cuboid δ, STSC computes both the extended skyline, $\mathcal{L}[\delta]$, and the set of points needed to construct the skyline of δ, $\mathcal{L}^+[\delta]$ (Lines 2 and 6). Synchronisation occurs after each level $|\delta|$ to ensure that the minimum-sized extended skylines (Line 5) are ready to use as input for level $|\delta| - 1$.

Hook Task parallelisation comes on Line 4, where each thread is assigned an independent cuboid δ and should then compute $\mathcal{S}_\delta$ and $\mathcal{S}^+_\delta$ (Line 6) using the globally shared set of now-read-only extended skylines for level $|\delta|+1$. Specialising the hook for some architecture x requires merely to use a well-chosen single-threaded skyline algorithm for x.

Discussion STSC is our most coarse-grained skycube template. The parallel tasks are long (an entire cuboid computation) and synchronisation is infrequent (once per lattice layer). Whereas parallel algorithms typically introduce some overhead (thread management or extra computations to improve throughput), specialisations of STSC can avoid introducing that overhead by potentially using a sequential skyline algorithm for each cuboid. Also, STSC is our only template that clearly still applies in settings where no parallel algorithm yet exists (e.g., metric space [7]).

On the other hand, the number of concurrent parallel tasks is limited by the width of the current lattice layer, which is troublesome on low-dimensional datasets (where the maximum lattice width is small) or highly parallel architectures (where hundreds of tasks are necessary to avoid starving the device's resources). In fact, on some highly parallel architectures, such as the GPU, there is no notion of a single-threaded algorithm; so, this template cannot be specialised for that device nor achieve cross-device parallelism. Furthermore, STSC does not use shared memory meaningfully; so, the threads are strictly in contention for resources. As each task accesses a different (although potentially overlapping) part of the dataset, the shared L3 cache becomes a scarce resource and threads eject each other's cache lines.

4.2.2 Single-device-single-cuboid (SDSC)

Parallel strategy Figure 2b illustrates the control flow of SDSC in relation to that of STSC and Algorithm 2 outlines its pseudocode. The general idea of this template is to exploit the existing research on parallel skyline algorithms. The differences from STSC are minor but of conceptual significance: One again conducts a top-down lattice traversal, but each cuboid δ is assigned to an entire device where $\mathcal{S}_\delta$ and $\mathcal{S}^+_\delta$ are computed with a parallel algorithm. If k devices

Algorithm 2 Single-device-single-cuboid (SDSC)

Input: A set of points $\mathcal{P}$
Output: The skycube of $\mathcal{P}$ as a lattice, $\mathcal{L}$

```
1: $\mathcal{L}, \mathcal{L}^+ \leftarrow$ empty lattices (vector of vectors)
2: $\mathcal{L}[2^d - 1] \leftarrow \mathcal{S}_{2^d-1}(P)$ *and*
   $\mathcal{L}^+[2^d - 1] \leftarrow \mathcal{S}^+_{2^d-1}(P) \setminus \mathcal{S}_{2^d-1}(P)$ **in parallel**
3: **for** lattice level $l = d-1, \ldots, 1$ **do**
4:     **for all** $\delta$, with $|\delta| = l$ **on parallel devices do**
5:         parent $\leftarrow \arg\min_{\delta':|\delta'|=l+1} |\mathcal{L}[\delta']| + |\mathcal{L}^+[\delta']|$
6:         $\mathcal{L}[\delta] \leftarrow \mathcal{S}_\delta(\mathcal{L}[\text{parent}] \bigcup \mathcal{L}^+[\text{parent}])$ *and*
           $\mathcal{L}^+[\delta] \leftarrow \mathcal{S}^+_\delta(\mathcal{L}[\text{parent}] \bigcup \mathcal{L}^+[\text{parent}]) \setminus$
                   $\mathcal{S}_\delta(\mathcal{L}[\text{parent}] \bigcup \mathcal{L}[\text{parent}])$ **in parallel**
7: **return** $\mathcal{L}$
```

are used, then k cuboids are concurrently computed, but only on the same layer of the lattice (as in STSC). Thus, an individual specialisation can capitalise on the data parallelism within each cuboid computation, while the general template exposes cross-device task parallelism.

Hook Although we retain some parallelism on Line 4 by parallelising across devices, the hook occurs on Line 6, where we exploit the well-known data parallelism in skyline computation [1, 2, 8, 10]. To specialise the hook for architecture x, one can use the state-of-the-art parallel algorithm for architecture x—e.g., [8] on multicore, [2] on GPUs, or [38] on FPGAs—again using the globally-shared now-read-only extended skylines from the previous level as a smaller input.

Discussion The strength of SDSC is that the threads cooperate on each cuboid computation. If the underlying parallel skyline algorithm is resource-friendly, then SDSC will also be resource friendly, since it only executes one cuboid computation at a time on each device. Meanwhile, there is minimal complexity introduced by cross-device task parallelism, because each device processes an independent atomic task (i.e., cuboid). It is relatively easy to incorporate new parallel skyline algorithms, algorithms that target particular settings (e.g., low cardinality domains [12, 13]), or specialisations on other architectures for which parallel skyline algorithms exist (e.g., FPGAs [38]).

On the other hand, SDSC scales poorly with d, since each new dimension doubles the number of times the parallel skyline algorithm is run. Although the algorithm exposes data parallelism (in computing a cuboid), that data parallelism is very limited for cuboids near the bottom of the lattice (where resources are starved by a dearth of work to parallelise). The algorithm incurs $2^d - 2$ synchronisation points.

4.3 Point-bitmask-based template

By shifting away from lattice traversals, we expose much more data parallelism in skycube computation. In particular, computing $\mathcal{B}_{p \notin \mathcal{S}}$ for each $p \in \mathcal{S}^+_\delta(P), \delta = 2^d - 1$ (i.e., δ is the original, unprojected data space) exposes $|\mathcal{S}^+_\delta(P)|$ data-parallel tasks. However, some ingenuity is required in order to calculate $\mathcal{B}_{p \in \mathcal{S}}$ efficiently. Our third template, *multiple-device-multiple-cuboid*, makes this shift and adapts the globally-shared quad tree techniques used in state-of-the-art multicore [8] and GPU [2] skyline algorithms, thereby obtaining *cross-device data parallelism*.

4.3.1 Multiple-device-multiple-cuboid (MDMC)

Algorithm 3 Multiple-device-multiple-cuboid (MDMC)

Input: A set of points $\mathcal{P}$
Output: Skycube of $\mathcal{P}$ as a HashCube, $\mathcal{H}$

```
 1: $\mathcal{H} \leftarrow$ empty HashCube (array of maps of vectors)
 2: Build quad tree $\mathcal{T}$ over $\mathcal{S}^+(P)$ **in parallel**
 3: **for all** points $p \in \mathcal{S}^+(P)$ **in parallel do**
 4:     $\mathcal{B}_{\notin \mathcal{S}} \leftarrow$ cuboid *non-membership*, all $2^d - 1$ bits unset
 5:     $\mathcal{B}_{\notin \mathcal{S}^+} \leftarrow$ *strict* non-membership, all $2^d - 1$ bits unset
 6:     **for** selected nodes $n$ of $\mathcal{T}$ **do**
 7:         Set each bit $\delta$ of $\mathcal{B}_{\notin \mathcal{S}}, \mathcal{B}_{\notin \mathcal{S}^+}$ if $n$ implies $q \prec\!\!\prec_\delta p$
 8:     **for all** unset bits $\delta$ of $\mathcal{B}_{\notin \mathcal{S}}$, relevant leaves $n$ of $\mathcal{T}$ **do**
 9:         $q \leftarrow$ point corresponding to $n$
10:         Compute $\mathcal{B}_{q<p}$, $\mathcal{B}_{q=p}$
11:         **if** $q \prec_\delta p$ *and* $\mathcal{B}_{q<p} | \mathcal{B}_{q=p}$ is not set in $\mathcal{B}_{\notin \mathcal{S}^+}$ **then**
12:             Set bits of all submasks of $\mathcal{B}_{q<p}$ in $\mathcal{B}_{\notin \mathcal{S}^+}$ *and*
                of $\mathcal{B}_{q<p} | \mathcal{B}_{q=p}$ in $\mathcal{B}_{\notin \mathcal{S}}$
13:     Insert $\mathcal{B}_{\notin \mathcal{S}}$ into $\mathcal{H}$
```

Parallel strategy Algorithm 3 outlines MDMC in pseudocode and Figure 3 illustrates the idea (although it will be presented more extensively for the specialisations). Each data-parallel task t_i processes a unique point $p \in \mathcal{S}^+(P)$, determining in which subspaces it is dominated (i.e., $\mathcal{B}_{p \notin \mathcal{S}(P)}$) and strictly dominated (i.e., $\mathcal{B}_{p \notin \mathcal{S}^+(P)}$). The overall strategy is a *filter and refine* algorithm over the subspaces: a cheap routine is run first (the filter phase, Lines 6–7) to rapidly rule p out of many subspace skylines; a more expensive routine is run next (the refine phase, Lines 8–12) to verify which of the remaining subspace skylines really contain p. The two phases are the two hooks for the MDMC template.

Both phases are accelerated by a read-only, cross-device-shared quad tree that minimises memory reads. The tree (shown at the top of Figure 3) is computed as the first step (Line 2), while calculating the extended skyline, $\mathcal{S}^+(P)$. The filter phase uses only the tree structure to prune subspaces, whereas the refine phase uses the tree and DTs with the actual data points. How and how effectively the phases work is left to the specialisations, but the filter phase's avoidance of actual data points should drastically reduce the load on the memory subsystem while its pruning should reduce the more expensive processing in the refine phase.

This template computes $\mathcal{B}_{p \notin \mathcal{S}^+(P)}$ to improve the asymptotic complexity from $\mathcal{O}(n(2^d n))$ to $\mathcal{O}(n(2^d + n))$. Line 10 produces a d-dimensional bitmask $\mathcal{B}_{p \leq q}$. There are only 2^d unique values for this bitmask, and, generally, $2^d \ll |\mathcal{S}^+(P)|$. So, if Line 10 produces a duplicate bitmask, then q conveys no new information about subspaces in which p is dominated—we skip the redundant work on Line 11.

We can be even more aggressive. We can set in $\mathcal{B}_{\notin \mathcal{S}^+(P)}$ every sub-mask of $\mathcal{B}_{p<q}$, because strict dominance propagates to all subspaces. Even considering non-strict dominance, if the $\mathcal{B}_{p \leq q}$'th bit is set in $\mathcal{B}_{\notin \mathcal{S}^+(P)}$, we know dominance has already been asserted for all submasks of $\mathcal{B}_{p \leq q}$. We just move on to the next point q' without processing $\mathcal{B}_{p \leq q}$ further. Lines 7 and 12 maintain $\mathcal{B}_{\notin \mathcal{S}^+(P)}$ by setting bits whenever p is determined to not be in a subspace extended skyline. Line 11 verifies that $\mathcal{B}_{p \leq q}$ is unprocessed before executing the expensive $\mathcal{O}(2^d)$ Line 12.

The quad tree must be defined with global pivots, as in [2]. The dynamic, recursive quad trees [8, 21, 23, 42, 43] require DTs at inner nodes when they are traversed by a point p,

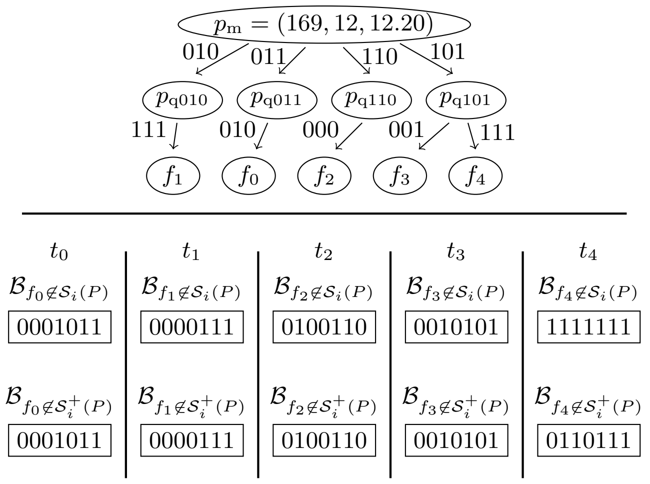

$p_\mathrm{m} = (169, 12, 12.20)$
010
011
110
101
p_q010
p_q011
p_q110
p_q101
111
010
000
001
111
f_1
f_0
f_2
f_3
f_4
t_0
t_1
t_2
t_3
t_4
$\mathcal{B}_{f_0 \notin \mathcal{S}_i(P)}$
0001011
$\mathcal{B}_{f_1 \notin \mathcal{S}_i(P)}$
0000111
$\mathcal{B}_{f_2 \notin \mathcal{S}_i(P)}$
0100110
$\mathcal{B}_{f_3 \notin \mathcal{S}_i(P)}$
0010101
$\mathcal{B}_{f_4 \notin \mathcal{S}_i(P)}$
1111111
$\mathcal{B}_{f_0 \notin \mathcal{S}_i^+(P)}$
0001011
$\mathcal{B}_{f_1 \notin \mathcal{S}_i^+(P)}$
0000111
$\mathcal{B}_{f_2 \notin \mathcal{S}_i^+(P)}$
0100110
$\mathcal{B}_{f_3 \notin \mathcal{S}_i^+(P)}$
0010101
$\mathcal{B}_{f_4 \notin \mathcal{S}_i^+(P)}$
0110111

Figure 3: MDMC data structures constructed from Table 1: a globally shared read-only quad-tree (top) and thread-local solution masks recording skyline and extended skyline membership per subspace (bottom). This quad tree uses the medians and quartiles of the data set to partition space as in [2]. Note that $\mathcal{B}_{f_4 \notin \mathcal{S}_0(P)} \neq \mathcal{B}_{f_4 \notin \mathcal{S}_o^+(P)}$ because $f_4[0] = f_3[0]$.

which incurs latencies in the tree traversal, even in our refine phase. If the pivots are defined globally, the tree can be traversed just by knowing the path to the leaf containing p [2]. To adapt the tree for skycubes, we add a third level (octiles). A deeper tree creates a traversal overhead that is nearly as expensive as conducting DTs. However a shallower tree, such as the two-level tree in SkyAlign [2], is ineffective in low-dimensional subspaces, because most bits/dimensions are not used. Adding a third layer (relative to [2]) doubles the information carried in the tree per dimension. Physically, we maintain all quartile and octile masks in flat arrays of length $\mathcal{S}^+(P)$, sorted in the order of the leaves (i.e., a reverse lookup from point to tree node), to coalesce reads (not as shown in the semantic Figure 3). Only the top, median level of the tree, which is insufficient to fully partition the dataset, is structured as an array of pointers to child nodes.

Hooks To specialise MDMC, the filter and refine phases must be defined for the given architecture. The filter phase should prune without using DTs and the refine phase should produce the exact subspaces in which point p is dominated. Both phases should minimise memory reads with the quad tree. Our specialisations described later should better illuminate for the reader the purpose of these hooks.

Discussion MDMC exposes significantly more parallelism, which can be distributed across devices, by defining a parallel task per data point rather than per cuboid. Since every task is independent, there is no need for synchronisation. Moreover, as MDMC can asynchronously insert new solution bitmasks into a Hashcube, the memory overhead can be an order of magnitude smaller than the other templates.

On the other hand, MDMC consumes a lot of local state for each work task on account of its two bitmasks of length $2^d - 1$. Specialising the template for a new architecture requires significant innovation, since the filter and refine phases must be newly defined and the challenge of traversing a spatial partitioning tree in parallel may need to be solved.

5. SPECIALISATIONS FOR MULTICORE

This section describes how we capture the considerations in Section 2.3 to specialise the templates for multicore CPUs.

5.1 CPU lattice-based specialisations

STSC is the most flexible template to specialise for CPUs because any skyline algorithm can be hooked into Line 4 of Algorithm 1. However, STSC does not compute subspace skylines in isolation; to the best of our knowledge, no research has investigated how shared-memory skyline algorithms scale as the number of concurrent queries increases.

The state-of-the-art sequential algorithm, BSkyTree [21, 23], and the similarly designed point-based partitioning methods OSP [43] and VMPSP [42], use a variable-depth—and therefore pointer-based—recursive quad tree to replace DTs with MTs. Consequently, the tree is not very compact (i.e., consumes more cache) and chases a lot of pointers, putting threads in greater contention for the limited cache resources. In contrast, the Hybrid [8] multicore algorithm, which we hook in instead, builds a compact, fixed two-level, array-based tree and uses modest intra-cycle parallelism [25] by representing the full path to each leaf in a single 32-bit word.

For SDSC, the choice of multicore skyline algorithm is more straight-forward, because the algorithm will be run in isolation on the CPU. APSkyline [26], Hybrid [8], and Scalagon [13] have never been formally compared to each other; however, APSkyline has not been shown to scale beyond four dimensions and Scalagon is designed for low-cardinality domains. Thus, we hook in for SDSC as well the more general Hybrid algorithm, providing the added advantage of a very fair comparison of lattice-based templates.

Note that some minor adaptation is necessary to hook Hybrid into STSC and SDSC to also produce $\mathcal{S}_\delta^+(P)$ and to conduct subspace DTs. Producing $\mathcal{S}_\delta^+(P)$ follows immediately from Definition 2 by replacing DTs with strict dominance tests and only using the relevant dimensions for the quad-tree partitioning. However, contrary to intuition, conducting MTs and DTs in subspaces is not cheaper than in the full space, because mask tests use constant-width 32-bit words (c.f., Section 2.2) and DTs should be SIMDized [9]. There is no advantage in projecting a d-bit mask when $d \leq 32$, because the bitwise operations are the same. Instead, we add the projection to the mask test (& δ in Equation 1), which increases instruction-level parallelism. On-the-fly projection of the DT requires rearranging specific dimensions from separate cache lines to align them for SIMD registers. We instead conduct the DT on the already-aligned d dimensions and then project the resultant bitmask afterwards.

5.2 CPU point-based specialisation

Our CPU specialisation of MDMC exploits the large L2 cache and tries to simplify the work of the branch predictor.

Filter hook The filter on Lines 6–7 of Algorithm 3 should very quickly determine a large percentage of subspaces δ in which point p is dominated (i.e., set many bits of $\mathcal{B}_{\notin \mathcal{S}_\delta(P)}$). For this, we exploit the information already in the path labels of the tree, with which we can assert transitive relationships without ever loading data points from memory. The top two levels of the tree can be entirely L2-resident, as they contain $\leq 4^d$ partitions; for $d \leq 8$, that requires ≤ 256 KB, which fits in L2 cache on Intel Ivy Bridge and Haswell architectures. For $d > 8$ or smaller caches, the top-two levels still typically fit in L2 cache, because most partitions are empty.

We iterate the top two layers in a predictable depth-first order, examining path labels. Consider f_4 in Figure 3 as an example, with a median mask $\mathcal{B}_{f_4<p_m} = 5$ and a quartile mask $\mathcal{B}_{f_4<p_{q101}} = 7$. For each other median mask in the tree, such as $\mathcal{B}_{f_0<p_m} = 3$, we construct a mask of dimensions in which that bitmask transitively implies strict dominance: $\sim\mathcal{B}_{f_1<p_m}$ & $\mathcal{B}_{f_4<p_m} = 4$. In the resultant subspace, $\delta = 4$, and all its subspaces, f_4 is clearly, strictly dominated.

The same logic applies to the quartile masks at the next level of the tree, except that we exclude bits where the median masks were unequal (i.e., the points are compared to different quartiles), producing $\delta' = 1$ in the example. We can conclude that f_4 is dominated in all subspaces of $\delta \mid \delta'$, having loaded nothing but the tree's path labels and branched on no conditions except the depth-first tree traversal. The same logic can be applied to the third level of the tree, but competing threads would start evicting cache lines.

Refine hook The refine phase (Lines 8–12 of Algorithm 3) must explicitly verify whether p is dominated in any of the subspaces that could not be filtered in the previous phase. We iterate the remaining subspaces in a top-down, breadth-first manner to exploit the converse of the insight in BUS [41]: if p is dominated by q in subspace δ, then it is dominated by (or equal to) q in all subspaces of δ.

For each subspace δ, we iterate the quad tree, using Equation 1 to prune subtrees. Whenever we reach a leaf with point q, we conduct a vectorised DT to produce the masks $\mathcal{B}_{p<q}$ and $\mathcal{B}_{p=q}$. If $\mathcal{B}_{p<q} \mid \mathcal{B}_{p=q}$ is a mask that has not yet been seen for point p, then we iterate through each unverified subspace δ and set p as dominated in any δ for which $(\mathcal{B}_{p<q} \mid \mathcal{B}_{p=q})$ & $\delta = \delta$ and $\mathcal{B}_{p=q}$ & $\delta \neq \delta$.

6. SPECIALISATIONS FOR THE GPU

This section describes how we capture the considerations in Section 2.3 to specialise the templates for the GPU.

6.1 GPU lattice-based specialisations

Selecting algorithms to hook into the lattice-based templates is straight-forward on the GPU. STSC cannot be specialised (a clear weakness of the template), because there are no single-threaded GPU algorithms. For SDSC, there are only three GPU skyline algorithms from which to choose: GNL [10], GGS [1], and SkyAlign [2]. The latter has been shown to be orders of magnitude faster on most workloads [2].

As with the CPU in Section 5.1, one must adapt the skyline algorithm to calculate $\mathcal{S}^+_\delta(P)$ and to conduct subspace MTs and DTs. $\mathcal{S}^+_\delta(P)$ again follows from Definition 2 by replacing DTs with strict dominance and partitioning data based on only the relevant dimensions. Also, mask tests are handled on the GPU exactly like on the CPU, by adding the projection to Equation 1. But, DTs are handled differently.

The GPU does not have SIMD registers for data-parallel DTs. Instead, the DTs in SkyAlign are unrolled to maximise instruction-level parallelism. Therefore, we project data points on-the-fly into subspaces to make DTs cheaper.

The projections provide a very minor acceleration with the row-major data layout, because the latency of the memory loads is the dominant cost of a DT and the projections do not substantially reduce the number of (anyway contiguous) cache lines read. Note that a column-major data layout would be worse, since we seldom access consecutive data points: a point is only accessed when used for a DT, which in turn only happens if the leaf of the quad tree at which the point is located cannot be pruned during the tree traversal.

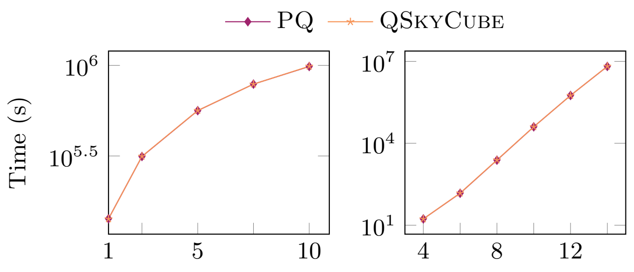


Figure 4: `QSkycube` and our parallelisation, on independent data, single-threaded: relative to $n \times 10^5$ (left) and d (right).

6.2 GPU point-based specialisation

Limited thread state drives our GPU specialisation of the MDMC template. The task-local variables, $\mathcal{B}_{\notin\mathcal{S}(P)}$, $\mathcal{B}_{\notin\mathcal{S}^+(P)}$, at a length of $2^d - 1$, consume a lot of state. At $d = 16$, each consumes 8 KB, but shared memory is only 96 KB per 2048 concurrent threads on our GPU card. Even at low dimensionality, there is not sufficient shared memory for each thread to keep a local copy of the task-local variables. Yet, because of the frequent reads and writes, the latencies are too high to keep them in global memory. Therefore, in our specialisation, all threads within a thread block cooperate on processing a single data point p and we adapt the thread block size to the amount of shared memory required, i.e., 2^d bits. (As d increases, more state is required and thus the number of threads processing each p is increased.) Consequently, we must expose substantial data-level parallelism within the already data-parallel task of processing p.

Filter hook Iterating a quad tree to skip work on behalf of a single point, even when defined with static, global octiles, is non-trivial with step-locked threads. However, we still wish to exploit the information already in the path labels of the quad tree to avoid any expensive loads of data points.

We leverage the fact that the tree has a constant height by traversing the leaves, not the tree, in strides equal to the number of threads t processing point p; i.e., thread t_i will process leaves $\langle i, t+i, 2t+i, \ldots\rangle$. At a given leaf l, we construct a composite bitmask comparing the entire path to the leaf containing p and the entire path to l. If the composite bitmask is unseen, we update $\mathcal{B}_{p\notin\mathcal{S}(P)}$ and $\mathcal{B}_{p\notin\mathcal{S}^+(P)}$.

This routine is effectively a massively data-parallel sequential scan of $\mathcal{S}^+(P)$ that uses the tree structure as a predicate to filter out subspaces. The only branch divergence comes after constructing the composite length-d bitmask, and this can occur at most $2^d \ll |\mathcal{S}^+(P)|$ times. By reading the entire tree, the GPU filter is much stronger than the CPU filter; by scanning the tree sequentially from left to right, the memory loads are all coalesced. However, the GPU filter does far more processing than the CPU filter.

Refine hook For the refine step of MTMC on the GPU, we employ the same strategy as on the filter step. We traverse all leaves in a strided, data-parallel fashion (i.e., do a second scan) and rederive combined masks as before, but this time generating both $\mathcal{B}_{q<p}$ and $\mathcal{B}_{q\leq p}$. If $\mathcal{B}_{q\leq p} \neq 0$ and the $\mathcal{B}_{q<p} \mid \mathcal{B}_{q\leq p}$'th bit is not set in $\mathcal{B}_{p\notin\mathcal{S}^+_i(P)}$, then the thread

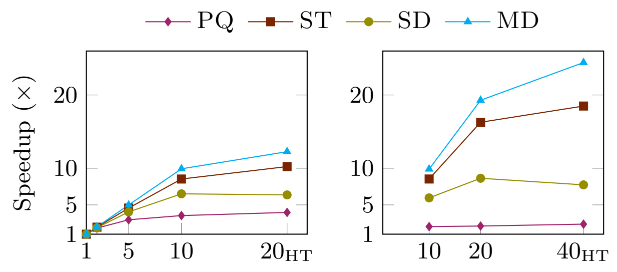


Figure 5: Speedup relative to single-threaded execution as # threads vary on one (left) or two (right) socket(s) (default).

elects to do a DT between q and p and update $\mathcal{B}_{p \notin \mathcal{S}(P)}$ and $\mathcal{B}_{p \notin \mathcal{S}^+(P)}$ accordingly. The warp performs a warp vote to see if *any* thread needs a DT, and all threads conduct DTs if the vote returns `true`. Because the majority of subspaces have already been pruned by the filter step (by the Pigeon Hole Principle), the warp vote is usually `false`. When the threads reach the end of the tree, $\mathcal{B}_{p \notin \mathcal{S}_i(P)}$ is asynchronously copied to the CPU and inserted into a HashCube.

7. EXPERIMENTAL EVALUATION

Here we evaluate the template specialisations relative to a baseline parallelisation of the sequential state-of-the-art [24]. Additional experiments appear in Appendix A.

7.1 Environment and setup

Hardware CPU specialisations are studied on a dual-socket, ten-core Intel Xeon E5-2687W v3 3.10 GHz CPU with hyper-threading enabled and 128GB of DDR4 memory, running OpenSuse Leap (kernel version 4.1.12). To improve the performance of the transaction lookaside buffer (TLB), we have transparent huge pages enabled. GPU specialisations are evaluated on an NVidia GTX 980 card. For cross-device experiments, we add an additional 980 and a GTX Titan to a separate socket of the CPU via PCIe3.

Implementations[4] Template code is written in `C++`, using *boost threads* to distribute tasks to devices. The CPU specialisations in `C++` use the OpenMP API (4.0) and *boost threads* for multi-threading and are compiled with `g++` (5.2.1) with the `-O3` optimisation flag. The GPU specialisations are written in CUDA to comply with compute capability 3.5, and all code is compiled using `nvcc` (8.0.44). We pin threads to cores with `numactl`, read hardware counters with `PAPI`, and time the algorithms—from after the datasets are loaded into CPU memory until CPU memory holds the entire skycube, including all PCIe transfers—in the software.

Datasets Following literature, we use the benchmark data generator [5] to produce synthetic workloads of anticorrelated (A), independent (I), and correlated (C) dimensions. We vary the number of points $n \in \{1, 2.5, 5, 7.5, 10\} \times 10^5$ and the number of dimensions $d \in \{4, 6, \ldots, 16\}$, with defaults of (I), $n = 500\,000$ and $d = 12$, as in [22,24], producing datasets that consume up to 100 MB in raw text. As the synthetic datasets importantly capture various levels of correlation between the attributes, but not data skew, we also conduct experiments with real data (Appendix A.1).

[4]http://github.com/sean-chester/skycube-templates

Establishing a baseline The existing parallel skycube algorithm [36] is coupled to the Anthill framework, which is not designed for a single node. So, as a baseline, we implement a parallel version of QSkycube [22,24] with parallel pragmas on the loop iterating cuboids (similar to STSC). We share resultant quad trees freely among child subspaces, but elect not to "merge results from multiple parents" to reduce the resultant traffic across sockets. Figure 4 compares the single-threaded performance and workload scalability of this baseline, called PQSkycube, to QSkycube code that we obtained from the authors [24]. The point to be made here is simply that PQSkycube does not introduce overhead to QSkycube. To the contrary, it obtains a minor speed-up by freeing memory that is no longer used (e.g., quad trees two levels above the currently processed lattice layer).

7.2 Results and discussion

Parallel scalability We first study how the CPU specialisations scale with threads. Figure 5 shows the speedup on the default workload for threads pinned to cores on 1 (left) or 2 (right) sockets. The right-most data point in each plot uses hyper-threading (HT). The common point, $t = 10$, allows us to analyse the effect of using more than one socket.

Both STSC (ST) and MDMC (MD) scale very well with physical cores and MD continues to scale well with HT. Neither algorithm suffers significantly from NUMA effects. SDSC (SD) is not drastically affected by NUMA, but is less scalable in general and degrades with HT, trends that are consistent with the underlying skyline algorithm [8]. PQSkycube (PQ) improves slowly with additional cores and HT, but obtains nearly no speed-up compared to single-threaded computation as soon as a second socket is introduced. This is consistent with our intuition that the extra sharing across sockets introduces prohibitive NUMA latencies and that threads compete for cache to store their larger pointer-based trees. Later, we investigate this directly.

Workload scalability (CPU) We next analyse how the CPU algorithms scale with datasets. We use 40 threads for MD and ST, 20 HT for PQ, and 20 threads on two sockets for SD so that each algorithm is running under its optimal thread configuration. (The results in Figure 5 hold generally across these workloads). Figure 6 shows the execution times (in ms), starting from just after the dataset is loaded into memory and ending once the lattice or HashCube has been fully constructed. The leftmost plots vary n; the rightmost plots vary d; correlation increases from top to bottom.

Across most workloads, MD is the fastest, followed by ST, SD, and then PQ, by significant margins (considering the logarithmic y-axis). PQ is up to two orders of magnitude slower than MD on account of its poor parallel scalability. ST, with better parallel scalability, achieves a several factor improvement over SD (the other lattice-based template).

The algorithms trend consistently with respect to n (left). On (A) and (I) distributions, the templates converge with increasing n, since this creates more parallel tasks for MD to complete, while adding an overhead to the lattice-traversal-based methods that may disappear by the lower levels of the lattice as the extra points are pruned anyway. On (C) data, SD is slower than PQ, although only requires ≈ 2 s for one million $12d$ points. The parallel tasks are small, because the extended skylines are small; so, SD never achieves high utili-

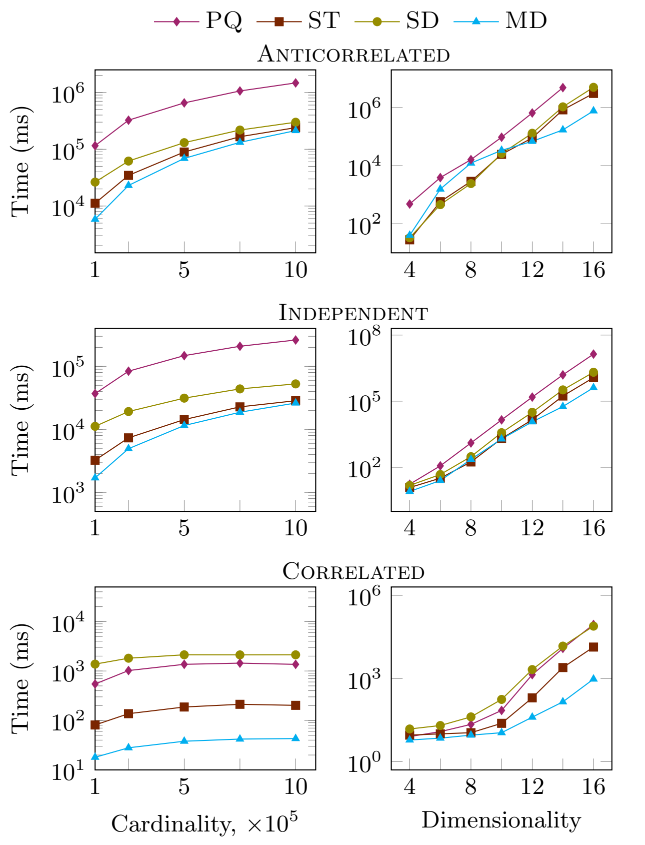


Figure 6: CPU execution times: relative to $n \times 10^5$ (three plots on left), d (three plots on right) and distribution ((A)-top; (I)-middle; (C)-bottom).

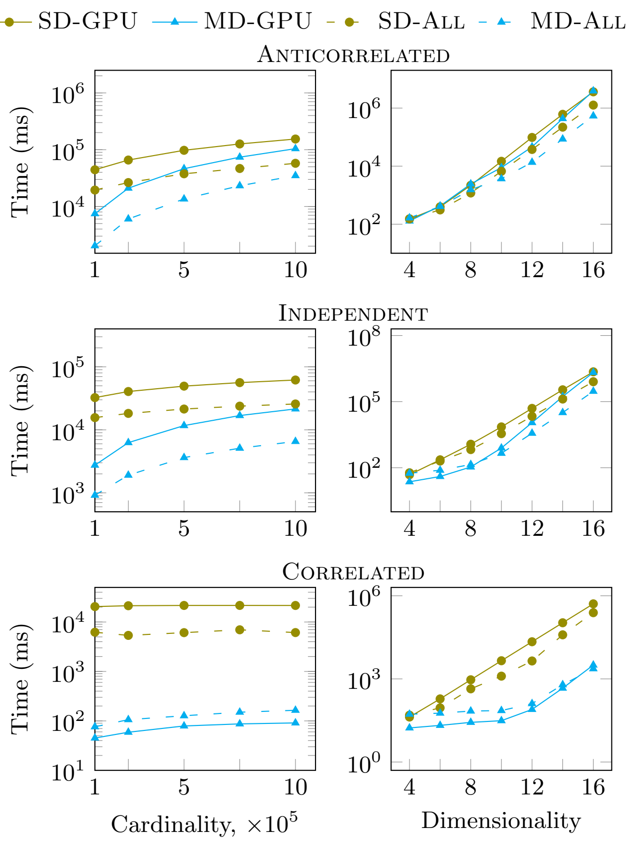


Figure 7: GPU and cross-device (-All) execution times: relative to $n \times 10^5$ (three plots on left), d (three plots on right) and distribution ((A)-top; (I)-middle; (C)-bottom).

sation, whereas PQ (and similarly ST) fully occupy each core concurrently computing the small, independent cuboids.

The trends are interesting with respect to d (right). For (A) and (I), PQ is the slowest. It does not complete $d = 16$ on (A), given the memory overhead of simultaneously maintaining $\binom{16}{8}$ non-compact trees. MD is typically the fastest, except from $d = 6$ to 10. As d increases towards 8, the extended skyline (i.e., input after Line 2 of the algorithm) grows very quickly, especially for (A), generating significantly more parallel tasks to complete. However, the growth in the extended skyline saturates after $d = 8$ and then the partitioning scheme with fixed height trees becomes particularly effective, since there are up to 4^d partitions in the first two levels of the tree (as opposed to the dynamic tree of PQ that degrades to a single layer at higher dimensions). ST and SD have consistent performance relative to each other.

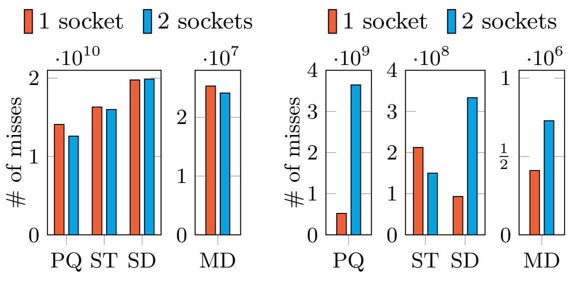
Throughput analysis (CPU) The experiments so far have shown the CPU algorithms have dramatically different parallel scalability and consequently execution times. However, they have not *explained* the variance in parallel scalability. Here we look at throughput metrics, measured with hardware counters, to better understand the earlier results.

First we measure *compute throughput* as cycles per instruction (CPI) in Figure 11 on the default workload, averaged across all threads. High CPI indicates that compute resources idle and is typical of memory-bound computation. The theoretically ideal value on our Intel Haswell machine is 0.25: on average, four instructions are retired per cycle. We show throughput on 10 cores using one socket (red) and split evenly over two sockets (blue). The latter doubles the L3 cache but induces long intersocket latencies (i.e., NUMA effects). To better observe the variance between the templates and the number of sockets, we render PQ with a separately scaled y-axis; otherwise, the large differences between SD and MD would be dwarfed by the scale used for PQ.

We see that the CPI of PQ is nearly doubled by the second socket, in line with the second-socket degradation of PQ in Figure 5. The CPI of the three templates is mostly stable across sockets but not algorithms. The data-parallel (and most efficient) MD template has the best compute throughput by a factor of 50% over the task-parallel SD. STSC also has much better throughput than SDSC, matching Figure 5.

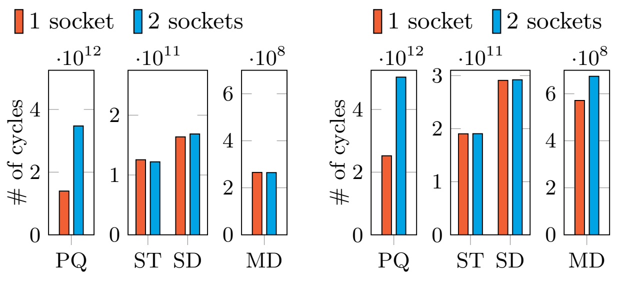
The high CPI and the degradation on two sockets of PQ suggest memory-boundedness. Interestingly, this is not true of PQ at lower thread counts, t; The CPI value increases with t: $\langle 0.92, 0.98, 1.12, 1.29, 1.41, 1.67, 1.83, 1.98, 2.20, 2.46\rangle$ for $t = \langle 1, \ldots, 10\rangle$ on a single socket. In other words, PQ is compute-bound when run sequentially, but becomes increasingly memory-bound as more cores have less L3. (In contrast, CPI is independent of t for MD.)

We argue that the compute throughput results from cache-consciousness. Figure 8 shows the total number of cache misses for the thread-local L2 cache (left) and the socket-local L3 cache (right). The algorithms differ by orders of



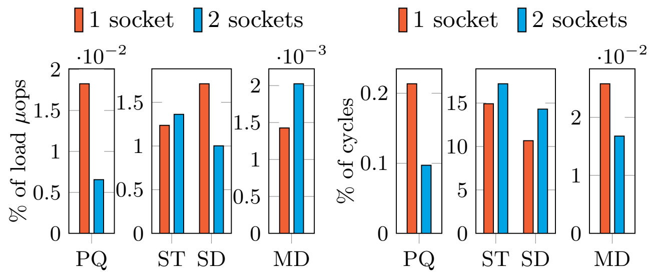
(a) L2 Misses (b) L3 Misses

Figure 8: Cache misses (default workload; 10 cores)



(a) L2 Stalls (b) L3 Stalls

Figure 9: Stalled cycles, load pending (default; 10 cores)



(a) Loads that miss STLB (b) Page walk duration

Figure 10: Data TLB performance (default; 10 cores)

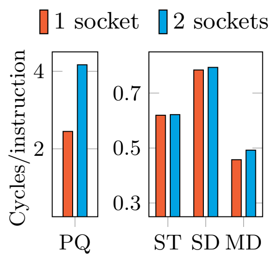


Figure 11: Cycles per instruction (default; 10 cores)

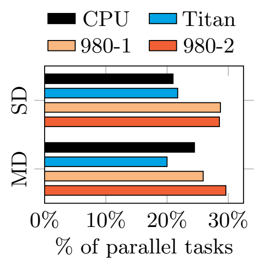


Figure 12: Portion of work done by each device (default)

magnitude, so we again split the algorithms into subplots with different axes to ensure socket comparisons are visible. A multiplier above the axes, e.g., $\cdot 10^{10}$, indicates the scale. Notably, the cache-conscious MD suffers 100-fold fewer L2 cache misses, which fuels the higher compute throughput. PQ, ST, and SD are not appreciably different in terms of L2 cache misses, but PQ has the fewest of the lattice-traversal-based methods, despite its pointer-based tree.

The differences become more pronounced for L3, both between algorithms and between socket counts. PQ suffers most with L3 when sharing it among all cores as well as when its memory is split across two sockets yielding a seven-fold increase in L3 misses. The cache conscious MD again performs the best owing to the small, static quad tree. It retains its order of magnitude cache performance advantage relative to the other algorithms, although the extra socket leads to a 43% increase in L3 misses. SD suffers a 3-fold increase in L3 misses on two sockets, as the tree used by the underlying algorithm must now be kept coherent between the two sockets. For MD, PQ, and SD, the prefetcher is unable to overcome the intersocket latency to populate L3 and so there are more misses with two sockets. However, ST benefits from having double L3 cache in which to work.

Cache misses do not tell the entire story, however, because the CPU can hide latencies, with, e.g., out-of-order execution. Figure 9 shows the number of cycles during which a core was stalled while it had a pending load request at a given level of the cache hierarchy. These are *cycles* during which the core idled because it had not loaded its operands; i.e., the latencies could not be hidden. Again, the algorithms are split into subplots with different scales due to different orders of magnitude. Here, at both L2 and L3, we observe trends that are *very* similar to the CPI numbers shown in Figure 11, suggesting that the differences in computational throughput come down to memory subsystem stalls. We observe that PQ is dramatically affected by NUMA, MD is minorly affected, and that SD and ST are NUMA-tolerant (with the latter benefitting from the extra cache). With MD and ST incurring the fewest L3 cache misses, the latencies can be hidden rather than converted into stalls.

Figure 10 further investigates the memory subsystem performance by analysing the translation lookaside buffer (TLB) for data loads (vs. data stores or instruction loads). The TLB entries map virtual memory addresses to physical memory addresses. Specifically, Figure 10a gives the percentage of load operations (μops) that miss the second-level, shared[5] TLB (STLB), causing a page walk to fetch the TLB entry from memory. Figure 10b gives the percentage of cycles consumed by those page walks (i.e., the *cost* of the TLB misses).

First, we observe that ST and SD incur a much higher percentage of TLB misses than the data-parallel MD template (for which only 0.002 % of TLB lookups miss the second-level cache). They also incur a higher percentage of misses than PQ, but mostly on account of issuing $\approx 4\times$ fewer load μops—the absolute numbers are comparable. This suggests that the cache-consciousness of MD promotes good spatial locality, whereas the DT-based tree traversals in the lattice-based methods make relatively frequent wide-range jumps in memory addresses, even with the compact, array-based tree representation used by Hybrid. The random access to data points for DTs at inner nodes of the recursive tree traversal often require memory addresses that are not TLB-resident.

The TLB misses have a notable effect on the cache performance, since missed entries must be retrieved from the mem-

[5]Shared between data and instruction TLBs, not cores.

ory subsystem. For direct comparison with Figure 9, the absolute number of cycles spent on page walks (one socket) are: $3.6 \cdot 10^9$, $1.3 \cdot 10^{11}$, $1.5 \cdot 10^{11}$, and $1.0 \cdot 10^6$ for PQ, ST, SD, and MD, respectively. Observe that for ST and SD, in particular, these numbers are proportionate to the number of cycles stalled on pending loads. Thus, not only is it that ST and SD spend a significant number of cycles retrieving TLB entries, but also that those TLB misses explain many of the additional stalls, relative to MD, in Figure 9b.

Overall, the cache and TLB metrics demonstrate the difference between the static tree of MD, which is traversed without loading data points, and the recursive tree whose traversal requires DTs. Comparing ST and PQ, in particular, which use the compressed Hybrid array-based tree versus the pointer-based, variable-depth SkyTree, respectively, we see the former maintains high throughput and reasonable cache performance, whereas the latter degrades quickly from unpredictably chasing pointers across sockets. (Thus Hybrid is preferred to BSkyTree in the STSC specialisation.) Still, ST and SD suffer from poor TLB performance compared to the static tree used in MD, incurring many latencies.

Workload scalability (GPU) Figure 7 repeats the workload scalability experiments from Figure 6 for the GPU specialisations (solid lines). To facilitate comparisons with the CPU specialisations, the y-axis scales match Figure 6. Encouragingly, the trends between templates mostly replicate on the GPU. MD outperforms SD: especially for the lower-dimensional cuboids with dramatically reduced input sizes, SD struggles to generate enough parallel tasks to fully utilise the GPU card. MD, on the other hand, always generates $\mathcal{S}^+(P)$ parallel tasks. The performance converges as n increases (except on (C)), favouring the cuboid-by-cuboid approach of SD. However, as d increases, the performance of the templates originally diverges, only to converge for $d \geq 12$. At this point, the state consumed by the local variables becomes a factor and the MD specialisation can support fewer concurrent data points. By $d = 16$, SD has matched MD again for (A) and (I). Given the trend that new generations of GPUs add more state per thread, this convergence may delay til higher d in the future.

Comparing to the CPU specialisations, we see that MD is generally faster on the CPU for (C) or $n \leq 250\,000$, when the number of parallel tasks is lower, but faster on the GPU for (A), where a large extended skyline generates substantial parallel work. Other workloads strike a balance; so, the CPU and GPU are competitive. With respect to d, MD is generally faster on the CPU at low values when there are fewer subspaces to iterate and faster on the GPU at moderate to high d. SD is generally faster on the CPU, owing to its larger cache sizes which facilitate cache-sharing between cuboids and the relative ineffectiveness of the GPU on the small inputs that occur near the bottom of the lattice. Scalability is better on the GPU with respect to n as more parallelism is exposed and comparable with respect to d.

Heterogeneous processing Our template algorithms can scale across heterogeneous co-processors. Consider Figure 7 again, which additionally shows the heterogeneous performance of MD and SD (dashed lines) when run on all 20 physical CPU cores and all three GPUs. Considering the figures on the left, i.e., scalability with respect to n, SD consistently gets nearly a 3× speedup by using all GPUs (of non-uniform model) and the CPU, relative to the single GPU (solid blue line), indicating near-linear scalability with co-processors. A similar pattern exists for MD on (I) and (A), while the small extended skyline cannot be distributed efficiently on (C). The MD specialisation is faster than that of SD but by a decreasing margin as n grows and exposes more tasks for the MD algorithm.

Considering the figures on the right, i.e., scalability with respect to d, we observe a similar pattern as SD outperforms its GPU counterpart by a factor that grows with d. Again we observe that enough parallel tasks must be available for MD to fully utilize the cross-device processing which happens at $d = 8$ for (I) and (A) but never for (C). The improved relative performance at higher values of d is attributable to increases in the number of parallel tasks: the larger extended skyline exposes more data parallelism for MD, and the exponential growth in the number of cuboids exposes more task parallelism for SD.

As argued in the introduction, we only benefit from the additional co-processors if the computation distributes evenly across them. Figure 12 shows the percentage of cuboids (SD) or data points (MD) processed by each GPU and by the two CPU chips. Each processor executes at least 20% of the parallel tasks, with a range of $\approx$ 10 %, indicating a good load balancing between the devices. For SD the two 980s each account for almost 29% of the work, but for MD there is a little more balance: the MD template gets better utilisation out of the CPU chips, which contribute $\approx$ 25 % of the result. We expect lower throughput on the older Titan GPU card, relative to the newer 980 cards, since it has fewer shared multiprocessors; on both templates it contributes roughly 20% of the work, representing a fairly small difference. Thus we obtain near-linear speedup from additional GPUs, despite their being of different generations. This promising result suggests our template methodology handles increasing heterogeneity in the compute ecosystem.

8. CONCLUSION

This paper introduced a novel templating methodology to obtain high utilisation on heterogeneous servers. The templates outline the high-level, hardware-independent control flow in a general algorithm, while the parallel steps are defined in architecture-specific specialisations. We apply the approach with three templates for skycube materialisation. The experiments reveal that our cache-conscious CPU specialisations are much more scalable on parallel architectures than a baseline solution. Most notably, the templates handle heterogeneity well, with each of three GPU cards and a dual-socket CPU contributing equally to the generation of results. Ultimately, we can compute a skycube over the most extreme workload that we study (10^6 points with 16 anticorrelated dimensions) in less than 9 minutes, an instance that the baseline fails to compute after several hours. Moreover, the templates accommodate simply “plugging-in” future concurrent-query or parallel skyline algorithms.

9. ACKNOWLEDGMENTS

This research was supported in part by the *WallViz* (Danish Council for Strategic Research) and *ExiBiDa* (Norwegian Research Council) projects. We thank J. Lee for code [24] and the anonymous reviewers for their suggestions to restructure the paper and add the experiments in Appendix A.2.

10. REFERENCES

[1] K. S. Bøgh, I. Assent, and M. Magnani. Efficient GPU-based skyline computation. In *DaMoN*, pages 5:1–6, 2013.

[2] K. S. Bøgh, S. Chester, and I. Assent. Work-efficient parallel skyline computation for the GPU. *PVLDB*, 8(9):962–973, 2015.

[3] K. S. Bøgh, S. Chester, and I. Assent. SkyAlign: a portable, work-efficient skyline algorithm for multicore and GPU architectures. *VLDB J*, 25(6):817–841, 2016.

[4] K. S. Bøgh, S. Chester, D. Šidlauskas, and I. Assent. Hashcube: A data structure for space- and query-efficient skycube compression. In *CIKM*, pages 1767–1770, 2014.

[5] S. Börzsönyi, D. Kossman, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[6] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. *Trans. Large-Scale Data- and Knowledge-Centered Systems*, 15:1–35, 2014.

[7] L. Chen and X. Lian. Dynamic skyline queries in metric spaces. In *EDBT*, pages 333–343, 2008.

[8] S. Chester, D. Šidlauskas, I. Assent, and K. S. Bøgh. Scalable parallelization of skyline computation for multi-core processors. In *ICDE*, pages 1083–1094, 2015.

[9] S.-R. Cho, J. Lee, S.-W. Hwang, H. Han, and S.-W. Lee. VSkyline: Vectorization for efficient skyline computation. *SIGMOD Rec.*, 39(2):19–26, 2010.

[10] W. Choi, L. Liu, and B. Yu. Multi-criteria decision making with skyline computation. In *IRI*, pages 316–323, 2012.

[11] J. Chomicki, P. Ciaccia, and N. Meneghetti. Skyline queries, front and back. *SIGMOD Rec.*, 42(3):6–18, 2013.

[12] M. Endres and W. Kießling. High parallel skyline computation over low-cardinality domains. In *ADBIS*, pages 97–111, 2014.

[13] M. Endres, P. Roocks, and W. Kießling. Scalagon: An efficient skyline algorithm for all seasons. In *DASFAA*, pages 292–308, 2015.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 325–330. Addison-Wesley, 1994.

[15] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled CPU-GPU architectures. *PVLDB*, 8(4):329–340, 2014.

[16] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[17] H. Im, J. Park, and S. Park. Parallel skyline computation on multicore architectures. *Inf. Syst.*, 36(4):808–823, 2011.

[18] W. Jin, A. K. H. Tung, M. Ester, and J. Han. On efficient processing of subspace skyline queries on high dimensional data. In *SSDBM*, pages 1–12, 2007.

[19] G. T. Kailasam, J.-S. Lee, J.-W. Rhee, and J. Kang. Efficient skycube computation using point and domain-based filtering. *Inf. Sci.*, 180(7):1090–1103, 2010.

[20] H. Köhler, J. Yang, and X. Zhou. Efficient parallel skyline processing using hyperplane projections. In *SIGMOD*, pages 85–96, 2011.

[21] J. Lee and S.-w. Hwang. BSkyTree: scalable skyline computation using a balanced pivot selection. In *EDBT*, pages 195–206, 2010.

[22] J. Lee and S.-w. Hwang. QSkycube: efficient skycube computation using point-based space partitioning. *PVLDB*, 4(3):185–196, 2010.

[23] J. Lee and S.-w. Hwang. Scalable skyline computation using a balanced pivot selection technique. *Inf. Sys.*, 39:1–24, 2014.

[24] J. Lee and S.-w. Hwang. Toward efficient multidimensional subspace skyline computation. *VLDB J*, 23(1):129–145, 2014.

[25] Y. Li and J. M. Patel. BitWeaving: fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[26] S. Liknes, A. Vlachou, C. Doulkeridis, and K. Nørvåg. APSkyline: Improved skyline computation for multicore architectures. In *DASFAA*, pages 312–326, 2014.

[27] MapD Technologies, Inc. The world's fastest platform for data exploration. White paper, 2016. http://go.mapd.com/rs/116-GLR-105/images/MapD%20Technical%20Whitepaper%20Summer%202016.pdf.

[28] S. Meraji, B. Schiefer, L. Pham, L. Chu, P. Kokosielis, A. Storm, W. Young, C. Ge, G. Ng, and K. Kanagaratnam. Towards a hybrid design for fast query processing in DB2 with BLU acceleration using graphical processing units: A technology demonstration. In *SIGMOD*, pages 1951–1960, 2016.

[29] S. Park, T. Kim, J. Park, J. Kim, and H. Im. Parallel skyline computation on multicore architectures. In *ICDE*, pages 760–771, 2009.

[30] J. Pei, W. Jin, M. Ester, and Y. Tao. Catching the best views of the skyline: a semantic approach based on decisive subspaces. In *VLDB*, pages 253–264, 2005.

[31] J. Pei, Y. Yuan, X. Lin, W. Jin, M. Ester, Q. Liu, W. Wang, Y. Tao, J. X. Yu, and Q. Zhang. Towards multidimensional subspace skyline analysis. *TODS*, 31(4):1335–1381, 2006.

[32] H. Pirk, S. Manegold, and M. L. Kersten. Waste not... efficient co-processing of relational data. In *ICDE*, pages 508–519, 2014.

[33] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood. Toward GPUs being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *DaMoN*, pages 11:1–11:8, 2015.

[34] C. Raïssi, J. Pei, and T. Kister. Computing closed skycubes. *PVLDB*, 3(1):838–847, 2010.

[35] Y. Tao, X. Xiao, and J. Pei. SUBSKY: Efficient computation of skylines in subspaces. In *ICDE*, pages 65–74, 2006.

[36] R. R. Veloso, L. Cerf, C. Raïssi, and W. Meira Jr. Distributed skycube computation with anthill. In *SBAC-PAD*, pages 33–40, 2011.

[37] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis. SKYPEER: Efficient subspace skyline computation over distributed data. In *ICDE*, pages 416–425, 2007.

[38] L. Woods, G. Alonso, and J. Teubner. Parallel computation of skyline queries. In *FCCM*, pages 1–8, 2013.

[39] T. Xia and D. Zhang. Refreshing the sky: the compressed skycube with efficient support for frequent updates. In *SIGMOD*, pages 491–502, 2006.

[40] T. Xia, D. Zhang, Z. Fang, C. Chen, and J. Wang. Online subspace skyline query processing using the compressed skycube. *TODS*, 37(2), 2012.

[41] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. X. Yu, and Q. Zhang. Efficient computation of the skyline cube. In *VLDB*, pages 241–252, 2005.

[42] K. Zhang, D. Yang, H. Gao, J. Li, H. Wang, and Z. Cai. VMPSP: Efficient skyline computation using VMP-based space partitioning. In *DASFAA Workshops*, pages 179–193, 2016.

[43] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *SIGMOD*, pages 483–494, 2009.

APPENDIX

A. ADDITIONAL EXPERIMENTS

A.1 Experiments with real data

The experiments in Section 7 focused on standard, synthetic benchmark datasets in order to study performance trends with respect to changes in the workload composition. Here we augment those experiments by studying per-

ID	Name	Source	n	d	$\lvert\mathcal{S}^+\rvert$	File size
NBA	Basketball	http://databasebasketball.com	17 264	8	1 796	1.4 MB
HH	Household	http://usa.ipums.org/usa/	127 931	6	5 774	7.5 MB
CT	Covertype	http://archive.ics.uci.edu/ml/datasets/Covertype	581 012	10	432 253	49.7 MB
WE	Weather	http://cru.uea.ac.uk/cru/data/hrg/tmc/	566 268	15	78 036	71.8 MB

Table 2: Sources and specifications of real datasets.

		NBA	HH	CT	WE
CPU	QSkycube	173	178	904 093	31 968 141
	PQSkyCube	70	128	301 989	15 058 017
	STSC	20	47	3 050 647	1 402 276
	SDSC	73	50	2 122 852	2 272 002
	MDMC	13	26	75 320	118 044
GPU	SDSC	636	217	185 859	559 173
	MDMC	16	26	30 731	23 693
All	SDSC	191	167	107 251	220 705
	MDMC	38	48	16 671	7 630

Table 3: Execution time (ms) on real data. Each CPU algorithm runs with its ideal thread configuration (c.f., Figure 5).

formance on real datasets that have natural data skew. We first describe the datasets (Section A.1.1) and then analyse the performance of the algorithms (Section A.1.2).

A.1.1 Dataset descriptions

Table 2 describes the four real datasets that we use. The first two, *Basketball* and *Household*, are commonly used in skyline and skycube literature (e.g., [22]). The following two datasets, *Covertype* and *Weather*, are more challenging on account of higher dimensionalities, much larger extended skylines, and data skew towards optimum values.

Basketball The NBA dataset is a table of player statistics from the NBA basketball league. Each record corresponds to one player's statistics (e.g., points scored, free throws made, and defensive blocks) for one season. In contrast to traditional analyses of sports statistics, which rank players on each attribute independently, the skyline also identifies players who do not excel in any one statistic but have good composite performance across statistics, i.e., are "well-rounded."

Although the NBA dataset is quite small and several attributes are correlated, it is perhaps the most frequently used real dataset for skyline and skycube research. So, we include it for comparability with previous literature.

Household The HH dataset reports how household income is distributed into different expense categories. Each record corresponds to one household in the US and indicates the percentage of household income expended on various categories (e.g., electricity and rent/mortgage).

The HH dataset is more than 7× larger than NBA but still contains a very small extended skyline of only 5 774 points. As with NBA, we include it primarily for comparability with the results reported in previous literature.

Covertype The CT dataset describes cartographic variables (e.g., elevation, slope, and distance to the nearest roadway) for 30 m × 30 m grid cells of the Roosevelt National Forest in Colorado, USA. We remove the binary and categorical attributes and scale all values to the range $[0, 1]$. Skyline points represent forest areas with unique cartography.

This dataset is challenging for skyline and skycube computation, because many data points share the extreme values. For example, the 3 *hillshade* indices are expressed on a scale of only 255 unique values. Thus, 74 % of the dataset is in the extended skyline and the skylines remain large in subspaces.

Weather The WE dataset reports monthly precipitation, latitude, longitude, and elevation for 566 268 terrestrial positions around the globe. Each record corresponds to a 10' latitude/longitude cell. A skyline record has a subset of months with extreme precipitation relative to its three-dimensional coordinate (preferring elevated, Northeastern positions).

The positional coordinates of the WE dataset are heavily clustered into continents and mountain ranges and the alternate biomes present different annual patterns of precipitation. Thus, the WE dataset captures quite well some non-trivial dependence among attributes.

A.1.2 Results and analysis

Table 3 presents the execution times for each algorithm on each of the real datasets and on each applicable architecture. Recall that QSkycube (the top row) is the current state-of-the-art, PQSkycube is our best parallelisation of it, and the other three algorithms are our template methods.

First consider the performance of the CPU algorithms on the small datasets, NBA and HH (first two columns). The state-of-the-art QSkycube can already solve both instances in less than 200 milliseconds; nonetheless the parallelisations still obtain a significant speed-up: MDMC, in particular, lowers the execution time to 13–26 milliseconds, which is a 6-fold speed-up on HH and an 13-fold speed-up on NBA. For all five algorithms, the running time is mostly dominated by that of computing the extended skyline; although, in the case of SD, the number of synchronisation points (i.e., 2^d) is also critical because there are so few points to process, with $|\mathcal{S}^+| < 6\,000$. Thus, ST is faster than SD on both datasets.

These factors are even more noticeable when SD is run on the GPU, where the small extended skyline activates too few threads to effectively utilise the card and the synchronisation points are especially expensive. As a result, SD is significantly slower on the GPU than on the CPU for these workloads. MD similarly is unable to fully utilise the GPU card, given the small number of data-parallel tasks; so, it is slightly slower relative to running on the CPU. For the cross-device execution, neither SD nor MD can utilize the available hardware, as the input is too small. This illustrates the importance of having enough work to distribute across the co-processors, if using more than one.

Turning to the larger datasets (last two columns), we observe differing behaviour. For CT, many skylines are very similar because of the large number of points that have the same optimum values. Thus, the information sharing between parent and child cuboids used in QSkycube and PQSkycube is especially effective, and PQSkycube is an order of magnitude faster than ST and 7× faster than SD.

In contrast, this is not observed on the WE dataset, where instead the parent sharing incurs extra communication and PQSkycube is an order of magnitude slower than ST. ST is faster on the higher-dimensional workload (WE), whereas SD is faster on the higher-cardinality workload (CT), illustrating each template's strength. In contrast, MD is more robust to these characteristics with a 4× and 11× speed-up over the closest competitor on CT and WE, respectively.

On these larger workloads, SD and MD both benefit significantly from the high-throughput GPU. For the heterogeneous processing we also see a gain for both SD and MD, although it is higher for SD, gaining 19× on CT. Generally SD gains the most from additional accelerators on CT, whereas MD gains the most on WE. Across all datasets, MD performs the best with speedups to the nearest competitor ranging from 1.5× to 28×. This reflects the advantage of exposing more parallel tasks and computing the skycube point by point, rather than cuboid by cuboid. Indeed, compared to the sequential state-of-the-art, our heterogeneous execution of the MD template improves execution time on CT and WE 54-fold and 4189-fold, respectively.

A.2 Partial skycube computation

In some scenarios, it may be desirable to generate only part of the skycube. For example, given the large percentage of points that make up high-dimensional skylines, it may be unnecessary to compute high-dimensional subspace skylines as they may not offer much utility to end users. Alternatively, there may be subspaces that are unlikely to interest any user, such as strange combinations of months in our WE dataset. These subspaces could be explicitly excluded from computation as an orthogonal means to accelerate skycube materialisation. Here, we study how well each algorithm benefits if we only need subspaces up to a fixed dimensionality, $d' \leq d$. Section A.2.1 describes our modifications to the algorithms to support partial skycube computation and Section A.2.2 analyses their performance post-modification.

A.2.1 Experiment setup

Modifications to lattice-based algorithms Recall that PQ, ST, and SD all compute the skyline with a top-down lattice traversal, using the results at lattice level $|\delta| + 1$ as reduced input at level $|\delta|$. If starting at level d', then computing level $d'+1$ is unnecessary. Thus, to omit all subspaces above level d', we start by computing the full extended skyline and use that as input at level d' (rather than using the results from level $d' + 1$). After computing level d', we can reuse results from immediately preceding levels as in the unmodified algorithms. Memory consumption is appreciably reduced if $d' > d/2$, because we need not store results in the lattice for subspaces δ where $|\delta| > d'$.

Observe the trade-off: the modified algorithms need to compute fewer subspace skylines, but they must compute the uppermost level with a larger input.

Modifications to MDMC Recall that MD consists of both a *filter* and a *refine* phase that are executed for each point, p. In the former phase, a number of subspace skylines are quickly identified as not containing p. In the latter phase, the remaining subspaces are enumerated in a list; for each subspace in the list, we explicitly verify whether p is in its skyline. Our modification simply omits adding a subspace δ to the list if $|\delta| > d'$. In other words, our filter phase may still prune some points from an "uninteresting" subspace δ if $|\delta| > d'$, since it is cheaper than verifying $|\delta|$ on the fly, but no correctness guarantee is offered for the skyline of said δ.

A post-processing step could unset "uninteresting" bits in the HashCube to offer more compression, but MD has low memory consumption anyway. Future work could reorganise bits of the HashCube by lattice level to improve its compression on partial skycubes: entire words could then be omitted if they only contained high-dimensional subspaces. This would particularly help the GPU specialisation of MD, because it would decrease the amount of "state" required to process each point: The limiting factor for concurrency in the GPU specialisation is the amount of shared memory devoted to storing the HashCube bitmasks.

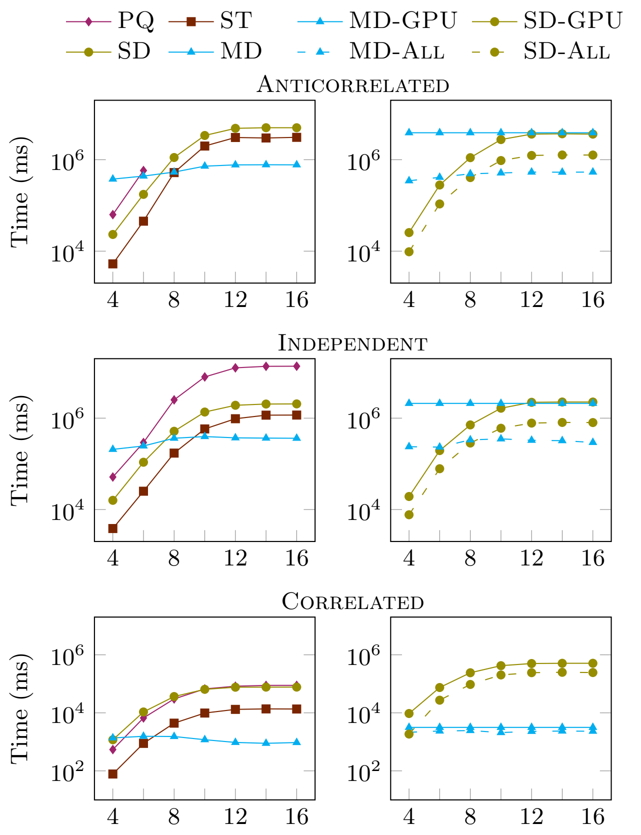


Figure 13: Execution times for partial skycube computation relative to the number of lattice layers computed and distribution ((A)-top; (I)-middle; (C)-bottom).

A.2.2 Results and analysis

Figure 13 evaluates the execution times of the modified algorithms on 16d synthetic datasets. We vary on the x-axis the maximum dimensionality of the "interesting" subspaces. The left-hand plots include the baseline, PQ, and the CPU specialisations (c.f., Figure 6) and the right-hand plots include the GPU specialisations and the cross-device (-All) runs (c.f., Figure 7). As before, the plots are sorted from top to bottom by increasing data correlation.

We observe first that on all architectures the lattice-based algorithms can obtain significant savings by computing only the lowermost levels, whereas the savings for MD are quite modest. In fact, on correlated data, MD is often unable to amortise the overhead of checking the dimensionality of subspaces that are added to the refine list. Thus, for MD, one might as well compute the entire skycube, but for the lattice-based methods, partial skycube computation should be exploited whenever appropriate for the use case.

For the lattice-based methods, the gains are appreciable if the maximum dimensionality d' is not more than half the input dimensionality. At $d = 8$, the thickest layer of the lattice must still be computed; so, < 40 % of processing is pruned. Computing the middle level of the lattice incurs the same peak memory consumption as the unmodified algorithms; in the case of PQ, which also retains its $\binom{16}{8}$ pointer-based quad trees to reuse at $|\delta| = 7$, this memory consumption is still prohibitive (for our 128 GB of memory) on the anticorrelated data.

For the CPU specialisations, MD generally outperforms SD and ST if computing at least 8 levels of the skycube and it outperforms PQ if computing at least 6 levels. ST is always preferable to SD, which in turn is always preferable to the baseline PQ. For the GPU specialisations, SD is typically preferable for partial skycube computation, unless the data is correlated. The cross-device experiments unsurprisingly show a mix of these trends: SD is preferable if the data is not correlated and fewer than 8 levels are to be computed.

B. A REVIEW OF SKYLINE CONCEPTS

In this appendix we review the HashCube data structure (Appendix B.1) and point-based partitioning for skylines (Appendix B.2), which are both used by MDMC.

B.1 HashCube compression

The HashCube [4] is a data structure for obtaining up to w-fold compression of skycubes. Whereas the lattice structure (Figure 1a) replicates each point id for every subspace skyline in which it appears, the HashCube (Figure 1b) stores each point id at most once per fixed group of w subspaces.

Specifically, each point p is represented by $\mathcal{B}_{p \notin \mathcal{S}}$, the set of subspaces in which p is dominated. This length $2^d - 1$ bitmask is then split into "words" of length w and each word is hashed independently to a list of point ids. Thus p appears in at most one list per w subspaces. For example, in Figure 1b (where $w = 4$), $\mathcal{B}_{f_1 \notin \mathcal{S}}$ is split into $w_1 = 000$ and $w_0 = 1011$. For hash function h_0, flight id f_1 will be stored in the list at key 1011 whereas for hash function h_1, it will be stored in the list at key 000. (If all bits of a word are set, i.e., a point is dominated in all w subspaces, the point id is not stored at all for that hash function.)

To retrieve the skyline $\mathcal{S}_\delta(P)$ of subspace δ, one iterates the keys of the hash function $h_{(\delta-1)/w}$ and concatenates the point lists for all keys on which the $((\delta-1)\%w)$'th bit is set. In this example, $\mathcal{S}_4(P)$ is recorded by h_0 and is the concatenation of lists for keys 9, 10, and 11.

Representating the skycube with per-point bitmasks facilitates the finer-grain parallelism of MDMC: each parallel task can produce one bitmask independently of the others.

B.2 Point-based partitioning

Recent shared-memory skyline algorithms [2, 8, 21, 23, 42, 43], including those in our STSC and SDSC hooks, are efficient because they utilise *point-based partitioning.* MDMC also makes use of this technique; so, we review it briefly here.

The key idea, illustrated in Figure 14 for the flights example, is to avoid explicit point-to-point comparisons using

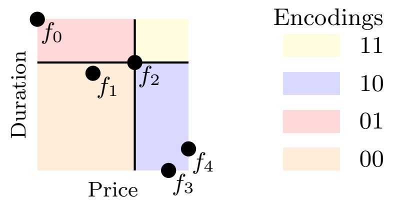


Figure 14: The mapping from (the first two attributes of) the flights in Table 1 onto 2^d bitmask encodings for point-based partitioning, using f_2 as a "pivot" point.

transitivity with respect to a common "pivot" point, in this case f_2. The position relative to the pivot of each point p is represented by a d-length bitmask in which the δ'th bit is set iff $p[\delta] \geq \text{piv}[\delta]$. (E.g., f_0—or any point in the red region—is assigned a bitmask 10 as it has a better price but worse (or equal) duration than the pivot, f_2.) As one rarely handles $d > 32$, the bitmasks can be stored as 32-bit integers and compared to each other with primitive bitwise operations. Specifically, if the bitmasks $\mathcal{B}_{\pi \leq p}$ and $\mathcal{B}_{\pi \leq q}$ indicate the dimensions on which the common "pivot" point π is better than p and q, respectively, then $p \prec_\delta q$ only if:

$$(\mathcal{B}_{\pi \leq q} \mid \sim\!\mathcal{B}_{\pi \leq p}) \,\&\, \delta = \delta. \tag{1}$$

If Equation 1 does not hold, then there clearly exists a dimension i of δ on which $q[i] < \pi[i] \leq p[i]$; so, $p \not\prec_\delta q$. Compared to an exact *dominance test* (DT), which evaluates Definition 1 by loading and comparing up to $|\delta|$ values for each point, this *mask test* (MT) only requires one value, reducing memory traffic and improving cache hit ratios.

For example, we immediately see that points in the red region (e.g., f_0) do not dominate points in the orange region (e.g., f_1), because Equation 1 does not hold for $\mathcal{B}_{\pi \leq p} = 01$ and $\mathcal{B}_{\pi \leq q} = 00$: the second bit reveals through transitivity with respect to f_2 that the red points have worse durations. On the other hand, Equation 1 holds in the opposite case of the orange region relative to the red region, but this is insufficient to conclude that all orange points dominate all red points (c.f., f_1 versus f_0). Thus, the mask tests are effective at skipping work only in the presence of incomparability. In the case of f_3 and f_4, which both have the same 10 encoding, the mask test reveals no information, despite that $f_3 \prec f_4$. In general, then, point-based partitioning becomes more effective as dimensionality increases, because: a) there is more incomparability to detect; and b) each d-length bitmask carries more bits of information.

In most point-based partitioning algorithms [8, 21, 23, 42, 43], these bitmasks are generated with a recursive, on-the-fly, quad tree partitioning of P, and the algorithms principally vary on how they select pivot points. For an as-yet-uninserted point p, one traverses the quad tree, generating new bitmasks relative to each node and then evaluating Equation 1. One only descends into a subtree after a failed mask test. MDMC adopts the non-recursive, virtual median- and quartile-based pivot points proposed in SkyAlign [2] (e.g., in Figure 3), because the partitioning and quad tree construction can be done statically in advance. Moreover, one need not load data points to generate new bitmasks in order to traverse the quad tree, because the bitmasks are statically defined.