Efficient caching for constrained skyline queries

Michael Lind Mortensen #1, Sean Chester #2, Ira Assent #3, Matteo Magnani *4

Aarhus University, Denmark * Uppsala University, Sweden

1 illio@cs.au.dk 2 schester@cs.au.dk 3 ira@cs.au.dk 4 matteo.magnani@it.uu.se

ABSTRACT

Constrained skyline queries retrieve all points that optimize some user's preferences subject to orthogonal range constraints, but at significant computational cost. This paper is the first to propose caching to improve constrained skyline query response time. Because arbitrary range constraints are unlikely to match a cached query exactly, our proposed method identifies and exploits similar cached queries to reduce the computational overhead of subsequent ones.

We consider interactive users posing a string of similar queries and show how these can be classified into four cases based on how they overlap cached queries. For each we present a specialized solution. For the general case of independent users, we introduce the Missing Points Region (MPR), that minimizes disk reads, and an approximation of the MPR. An extensive experimental evaluation reveals that the querying for an (approximate) MPR drastically reduces both fetch times and skyline computation.

1. INTRODUCTION

Constrainted Skyline A constrained skyline query [3] is an effective way of filtering a constrained dataset to points that express all optimal trade-offs of the dataset's attributes. For example, if searching for cheap 3+ star hotels near a conference venue, one hotel is said to *dominate* another if it is at least as highly rated, well priced and near as the latter, yet strictly better than the other hotel on at least one of these metrics. The *constrained skyline* is the set of points that satisfy the given constraints and are not dominated by any others that satisfy the constraints. In practice, the constraints are critical in allowing users to determine the skyline on the data relevant to them. E.g. average income earners may not be interested in luxury hotels nor backpacker hostels, so a non-constrained skyline cannot capture their preferences. Constraints reduce the input size, yet, paradoxically, makes computing the skyline quite challenging, because, unlike an unconstrained skyline which can simply be pre-materialized, the skyline points are unpredictable.

The naive approach, presented in [3], is to execute a range query to fetch points satisfying the constraints, and then compute the skyline over those points using an efficient skyline algorithm (e.g., [7, 8, 16, 23]). This has the advantage of simplicity, but the performance is highly sensitive to the selectivity of the range query. The best known technique is the I/O-optimal *BBS* algorithm [19], which uses an R-tree index and a heap-based priority queue to guide the search for skyline points, while pruning paths in an R-Tree if outside the constraints. In this paper, we outperform *BBS* by reusing partial query solutions.

Caching A fair assumption is that many users will pose constrained skyline queries on the same dataset. Where users have similar needs, the constraint regions likely overlap. For example, young backpackers will all typically search with price constraints that match a cheap budget, producing similar queries. Business travellers, conversely, may be more concerned with location than price; a distinct set of similar queries.

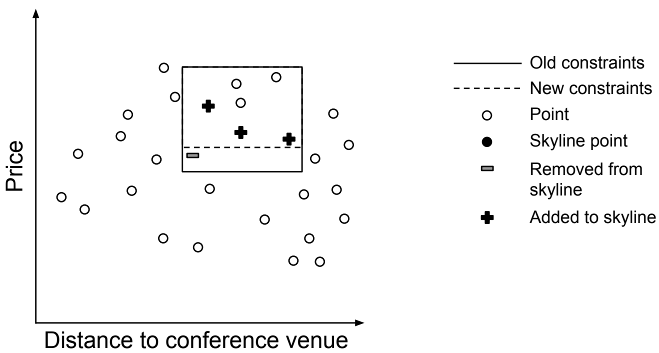


Figure 1: Small constraint changes have large impacts on skylines

Additionally, an iterative, exploratory query-refine cycle is common in search tasks [18], where a user issues a query, observes the results, and then adjusts constraints to manipulate the results. Hence, even a single user can produce strings of highly similar queries, each with distinct skylines [6, 17].

So, this paper addresses a natural question: *how can the results of a constrained skyline query be reused to speed up subsequent, similar ones?* In contrast to existing techniques (e.g., [3, 19]), we can obtain significant speed-up by decomposing a range query into disjoint smaller ones, and discarding those that a previous cached query result implies are unnecessary, hence reading fewer data points than the isolated query necessitates.

Challenges Despite the apparent simplicity of overlapping two range queries to compare results, caching constrained skylines is deceptively challenging. Unlike previous research on caching (subspace) skyline queries [2, 14, 20] where constraints are not considered, cache hits with exact matching constraints are quite unlikely, especially for real-valued and high-dimensional data. Therefore,

10.5441/002/edbt.2015.30

we investigate how to infer partial skyline solutions from *overlapping*, rather than matching, query constraints, which would yield a low cache hit rate.

This, however, introduces a new challenge, because small changes to constraints can have a profound impact on the skyline [6]. For example, Figure 1 shows an "old" (solid) and "new" (dashed) constrained skyline query on a toy hotel example, where the minor increase to the $Price$-constraint from the "old" to the "new" query is enough to eliminate a skyline point (minus), which promotes three previously dominated points into the skyline (plus). To address this challenge, we introduce the notion of *stability* which characterizes when solution points will be shared among old and new queries. Even in difficult, *unstable* cases, we show how a previous query's solution points, even when not satisfying the current constraints, can prune the current search range.

Finally, dimensionality poses a natural challenge for caching constrained skylines by increasing the pruning complexity (as we show in Section 5.3). Therefore, we present an effective approximation technique that balances the number and selectivity of small range queries. As a result, we can outperform baseline and the I/O-optimal BBS algorithms by several factors, and scale elegantly with increasing workloads.

Contributions Despite the cost of constrained skylines, this paper is the first to investigate how caching can drastically improve their running time. After reviewing related work (Section 2) we introduce the problem formally (Section 3) and make the following novel contributions (Sections 4-6) before concluding the paper (Section 8).

- For the exploratory use case, in which subsequent queries differ only by one constraint, we present a case-by-case breakdown of the four possible overlap relationships, along with how to compute the constrained skyline for each (Section 4);
- For the general case of arbitrary constraint overlap, we introduce an algorithm to compute the *Missing Points Region* (MPR), which is the minimal region that must be queried from disk. We also introduce an *Approximate MPR* (aMPR), sacrificing minimality for fewer independent range queries (Section 5);
- We introduce a caching algorithm, *Cache-Based Constrained Skyline* (CBCS) that handles cache searching, management and use based on the earlier analysis (Section 6); and
- We conduct an extensive experimental evaluation of our method to show when our caching yields superior efficiency for related queries relative to baselines and state-of-the-art (Section 7).

2. RELATED WORK

Constrained skylines The skyline operator [3] was introduced along with a straightforward extension to constrained skyline queries that first retrieves all data satisfying the constraints, and then applies any skyline algorithm (e.g., [7,8,16,23]). Subsequently, the R-tree-based method *BBS* [19] supports constraints by pruning paths in the R-tree if they are outside the constraint region. *BBS* is I/O-optimal and state-of-the-art when not using caching. We include it in our empirical study (Sect. 7).

For arbitrary subsets of dimensions, known as subspaces, [10] partitions data and queries vertically onto several low-dimensional R-trees. Without subspaces, their approach is essentially a constraint-based version of the *NN* method [15], shown in [19] to be inferior to *BBS* for constrained skylines. [9] study distributed constrained skylines where distributed local skylines are merged into a global result. Efficiency gains come from computing independent local skylines at data sites in parallel, meaning that a non-distributed application is equivalent to computing the constrained skyline naively. [1] study constrained subspace skylines in a horizontally partitioned P2P environment. Constrained subspace skylines are computed in order of potential dominance on each node, avoiding those pruned by earlier nodes. It suffers the same limitations as [9]. [22] study continuous constrained skyline queries for streams, determining areas that could influence the current skyline. The problem is different from ours, namely maintenance of fixed constrained skylines for dynamic data, rather than dynamic constraints. [11] study query optimization of *Semi-skylines*, which use partial order preferences. If applied only to traditional constraints, the method corresponds to recomputation from scratch for any change in constraints. [4] estimates the cardinality of (constrained) skylines in a DBMS and can be used to assess which skyline algorithm to apply in the naive approach.

A user study [17] uses existing constrained skyline algorithms to investigate how users understand, issue and interact with constrained skylines. Finally, [6] study how dynamic changes of constraints and subspaces affect skyline membership. Neither [6] nor [17] offer algorithmic contributions.

Caching skylines [12] and [5] study caching of subspace skylines in a P2P setting using local caches with a superpeer network and a centralized index, respectively. Neither support constrained skylines. [2] study caching of subspace skylines, where results are cached directly and used to answer queries in related subspaces. [14] caches partial-order domain user preferences to process queries with similar user preferences. [20] caches dynamic skylines, where domination is based on the distance to a query. None of them consider constraints and thus suffer from the same issues as [12] and [5].

In conclusion, existing constrained skylines algorithms demand recomputation from scratch if constraints differ even slightly. Also, existing caching approaches only support identical constraints, which is unlikely to occur in practice, especially when considering, e.g., exploratory search scenarios, real-valued data, multiple users and several dimensions. In this work, we limit the number of points read and dominance tests performed, by reusing cached results on similar constraints.

3. PRELIMINARIES

Let S be a set of data points over an ordered set of numerical dimensions D, where the value of $s \in S$ in dimension $i \in D$ is denoted $s[i]$. A *set of constraints*, $C = \langle \underline{C}, \overline{C} \rangle$, is a pair of points indicating the minimum value, $\underline{C}[i]$, and the maximum value, $\overline{C}[i]$, for each dimension $i \in D$. A *constraint region*, R_C, is the set of all possible points satisfying constraints C:

$$R_C = \left\{ p \in R^{|D|} \mid \forall i \in D : \underline{C}[i] \leq p[i] \leq \overline{C}[i] \right\}.$$

Observe that R_C describes a $|D|$-dimensional hyper-rectangle, like the rectangles in Figure 1. Similarly, the *constrained data*, S_C, is the set of data points that satisfy constraints C:

$$S_C = \left\{ s \in S \mid \forall i \in D : \underline{C}[i] \leq s[i] \leq \overline{C}[i] \right\}.$$

We note the following properties relating constraint regions and constrained data:

1. Given constraints C, the set of points S_C satisfying constraints C form a (possibly empty) subset of the set of points in the region R_C described by C: $S_C \subseteq R_C$.

Data & space notation	
S	Dataset S
D	Dimensions of S
$R = \left\{p \in \mathbb{R}^{\lvert D \rvert}\right\}$	Region of potential $\lvert D \rvert$-dimensional points
S_C	Points in S limited by constraints C
R_C	Region R limited by constraints C
p, q	Points in region R
s, u, t, v	Points in dataset S.
$p[i], s[i]$	Value in dimension i of point p, s, resp.
Query notation	
$C = \langle \underline{C}, \overline{C} \rangle$	Constraints consisting of low/high limits
$\underline{C}[i], \overline{C}[i]$	Lower/upper constraint on dimension i
$Sky(S, C)$	Skyline on S constr. by C
$s \succ t$	Point s dominates point t
$DR(s)$	Dominance region of point s
$DR(s, C)$	Dominance region of point s constrained by C
$RQ(C) = (S_C \cap R_C)$	Range query on the region constrained by C

Table 1: Notation

2. (Data determines space) $s \in S_C \implies s \in R_C$.
3. (Space only contains constrained data) $p \in R_C \implies p \in S_C \vee p \notin S$.

In this paper we study how to efficiently answer *constrained skyline queries* (Def. 1) given S and C, if an in-memory cache (Def. 3) is available. The skyline is defined through the concept of *dominance* [3]: a point $s \in S$ (or, analogously, $s \in R$) *dominates* another point $t \in S$, denoted $s \succ t$, iff $\exists i \in D$: $s[i] < t[i]$ and $\forall j \in D, s[j] \leq t[j]$.[1] In order words, s is at least as small as t on every attribute, and strictly smaller on at least one.

The *conventional skyline* of S, denoted $Sky(S)$, is the subset of points not dominated by any other points in S:

$$Sky(S) = \{s \in S \mid \nexists t \in S : t \succ s\}\,.$$

These are exclusively those points that minimize some linear function over the dataset's dimensions. Figure 2 illustrates the skyline with black points for our running hotel example.

The *constrained skyline* is the set of points that satisfy the constraints while not being dominated by any other points that also satisfy the constraints (Definition 1). Equivalently, it is the skyline over input S_C.

Definition 1 (Constrained skyline [3, 19])**.**
Given S, C, *the* constrained skyline*, denoted* $Sky(S, C)$*, is:*

$$Sky(S, C) = Sky(S_C) = \{s \in S_C \mid \nexists t \in S_C : t \succ s\}\,.$$

Figure 2 also illustrates a *constrained skyline* as the set of gray points in the rectangle spanned by constraints $C = \langle \underline{C}, \overline{C} \rangle$. Note here again that the constrained skyline can be very different from the conventional skyline (black points).

Every point $s \in S$ (or $s \in R$) has a *dominance region* [10], denoted $DR(s)$, which is the hyper-rectangular region in which any point p is dominated by s (Definition 2).

[1] Assumed without loss of generality: a preference for maximization can be handled by multiplying an attribute by -1.

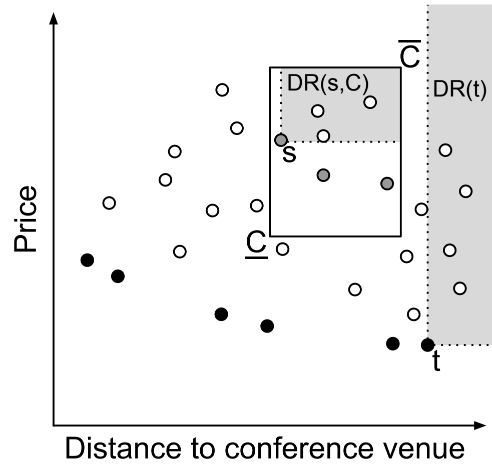


Figure 2: Illustration of a skyline (black points) and constrained skyline (grey points inside the rectangle) on our running example. Also shown are dominance regions (solid gray rectangles).

Definition 2 (Dominance region [10])**.**
For point $s \in S$*, the* dominance region *is defined as:*

$$DR(s) = \{p \in R \mid s \succ p\}\,.$$

For any $s \in S$, $DR(s) \cap Sky(S) = \emptyset$; so dominance regions help detect subsets of points that need not be fetched from disk. In the presence of constraints C, each point s also induces a *constrained dominance region*, denoted $DR(s, C)$, which is the portion of $DR(s)$ that satisfies C. The gray rectangles in Figure 2 illustrate $DR(t)$ for a conventional skyline point and $DR(s, C)$ for a constrained skyline point.

Lastly, our objective is to resolve constrained skyline queries using in-memory constrained skyline cache items. A *cache item* (Definition 3) is a 3-tuple consisting of an earlier queried set of constraints, the resultant constrained skyline, and the skyline's *minimum bounding rectangle* (*MBR*).

Definition 3 (Constrained skyline cache)**.**
An in-memory cache holding n cache items $\{\mathcal{I}_1, \ldots, \mathcal{I}_n\}$*, where each cache item* $\mathcal{I}_i$ *is a* 3*-tuple:*

$$\mathcal{I}_i = \langle Sky(S, C), \mathrm{MBR}, C \rangle$$

$Sky(S, C)$ *is the skyline result on constraints* C *and* MBR *is the minimum bounding rectangle of* $Sky(S, C)$.

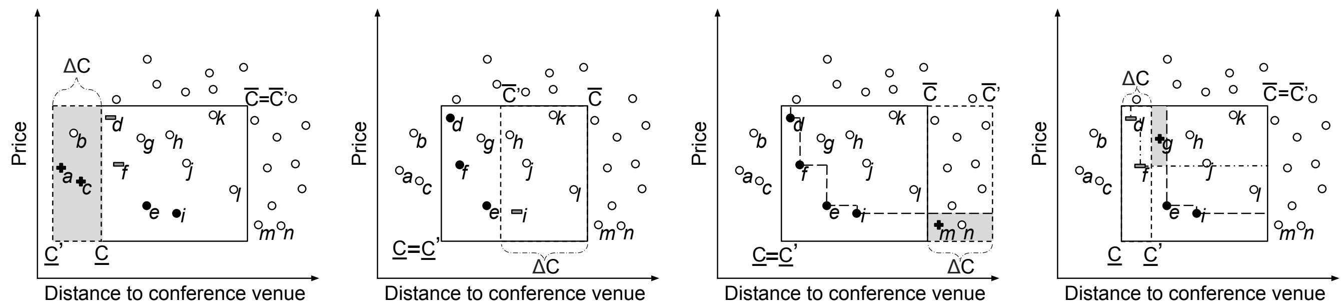
With the notation in place (and summarized in Table 1), the problem studied in this paper can now be stated as follows:

Problem Statement (Cache-based constrained skyline).
Given S, C', and an in-memory cache $\{\mathcal{I}_1, \ldots, \mathcal{I}_n\}$, utilize a cache item $\mathcal{I}_i$ to compute $Sky(S, C')$ without fetching all of $S_{C'}$.

4. EXPLOITING RELATED QUERIES

Recall from Section 2 that existing caching techniques for skylines require an exact match on constraints. A user who continually modifies, say, the *price* or *distance* constraints as he/she refines his/her hotel search therefore produces a long string of cache misses, despite having made only small, incremental changes to his/her query.

In this section, we focus on these *incremental changes*, where old constraints C and new constraints C' overlap in all but one



(a) Decreasing lower constraint (b) Decreasing upper constraint (c) Increasing upper constraint (d) Increasing lower constraint

Figure 3: Cases (a)-(d) of incrementally changing one constraint at a time; our solutions fetch only the points in the gray regions.

dimension. In doing so, we both address the potential for large computational savings in this principal case and build intuition for the general methods presented in Section 5.

Specifically, we use $Sky(S, C)$ to limit how much of $S_{C'}$ that must be fetched from disk to determine $Sky(S, C')$. We first introduce the concept of *stability* to characterize when constrained skylines share solution points (Section 4.1). Then, we identify the four possible (and easily detectable) manners in which incremental constraint changes may overlap, presenting specialized solutions for each (Section 4.2).

4.1 Skyline stability

Clearly, a cache item for $Sky(S, C)$ indicates which points from S are in $Sky(S, C)$. More importantly, it also implies which points from S_C *are not* in $Sky(S, C)$. *Stability* (Definition 4) captures this insight relative to new constraints, C'.

Definition 4 (Constrained skyline stability)**.**
We say that $Sky(S, C)$ is stable *relative to C' iff:*

$$s \in Sky(S, C') \implies (s \notin S_C) \vee (s \in Sky(S, C))$$

In other words, $Sky(S, C)$ is stable relative to C' if points from S_C not in $Sky(S, C)$ are also not in $Sky(S, C')$. Otherwise, we call $Sky(S, C)$ *unstable* relative to C'. We observe in Theorem 1 that stability is guaranteed when, for all $i \in D$, $\underline{C}[i] \geq \underline{C}'[i]$ (or, trivally, when constraints do not overlap).

Theorem 1 (Guaranteed stability)**.**
$Sky(S, C)$ is guaranteed to be stable relative to C' iff:

$$(\forall i \in D\colon \underline{C}'[i] \leq \underline{C}[i]) \ \vee$$
$$(\exists i \in D\colon \underline{C}'[i] > \overline{C}[i] \vee \overline{C}'[i] < \underline{C}[i])$$

The full proof of Theorem 1 is in Appendix 9, but the intuition is that new constraints can only invalidate the skyline if they shrink the constraint region, removing skyline points and thereby their dominance region influence on the skyline result. Stability is gauranteed since no point $s \in S_C$ dominated by removed point $t \in Sky(S, C)$ can satisfy an upper constraint that t does not satisfy.

From Definition 4 and Theorem 1 come two natural consequences. First, for stable cases, we need only fetch points in the new part of the constraint region that did not satisfy the old constraints (Corollary 1). Second, a skyline result is unstable if and only if an "old" skyline point $s \in Sky(S, C)$ is outside the "new" constraints C', and it dominated points that still satisfy the new constraints (Corollary 2).

Corollary 1. *If $Sky(S, C)$ is stable relative to C' then:*

$$\forall s \in Sky(S, C')\colon \ (s \in Sky(S, C)) \ \underline{\vee} \ (s \in (S_{C'} \setminus S_C))$$

Corollary 2. *$Sky(S, C)$ is unstable relative to C' iff:*

$$\exists t \in Sky(S, C)\colon t \notin S_{C'} \ \wedge$$
$$\exists s \in (S_C \cap S_{C'})\colon t \succ s \ \wedge$$
$$\nexists u \in (Sky(S, C) \cap S_{C'})\colon u \succ s$$

4.2 Incremental constraint changes

With the theory of stability in place, we show how any incremental change can be solved with minimum points read. For each of four possible cases, we prove the correctness and minimality (proofs in Appendix 9) of our solution and illustrate the intuition of the ideas by example/illustration.

Figures 3a-3d show the four cases for incremental changes of constraints C (the solid rectangle), one dimension at a time: (a) decreasing a lower constraint, (b) decreasing an upper constraint, (c) increasing an upper constraint and (d) increasing a lower constraint. Note that we always have only these four cases, regardless of dimensionality. The initial constraints $C = \langle \underline{C}, \overline{C} \rangle$ and new constraints C', as well as the change ΔC from C to C' are displayed in each figure. For each case, the part of $S_{C'}$ that we fetch is enclosed in the gray region. Note that while the illustrated gray regions are all rectangular, this only holds for $|D| = 2$ as we will show in Section 5.3.

Case (a): Decreasing a lower constraint (Fig. 3a) From Theorem 1, $Sky(S, C)$ is stable relative to C', and from Corollary 1 all new skyline points lie in ΔC. Instead of fetching all of $S_{C'}$, we can fetch just the points in $(R_{C'} \setminus R_C)$. Further pruning is not possible: no points in $Sky(S, C)$ dominate any part of ΔC.

Theorem 2 (Case (a) solution)**.**
If $\overline{C}' = \overline{C}$, $\exists i\colon \underline{C}'[i] < \underline{C}[i]$, $\forall j \in D \setminus \{i\}\colon \underline{C}'[j] = \underline{C}[j]$, then:

$$Sky(S, C') = Sky(Sky(S, C) \cup S_{\Delta C}, C')$$

In the example (Fig. 3a), a, b and c are fetched from the database with a, c as new skyline points (illustrated by a plus sign), while existing d, f are dominated by a and c under constraints C' (illustrated by a minus sign). The final skyline is a, c, e, i. Without the cached $Sky(S, C)$, we must read 12 points, a-l, from disk.

Case (b): Decreasing an upper constraint (Fig. 3b) Again, from Theorem 1, $Sky(S, C)$ is stable wrt C'. From Corollary 1 the skyline points in $Sky(S, C')$ are in $Sky(S, C)$ or in ΔC. Since $R_{C'}$ is enclosed in R_C, we need simply remove the previous skyline points not satisfying the new constraints.

Theorem 3 (Case (b) solution)**.**
If $\underline{C}' = \underline{C}$, $\exists i\colon \overline{C}'[i] < \overline{C}[i]$, $\forall j \in D \setminus \{i\}\colon \overline{C}'[j] = \overline{C}[j]$, *then:*

$$Sky(S, C') = Sky(S, C) \cap S_{C'}$$

In this example (Fig. 3b), only i falls outside the new constraints and is simply removed to obtain the new skyline.

Case (c): Increasing an upper constraint (Fig. 3c) As before, $Sky(S, C)$ is stable relative to C' (Theorem 1), and new skyline points in $Sky(S, C')$ are in $Sky(S, C)$ or in ΔC (Cor. 1). Unlike before, however, we use the dominance regions of points in $Sky(S, C')$ to further prune parts of ΔC:

Theorem 4 (Case (c) solution)**.**
If $\underline{C}' = \underline{C}$, $\exists i\colon \overline{C}'[i] > \overline{C}[i]$, $\forall j \in D \setminus \{i\}\colon \overline{C}'[j] = \overline{C}[j]$, *then:*

$$Sky(S, C') = Sky(Sky(S, C) \cup \{s \in S_{\Delta C} \mid \nexists t \in Sky(S, C)\colon t \succ s\}, C')$$

We thus prune $S_{\Delta C}$ such that we only read $((S_{C'} \setminus S_C) \setminus \{s \in S_{\Delta C} \mid \exists t \in Sky(S, C)\colon t \succ s\})$. In the example (Fig. 3c), $R_{C'}$ contains 18 points, the logic of Case (a) reduces it to 9 points, and we eventually fetch only 2 points, m and n.

Case (d): Increasing a lower constraint (Fig. 3d) Unlike Cases (a)-(c), $Sky(S, C)$ is not stable relative to C'. Despite this instability, we can use the old skyline result by determining invalidated parts of the cache item and reevaluate these under constraints C'. Using what is left of $Sky(S, C)$ within the queried constraints C' we prune regions before reading from disk. Since no two skyline points dominate each other (Def. 1), the remaining part of $Sky(S, C)$ is not invalidated:

Theorem 5 (Case (d) solution)**.**
If $\overline{C}' = \overline{C}$, $\exists i\colon \underline{C}'[i] > \underline{C}[i]$, $\forall j \in D \setminus \{i\}\colon \underline{C}'[j] = \underline{C}[j]$, *then:*

$$\begin{aligned} Sky(S, C') = Sky((&Sky(S, C) \cap S_{C'}) \cup \\ &\{s \in (S_C \cap S_{C'}) \mid \\ &\exists t \in (Sky(S, C) \cap S_{\Delta C})\colon t \succ s \ \wedge \\ &\nexists u \in (Sky(S, C) \cap S_{C'})\colon u \succ s\}, C') \end{aligned}$$

Thus, we avoid reading $(S_C \cap S_{C'})$ fully, and retrieve only $((S_C \cap S_{C'}) \setminus \{s \in (S_C \cap S_{C'}) \mid \nexists t \in (Sky(S, C) \cap S_{\Delta C})\colon t \succ s \vee \exists u \in (Sky(S, C) \cap S_{C'})\colon u \succ s\})$.

In the example (Fig. 3d) instead of 7 points, we only fetch g. Throughout all examples, we save the latency of fetching unnecessary points, and the cost of conducting dominance tests over an otherwise larger input.

5. ARBITRARY CONSTRAINT CHANGES

In this section, we build on the intuition from Section 4 to handle the general case where the number of constraint changes is arbitrary. We first generalize the gray regions of the previous section into the *Missing Points Region* (*MPR*), the minimal area that must be fetched (Section 5.1), and introduce an efficient algorithm to compute it (Section 5.2). We then illustrate how the MPR grows arbitrarily complex with dataset dimensionality (Section 5.3) and introduce an effective approximation to reduce that complexity (Section 5.3).

5.1 The Missing Points Region

Given constraints C and C', the *Missing Points Region* (the gray rectangles in Figure 3) is the minimum, possibly disjoint, region of points for which neither C' nor $Sky(S, C)$ can be used to infer said points' inclusion/exclusion in $Sky(S, C')$. It is comprised of those parts of $R_{C'}$ that do not overlap the dominance region of any point $s \in Sky(S, C)$, lies outside R_C, or, in unstable cases, where $\exists t \in (Sky(S, C) \cap (S_C \setminus S_{C'})), s \in (S_C \cap S_{C'})\colon t \succ s$ (Definition 5).

Definition 5. *(Missing Points Region)*
Given $Sky(S, C)$*,* C'*, the* Missing Points Region*,* MPR*, is:*

$$\begin{aligned} \mathrm{MPR} = \{p \in R_{C'} \mid (&p \in (R_{C'} \setminus (R_C \cap R_{C'})) \ \vee \\ &\exists t \in (Sky(S, C) \cap (R_C \setminus R_{C'}))\colon p \in DR(t, C)) \ \wedge \\ &\nexists u \in (Sky(S, C) \cap R_{C'})\colon p \in DR(u, C')\} \end{aligned}$$

The MPR is both *complete* and *minimal* in the sense that, with knowledge only of $Sky(S, C)$ and C', any point in MPR could be in $Sky(S, C')$ (Theorems 6 and 7, respectively).

Theorem 6. *(Completeness)*
Given $Sky(S, C)$*,* C'*, where* $R_C \cap R_{C'} \neq \emptyset$*, we have:*

$$Sky(S, C') = Sky((Sky(S, C) \cap S_{C'}) \cup (\mathrm{MPR} \cap S_{C'}), C')$$

The full proof of Theorem 6 is in Appendix 9, but the intuition is that there are only two ways in which points can be missing: (1) Expansion of C and (2) Invalidation of C. All expanded and invalidated areas are fetched unless guaranteed excluded by known points from $Sky(S, C)$. The remaining non-invalidated regions of S_C remain stable.

Theorem 7. *(Minimality)*
Given only $Sky(S, C)$*,* C'*, where* $S_C \cap S_{C'} \neq \emptyset$*, any point in* $\mathrm{MPR} \cap S_{C'}$ *could be in* $Sky(S, C')$*.*

The full proof of Theorem 7 is also in Appendix 9, but the intuition is that by definition no known point outside MPR can dominate a point inside MPR, only points inside MPR can dominate each other. Thus to minimize MPR further, we must know the contents of MPR, which we cannot do without fetching the points in MPR.

5.2 Computing the MPR

We present our algorithm to compute the MPR, which, per Theorem 7 is used to minimize points fetched. Our general approach is to start with the hyper-rectangle $\langle \underline{C}', \overline{C}' \rangle$ and continually split it using C and $Sky(S, C)$ into sub-hyper-rectangles such that many of them can be immediately discarded. At the end, we are left with a set $\mathcal{H}$ of disjoint, axis-orthogonal hyper-rectangles (i.e., range queries) covering the exact region of the MPR.

The advantage of this approach is three-fold: 1) it calculates the MPR in a form (set of range queries) that can be queried directly; 2) the primary operation, splitting axis-orthogonal hyper-rectangles with axis-orthogonal hyperplanes, is simple and efficient; and 3) the continual discarding of hyper-rectangles controls $|\mathcal{H}|$, important because the algorithm runs $\mathcal{O}(|\mathcal{H}| \cdot |Sky(S, C)| \cdot |D|)$.

In general, the algorithm consists of three steps: taking regions unknown to the cache; adding invalidated regions (in the unstable case); and removing the dominance regions of cached skyline points. Algorithm 1 presents the pseudocode (with unstable case handling omitted due to space constraints).

Lines 2–10 calculate the *overlap region*, $o = \langle \underline{o}, \overline{o} \rangle$, the area satisfying both C and C', by splitting the space into sections based on the boundary of the cache item for each dimension, eventually yielding the *overlap region* and disjoint regions around it. In the stable case o can simply be removed (Line 11); we discuss the unstable case later. Line 12 discards any hyper-rectangle $h = \langle \underline{h}, \overline{h} \rangle$ for which $\underline{h} = \overline{o}$, since h is clearly in a dominance region. After Line 12, the first of the three steps is complete, and $\mathcal{H}$ captures

Algorithm 1 MPR - Stable $Sky(S, C)$ relative to C'

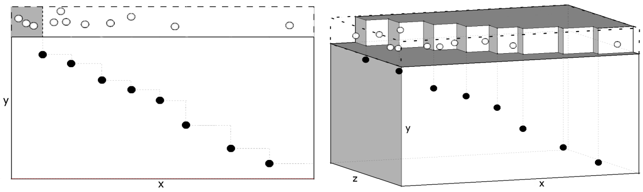
Input: $I = \langle Sky(S, C), MBR, C\rangle, C'$
Output: A set of range queries

1: $\mathcal{H} \leftarrow$ Set of hyperrectangles (Initially only $R_{C'}$)
2: **for all** dimensions $i \in D$ **do**
3:    **for all** hyperrectangles $r \in \mathcal{H}$ **do**
4:       Copy r to $r_{\le}$, $r_{\cap}$, and $r_{\ge}$
5:       Add to $r_{\le}$ constraint $p[i] \le \underline{C}[i]$
6:       Add to $r_{\cap}$ constraints $\underline{C}[i] \le p[i] \le \overline{C}[i]$
7:       Add to $r_{\ge}$ constraint $\overline{C}[i] \le p[i]$
8:       Del r from $\mathcal{H}$ and, if satisfiable, add $r_{\le}, r_{\cap}$ and $r_{\ge}$
9:    **end for**
10: **end for**
11: Remove overlap region $o = (R_C \cap R_{C'})$ from $\mathcal{H}$
12: Remove $h \in \mathcal{H}$ where $\underline{h} = \overline{o}$.
13: **for all** skyline points $u \in Sky(S, C)$ **do**
14:    **for all** dimensions $i \in D$ **do**
15:       **for all** $r \in H$ not marked with u and $DR(u) \cap r \neq \emptyset$ **do**
16:          Copy r to $r_{\le}$ and $r_{\ge}$
17:          Add to $r_{\le}$ constraint $p[i] \le u[i]$
18:          Add to $r_{\ge}$ constraint $p[i] \ge u[i]$
19:          Mark $r_{\le}$ with u and flag $r_{\ge}$ as dominated
20:          **if** $r_{\le}$ and $r_{\ge}$ are satisfiable **then**
21:             Remove r and add $r_{\le}$ and $r_{\ge}$ to $\mathcal{H}$
22:          **end if**
23:       **end for**
24:    **end for**
25:    Discard all $r \in H$ flagged as dominated
26: **end for**
27: Return H as range queries

all regions outside the cache in which missing skyline points could exist. Lines 13–26 conduct the third step, looping through each dimension of each skyline point to split the remaining $h \in \mathcal{H}$. With each split we flag one part of h as *processed* by the current point and the other as being *dominated*. The dominated part can be discarded on Line 25, and the flagging of the other part avoids unnecessarily resplitting it. For simplicity of presentation, we assumed no points lie on a range query border. This assumption can be removed by setting either inequality to be strict on Lines 5–7, 16–21. Finally, the unstable case is solved similarly; we simply run a slight modification of Algorithm 1 as a preprocessing step to determine the invalidated regions, and add those to the set $\mathcal{H}$ between Lines 11 and 12. The modification is an inversion of the logic: we want to process (not discard) the overlap region, o, and discard (not keep) the rest. We want to keep (not discard) that which overlaps dominance regions and discard (not keep) the rest. This inverted logic produces a quite small set $\mathcal{H}'$ that exactly represents the unstable, invalidated region. By adding $\mathcal{H}'$ to $\mathcal{H}$ after Line 11, it is then reduced exactly the same as the stable part of the MPR.

5.3 Approximating the MPR

In $2D$ cases (such as Figures 3 and 4a), the MPR (gray region) of the changed constraints (the dashed lines) relative to the old skyline (solid black points) is a simple rectangle. However, each new dimension adds complexity. By considering a third dimension (Figure 4b), the same 8 points and set of constraints produces an MPR consisting of 8 rectangular regions (the hollow part on top). This complexity grows for each distinct z-coordinate because the dominance region of each skyline point is (logically) subtracted from the MPR.



(a) 2D projection   (b) 3D projection

Figure 4: More dimensions = more complex dominance regions

Therefore, we introduce the *Approximate MPR* (aMPR). The aMPR is a conservative approximation of the MPR which produces no false negatives by simplifying the structure of MPR, thus creating a structure that decomposes into fewer, but larger, disjoint range queries. This in turn produces a superset of the points in MPR, thus guaranteeing completeness at a lower processing cost. The aMPR represents a middle ground approach between the minimum reads of the MPR and the maximum points read of the naive approach in [3].

The aMPR arises from a simple observation. As mentioned earlier, the complexity of the MPR comes from pruning with many multidimensional dominance regions at once. However, of all skylines points, those nearest to $\underline{C}'$ are likely to prune the most points. (This is the same intuition as for sort-based skyline algorithms [8].) So, we use only the dominance region of a small set of k nearest neighbors (NN) to $\underline{C}'$, rather than all skyline points. Algorithmically, the loop on Line 13 is replaced with the assignment, $u \leftarrow \{\mathrm{NN}_1, \ldots, \mathrm{NN_k}\}$. The optimal number of nearest neighbors to use and the trade-off presented by the approximation is evaluated experimentally in Section 7.

6. CACHE-BASED CONSTRAINED SKYLINE

With the components introduced in Sections 4-5, our Cache-Based Constrained Skyline (CBCS) method works as follows. We assume an in-memory cache with n cache items $\{I_1, \ldots, I_n\}$, organized by an R^*-tree indexing the MBR of each cached skyline. Upon receiving a query $Sky(S, C')$, we perform a search on the R^*-tree fetching all cache items where $R_{C'} \cap \mathrm{MBR} \neq \emptyset$. If none exist, $Sky(S, C')$ is computed naively. If more than one cache item is returned, we select the most efficient based on a cache search strategy (Section 6.1). We then compute the MPR as per Section 5. Finally we fetch the points in the MPR, merge them with the cached $Sky(S, C)$, and compute $Sky(S, C')$.

6.1 Cache search strategies

A cache search strategy takes m query-overlapping cache items $\{I_1, \ldots, I_m\}$ as input and aims to return the cache item most efficient for computation of the query. We suggest several cache search strategies, which we will compare experimentally in Section 7. *Random* chooses a cache item uniformly at random among the m overlapping ones. *MaxOverlap* chooses the cache item with the highest degree of overlap with the query region. *MaxOverlapSP* functions as MaxOverlap, except it prefers cache items whose skyline $Sky(S, C)$ is stable relative to C' even if there is an unstable option with a higher degree of overlap. *Prioritized1D* gives preference to simple cases of single changes (as in Sect. 4.2) as follows: Case 2, Case 3, Case 1, General case stable (i.e. not 1D), Case 4 and General case unstable. Ties are broken using MaxOverlap. The case prioritizes were chosen by experimental evaluation. *PrioritizednD(C1,C2,C3,C4)* generalizes this case-prioritization idea, by independently scoring the four cases (i.e., $C1 \dots C4$) and penalizing cache items for each dimension where constraints differ from the queried. Initial experiments have shown *PrioritizednD(10,0,5,20)* performs well, thus we use it as *PrioritizednD (Std)*. To demonstrate the importance of proper priorities, we also include a variant *PrioritizednD(10,50,30,0)* denoted *PrioritizednD (Bad)*. Finally *OptimumDistance* chooses the cache item whose lower constraint corner is closest to the lower corner of the queried constraints, to give priority to likely dominating regions.

6.2 Cache replacement & dynamic data

Common cache replacement strategies (i.e., LRU, LCU) are supported by insertion and use counters on the R^* tree. Dynamic data can be supported by viewing each cache item as a separate dataset with a continuous skyline query maintained by any existing method (e.g. [13, 19, 21]). Due to space constraints, evaluation of cache replacement strategies and dynamic data are omitted from Section 7 and left for future work.

6.3 Multiple cache items

As an extension to the work presented in this paper, one might consider exploiting more than one overlapping cache item during processing. Such a strategy could be beneficial given the increased pruning ability from two cache items. However due to the added complexity, more range queries would be generated, cache search strategies would become more complicated and the number of different overlap cases would require elaborate specialized processing methods. These added complexities merit a separate research effort into such a method and thus we leave this for future work.

7. EXPERIMENTAL EVALUATION

In this section, we provide an extensive experimental evaluation of our CBCS method, investigating scalability, the effectiveness of the approximate MPR, and the cache search strategies. We experimentally compare CBCS to the existing *BBS* method for computing constrained skylines, as well as a *Baseline* method that fetches all points in S_C with a range query and applies a standard skyline algorithm (as suggested in [3]). We use the Sort-Filter Skyline (SFS) [8] algorithm in both the *Baseline* method and our own *CBCS* method. While more complex skyline algorithms, e.g., BSkyTree [16], might produce faster overall runtimes, our contribution is orthogonal in that the benefit of our *CBCS* method is independent of the skyline algorithm used, as we show in Section 7.3.

Experiments are performed on Linux with kernel 3.2.0-61-generic, an Intel Core 2 Quad Q8400 2.66 Ghz CPU and 8 GB memory. All algorithms are implemented in Java, using a publicly available Java-based R-tree implementation [2]. Data is stored in PostgreSQL 9.1.13 with each dimension indexed by a standard B-tree. The cache is implemented as a simple in-memory cache, organized through an R^*-tree that indexes the MBR for each cache item.

All methods are evaluated in separation with the DBMS restarted between runs for fair comparison. In preliminary experiments, we also tested a baseline using sequential scan, but it was consistently slower than the baseline using the indexes; so, we omit it for space.

We evaluate with synthetic data by generating independent, correlated and anti-correlated data using the standard generator from [3]. For real data we use a Danish real estate dataset covering almost 4.2 million properties in Denmark as of 2005. The full 2005 dataset is not publicly available but the current 2013 version can be browsed online [3].

[2]http://libspatialindex.github.com
[3]https://www.ois.dk/

7.1 Query workload generation

Existing constrained skyline work does not study sets of queries, but only single queries. We therefore construct a query generator mimicking interactive search patterns as studied also in relation to constrained skyline queries earlier [6, 17]. The generator chooses an initial set of constraints for each $i \in D$ with $\underline{C}[i]$ and $\overline{C}[i]$ set randomly between 0 and 3 standard deviations from the mean of dimension i, modeling that, for example, average-sized houses are most likely to be searched. Subsequent constraint changes are modeled as follows: 1) The dimension to vary is chosen randomly; 2) whether to increase/decrease lower/upper constraints is chosen at random; and 3) a new query is generated from the old, with a $5\% - 10\%$ change in the chosen dimension and direction. Step 3) is repeated $1 - 10$ times to mimic an interactive scenario with one user posing several similar queries. All steps are repeated until the desired number of queries has been generated.

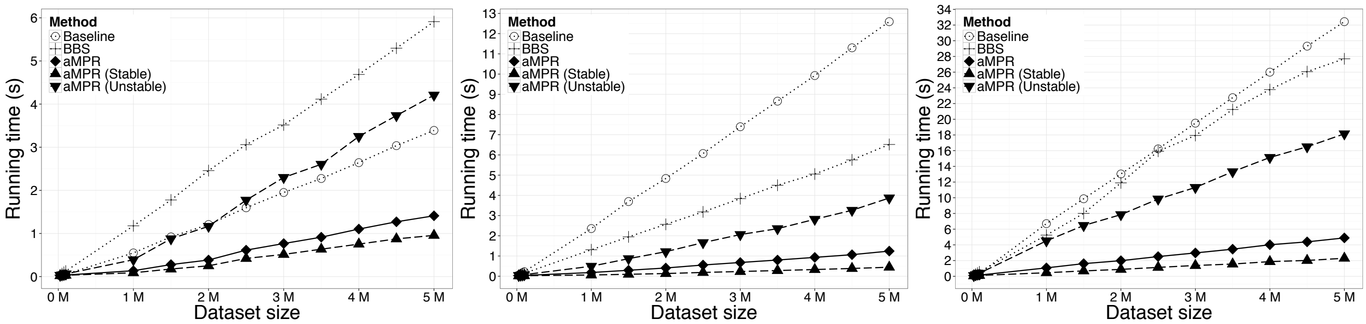
We evaluate all methods with two different query workloads: (1) the aforementioned *Interactive exploratory search* and (2) *Independent* queries in a multi-user system. Workload (1) assumes an empty cache and uses the generator to create 5 independent sets of 100 queries mimicking 5×100 actions in an interactive exploratory search. Workload (2) assumes a preloaded cache with 2000 queries, where we receive a number of new independent single queries each generated like the initial query in the generator.

Unless stated otherwise locally, the cache search strategies used are *MaxOverlapSP* for interactive exploratory search queries and *PrioritizednD (Std)* for independent queries.

7.2 Interactive Search - Dataset size & Dimensionality

Figures 5a-5c show the average running time of our *CBCS* using *aMPR* compared to *BBS* and *Baseline*, for increasing dataset size on $5D$ data. Initial experiments showed using 1 NN for *aMPR* gave the most consistently good results for interactive search scenarios. *CBCS* average performance is labeled *aMPR*, further broken down into *aMPR (Stable)* and *aMPR (Unstable)* for performance on stable/unstable cases respectively. Results are averaged over the same 5×100 interactive exploratory search queries. For distributions we see that all methods scale approximately linearly. This is expected since the same range query will require a linear amount of extra processing for each increase in dataset size, regardless of the resulting constrained skyline. By comparing *Baseline* to the *CBCS* methods, we also see that we scale significantly better than the *Baseline* on average for all distributions, especially when the cached skyline is stable relative to the queried constraints in *aMPR (Stable)*. In these cases, a partial skyline result requires fetching only a small subset of what *Baseline* fetches. Thus we both read fewer points and conduct fewer dominance tests. However, while *aMPR* scales well on average and the stable cases in *aMPR (Stable)* scale very well, the unstable cases in *aMPR (Unstable)* fare less well. As discussed in Section 4.2, instability can cause recomputation if a cached skyline point falls outside of the new constraints. Still, the only case in which the *aMPR (Unstable)* does not outscale *Baseline* is on independent data with $\geq 2M$ points.

Interestingly, *BBS* performs worse than *Baseline* in several cases and consistently for independent data. This is most likely due to the overhead in R-tree queries when few entire regions are pruned or included in the skyline. As a final note, observe the scales in Figures 5a-5c differ and that, perhaps surprisingly, correlated data is more of a challenge for the methods than independent data. Broadly speaking independent data is evenly distributed and correlated data is grouped in sections of the dataspace. The same queries that returned a given number of datapoints for the independent data



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

Figure 5: Scalability with increasing dataset size for interactive exploratory search queries ($|D| = 5$)

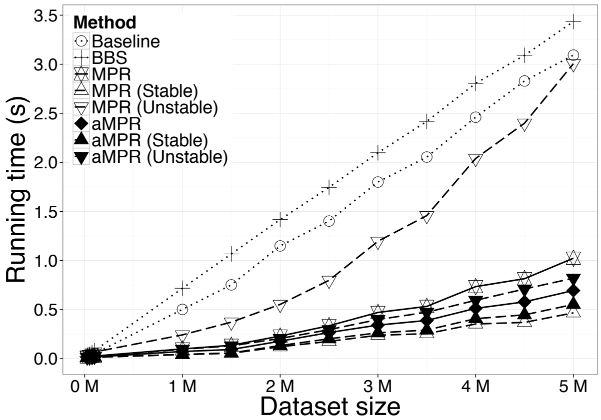


Figure 6: Scalability with increasing dataset size for interactive exploratory search queries (Independent, $|D| = 3$)

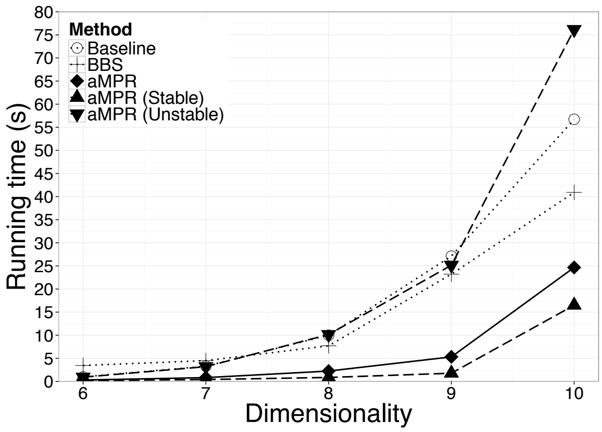


Figure 7: Efficiency with increasing dimensionality ($|S| = 1M$, $|D| > 5$)

can thus return significantly more for correlated data, if the queries happen to cover where the data is most concentrated. This is exactly what we see here, since the average number of points read is 7 times higher for correlated data than for independent data. Note that despite this, the performance of each method is not reduced 7 times, because the computation of the skyline on corrrelated data points is much faster.

Figure 6 shows the same experiment as in Figures 5a-5c but for a 3-, rather than 5-, dimensional independent dataset. We include the exact *MPR* with a stable/unstable split as with *aMPR*. Just as in Figure 5a, we see that *BBS*, *Baseline* and *aMPR* all scale linearly. However while *BBS* performs better, the *Baseline* is still faster for independent data, since the increased efficiency of the R-tree in $|D| = 3$ is equally matched by the benefit of simpler dominance tests in the *Baseline*. The *aMPR* method remains superior to the *Baseline* as in Figure 5a, but due to the decreased dimensionality even unstable cached skylines in *aMPR (Unstable)* are scaling well. Finally the use of the exact *MPR* rather than the *aMPR* means stable cache items yield superior results since *MPR* prunes more of the search space than *aMPR*. However while the same superior pruning applies to unstable cache items, the *MPR* method is significantly slower than the *aMPR*, since cache invalidation yields a prohibitive amount of range queries with subsequent random access latency for *MPR*. We will discuss a further breakdown of these performance factors for *aMPR* and *MPR* in Section 7.3. Finally we observe scalability with regard to $|D|$ in Figure 7. Note that, unlike unconstrained skyline queries, fixing the dimensionality in constrained skylines has some important implications. Depending on the constraints, adding a dimension may in fact increase the efficiency of a constrained skyline query by reducing the input size. In order to avoid such arbitrary effects, we expand the queries from Figures 5a-5c by adding an unconstrained dimension to each query for each dimension over 5. The dimensionality results for $|D| = 8$ are thus constrained on 5 dimensions and unconstrained on 3.

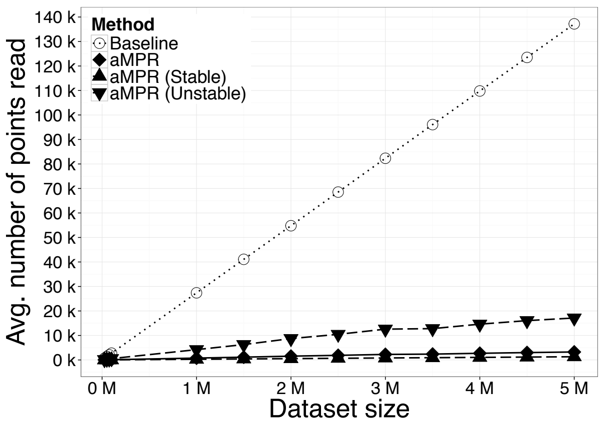
As expected, all methods deteriorate exponentially with $|D|$ as the skyline size increases. For *BBS*, the performance of the underlying R-tree degrades and for the *aMPR*, the number of range queries generated increases (see Section 7.3).

7.3 CBCS performance breakdown

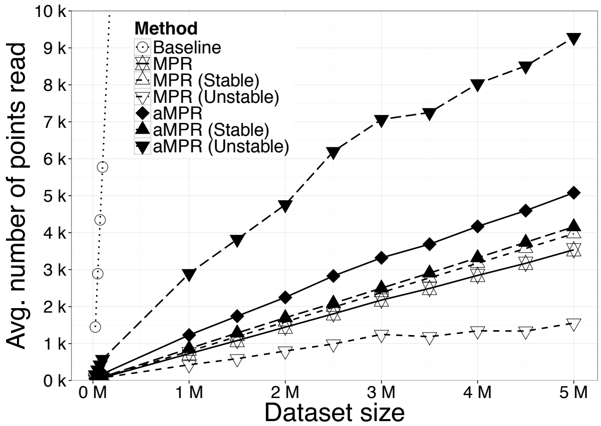
We further analyze *CBCS* by investigating 3 key factors: the number of points fetched, the number of range queries issued, and the types of constraint changes.

7.3.1 Number of points read from disk

Figure 8a shows points read by *Baseline* and *aMPR* for the experiment from Figure 5a. The number of points read by *Baseline* increases significantly with dataset size, while the increase for *aMPR* is limited except for the unstable cases in *aMPR (Unstable)*. This is key to the performance of *aMPR* since the number of points read is primarily influenced by the difference in cardinality between sets

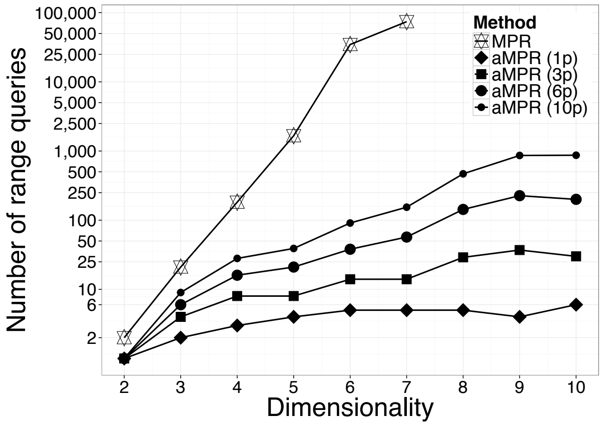


(a) $|D| = 5$

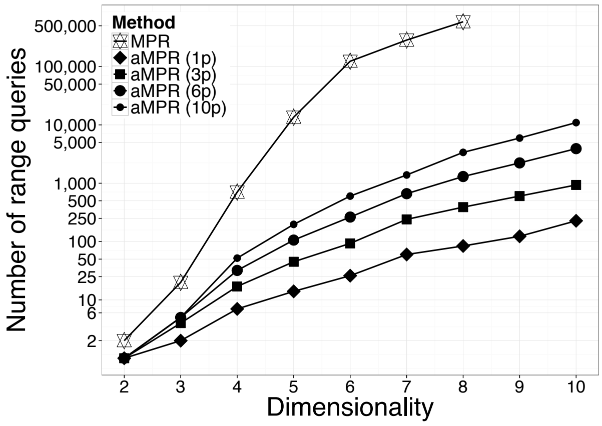


(b) $|D| = 3$

Figure 8: Avg. number of points read (Independent, $|S| = 1M$)



(a) Interactive exploratory search queries



(b) Independent queries

Figure 9: Avg. number of range queries generated (Independent, $|S| = 5k$)

S_C and $S_{C'}$, rather than the actual size of each set. The larger increase for *aMPR (Unstable)* arises both from the likelihood of cached skyline points being outside the queried constraints and increasing the number of new points included due to invalidation inside the cache item.

Figure 8b which shows the average number of points read from disk for *Baseline*, *aMPR* and *MPR* in comparison, corresponding to Figure 6. The same pattern as in Figure 8a can be seen for *Baseline* relative to *aMPR*, while *MPR* consistently reads fewer points than *aMPR* from disk, since it is the minimal set. While an unstable cached skyline yields relatively many reads for *aMPR*, the exact opposite is the case for *MPR*. This illustrates that while *computing* the exact invalidation of a cache item is computationally expensive, the *extent* of invalidation is limited.

7.3.2 Number of range queries generated

Figure 9 shows the average number of range queries for *MPR* and *aMPR* with 1,3,6 and 10 NNs on interactive (Figure 9a) and independent (Figure 9b) workloads. Both graphs use logarithmic scales and the dataset is limited to $5k$ points so that we can scale *MPR* to higher dimensions. Figure 9a reveals that the exact *MPR* rapidly generates extra queries as $|D|$ increases, e.g., a $6D$ query/cache item pair generates almost $50k$ disjoint range queries to cover the MPR. This number would be even higher for $> 5K$ points. For the *aMPR*, the reduced hyperplane splitting, while increasing the number of points read, greatly decreases the amount of queries generated dependent on the amount of NNs used. Note that we did not include results for *MPR* for dimensionalities 8,9 and 10, since just generating the range queries here took several hours for each query. Thus the approximation really improves scalability as $|D|$ increases.

Figure 9b confirms these trends for independent queries as well, but both methods generate more queries and the increase is more rapid, because the queries generated overlap less in higher dimensions. Observe that the number of NNs can be used to manipulate the tradeoff between reading few points from disk and decreasing random access. While large quantities of range queries seem problematic, they do not necessarily deteriorate the performance of the method. Considering e.g. Figure 9b for $|D| = 4$ and $\#NN = 10$, on average ≈ 61 queries were generated for *aMPR*, but the number actually reading data only averaged 33. The remaining queries were discarded by the DBMS without any disk seeks because the B-trees detect the empty queries. For $|D| = 10$ and $\#NN = 10$ these numbers increase to 13353 queries generated of which only 114 read data from disk.

Note that with only $5K$, no stable cases were generated for $|D|$>4. From Theorem 1, this is not surprising, since just one dimension i where $\underline{C}'[i] > \underline{C}[i]$ causes instability of $Sky(S, C)$ relative to C'.

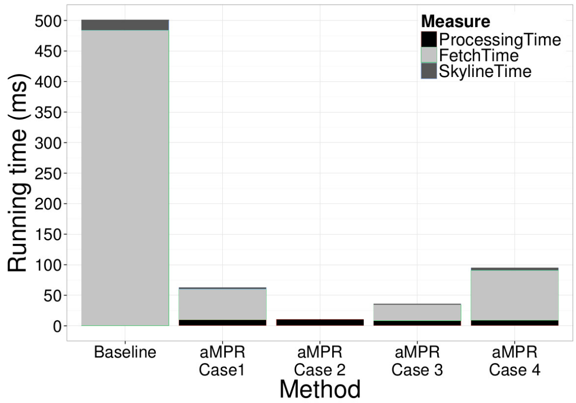
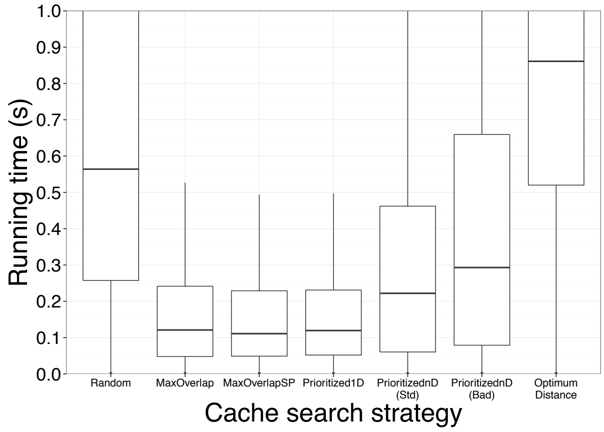
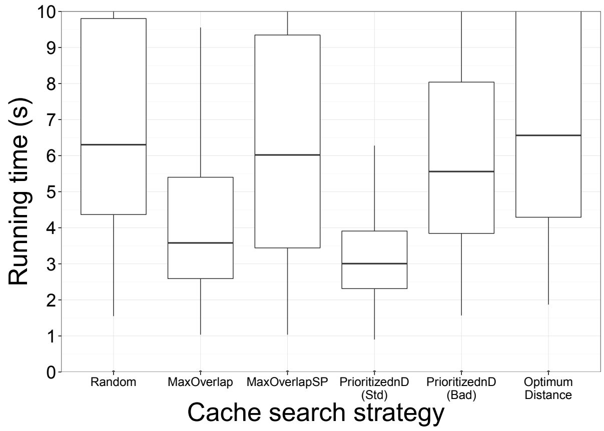


Figure 10: Avg. ms per stage (Independent, $|S| = 1M$, $|D| = 3$)



(a) Interactive exploratory search queries



(b) Independent queries

Figure 11: Cache search strategies ($|S| = 1M$, $|D| = 5$)

So, for independent queries, *CBCS* methods work best for lower dimensional settings, and shows most benefit for exploratory queries.

7.3.3 Types of constraint changes

Figure 10 breaks down computation for $1M$ points (settings as in Figure 6) into 3 stages: processing, fetching, and skyline computation, corresponding to the main-memory selection of range queries, the latency to read points from disk, and the running of SFS, respectively. *Baseline* has no processing stage, but suffers long fetching. Conversely, *aMPR* has light processing, which reduces fetching and then, having fewer input points, skyline computation. Considering specific types of changes, *aMPR Case 2* has no fetching stage or computation stage, since this is a simple case which only requires removing cached skyline points. *aMPR Case 3* shows a slight processing stage followed by a significantly smaller fetching stage than both *Baseline* and *aMPR Case 1* since we are able to prune the search space significantly using cached skyline points. Note that while the relative gains of the fetching stage from *aMPR Case 1* to *aMPR Case 3* are only half, a larger portion of points is pruned, with random access being more time consuming.

To conclude, we see that the superior performance of *aMPR* and *MPR* arises primarily from the reduced reads from disk, which reduces both fetching and skyline computation. Also, the performance is independent of the skyline algorithm used, since this is anyway not a bottleneck. Finally we see that the *MPR* requires too many range queries for mid- to high-dimensional data, but the *aMPR* generates a small, stable number of range queries independent of the dataset size.

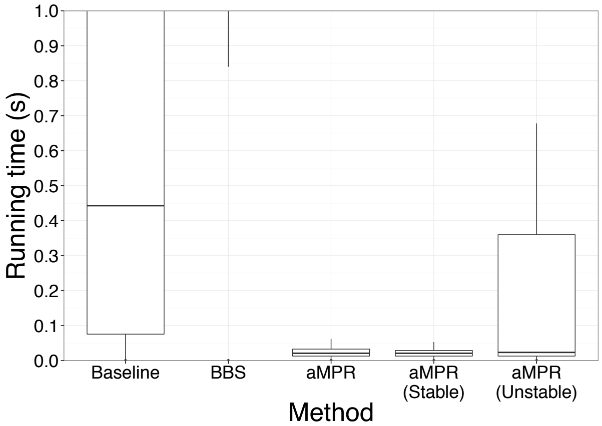
7.4 Cache search strategies

Our last synthetic experiment shows the distribution of response times for each proposed cache search strategy from Section 6.1, using *aMPR* on 5×100 interactive queries (Figure 11a) and 500 independent queries (Figure 11b).

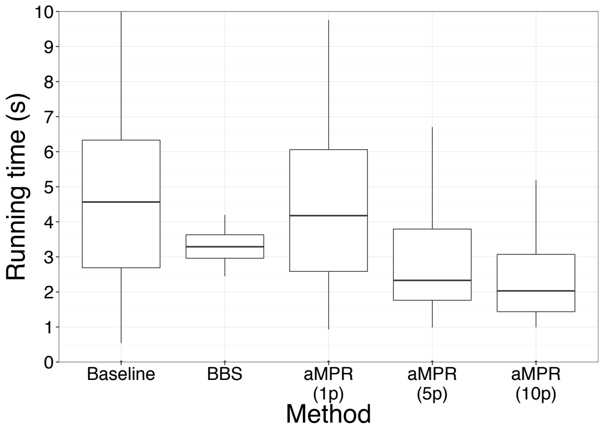
Compared to the *Random* strategy we can see there is a clear benefit in using overlap as a guiding factor (observe *MaxOverlap*, *MaxOverlapSP* and in part *Prioritized1D* which uses *MaxOverlap* to settle ties). High overlap yields smaller MPRs, especially in stable cases. On the other hand, prioritizing only stable cases is not a good strategy for independent queries, as is clear from *MaxOverlapSP*: such queries are likely to vary in several dimensions such that choosing solely on stability may select an item with poor overlap or many changed dimensions. Instead a balanced ranking-based approach like *PrioritizednD (Std)* is most promising, since it not only considers stability but also case complexity. *PrioritizednD (Bad)* shows poor performance demonstrating that the case-based scoring is important for performance. Finally, *OptimumDistance* performs poorly: considering only closeness in terms of dominance fails to capture the complexity of cases and overlap.

7.5 Real data

We study real data covering ~ 4.2 million properties in Denmark as of 2005 using 4-dimensions suitable for constrained skyline computation: *year* (year of construction), *sqrm* (size in m^2), *valuation* (property tax valuation) and *price* (actual sales price). The final dataset size is $1.28M$ records after removing records with missing data. Figures 12a and 12b respectively show the distribution of response times for 10×100 exploratory search queries and 50 independent queries. Figure 12a shows our *aMPR* method is superior to both *Baseline* and *BBS*, with *BBS* managing about 2.2 seconds on average per query and *Baseline* performingly significantly better at about 0.45 seconds. Note the average response time of *aMPR (Unstable)* is actually low due to limited invalidation, while the worst case invalidation yields response times above the average for *Baseline*. Figure 12b shows a set of independently generated queries. Here *Baseline* varies heavily in performance due to varying query selectivities, while *BBS* is stable with the changing constraints. The remaining 3 plots show *aMPR* with varying numbers of NNs. The number of NNs chosen in this case has a large impact on performance, since using only 1 as with the ex-



(a) Interactive exploratory search queries



(b) Independent queries

Figure 12: Performance on Danish property data ($|S| = 1.28M$, $|D| = 4$)

ploratory queries yields poorer results than *BBS*, while 5 NNs or 10 NNs greatly outperforms it. Using more than 10 NNs however did not provide any significant further benefit in this case.

In conclusion, the major performance factors are the number of disk reads performed and the degree of random access due to multiple range queries. We have shown that *CBCS* performs substantially better on related queries than existing methods. The performance benefit remains stable for increasing dataset size and dimensionality, when using approximate *aMPR*. The cache search strategy is a clear determinator of the resulting performance with the main factors being *average overlap*, *stability* and *case distributions*.

8. CONCLUSION

In this paper we introduced a novel method for computing constrained skylines using an in-memory cache. Our method was envisioned under two common types of query workloads with related queries, namely interactive search and multi-user systems. We determined four possibilities for incremental query/cache overlap, analyzed them and presented specialized techniques for each. For general query/cache overlap, we introduced the *Missing Points Region*, which minimizes points read from disk by exploiting the cache item's relation to the query. To increase the practical performance of this general method, we introduced a conservative approximation of the MPR, called aMPR, to balance the trade-off between resultant range queries and points read from disk. Finally we introduced a set of heuristics to choose the most efficient overlapping cache item for a query. Our extensive experimental evaluation revealed, among other things, that the choice of cache item to use for processing has a big impact on performance and that our method significantly outperforms existing approaches when related queries are present.

9. ACKNOWLEDGMENTS

This research was supported in part by the Danish Council for Strategic Research, grant 10-092316.

10. REFERENCES

[1] K. M. Banafaa and R. Li. Efficient algorithms for constrained subspace skyline query in structured peer-to-peer systems. In *Proc. WAIM*, 2012.

[2] A. Bhattacharya, B. P. Teja, and S. Dutta. Caching stars in the sky: a semantic caching approach to accelerate skyline queries. In *Proc. DEXA*, 2011.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline operator. In *Proc. ICDE*, 2001.

[4] S. Chaudhuri, N. Dalvi, and R. Kaushik. Robust cardinality and cost estimation for skyline operator. In *Proc. ICDE*, 2006.

[5] L. Chen, B. Cui, L. Xu, and H. Shen. Distributed cache indexing for efficient subspace skyline computation in p2p networks. In *Proc. DASFAA*. 2010.

[6] S. Chester, M. L. Mortensen, and I. Assent. On the suitability of skylines queries for data exploration. In *Proc. ExploreDB*, 2014.

[7] S. Chester, D. Sidlauskas, I. Assent, and K. Bøgh. Scalable parallelization of skyline computation for multi-core architectures. In *Proc. ICDE*, 2015.

[8] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting: Theory and optimizations. In *Intelligent Information Systems*, 2005.

[9] B. Cui, H. Lu, Q. Xu, L. Chen, Y. Dai, and Y. Zhou. Parallel distributed processing of constrained skyline queries by filtering. In *Proc. ICDE*, 2008.

[10] E. Dellis, A. Vlachou, I. Vladimirskiy, B. Seeger, and Y. Theodoridis. Constrained subspace skyline computation. In *Proc. CIKM*, 2006.

[11] M. Endres and W. Kiessling. Semi-skyline optimization of constrained skyline queries. In *Proc. ADC*, 2011.

[12] K. Fotiadou and E. Pitoura. BITPEER: continuous subspace skyline computation with distributed bitmap indexes. In *Proc. DAMAP*, 2008.

[13] Y.-L. Hsueh, R. Zimmermann, and W.-S. Ku. Efficient updates for continuous skyline computations. In *Proc. DEXA*, 2008.

[14] Y.-L. Hsueh, R. Zimmermann, and W.-S. Ku. Caching support for skyline query processing with partially-ordered domains. In *Proc. SIGSPATIAL*, 2012.

[15] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proc. VLDB*, 2002.

[16] J. Lee and S.-w. Hwang. Scalable skyline computation using a balanced pivot selection technique. *Inf. Syst.*, 39, 2014.

[17] M. Magnani, I. Assent, K. Hornbæk, M. R. Jakobsen, and K. F. Larsen. SkyView: A user evaluation of the skyline

operator. In *Proc. CIKM*, 2013.

[18] Y. Mass, M. Ramanath, Y. Sagiv, and G. Weikum. Iq: The case for iterative querying for knowledge. In *Proc. CIDR*, 2011.

[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30, 2005.

[20] D. Sacharidis, P. Bouros, and T. Sellis. Caching dynamic skyline queries. In *Proc. SSDBM*, 2008.

[21] P. Wu, D. Agrawal, O. Egecioglu, and A. El Abbadi. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In *Proc. ICDE*, 2007.

[22] L. Zhang, Y. Jia, and P. Zou. A grid index based method for continuous constrained skyline query over data stream. In *Advances in Web and Network Technologies, and Information Management*. 2009.

[23] S. Zhang, N. Mamoulis, and D. W. Cheung. Scalable skyline computation using object-based space partitioning. In *Proc. SIGMOD*, 2009.

APPENDIX

Proof of Theorem 1. Consider left [L] and right [R] sides of the OR expression. From [R] we have $(\exists i \in D\colon \underline{C}'[i] > \overline{C}[i] \vee \overline{C}'[i] < \underline{C}[i]) \implies R_C \cap R_{C'} = \emptyset$, thus $Sky(S, C)$ is stable in this case. For [L] we prove $(\forall i \in D\colon \underline{C}'[i] \leq \underline{C}[i])$ implies stability, by contradiction. Assume $\exists s \in Sky(S, C')\colon (s \in S_C) \wedge (s \notin Sky(S, C))$. From Def 1, this implies $\exists t \in S_C\colon t \succ s$. This in turn means $s \in Sky(S, C') \implies t \notin S_{C'} \implies \exists i \in D\colon \overline{C}'[i] < t[i] \leq s[i] \leq \overline{C}[i]$, i.e. t and s both do not satisfy constraints C' and thus $s \notin Sky(S, C')$ contradicting the assumption. Observe that [L] and [R] are the only situations with guaranteed stability, since cases not satisfying Thm 1 must have $R_C \cap R_{C'} \neq \emptyset$ and $\exists i \in D\colon \underline{C}[i] < \underline{C}'[i] \leq \overline{C}[i]$. Thus for $u \in Sky(S, C)$, $\underline{C}[i] \leq u[i] < \underline{C}'[i] \leq \overline{C}[i]$ we could have $v \in S_C$, $u \succ v$ and $\underline{C}[i] \leq u[i] < \underline{C}'[i] \leq v[i] \leq \overline{C}[i]$. Since $u \notin S_{C'}$ we might then have $v \in Sky(S, C')$, making $Sky(S, C)$ unstable relative to C'. □

Proof of Theorem 2. Since $Sky(S, C)$ is stable relative to C' given Thm 1, equality follows from Cor 1. Minimality holds since $\nexists s \in Sky(S, C), t \in S_{\Delta C}\colon s \succ t$. □

Proof of Theorem 3. Observe $S_{C'} \subset S_C$ and that $Sky(S, C)$ is stable relative to C' given Thm 1. Thus we must have $Sky(S, C') \subseteq Sky(S, C)$ and $Sky(S, C') = Sky(S, C) \cap S_{C'}$ follows. Minimality holds since the reduction simply removes cached skyline points and no further reads are necessary. □

Proof of Theorem 4. Observe $Sky(S, C)$ is stable relative to C' given Thm 1. Thus equality follows from Cor 1 since for $s \in Sky(S, C')$ we either have $s \in Sky(S, C)$ or $s \in S_{\Delta C}$, where $\nexists t \in Sky(S, C)\colon t \succ s$.

Minimality holds since $\forall u \in Sky(S, C), v \in (Sky(S, C) \cup \{s \in S_{\Delta C} \mid \nexists t \in Sky(S, C)\colon t \succ s\}\colon u \not\succ v)$, i.e. we have no further known points to prune $S_{\Delta C}$ with. □

Proof of Theorem 5. Given Thm 1, $Sky(S, C)$ may be unstable relative to C' and we observe $S_{C'} \subset S_C$. At this point we have two possibilities given Cor 2: [St] $Sky(S, C)$ is stable relative to C', or [Ust] $Sky(S, C)$ is unstable relative to C'. If case [St] holds, then $Sky(S, C') = Sky(S, C) \cap S_{C'}$ since there is no invalidation. If case [Ust] holds, then from Cor 2 we have $\exists t \in Sky(S, C)\colon t \in S_{\Delta C} \wedge \exists s \in (S_C \cap S_{C'})\colon t \succ s \wedge \nexists u \in (Sky(S, C) \cap S_{C'})\colon u \succ s$, i.e. there exists a removed skyline point which has invalidated part of the cache. Since [St] and [Ust] correspond to the two unioned skyline inputs in Thm 5, completeness holds by virtue of Def 1.

To observe that minimality holds, we first note that from Def 1 we must have $\nexists s \in Sky(S, C), t \in (Sky(S, C) \cap S_{\Delta C})\colon s \succ t \implies \forall s \in (Sky(S, C) \cap S_{C'})\colon s \in Sky(S, C')$, i.e. a removed skyline point cannot dominate a remaining skyline point. Thus minimality holds since only $\{s \in (S_C \cap S_{C'}) \mid \exists t \in (Sky(S, C) \cap S_{\Delta C})\colon t \succ s\}$ is affected by instability given Cor 2 and only $Sky(S, C) \cap S_{C'}$ is available for pruning. □

Proof of Theorem 6. We prove equality in right [R] and left [L] directions. For [R] we assume $w \in Sky(S, C')$. We then have two cases: [1] $w \in (R_C \cap R_{C'})$ and [2] $w \in (R_{C'} \setminus (R_C \cap R_{C'}))$, i.e. w is either in the overlapping region between cache and query ([1]) or outside the cache ([2]).

If we have case [1], then given Cor 2 we have $w \in (Sky(S, C) \cap R_{C'}) \vee (\exists t \in (Sky(S, C) \cap (R_C \setminus R_{C'}))\colon w \in DR(t, C) \wedge \nexists u \in (Sky(S, C) \cap R_{C'})\colon w \in DR(u, C'))$, i.e. w is a cached skyline point or a point included due to invalidation. If instead case [2] holds, then we simply have $\nexists u \in S_{C'}\colon u \succ w$ due to Def 1.

Combined we thus have $(w \in (Sky(S, C) \cap R_{C'})) \vee (w \in \{p \in R_{C'} \mid (p \in (R_{C'} \setminus (R_C \cap R_{C'})) \vee \exists t \in (Sky(S, C) \cap (R_C \setminus R_{C'}))\colon p \in DR(t, C)) \wedge \nexists u \in (Sky(S, C) \cap R_{C'})\colon p \in DR(u, C')\})$ which is equivalent to $w \in Sky((Sky(S, C) \cap S_{C'}) \cup (MPR \cap S_{C'}), C')$ by virtue of Def 1.

For the opposite direction, [L], we assume $w \in Sky((Sky(S, C) \cap S_{C'}) \cup (MPR \cap S_{C'}), C')$. We again have two cases: [St] $Sky(S, C)$ is stable relative to C', and [Ust] $Sky(S, C)$ is unstable relative to C'.

If we have case [St], then given Cor 1 we have $(w \in (Sky(S, C) \cap S_{C'})) \veebar (w \in (R_{C'} \setminus R_C))$, i.e. w is either a cached skyline point or outside the cache.

If instead we have case [Ust], then given Cor 1 and 2, we have $((w \in (Sky(S, C) \cap S_{C'})) \veebar (w \in (R_{C'} \setminus R_C))) \vee (w \in (R_C \cap R_{C'}) \wedge (\exists t \in (Sky(S, C) \cap (R_C \setminus R_{C'}))\colon w \in DR(t, C)) \wedge (\nexists u \in (Sky(S, C) \cap R_{C'}))\colon w \in DR(u, C'))$, i.e. w is either a cached skyline point, a point outside the cache or a point included due to invalidation.

Now observe that the region $R_{C'}$ can be expressed as $R_{C'} = (R_C \cap R_{C'}) \cup (R_{C'} \setminus (R_C \cap R_{C'}))$. We now prove by contradiction that any w satisfying the right hand side of Thm 6 must also be in $Sky(S, C')$. So we assume $w \notin Sky(S, C') \implies \exists v \in Sky(S, C')\colon w \in DR(v, C')$ given Def 1. We then have two cases [I] $v \in (R_C \cap R_{C'})$ and [O] $v \in (R_{C'} \setminus (R_C \cap R_{C'}))$.

If [I] then $v \in Sky(S, C) \vee \exists t \in (Sky(S, C) \cap (R_C \setminus R_{C'}))\colon v \in DR(t, C))$ which implies $w \notin Sky((Sky(S, C) \cap S_{C'}) \cup (MPR \cap S_{C'}), C')$ given Def 1 yielding a contradiction.

If [O] then $v \notin (R_C \cap R_{C'}) \implies v \notin (Sky(S, C) \cap R_{C'}) \implies \nexists u \in (Sky(S, C) \cap R_{C'})\colon u \succ v \implies v \in MPR$ given Def 5, also yielding a contradiction. Hence $w \in Sky(S, C')$. □

Proof of Theorem 7. Proof by contradiction. Assume $p \in MPR$ and $p \notin Sky(S, C')$ can be guaranteed. From Def 1, $p \notin Sky(S, C') \implies \exists t \in Sky(S, C')\ :\ t \succ p$. Observe that t must be known before fetching for us to prune with it, thus $t \succ p \implies t \in (Sky(S, C) \cap R_{C'}) \implies t \in Sky(S, C)$, i.e. t is part of the cached skyline. By the definition of MPR (Def 5), this contradicts our assumption since all $p \in R_{C'}$ where $\exists u \in (Sky(S, C) \cap R_{C'})\colon p \in DR(u, C')$ are excluded from MPR. Thus the MPR is minimal. □