

Hashcube: A Data Structure for Space- and Query-Efficient Skycube Compression

Kenneth S. Bøgh, Sean Chester, Darius Šidlauskas, and Ira Assent
Aarhus University, Åbogade 34, Aarhus N, Denmark 8200
{ksb,schester,dariuss,ira}@cs.au.dk

ABSTRACT

The skyline operator returns records in a dataset that provide optimal trade-offs of multiple dimensions. It is an expensive operator whose query performance can greatly benefit from materialization. However, a skyline can be executed over any subspace of dimensions, and the materialization of all subspace skylines, called the skycube, dramatically multiplies data size. Existing methods for skycube compression sacrifice too much query performance; so, we present a novel hashing- and bitstring-based compressed data structure that supports orders of magnitude faster query performance.

Categories and Subject Descriptors

H.2.4 [Database Management]: Systems—*query processing*

Keywords

skycube, compression, hashmap, data structure

1. INTRODUCTION

The *skyline* operator [1] selects from a database *all* tuples that are not clearly less interesting than *any* others. For example, Table 1 lists some top movies from IMDB. Whether one is interested in movies that are newer, higher-rated, or higher-grossing, or any combination of these attributes, *Titanic* is still less interesting than *Avatar*: the latter has higher values on every attribute than the former. By contrast, *The Shawshank Redemption* is older and lower-grossing than *Avatar*, but still interesting for its high rating.

The skyline includes all data points that are strictly higher on at least one attribute or equal on every attribute, when compared to all other points (like *The Shawshank Redemption* but not *Titanic*). These are the most interesting points.

Subspace skylines Often, it is advantageous for a user to pose a skyline query on only the few attributes that are relevant to him/her: a typical moviegoer is unconcerned with

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM '14 Shanghai, China

Copyright 2014 ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

Movie Title	Year	Rating	Sales ($\times 10^6$)
Avatar	2009	7.9	2784 USD
The Avengers	2012	8.2	1514 USD
The Godfather	1972	9.2	245 USD
The Shawshank Redemption	1994	9.2	59 USD
Skyfall	2012	7.8	1108 USD
Titanic	1997	7.6	2186 USD

Table 1: Some top movies, courtesy IMDB.com.

a movie's sales figures, so is better served by the skyline on just the *year* and *rating* attributes. On the other hand, a studio accountant may have a very different perspective.

A subspace skyline query [6,9,10] returns the skyline computed over a subset of attributes specified by the user, personalizing the result. However, it is nearly as expensive to compute skylines in arbitrary subspaces as the full dimensionality, a cost amplified when users pose a series of queries in different subspaces (such as in exploratory scenarios [2]).

Skycube To offer the *best possible* response time for a subspace skyline query, one solution is to precompute the answer. To do so for every possible subspace skyline is to construct the *skycube* [6,11], a set of $2^d - 1$ subspace skylines. However, storage of the skycube is quite large. Although compressed skycube data structures exist [8,10], query performance on the state-of-the-art structure is inadequate.

Therefore, we introduce Hashcube to compress a skycube with bitstrings and hash maps. It achieves an order of magnitude compression over the default structure, while providing query performance 1000 \times faster than state-of-the-art.

2. BACKGROUND AND RELATED WORK

We assume a table P of n records, each described by d ordinal attributes. We denote the i 'th record by p_i and the j 'th attribute of p_i by $p_i[j]$. Our approach is based on bitstrings (fixed-length sequences of binary values).¹ We denote the j 'th bit of a bitstring B_i by $B_i[j]$ and the substring of B_i from bit j to k , inclusive, by $B_i[j, k]$. Additionally, a subspace s is represented by a bitstring of length d , where $s[i] = 1$ iff the subspace includes the i 'th dimension.

In this paper, we propose a compact data structure to rapidly answer skyline queries [1] over arbitrary subsets of attributes, which relies on a notion called *dominance* [1]:

Definition 1 (subspace dominance (p, q, s)). Given points $p, q \in P$ and a bitstring s of length d , let EQ, GT also be

¹Bitstrings and integers here mean both an integer value and the bitstring representing that value (e.g., 7 and 1110).

id	Movie Title	DOM _{<i>i</i>}	
		binary	integer
0	Avatar	1110 000	7 0
1	The Avengers	0101 000	10 0
2	The Godfather	1011 100	13 1
3	The Shawshank Redemption	1001 110	9 3
4	Skyfall	0111 111	14 7

Table 2: Table 1 movies and their domspaces vector (using subspace order ⟨Y,R,YR,S,YS,RS,YRS⟩ and big-endian).

bitstrings of length d , where:

$$\begin{aligned} \text{EQ}[i] &= 1 \text{ iff } p[i] = q[i] \\ \text{GT}[i] &= 1 \text{ iff } p[i] > q[i]. \end{aligned}$$

Then, p dominates q in subspace s , denoted $p \succ_s q$, iff:

$$((\text{EQ} \ \& \ s) \neq s) \wedge ((\text{EQ} \mid \text{GT}) \ \& \ s) = s.$$

If all data values are unique, known as *Distinct Value Condition* [7], EQ fades from Definition 1. A *subspace skyline* [6] is the subset of points not dominated in the subspace:

Definition 2 (subspace skyline (P, s)). Given a set of records P and a bitstring s of length d , the *subspace skyline* of P is:

$$\text{SKY}(P, s) = \{p_i \in P : \nexists p_j \in P, p_j \succ_s p_i\}.$$

If $s = 2^d - 1$, Definition 2 produces the *full skyline*. The *skycube* [6, 11] is the set of subspace skylines (each called a cuboid [11]) for all non-zero bitstrings of length d .

Finally, we define for our data structure a mapping between points and subspace skylines (examples in Table 2):

Definition 3 (domspaces vector of p_i). Point p_i 's *domspaces vector*, denoted DOM_i , is a bitstring of length $2^d - 1$ where:

$$\text{DOM}_i[j] = 1 \text{ iff } p[i] \notin \text{SKY}(P, j).$$

In other words, a domspaces vector records the subspaces in which point p_i is dominated (*not* in the skyline).

The objective in this paper is to store a compact representation of all cuboids that can support more efficient subspace skyline queries than state-of-the-art algorithms.

Skycube algorithms Börzsönyi et al. [1] introduced the skyline with (external-memory) algorithms *block-nested-loops* (BNL) and *divide-and-conquer* (DC). *Sort-First Skyline* (SFS) [3] improves BNL, pre-sorting the data so points will first be compared to those more likely to dominate them. *Object-based Space Partitioning* (OSP) [12] improves DC by recursively partitioning points based on existing skyline points, rather than a grid. BSKyTree [4] improves OSP by optimally choosing the points with which to partition P .

The skycube was introduced independently by Yuan et al. [11] and Pei et al. [6], with adaptations of the DC [1] and SFS [3] skyline algorithms, respectively. More recently, QSKyCube [5] adapted the BSKyTree algorithm [4]. These algorithms compute cuboids one-by-one, using the corresponding skyline algorithm. Based on results reported in [4, 5], BSKyTree and QSKyCube are state-of-the-art.

Skycube data structures The default skycube data structure is the *lattice*, used in QSKyCube [5]. It is an array of $2^d - 1$ vectors, and the i 'th vector contains all points in the i 'th cuboid. Naturally, this has optimal performance: one retrieves the proper vector from the array and then reports all points lying therein. However, it is *maximal* in terms of

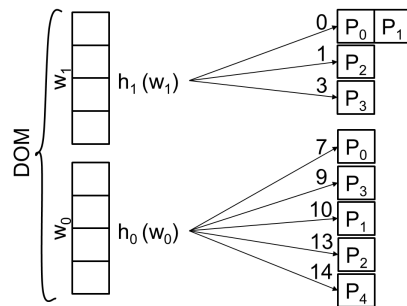


Figure 1: The Hashcube, built from Table 2 with $|w_i| = 4$.

space: each point is duplicated for every cuboid it is in, $\frac{1}{2} 2^d$ times for points with maximal values on some attribute.

Two smaller data structures have been proposed. The *closed skycube* [8] defines equivalence classes over subspaces and avoids duplicating points within an equivalence class.

The more recent *compressed skycube* [10] defines *minimal subspaces* skyline points and constructs a bipartite membership graph between points and minimum subspaces. Thus a point is not duplicated for any subspaces between its minimal subspaces and the full skyline. In the absence of Distinct Value Condition, it introduces overhead to rederive any particular cuboid, because false positives must be verified with dominance tests in all subspaces of the query subspace.

3. THE Hashcube DATA STRUCTURE

Here, we introduce the Hashcube, obtaining up to $|w|$ -fold compression (for $|w|$, the number of bits in each logical word) and state-of-the-art query performance.

3.1 Layout of the Hashcube

We illustrate a Hashcube in Figure 1, using the data from Tables 1 and 2. The high-level idea is to split the domspaces vectors for each point into words of length $|w|$ (4 in examples, 32 in experiments), and to index the points by their resultant substrings using hash maps. Since the domspaces vector has length 2^d , each point will be indexed $\leq 2^{d-|w|}$ times. The substrings are the keys for the hash maps. More precisely, if $\Sigma = \{0, 1\}^{|w|} \setminus \{1\}^{|w|}$ denotes the set of length $|w|$ bitstrings containing at least one zero, and $k = \max(1, 2^{d-|w|})$:

Definition 4 (Hashcube (P)). A *Hashcube on P* is a set of k hash maps, h_0, \dots, h_{k-1} , each mapping from valid bitstrings in Σ to subsets of P , $h_j : \Sigma \rightarrow \mathcal{P}(P)$, where:

$$p_i \in h_j(B) \text{ iff } \text{DOM}_i[|w|j, |w|(j+1) - 1] = B.$$

That is, each hash map corresponds to a group of $|w|$ cuboids. Points are binned according to the combination of those cuboids in which they appear. For example, in Figure 1, w_1 corresponds to subspaces $\{\text{Year, Sales}\}$, $\{\text{Year, Rating}\}$, and $\{\text{Year, Rating, Sales}\}$, respectively. Both *Avatar* and *The Avengers* are binned to 0, since they appear in all three cuboids. Although *The Godfather* appears in the last two cuboids, it does not appear in $\{\text{Year, Sales}\}$: it has a different combination, namely 1, and maps to that bin instead.

Compression for a Hashcube depends on the number of clear bits in the substring of a domspaces vector, up to $|w|$. Note, first, that a point is only ever indexed by a hash map if it has a zero bit, i.e., if it appears in at least one of the

Algorithm 1 Querying the Hashcube

Input: Hashcube; query subspace, B ; word length, $|w|$ **Output:** The skyline of subspace B

```
1: Let  $j = B/|w|$ 
2: Let  $mask = (1 \ll (B\%|w|))$ 
3: for all active hash keys  $k_i$  of  $h_j$  do
4:   if  $(k_i \& mask) == 0$  then
5:     Output all  $pid$  in  $h_j(k_i)$ 
```

corresponding $|w|$ cuboids. If so, it must also be indexed for that cuboid by the lattice. Conversely, a point is only indexed once by each hash map, no matter how many of the $|w|$ cuboids in which it appears; the lattice may index the point $|w|$ times. Further compression comes by not storing unused hash keys and by points mapping to identical bins.

3.2 Querying the Hashcube

Notice from Definition 3 that the j 'th cuboid consists of all points p_i for which $DOM_i[j] = 0$. So, for the Hashcube, the query operation is to concatenate all vectors of point ids for which that bit is not set. Because Definition 4 treats each group of $|w|$ bits/cuboids independently of the rest, the query can be resolved with just one of the $2^{d-\lg|w|}$ hash maps. Algorithm 1 describes the query operation: first the relevant hash map is determined, and then all $\leq 2^{|w|}$ active hash keys for that hash map are iterated. For those that have the relevant bit clear, the entire vector of point ids is output. No point will be output twice, because each point id is stored at most once per hash map. The iteration of active hash keys is the primary source of overhead relative to the lattice, a cost of at most $2 * 2^{|w|}$ binary/logical operations.²

The cost of querying the data structure is also very low. Lines 1 – 2 require constant computations. We then read up to $2^{|w|}$ active hash keys, perform two operations, and (possibly) output some unique point ids (if the condition on Line 4 evaluates true). So, if there are m point ids to output, then the cost of querying the Hashcube is $\mathcal{O}(2^{|w|} + m)$.

4. EXPERIMENTAL EVALUATION

We compare Hashcube ($|w| = 32$) to the compressed sky-cube (CSC) [10], the lattice, and computation from scratch using the BSkyTree [4] skyline algorithm. (Note that larger $|w|$ improves compression; smaller $|w|$ improves query time.) We implement (code available³) the data structures and query algorithms in C++.⁴ The lattice is built as an array of vectors of point ids. CSC is strongly implemented, evidenced by the faster performance than reported in [10] (albeit on newer hardware). The implementation of BSkyTree was provided by the authors, but adapted to handle subspace queries. We use an Intel Core i7-2700 machine with four 3.4 GHz cores and 16 GB of memory, running Linux (kernel version 3.5.0).

We evaluate the data structures in terms of space and query time. We measure space by counting 32-bit point ids and hash keys used, a more robust measure than physical disk space because of external libraries (e.g., `std::map`).

²The Hashcube also requires outputting up to $2^{|w|-1}$ separate lists, rather than one, long contiguous one.

³source at: <http://cs.au.dk/research/research-areas/data-intensive-systems/>.

⁴compiled using g++ (4.7.2) with the -O3 optimization flag

We measure query time by dividing the total time to sequentially query every subspace by $2^d - 1$. In contrast to uniformly sampling subspaces with replacement as in [10], this better estimates expected performance: the worst cases ($(d-1)$ -dimensional subspaces) are otherwise unlikely included. Output to an array in memory, but not init time, is included.

We evaluate how the structures scale with respect to both d and $|P|$ on anti/correlated distributions, generated as in [1]. We adopt default values, $d = 12$ and $|P| = 500K$ from [4].

Experiment Results Overall, CSC achieves the most compression. Figure 2 shows that all data structures scale well with $|P|$ in terms of size, since the size of each cuboid grows sub-linearly with $|P|$. That the CSC has a worse compression rate on anti-correlated data is intuitive, because the minimum subspaces for each point are larger. In Figure 3, see that CSC's compression relative to the lattice increases with d , because there are longer paths between minimum subspaces and the full skyline. Hashcube is generally closer to CSC than to the lattice. Relative to the lattice, it obtains $\approx 10\times$ compression and permits storing in the same amount of space 2–4 more dimensions (4–16 \times more cuboids).

The same trends exist for physical space (not shown). We use standard libraries, rather than more space-efficient, custom-built containers. Still, for $d = 12$ and $|P| = 500K$, HashCube achieves a compression ratio (in bytes) of 7.9 \times (compared to 13.8 \times). The same ratios apply for CSC.

Figures 4 and 5 report average query performance. The Hashcube performs very strongly, closely following the optimal performance of the lattice, typically 1000 – 10000 \times faster than CSC. The iteration of all hash keys only takes 5 – 10 \times as long as the direct lookup in the lattice. Further, on anticorrelated data, Hashcube converges towards the lattice with increasing $|P|$ (Figure 4). By contrast, CSC is rather slow, beaten in most instances by simply computing the skyline from scratch with BSkyTree. CSC outperforms BSkyTree only on small, correlated instances of $< 200\mu s$.

The poor query performance of CSC results from dominance tests required to reconstruct each cuboid. As d increases, exponentially more subspaces of a query space must be examined for false positives. With respect to $|P|$, trends match the size plots. The correlation is expected: for each subspace, the number of dominance tests is quadratic in the number of points for which that is the minimum subspace. It should also be noted that the variance of query times for HashCube is small, i.e., never exceeds 1ms, while CSC typically spends minutes on high-dimensional queries. This is a result of the split into several bitstrings of size $|w|$, which limits the number of hash keys for each query, while CSC needs to iterate the data points in all subspaces of the chosen dimensions and needs to perform dominance checks.

Hashcube is efficient to query, typically 1000 – 10000 \times faster than CSC and computing from scratch with BSkyTree. The iteration of all hash keys only slows Hashcube 5 – 10 \times relative to the lattice. Further, on anticorrelated data, Hashcube converges towards the lattice with increasing $|P|$. The cost of outputting longer contiguous vectors is negligible; so, the increased input size only slows the data structure if new points associate with as-yet-unused hash keys. With respect to d , the curve follows that of the lattice quite closely.

We call particular attention to Figure 5, because it expresses very well the balance that Hashcube obtains. We are unable to finish the plot for both the lattice and CSC, but for opposite reasons. The lattice does not fit in 16 GB

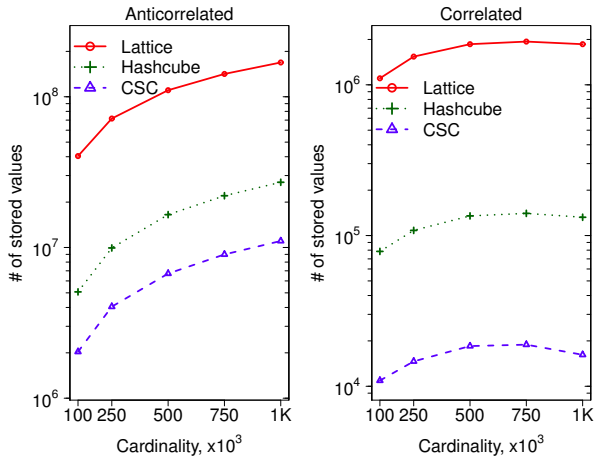


Figure 2: Size of the data structures w.r.t. to n ($d=12$).

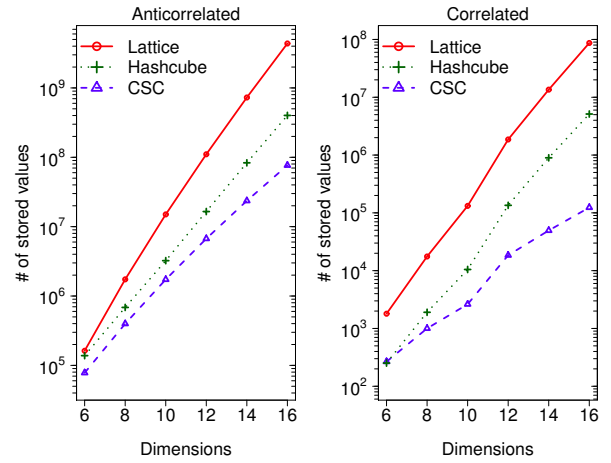


Figure 3: Size of the data structures w.r.t. to d ($n=500K$).

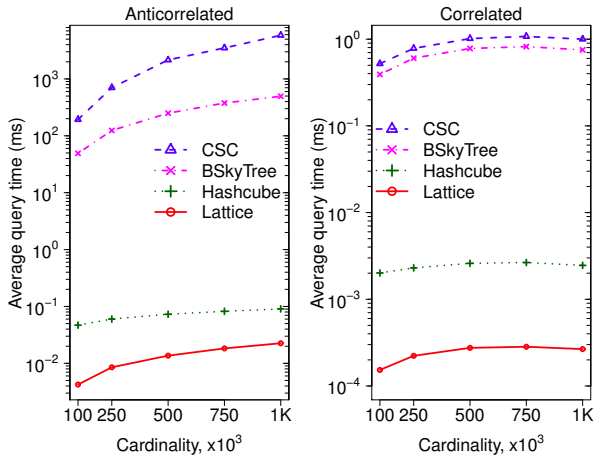


Figure 4: Data structure query time w.r.t. to n ($d=12$).

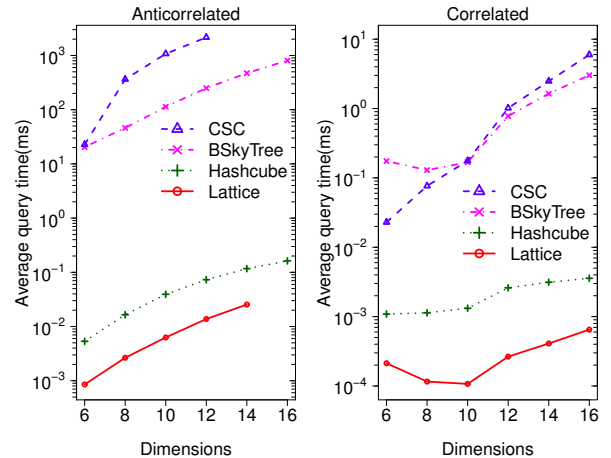


Figure 5: Data structure query time w.r.t. to d ($n=500K$).

of memory; so, we cannot query it fairly. On the other hand, CSC achieves good compression, but has prohibitive query time (> 48 hrs total). Hashcube is very efficient in both respects and supports this 16-dimensional, anticorrelated case. It compresses well and still, across all tested combinations of $|P|$ and d , can be queried on average in less than $200\mu s$.

5. CONCLUSION AND OUTLOOK

We introduced a compressed skycube based on bitstrings, the Hashcube. Relative to the lattice, it achieves $\approx 10\times$ compression. Relative to the state-of-the-art compressed skycube, queries are $\approx 1000\times$ faster. Further, we showed that, while the compressed skycube is updatable, it is outperformed by skyline computation from scratch. Thus, updating skycubes is still a challenging open problem. For future work, we believe equivalence class ideas from [8] and/or more sophisticated cuboid grouping choices can be integrated into the Hashcube to further improve compression. Small, auxiliary structures may help handle some update types.

6. ACKNOWLEDGMENTS

This research was supported in part by the Danish Council for Strategic Research, grant 10-092316. We thank the BSKyTree authors [4] for their skyline implementation.

7. REFERENCES

- [1] S. Börzsönyi et al. The skyline operator. In *Proc. ICDE*, pages 421–430, 2001.
- [2] S. Chester et al. On the suitability of skyline queries for data exploration. In *Proc. ExploreDB*, pages 6:1–6, 2014.
- [3] J. Chomicki et al. Skyline with presorting. In *Proc. ICDE*, pages 717–719, 2003.
- [4] J. Lee and S.-w. Hwang. Scalable skyline computation using a balanced pivot selection technique. *Inf. Syst.*, 39:1–24, 2014.
- [5] J. Lee and S.-w. Hwang. Toward efficient multidimensional subspace skyline computation. *VLDB J*, 23(1):129–145, 2014.
- [6] J. Pei et al. Catching the best views of the skyline: a semantic approach based on decisive subspaces. In *Proc. VLDB*, pages 253–264, 2005.
- [7] J. Pei et al. Towards multidimensional subspace skyline analysis. *TODS*, 31(4):1335–1381, 2006.
- [8] C. Raïssi et al. Computing closed skycubes. *PVLDB*, 3(1):838–847, 2010.
- [9] A. Vlachou et al. Skypeer: Efficient subspace skyline computation over distributed data. In *Proc. ICDE*, pages 416–425, 2007.
- [10] T. Xia et al. Online subspace skyline query processing using the compressed skycube. *TODS*, 37(2), 2012.
- [11] Y. Yuan et al. Efficient computation of the skyline cube. In *Proc. VLDB*, pages 241–252, 2005.
- [12] S. Zhang et al. Scalable skyline computation using object-based space partitioning. In *Proc. SIGMOD*, pages 483–494, 2009.