Index Intersection for High-Dimensional Range Queries

Maximilian Berens
maximilian.berens@cs.tu-dortmund.de
TU Dortmund University
Dortmund, Germany

Jens Teubner
jens.teubner@cs.tu-dortmund.de
TU Dortmund University, Lamarr Institute for Machine Learning and Artificial Intelligence
Dortmund, Germany

ABSTRACT

For high-dimensional analytics queries in scientific domains, a full table scan is often seen as the only feasible execution path, even if result cardinalities are known to be small. In this paper, we argue for intersecting multiple indices built over medium-sized attribute subsets (Teams) as a means to produce a list of (candidate) tuple IDs for further processing. While this strategy is compatible with various index structures, significant discriminative power lies in the combined selectivity of multiple predicates. Akin to bitmap indices and VA-files, adopting simple and lightweight index approaches for each Team, instead of highly accurate but costly ones, still enables high overall selectivity and precision. Thus, the focus shifts away from individual indices towards their efficient intersection. Teams with just 1 member/attribute (bitmap indices) are outperformed for selective queries due to the inability to avoid access to large parts of the index. For example, Teams with 5 members are up to 6-7 times faster for 85 dimensions and require 1.58-2.07 times less storage. Team-based Indexing is most useful for queries with high selectivity and dimensionality, such as the search for rare objects.

PVLDB Reference Format:
Maximilian Berens and Jens Teubner. Index Intersection for High-Dimensional Range Queries. PVLDB, 19(4): 767 - 779, 2025.
doi:10.14778/3785297.3785315

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/maximilianberens/TeamIndex.

1 MOTIVATION

High-dimensional range queries often produce small results, i.e., offer high selectivity. Especially at the very large scale, having the ability to reduce petabytes of data early on to a small percentage offers significant advantages over *full-table* scans, where data is processed exhaustively. However, traditional index structures are ill-suited for high-dimensional data and often incur storage costs comparable to, or even exceeding, those of the base table. As a result, multi-dimensional index structures (MDIS) are often deemed infeasible, suggesting that scans are the only viable alternative.

In order to process their large amounts of data at all, physicists at the large hadron collider (LHC) at CERN have to swallow a rather bitter pill. In the form of so-called "stripping lines," they precompute answer sets to often-needed queries [1]. However, this not only costs them limited storage volume; physicists also have to sacrifice flexibility and/or accuracy of their queries. The absence of any index structure also complicates predicate fine-tuning to achieve a desired selectivity, as accurate high-dimensional cardinality estimates are notoriously difficult to obtain [25].

Instead, in this work we aim to offer extreme scalability for multi-dimensional range predicates, high flexibility for queries, and low cost—all at the same time. Our work will benefit the LHC and similar workloads in the sciences (e.g., processing Sloan Digital Sky Survey data [30]) but also in other domains.

Indexing At Extremes: In the context of databases, secondary indexing for range queries is usually approached at the two extremes: Either all dimensions are indexed together by one MDIS or each attribute has its own single-dimensional index structure (SDIS). However, either approach has considerable caveats: MDIS, e.g., X-trees [7] or R*-trees [5], suffer greatly from the curse of high dimensionality: If applied over a high-dimensional space, storage- and/or execution costs become prohibitive [7, 28]. On the other hand, creating indices for all dimensions *individually*, e.g., via bitmap indices [29], does not suffer from the curse. However, the need for presenting all tuple references for every dimension independently leads to a large storage footprint (despite compression). Moreover, individual predicates usually offer only moderate selectivity, which entails access to large parts of the index, and thus high intersection cost.

Team-based Indexing: Instead of "one (index) for all" attributes or "one for each," we explore the middle ground: Creating indices over *moderately-sized subsets* of attributes, which we dub "Teams" in the following. A Team-based approach enables trade-offs between (individual) index structure complexity, storage overhead and index intersection performance. Similar to bitmap indices, using moderately-sized Teams can still cope with the curse of high dimensionality and still harness benefits from multi-dimensional indexing. However, we also have to consider new challenges: For one, there are multiple ways for "composing Teams," as well as suitable (and potentially differing) index structure choices for each. Further, given access performance of modern SSDs, an efficient index intersection requires a hardware-concious implementation and logical optimization of the index intersection, since index intersection is usually not limited by SSD bandwidth alone.

Contribution: In this paper, we use a simple and configurable index structure (the "grid index") to demonstrate the general efficacy and discuss challenges of a Team-based approach. We discuss necessary conditions when using a Team-based approach for producing a set of tuple ids (the *index result*) for high-dimensional range queries, as well as the two primary challenges, namely *Team Composition* and *Index Intersection*. We categorize and experimentally study general

Proceedings of the VLDB Endowment, Vol. 19, No. 4 ISSN 2150-8097.
doi:10.14778/3785297.3785315

performance factors and discuss logical runtime optimizations for index intersection using a prototype implementation and both real-world and synthetic data.

2 METHOD & PRELIMINARIES

Team-based indexing generalizes the traditional approaches—either one D_{table}-D(imensional) index structure or D_{table}-many 1-D indices—for multi-dimensional range query evaluation on a D_{table}-D table.[1] Bitmap indices, such as FastBit [29], therefore present an edge case of Team-based indexing, which is generally proceeds in the following way:

(1) **Map** the query to relevant Teams and ID lists (or "leaves")
(2) **Optimize** the intersection logically
(3) **Retrieve & intersect** relevant lists
(4) (Refine candidates)

For example, with bitmap indices, the query is first (cheaply) mapped to relevant bitmaps, and per-attribute results are subsequently intersected. At the other extreme, a single high-D (tree-based) Team index is traversed to find relevant leaves, whereas the subsequent intersection phase is trivial. Logical optimization (of the intersection) becomes important for non-trivial index intersections over many lists, where inter-operator parallelism is required (see Sec.4).

In general, index evaluation may lead to *false-positive* results, i.e., tuple ids that are near the query region but lie outside it. This is the case if the query rectangle does not exactly align with how the index decomposes the data space. For example, a (binned [23]) bitmap index has to consider all bitmaps that even partially overlap with the query. Similarly, an X-tree's leaf may contain irrelevant data, if a bounding box only partially overlaps. False-positive results need to be pruned later ("candidate refinement") when fetching actual data with the index result and therefore inflate the overall index selectivity—the index looses *precision.*[2] Note that we do not further discuss candidate refinement in this work, since the efficient usage of the index result is orthogonal to the indexing approach.

2.1 Index Structure Choice

Generally, *any* (secondary) index structure can serve as a Team index and each Team may select a different index. Index structures implement various trade-offs in terms of precision and leaf count, may opt for specific storage representations [29] or may produce overlapping leaves [5, 7]. However, index intersection is generally agnostic to how queries are *mapped to leaves* ("tree traversal")—the input is a set of ID lists. Thus, the index structure choice becomes a potential variable to be tuned in a Team-based indexing approach. But given high query dimensionality, requirements on index precision, and thus index complexity and costs, can relaxed.

Exploiting Combined Selectivity: In a Team-based approach, leaf retrieval (which involves storage access) and the subsequent intersection across Teams present the dominant runtime cost. Since the smallest achievable selectivity *per Team* is ultimately limited by *the query*, minor improvements in index precision for individual Team indices have diminishing benefit: Whereas a single predicate may or may not offer high selectivity, and thus lower ID volume in the intersection, a conjunction of even a few somewhat-selective predicates drops overall access volume significantly and reliably (see Sec.6.4). Combined selectivity is benficial for access performance, but also for overall false-positive rate (see Sec.2.6). Moreover, a very fine-grained space decomposition increases the cost of per-Team index traversal, index intersection, as well as costs for storing and retrieving index data. As a result, "cheap," but more imprecise indices can still enable significant *overall* selectivity and fast runtime, if considered in a Team-based setting. This core idea also lies at the core of VA-files [28] and bitmap indices [29], which rely on coarse per-attribute quantization/binning.

To this end, we focus our experiments on an index structure that is easy to understand and analyze, as well as flexibly tunable to different levels of precision to showcase the general efficacy of a Team-based approach. We briefly discuss alternate choices in Sec.3.

The Grid Index: A grid index is a simple, Cartesian grid-based partition of the Team-space into a fixed number of (disjoint) "bins" (or cells). To partition a d-D space, we define bin borders along each dimension/attribute by $b - 1$ equidistant statistical quantiles. For example, with $b = 10$, the first bin of one dimension spans the interval $(-\infty, q_{0.1}]$ and the last bin covers $(q_{0.9}, \infty)$, where $q_{0.1}$ and $q_{0.9}$ denote the 0.1 and 0.9 quantile values, respectively. Using quantiles implies that the marginals[3] along each grid dimension have an approximately uniform distribution, i.e., cardinalities of all bins in an interval sum up to $1/b$-th of the data. The way this transforms the data representation is illustrated in Fig. 1—dense regions are, to a degree, spread out and some bins may remain empty. Note that the relative position of the values *within* each bin (Fig. 1, right) is *not* retained, which makes it impossible for the index to distinguish true-positive and false-positive results. Overall, there are exactly b^d bins and each bin has a relative cardinality ranging between 0 to $1/b$. We use parameter b to simulate different levels of index precision and leaf counts.

We will call a subset of attributes a *Team*, the associated index structure *a Team index* and the result the index produces the *Team result.* A specific set of Teams forms a *Team composition.* A *leaf* refers to the posting list associated with a specific, *non-empty* bin in one of the Team grids, akin to a leaf in a R*-tree or a bitmap in FastBit. A *list* may refer to either a leaf or an intermediate result.

2.2 Application Conditions

For high-dimensional range queries we will pay particular attention to the following properties:

SSP: Sufficiently Selective Predicates
SAC: Sufficient Attribute Coverage

Predicates are considered *sufficiently selective*, if they prune at least one bin. For example, for $b = 10$, a predicate with selectivity $s > 0.9$ has to consider all bins in order to avoid overlooking results, i.e., false-negatives. Overall, we found b between 8 and 10 to be still sufficient for achieving high overall selectivity. Second, most attributes in query-relevant Teams should have *SSPs*, i.e., *coverage* of the considered Teams, otherwise the access and intersection performance per Team degrades (see Sec.5.2).

[1] We consider only (hyper-)*rectangular* queries, i.e., conjunctions of range predicates.
[2] Precision = $\frac{\#\text{relevant IDs}}{\#\text{returned IDs}}$.

[3] "Marginal" refers to the discrete 1-D distribution resulting from summing up bin cardinalities over all dimensions except one, rather than over an arbitrary, proper subset of dimensions.

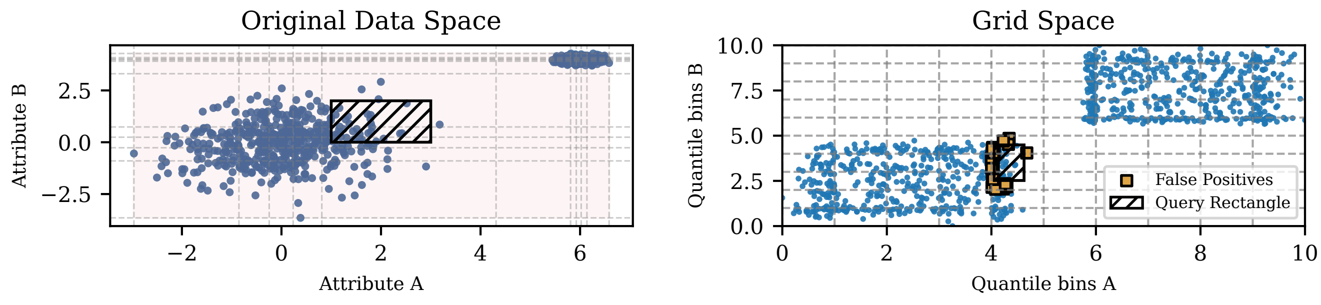

Figure 1: 2-D Team data space and the corresponding Cartesian-grid space with $b = 10$. False-positive results are elements located outside the query rectangle but still inside query-relevant bins.

Lastly, the query needs to be highly selective to make indexing in general worthwhile, and sufficiently high-dimensional to make Team-based indexing worthwhile. We classify queries as *highly* selective if they have selectivities of $s_{\text{total}} < 0.01$. Queries are considered high-dimensional, if they require more dimensions than a single index can handle effectively, i.e., usually about 15 and more.

2.3 Index Creation

Given limited Team size and when using the Cartesian grid index, Team indices can be built in linear time of both table cardinality N and dimensionality D_{table}:

(1) **Determine** b and per-attribute quantiles (see Sec.2.6)
(2) **Set** the maximum Team size (see Sec.6.1)
(3) **Compose** Teams (see Sec.6)
(4) **Assign** tuples to exactly one *bin per Team* using quantiles
(5) **Persist** assignments

In an initial pass over the data, quantiles can be obtained using a suitable approximation [21]. After Teams are determined, bin assignment requires a second pass. Assignments are then stored as posting lists (*leaves*), i.e., sorted sequences of tuple IDs. Empty bins do not result in leaves and are ignored during access.

For comparison, instead of an "inverted" approach, the aforementioned VA-files use the same binning schema but store the *bin IDs in table order* [28]. This representation can be scanned as a proxy to the actual data for producing the index result, but it does not offer any way to selectively read only relevant IDs to a specific bin.

Storage Layout: Posting lists are always stored at the beginning of a page with padding at the end of a partially filled page. In all our experiments, we use serialized 32-bit Roaring bitmaps [9] when beneficial, and no compression otherwise (i.e., for short leaves).

Roaring partitions the 32-bit integer domain into 2^{16} blocks based on the upper 16 bits of each integer. For each block, a "container" stores the lower 16 bits of the values. Containers can be stored either as arrays (for sparse data), bitmaps (for dense data), or runs (for sequences) for more compact representations and faster processing. The library provides SIMD-accelerated set operations such as union, intersection, and difference.

We also tested other compressions and found that costly decompression can defeat any gains from higher compression ratios [27], resulting in worse runtime performance. If storage cost is of larger concern, a combination of delta- and varint-encoding with ZStd general purpose compression [10] offered the best compression rates for basically all tested configurations. For example, for a composition with $b = 10$ and $d = 4$ on the 16-D LHCb dataset, the index was ≈ 19.6 GB in size without compression, ≈ 10.75 GB with Roaring and ≈ 2.98 GB with varint + ZStd.

Index Updates: Update support is inherited by the underlying Team index structures. For our implementation of the grid index, appending new tuples is straight forward, whereas moving them from one bin to another requires rebuilding both bins. Note that inserting large amounts of tuples may cause the quantiles to shift, potentially leading to skew in the grid's marginal distributions/quantiles. In this case, rebuilding the index may become necessary.

2.4 Meta Data & Query Mapping

During index creation, we keep track of bin cardinalities, storage offsets of the associated leaves, as well as compressed sizes for each Team. Further, we store the compression codec in use, i.e., Roaring and *uncompressed* for the shown experiments, for each leaf. Each d-D matrix requires b^d unsigned integers. For the considered configurations of b and d, meta data overhead was negligible and fixed with respect to N. Just like bitmap indices, the grid index may be considered "leaf-only" due to the absence of complex tree-like structures. Using this meta data and the grid quantiles, queries can be mapped to relevant bins at negligible cost for the considered configurations. Fig. 1 illustrates this process.

2.5 Workload & Hardware Setup

In our experiments throughout the paper, we use real-world datasets from the LHCb experiment at CERN [1] representing particle decays and a "data sweep" from the Sloan Digital Sky Survey (SDSS) representing stellar objects identified as stars.

The **LHCb dataset** has $D_{\text{table}} = 16$ double precision attributes describing particle decays. It has $N = 1.222 \cdot 10^9$ tuples/decay, which amounts to ≈ 153 GB, or ≈ 19 GB in Snappy-compressed [12] Parquet format [2]. The large compression factor stems from repeated values and beneficial tuple order. Attributes describe properties of multiple particles in a specific decay, such as impulse or mass.

The **SDSS dataset** has $D_{\text{table}} = 85$ single precision attributes and $N = 0.434 \cdot 10^9$, which amounts to ≈ 147 GB. The dataset was overall incompressible (Snappy). The attributes are made up of 17 underlying variables, such as flux and noise statistics, that are split into 5 separate optical *band* attributes, each representing a different wave-length. We will use this grouping property later to investigate domain-specific Team composition (see Sec.7).

Running Example: Our running example query will be a highly selective real-world query with 14 predicates over the 16-D LHCb data, ranging in selectivity between ≈ 0.1 and ≈ 0.92. For $b \leq 10$, the query has 12 SSPs due to the lack of per-attribute precision. The index results for $b \in \{5, 10, 16\}$ yield 38109, 3728 and 1312 tuple ids, respectively. This illustrates that the grid index can indeed achieve significant selectivity of $3.12 \cdot 10^{-5}$–$1.07 \cdot 10^{-6}$ for this query.

Randomization: We also created a uniform version of the LHCb dataset without data dependencies but used equivalent marginal cardinalities for the indices. Thus, *individual* predicates evaluated on similarly-built indices on both datasets have exactly the same selectivity. The dataset was incompressible ($\approx$ 151 GB parquet file).

We also make use of Team compositions drawn uniformly at random in the following way: First, a list of attributes is shuffled and then split according to an integer partition of its length, e.g., 16 elements were split into three lists of size 6, 5, and 5, respectively. Further, we created random queries by translating a query (rectangle) uniformly at random in grid-space. Note that, despite equal predicate selectivities, overall query selectivity still varies slightly due to intra-Team attribute dependencies.

Hardware Platform: Experiments always use all 32 threads on an AMD EPYC 9124 processor with 16 cores, 384 GB DDR5 RAM and Linux 6.8.0-55-generic kernel. Indices were stored on a Kioxia CM7-R SSD with up to 14 GB/s PCIe 5.0 read bandwidth.

2.6 Precision & Team Size: Diminishing Returns

Generally, indices can produce false-positive results, and the corresponding loss in precision *inflates* overall selectivity. Similarly, increasing Team size *improves* per-Team selectivity. In this section, we discuss the interplay and diminishing benefits of these two tuning knobs.

Sparsity of High-Dimensional Spaces: Especially for the Cartesian-grid index with low bin count b and low-D queries, the error rate can be large. However, index *precision* is not just a matter of the "grid resolution" along *individual* attributes: For (rectangular) queries, combining several only moderately-selective predicates can offer substantial pruning potential. The reason for this is the *sparsity of high dimensional spaces*: Intuitively, irrelevant tuples differ sufficiently in some dimension (of the grid) *eventually*. With growing dimensionality, the chance for this variation increases exponentially, even if the number of possible values/bins per dimension is limited. For a grid index with quantile-based bins, this allows to shrink the volume of a query rectangle (in unit-cube space $[0, 1]^D$) with every additional SSP (Sec.2.2). Large portions of the data space can thus consistently being discared, since bins are defined independently for each dimension.

Another benefit of high query dimensionality is that a few unused attributes or poorly-selective predicates are more easily compensated for—the query may still be highly selective. In this sense, high (query) dimensionality *blesses* us with high levels of index precision, as well as some robustness for low-selectivity attributes.

To illustrate the ability of a quantile-based grid index to distinguish tuples, we have partitioned the LHCb dataset into a 10^{16} dimensional grid. For a Team with $d = 16$, the resulting leaf count is $\approx 0.33 \cdot N$, primarily due to repeated values. For a more uniform dataset, the number of leaves would be closer to $1 \cdot N$, where every tuple is essentially placed in its own bin. But even with 3 IDs per bin (on average), identifying relevant parts of a 16-D grid quickly becomes the most expensive step.

Sparsity Without Exponential Cost: The value b should be selected with Team size in mind: A grid-based Team index eventually suffers from the *curse of high dimensionality* due to the exponential increase in complexity, just like other MDIS. Thus, *moderately-sized* Teams exploit some of the combined selectivity for more efficient access, while *intersection* allows us to tap into the remaining selectivity for an overall high index precision. For example, intersecting 4 Teams with 4 dimensions (instead of one 16-D Team) still allows differentiation between all b^{16} disjoint regions of the data space, even if Teams consider only 4-D projections of the table. At the same time, combining 4 predicates still allows for more selectivity and thus lower access volume per Team (than 1-D Teams).

Diminishing Returns: Generally, retrieving fewer IDs is beneficial for index intersection. But given the sparsity of high-dimensional space, choosing a larger b only gives diminishing returns for precision. This is illustrated by our example query (see Sec.2.5), where going from $b = 10$ to $b = 16$ bins barely decreases result size, reflecting the rapidly saturating relative gain in selectivity. Similarly, larger Teams improve their respective *selectivity* and reduce overall Team count, but each increase in dimensionality also increases cost and the relative improvement in selectivity diminishes, too.

This trade-off suggests a sweet spot for both Team size (or, equivalently, Team count [6]) and index precision (represented by b). Overall, if queries are high-dimensional, Team-based indexing shifts focus from individual index quality towards efficient index intersection. Conversely, if query dimensionality is low, the precision of individual predicates matters more and a single Team with a precise MDIS becomes the superior option.

Notably, one may argue that higher compression rates for 1-D Teams reduce the number of *bytes* involved and therefore compensates for bad predicate selectivity. In Sec.6, we show that the need to (implicitly or explicitly) represent every tuple ID *at least once per Team* generally outweighs the effect of compression w.r.t. overall smaller storage footprint and access volume.

3 OTHER RELATED WORK

The concept of *vertically splitting a higher dimensional space into lower-dimensional projections and to consider them separately* finds application in various works. The most relevant is tree striping [6], where Team-based indexing is suggested as a new *clustered* MDIS. A key difference to our work is that they consider the *index intersection* as an external merge-sort problem. Modern DRAM capacity and lightweight compression algorithms allow executing the intersection of *secondary* indices in-memory, even for very large problems. Beyond tree striping, the overarching concept also finds application in other fields, e.g., product quantization [16] for approximate nearest neighbor search.

Adaptive Indices: Generally, MDIS can suffer from false-positives if query rectangle and leaf geometry, e.g., bounding boxes, do not overlap perfectly. Adaptive MDIS [24], such as X-tree [7] or R*-tree [5], typically follow the underlying, multi-dimensional data distribution and split denser regions more frequently, whereas a Cartesian grid implies a rigid, fixed sized *partition*. This primes adaptive methods for extremely selective or point-wise access, where–ideally–only a single leaf needs to be ultimately retrieved. However, with only moderate Team size, the combined selectivity of the respective predicates may be insufficient to fully benefit from this precision. Although the specific index choice is workload dependent, less complex (and thus more compact, index structures may be preferable, especially given that Teams are meant to be intersected.

For large N, index structures with leaf counts independent of N (not necessarily only the grid index) can offer an advantage. To illustrate, a k-d-tree index [24] with fixed 4 KiB leave size and 4 B IDs has *at least* $\lceil N/1024 \rceil$ leaves, whereas a grid index has at most b^d. Another notable difference is that leaves of indices with an upper limit to leaf count do not have equal cardinality leaves, which can become larger than a typical storage page of 4 KiB.

Lightweight Indices: Another related MDIS is the Z-order index [26], which encodes multi-dimensional values via bit-interleaving. It implicitly defines a regular (hierarchical) grid whose resolution (and thus the number of bins) depends explicitly on the chosen bit-precision. Bins defined by Z-order indexing are fixed by this precision and their occupancy depends on the underlying data distribution, meaning many bins may remain empty. Note that a Z-order index can be applied on a quantile-based, quantized representation. In this sense, the Z-order index could be used "on top" of our Cartesian grid index to find relevant leaves. It is usually employed for *clustered indexing*.

A direct end-to-end runtime comparison between existing MDIS implementations or an integration into our framework is not straightforward. Most implementations, such as FastBit, are single-threaded and make use of synchronous I/O. Others are not optimized for range queries, are clustered or purely in-memory. Our prototype relies on secondary indices, makes use of asynchronous I/O, lightweight compression (Roaring) and task-based multi-core parallelism. We consider compositions with singleton Teams to be equivalent to traditional bitmap indices.

Alternative Bin Borders: In practice, marginals of our grid index do not have to be exactly uniform and the bins can be chosen independently and with some flexibility. For example, in our running example (see Sec.2.5), we use a separate bin for non-domain values (`NULL` and similar) in some attributes, which slightly skews the marginal distribution of the grid. The use of such out-of-support- or "sentinel" values is not uncommon in scientific applications and can yield spurious quantiles, resulting in non-uniform marginal distributions. Further, we found the chance of *precisely* evaluating predicates involving these special values (e.g., `A == NULL`) to be an efficient application of domain knowledge.

Choosing grids with (roughly) uniform marginals is the most robust choice in the absence of domain knowledge. However, if information on frequently queried regions of space is available, e.g., from previous queries, bin borders may be adapted accordingly. For example, using more bins in hot regions of an attribute domain can reduce false-positive rates. Note that the number of bins per attribute is ultimately limited, which implies a trade-off: when specializing bin borders, predicates formulated over other regions become more expensive or potentially non-selective.

Compression: An important factor influencing the performance of search engines and bitmap indices is the efficient persistence of posting lists (on storage) and their intersection (in memory); similar insights can be leveraged within our framework. Of particular interest are various compression schemes and encodings for (sorted) sets of unsigned integers, such as WAH [29], PFOR [18] and Roaring Bitmaps [9]. These schemes were developed to provide efficient set operation algorithms, such as (k-way) galloping intersection [4, 11, 17]. These specialized operations allow to adapt processing to the properties of the participating lists. For example, in case the first list is small, galloping (or *doubling search*) improves intersection speed by skipping increasingly larger portions of the second (larger) list, reducing the search cost for matching IDs asymptotically. See [27] for an overview on compressions.

Cost-based Optimization: Another relevant topic is the cost-based optimization of an execution plan for list intersections based on system and workload characteristics. Kim et al. [17] suggest a hardware-concious cost-based optimizer, using the CPU cache line size, and data characteristics to yield better runtime in the context of search engines. In our work, we focus on increasing inherent plan-level parallelism of index intersection, while further optimizations remain future work.

4 INDEX INTERSECTION & OPTIMIZATIONS

Logical runtime optimizations can greatly contribute to a more efficient index intersection, especially if it involves many leaves. In the following, we will describe *intersection push-down* and *leaf grouping* as two tuning knobs. The described measures are agnostic of the actual MDIS in use and we make no assumptions on whether a Team index has overlapping leaves or strictly partitions the data space. We will also discuss directions for further improvements and detail our (physical) prototype implementation.

4.1 Intersection Push-Down

Generally, intersecting indices can be implemented by first collecting (unifying) all component parts of each Team index result and then intersect the results, either pairwise or potentially all at once. An execution order (or *plan*) can be represented as a set-algebraic intersection expression. A (Team-) "union-first" intersection of three Team indices, A, B, and C, where each denotes *a set of leaves*, can be written as:

$$R = \Big(\bigcup_i A_i\Big) \cap \Big(\bigcup_j B_j\Big) \cap \Big(\bigcup_k C_k\Big), \tag{1}$$

where i, j and k enumerate the query-relevant leaves of every Team, respectively. However, a naive union-first approach has significant drawbacks: To facilitate efficient intersection as a second step, forming a union requires first fetching all IDs and then either sort them or insert them into a suitable data structure, such as a Roaring bitmap. A Team union represents a forced synchronization point (before any intersection can start) and produces a potentially large intermediate result, even if the query result may end up being empty. For our grid index, a union of leaves always produces the largest possible size, whereas MDIS with overlapping leaves requires (implicitly or explicitly) removing irrelevant/duplicate IDs. In addition, such a plan offers low inherent parallelism.

There are various ways to improve upon a union-first approach, increase potential parallelism, avoid costly unions and prune irrelevant IDs early. One way is to re-write the equation and push intersections into a common union operator. In its most extreme form, we obtain an "intersection-first" approach:

$$R = \bigcup_{i,j,k} \Big(A_i \ \cap \ B_j \ \cap \ C_k \Big) \tag{2}$$

where only actual result IDs survive an intersection chain and require unification. Although Eq. (2) minimizes the number of IDs we have to unify and increases the inherently available parallelism, it also forces us to consider every combination of lists separately. In this example, the big union indicates that there are $|A| \cdot |B| \cdot |C|$-many intersection terms, which quickly results in an infeasible number of intermediate results. For more than two Teams, it also results in redundant work, e.g., $A_i \cap B_j$ is (independently) evaluated k times. Thus, we opt for a middle ground, where we distribute the terms of C over combinations of (i, j) pairs and therefore "expand" only the terms of A and B:

$$R = \bigcup_{i,j} \underbrace{\Big[\bigcup_{k} \Big(\Big(A_i \ \cap \ B_j \Big) \ \cap \ C_k \Big) \Big]}_{\text{Independent Sub-Expression}}, \tag{3}$$

In this example, there are $|A| \cdot |B|$ many "Independent Sub-Expressions" (ISEs), where each evaluates $1 + |C|$ operations and requires two intermediate results: one for $A_i \cap B_j$, and another for the union over k. Note that longer ISEs may terminate early, if the result set becomes empty.

4.2 Leaf Grouping & Good ISE Count

To control the number of terms per Team, we employ partial unions over subsets of leaves ("leaf groups"), which allows us to reduce, e.g., $|A_i|$ many leaves to a freely configurable number of terms. Since this directly controls the number of ISEs, it provides another important lever to balance parallelism, memory consumption and overhead of the execution. Detailed information on the number of terms and the involved leaf cardinalities are readily available from the meta data of our simple grid-based index approach, before any leaf data has to be accessed (see Sec.2.4).

In practice, we found that expanding the smallest Team (by *accessed data volume*) first consistently yielded good results. In cases where the participating lists in the smallest Team were too few, we also expand a second Team to increase the ISE count for utilizing the available CPU threads more efficiently. We usually paired the expansion of a second Team with leaf grouping in order to ensure that the overall number of ISEs was never much larger than 2-3 times the number of CPU threads.

4.3 Further Logical Optimization

Beyond the already mentioned optimizations several others may be beneficial but are not discussed in depth in this work.

Intersection Order & Operator Selection: The order of intersecting more than two lists, i.e., 3 or more Teams, can impact performance. In our implementation, the order of (pairwise) ISE operations is determined globally, based on the total combined result sizes of the Teams. However, the ideal order within a specific ISE may differ. As a first step, reordering set operations individually based on the respective list cardinalities can already give performance improvements. However, while the low cardinality of one list is a useful and cheap indicator, the ideal processing order also depends on data dependencies and favorable tuple ordering. Thus, an accurate assessment requires additional information. For example, fixed-size, prebuilt MinHashes [8] can yield an estimation of the Jaccard index $J = |A_i \cap B_j| / |A_i \cup B_j|$ [20] and subsequently the expected volume of an intersection. A HyperLogLog sketch-based approach [22] may offer an attractive alternative for platforms that already utilize this data structure. This meta-indexing approach incurs a per-leaf increase in both storage footprint and runtime cost. Note that optimization should be considered in conjunction with the selection of "physical" k-way operator implementations. Aside from what Roaring's implementation already offers, we did not implement the selection of the most suitable *physical* implementation, e.g., for pairwise (or multi-way) intersection [4, 11],

Taking the Complement: In cases where individual Team results require accessing more than 50% of all IDs of a Team, it may be worthwhile to operate on the *complement* of the relevant leaves instead, i.e., on the IDs of bins that lie strictly outside of the query region. In place of set intersection, we then perform *set subtraction.* This optimization gives us the theoretical benefit of never requiring access to more than about 50% of any Team and can be straightforwardly integrated into Eq. (3). Because this use-case rarely applies when utilizing two or more dimensions per Team and due to space constraints, we refrain from discussing it further in this work. However, our implementation does support the optimization and it finds notable application in the edge case of 1-D Teams, where every predicate with $s > 0.5$ will always be considered as complement.

Streamed Execution: Depending on the application, we may skip the "big union" and directly use results from finished ISEs. Although this risks accessing the same location multiple times, it enables the seamless integration of subsequent operations—candidate refinement or `COUNT` aggregation—into the workflow and avoids the potentially unnecessary synchronization step.

4.4 Physical Implementation

Implementing the intersection of storage-resident leaf data requires employing various techniques in order to make use of modern hardware, most notably fast SSD storage (or networks), and multi-core CPUs. To this end, our C++ prototype implementation uses liburing [3] (with kernel polling /"IOPOLL") for asynchronous I/O, Roaring bitmaps [9] for set operations and the Taskflow library [15] for fine-grained parallelism and work-stealing-based load-balancing. To perform the intersection, we build a static, directed task graph of set operations over leaves initially retrieved from storage, with each operation being executed as soon as all resources are available.

Lastly, note that runtime for pre-processing, such as plan-building and optimization, were negligible for all shown configurations. In our current implementation, ISEs are executed sequentially in fixed, global order and the operation is considered finished when all result tuples are collected into one final Roaring bitmap (see "big union" over i, j in Eq. (3)). Roaring offers cheap short-circuit operations if all values are positioned in disjoint blocks or any one set is empty. For leaf grouping, assignment was greedy to to the smallest group for roughly balanced unions.

In some cases, our prototype was sensitive to fine-tuning of logical optimization parameters due to inefficient work balancing in situations involving many small leaves. Although manual fine-tuning was able to achieve better efficiency and more consistent performance, we refrained from doing so for the sake of consistency.

I/O: I/O requests were fully issued to leaves in the order of Team selectivity. We also implemented I/O-request merging for accesses to adjacent regions in the file, with mixed results. Although merging amortizes some of the I/O overhead, we disabled it, since it can also increase access latency and subsequently runtime. Further adapting the spawning of I/O requests to account for leaf group assignments and the order of necessity offer another avenue of optimization.

5 PERFORMANCE FACTORS

Having tools for guiding index intersection, we now require metrics to assess the performance of our execution. Intersection runtime is not necessarily only determined by the overall number of operations [17]. For example, intersection push-down increases the number of operations and union-first has the minimal number of operations but can significantly improve performance. We differentiate three categories of cost factors: *Volume*, *Overhead* and *Imbalance*.

5.1 Volume & Overhead

The influence of *Volume*, the sum of the corresponding *compressed* leaf sizes, is a straightforward—less work has to be done if fewer IDs are involved or if these IDs are highly compressible. Volume is reduced with higher predicate selectivity, but also with higher Team dimensionality by allowing to combine more predicates and therefore to reduce the involved ID volume drastically. However, the latter comes with a cost of additional *Overhead* costs due to a larger number of *relevant leaves* (or bins), as well as reduced compressibility. There are various forms of Overhead, e.g., the need for more tasks to run set operations in our execution plan, more I/O requests and increased read amplification due to storage padding. Eventually, at high Team dimensionality, Overhead is no longer amortized and essentially *paid per id*. Overhead costs are therefore the primary reason why we consider *moderately*-dim. Teams. With our Cartesian grid index, we consider Overhead costs to be proportional to the total number of *relevant leaves*. Occasionally, we will use the number of relevant *bins* and the union *cardinality* of query-relevant leaves as stand-ins in back-of-the-envelope calculations for Overhead and Volume, respectively.

As we will confirm in Fig. 5, Volume *decreases* exponentially with Team dimensionality, whereas Overhead *increases* exponentially. Thus, 1-D Teams have the lowest Overhead, but usually also the highest Volume. In contrast, D-dim. Teams consider the lowest Volume but also the highest Overhead. This implies an optimal

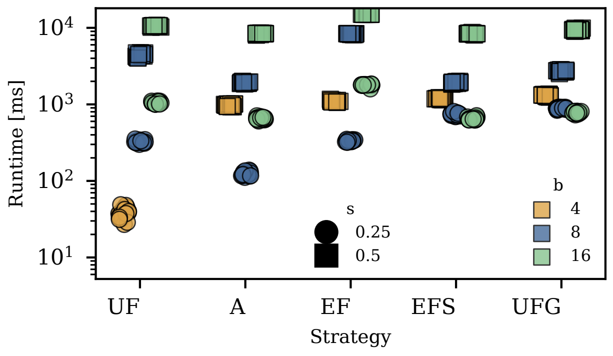

Figure 2: Runtime for different execution strategies over two sets of random queries and a uniform dataset.

configuration, where Volume reduction and Overhead increase balance out, but *Imbalance* factors can shift performance towards better or worse.

Evaluating Strategies: To first illustrate different execution strategies with respect to Volume and Overhead, we measured intersection runtime on a *uniform* dataset with $D = 12$ and one composition with 3 Teams, i.e., $d = 4$. To control Overhead, we varied $b \in \{4, 8, 16\}$, such that each Team has b^d bins. Queries were created randomly with equally selective predicates over all D dimensions with $s \in \{0.25, 0.5\}$, such that we access $(s \cdot b)^d \cdot 3$ many bins in every query. For uniform data, the overall selectivity of the query is given by s^D. We created 20 random queries for each pair (s, b) and executed them with different execution strategies, namely *union-first* (with and without grouping), as well as various forms of partial expansions. We expanded only one Team, exactly two Teams or between one to two Teams in an adaptive manner. Except for *expand-first*, we also varied the number of leaf groups. When applying grouping to a Team A, we reduced the number of leaves to $\sqrt{(|A|)}$. The runtime (log-scale) is shown in Fig. 2, with a small horizontal jitter to each group of measurements for visual clarity.

Performance generally degrades with larger Volume and increasing Overhead, i.e., larger s and number of leaves b. The strategy with the best and most consistent performance was an adaptive *expand-some*, which always expanded the first/smallest-Volume Team and potentially the second, if the first had too few lists (< 16). Grouping for expand-some was only applied (on either Team) when the number of ISEs grew beyond 128. We found that the union-first strategy can be improved with grouping for additional parallelism, but is still dominated by other strategies. Further, grouping is necessary for queries/indices with many leaves (cmp. expand-first), but can also deteriorate performance, especially for configurations with few lists ("expand first sqrt group" for $s = 0.25$ and $b = 8$). The execution strategy for $s = 0.25, b = 4$ is always trivial, since there is only a single list per Team and therefore no union required. We show the corresponding measurement in the union-first group. Overall, we observe that partial expansion is universally helpful, but may require additional grouping for queries that access many leaves. We will use the adaptive expansion strategy in the following.

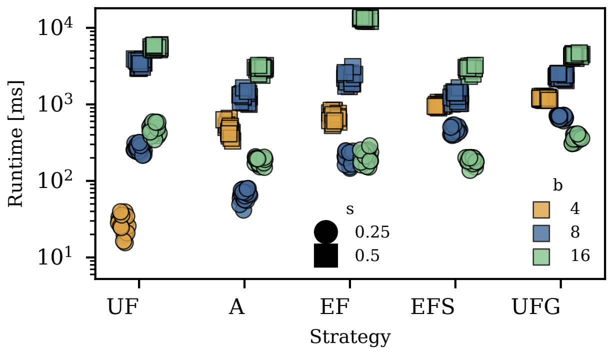

Figure 3: Runtime for different execution strategies over two sets of random queries and the LHCb dataset.

I/O Overhead: To assess the impact of storage access Overhead, we ran most of our measurements on already DRAM-resident index data as well. For example, for the experiments shown in Fig. 2, DRAM-based access improved intersection runtime by a factor of 1.16 × to 1.78 ×. The impact grew with larger b and smaller s, where leaves were smaller and more numerous, indicating that the costs are mostly associated with Overhead.

5.2 Imbalance

There are various Imbalance factors that impact Overhead, Volume and the efficiency of individual operations. The absence of data dependencies and differences in predicate selectivities across Teams presents a worst-case scenario for indexing for various reasons. First, given the lack of any favorable tuple order, compression efficiency of individual leaves deteriorates significantly. This results in higher Volume and storage costs, but set operations are also less efficient: When intersecting two lists from a uniform distribution, no clustering or other forms of exploitable structure imply that intersection essentially degrades to an exhaustive, pairwise comparison of IDs. Further, leaves are about equal in cardinality, which also degrades operation efficiency. If integer set cardinalities differ significantly, we can make use of specialized implementations for intersection, such as galloping [17]. Union operations similarly benefit from inter-attribute and inter-leaf dependencies within a grid. Further, differences in Team sizes and predicate selectivity can further impact the performance of index intersection as a whole.

Imbalanced Data Dependencies: To illustrate the impact of data-induced Imbalances, we ran the same benchmark as in Fig. 2 for the real LHCb dataset. As shown in Fig. 3, using real-world data yielded consistent performance improvement, e.g., about 1.85 × to 2.5 × faster for $b \in \{8, 16\}$ and $s \in \{0.25, 0.5\}$. Performance differences between the strategies shrank markedly, however the adaptive strategy still dominates. One reason for this improvement is the beneficial tuple order and correlations, which give overall better compressibility due to larger leaves and more efficient intersections. From a probabilistic view, the *conditional* bin cardinality can vary widely depending on where predicates restrict the support, even if individual selectivities stay the same.

Imbalanced Team Sizes: Team compositions over the same attributes but with different Team sizes can perform differently. To examine this in isolation, we ran 20 random queries with $b = 10$ and uniformly selective predicates $s = 0.4$ on two compositions: *balanced* (2×4-D Teams) and *imbalanced* (Teams of size 6 and 2), using both real and uniform LHCb.

Volume and Overhead differed markedly. The balanced Team required $2 \cdot 0.4^4 \cdot N \approx 0.0512N$ IDs, whereas the imbalanced Team $(0.4^6 + 0.4^2) \cdot N \approx 0.1641 \cdot N$ IDs—3.1 × more. Relevant bins also increased from $2 \cdot 4^4 = 512$ to $4^6 + 4^2 = 4112$. For uniform data, the imbalanced case was 1.5 × slower (0.919 vs. 0.583 s). On real data, the runtime slowdown shrank to 1.23 × (0.475 vs. 0.385 s) with compression reducing the Volume gap to 2.5 × and correlations between lists offering more efficient intersection. To verify this claim, we measured the CPU time difference: the imbalanced case needed 4.4 × more on real data (uniform: 5.6 ×). Grouping was also cheaper with a difference of 1.58 × vs. 2.45 × in the uniform case.

Overall, imbalance increases *Overhead* and access *Volume*, but actual runtime differences can be smaller.

Unused Attributes: Including non-selective attributes in an index adds overhead without improving selectivity. To illustrate, we compared a 2-D and a 6-D Team, both with selectivity $s = 0.04$, and each combined with another 4-D Team accessed with selectivities $0.3^4 \approx 0.0081$ and $0.7^4 \approx 0.24$. With the more selective 4-D Team (0.0081), intersecting via the 2-D Team was about 12 × faster (0.259 s vs. 3.145 s). With the less selective one (0.24), the speedup fell to 1.5 × (3.36 s vs. 5.28 s), since runtime was dominated by data volume. Such inefficiency from unused attributes affects both grid- and tree-based MDIS.

6 TEAM COMPOSITION

Both intersection performance and storage costs are impacted by how attributes are grouped, i.e., how Teams are composed. In this work we usually consider a Team composition to be a *disjoint set partition* of the table attributes, where every attribute appears in exactly one Team. For example, attributes $\{A, B, C, D, E\}$ may form composition $C = \{\{A, B\}, \{C, D, E\}\}$. Note that we will discuss in Sec.6.3 below that overlap between Teams may be beneficial and “augment” overall performance.

In the following, we will discuss the general design space of Team composition, and how we may approach Team composition.

6.1 Choosing Team Sizes

Although larger Teams allow combining more predicates/attributes for greater selectivity and therefore less Volume, growing Team size d also comes with escalating Overhead—regardless of the MDIS in use. In order to limit this Overhead, the Team size is effectively limited by table cardinality and the required precision per Team.

For non-adaptive grid index, a fixed Team composition and index precision b, we expect larger tables to be more efficiently indexed. With fixed parameters, Overhead costs, e.g., for formulating I/O requests or execution plan traversal, are essentially constant. An additional aspect is compression efficiency, which improves with leaf cardinality. However, these benefits only really matter for moderate Team sizes. Meaningfully increasing the average cardinality $(\frac{N}{b^d})$ of bins in high-dim. grids requires exponentially more data. To illustrate, we decided to run later experiments with the SDSS data set with only $d = 5$ (and $b = 8$) to ensure average leaf size is sufficient. Although strong data dependencies can help with concentrating the "mass" in only a subset of the high-dimensional grid's bins, the available table size is still ultimately limited. For the LHCb dataset, we found $d \approx 6$ dimensions (with $b \approx 10$) to be the limit.

Balanced Team Sizes: Given a limit for the maximum Team size d, we still have to decide on how to distribute D_{table} many attributes over $\lceil D_{\text{table}}/d \rceil$ or more Teams. As indicated in Sec.5.2, compositions with imbalanced Team sizes can may be preferable over unused Team attributes. However, under the assumption that all indexed attributes feature SSPs, balanced Teams of maximum size d can minimize storage costs and overall Volume for a larger workload. This still leaves a very large design space for assigning attributes: Even for true set partitions of D_{table}-many attributes, the number of possible composition grows with the Stirling numbers of the second kind [14], i.e., exponentially in D_{table}. For example, for $D_{\text{table}} = 16$ and $d = 4$, i.e., 4 Teams, we already have $S2(16, 4) = 171{\cdot}10^6$ possible compositions. We will discuss how to further reduce this design space after motivating additional criteria to assess compositions.

6.2 Correlation & Coverage

Attribute Correlations: One factor that impacts the value of compositions is the *correlation* between attributes. Correlation implies that data is more densely concentrated on some regions of data space than in others. By calculating pairwise statistics for Teams, we may deliberately encourage internal correlations, or discourage them by choosing Teams randomly.

For the grid index, correlated attributes cause some bins to be larger whereas others shrink or become empty. This further improves compression rates and can potentially lower Overhead (we observe this later in Sec.7). On the flip-side, a static grid is unable to further split these full leaves, potentially leading to additional false-positives. Adaptive index structures avoid this issue by adapting to the correlations. This, however, comes at the cost of a more complicated (and expensive) leaf geometry. For example, k-d-trees tend to produce partitions that are large in some dimensions but small in others, whereas quad-trees produce square cells [24], which makes the chance for false-positive overlap query-specific. High correlations within a Team therefore offer both advantages and disadvantages, depending on the index structure and the workload. However, correlations are also impactful across Teams. Or in other words, correlation not exploited by the Team indices can still be exploited during intersection.

Although we assume a workload agnostic stance in this work, we investigated the impact of Team composition in various experiments. One observation we consistently made was that the difference between randomly compositions is comparatively low (Fig. 3, Fig. 6), if compared to strategic decisions, such as Team size, the general Overhead of the index structure (total leaf count) and optimizations of the logical plan or the implementation. Most importantly, the impact of attribute correlations eventually diminishes for high dimensional queries (see Sec.7).

Attribute Coverage: Another important factor is the general *coverage* associated with a composition. A query with maximum coverage uses either all or none of any Team's attributes. We already showed that attributes without SSP incur additional costs (see Sec.5.2)—an issue generally shared by multi-dimensional index structures. Althought 1-D Teams offer the ability to skip irrelevant attributes, it is bought with the inability to combine predicates (as well as higher storage costs). Therefore, a strong priority in Team composition should be to maximize coverage (Sec.2.2) while allow for *moderate-dimensional* Teams. For the later experiments, we only consider queries with ideal coverage and leave a coverage-focused Team composition for future work.

6.3 Estimating Optimal Volume & Reusing Attributes

In this subsection, we share our initial thoughts on how Team compositions may be approached systematically. A possible strategy may be based on the *expected Volume* a composition may require for processing. The Volume (sans compression) is generally proportional to the Team-specific selectivity, which can be added up to assess the overall Volume of the intersection. This leads us to the general idea that selectivity is ideally spread across multiple Teams, instead of being concentrated into few. As a starting point, we estimate the combined selectivity per Team as a product of predicate selectivities, which assumes independence between the attributes. To illustrate why selectivity should be evenly distributed, consider that $0.1 \cdot 0.1 + 0.9 \cdot 0.9 = 0.82$ and $0.1 \cdot 0.9 + 0.9 \cdot 0.1 = 0.18$. Although performance generally improves with reduced Volume (see Sec.5), its influence, as we will confirm later, becomes more dominant for high dimensional queries.

Workload-Aware, Optimal Volume: If a query workload with a set of per-attribute selectivities is available, we can use it to systematically find a Team composition that minimizes Volume over an entire workload. For a Team composition C (set partition of D_{table} attributes), workload W (set of queries) and the selectivity $s_q(a)$ of query q and attribute a, we may minimize the objective function $\sum_{q \in W} \sum_{T \in C} \prod_{a \in T} s_q(a)$. Note that we also restrict the maximum Team size to d, which reflects a limit on Overhead. The term $\prod_{a \in T} s_q(a)$ is *supermodular* and (non-strictly) *monotone decreasing* in T: adding additional attributes/predicates cannot increase Volume, and the marginal decrease diminishes with increasing Team size. Exact minimization of a supermodular function is NP-hard, since it is equivalent to maximization of a submodular function [13]. However, there are various ways to simplify the problem and enable heuristics, for example by restricting $s_q(a)$ to a discrete domain. Another way would be to soften the requirement on Team count (or alternatively *size*), which turns it into a *weakly-α super-modular* problem [19]. Allowing overlap between Teams increases the solution space further and turns the problem into a set-cover problem, but could open up additional heuristics.

Volume-Optimality vs. Runtime: To show the actual impact of choosing Volume-optimal (as modeled above) Team compositions, we determined the ideal and worst compositions for our running example query using exhaustive search with strict partition requirement and Team count restriction. The measured runtime for $b = 10$ and $D_{\text{table}} = 12$ SSP attributes (Y-axis) is shown in Fig. 4. The only composition with $d = 1$ is also shown for reference (right).

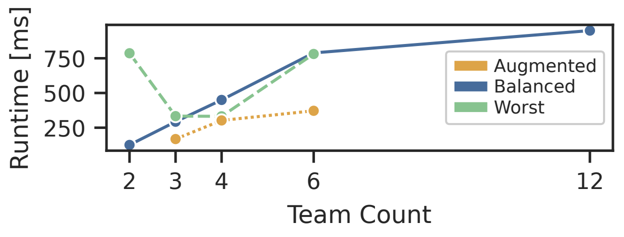

Figure 4: Intersection runtime for compositions with lowest and highest Volume. Lowest-Volume wins for larger Teams. Augmentation can improve performance.

The Volume difference between best and worst composition ranged from 1.25× (low d) to 10.52× (high d), but runtime was only different for $d = 6$, where the difference was maximized. Generally, for fixed b, d and number of Teams, a reduction in relevant Volume obtained solely by swapping attributes between Teams, correlates with a lower Overhead as well. This is most noticeable for $d = 6$, where the worst had about 3.5× as much Overhead as the best.

Overall, a holistic cost model for Team composition—beyond Volume estimation and simple constraints like Team size—should not only account for Overhead but also coverage.

Allowing Overlap: Another interesting application of selectivity information is the re-use of highly-selective attributes across multiple Teams. This can improve overall Team selectivity at the cost of storage and larger Overhead (or more relevant Teams) at runtime. To show the effect of this domain-specific optimization, we *augmented* the ideal compositions by adding the most selective attribute to each of the other Teams as well, increasing d. Results are also shown in Fig. 4. Although Overhead was larger, the reduced Volume from the additional selectivity significantly improved intersection runtime compared to compositions without overlap. Augmentation has diminishing returns for already high dimensional Teams due to already low selectivity.

6.4 Cost Factor Evaluation

In the following, we consider how different choices of d and b affect Overhead, Volume and also actual real-world performance, i.e., runtime and storage overhead.

Overhead vs. Volume: The reduction in access Volume with increasing Team dimensionality is not specific to the example query used above. Generally, as Volume decreases, Overhead, i.e., the number of relevant leaves, increases. To demonstrate this, we measured 16-D random queries and 10 random Team compositions for various levels of precision and dimensionalities on our LHCb dataset. We evaluated $b \in \{5, 10, 16\}$ bins per attribute, balanced Team compositions with $16/d$-many Teams and Team sizes $d \in \{1, 2, 4, 6, 8\}$. For $d = 6$, we used one 6-D Team and two 5-D Teams, otherwise Team sizes are equal. We omitted configurations with $b^d > 10^6$ due to prohibitively long runtimes. As workload, we generated five queries in three different *flavors*: Two with equal predicate selectivities of $16 \times s$ for $s \in \{0.2, 0.5\}$ ("balanced"/"balanced selective") and one with varying selectivities—$4 \times 0.2, 4 \times 0.4, 4 \times 0.6, 4 \times 0.8$ — denoted as "diverse". Overhead versus accessed Volume is shown in Fig. 5

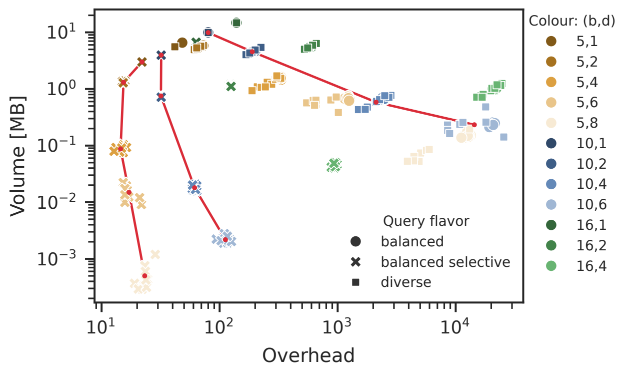

Figure 5: Number of accessed leaves (Overhead) vs. accessed data Volume for different configurations (d and b) and 10 balanced random Team compositions. Access Volume decreases as Team dimensionality—and thus Overhead—increases.

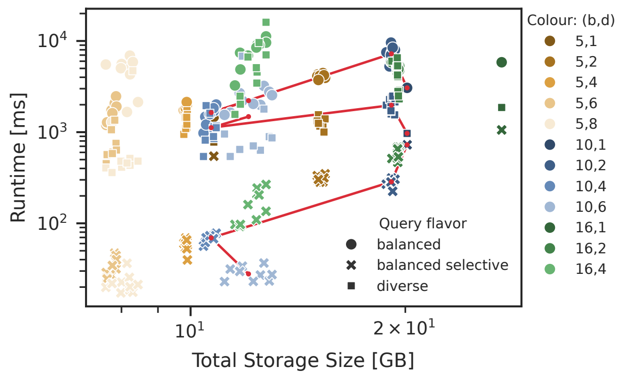

Figure 6: Storage costs vs. intersection runtime. 1-D Teams incur higher storage costs. For highly selectivity queries, higher dimensional Teams offer better runtime.

(log-log). A different color indicates a different index configurations, i.e., pairs (b, d), and symbols represent the query flavor. Points with identical color, hue and symbol represent different random indices. As indicated by the "constellations", points with the same color *group* (green, brown, blue) and symbol are directly comparable.

Generally, the trend from the upper-left (low d) to lower-right corner (high d) shows the exponential trade-off between Overhead and Volume. Further, larger b increases Overhead and slightly decreases Volume. Balanced, highly selective queries (×) require both less Overhead and access less Volume than less selective (•) or imbalanced (▪) query flavors. Within a specific configuration of b and d, i.e., identical color, accessing less data also implies fewer relevant leaves. Variance within configurations is low, with the exception "diverse" queries and $b = 10, d = 6$, where the average bin cardinality is only $N/b^d \approx 1221$. This causes both metrics to vary greatly due to differences in predicate selectivity.

Storage Costs vs. Runtime: The two core metrics for evaluating index performance are storage cost and runtime. When using the grid index, runtime is almost entirely determined by the cost of intersection. Fig. 6 shows the impact of Team compositions on these metrics for the same experiment as in Fig. 5. As before, measurements with the same symbol and color group (b) are directly comparable. One-dimensional Teams require more storage space with the notable exception of $b = 5$, where each attribute is represented with only five bitmaps. However, this advantage vanishes quickly for increased levels of precision. Otherwise, storage costs decrease as d increases (from right to left), due to a decrease in Team count. For configurations with many bins, i.e., $b = 5, d = 8$ and $b = 10, d = 6$, decreasing compressibility and (for this D_{table}) diminishing gains from fewer Teams again lead to a slight increase. Absolute (horizontal) variation across random Team compositions is low if bin count is not too large.

Results vary in runtime w.r.t query flavor and (b,d). *Balanced* queries perform worse than the *diverse* queries (black square), especially for small d. *Selective-balanced* was fastest by a wide margin, due to very small Volume *and* Overhead. Difference across flavors are primarily due to $2.5 - 10\times$ smaller results (depending on b) for the *diverse* queries and 0 (or very close) for the $\times$-flavor. Smaller results lead to faster execution, especially for low-dimensional Teams, where ISEs involve intersections across more Teams and a larger result implies that fewer ISEs can terminate early due to an empty result. Teams with $d = 2$ offer perform worst due to bad compressibility and selectivity.

The most interesting combination was $b = 10$ for the diverse queries, where 1-D Teams were outperformed by some compositions of $d = 4, 6$, whereas others performed significantly worse and had large variance. We found that the composition-dependent differences were caused by multiple factors. First, our implementation lacks the ability to alter intersection orders per-ISE and is unable to adapt to size differences on the leaf level. For larger d and large variety of predicate selectivities, differences in data dependencies were more likely to be noticeable for this flavor. This is also reflected in the larger variance of Volume- and Overhead shown in Fig. 5.

Discussion: In our experiments, we observed that random compositions performed similar and general configuration (b and d) were much more impactful. Further, large Teams can reduce Volume, but this scaling has limits and gets diminished by Overhead costs. For highly selectivity predicates, e.g., $s = 0.2$, Team-based indexing yields significant reductions in both storage footprint and runtime.

7 SCALING

Early Team Intersections: To quantify scaling behavior, we evaluated runtime while gradually increasing query dimensionality D by starting with a single attribute and adding new ones (in table order). This also drops selectivity and increases Overhead and Volume. We investigate the LHCb dataset with $D_{\text{table}} = 16$, which showcases the performance of intersecting few Teams. Keeping $s = 0.4$ constant for all new predicates, grid indices had $b = 10$ and $d \in \{2, 4\}$ for $D \in \{1, 2, 4, \ldots, D_{\text{table}}\}$, as well as 20 random queries per D. We add two attributes at a time, i.e., the index for $D = 6$ has two Teams of size 4 and 2, $D = 8$ two Teams of size 4, etc. Fig. 7 shows runtime versus selectivity (log-log scale) for each query and index.

Performance *improves* by adding Teams and therefore increasing query selectivity, both across dimensionality groups and between

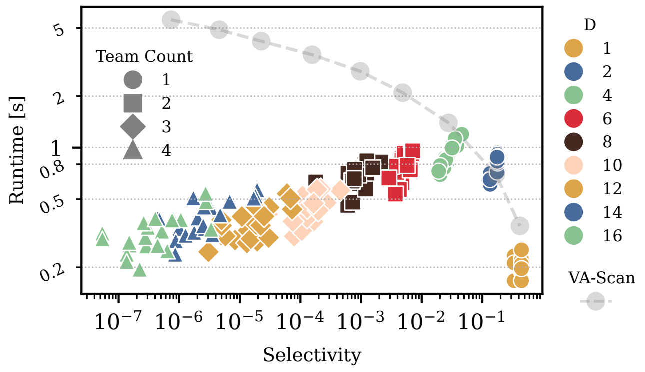

Figure 7: Intersection runtime (log y-axis) for increasing query dimensionality D/predicate count and fixed $s = 0.4$. For small D, runtime decreases with selectivity.

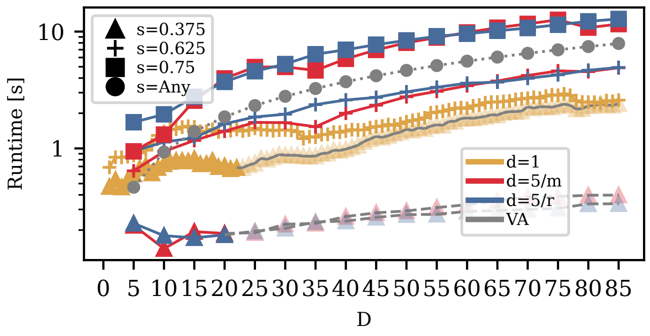

Figure 8: Intersection runtime (log y-axis) for increasing D. Runtime scales linearly and Team compositions become similar. VA-files and Teams with $d = 1$ are faster for large s.

queries (same symbol & color). For few Teams, *intersection* translates increasing query selectivity into a faster runtime: a union over many small leaves is costly and pruning via intersection reduces intermediate result size–and thus runtime–considerably. For few Teams and predicates, intermediate results are initially relatively large and set operation cost outweighs the cost of fetching more leaves. Moreover, gradual addition of Teams offers new opportunities to start the intersection order: freshly added Teams with high selectivity "jump the line" and intersect early, reducing runtime in the process. This behavior requires sufficient s, or the cost of fetch costs diminish any benefits from early pruning (see below).

For reference, we have also calculated the *ideal* runtime of evaluating VA-files—also build over Teams and without the cost of intersecting multiple VA-files—which is bottle-necked by PCIe bandwidth (of 14 GB/s). Due to the inverted access, Team-based indices outperform VA-files starting with the first full Team ($D = d = 4$). An interesting outlier is the low runtime for $D = 1$ (lower right corner), which only requires retrieving and unifying four (of $b = 10$) highly compressible leaves.

High-Dimensional Queries: For high-dimensional queries, we expect runtime to eventually increase by a constant factor (per additional Team). After the *operation-dominated phase*, the working set is considerably reduced. This leads to (pairwise) intersection operations between very disproportionately sized sets. Theoretically, the grid index would allow for a single surviving ID to be compared to a leaf with maximum cardinality of N/b, which essentially implies a single set membership test. Thus, compared to the cost of fetching leaves from storage in the first place, set operations become cheap and we enter a *fetch-dominated phase* of index intersection.

Note that Independent Sub-Expressions (see Sec.4.1) with empty intermediate sets finish processing early and allow skipping (or short-circuiting) subsequent operations, effectively removing these sets from the final result. However, the presence of even a single unfinished ISE still requires fetching and intersecting *all* leaves of later Teams. As a result, I/O can only be skipped if *all* partial results become empty, resulting in an empty set overall.

Linear Scaling: To show how the cost of index intersection evolves for high-dimensional queries, we performed a similar experiment as in Fig. 7 for the SDSS dataset, which features up to 85 dimensions. We varied selectivity $s \in \{0.375, 0.625, 0.75\}$ and created indices for $b = 8$ in three different flavors: a manually-chosen composition (indicated as *m*; see below) and a random composition (indicated as *r*) with $d = 5$ each, and a composition with $d = 1$ that represents bitmap indices. The manual composition formed Teams of 5 wavelength bands (see Sec.2.5). The three indices required 13, 17 and 27 GB of storage for *m*, *r* and $d = 1$, respectively. Runtime averages over 5 random queries and 5 repetitions (log scale) vs. query dimensionality (linear scale) are shown in Fig. 8.

Starting with a single Team, we observe that adding more Teams generally grows runtime by a constant factor (we also show VA-file runtime for reference). Similar to Fig. 7, we can also observe the initial dip in runtime for $s = 0.375$. Note that the result (of all 5 queries) eventually becomes *empty* and execution may be terminated early (as indicated by gray line continuations).

Team Composition vs. D: To show the impact of Team composition for large D, we compare a *hand-crafted-* and a randomly drawn Team composition (*r*) for the SDSS dataset. In the *manual* composition (*m*), wavelength measurements can have very strong correlation (up to Pearson coefficient of 1) compared to *r*, where strong correlations are unlikely. To illustrate, the *smallest* average pairwise correlation within any Team of *m* is 0.27, whereas *random*'s averages are *at most* 0.28. Higher combined selectivity made it more likely for correlated Teams to jump the intersection chain. Accordingly, runtime is lower early (where pruning potential matters most), but can still drop runtime later on. For $D = 35$, the new Team offered significant pruning at significantly lower Overhead cost: *r* processed 42851 leaves in 6.3s, whereas *m* requires 4.6s for only 30868. However, *r* does occasionally outperform *m* for $s \geq 0.625$ and most D at $s > 0.325$. Moreover, in the fetch-dominated phase—where queries are truly high-dimensional—runtime differences due to Team composition eventually become negligible.

Baseline Comparison: For reference, we have again calculated the ideal time of evaluating VA-files. Due to the inverted access, Team based indices outperform VA-files for selective queries (around $0.625 < s < 0.75$) due to their inability to skip any data. Similar results can be seen for $d = 1$, albeit for lower s. For $s = 0.375$, Teams with $d = 5$ members can reduce overall Volume by up to $80\times$ (for $D = 85$) and subsequently offer a speedup of $6\times$ to $7\times$.

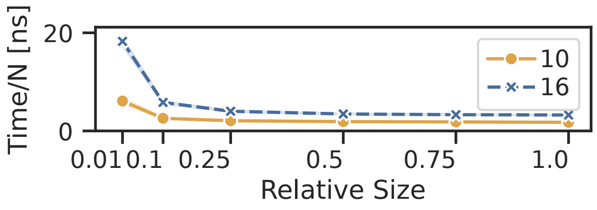

Figure 9: Runtime per database tuple for increasingly larger tables. Overhead costs are quickly amortized.

Here, 1-D Teams can not exploit *combined* selectivity and are bound by Volume alone (Overhead is 3 per Team).

7.1 Varying Table Size

To confirm our earlier claim that Overhead is amortized for larger tables, we measured runtime per tuple for our running example query, a fixed 4×4 Team composition and $b \in \{10, 16\}$. Indices were build over increasingly more tuples from the LHCb dataset ($N = 1.222 \cdot 10^9$ tuple for scale 1.0) to retain the beneficial order and thus compression efficiency. The runtime per tuple is shown in Fig. 9: The constant Overhead for this workload was quickly amortized and is only noticeable for small tables.

8 CONCLUSION

Team-based indexing is not a replacement for already existing MDIS, such as the R^*-tree. Rather, it is a framework that generalizes the approach of bitmap indices and focuses on efficient intersection of partial results. By exploiting combined selectivity, Team-based indexing improves uppon our baselines—bitmap indices and VA-files—and enables index access for high-dimensional queries and reduces overall storage costs to a fraction of the table size.

Although our investigations indicate that even conservative design decisions, such as usage of an inaccurate grid index, show promising results, many avenues for further optimization still remain. For instance, our implementation was not yet able to fully capitalize on the significant reduction in intersection Volume and thus efficiently translate query selectivity into performance, which hinders the investigation of more subtle aspects, such as Team composition. Further directions are physical and logical plan optimizations for the intersection, cost-based Team composition, meta-indexing and a thorough investigation on the impact of different space-partitioning strategies with an emphasize to support efficient intersection. Lastly, many of these directions require suitable test workloads to allow fine-tuning Team-based indexing and fully explore its potential.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their valuable feedback. We also thank the particle physics group at TU Dortmund University for their support and for helping us understand data processing at the LHCb experiment at CERN. The work has partly been funded by the Federal Ministry of Education and Research of Germany and the state of North-Rhine Westphalia as a part of the Lamarr-Institute for Machine Learning and Artificial Intelligence.

REFERENCES

[1] R Aaij, J Albrecht, F Alessio, S Amato, E Aslanides, I Belyaev, M Van Beuzekom, E Bonaccorsi, R Bonnefoy, L Brarda, et al. 2013. The LHCb trigger and its performance in 2011. *Journal of Instrumentation* 8, 04 (2013), P04022.

[2] Apache Software Foundation. 2013. Apache Parquet Format Specification. https://github.com/apache/parquet-format. Version 2.9.0, Accessed: 2025-05-31.

[3] Jens Axboe. 2025. liburing: io_uring library for Linux. https://github.com/axboe/liburing. Accessed: 2025-05-31.

[4] Ricardo Baeza-Yates and Alejandro Salinger. 2010. Fast Intersection Algorithms for Sorted Sequences. In *Algorithms and Applications, Essays Dedicated to Esko Ukkonen on the Occasion of His 60th Birthday (Lecture Notes in Computer Science)*, Tapio Elomaa, Heikki Mannila, and Pekka Orponen (Eds.), Vol. 6060. Springer, 45–61. https://doi.org/10.1007/978-3-642-12476-1_3

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*, Hector Garcia-Molina and H. V. Jagadish (Eds.). ACM Press, 322–331. https://doi.org/10.1145/93597.98741

[6] Stefan Berchtold, Christian Böhm, Daniel A. Keim, Hans-Peter Kriegel, and Xiaowei Xu. 2000. Optimal Multidimensional Query Processing Using Tree Striping. In *Data Warehousing and Knowledge Discovery, Second International Conference, DaWaK 2000, London, UK, September 4-6, 2000, Proceedings (Lecture Notes in Computer Science)*, Yahiko Kambayashi, Mukesh K. Mohania, and A Min Tjoa (Eds.), Vol. 1874. Springer, 244–257. https://doi.org/10.1007/3-540-44466-1_24

[7] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree : An Index Structure for High-Dimensional Data. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda (Eds.). Morgan Kaufmann, 28–39. http://www.vldb.org/conf/1996/P028.PDF

[8] Andrei Z Broder. 2000. Identifying and filtering near-duplicate documents. In *Annual symposium on combinatorial pattern matching*. Springer, 1–10.

[9] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with Roaring bitmaps. *Softw. Pract. Exp.* 46, 5 (2016), 709–719. https://doi.org/10.1002/SPE.2325

[10] Yann Collet and Murray S. Kucherawy. 2021. Zstandard Compression and the 'application/zstd' Media Type. *RFC* 8878 (2021), 1–45. https://doi.org/10.17487/RFC8878

[11] J. Shane Culpepper and Alistair Moffat. 2010. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.* 29, 1 (2010), 1:1–1:25. https://doi.org/10.1145/1877766.1877767

[12] Jeff Dean, Sanjay Ghemawat, and Steinar H. Gunderson. 2011. Snappy: A Fast Compressor/Decompressor. https://github.com/google/snappy. Accessed: 2025-05-31.

[13] Uriel Feige, Vahab S. Mirrokni, and Jan Vondrák. 2011. Maximizing Non-monotone Submodular Functions. *SIAM J. Comput.* 40, 4 (2011), 1133–1153. https://doi.org/10.1137/090779346

[14] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science, 2nd Ed.* Addison-Wesley. https://www-cs-faculty.stanford.edu/%7Eknuth/gkp.html

[15] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Trans. Parallel Distributed Syst.* 33, 6 (2022), 1303–1320. https://doi.org/10.1109/TPDS.2021.3104255

[16] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 2011. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.* 33, 1 (2011), 117–128. https://doi.org/10.1109/TPAMI.2010.57

[17] Sunghwan Kim, Taesung Lee, Seung-won Hwang, and Sameh Elnikety. 2018. List Intersection for Web Search: Algorithms, Cost Models, and Optimizations. *Proc. VLDB Endow.* 12, 1 (2018), 1–13. https://doi.org/10.14778/3275536.3275537

[18] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw. Pract. Exp.* 45, 1 (2015), 1–29. https://doi.org/10.1002/SPE.2203

[19] Edo Liberty and Maxim Sviridenko. 2017. Greedy Minimization of Weakly Supermodular Set Functions. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA (LIPIcs)*, Klaus Jansen, José D. P. Rolim, David Williamson, and Santosh S. Vempala (Eds.), Vol. 81. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:11. https://doi.org/10.4230/LIPICS.APPROX-RANDOM.2017.19

[20] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval.* Cambridge University Press. https://doi.org/10.1017/CBO9780511809071

[21] Charles Masson, Jee E. Rim, and Homin K. Lee. 2019. DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees. *Proc. VLDB Endow.* 12, 12 (2019), 2195–2205. https://doi.org/10.14778/3352063.3352135

[22] Magnus Müller, Daniel Flachs, and Guido Moerkotte. 2021. Memory-Efficient Key/Foreign-Key Join Size Estimation via Multiplicity and Intersection Size. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021.* IEEE, 984–995. https://doi.org/10.1109/ICDE51399.2021.00090

[23] Doron Rotem, Kurt Stockinger, and Kesheng Wu. 2004. Efficient binning for bitmap indices on high-cardinality attributes. (2004).

[24] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures.* Academic Press.

[25] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97. https://doi.org/10.14778/3485450.3485459

[26] Hermann Tropf and Helmut Herzog. 1981. Multimensional Range Search in Dynamically Balanced Trees. *Angew. Inform.* 23, 2 (1981), 71–77.

[27] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 993–1008. https://doi.org/10.1145/3035918.3064007

[28] Roger Weber, Hans-Jörg Schek, and Stephen Blott. 1998. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 194–205. http://www.vldb.org/conf/1998/p194.pdf

[29] Kesheng Wu, Sean Ahern, E Wes Bethel, Jacqueline Chen, Hank Childs, Estelle Cormier-Michel, Cameron Geddes, Junmin Gu, Hans Hagen, Bernd Hamann, et al. 2009. FastBit: interactively searching massive data. In *Journal of Physics: Conference Series*, Vol. 180. IOP Publishing, 012053.

[30] Donald G. York, J. Adelman, John E. Anderson, Jr., Scott F. Anderson, James Annis, Neta A. Bahcall, J. A. Bakken, Robert Barkhouser, Steven Bastian, Eileen Berman, William N. Boroski, Steve Bracker, Charlie Briegel, John W. Briggs, J. Brinkmann, Robert Brunner, Scott Burles, Larry Carey, Michael A. Carr, Francisco J. Castander, Bing Chen, Patrick L. Colestock, A. J. Connolly, J. H. Crocker, István Csabai, Paul C. Czarapata, John Eric Davis, Mamoru Doi, Tom Dombeck, Daniel Eisenstein, Nancy Ellman, Brian R. Elms, Michael L. Evans, Xiaohui Fan, Glenn R. Federwitz, Larry Fiscelli, Scott Friedman, Joshua A. Frieman, Masataka Fukugita, Bruce Gillespie, James E. Gunn, Vijay K. Gurbani, Ernst de Haas, Merle Haldeman, Frederick H. Harris, J. Hayes, Timothy M. Heckman, G. S. Hennessy, Robert B. Hindsley, Scott Holm, Donald J. Holmgren, Chi-hao Huang, Charles Hull, Don Husby, Shin-Ichi Ichikawa, Takashi Ichikawa, Željko Ivezić, Stephen Kent, Rita S. J. Kim, E. Kinney, Mark Klaene, A. N. Kleinman, S. Kleinman, G. R. Knapp, John Korienek, Richard G. Kron, Peter Z. Kunszt, D. Q. Lamb, B. Lee, R. French Leger, Siriluk Limmongkol, Carl Lindenmeyer, Daniel C. Long, Craig Loomis, Jon Loveday, Rich Lucinio, Robert H. Lupton, Bryan MacKinnon, Edward J. Mannery, P. M. Mantsch, Bruce Margon, Peregrine McGehee, Timothy A. McKay, Avery Meiksin, Aronne Merelli, David G. Monet, Jeffrey A. Munn, Vijay K. Narayanan, Thomas Nash, Eric Neilsen, Rich Neswold, Heidi Jo Newberg, R. C. Nichol, Tom Nicinski, Mario Nonino, Norio Okada, Sadanori Okamura, Jeremiah P. Ostriker, Russell Owen, A. George Pauls, John Peoples, R. L. Peterson, Donald Petravick, Jeffrey R. Pier, Adrian Pope, Ruth Pordes, Angela Prosapio, Ron Rechenmacher, Thomas R. Quinn, Gordon T. Richards, Michael W. Richmond, Claudio H. Rivetta, Constance M. Rockosi, Kurt Ruthmansdorfer, Dale Sandford, David J. Schlegel, Donald P. Schneider, Maki Sekiguchi, Gary Sergey, Kazuhiro Shimasaku, Walter A. Siegmund, Stephen Smee, J. Allyn Smith, S. Snedden, R. Stone, Chris Stoughton, Michael A. Strauss, Christopher Stubbs, Mark SubbaRao, Alexander S. Szalay, Istvan Szapudi, Gyula P. Szokoly, Anirudda R. Thakar, Christy Tremonti, Douglas L. Tucker, Alan Uomoto, Dan Vanden Berk, Michael S. Vogeley, Patrick Waddell, Shu-i Wang, Masaru Watanabe, David H. Weinberg, Brian Yanny, and Naoki Yasuda. 2000. The Sloan Digital Sky Survey: Technical Summary. *The Astronomical Journal* 120, 3 (sep 2000), 1579. https://doi.org/10.1086/301513