Fast Set Intersection through Run-time Bitmap Construction over PForDelta-compressed Indexes

Xiaocheng Zou[1,2], Sriram Lakshminarasimhan[3], David A. Boyuka II[1,2], Stephen Ranshous[1,2], Houjun Tang[1,2], Scott Klasky[2], and Nagiza F. Samatova[1,2]

[1] North Carolina State University, Raleigh, NC 27695, USA
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA
[3] IBM India Research Lab, Bangalore - 560045, India
[4] Corresponding author:samatova@csc.ncsu.edu

Abstract. Set intersection is a fundamental operation for evaluating conjunctive queries in the context of scientific data analysis. The state-of-the-art approach in performing set intersection, compressed bitmap indexing, achieves high computational efficiency because of cheap bitwise operations; however, overall efficiency is often nullified by the HPC I/O bottleneck, because compressed bitmap indexes typically exhibit a heavy storage footprint. Conversely, the recently-presented PForDelta-compressed index has been demonstrated to be storage-lightweight, but has limited performance for set intersection. Thus, a more effective set intersection approach should be efficient in both computation and I/O.
Therefore, we propose a fast set intersection approach that couples the storage light-weight PForDelta indexing format with computationally-efficient bitmaps through a specialized on-the-fly conversion. The resultant challenge is to ensure this conversion process is fast enough to maintain the performance gains from both PForDelta and the bitmaps. To this end, we contribute two key enhancements to PForDelta, *BitRun* and *BitExp*, which improve bitmap conversion through bulk bit-setting and a more streamlined PForDelta decoding process, respectively. Our experimental results show that our integrated PForDelta-bitmap method speeds up conjunctive queries by up to 7.7x versus the state-of-the-art approach, while using indexes that require 15%-60% less storage in most cases.

1 Introduction

Set intersection is a fundamental operation for evaluating conjunctive queries in the context of scientific data analysis, as well as in other fields [1,2]. For example, consider the following conjunctive query used in detecting atmospheric rivers [3]: "water vapor $>$ 20mm, length $>$ 2000km and width $<$1000km, and spatial constraints". In order to answer this query (i.e., retrieve the common records satisfying all constraints), it is necessary to intersect multiple record sets, each of which satisfies an individual query constraint. Indexes are often used in support of such intersection operations, as they accelerate the evaluation of the individual constraints.

Compressed bitmap indexing methods are commonly used in scientific data analysis to support set intersections [4–6]. These methods successfully provide computationally-efficient set intersections by means of cheap bitwise operations. However, this efficiency

may be nullified by the I/O bottleneck common in High-Performance Computing (HPC) environments, as compressed bitmap indexes often have large storage footprints [7]. Thus, effective set intersections in an HPC context require not only efficient computation, but also low I/O overhead.

Recently, PForDelta-compressed indexes have been demonstrated as a promising alternative to compressed bitmap indexes [8, 9]. By using PForDelta compression [10–12], a dynamic data-locality-based encoding, these indexes are often able to achieve multi-fold storage reduction relative to compressed bitmap indexes. However, existing algorithms for set intersections on PForDelta-compressed indexes are not able to deliver good compute performance in the context of HPC. For example,"comparison-based" set intersection methods [13–15] usually require sorted set operands, whereas a PForDelta-compressed index over scientific data is typically value-partitioned (see Section 2.2), resulting in a partially-sorted data structure that is expensive to sort on-the-fly. Likewise, the "skipping" method [2, 16] shares this sorting requirement, and further assumes that at least one individual constraint is highly selective, a condition not always met in scientific analysis. Thus, while PForDelta indexes surmount the I/O bottleneck challenge, existing PForDelta set intersection methods conversely fall short in computational efficiency.

Therefore, we propose a new set intersection approach based on coupling the PForDelta indexing format and bitmaps through a specialized on-the-fly conversion. Our key insight is that PForDelta indexing and bitmap intersection can bring lightweight I/O and computational-efficiency benefits to the set intersection operation, respectively.

The challenge is to ensure this conversion process is fast enough to avoid diminishing the performance gains from this integrated approach. To this end, we contribute two key enhancements to PForDelta, *BitRun* and *BitExp*, that greatly improve bitmap conversion through bulk bit-setting and a more streamlined PForDelta decoding process, respectively. Our experimental results show that our integrated PForDelta-bitmap approach speeds up conjunctive queries by up to 7.7x versus the state-of-the-art pure bitmap-based method, while using 15-60% less index storage space in most cases.

2 Related Work and Background

2.1 Set Intersection

Methods for achieving efficient set intersection are a focus in several fields of research. For scientific data analysis, compressed bitmaps are the state-of-the-art approach for set intersections. Word-aligned Hybrid (WAH) [4–6], a prominent compressed-bitmap approach, achieves compression by capturing long runs of 0-bits and 1-bits with *fill words*. Each fill word begins with a "fill flag bit," followed by the fill type (0 or 1) and count. Bits not part of a run are encoded using *literal words*, denoted by a "literal flag bit," which simply contain the exact bits to encode. During set intersection, words from multiple bitmaps are "matched up" in a bitwise AND operation. Fill words enable large segments of bitmap to be processed in bulk, saving a substantial compute time. However, compressed bitmaps often exhibit heavy storage footprints [7], which is an impediment in an HPC context due to the ubiquitous I/O bottleneck.

In fields such as information retrieval and web search, "comparison-based" set intersection methods are more common [13–15]. Most research on such methods aims to reduce the asymptotic complexity and/or practical run time of set intersections over uncompressed, sorted integer lists. Another, similar technique is the "skipping" method [2, 16], which performs set intersection over PForDelta-compressed sets. By leveraging min/max element metadata in each chunk, this method is able to entirely skip some compressed chunks, reducing overall decompression cost.

While well-suited for their designed purposes, these set intersection methods do not translate well to scientific data indexing and querying. First, most are not designed to operate on compressed indexes (unlike our approach); second, they assume sorted operand sets, which is not the case for finely-partitioned value-binning indexes (explained next); third, some additional, method-specific limitations may apply (e.g., the skipping method depends on high skewness in query constraint selectivities, which does not hold for many scientific queries). Our methodology aims to overcome these barriers.

2.2 PForDelta-compressed indexes

PForDelta-compressed indexing is a recent approach to scientific data indexing. It is built on *value binning*, an indexing approach which groups data that have similar values into histogram bins to produce broader bin values. "Similar" may be defined several ways, such as bit-level binning [7], interval binning [6], and equality binning [6]. In this work, we elect to use bit-level binning, which partitions data according to the 16 most significant bits of their floating-point representation (equivalent to roughly 1 to 2 base-10 significant digits in scientific notation). The binning process collects the *record IDs* (or *RIDs*, integer IDs specifying datapoint locations via some linearization) of all values contained in each bin. These bins of RIDs are then compressed using PForDelta.

PForDelta compression operates on a list of integers (record IDs, in this case) in a chunk-wise manner. Within each fixed-size chunk, PForDelta computes the deltas between consecutive integers, then selects a reduced bit-width b (bits-per-element) that can encode the majority of these deltas. The remaining deltas that can not be encoded with b bits are termed *exceptions*, and are stored uncompressed alongside the bit-packed deltas. Since most delta values may typically be encoded using far fewer bits than the standard word size, a high compression ratio can be achieved, especially when some data locality is present in the original RIDs, which translates to many small deltas.

Some refinements to the base PForDelta encoding have been proposed. For example, Zhang et al. [11] use a flexible number of bits to encode exceptions, and Yan et al. [12] use the vacant b-bit-wide slots at the exception positions in the delta list to store the low bits of exception values. In this work, we leverage the version of PForDelta used in our previous work [9], as its design is targeted to achieve good compression on scientific data indexes. Specifically, we use 0 delta values to identify exception positions, instead of the explicit offset list used in other works. We also incorporate Zhang et al. [11]'s variable exception bit-width approach. These modifications form a baseline for our work; we present additional, new PForDelta enhancements in Section 3.

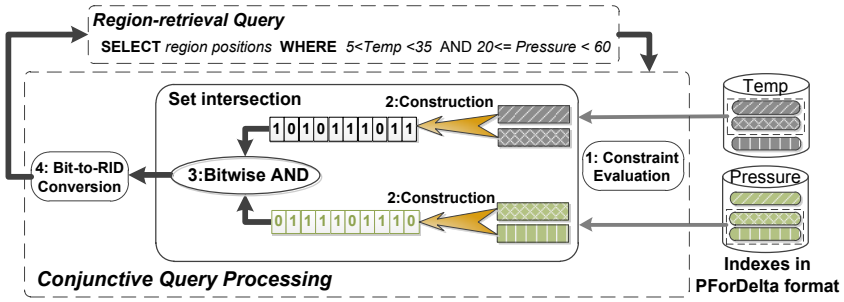


Fig. 1. Overview of our set intersection approach using PForDelta-compressed indexes fits into conjunctive query processing.

3 Method

To recap, both I/O and computational efficiency barriers must be overcome to achieve fast set intersection in an HPC context. Our approach is to couple the storage-lightweight PForDelta indexing format with computationally-efficient bitmaps for set intersections via on-the-fly PForDelta-to-bitmap conversion. However, to retain the benefits of PForDelta and bitmaps, it is critical to ensure the efficiency of the conversion process.

To this end, after reviewing the flow of conjunctive query processing in Section 3.1, we present two techniques, *BitRun* (Section 3.2) and *BitExp* (Section 3.3), that enhance PForDelta to increase bitmap conversion throughput. *BitRun* uses a run-length encoding approach to speed up bitmap construction for high-data-locality scientific datasets, whereas *BitExp* tunes PForDelta to increase decoding throughput on data that show moderate locality. Additionally, in Section 3.4, we explain how to apply *BitRun* and *BitExp* simultaneously in a flexible, combined approach.

3.1 The Role of Set Intersection in Conjunctive Query Processing

Figure 1 illustrates an overall flow of conjunctive query processing, which is the context of our methods. Initially, each scientific dataset is indexed by *value binning*, which partitions the record IDs (RIDs) of data elements into a series of bins, each of which has associated value interval. Each index bins is then compressed using PForDelta, which compresses the list of RIDs in a chunk-wise manner.

When a conjunctive query arrives, each individual constraint in the conjunction is evaluated using its respective index, causing a subset of bins to be retrieved. These retrieved bins serve as the inputs to the set intersection operation, which proceeds in three steps. First, these retrieved bins for each query constraint are converted to a single bitmap (which is also an implicit union operation). Second, the bitmaps built from these constraints are then intersected via bitwise AND operations, which yield a single, final bitmap. Last, the final bitmap is transformed to a list of RIDs (using a lookup table for speed), which serves as the region result for the original query. Note that, in

this work, we limit our consideration to region-retrieval queries with value constraints aligned to bin boundaries; generic methods for supporting unaligned constraint have been developed [6], and could easily be applied on top of our approach.

In addressing the key step of PForDelta-to-bitmap conversion, a straightforward approach is to first decode each retrieved bin, and then scan the resulting RID lists, setting the appropriate bits in the bitmap. However, this requires making two complete passes (first to decompress, then to set bits), and mandates retaining the intermediate uncompressed form in memory during this process, leading to poor cache efficiency and high memory usage. As a refinement, we instead set bits in the bitmap immediately after each PForDelta chunk is decoded, resulting in memory and cache efficiency. Additionally, to limit the output bitmap size when the original dataset is large (since each datapoint requires one bit in the bitmap), we perform "data partitioning" during index building to split the dataset into manageable blocks, preventing bitmaps from becoming unwieldy during query processing.

However, there is still substantial room for improvement, and thus, we devote our effort to improving the efficiency of bitmap conversion in the following subsections.

3.2 *BitRun*: Incorporating Run-length Encoding into PForDelta

A frequently observed phenomenon in scientific simulations is the data locality, in which data from contiguous time-steps or spatial regions exhibits very close value. Consequently, when the value binning is applied on the datasets, many RIDs in the indexes show a high degree of consecutiveness.

However, PForDelta is not efficient for building a bitmap from consecutive RIDs, and so a more specialized encoding within PForDelta is needed to optimize for this case. To see this, consider the example illustrated in Figure 2, which shows how PForDelta encodes a chunk of mostly-sequential RIDs. The first step of PForDelta encoding calculates the deltas between consecutive RIDs, producing a list of deltas, most of which are equal to 1. In the second step, the encoding bit-width is determined to be 1, since most deltas can be encoded in a single bit. Lastly, the deltas are bit-packed, producing the packed delta array B and the exception array E.

In the above example, suppose this encoded chunk were converted to a bitmap. This bitmap would then exhibit a few long sequences of 1-bits and 0-bits. Unfortunately, during this conversion, each 1-bit in the bitmap is set individually, because each corresponding RID is decoded separately in PForDelta. If we had a more concise encoding for long sequences of consecutive 1-bits, then these bits could be set in bulk, which is potentially more efficient.

In response to this, we develop *BitRun*, a method inspired by run-length encoding, to speed up bitmap construction by capturing the bit consecutiveness of high-locality chunks. Within such a chunk, instead of encoding the deltas as before, we encode the lengths of alternating runs of 1-bits and 0-bits in the expected bitmap. These lengths are stored in a bit-setting (S) and a bit-clearing (C) array, respectively.

To compute S and C, we could directly scan through the original chunk data, counting the number of consecutive RIDs and measuring gaps between them. However, since PForDelta has already computed the exception (E) and exception position (P) arrays

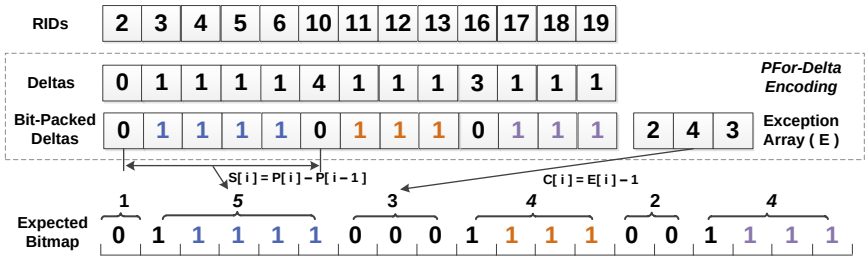


Fig. 2. An example of how PForDelta encodes a chunk of mostly-sequential RIDs, and the bitmap resulting from decoding this chunk. Note how each 1-bit in the expected output bitmap (except the first in each consecutive run) is encoded with a separate delta value. Also shown is the calculation of the S and C arrays used in our more-concise *BitRun* encoding.

for the chunk (with length denoted by n), we can instead compute S and C as follows:

$$\text{for } i = 1,\ldots,|E|: \;\; C_i = E_i - 1,\; S_i = \begin{cases} P_{i+1} - P_i & \text{if } i < |E| \\ n - P_i & \text{otherwise} \end{cases}$$

This computation only requires $|E|$ iterations, rather than n. After this, S and C serve as the encoding output for the chunk (instead of the normal PForDelta chunk output). Since this S and C computation relies on the intermediate state of PForDelta *BitRun* only replaces the last step of PForDelta (delta bit-packing), leaving the other steps identical.

The corresponding decoding algorithm for *BitRun* recovers the bitmap encoded by the S and C arrays. It works by starting at the beginning of the bitmap, then alternately skipping $C[i]$ bits followed by setting the next $S[i]$ bits to 1 for $1 \leq i \leq |E|$. The bit-setting step is accomplished using highly-efficient bitmask OR operations. Additionally, the contiguous 0-bits described by C can merely be skipped, not cleared, because the bitmap is zero-initialized at the start. Since both 0-bit and 1-bit runs are handled in bulk, this decoding process is a substantially faster alternative to standard PForDelta for chunks with $b = 1$.

3.3 *BitExp*: Expanding the PForDelta Encoding Bit-Width

In addition to *BitRun*, which is designed for high chunk data locality, we also develop *BitExp*, a complementary approach for the case when chunk data displays less locality. The key insight of *BitExp* comes from profiling each PForDelta decoding step. By doing so, we are able to improve PForDelta decoding throughput, and thus, to improve the efficiency of the bitmap construction .

We start by looking at the performance of each of the three main steps in PForDelta decoding: 1) *unpack deltas*, which expands the packed deltas to a delta array; 2) *exception patch*, which patches exceptions back to the delta array; and 3) *cumulative sum*, which recovers the original values by summing the deltas. The performance of the

cumulative sum step is relatively stable, as the execution time of sum operation is determined only by the fixed chunk size D. In contrast, the performance of the *unpack deltas* and *exception patch* steps fluctuate. The former depends on the bit-width b, which determines the number of bitwise operations in the step, whereas the latter is affected by the number of exceptions ($|E|$), which determines the number of patch operations.

We next measure the performance sensitivity of the *unpack deltas* and *exception patch* steps as b and $|E|$ vary. We accomplish this by collecting PForDelta decoding performance on compression chunks generated with every combination of b and $|E|$ varying from 2 to 31 and from 0 to 64, respectively. The range of b is chosen because $b = 1$ is already covered by *BitRun* and PForDelta mandates $b < 32$, and the range of $|E|$ covers most realistic encoding cases. We uniformly distribute exceptions in our compression chunks.

The results are shown in Figure 3. We see that the decompression time is almost linear with $|E|$ when b is fixed, whereas the decompression time is quite stable regardless of b while fixing $|E|$. Thus, the number of exceptions plays a dominant role in determining the decoding throughput.

Given this trend, one could maximize decoding throughput by expanding b in every chunk to eliminate all exceptions; however, this would lead to a large storage increase, increasing I/O time and potentially countering any improved decoding throughput. Therefore, *BitExp* opts for another approach: rather than choosing every chunk for b-expansion, *BitExp* selects only those chunks where all exceptions can be eliminated via a small increase in b, leaving other, "harder" chunks untouched. This way, only some chunks use a larger b, balancing decreased compression ratio and increased decoding throughput. Specifically, when encoding each chunk, *BitExp* considers two options: 1) using a larger b to produce an "exceptionless chunk" (which has a larger compressed size, but is faster to decode), or 2) retaining the original b and chunk encoding. The decision is made using a threshold ratio: if the increase in compressed size by using an exceptionless chunk is below this threshold, b is expanded; otherwise, the original b is retained. This b-expansion only affects the encoding process, and does not change the PForDelta format; thus, the usual decoding process is still used, though it now benefits from improved decoding throughput.

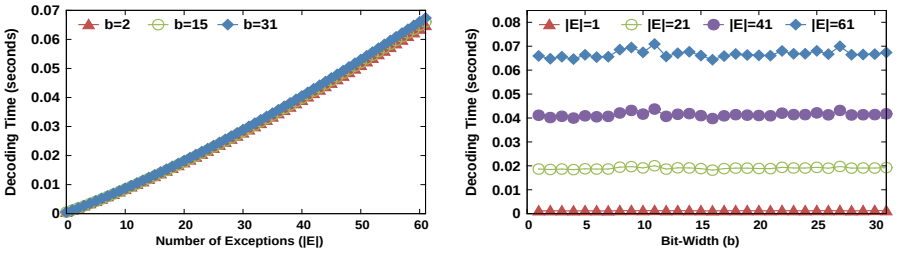


Fig. 3. PForDelta decoding time when varying $|E|$ while keeping b fixed (left), and when varying b while keeping $|E|$ fixed (right). Reported timings are for decoding 1000 PForDelta chunks. Trends seen are representative of experiments with other b and $|E|$ parameters.

3.4 BitRun-BitExp: Handling Set Intersections across Heterogeneous Datasets

Often, queries over scientific data induce set intersections across datasets with differing levels of locality. However, neither *BitRun* nor *BitExp* can effectively tackle this scenario, as they each target a specify level of data locality. Fortunately, these methods are both complementary and have compatible output forms (bitmaps). Therefore, by using *BitRun* for high-locality datasets and *BitExp* for low-locality datasets simultaneously, or even by interleaving *BitRun* and *BitExp* in a single dataset on a PForDelta-chunk basis, we can attain performance beyond what either method can provide alone. We term this approach "BitRun-BitExp (*BRBE*)."

4 Results

4.1 Experimental Setup

Computing Environment: All experiments are executed on the "Sith" cluster at Oak Ridge National Lab. Sith consists of 40 compute nodes, each equipped with four 2.3 GHz 8-core AMD Opteron processors and 64 GB of memory. All data are stored on the Lustre parallel file system using the default striping parameters.

Index Preparation: We use double-precision floating-point datasets from the S3D combustion [17] and FLASH astrophysics [18] simulations. Selected datasets have three locality categories: low, medium, and high. The category indicates the compressibility of dataset. "High" means the dataset is easy to be compressed, whereas, "low" means the dataset is hard to be compressed. All selected datasets have 2GB data size. We construct 5 indexes on each variable: our three index types (*BitRun*, *BitExp*, *BRBE*), PForDelta index, and FastBit's *WAH* compressed bitmap index. The index sizes of these datasets obtained from these 5 indexes methods are shown in Table 1. Additionally, these indexes are all binned using ALACRITY-style [8] significant-bit-based method, which roughly corresponds to binning with between 1 and 2 significant digits in base-10 scientific notation. Since FastBit does not normally support ALACRITY-style binning, in order to ensure a fair comparison, we emulate this binning strategy using a mapping technique to produce an equivalent set of bins in FastBit to match those in our methods. For *BitExp* and *BRBE*, we use an expansion threshold ratio of 1.6, as this value gives reasonable performance based on preliminary experimentation. Finally, *BRBE* is configured to use *BitRun* indexes on S3D datasets marked "medium locality" in Table 1, and *BitExp* indexes for those marked "low locality."

Conjunctive Query Processing and Queries: Our conjunctive query processing (shown in Section 3.1) is built as an extension of the ALACRITY univariate query processing engine. To clearly evaluate our method, all queries used for evaluation are region-retrieval queries, and use constraints that are aligned to index bin boundaries as value-retrieval so that candidate checks are not needed. We use a set of queries in our experiments: 5 overall selectivities ranging from 0.001% to 10% in powers of 10, using 3 or 4 query constraints. We choose these queries as the scientific queries typically have multi-variate constraints and relatively low selectivities [19].

4.2 Comparison with WAH bitmap indexes

We compare our PForDelta-based indexing against *WAH*-based indexing, in two respects: storage size, and query performance as both of these are important aspects of set intersection performance.

The index storage footprints shown in Table 1 demonstrate that the storage requirement for our main method. The index size of *BRBE* is substantially lower, ranging from 15% to 60%, than that of *WAH* for most datasets, except the high locality one. We believe several factors are at play here. First, fewer consecutive runs of bits occur in a *WAH* index for low locality datasets, leading to significant drops in compression due to loss of fill word encoding. In contrast, PForDelta is delta-oriented, and thus its compression degrades more gracefully in the face of slight losses in consecutiveness of RIDs. Furthermore, *BitRun* does not exhibit the same penalty as *WAH* because its run-length encoding mode is only rarely triggered in low locality datasets, minimizing any increase in index size.

Complementarily, Figure 4(a) and 4(b) show the comparison of end-to-end query processing times between a FastBit's *WAH* bitmap index and a PForDelta-compressed index using our method. As FastBit currently does not support performance breakdown, we show this by comparing the total response time, which includes CPU and I/O time.

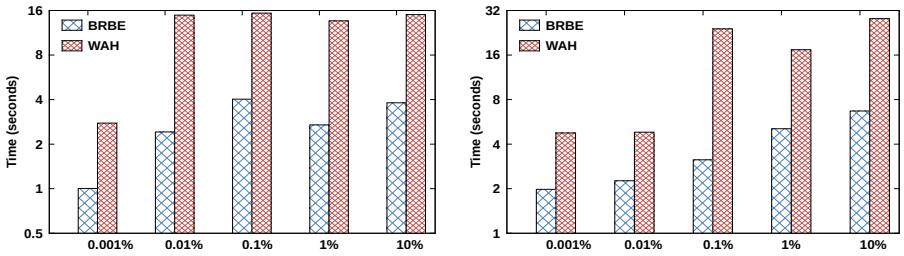
We see that *BRBE* yields query response times that are 2.1x to 7.7x faster than *WAH* on this range of queries. We attribute this trend to *BRBE*'s relatively higher index compression ratio, which leads to less I/O than that induced by *WAH*. A "flattening-off" effect is also apparent in *WAH*'s times for lower selectivities. One possible explanation is the query optimization in *WAH* (which is also common in database management systems): when selectivity on some query constraint is very low, a sequential scan is used instead of the index (for that constraint only), thus avoiding the cost of processing a large portion of index. With *BRBE*'s smaller index, however, this strategy does not appear to be necessary.

4.3 Performance Breakdown of PForDelta-compressed Index Approaches

We breakdown the end-to-end query processing time to gain further insight of our methods. In addition to evaluating our three methods (*BitRun*, *BitExp*, and *BRBE*), we also include methods "*RawIndex*" and "*Simple*" as comparison baselines to demonstrate the benefit of our refinements. The *RawIndex* method uses a simple, uncompressed

Table 1. Storage footprints of different indexing methods on several scientific datasets.

Dataset	**Data Locality**	**Index Size (as % of Original Data Size)**				
		PForDelta	***BitRun***	***BitExp***	***BRBE***	***WAH***
FLASH_gamc	High	2.5%	2.0%	2.5%	2.0%	0.3%
FLASH_vely	Medium	4.4%	5.4%	4.4%	5.7%	6.8%
S3D_temp	Medium	5.3%	6.7%	5.9%	7.0%	8.8%
S3D_uvel	Medium	6.6%	10.6%	7.4%	11.0%	15.1%
S3D_vvel	Low	15.1%	19.2%	17.7%	21.1%	55.0%
S3D_wvel	Low	15.1%	18.9%	17.6%	20.8%	55.0%



(a) Queries with 3 variable constraints (b) Queries with 4 variable constraints

Fig. 4. Comparison of query response times between *BRBE* and *WAH*. Results for two-constraint queries are similar, but are omitted for space considerations.

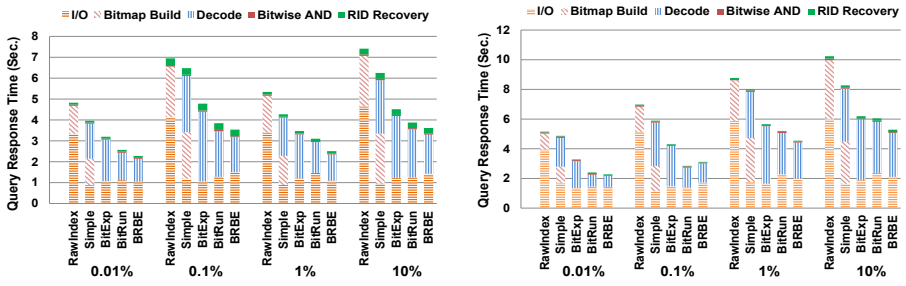
inverted index, and build an uncompressed bitmap directly from this representation before performing set intersection. The *Simple* method instead stores its inverted indexes in standard PForDelta-compressed form, and fully decompresses retrieved bins before building the uncompressed bitmap in the same manner as *RawIndex*. We show results for *RawIndex* to illustrate the need for a storage-lightweight approach, and *Simple* to demonstrate the need for an efficient PForDeltato-bitmap conversion process.

The performance breakdown includes each component in the query evaluation process: "I/O," "Decode," "Bitmap Build," "Bitwise AND," and "RID Recovery." "I/O" measures the time to read the compressed or uncompressed index bins from storage. We would like to point out there is a relatively stable "I/O" performance throughout our experiments as we observe the average standard deviation and variance of "I/O" are 0.1024 and 0.0133, respectively. "Decode" indicates the time to decompress the compressed index bins (this phase is not present for *RawIndex*). "Bitmap Build" shows the time to build the uncompressed bitmap from these bins (this phase is not present for *BitRun*, *BitExp* or *BRBE*, as they do this in situ during the "Decode" step). Finally, "Bitwise AND" is the time to intersect the bitmaps from each constraint into the final output bitmap, and "RID Recovery" measures the last step of extracting RIDs from this result bitmap (both of these steps are shared by all five methods).

Figures 5(a) and 5(b) show the performance breakdown of each component in the query evaluation. Performance of the *RawIndex* and *Simple* approaches are predictable: *RawIndex*'s performance is dominated by long I/O times due to its lack of compression, whereas *Simple* suffers from having separate decompression and bitmap construction steps, demonstrating the need for a more efficient bitmap conversion process.

We see that *BitRun*, *BitExp*, and *BRBE* are consistently faster than *RawIndex* and *Simple* in almost all cases. This is because decoding time is greatly reduced for our methods, more than offsetting their slightly higher I/O cost. This decoding speedup is partly because all three of our methods employ a cache-efficient, in-place bitmap construction technique. Additionally, both *BitRun* and *BitExp* add their own optimizations to the decoding process (as described in Sections 3.3 and 3.2).

Between *BitRun* and *BitExp*, we see *BitRun* demonstrates the larger performance gain, presumably because it exploits the more "lucrative" possibility of setting runs of



(a) Queries with 3 variable constraints (b) Queries with 4 variable constraints

Fig. 5. Comparison of query response times among *BitRun*, *BitExp*, and *BRBE*, as well as two baseline index methods.

bits in bulk. However, *BRBE* generally performs better than when either *BitRun* or *BitExp* is applied alone. This is because *BRBE* combines the benefits of both its component methods, and thus can handle a mix of higher-locality and lower-locality dataset variables, whereas *BitRun* or *BitExp* alone can only effectively handle one of these cases.

5 Conclusion and Future Work

Set intersection is a critical operation for conjunctive query processing in scientific data analysis. In this work, we present a fast set intersection approach based on coupling the storage-lightweight PForDelta indexing format with computationally-efficient bitmaps via an on-the-fly conversion process. We address the key challenge of minimizing the bitmap conversion cost through enhancements to PForDelta, which drastically improve bitmap construction time. Results indicate our method achieve speedups of between 2.1x and 7.7x versus state-of-the-art method while only requiring 15-60% less storage space.

In the future, we plan to extend this work in two directions: 1) we will evaluate the end-to-end impact of our method in real scientific applications by integrating our work in two widely used I/O middlewares: HDF5 and ADIOS; 2) we will consider parallelizing our method in the GPU environment, in which the massive number of parallel threads helps to decompress tremendous PForDelta chunks simultaneously.

6 Acknowledgment

We would like to thank the Leadership Computing Facilities at Argonne National Laboratory and Oak Ridge National Laboratory for the use of resources. Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under Contract DE-AC05-00OR22725. This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs).

References

1. Demaine, E., López-Ortiz, A., Munro, J.: Adaptive set intersections, unions, and differences. In: Proc. Symposium on Discrete Algorithms (SODA). (2000)
2. Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Transactions on Information Systems (1996)
3. Byna, S., Wehner, M., Wu, K., et al.: Detecting atmospheric rivers in large climate datasets. In: Proc. Workshop on Petascal Data Analytics: Challenges and Opportunities. (2011)
4. Wu, K., Otoo, E., Shoshani, A.: Compressing bitmap indexes for faster search operations. In: Proc. Scientific and Statistical Database Management (SSDM). (2002)
5. Wu, K., Otoo, E., Shoshani, A.: On the performance of bitmap indices for high cardinality attributes. In: Proc. Very Large Data Bases (VLDB). Volume 30. (2004)
6. Wu, K.: FastBit: an efficient indexing technology for accelerating data-intensive science. Journal of Physics: Conference Series (2005)
7. Jenkins, J., Arkatkar, I., Lakshminarasimhan, S., et al.: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In: Proc. Database and Expert Systems Applications (DEXA). (2012)
8. Jenkins, J., Arkatkar, I., Lakshminarasimhan, S., et al.: Alacrity: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In: Transactions on Large-Scale Data-and Knowledge-Centered Systems X. (2013)
9. Lakshminarasimhan, S., Boyuka II, D., et al.: Scalable in situ scientific data encoding for analytical query processing. In: Proc. High-performance Parallel and Distributed Computing (HPDC). HPDC '13 (2013)
10. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar RAM-CPU cache compression. In: Proc. International Conference on Data Engineering (ICDE). (2006)
11. Zhang, J., Long, X., Suel, T.: Performance of compressed inverted list caching in search engines. In: Proc. World Wide Web (WWW). (2008)
12. Yan, H., Ding, S., Suel, T.: Inverted index compression and query processing with optimized document ordering. In: Proc. World Wide Web (WWW). (2009)
13. Barbay, J., López-Ortiz, A., Lu, T.: Faster adaptive set intersections for text searching. In: Experimental Algorithms. Volume 4007 of Lecture Notes in Computer Science. (2006)
14. Baeza-Yates, R.: A fast set intersection algorithm for sorted sequences. In: Combinatorial Pattern Matching. Volume 3109 of Lecture Notes in Computer Science. (2004)
15. Chatchaval, J., Boonjing, V., Chanvarasuth, P.: A skipping SvS intersection algorithm. In: Proc. International Conference on Computing, Engineering and Information (ICC). (2009)
16. Jonassen, S., Bratsberg, S.: Efficient compressed inverted index skipping for disjunctive text-queries. In: Advances in Information Retrieval. Volume 6611 of Lecture Notes in Computer Science. (2011)
17. Chen, J. and Choudhary, A. and Supinski, B. and et al.: Terascale direct numerical simulations of turbulent combustion using S3D. Computational Science & Discovery (2009)
18. B. Fryxell, K. Olson, P. Ricker and et al.: FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. The Astrophysical Journal Supplement Series (2000)
19. Sinha, R.R., Winslett, M.: Multi-resolution bitmap indexes for scientific data. ACM Transactions on Database Systems (TODS) (2007)