School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

An open experiment in SIMD-based set intersection

by

Alex Brown

Thesis submitted as a requirement for the degree of

Bachelor of Engineering (Software)

Submitted: Nov 2023

Supervisor: A/Lect. Zhengyi Yang Student ID: z5169792

Abstract

Set intersection is a fundamental computing operation employed widely in search engines, relational databases and graph processing. These stakeholders present various characteristics of data which create a vast space of algorithms adapting to each input type. Furthermore, as hardware evolves, we see algorithms exploiting architectural features such as branch prediction, SIMD instructions and mask registers. These algorithms have not been comprehensively compared, in part due to closed-source implementations and experiments. In light of these issues, we present an open-source benchmark suite enabling reproducible, verifiable and extensible experiments. We employ this suite to perform an updated experimental analysis, comparing AVX-512 extensions against existing algorithms. We present a new method of SIMD intersection which utilises *broadcast* instructions, and find that this method scales better to higher vector widths. The variance in our results highlights the importance of benchmarking these algorithms with their target hardware and input, further motivating open-source experiments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Set intersection is a deceptively simple operation. We take two sets of elements, usually integers, and find the set of common elements as fast as possible. Databases, graphs and search engines rely on these operations to execute queries at rates scalable to millions of users. As a core component of these queries, the performance of an intersection algorithm has a direct impact on both user experience and operational efficiency.

The complexity of this space comes first with an almost unlimited variability in the underlying dataset. Several characteristics of input each drastically affect the performance of a particular algorithm. What's more, each use case presents its own unique datasets which may or may not conform to these units of measure. This forces us to consider data in both a bottom-up manner, with parameters synthetically defined – and a top-down manner, verifying that results carry to real input.

The complexity comes next through the diversity of algorithms and data structures. With related research dating back to the 70's [BY75], we observe shifts in both the underlying technology and the metrics to which algorithms are evaluated. This compounds with assorted use cases to give rise to dozens of algorithms each characteristic of their target input and the technology available at the time.

Not only does this complexity elicit several gaps in comparison, it severely limits the

development and adoption of new algorithms. Current benchmarks are not comprehensive, and in some case lack transparency through closed-source implementations. Researchers cannot easily compare their algorithms with existing implementations – stalling new development. Given the sensitivity these algorithms have to their hardware and input, it is crucial adopters are able to run experiments within their own environment. In the case of closed-source experiments, researchers and adopters are unable to verify the correctness of implementations and benchmarks. Many SIMD algorithms are open to extensions to higher vector widths, but these are yet to be included in an experiment.

As a remedy to these issues, we present an open-source benchmark suite with high coverage of SIMD algorithms. This includes a suite of tools to declaratively outline experiments, generate datasets, run benchmarks, and plot results using simple commands. We also include both a test suite and a program to verify datasets adhere to their predefined characteristics.

We then perform an updated experimental analysis using this suite, with particular focus on extensions to SIMD algorithms, summarised as follows.

1. To find the optimal implementation of each algorithm, we investigate the impact of branching, and find that the optimal configuration depends on both the algorithm and underlying hardware.
2. We present a simple SIMD algorithm which uses *broadcast* instructions to perform an all-pairs comparison, and observe it scales better to higher vector widths.
3. Benchmarking the AVX-512 version of *Roaring* shows it now outperforms BSR algorithms. It also outperforms the array-based algorithms for densities as low as 1%.
4. We reimplement FESIA, and find our implementation is likely suboptimal, leading to questions around the efficacy of the closed-source algorithm.
5. Finally, benchmarking on the inverted index dataset *WebDocs* reveals *Roaring* performs strongly for low set counts, and *Small Adaptive* scales well to higher set counts. This motivates additional research into adaptive algorithms.

In Chapter 2 we begin by providing context into use cases, characteristics of data and technologies employed by algorithms. We then review existing works outlining several gaps in comparison in Chapter 3. In Chapter 4 we present our open-source benchmark suite discussing various design decisions and implementation details. In Chapter 5 we employ this suite to perform an updated experimental analysis on set intersection algorithms. Finally, in Chapter 6 we draw conclusions and discuss avenues for further research.

Chapter 2

Background

2.1 Set Intersection

An intersection is defined as the set of common elements between two or more input sets. To begin to categorise these inputs, their origin must be understood. Given its fundamental nature, set intersection may be used anywhere, but its primary stakeholders are search engines, relational databases and graph algorithms. These applications dictate input *top-down*, where the performance of an algorithm is unique to this type of input. To form a complete picture into relative performance of these algorithms, a *bottom-up* approach must be taken, where data is generated synthetically based on well-defined parameters. These parameters are then varied to form a model for why certain algorithms perform well for certain real-world inputs. Due to the number of variables, however, the process of comparison is not simple. For instance, with the set of parameters used in this thesis, a four-dimensional analysis would be required to form a complete picture of how these algorithms compare. As a result, a balance must be struck between a bottom-up approach which carefully selects parameters as they appear in real data, and a top-down approach verifying these findings carry to real-world input.

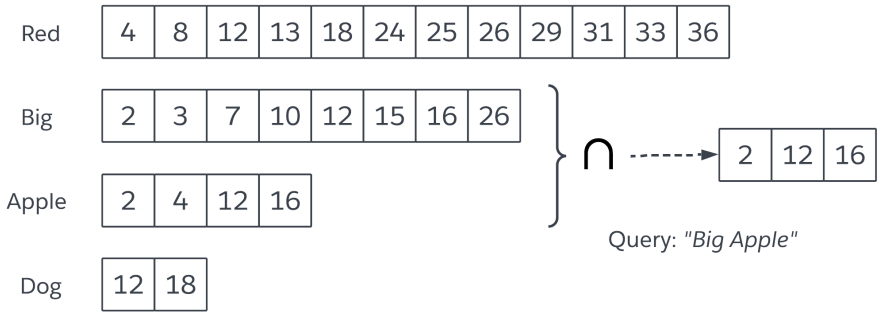


Figure 2.1: Inverted index query example

2.1.1 Use Cases

As discussed, the set intersection space is first coloured by its various stakeholders. The first of these is *search engines*, sometimes referred to as *text databases*. The fundamental operation of a search engine is to input a list of words and output the set of documents which contain these words, ranked based on some heuristic. An inverted index is a key component of this translation, mapping each word to a set of document IDs which contain such word. Given a query, to find the set of documents which contain all words in the query, an intersection is performed on the sets which correspond to each word in the inverted index, as depicted in Figure 2.1. Since these indexes may span millions of documents, a high rate of intersection is critical to ensure liveliness.

We next see set intersection used heavily in relational databases. Conceptually, whenever a conjunctive query is executed (i.e., a query with an `AND` operator) an intersection is performed between the set of records for which the left operand is true, and the set of records for which the right operand is true. In practice, the underlying operation depends on the implementation of the DBMS or whether indexes are being used to optimise the query. For example, take a table `Person` with columns `Gender` and `IsStudent`. These columns are both of low cardinality, so a bitmap index could be constructed over each of these columns to optimise the query performance for queries involving these terms. To find the set of people who are both *male* and *students*, an intersection is performed through a bitwise `AND` between the index `IsMale` and `IsStudent`.

Graph databases are set intersection's last major stakeholder though applications such as data mining, recommendation systems and fraud detection. Since these graphs are typically low density, their edges are represented through an adjacency list to maximise space efficiency. These adjacency lists may be stored in any form, but conceptually represent a set of nodes which are adjacent to the current node. To find the set of common neighbours between two nodes, an intersection is performed between each of these nodes' adjacency lists. Through this property, set intersection is a significant component in many common graph algorithms such as triangle counting, clique detection and subgraph matching [HZY18].

2.1.2 Characteristics of Input

Each of these use cases present their own set of input characteristics. The primary characteristics this thesis considers are *selectivity*, *density*, *skew*, and whether the operation is *2-set* or *k-set*. These characteristics have a significant impact on the relative performance of various data structure and algorithms.

Selectivity. For a particular intersection, selectivity is defined as the ratio of the output cardinality to the cardinality of the smallest input set. Existing experiments have found that, in practice, the selectivity of an intersection is generally low (less than 30%) [DK11, LBK14, IOT14].

Density. A set's density is defined as the ratio between its cardinality and the cardinality of the set of all possible elements. In the case of inverted indexes, the density is generally low as an arbitrary word is likely to appear in only a small subset of documents. In a relational databases and graphs, the density depends heavily on the underlying data. Density is generally only considered when deciding whether to employ a bitmap data structure, where a high set density improves the parallelism of each bitwise `AND` operation.

Skew. Skew is defined as the ratio of cardinalities between two sets. Sets which are heavily skewed will vary significantly in cardinality. In the case of an inverted index, since the cardinality of an entry is equal to that entry's frequency, due to Zipf's Law [Zip36], these cardinalities will be inversely proportional to their rank. This means that if two sets are chosen at random, these sets are likely to differ in size, and are hence commonly *skewed*. In contrast, the level of skew in a relational or graph database depends on the particular dataset and the operation being performed.

2-set vs. k-set. An intersection can be further characterised by the number of input sets. Since a search query may have multiple terms, inverted index intersection is generally k-set (i.e., more than two sets). On the contrary, most graph algorithms tend to only intersect two sets at a time (i.e., 2-set) [HZY18]. This distinction is relevant as many adaptive algorithms take advantage of this property by intersecting multiple sets simultaneously. These adaptive algorithms are covered in Section 3.2.3.

2.2 SIMD

Set intersection has been researched for multiple decades, and in this time various shifts in hardware have complicated the space. In the early 2010s, set intersection saw a wide adopted a class of CPU instructions known as SIMD instructions. SIMD stands for Single Instruction Multiple Data and refers to a general class of instructions which are able to perform a single operation between two or more *vectors* of elements in parallel. These instructions increase the parallelism of algorithms by allowing multiple elements to be operated on in a single instruction.

A SIMD algorithms's *vector width* characterises its level of parallelism. Intel's Streaming SIMD Extensions (SSE), the first widely adopted SIMD extension for x86, supports up to 128-bit vectors and was released in 1999 [KP99]. This was increased to 256-bit registers in Intel and AMD's Advanced Vector Extensions (AVX), first adopted by Sandy Bridge CPUs in 2011 [Kan11]. The most recent increase to this vector width was with Intel's AVX-512 extension, proposed by Intel in 2013 [Rei13]. An increased

vector width naturally improves the parallelism of an algorithm, potentially resulting in faster intersection. In this experiment, for simplicity, we only consider x86 SIMD extensions as these are most common.

Although less relevant to research, it is important to acknowledge the negative impact SIMD instructions have on compatibility. SIMD instruction sets and tooling is still maturing and not all instructions in one architecture have an equivalent in another architecture. Portable SIMD libraries such as Google's `Highway`[1] and Rust's `portable-simd`[2] do their best to provide a unified interface however there is no guarantee all high-level primitives will translate to a single instruction in all architectures. Even within a specific architecture, not all CPUs will support all instructions, so care must be taken distributing binaries. These issues generally see newer SIMD instructions like AVX-512 used primarily in enterprise environments where hardware is known and throughput is critical. This makes SIMD a good fit for the commercial use cases described in Section 2.1.1.

[1] `https://github.com/google/highway` [2] `https://github.com/rust-lang/portable-simd`

Chapter 3

State of Research

With an understanding of the data and hardware landscape of set intersection, we may now begin to define the scope of this thesis. We first restrict our research to algorithms which operate on a CPU and do not consider GPU-based algorithms or approximate methods such as bloom filters. From this subset, existing algorithms are categorised as follows (see Figure 3.1). We begin by separating the *scalar* algorithms from the *vector* (i.e., SIMD) algorithms, ignoring the underlying data structure. This split is loosely correlated with the date these algorithms were published as most scalar algorithms precede most vector algorithms.

The space is then partitioned by data structure. Primarily motivated by inverted indexes, a significant amount of research covers algorithms which operate on a *sorted array*. This format is often employed as it a simple method to store low-density sets with support for fast searching. Within the category of *"scalar algorithms which operate on a sorted array"* there exists a number of *classical* algorithms and *adaptive* algorithms which extend these to adapt to real-world data. Stepping back, many *"alternative data structures"* have also been developed to target higher input density and/or higher intersection speed. This category has been left broad as most data structures are unique to their target algorithm.

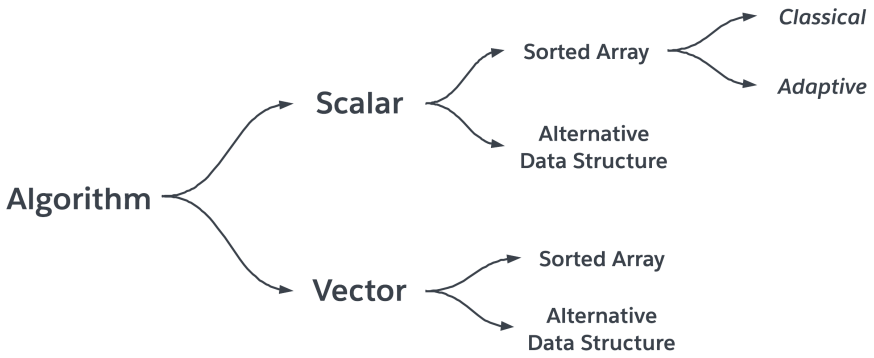


Figure 3.1: Algorithm categories

3.1 Data Structures

As discussed in Section 2.1.1, search engines usually store document IDs as a sorted array of integer values. Due to the massive scale of text databases, several compression methods have been developed for these sorted integer arrays, neatly summarised in [WLPS17]. Although independent operations, intersection may be interleaved with decompression to avoid decompressing the whole array, potentially impacting intersection time [WLPS17]. For simplicity, since intersection algorithms are typically developed agnostic to compression algorithms, this proposal does not cover these structures.

The high-density analogue for a sorted array of integers is a bitmap. These structures aim to offer increased intersection parallelism through bitwise `AND` operations, providing greater space efficiency at high density. As the density reduces, the bitmap will be filled with an increasing number of 0's which redundantly indicate an element is not contained in the set. To mitigate this issue, many compressed bitmap representations have been developed each presenting different intersection times. Since the intersection operation is usually performed directly on the compressed bitmaps, this may have a large impact on intersection time, and hence, our experiment considers these structures.

In addition to sorted arrays and bitmaps, various alternative data structures have been developed specifically aimed at improving intersection time. Some examples include treaps [BRM98], skip-lists [ST07, MRS08], buckets [ST07], hashing [BPP07, DK11] and bitmaps [DK11]. These works are covered in Section 3.2.4. Another space of

research which extends the above works is the *fast set intersection* problem, proposed by Cohen and Porat in 2010 [CP10]. In this problem, an entire database worth of pre-determined sets are processed offline in order to improve the speed of "online" intersections. This is particularly applicable to the field of text databases where inverted indexes are recomputed relatively infrequently. Many algorithms have been introduced in this space [CP10, CS16, KW20], however since they solve a fundamentally different problem to other algorithms discussed, they are not considered in this experiment.

3.2 Scalar Algorithms

3.2.1 Merge-based Intersection

The merge method is a classical set intersection algorithm which operates on two sorted arrays of integers. Similar to the merge stage of *merge-sort*, the algorithm steps through both arrays in concurrently, but instead of outputting every element, it only outputs elements contained in both sets. The comparisons made are illustrated in Figure 3.2. An enhanced version of the merge algorithm has been presented by multiple authors [SWL11, IOT14], improving branch prediction by replacing the hard-to-predict "less than" branch with simple pointer arithmetic. Due to its ubiquity and simplicity, the merge algorithm *NaiveMerge* or its "branchless" cousin *BranchlessMerge* are often used as a baseline in experiments.

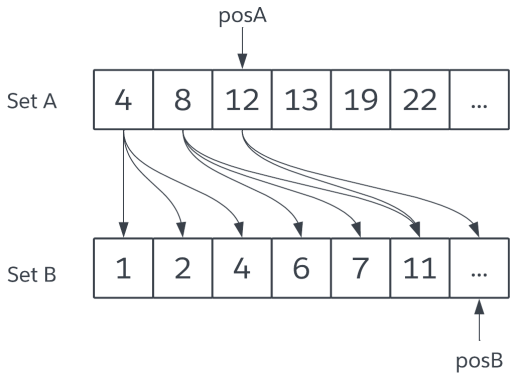


Figure 3.2: Comparisons in the merge algorithm

3.2.2 Search-based Intersection

It was soon observed that merge-based intersection performs many redundant comparisons if the input sets are not perfectly interleaved [DLOM00]. This is further exacerbated when the two sets differ significantly in size. In this case, the intersection can be improved by performing a search for each element in the smaller set to test if it is in the larger set. The most widely used search algorithm for this task is the Galloping search algorithm, presented by Bentley and Yao in 1975 [BY75]. This algorithm begins the search at the start of the array, then performs doubling probes until it finds an element greater than the target. Then, it performs a standard binary search between the two most recent probes to determine whether the element is within the set, illustrated in Figure 3.3. There are many alternatives to Galloping such as Golomb search [HL72], interpolation and extrapolation [BLOL06] which are explored in Section 3.2.3.

3.2.3 Adaptive Algorithms

As presented, the above algorithms do not define a method to intersect more than 2 sets at once. Search engine queries often necessitate the intersection of more than 2 sets at a time (see Section 2.1.2). The simplest way to extend the above algorithms to k sets is to perform $k - 1$ pairwise intersections. This section explores *adaptive* set intersection algorithms which aim to improve upon this method, optimising for real-life, non-uniform datasets.

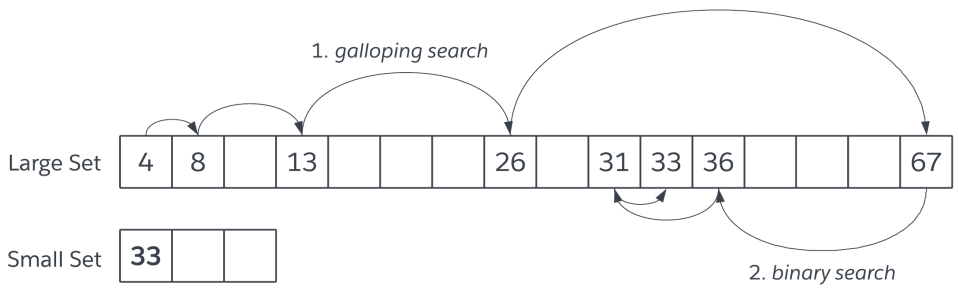


Figure 3.3: Galloping search

Adaptive. The first adaptive algorithm, aptly named *Adaptive*, was introduced by Demaine et al. in 2000 [DLOM00]. The main idea of this algorithm is to select the first element of the current set as the *eliminator*, then to cycle through the sets, performing a single Galloping search step on each. If the gallop overshoots, the standard binary search is performed to check if the eliminator exists in the current set. If it does, the occurrence counter is incremented, and the element is output if this counter reaches k. Otherwise, the eliminator is updated to the first element of the set with the fewest remaining elements and the process repeats.

Small Adaptive and SvS. Demaine extended this work by presenting two more adaptive algorithms and performing an experimental analysis in a paper published in 2001 [DLOIM01]. The first algorithm, denoted *Small vs. Small* or *SvS*, repeatedly intersects the two smallest sets until the intersection is complete. Although the 2-set intersection algorithm used is interchangeable, the paper recommends Galloping search as it outperforms binary search. The second algorithm introduced, *Small Adaptive*, aims to provide a middle ground between *Adaptive* and *SvS*. First, the smallest element of the smallest set is searched for in the second-smallest set. If it exists, the element is then searched for in the third-smallest set, and so on, until it is determined that the element exists in all sets. For the inverted index dataset used in the experiment it was found that *Small Adaptive* outperforms both *Adaptive* and *SvS*. It was found that *Adaptive* outperforms *SvS* for 3 or fewer sets, and *SvS* outperforms *Adaptive* for intersections of more than 3 sets.

Sequential and RSequential. *Sequential* is an adaptive algorithm introduced by Barbay and Kenyon in 2002 [BK02]. This algorithm behaves almost identically to *Small Adaptive*, but instead of picking the eliminator from the set with the fewest remaining elements, it is chosen from the current set in the cyclic order. Barbay also published a randomised variant of this algorithm in 2003 [Bar03], denoted *RSequential*, which picks the next eliminator from a random candidate as opposed to the next candidate in cyclic order. Although this theoretically reduces the number of comparisons, a later study by Barbay et al. found that both *Sequential* and *RSequential* are uncompeti-

tive compared to other adaptive algorithms due to the higher number of comparisons performed [BLOLS10].

Recursive intersection. Baeza-Yates and Salinger took a unique approach in their adaptive algorithm published in 2004 [BY04]. Denoted *BaezaYates*, their algorithm first retrieves the median of the smaller set, then searches for it in the larger set, adding it to the result if found. It then partitions these sets based on the location of these elements, then recursively computes the intersection of the left and right partitions. This is illustrated in Figure 3.4. The original paper only concluded that *BaezaYates* was faster than *Adaptive* and *NaiveMerge*, but a subsequent study [BLOLS10] found that for certain datasets, *BaezaYates* may outperform *Small Adaptive* and *SvS*, among others.

Interpolation and extrapolation. Barbay et al. extended these works by proposing the use of interpolation and extrapolation search techniques in the galloping phase of existing algorithms [BLOL06]. The most successful variant from the paper, *ExtrapolAheadSmallAdaptive*, replaces the doubling gallop with an extrapolation based on the current element and an element l steps ahead. The value for offset l may vary, however the paper found that $l = \log n$ yields the strongest results. An example is illustrated in Figure 3.5. In a later study by Barbay et al., this algorithm was found to perform the best in terms of number of comparisons performed, but it did not perform as well as other algorithms in terms of CPU time due to its worse cache performance.

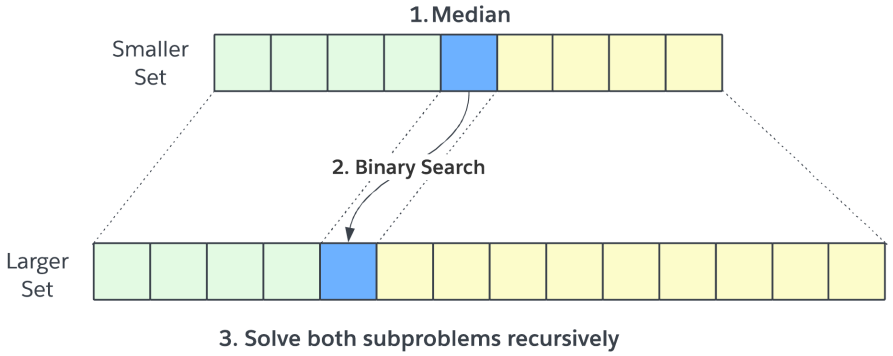


Figure 3.4: *BaezaYates* algorithm

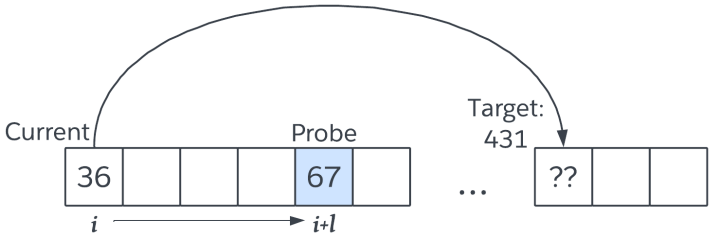


Figure 3.5: *ExtrapolateAhead* example

Max. The most recent algorithm in this category is another *Small Adaptive* variant proposed by Culpepper and Moffat in 2010, called *Max Successor Intersection* or *Max*. The premise of this algorithm is to improve the cache efficiency of *Small Adaptive* by selecting the new eliminator as "the larger of the mismatched value and the successor from the smallest set." On the dataset employed, they found that their new algorithm performed similarly to *BaezaYates* and *SvS* which respectively outperform *Adaptive* and *Sequential*.

3.2.4 Alternative Data Structures

Due to the pre-computed nature of inverted indexes, there is no mandate for systems to store the sets as sorted arrays. This creates the opportunity for the use of specialised data structures to achieve faster intersection over what is possible with array-based algorithms. As a result, many alternative formats were developed in parallel to the above algorithms, with early examples including treaps [BRM98] and skip-lists [ST07, MRS08]. In 2007, Sanders and Transier proposed an algorithm called *Lookup*, where items are separated into buckets based on their most significant bits. Also in 2007, Bille et al. proposed a similar structure where elements are hashed before being placed into buckets (denoted *BPP*) [BPP07]. These algorithms were succeeded by *RanGroupScan*, presented by Ding and König in 2011 [DK11]. This method first hashes the set elements, then places them in a bitmap for fast intersection. A reverse mapping is stored to recover the original set elements from their hash.

The experiment by Ding et al. [DK11] compares *RanGroupScan* with *NaiveMerge*,

SkipList, *Hash* (standard hash table), *BPP*, *Lookup* and various adaptive algorithms. From this experiment, three algorithms stand out.

1. ***RanGroupScan***. For a 2-set intersection of similarly sized sets, *RanGroupScan* outperformed both *Lookup* and *BPP*. When varying skew, it was found that *RanGroupScan* performed the best for size ratios in the range $[1, 32] : 1$. For both k-set intersection and the real dataset employed in the experiment, *RanGroupScan* was optimal.

2. ***Lookup***. *Lookup* outperformed other algorithms for ratios of $(32, 100) : 1$.

3. ***Hash***. For size ratios greater than $100 : 1$, *Hash* performed the best, followed by *Lookup* and *HashBin*.[1]

3.3 Vector Algorithms

SIMD Hierarchical. The surge in SIMD adoption in commodity processors through the 2000s gave rise to a new class of algorithms specialised for these parallel instructions. The first such published was a paper by Schlegel, Willhalm and Lehner in 2011 [SWL11], aiming to harness the STTNI instructions of SSE 4.2. STTNI stands for *string and text processing new instructions*, and allows complete comparison of eight 16-bit values (or sixteen 8-bit values) in parallel. This is highly applicable to set intersection algorithms, particularly *NaiveMerge*, as it compresses 64 comparisons of 16-bit set elements into a single CPU instruction. This approach is limited as set elements are often larger than 16 bits, so a hierarchical representation is proposed to extend the method to 32 bits. The success of the 32-bit variant is tied to the number of elements that share the same upper 16 bits – when low, the algorithm performs up to twice as fast as *BranchlessMerge*, but as the domain increases, performance degrades to the point where it performs worse than *NaiveMerge*. Another limitation of this algorithm is that

[1] *HashBin* is another structure proposed in [DK11] aimed at fast skewed intersection.

the 32-bit variant, denoted *Hierarchical*, requires a modified data structure and cannot operate on raw sorted arrays.

SIMD Shuffling. In his 2012 blog post [Kat12], Ilya Katsov presents an algorithm which utilises a shuffling technique to parallelise the comparisons without resorting to STTNI instructions (denoted *Shuffling*). In general, based on the following studies, this algorithm performs very strongly, however *Hierarchical* outperforms it for some datasets where the domain is small.

1. Katsov's blog presents an experiment with synthetic, uniformly distributed sets of 1M elements in the range [0,3M] (i.e., 33% density) and 30% selectivity. *Hierarchical* is shown to perform almost twice as fast as *Shuffling*, which in turn performs roughly 2.7 times faster than *NaiveMerge*. *Hierarchical*'s strong performance can likely be attributed to the relatively small domain of [0,3M].

2. A later study by Lemire et al. [LBK14] found that *Shuffling* only performs well for arrays of similar size. This is expected as *Shuffling* walks through the arrays similar to the classical merge algorithm, which is also not optimal in this case.

3. In [IOT14], *Shuffling* outperforms *Hierarchical* with synthetic, uniformly distributed sets of 10M elements at any selectivity. This is discrepancy is likely due to the increased domain size used in this experiment – with a density of 1%, it can be assumed that the domain is approximately [0,1B] which is much larger than [0,3M] as used by Katsov. *Shuffling* also outperforms many more recent algorithms (covered later), such as *BMiss* when selectivity is greater than 10%, and *QFilter* when selectivity is greater than 40%. We also see that *Shuffling* does not perform as well for skewed sets, supporting the claim made in (2).

V1, V3 and SIMDGalloping. As alluded to above, both *Hierarchical* and *Shuffling* do not handle skewed sets optimally. As a remedy, Lemire, Boytsov and Kurz proposed algorithms *V1*, *V3* and *SIMD Galloping* in their 2014 paper [LBK14]. *V1* can be summarised as a modified *NaiveMerge* where elements in larger set are compared in

parallel to a single element in the smaller set. *V3* adds another layer of branching to the parallel comparison to target higher levels of skew. *SIMD Galloping* extends classical galloping intersection by performing gallops in units of the vector width, then utilising a parallel comparison at the final stage of the binary search. It is concluded that an optimal combination of these algorithms is to employ *V1* for skew ratios up to $50:1$, then *V3* for skews up to $1024:1$, otherwise *SIMD Galloping*. It is clear from their experiments that *Galloping* and *SIMD Galloping* are optimal for skews greater than $30:1$. *SIMD Galloping* is shown to be faster than *Galloping*, however this gap converges as the skew tends to infinity.

BMiss. Inoue, Ohara and Taura improve on existing works by optimising for *branch prediction* in their 2014 algorithm denoted *BMiss* [IOT14]. Modern CPUs are heavily pipelined, with successive multi-clock-cycle instructions being staggered to increase parallelism. In order to pipeline branches, the CPU's *branch predictor* must assume that one particular branch will execute before the CPU has evaluated the branch condition. A common approach is to predict the outcome based on previous outcomes, however if the branch is *mispredicted* the pipeline needs to restart wasting many clock cycles. This has led various algorithms such as the classical merge algorithm to be reconfigured in such a way that either removes branches or makes them easier to predict.

Inoue et al. first present *BranchlessMerge* which replaces the hard-to-predict *if-greater* branches with simple pointer arithmetic. They present an alternative approach, denoted *BMissScalar*, which compares elements in blocks of size S with S^2 equality tests (as opposed to S inequality tests). Although this increases the number of comparisons, since the equality operator is significantly easier to predict compared to inequality operators (e.g., `>`), this leads to speedups even greater than *BranchlessMerge*. In their SIMD variant, they utilise STTNI instructions to compare only part of each element, filtering out the expensive scalar comparisons to the rarer case where elements share a common bit prefix. The following claims are all supported by their experimental results which compare variations of their proposed algorithms against *NaiveMerge*, *Galloping* and *V1*.

1. Since *BMiss* and *BranchlessMerge* are derived from *NaiveMerge*, they are suboptimal for highly skewed sets when compared with galloping.

2. *BMiss*'s best case is when the selectivity is 0, as more elements are likely to be filtered, avoiding expensive comparisons. In this case, it outperforms *V1* and merge-based intersection by a significant margin. As the selectivity increases, there is a point where *BMissScalar* overtakes *BMiss* in performance, however this point depends on the block size S used by each algorithm.

Roaring. So far, most algorithms discussed operate on a sorted array or a similar low-density derivative. The next method of note, *Roaring Bitmaps* [CLKG16, LKK$^+$18], utilises a hybrid data structure to support fast set operations. Sets are stored as an array of containers, where each container may be a sorted array of elements (for densities $\leq 6.25\%$) or a bitmap (for densities $> 6.25\%$). This data structure is relevant as it provides competitive SIMD-based set intersection when compared with existing bitmap and inverted index compression methods (both SIMD and non-SIMD) [WLPS17].

QFilter and BSR. Another not-yet-covered topic is the graph database use case: aforementioned algorithms all either target generic set intersection, inverted index applications or relational database applications. Han, Zou and Yu build on existing research by proposing a new bitmap-based set intersection data structure and algorithm targeted at graph applications [HZY18]. Their data structure *BSR* (Base and State Representation) improves the intersection speed of traditional bitmap-based set intersection by removing binary words which contain only 0's. *QFilter* is a SIMD-based set intersection algorithm designed to operate on *BSR* sets, however it can also be modified to work on sorted arrays. For simplicity, these variants are labelled *QFilter* and *QFilterBSR*. Han et al. provide an experimental analysis comparing their algorithms to *NaiveMerge*, *Galloping*, *Shuffling*, *BMiss*, *Hierarchical*, *Roaring* and *SIMDGalloping*. They also compare with implementations of these existing algorithms with BSR sets. The findings can be summarised as follows.

1. For same-size intersection with density is fixed at 1%, *QFilter* outperforms all

other algorithms when selectivity is less than 30%. For selectivity greater than 30%, *Shuffling* outperforms it, but *QFilter* is second-best.

2. When varying skew ratio, *SIMDGalloping* is best for skews greater than 32:1.

3. For same-size intersection with varying density, *QFilterBSR* performs the best out of all *BSR* variants when density is greater than 10%, closely followed by *ShufflingBSR*.

4. For skewed intersection (256:1) with varying density, *Roaring* slightly outperforms *SIMDGallopingBSR* when the density is greater than 10%.

FESIA. The third neglected aspect seen in the above SIMD-based algorithms is the k-set intersection problem as motivated by search engines. The first paper to bridge this gap was published by Song et al. in 2017 [SYL17], however the technique proposed is not considered here as it requires profiling to adjust algorithm parameters. FESIA [ZLSF20], published by Zhang et al., also bridges this gap by providing fast SIMD-based intersection extendable to k-sets. This paper presents a data structure consisting of a bitmap of hashed set elements and a list of segments which store the actual elements corresponding to each hash value. To perform an intersection, the two current bitmap words are first `AND`-ed together to filter out unnecessary comparisons. Then, specialised SIMD "kernels" are used to intersect the corresponding entries in the reversed mapping arrays. Each kernel corresponds to a particular intersection size. For example, `Intersect2x3` is used to intersect a set of two elements with a set of three elements. These specialised kernels are stored in a jump table and are implemented using either SSE, AVX2 or AVX-512 instructions. In the case where the sets are heavily skewed, an alternate intersection algorithm is employed. For each element in the smaller set, this algorithm checks whether the element exists in the larger set by first hashing the element then checking the corresponding bit in the bitmap. Only when the bit is set, the element is looked up in the reversed mapping.

In the same paper, Zhang et al. also provide an experiment comparing *FESIA* to *BranchlessMerge*, *Shuffling*, *BMiss*, *Galloping* and *SIMDGalloping*. It is clear from

these results that for same-size sets with selectivity 1%, all *FESIA* variants are significantly faster than the other algorithms included. However, it is shown that this lead deteriorates as the selectivity increases, with *Shuffling* beating out $FESIA_{\text{SSE}}$ when the selectivity exceeds 10%. The higher vector-width variants $FESIA_{\text{AVX2}}$ and $FESIA_{\text{AVX-512}}$ scale better to greater selectivities, however this does not consider the performance of a hypothetical AVX2 or AVX-512 variant of *Shuffling*. For skews from 16 : 1 to 32 : 1, it is shown that the variant *FESIA Hash* performs significantly faster than all other algorithms. In [HZY18], it was found that *SIMD Galloping* was optimal for ratios greater than 32 : 1. Zhang et al. do not consider skew ratios greater than 32 : 1, so additional experiments are required to confidently assume *FESIA Hash* is optimal for all skews greater than 32 : 1.

It is important to note that FESIA does not perfectly fit into the class of SIMD algorithms described previously. Most notably, the algorithm does not output elements in a sorted order due to the elements being hashed. If sets are already represented as sorted arrays, significant processing is required to hash these elements and construct the required lists. This algorithm is also one of the few which does not have an open-source implementation. In light of these issues, this algorithm is however the first covered which can be scaled to multiple CPU cores to further increase throughput.

Conflict Intersect. A number of SIMD-based set intersection implementations have been published by Frank Tetzel on GitHub.[2] Most notably, this repository contains an AVX-512 algorithm utilising the `vp2conflictd` instruction. This algorithm concatenates two 8-element vectors then uses `vp2conflictd` to find the common elements. Although these implementations are part of a benchmark suite, results are yet to be published, so the performance of this algorithm is unknown.

[2] `https://github.com/tetzank/SIMDSetOperations`

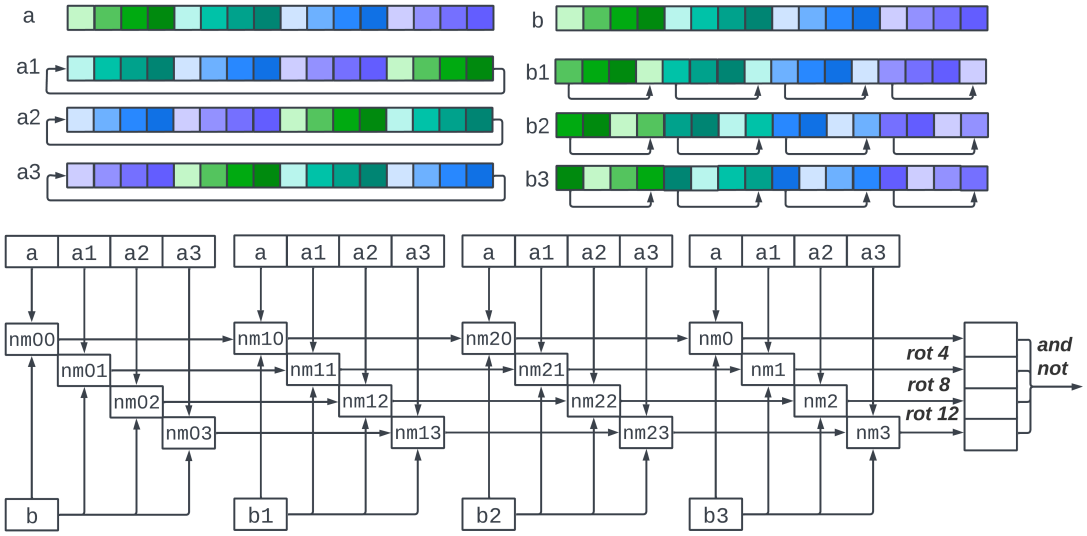


Figure 3.6: *Vp2intersectEmulation* all-pairs comparison

VP2INTERSECT Emulation. `VP2INTERSECT` is an AVX-512 instruction which performs an all-pairs comparison between two 16-element vectors. Existing SIMD algorithms *Shuffling*, *BMiss* and *QFilter* each perform an all-pairs comparison through various shuffling operations and/or lookup tables. It is reasonable to assume that this hardware solution should outperform equivalent software implementations, however Díez-Cañas showed this to not always be the case [DC21]. `VPINTERSECT2` can be emulated by performing inter-block rotations on set a, and intra-block rotations on set b, where blocks consist of 4 elements. All permutations of a and b are then compared and combined with bit operations, depicted in Figure 3.6. This emulation technique, denoted *Vp2intersectEmulation*, was shown by Díez-Cañas to perform better than the native instruction, and the native instruction has since been deprecated. Neither of these have been compared with existing intersection algorithms.

Table 3.1: SIMD widths included in an existing experiment

SIMD Algorithm	Included in an experiment	Not yet included
Shuffling	SSE	AVX2, AVX-512
SIMD Galloping	SSE	AVX2, AVX-512
BMiss	SSE	AVX2, AVX-512
BMissSttni	SSE	-
QFilter	SSE	-
Vp2IntersectEmulation	AVX-512	-
ConflictIntersect	AVX-512	-
Roaring	AVX2	AVX-512
FESIA	SSE, AVX2, AVX-512	-
Broadcast	-	SSE, AVX2, AVX-512

3.4 Gaps in Comparison

3.4.1 AVX2 and AVX-512 Extensions

For each SIMD algorithm, Table 3.1 outlines the widths which these algorithms have been benchmarked with versus possible not-yet-benchmarked widths. Many of these algorithms either have implemented or hypothesised extensions which have not been included in a published experiment. In particular:

- Both ***Shuffling*** and ***SIMD Galloping*** have natural AVX2 and AVX-512 extensions which are yet to be included.
- We hypothesise that it is possible to extend ***BMiss*** to AVX2 or AVX-512, however this extension is non-trivial, so we leave this to future research.
- In the case of ***BMissSttni***, extension is unlikely to be possible as the STTNI instruction does not have an AVX2 or AVX-512 equivalent.
- For ***QFilter***, the SSE version requires a lookup table of 2^{16} elements for the "byte check" phase. Implementing an AVX2 version would require a lookup table of 2^{32} elements which would take several gigabytes of memory, and is hence infeasible.
- ***Vpconflictd*** and ***Vp2intersect Emulation*** are also yet to be included in a published set intersection experiment.

- In March 2023, AVX-512 support was added to the ***CRoaring*** library [Lem23]. This extension is yet to be included in an experiment.

In addition to these extensions, we propose a natural analog to *Shuffling* where each element in set b is ***broadcast*** into its own vector (replicated in all positions), and these vectors are each compared against the other set to produce the final result (shown in Figure 3.7). Although relatively straightforward, this variant is yet to be evaluated against existing SIMD-based algorithms.

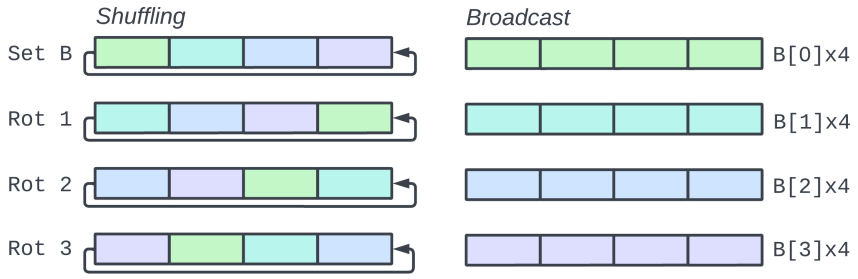


Figure 3.7: *Shuffling* vs. *Broadcast*

3.4.2 FESIA

On first inspection, *FESIA* may appear as a candidate for the optimal algorithm for web, database and graph applications, however, several gaps in comparison prevent this claim from being made. *FESIA* is a fast bitmap-based set intersection algorithm, however it does not compare itself with other leading bitmap algorithms such *Roaring* or *QFilterBSR*. It is important to note that *Roaring* and *BSR* variants benefit from high density input, but *FESIA* should not as elements are hashed. *FESIA* also claims strength in graph applications, however it only compares itself to *Shuffling* and does not consider *QFilterBSR*, an algorithm specifically designed for graph databases. *FESIA* also does not compare itself to *RanGroupScan* – a similar hashed bitmap method.

3.4.3 QFilter for Non-graph Applications

QFilter is designed for graph applications, though its strong performance for same-size sets could position it well in a hybrid algorithm similar to the one found in [IOT14]. In this scenario, it makes sense to test its performance on inverted index or relational database workloads to determine its suitability. These comparisons are yet to be made.

3.4.4 Relational Database Workloads

Many of the discussed algorithms are motivated by the performance of relational database workloads, however the only experiment which employs a real database workload is [WLPS17]. This paper compares bitmap and inverted index compression methods, and does not include any of the discussed algorithms apart from *Roaring*. To verify that synthetic benchmarks translate to real-world input, all set intersection algorithms should be compared in a database environment.

3.4.5 SIMD vs. Scalar Algorithms

Currently, apart from the limited comparisons made in [SYL17], few comparisons have been made between adaptive algorithms and SIMD-based algorithms. Although it could be assumed that SIMD variants outperform the adaptive algorithms due to increased parallelism, this is yet to be comprehensively verified. Similarly, no comparison currently exists between the SIMD algorithms and scalar algorithms with an alternative data structure (e.g., *RanGroupScan*).

3.5 State of Experimentation

3.5.1 Fragmentation

To begin to understand root of these deficiencies, we examine the state of current experiments and determine the various roadblocks. In general, many of the gaps outlined above can be attributed to a fragmentation experiments, where each only includes a subset of datasets and algorithms. In Figure 3.8, we see that experiments are primarily fragmented by *scalar* and *vector* algorithms. We also see that no one experiment comprehensively compares all algorithms in a given category. For example, in FESIA's experiment, it compares itself with *BMiss*, *SIMDGalloping* and *Shuffling*, however it does not include *Roaring* or *QFilter*.

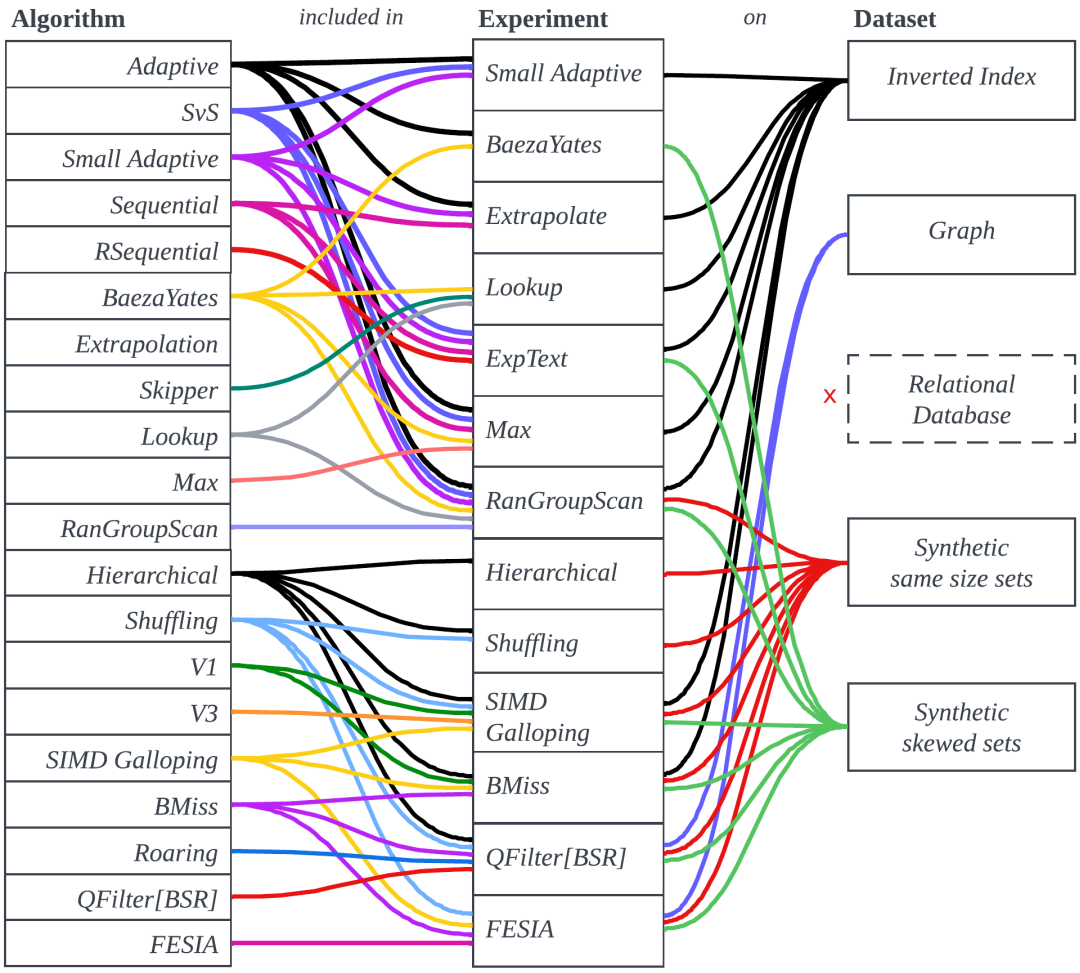


Figure 3.8: Fragmentation by *algorithm* and *dataset* (original papers in Appendix A.1)

3.5.2 Open-source Algorithms

This fragmentation is partially driven by the burden of reimplementing past algorithms for a given experiment. Table 3.2 outlines the open and closed-source algorithms and where open-source code can be found. Many of these algorithms, particularly those which operate on an alternate data structure such as *Skipper*, *Lookup*, *RanGroupScan* and *FESIA*, do not have open-source implementations (to our knowledge). These algorithms are generally more complex than array-based implementations – creating a high barrier to entry for researchers and adopters. In contrast, algorithms which operate on sorted arrays such as the adaptive and SIMD-based algorithms are well represented in open-source implementations, making them easier to include in new research.

Table 3.2: Open and closed-source algorithms

Open-source	*Merge*,[3] *Galloping*,[3] *Adaptive*,[4] *SvS*,[4] *SmallAdaptive*,[4] *Sequential*,[4] *BaezaYates*,[4,5] *Max*,[4] *Hierarchical*,[5,6] *Shuffling*,[5,6,7] *V1*,[5] *V3*,[5] *SIMD Galloping*,[5,6,7] *BMiss*,[6] *Roaring*[8], *QFilter*.[6]
None found	*RSequential*, *Inter/Extrapolation*, *Skipper*, *Lookup*, *RanGroupScan*, *FESIA*.

3.5.3 Open-source Experiments

More important than an open-source algorithm is an open-source experiment. In an open-source experiment, all algorithms are open as a baseline, so reimplementation is straightforward. Furthermore, the algorithms, datasets and benchmarking infrastructure is *verifiable* and *transparent* leading to more trustworthy results. Researchers and adopters are also able to reproduce experiments and translate them to more modern hardware if necessary. Finally, for a new algorithm to be benchmarked, it may be simpler for researchers to include it in an existing open-source experiment instead of constructing a new experiment from the ground up. Of the experiments included in Figure 3.8, only *Shuffling* [Kat12], *SIMD Galloping* [LBK14] and *QFilter* [HZY18] are open-source.

FESIA is a prime example. As a symptom of a *fragmented* comparison-base, *FESIA* is yet to be compared with *Roaring* or *QFilter*. If the experiment were *open-source*, researchers could either add *Roaring* and *QFilter* to *FESIA*'s experiment, or take the *FESIA* source code and include it in a new experiment. Unfortunately this is not the case, and we did not hear back after multiple requests for the source code. Due to the various gaps outlined in Section 3.4, *FESIA* cannot be assumed to be the current optimum – we were left with no option other than to reimplement it.

[3] Included in several repositories listed. [4] github.com/lemire/SIMDIntersections
[5] github.com/lemire/SIMDCompressionAndIntersection [6] github.com/pkumod/GraphSetIntersection
[7] github.com/tetzank/SIMDSetOperations [8] github.com/RoaringBitmap/CRoaring

Chapter 4

Benchmark Suite

As a solution to the lack of comprehensiveness and transparency of existing experiments, we present an open-source set intersection library and benchmark suite.[1] We first outline the algorithms we chose to include in Section 4.1, followed by our implementation decisions in Section 4.2. We then outline how we ensure correctness in Section 4.3 and describe the benchmarking workflow in Section 4.4.

4.1 Algorithms

Although there is a large gap in comparison between SIMD and adaptive algorithms, we put more weight on SIMD algorithms as these are hypothesised to be faster due to their increased parallelism. Ideally, since there is no clear "optimal" adaptive algorithm for all datasets, we would include at least *Small Adaptive*, *SvS*, *BaezaYates*, *ExtrapolLookAhead* and *Max*. Due to time constraints and to limit the scope of this experiment we only implement *SvS* and *Small Adaptive*. If these algorithms perform well compared to SIMD algorithms, this may motivate their inclusion in a subsequent extension to our library. The same applies to scalar algorithms which operate on an

[1] `https://github.com/UNSW-database/simd_set_operations`

alternative data structure such as *RanGroupScan*. We choose to focus primarily on SIMD algorithms and leave these to further experiments.

4.1.1 Array-based Algorithms

Scalar and vector array-based algorithms are selected with the aim of providing a more compete comparison within this category, and to later provide further comparison of these with non-array-based algorithms. This category can be further split into scalar baseline algorithms, existing SIMD algorithms and not-yet-compared SIMD algorithms.

1. **Baselines.** As baselines, we include *NaiveMerge*, *BranchlessMerge*, and the scalar versions of *BMiss* ($BMissScalar_{3\times}$ and $BMissScalar_{4\times}$). We also include skewed intersection baselines *Galloping*, *Binary Search* and *BaezaYates*.

2. **Well-tested SIMD algorithms.** We include the well-tested array-based SIMD algorithms *Shuffling*, *SIMD Galloping*, *BMiss*, and *QFilter*. Note, these are the 128-bit versions of these algorithms. Since we will be extending *Shuffling* and *SIMD Galloping* to greater vector widths, we denote these $Shuffling_{SSE}$ and $Galloping_{SSE}$ respectively.

3. **Extensions and untested SIMD algorithms.** We are extending existing experiments here in two ways. First, we include higher-vector-width implementations of algorithms such as $Shuffling_{AVX2/AVX512}$ and $Galloping_{AVX2/AVX512}$. We do not include AVX2 or AVX-512 versions of *BMiss* or *QFilter* for reasons described in Section 3.4.1. Secondly, we include *Broadcast* variants for all SIMD widths as described in Section 3.4. Finally, we include the currently-untested *Vpconflictd* and *Vp2intersect Emulation* algorithms.

4.1.2 Non-array-based Algorithms

A significant driver of this experiment is the lack of comparison between *FESIA* and *Roaring*, *QFilter*, and *BSR* variants. Naturally, we include all of these for comparison.

We include both the C and Rust implementation of *Roaring* as the C version supports AVX-512 while the Rust version is only SSE. The Rust version also includes extra bounds checks introduced by the Rust compiler.

4.1.3 Hybrid Algorithms

Hybrid algorithms are algorithms which combine multiple "sub-algorithms" to adapt to multiple different input types. For example, a hybrid could be made between *Merge* and *Galloping* where *Merge* is used for skews less than $1 : 32$, and *Galloping* is used for greater skews. We do not include these algorithms in our initial comparisons as they obscure the performance results of their "sub-algorithms." Only after an analysis has been performed on individual algorithms, these hybrid algorithms may be re-introduced to compare the holistic performance of different underlying data structures. This is left to further research.

4.2 Implementation

4.2.1 Rust and Portable SIMD

We implement, test and benchmark these algorithms using the Rust programming language. Rust is a systems language similar to C or C++ with an emphasis on memory safety and reliability. The main reason we use Rust is due to its *Portable SIMD* interface. One down-side of using C or C++ is that compiler intrinsics are required to force the use of SIMD instructions unless libraries like `highway`[2] or `libsmdpp`[3] are used. These tend to make code unreadable, for example, to create a vector of four floats, `f32x4::splat(10.0)` would be written in Rust while the C intrinsic equivalent is `_mm_set1_ps(10.0f)`. It must be noted that Rust's `portable-simd` library is not a

[2] `https://github.com/google/highway` [3] `https://github.com/p12tic/libsimdpp`

stable feature and must be enabled before compilation, however, it is still favourable since no third-party libraries are required.

The generic nature of the `portable-simd` library is another reason this library has been chosen. For example, generic variables of the type `Simd<T,N>` are used in some cases instead of fixing the type to `i32x8`. This allows some algorithms such as *SIMD Galloping* to be implemented agnostic to the underlying datatype and vector width, reducing code repetition. This also makes algorithms more likely to be compatible with future revisions and extensions to SIMD instruction sets. It is important to note, however, that in some cases the `portable-simd` library does not provide access to the specific instructions necessary to implement an algorithm. In this case, non-portable alternatives in Rust's `std::arch` library are used instead. For example, most of *Shuffling* is implemented using portable SIMD, however the final shuffle uses the `_mm_shuffle_epi8` intrinsic as Rust does not support shuffling based on a non-constant pattern.

We also develop in Rust as the language does not have many native set intersection implementations. The only included algorithm currently implemented in native Rust is *Roaring*, so implementing these algorithms in Rust may eventually provide the opportunity for further use in research and development projects (copyright permitting). Rust also has a strong suite of benchmarking and testing libraries, all easily accessible through the `cargo` package manager. Libraries in C/C++ are notoriously difficult to manage, with many resorting to third-party package managers like `vcpkg` and `Conan`.

4.2.2 Set Intersection Interface

To make algorithms interchangeable for benchmarking, we introduce a common interface. For array-based algorithms, sets are always input as a slice of elements (i.e., a reference to a contiguous block of elements). In many cases, this is a completely *generic* slice where the item may be any type which presents an ordering. For SIMD algorithms, the element size is fixed to 4-bytes which restricts inputs to `u32`, `i32`.

Next, we make method of output generic by introducing a *visitor* pattern inspired by the `roaring-rs` library. All algorithms input a reference to a struct implementing the `Visitor` trait, and algorithms call `.visit(a)` to add an element `a` to the output set. This allows algorithms to be used and benchmarked in two configurations: one where simply the size of the intersection is returned (using the `Counter` implementation) and another which returns the resulting set as a `Vec` (using the `VecWriter` implementation). This is relevant as some existing experiments only count elements (e.g., *FESIA*), and other output the whole resulting set. The SIMD algorithms input a struct inheriting `SimdVisitor` which allows visiting using a vector of elements with a bit-mask to denote included elements. The *BSR* algorithms have corresponding `BsrVisitor` and `SimdBsrVisitor` traits. We also include the `SliceWriter` implementation which allows output to be written directly to a slice instead of a vector. This method of abstraction has no overhead as the Rust compiler outputs separate machine code for each implementation, similar to C++ templates.

4.2.3 Performance Overhead

Safety is a defining feature of Rust, however it has the potential to impact the performance of our algorithms. Since we include the `CRoaring` library in our benchmarks, we must ensure that our Rust code has "C-like" performance, i.e., no run-time checks when accessing memory. In Rust, array-accesses often include a bounds check noticeably degrading the performance of set intersection algorithms which frequently access arrays. Although the compiler sometimes removes these checks if determined safe, we negate this issue by accessing all arrays and slices through the unsafe `get_unchecked` method. We review the resulting assembly and confirm these branches are removed.

A similar issue occurs with the representation of sets as vectors in our experiments. When adding an element to the result set, a branch is performed to check if the result vector has the required capacity. We get around this by implementing an alternative `UnsafeWriter` (for benchmarking only) which assumes the vector already has enough capacity.

4.2.4 FESIA Implementation

We explicitly outline our design choices for our FESIA implementation due to its complexity compared to other algorithms. Firstly, our implementation is fully generic in *hash function*, *segment size*, and *SIMD width*. These characteristics define the performance of the algorithm, and such must be easily varied with little code duplication. As in Section 4.2.2, we verify these abstractions to be zero-overhead. The main downside to this abstraction is the boilerplate required by Rust's generic SIMD library. For example, in the struct definition in Figure 4.1, we must explicitly tell Rust that our SIMD vector must support the `BitAnd` and `SimdPartialEq` operations, and that it must be convertible to a bitmask.

We also take steps to ensure our intersection kernels are consistent with the original paper, however we make some assumptions. The kernels as described in the paper only support counting the elements, so we extend this to support arbitrary output using our `Visitor` pattern – substituting the final `popcnt` for a `visitor.visit()`. Zhang et al. state that the cache footprint can be reduced by sub-sampling kernel combinations, however they do not include whether their final benchmarks use this optimisation. We choose a middle ground where we sub-sample kernels in one axis and fully enumerate them in the other, as shown in Table 4.1. This makes the space occupied by the kernels linear in SIMD width as opposed to quadratic.

```
pub struct Fesia<H, S, M, const LANES: usize>
where
    H: IntegerHash,
    S: SimdElement + MaskElement,
    LaneCount<LANES>: SupportedLaneCount,
    Simd<S, LANES>: BitAnd<Output=Simd<S, LANES>> +
                    SimdPartialEq<Mask=Mask<S, LANES>>,
    Mask<S, LANES>: ToBitMask<BitMask=M>,
    M: num::PrimInt
```

Figure 4.1: FESIA generic *where* clause

SIMD Width	Kernel Dimensions							
SSE	1x4	1x8	2x4	2x8	3x4	3x8	...	7x8
AVX2	1x8	1x16	2x8	2x16	3x8	3x16	...	15x16
AVX-512	1x16	1x32	2x16	2x32	3x16	3x32	...	31x32

Table 4.1: FESIA kernel dimension sub-sampling

4.3 Testing

We first test our implementations using the property-based testing library QuickCheck.[4] We use this library to generate random input sets, and verify each algorithm produces the correct result using these inputs. Especially for k-set intersection, these random sets are unlikely to share elements, so the selectivity is usually low. We get around this in two ways. First, we generate sets together, and ensure there exists a common subset of elements. We also implement a program `realdata_test` which performs these same tests using real world input. To ensure our implementations only read and write within array bounds, we run all these tests through an address sanitiser.

4.4 Benchmarking

We initially attempt to run benchmarks using the `Criterion.rs`[5] microbenchmarking tool, though this proves unsuitable for a number of reasons. First, the types of benchmarks we want to run are not *micro*benchmarks – we would like to intersect large sets to reduce measurement overhead and uncertainty. Generating more than 1M items per set for hundreds of inputs per benchmark causes benchmarking to be unreasonably slow. Another issue with this method is that datasets and experiments are required to be defined programmatically. This is fine for microbenchmarks, but for an experiment with dozens of datasets and algorithm combinations, the overhead of recompilation and managing such a codebase becomes excessive.

[4] `https://github.com/BurntSushi/quickcheck` [5] `https://github.com/bheisler/criterion.rs`

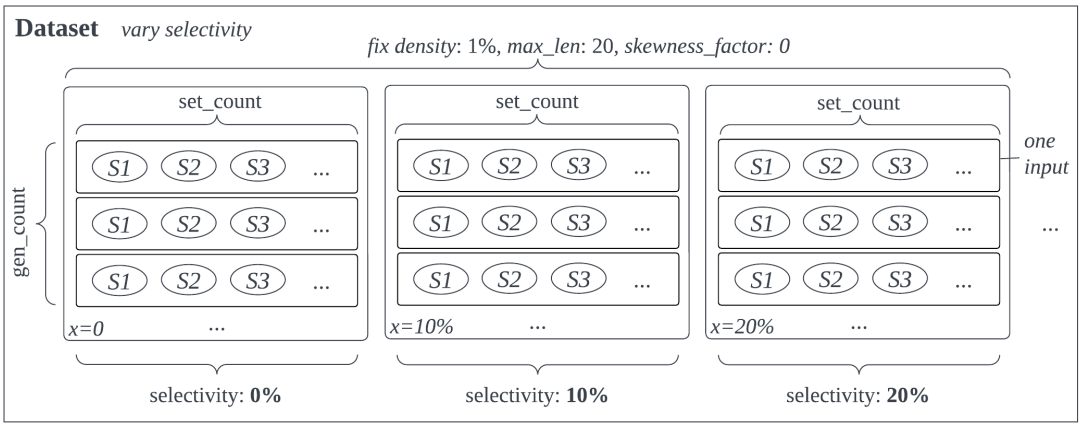


Figure 4.2: Dataset structure

To mitigate these issues, we present a custom benchmarking suite. First, the user declaratively describes datasets and experiments in `experiment.toml`. Next, datasets are pre-generated using the `generate` command. The benchmarks are then run using the `benchmark` command which outputs results to a JSON file. Finally, these results can be summarised using our plotting scripts or an alternative workflow if required.

4.4.1 Defining Datasets and Experiments

Users first define a list of datasets and experiments in `experiment.toml`. Datasets are specified as entries in the `[[dataset]]` list. Synthetic datasets are a collection of groups of input sets, where each input group forms an intersection of a specific *selectivity*, *density*, *skew* and *set count*. One of these parameters is varied along the x-axis, and the others remain fixed. For each x-value, `gen_count` groups of inputs are stored allowing results to later be averaged. Each of these groups contain `set_count` sets. Figure 4.2 shows an example dataset which varies selectivity in increments of 10%.

Figure 4.3 reflects this dataset as it appears in `experiment.toml`. We see that this dataset varies selectivity from `0` to `1000` in increments of `100`. This corresponds to a **selectivity** varied from 0 to 100% with an increment of 10%.

```toml
[[dataset]]
name = "2set_vary_selectivity" # unique id
type = "synthetic"
set_count = 2          # pairwise intersection
gen_count = 10         # generate 10 pairs for each x
vary = "selectivity"   # vary selectivity along x-axis
selectivity = 0        # from 0-100% with a step of 10%
to = 1000
step = 100
skewness_factor = 0    # skew of 1:2^0 === 1:1
density = 10           # density 1%
max_len = 20           # each 2^20 (approx 1M) elements
```

Figure 4.3: Example dataset definition in `experiment.toml`

We then see that the **density** is fixed to `10` (i.e., 1%). This defines the ratio of the size of the largest set to the size of the element space m, where elements are randomly sampled from the set $\{0, 1, ..., m-1\}$. For example, if the size of the largest set is 16 and the density is 25%, the elements are sampled from $\{0, 1, ..., 63\}$.

Next, the **size** of the largest set is specified by setting `max_len` to `20`. We define a `max_len` of n to result in an actual set size of 2^n, so in this example, the largest set has $2^{20} \approx 1$M elements.

Finally, we define the **skew** by specifying a `skewness_factor` of `0`. This variable defines the size of sets in relation to the size of the largest set. It is represented by an integer s which maps to the floating point number $f = s/1000.0$. For an intersection $S_1 \cap S_2 \cap ... \cap S_k$ where sets are ordered from largest to smallest, the size of the kth set is $|S_k| = |S_1|/k^f$. In this example, we simply want a skew of 1:1, so we set $f = 0$. To represent a skew of $1 : 2^5$ for example, we'd set $f = 5$. This notion of skewness allows skew to be extended to k-sets. Since a set's size is inversely proportional to its rank, this somewhat follows Zipf's Law [Zip36] (with the caveat that these set sizes aren't randomly sampled from large corpus).

Now that we have a dataset, we need to specify the algorithms we want to run on it. To do this, we create an `[[experiment]]` as shown in Figure 4.4. We specify that we want to run *NaiveMerge*, *BranchlessMerge*, and both *BMissScalar* variants on the dataset we defined previously. This will later be used when running benchmarks.

```toml
[[experiment]]
name = "scalar_2set_vary_selectivity"
title = "Scalar 2-set varying selectivity"
dataset = "2set_vary_selectivity"
algorithms = [ "naive_merge", "branchless_merge",
               "bmiss_scalar_3x", "bmiss_scalar_4x" ]
```

Figure 4.4: Example experiment definition in `experiment.toml`

4.4.2 Generating Datasets

We automate dataset generation through the `generate` command. This program interprets `experiment.toml` and generates synthetic sets in parallel, writing them to `datasets/id/x/i`, where `id` is the dataset ID, `x` is the current x-value and `i` is the ith input group for this x-value. We design a simple input group file format called `datafile` which allows inputs to be read into a vector with a single copy of the bytes as they appear in the file. This is significantly faster than storing the sets as text. Generating datasets up-front in this manner separates this overhead from benchmarking, tightening the rate at which benchmarks can be iterated on.

We also create a program `datatest` to validate these output datasets, ensuring their *selectivity*, *density*, *skew* and *set count* match the parameters in `experiment.toml`.

4.4.3 Running Benchmarks and Plotting Results

We develop a custom benchmarking runtime which times algorithms on datasets as specified in `experiment.toml`, outputting results to a JSON file. The first notable feature of this runtime is that if an algorithm appears in multiple experiments on the same dataset, it will only be benchmarked once per dataset. This means we can make several comparisons on the same dataset without increasing benchmark time. Another feature of this suite is that it supports benchmarking by both *wall-clock* time and CPU *clock-cycles*. Although we prefer *wall-clock* time to better reflect CPU frequency scaling, we choose to also support *clock-cycles* to replicate experiments like *FESIA*. Since we are intersecting sets of millions of items, the added overhead and variance of

getting the current time does not impact results.

To plot results, we simply create a script `./scripts/plot.py` which uses Pandas[6] and Matplotlib[7] to process and plot results. We also include a script `summary.py` which generates a web document with all plots and datasets.

4.4.4 Real Datasets

A large requirement of our benchmark suite is the ability to benchmark set intersection algorithms on real datasets. To simplify this process, we define a "real" dataset as a collection of pre-defined sets. Input groups are simply a random sample of sets from this corpus. This has the downside that these datasets are only "real" in the sense that the underlying sets are real, however the queries which select these sets are not. Ideally, we would also employ these real queries, however the datasets we chose did not contain this information.

Based on this structure, all real datasets are first transformed into text files where each line holds a set of integers separated by whitespace. The process to convert real datasets into this form usually just requires a simple Python script. From here, these text files are placed in the `datasets` directory and added to `experiment.toml` as shown in Figure 4.5. These are automatically processed into set groups which are output as `datafile`s to the `dataset` directory exactly the same as synthetic datasets. This means the benchmark implementation is almost completely agnostic to whether the underlying dataset is real or synthetic.

[6] `https://pandas.pydata.org/`

[7] `https://matplotlib.org/`

```toml
[[dataset]]
name = "webdocs"    # Unique id
type = "real"       # Real dataset
source = "webdocs"  # Look for sets in datasets/webdocs.dat
gen_count = 1024    # Take 1024 groups of sets
set_count_start = 2 # Intersect 2,3,4,..,8
set_count_end = 8   # sets on the x-axis
```

Figure 4.5: Example real dataset in `experiment.toml`

Chapter 5

Experiment

5.1 Outline

We now employ our benchmark suite to perform an updated experimental analysis. To focus our research and reduce scope, we frame our experiment around three central questions.

1. Which array-based algorithms are optimal when we include the extensions described in Section 3.4.1?
2. How does the AVX-512 version of *Roaring* compare to these extended and existing array-based algorithms?
3. Where does FESIA fit within this picture?

To answer these questions, we first benchmark *bottom-up* with synthetic data varying *selectivity*, *density* and *skew*. We then analyse how these results carry to real input using *WebDocs* [LOPS04] – an inverted index dataset. For each of these datasets, we benchmark on both an *Intel Xeon Gold 6342* and an *Intel Core i5 7600k*. The Xeon is an Ice Lake AVX-512 CPU and the 7600k is Kaby Lake up to AVX2. Through this,

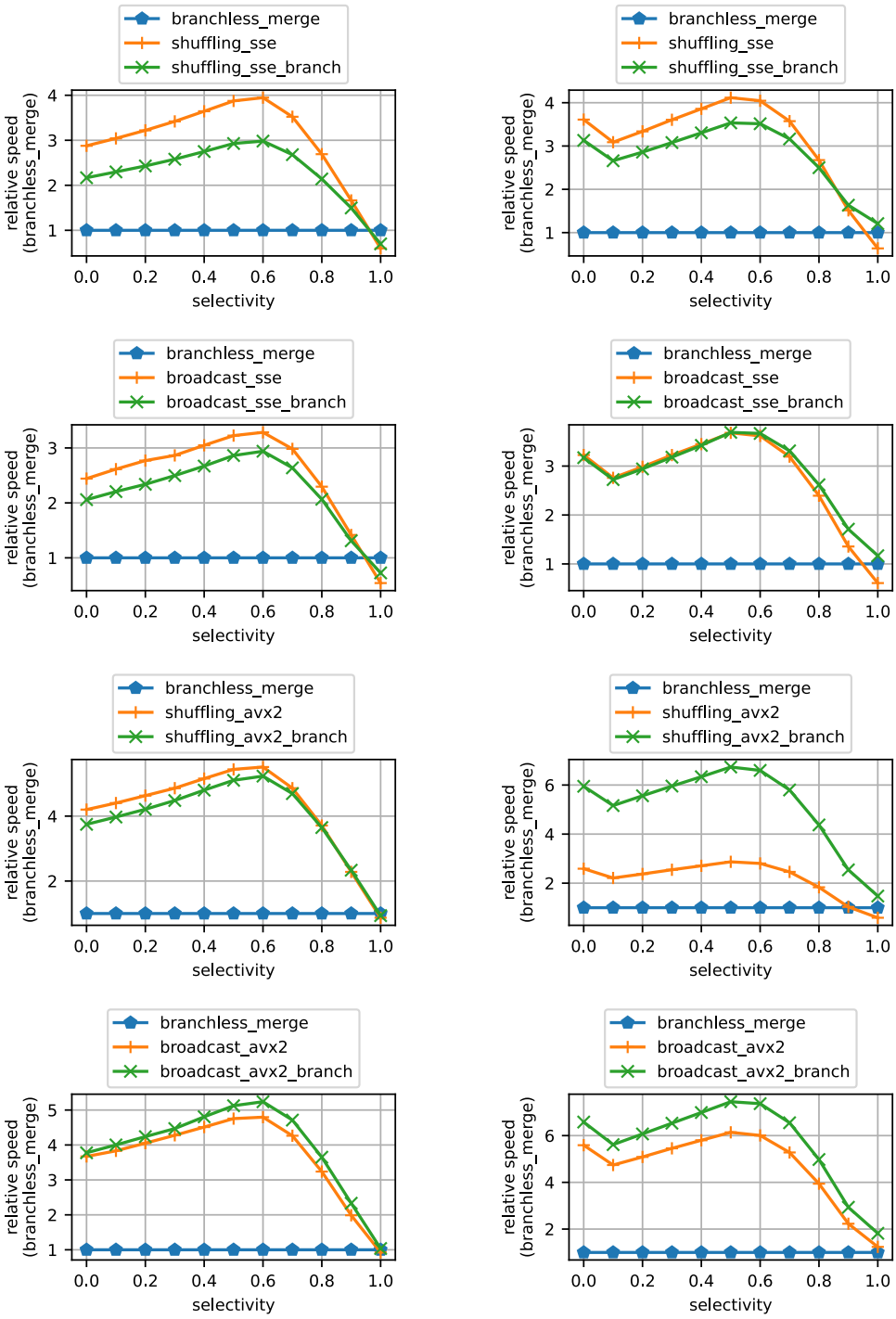
we explore the impact of differing SIMD architectures using a modern AVX-512 server chip and a slightly older AVX2 consumer chip as a case study.

Due to the number of algorithms, for each of these datasets, we individually introduce the SSE, AVX2 and AVX-512 array-based algorithms, only including the top performers from the previous SIMD width. Three variants of Roaring: *CRoaring*, *CRoaringOpt*, and *RoaringRs*, are then compared with these top performers. *CRoaring* and *CRoaringOpt* use the C implementation of the Roaring library, where the *Opt* version includes a call to `run_optimise()` before intersection. *RoaringRs* is the Rust implementation, only supporting SSE. We do not introduce FESIA until after these synthetic benchmarks as its results are not regular. We only include *Small Adaptive* for the k-set *WebDocs* dataset since it is not designed for 2-set intersection.

5.2 Branching During Iteration

To determine the optimal implementation of each algorithm, we first study the impact of branching on set intersection performance. The default iteration type *branchless* uses a conditional move to update the set indices and loads a vector for each set every iteration. The alternate iteration type *branch* performs conditional branches to check whether `set_a`'s index should be incremented, `set_b`'s index should be incremented, or both. The vector for each set is only re-read from memory if that set's index is incremented. Here, unnecessary loads are avoided at the cost of performing a potentially expensive branch.

For each SIMD algorithm, we compare the performance of *branch* vs. *branchless* on both the Xeon and 7600k CPU. Figure 5.1 shows a subset of these results for the *Shuffling* and *Broadcast* algorithms, with the rest of the results in Appendix A.2. For the array-based algorithms, we perform a 2-set intersection varying selectivity from 0 to 100%, with a density of 0.1% and a skew of 1:1. First, focussing on the 7600k, we observe that neither branch nor branchless are optimal for all algorithms: branch is better for $Broadcast_{\text{AVX2}}$ and branchless is better for $Broadcast_{\text{SSE}}$, $Shuffling_{\text{SSE}}$ and



(a) Intel i5 7600k (b) Intel Xeon Gold 6342

Figure 5.1: Branch vs. branchless algorithms
(2-set intersection varying selectivity, $2^{20} : 2^{20}$ elements, 0.1% density)

Shuffling$_{\text{AVX2}}$. Secondly, we observe that these results are not consistent when carried to the Ice Lake Xeon. For example, the branchless version of *Shuffling*$_{\text{AVX2}}$ is optimal on the 7600k, but the branch version is significantly faster on the Xeon. A number of factors could be impacting the relative cost of branching compared to always loading the SIMD vector, causing this behaviour. These may include

- the differing number and types of instructions and produced on the Ice Lake CPU, namely the use of mask registers,
- differing branch predictors, where the relative cost of branching on the Xeon is lower than the cost on the *i5 7600k*,
- differing SIMD vector load costs, where the relative cost of loading on the Xeon is higher than the cost on the *i5 7600k*.

Further research must be done to understand which of these factors contribute to this inconsistency.

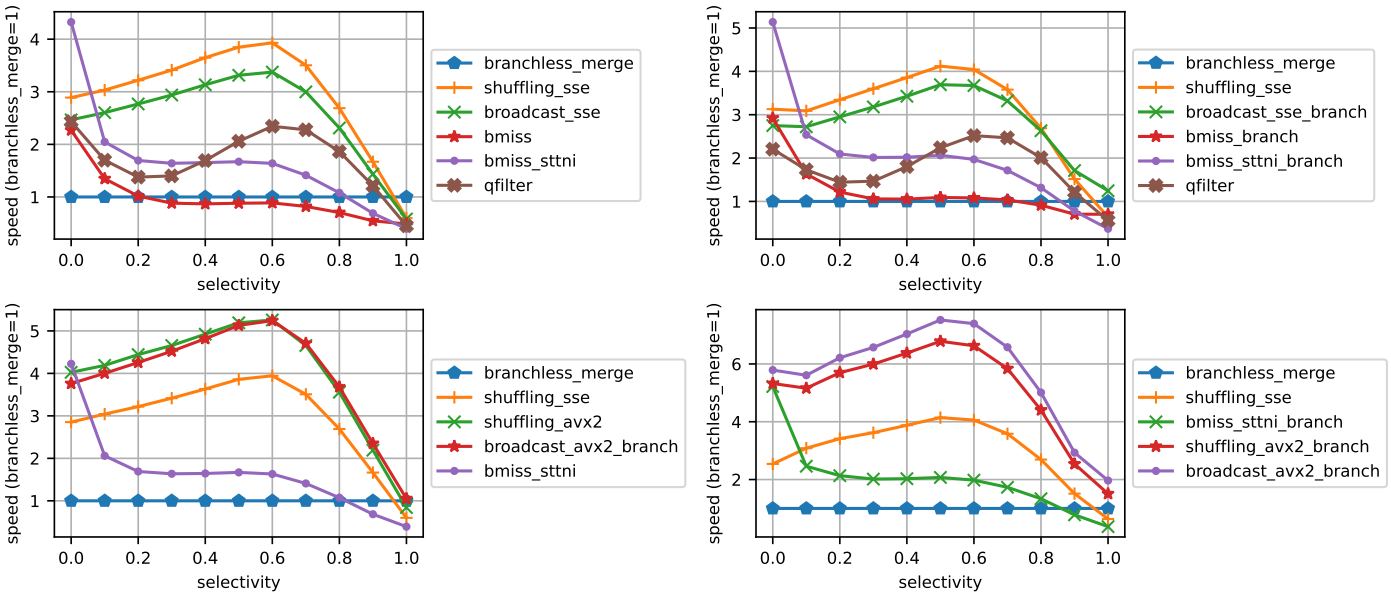
To measure the impact of branching for the BSR algorithms, we perform a 2-set intersection varying density from 0 to 100%, with a selectivity of 3% and a skew of 1:1. From the results in Figures A.3, A.4 and A.5, we find that branching is optimal for all BSR algorithms of all vector widths on both the Xeon and 7600k. These results are expected. On each index update, BSR algorithms load two vectors per set: one for the *bases* and one for the *states*. This doubles the cost of loading with respect to branching compared to non-BSR algorithms. Since the branching variant performs fewer loads, it is significantly faster than the branchless variant for BSR algorithms.

5.3 Results Varying Selectivity

With the optimal branching configuration of each algorithm determined, we now begin our experiment by comparing algorithms varying selectivity. We time 2-set intersections of same size sets varying selectivity, with density fixed at 0.1%. As mentioned, we will

introduce the algorithms by vector width and only include the top performers from the previous vector widths.

SSE Array-based Algorithms. In Figure 5.2, we see that *BMissSttni* is optimal for selectivities less than 5% and $Shuffling_{\text{SSE}}$ is optimal for all other reasonable selectivities (generally selectivity is less than 30% [IOT14, ZLSF20]). The key takeaway from this comparison is that *Broadcast* does not outperform *Shuffling* with SSE instructions. Interestingly, *QFilter* does not perform as expected. This is unusual as our implementation is an almost identical translation of the one found in Han et al.'s open source repository. This could be due to differing hardware – QFilter makes heavy use of lookup tables whose performance could be impacted by varying cache sizes and/or RAM latencies. Another factor could be a differing compiler – we compile with `rustc` and LLVM however it is possible that other compilers such as GCC may produce different output. Further investigation is required.



(a) Intel i5 7600k (b) Intel Xeon Gold 6342

Figure 5.2: Array-based algorithms varying selectivity (SSE upper, AVX2 lower) (2-set intersection, $2^{20} : 2^{20}$ elements, density 0.1%)

AVX2 Array-based Algorithms. Extending the vector width to AVX2, we find that the AVX2 extensions to these array-based algorithms are generally more performant compared to their SSE counterparts (on both CPUs). The only exception is that *BMissSttni* performs marginally better than $Shuffling_{\text{AVX2}}$ and $Broadcast_{\text{AVX2}}$ for particularly low selectivities ($< 1\%$) on the 7600k. Interestingly we find that $Shuffling_{\text{AVX2}}$ performs marginally better than *Broadcast* on the *i5 7600k*, but $Broadcast_{\text{AVX2}}$ performs significantly better than *Shuffling* on the Xeon. We hypothesise this is due to the differing way the *broadcast* instruction is represented on *Kaby Lake* (AVX2) versus Ice Lake (AVX-512). In Figure 5.3, we see that the broadcast operation has its own instruction `vpbroadcastd` on the *Kaby Lake* machine. On the Ice Lake machine, it is instead merged into the `vpcmpeqd` instruction. This reduced instruction count for *Broadcast* on the Ice Lake machine is a likely cause for its improved performance over *Shuffling*. At the very least, it improves instruction cache performance. It also may reduce the latency of this operation.

```
vpbroadcastd ymm0, dword ptr[r12+4*r14]
vpbroadcastd ymm1, dword ptr[r12+4*r14+4]
vpbroadcastd ymm2, dword ptr[r12+4*r14+8]
...
vpcmpeqd ymm0, ymm8, ymm0
vpcmpeqd ymm1, ymm8, ymm1
vpor ymm0, ymm1, ymm0
vpcmpeqd ymm1, ymm8, ymm2
vpcmpeqd ymm2, ymm8, ymm3
vpor ymm1, ymm1, ymm2
...
```

(a) Intel i5 7600k (Kaby Lake)

```
vpcmpeqd k0, ymm1, dword ptr
                    [r12+4*r14]{1to8}
vpcmpeqd k1, ymm1, dword ptr
                    [r12+4*r14+4]{1to8}
korb k0, k1, k0
vpcmpeqd k1, ymm1, dword ptr
                    [r12+4*r14+8]{1to8}
vpcmpeqd k2, ymm1, dword ptr
                    [r12+4*r14+12]{1to8}
korb k1, k1, k2
korb k0, k0, k1
```

(b) Intel Xeon Gold 6342 (Ice Lake)

Figure 5.3: $Broadcast_{\text{AVX2}}$ assembly output

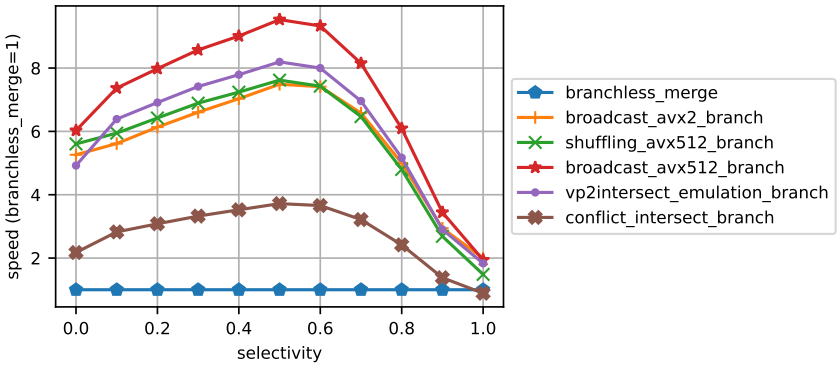
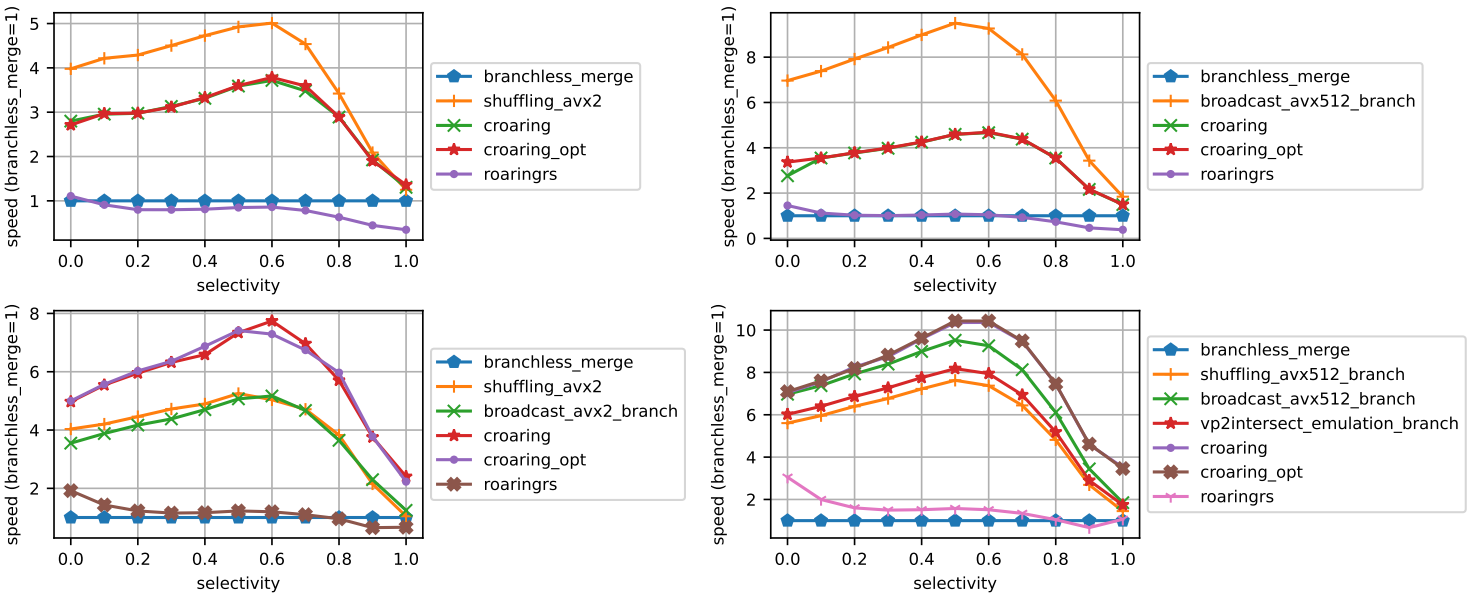


Figure 5.4: AVX-512 array-based algorithms varying selectivity (Xeon)

AVX-512 Array-based Algorithms. Introducing the AVX-512 algorithms on the Xeon in Figure 5.4, we find that $Broadcast_{\text{AVX-512}}$ outperforms all other array-based algorithms for all selectivities on this dataset. On this hardware, the AVX-512 broadcast-based algorithms outperform their shuffling-based peers. The reason for this could be again due to the reduced instruction count seen with broadcast on Ice Lake. The shuffling-based variants require a separate instruction to first load then shuffle the vectors, whereas the broadcast-based variants can simply load and compare in a single instruction.

Roaring. Finally, in Figure 5.5, we compare the Roaring variants against the top performers from the previous experiments. At 0.1% density, We find that none of the Roaring variants are competitive with the best array-based variants on either CPU. The Rust implementation of Roaring performs significantly worse than the C implementations due to its reduced vector width and additional branching introduced by the Rust compiler. We also see that calling `run_optimise()` does not significantly impact performance for this dataset.



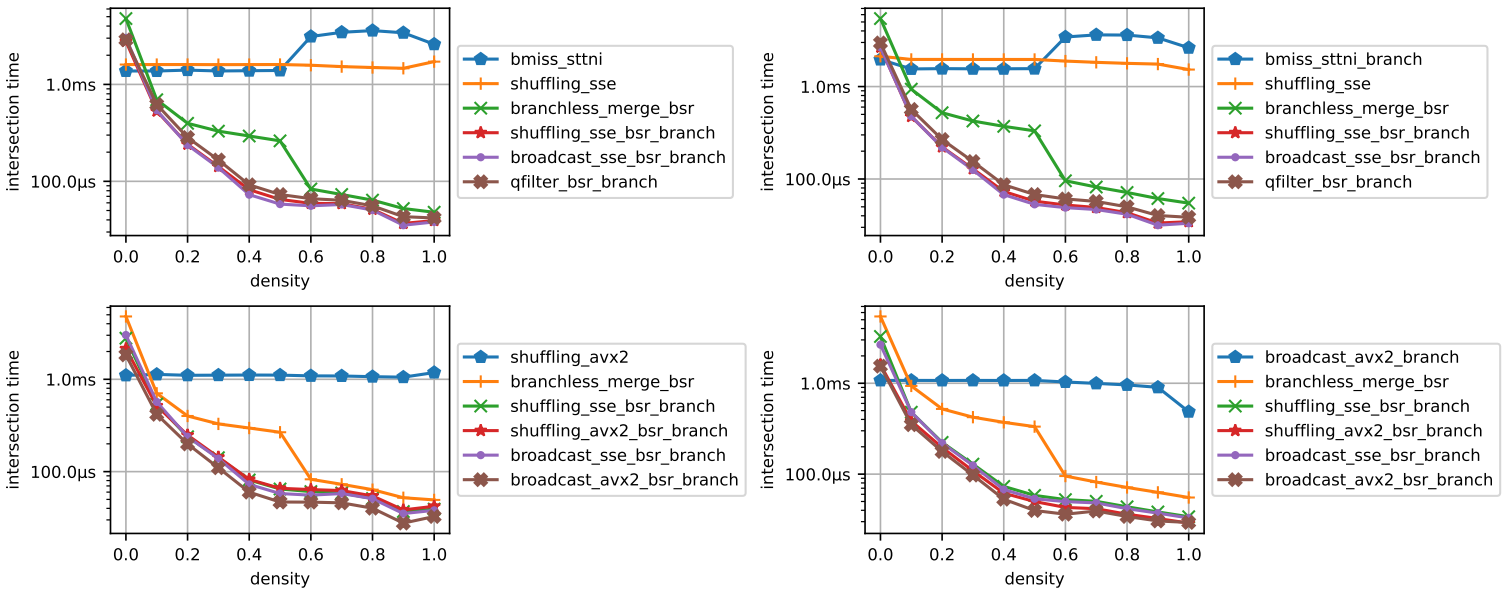
(a) Intel i5 7600k (b) Intel Xeon Gold 6342

Figure 5.5: Roaring varying selectivity (0.1% density upper, 1% density lower)

Interestingly, when we increase the density to 1%, *CRoaring* significantly outperforms the array-based algorithms. This shows us that the AVX2 and AVX-512 versions of Roaring can be optimal for densities as low as 1%. To better understand this relationship, we now benchmark varying density.

5.4 Results Varying Density

In this section, we benchmark 2-set intersections of same size sets varying density, with selectivity fixed at 3%. We introduce the BSR variants of the array-based to understand how these perform extended to higher vector widths. We finally compare Roaring with the best algorithms from each category. It is important to note that after the density increases above 50%, it is no longer possible to fix the selectivity. After this point, as density approaches 100%, selectivity also approaches 100%. This is why we see a change in performance in some algorithms for densities greater than 50%. Also note that the plots in this section show intersection time, so lower is better.



(a) Intel i5 7600k

(b) Intel Xeon Gold 6342

Figure 5.6: Array & BSR algorithms varying density (SSE upper, AVX2 lower)
(2-set intersection, $2^{20} : 2^{20}$ elements, selectivity is 3% from 0–50% density)

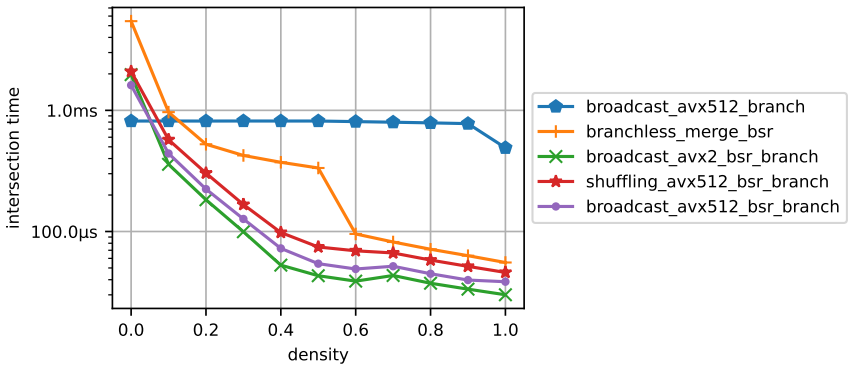
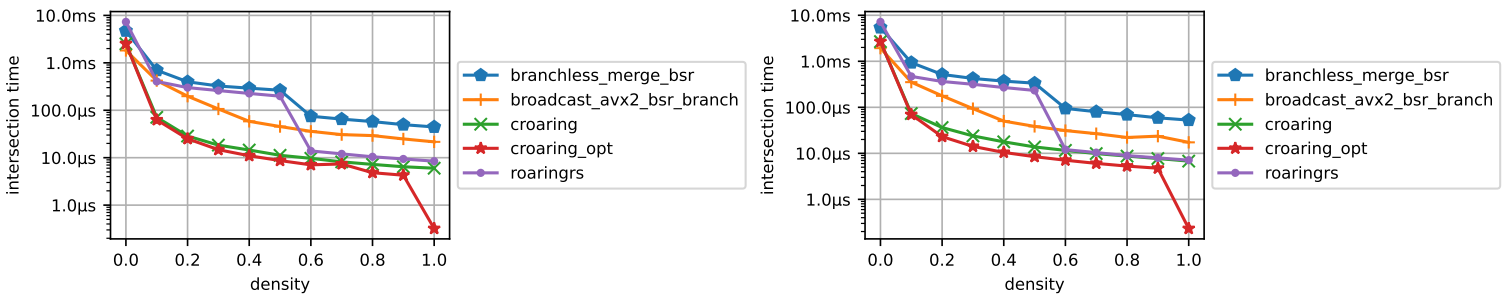


Figure 5.7: AVX-512 array & BSR algorithms varying density (Xeon)

SSE & AVX2 Array-based & BSR Algorithms. First off, in Figure 5.6, we see that the BSR algorithms overtake the array-based algorithms at around 5–10% density. This is expected as the size of the BSR vectors decrease as the density in increases. For both SSE and AVX2, on both the 7600k and the Xeon, we find that the higher-width broadcast variants outperform the equivalent-width shuffling variants.

AVX-512 Array-based & BSR Algorithms. When extending the vector width to AVX-512 on the Xeon processor in Figure 5.7, we observe that the AVX2 version of *Broadcast* outperforms both AVX-512 variants. This is a good example of how the performance boost of an increased vector width is not free – not only is it more expensive to load 512-bit vectors, more shuffle/broadcast instructions are required in the loop of an AVX-512 algorithm. For the array-based algorithms, we see that the performance benefit due to increased parallelism outweighs the added cost. BSR algorithms, however, require additional loads and broadcasts/shuffles within the loop, such that the cost of extending to AVX-512 outweighs performance gains due to parallelism. A more precise analysis of the assembly instructions is required to fully understand this dynamic.

Roaring. Introducing Roaring in Figure 5.8, we see that *CRoaring* significantly outperforms the BSR algorithms on both CPUs. Similar to the previous dataset, *CRoaring* beats *RoaringRs* due to its increased vector width and reduced branching. Unlike the previous dataset, we see that calling `run_optimise` does marginally improve *CRoaring*'s



(a) Intel i5 7600k (b) Intel Xeon Gold 6342

Figure 5.8: Roaring & BSR algorithms varying density

performance. This is because this function compresses large runs of elements, which only tend to appear in synthetic data when the density approaches 100%. Hence, for 100% density, *CRoaring* performs significantly faster after calling `run_optimise`.

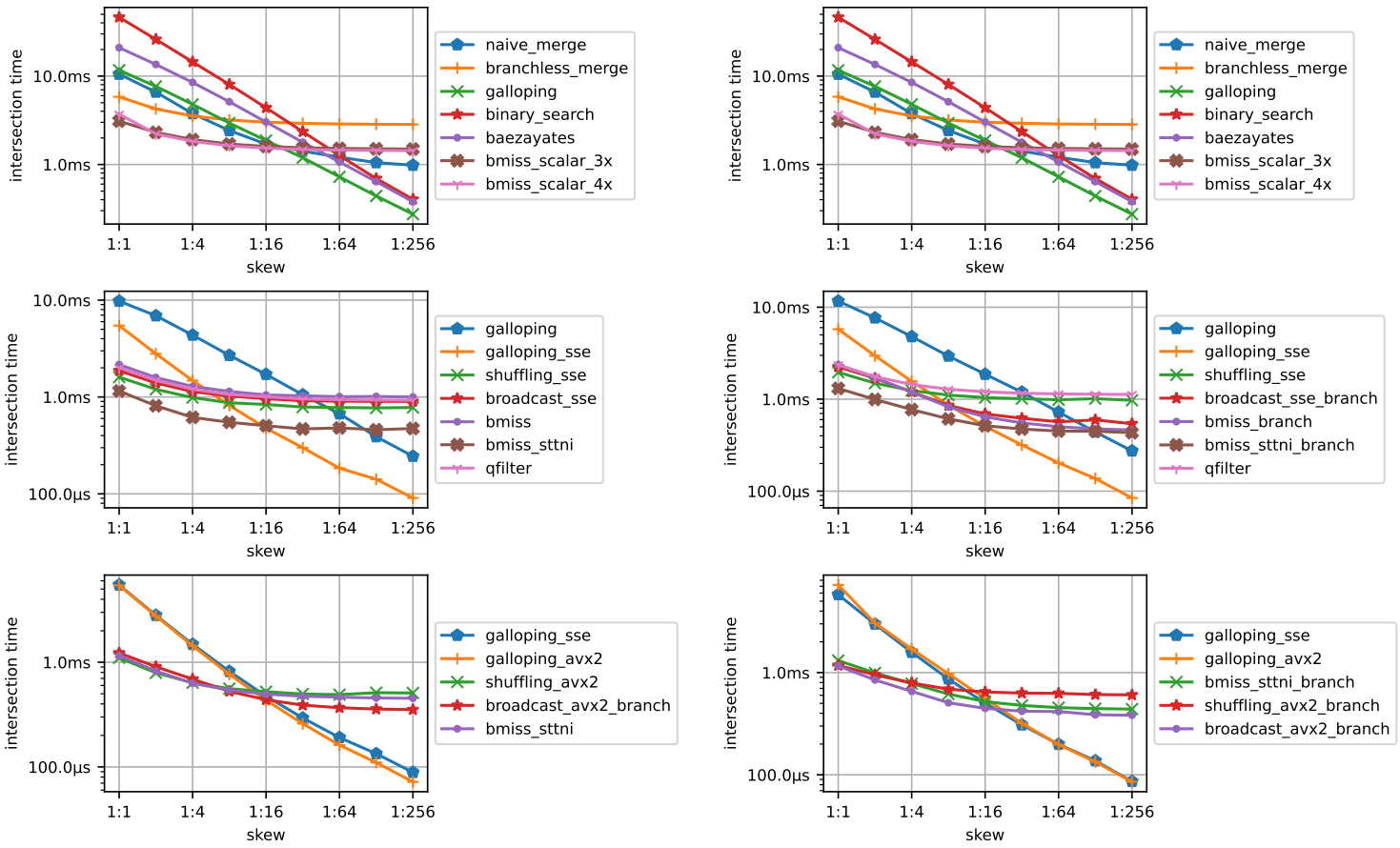
5.5 Results Varying Skew

So far sets we have intersected have been of the same size, but in reality, sets are often skewed. Existing research has shown *SIMD Galloping* is optimal when intersecting arrays of skews greater than 1:32 [LBK14]. We now benchmark 2-set intersection varying skew with the goal of understanding how extensions to *SIMD Galloping* compare to the SSE implementation. We fix density at 0.1% and selectivity at 1%.

In Figure 5.9, out of the scalar algorithm, *Galloping* performs the best for skews greater than 1:32 (on both CPUs). Extending this to SSE, we see that $Galloping_{\text{SSE}}$ (i.e., SIMD Galloping) performs significantly faster than the other algorithms for skews greater than 1:32. Introducing the AVX2 and AVX-512 variants, we find these variants do not offer significant performance improvement over the SSE version. This is because extending the vector width only increases the parallelism of the *last* comparison. This is a constant time improvement, unnoticeable at such large time scales.

We finally introduce Roaring in Figure 5.11 to evaluate its updated performance on skewed datasets. As expected, Roaring does not scale as well to high skews compared to SIMD Galloping. This is because it is a bitmap-based algorithm which does not

employ any sort of binary/galloping search. We can see that it performs much better with a density of 1% compared to 0.1%, however it is still beaten by SIMD Galloping for skews greater than 1:32.



(a) Intel i5 7600k

(b) Intel Xeon Gold 6342

Figure 5.9: Array-based algorithms varying skew (top to bottom: scalar, SSE, AVX2) (selectivity 1%, density 0.1%)

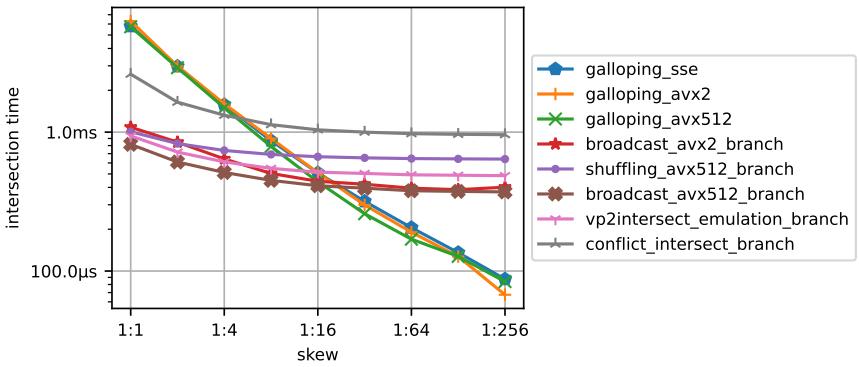
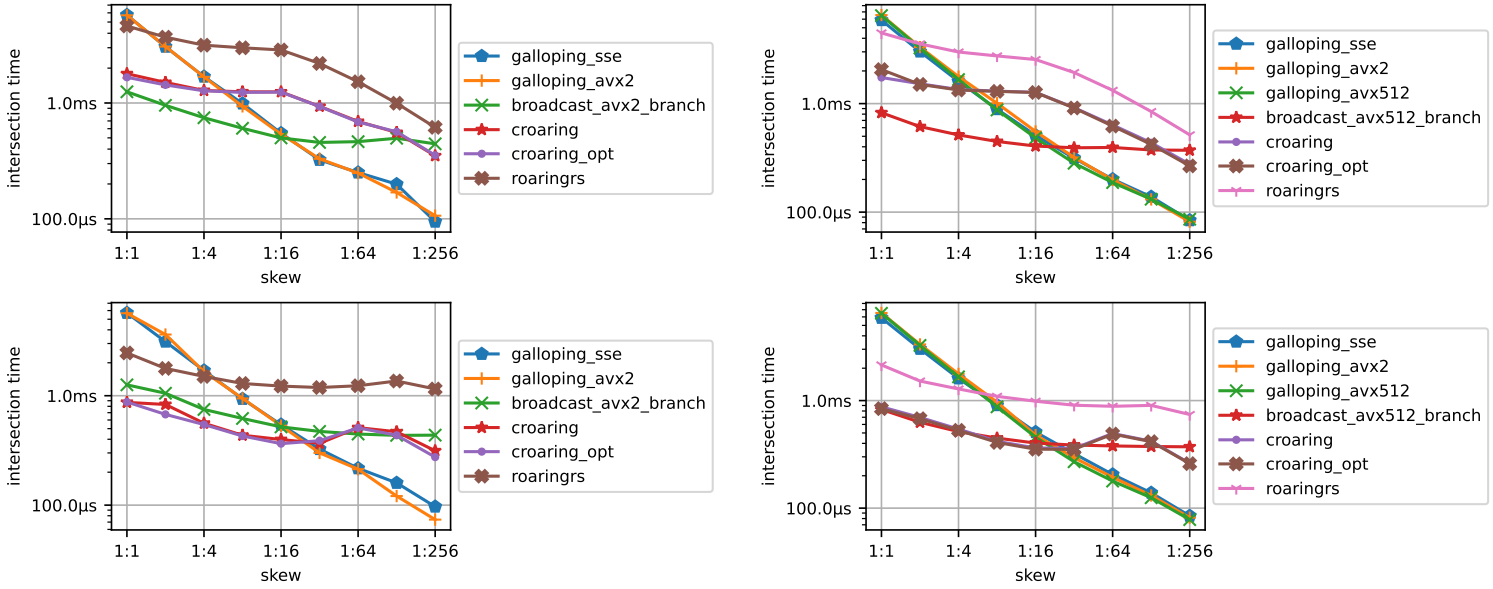


Figure 5.10: AVX-512 array & BSR varying skew (Xeon)



(a) Intel i5 7600k

(b) Intel Xeon Gold 6342

Figure 5.11: Roaring varying skew

5.6 Performance of FESIA

We now investigate where FESIA fits into this picture using our custom implementation described in Section 4.2.4. The FESIA data structure has three variables which can be varied: *segment size*, *SIMD width* and *hash scale*. For each SIMD width, we experimentally determine the best segment size and hash scale balancing high performance at low selectivities with reasonable scaling to higher selectivities. These variants are outlined in Table 5.1. FESIA Hash denotes the non-SIMD hash table version of FESIA aimed at higher skews. FESIA Hash's performance increases with hash scale,

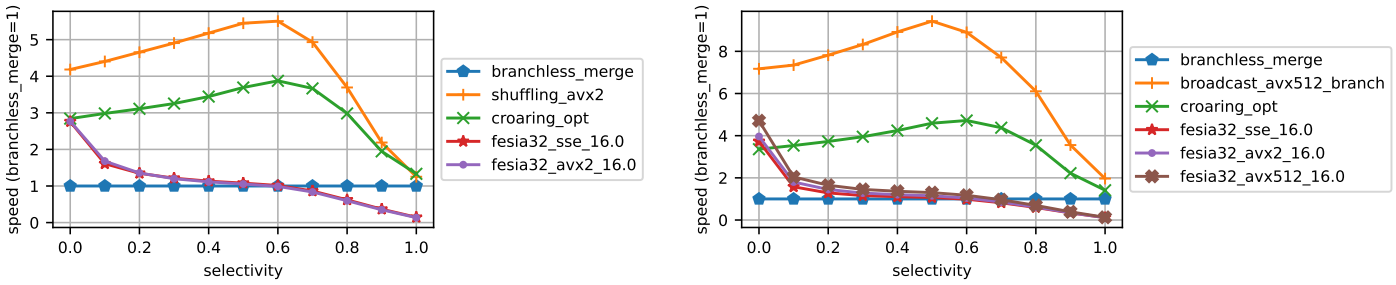
Table 5.1: Optimal FESIA variants for each SIMD width

Variant	Segment size	Hash scale	Name
FESIA SSE	32	16.0	`fesia32_sse_16.0`
FESIA AVX2	32	16.0	`fesia32_avx2_16.0`
FESIA AVX-512	32	16.0	`fesia32_avx512_16.0`
FESIA Hash	8	64.0	`fesia_hash8_64.0`

so to ensure we don't use too much memory, we cap hash scale at 64.0.

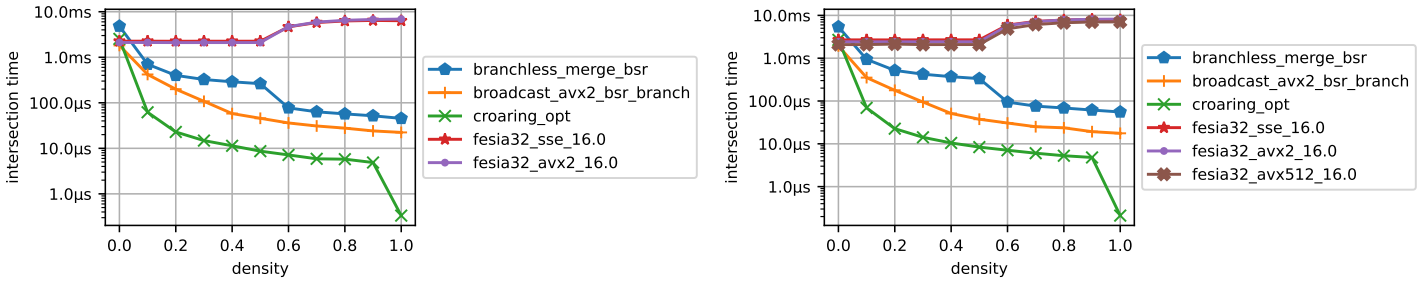
We benchmark FESIA against the better-performing algorithms from the previous section, first varying selectivity in Figure 5.12. We see that our implementation is not competitive compared to *Shuffling* or *Broadcast* even at low selectivities. Varying density in Figure 5.13, as expected, FESIA does not scale with density as it hashes elements before inserting them in its internal bitmap. Varying skew in Figure 5.14, we see that the standard variants of FESIA do not scale with skew, as expected. Interestingly, FESIA Hash seems to scale better than SIMD Galloping for skews greater than 1:64. This makes sense as FESIA Hash has $O(1)$ search while galloping has $O(\log n)$ search.

FESIA's suboptimal performance is likely at least partially due to inconsistencies with Zhang et al.'s implementation. Our implementation is written completely generically, so we may be missing out on optimisations which could be made for particular segment sizes and SIMD widths. However, due to its complexity and closed-source nature, this raises questions as to the efficacy of this algorithm given a best-effort reimplementation cannot replicate results as shown in the paper. Given the simplicity of the array-based algorithms and the open-source nature of Roaring, these algorithms are more suited to real-world use.



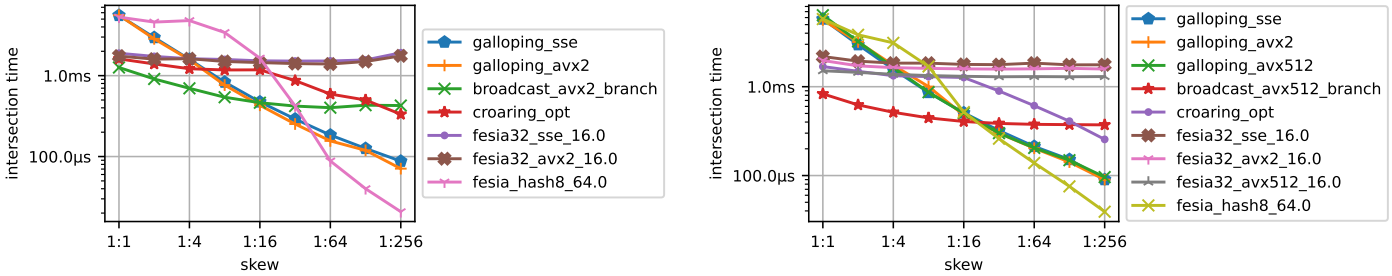
(a) Intel i5 7600k (b) Intel Xeon Gold 6342

Figure 5.12: FESIA varying selectivity (2-set intersection, $2^{20} : 2^{20}$ elements, density 0.1%)



(a) Intel i5 7600k (b) Intel Xeon Gold 6342

Figure 5.13: FESIA varying density (2-set intersection, $2^{20} : 2^{20}$ elements, selectivity 3%)



(a) Intel i5 7600k (b) Intel Xeon Gold 6342

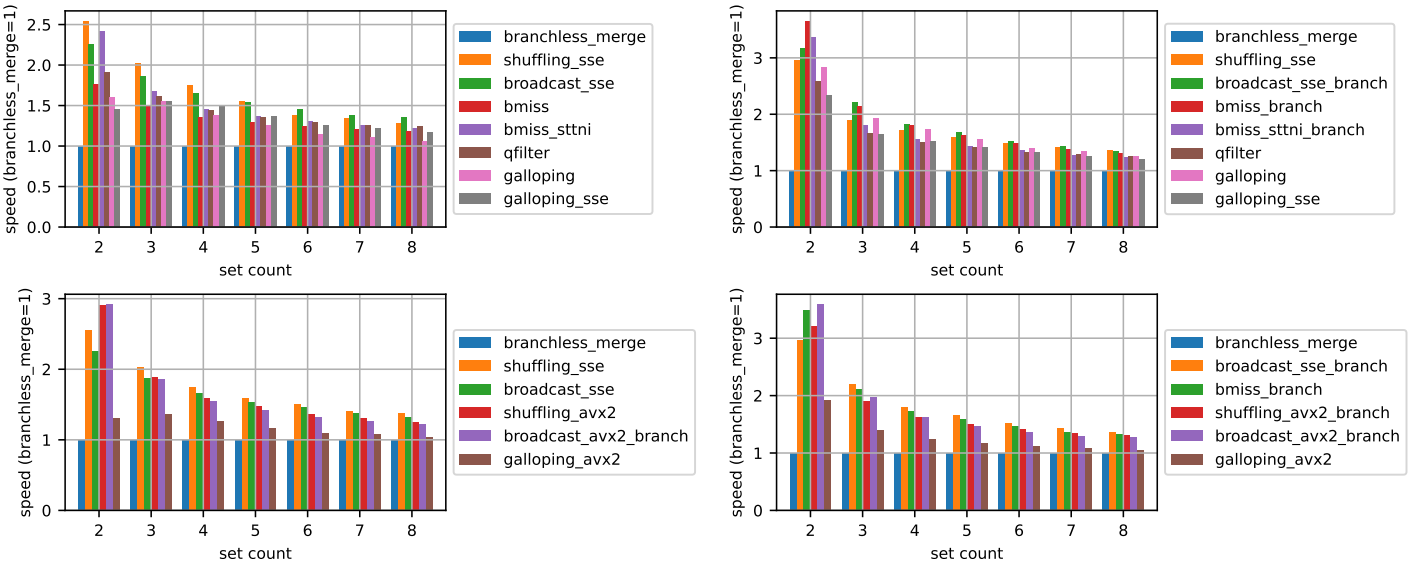
Figure 5.14: FESIA varying skew (2-set intersection, selectivity 1%, density 0.1%)

5.7 Results on WebDocs

To see how these results carry to real-world data, we now benchmark on WebDocs: and inverted index dataset. As described in Section 4.4.4, although the underlying sets are real, the queries (groups of sets) are randomly sampled for each set count. The average selectivities for each set count are shown in Table 5.2. To estimate the density, we first estimate the size of the element space by finding the smallest and largest value in each set. We then divide each set size by the element space size and average these results, resulting in an estimated density of 0.015%. Finally, we calculate the average skew of a 2-set intersection as approximately 1:10.

Table 5.2: WebDocs average selectivity for each set count

Set count	2	3	4	5	6	7	8
Avg. Selectivity	15.9%	6.31%	4.11%	3.10%	2.18%	1.86%	1.51%



(a) Intel i5 7600k (b) Intel Xeon Gold 6342

Figure 5.15: WebDocs varying set count (SSE upper, AVX2 lower) (inverted index dataset, synthetic queries)

On both the Xeon and 7600k, we introduce the array-based algorithms by vector width, followed by FESIA, Roaring and Small Adaptive. First in Figure 5.15, out of the SSE array-based algorithms, $Shuffling_{\text{SSE}}$ and $Broadcast_{\text{SSE}}$ perform well on the 7600k, and $Broadcast_{\text{SSE}}$ and *BMiss* perform well on the Xeon. Moving to AVX2, both *Shuffling* variants perform well on the 7600k and both *Broadcast* variants perform well on the Xeon. SIMD Galloping does not perform well as the sets only have a skew of 1:10. Interestingly, in Figure 5.16, introducing the AVX-512 algorithms on the Xeon processor does not lead to any performance gains. It is hypothesised that the increased vector width benefits skewed intersection less, favouring lower width algorithms which have lower read overhead.

When we include Roaring, FESIA and Small Adaptive in Figure 5.17, the results are more interesting. Roaring is significantly faster than the other algorithms for 2 and 3-set intersection. This contradicts the dataset's estimated low skew, however, this could be caused by clustering in the dataset (local regions of high density). Another finding is that FESIA scales fairly well to higher set counts, performing the best from 3 to 6-set intersection on the Xeon processor. This makes sense as FESIA has a specialised k-set

variant which intersects all sets concurrently. Finally, surprisingly, we find that the scalar algorithm Small Adaptive outperforms the others for the highest of set counts (7 and 8 sets). This is again due to how the algorithm intersects all sets concurrently. This motivates further comparison of adaptive algorithms against the SIMD algorithms covered in this experiment.

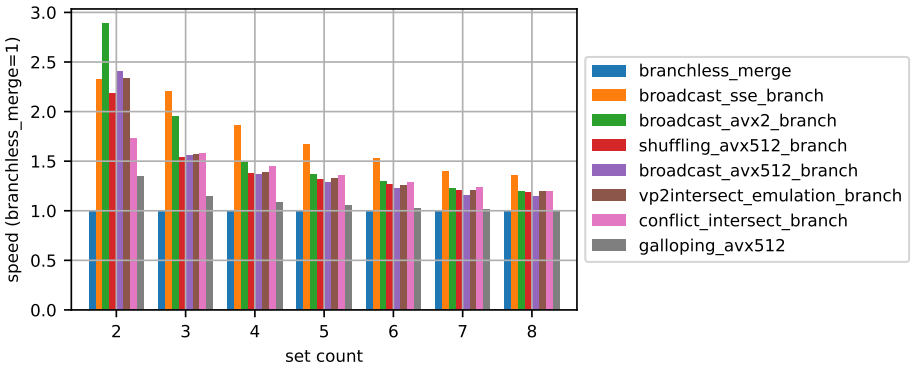
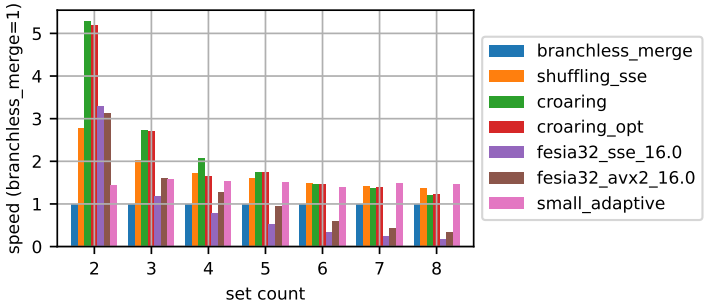
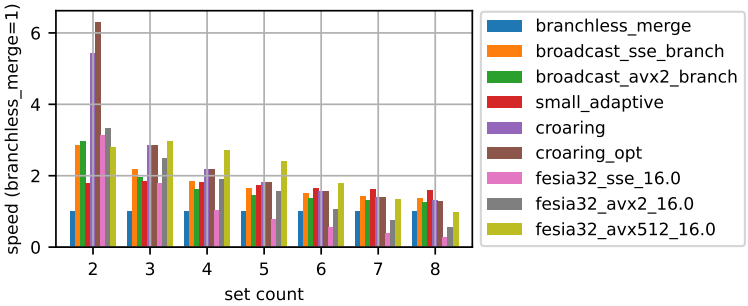


Figure 5.16: WebDocs varying set count (AVX-512, Xeon)



(a) Intel i5 7600k



(b) Intel Xeon Gold 6342

Figure 5.17: WebDocs varying set count (incl. FESIA, Roaring & Small Adaptive)

5.8 Key Takeaways

The differing results we see between the Xeon and 7600k processor across the various datasets highlights how dependent the performance of these algorithms is on the underlying hardware and data. We also see that strong performance with synthetic data does not necessarily correlate to strong performance on real data. First, this is highlighted through iteration method, where branching has a significant, inconsistent performance impact on the varying algorithms and hardware. Next, we see this through the *Broadcast* and *Shuffling* variants, where *Broadcast* performs better on the AVX-512 Xeon than it does on the 7600k. These inconsistencies necessitate open-source algorithms and experiments to allow these benchmarks to be run in their target environment.

We have also shown that extensions to *Shuffling* and *Broadcast* often outperform lower vector widths, but not in all cases. In particular, the new *Broadcast* variant scales better to higher vector widths, especially for AVX-512 hardware and BSR algorithms. We have also seen how competitive the new AVX-512 version of Roaring can be. The *CRoaring* library significantly outperforms the BSR algorithms at densities greater than 1%, and also performs well on the real dataset WebDocs.

Next, we found that our implementation of FESIA is likely not optimal, however this raises questions as to the efficacy of this algorithm. In its current closed-source state, its performance cannot be verified restricting its use in real-world projects.

Finally, through experiments on WebDocs, we found that adaptive algorithms such as Small Adaptive may be competitive, however further analysis is required.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

If there is anything this thesis has shown, it is that the space of set intersection is very complicated. There exist dozens of different algorithms, each targeting a subset of use cases and datasets, many operating on unique data structures. Some of these algorithms, namely the adaptive algorithms, aim to reduce the number of comparisons, while others tackle modern hardware features such as SIMD and branch prediction. These algorithms, both within their respective categories and between categories, have not been thoroughly compared. This is due in large part to the closed-off nature of algorithms and experiments.

As a solution to these issues, we have presented an open-source benchmark suite. This suite exposes a set intersection library including various untested algorithms and extensions. It works with the portable and open nature of the Rust ecosystem while side-stepping the performance pitfalls introduced in the name of safety. The library comes packaged with a custom benchmark suite which provides a declarative interface to define datasets and experiments. These datasets can be built ahead of time with multiple CPU cores, then benchmarks can be run and plotted with a simple command. We have included a number of additional tools to both test algorithms and verify the

datasets adhere to predefined characteristics. In presenting this as an open-source toolkit, we hope this will open the door to faster algorithms, more comprehensive and reproducible experiments, and further industry adoption.

In addition to presenting this open-source benchmark suite, we have bridged various gaps through our updated experimental analysis. We have shown how critical it is to benchmark on the target hardware and dataset. We have analysed the inconsistent impact branching has on various algorithms' performance on different CPUs. As one of our extensions, we presented a simple SIMD algorithm which uses *broadcast* instructions to perform an all-pairs comparison. It is observed that this algorithm scales better to higher SIMD widths, especially for AVX-512 hardware and BSR algorithms. In general, a deeper analysis is required to fully understand the nuances of branching, shuffling versus broadcast, and the use of mask registers. At the most basic level, we have shown which of these variants heuristically perform the best on different hardware.

Our experiments have also shown that the AVX-512 version of Roaring is very competitive for densities as low as 1%, beating out the BSR algorithms. We are unable to verify how optimal our implementation of FESIA is due to the closed-source nature of the algorithm. Finally, through our experiments on WebDocs, we have seen that adaptive algorithms such as *Small Adaptive* may compete with modern SIMD algorithms at higher set counts. This motivates the inclusion of both adaptive algorithms and other scalar algorithms like *RanGroupScan* in further research.

6.2 Future Work

As discussed, we have made some compromises in our experiment to reduce the scope of this thesis. Although better than existing experiments, our benchmark suite is still not comprehensive, and we leave some open questions.

1. We only provided a heuristic-based analysis of the affect of branching on SIMD set intersection algorithms. An assembly-level analysis is required to fully understand

how this interacts with the various load and shuffle operations on different CPUs at different SIMD widths.

2. We also only provide a heuristic-based analysis of the impact that shuffling versus broadcasting has on performance. A detailed analysis of the various instruction latencies, execution units and dependencies is required to explain the differing behaviour found with each SIMD width and CPU.

3. QFilter did not perform as expected in our experiments. To investigate why, the original open-source experiment should be run in our updated environment. If this does not explain the performance difference, an assembly-level comparison can be performed to ensure our implementation is equivalent to the original.

4. Some comparisons with scalar algorithms are yet to be made. For instance, SIMD algorithms are yet to be compared against adaptive algorithms, or scalar algorithms which operate on an alternative data structure (e.g., *RanGroupScan* and *Lookup*).

5. We only benchmark these algorithms on one real dataset. An experiment is yet to benchmark set intersection algorithms on real database workloads. Including a wider array of real datasets with database workloads would strengthen experimental results. It would also benefit to use real queries over synthetic ones.

6. Finally, it would be beneficial to compare these algorithms against solutions from similar areas of research. For example, solutions which pre-process an entire database of sets [CP10], and GPU-based set intersection [GMB12].

Bibliography

[Bar03] Jérémy Barbay. Optimality of randomized algorithms for the intersection problem. In *Stochastic Algorithms: Foundations and Applications: Second International Symposium, SAGA 2003, Hatfield, UK, September 22-23, 2003. Proceedings 2*, pages 26–38. Springer, 2003.

[BK02] Jérémy Barbay and Claire Kenyon. Adaptive intersection and t-threshold problems. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 390–399, USA, 2002. Society for Industrial and Applied Mathematics.

[BLOL06] Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. Faster adaptive set intersections for text searching. In *Experimental Algorithms: 5th International Workshop, WEA 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006. Proceedings 5*, pages 146–157. Springer, 2006.

[BLOLS10] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. An experimental investigation of set intersection algorithms for text searching. *ACM J. Exp. Algorithmics*, 14, Jan 2010. `doi:10.1145/1498698.1564507`.

[BPP07] Philip Bille, Anna Pagh, and Rasmus Pagh. Fast evaluation of union-intersection expressions. In *Algorithms and Computation: 18th International Symposium, ISAAC 2007, Sendai, Japan, December 17-19, 2007. Proceedings 18*, pages 739–750. Springer, 2007.

[BRM98] Guy E Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 16–26, 1998.

[BY75] Jon Louis Bentley and Andrew Chi-Chih Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(SLAC-PUB-1679), Nov 1975.

[BY04] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Annual Symposium on Combinatorial Pattern Matching*, 2004.

[BYS05] Ricardo Baeza-Yates and Alejandro Salinger. Experimental analysis of a fast intersection algorithm for sorted sequences. In Mariano Consens and

Gonzalo Navarro, editors, *String Processing and Information Retrieval*, pages 13–24, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[CLKG16] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.

[CM10] J Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)*, 29(1):1–25, 2010.

[CP10] Hagai Cohen and Ely Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40-42):3795–3800, 2010.

[CS16] Yangjun Chen and Weixin Shen. An efficient method to evaluate intersections on big data sets. *Theoretical Computer Science*, 647:1–21, 2016.

[DC21] Guille Díez-Cañas. Faster-than-native alternatives for x86 VP2INTERSECT instructions. *arXiv preprint arXiv:2112.06342*, 2021.

[DK11] Bolin Ding and Arnd Christian König. Fast set intersection in memory. *arXiv preprint arXiv:1103.2409*, 2011.

[DLOIM01] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Experiments on adaptive set intersections for text retrieval systems. In *Algorithm Engineering and Experimentation: Third International Workshop, ALENEX 2001 Washington, DC, USA, January 5–6, 2001 Revised Papers 3*, pages 91–104. Springer, 2001.

[DLOM00] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 743–752, 2000.

[GMB12] Oded Green, Robert McColl, and David A. Bader. GPU merge path: A GPU merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 331–340, New York, NY, USA, 2012. Association for Computing Machinery. `doi:10.1145/2304576.2304621`.

[HL72] Frank K. Hwang and Shen Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[HZY18] Shuo Han, Lei Zou, and Jeffrey Xu Yu. Speeding up set intersections in graph algorithms using SIMD instructions. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1587–1602, New York, NY, USA, 2018. Association for Computing Machinery. `doi:10.1145/3183713.3196924`.

[IOT14] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with SIMD instructions by reducing branch mispredictions. *Proceedings of the VLDB Endowment*, 8(3):293–304, 2014.

[Kan11] David Kanter. Intel Sandy Bridge microarchitecture, September 2011. URL: `https://www.realworldtech.com/sandy-bridge/6/`.

[Kat12] Ilya Katsov. Fast intersection of sorted lists using SSE instructions, Jun 2012. URL: `https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/`.

[KP99] Jagannath Keshava and Vladimir Pentkovski. Pentium III processor implementation tradeoffs. *Intel Technology Journal*, 3(2):1–11, 1999.

[KW20] Tsvi Kopelowitz and Virginia Vassilevska Williams. Towards optimal set-disjointness and set-intersection data structures. *Leibniz International Proceedings in Informatics (LIPIcs)*, 168, 2020.

[LBK14] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. SIMD compression and the intersection of sorted integers. *CoRR*, abs/1401.6399, 2014. URL: `http://arxiv.org/abs/1401.6399`, `arXiv:1401.6399`.

[Lem23] Daniel Lemire. CRoaring version 1.0.0, Mar 2023. URL: `https://github.com/RoaringBitmap/CRoaring/releases/tag/v1.0.0`.

[LKK+18] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, 2018.

[LOPS04] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. WebDocs: a real-life huge transactional dataset. In *FIMI*, volume 126, 2004.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*, chapter Faster postings list intersection via skip pointers. Cambridge University Press, USA, 2008.

[Rei13] James R Reinders. Intel® AVX-512 instructions, July 2013. URL: `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html`.

[ST07] Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 71–83. SIAM, 2007.

[SWL11] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. Fast sorted-set intersection using SIMD instructions. *ADMS@ VLDB*, 1(8), 2011.

[SYL17] Xingshen Song, Yuexiang Yang, and Xiaoyong Li. SIMD-based multiple sets intersection with dual-scale search algorithm. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, pages 2311–2314, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3132847.3133082.

[WLPS17] Jianguo Wang, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 993–1008, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3035918.3064007.

[Zip36] George Kingsley Zipf. *The psycho-biology of language: An introduction to dynamic philology.* London: Routledge, 1936.

[ZLSF20] Jiyuan Zhang, Yi Lu, Daniele G. Spampinato, and Franz Franchetti. FESIA: A fast and SIMD-efficient set intersection approach on modern CPUs. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1465–1476, 2020. doi:10.1109/ICDE48307.2020.00130.

Appendix

A.1 Algorithms and Experiments

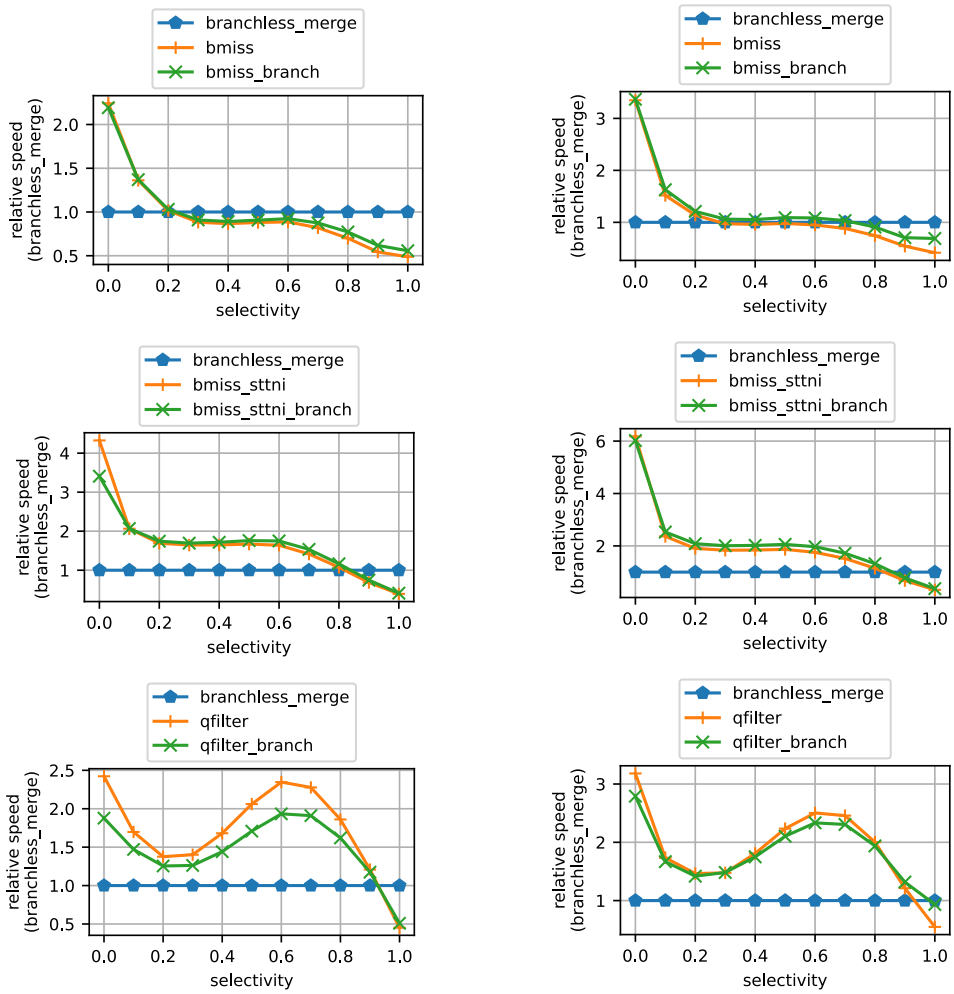
Table A.1: Algorithms

Adaptive	[DLOM00]
SvS	[DLOIM01]
Small Adaptive	[DLOIM01]
Sequential	[BK02]
RSequential	[Bar03]
BaezaYates	[BY04]
Extrapolation	[BLOL06]
Skipper	[ST07]
Lookup	[ST07]
Max	[CM10]
RanGroupScan	[DK11]
Hierarchical	[SWL11]
Shuffling	[Kat12]
V1	[LBK14]
V3	[LBK14]
SIMD Galloping	[LBK14]
BMiss	[IOT14]
Roaring	[CLKG16, LKK$^+$18]
QFilter[BSR]	[HZY18]
FESIA	[ZLSF20]

Table A.2: Experiments

Small Adaptive	[DLOIM01]
BaezaYates	[BY04]
Extrapolate	[BLOL06]
Lookup	[ST07]
ExpText	[BYS05]
Max	[CM10]
RanGroupScan	[DK11]
Hierarchical	[SWL11]
Shuffling	[Kat12]
SIMD Galloping	[LBK14]
BMiss	[IOT14]
QFilter[BSR]	[HZY18]
FESIA	[ZLSF20]

A.2 Branch vs. Branchless Plots



(a) i5 7600k (b) Xeon Gold 6342

Figure A.1: Branch vs. Branchless (continued)

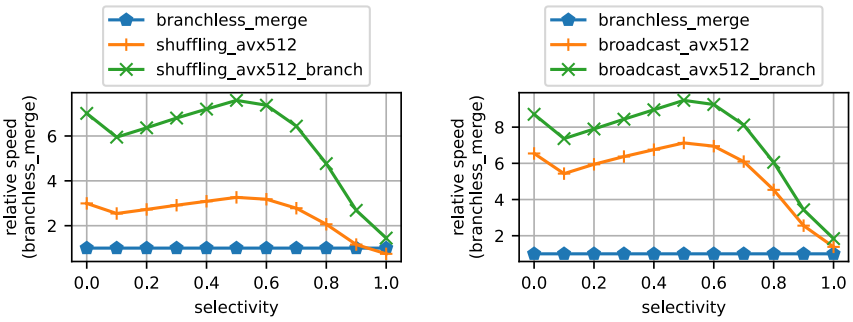
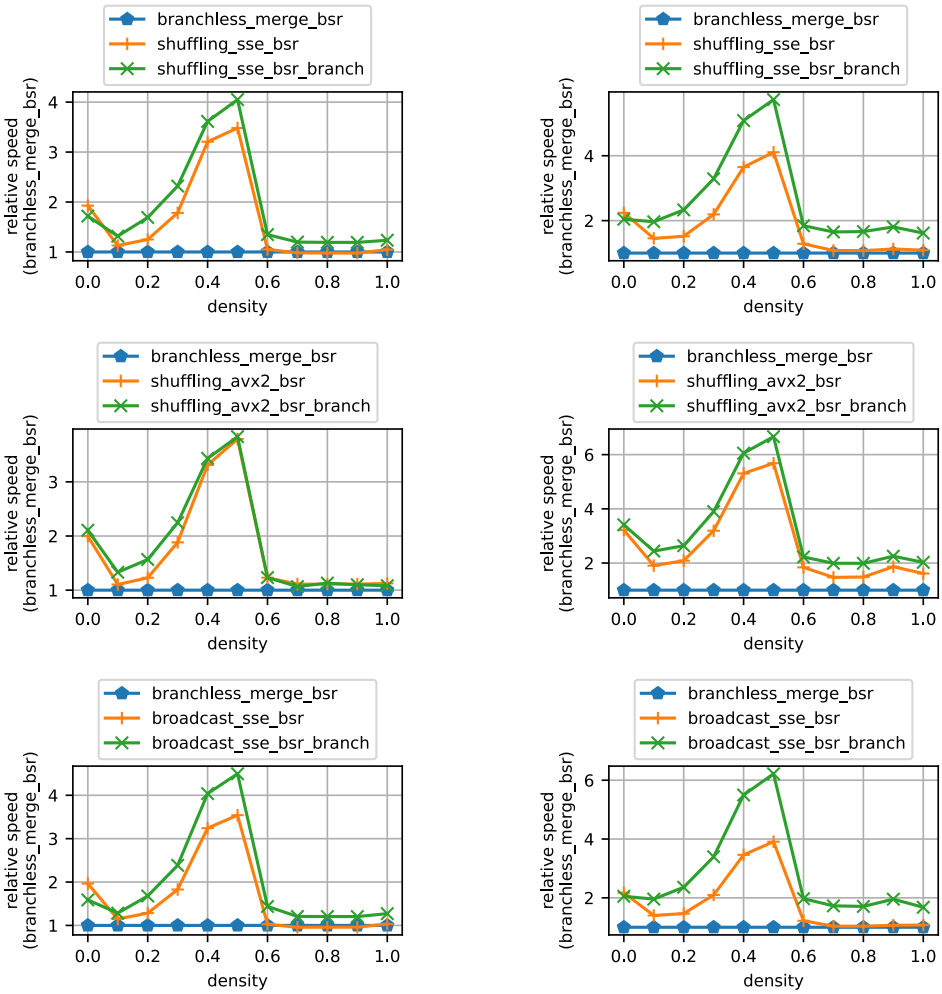
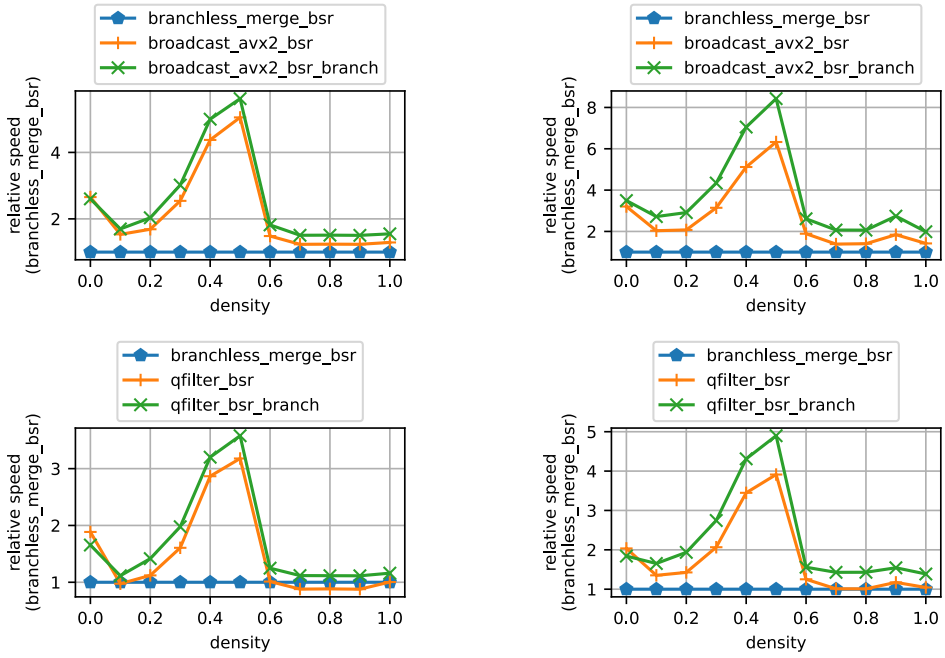


Figure A.2: Branch vs. Branchless (Xeon Gold 6342) (continued)



(a) i5 7600k (b) Xeon Gold 6342

Figure A.3: Branch vs. Branchless (continued)



(a) i5 7600k (b) Xeon Gold 6342

Figure A.4: Branch vs. Branchless (continued)

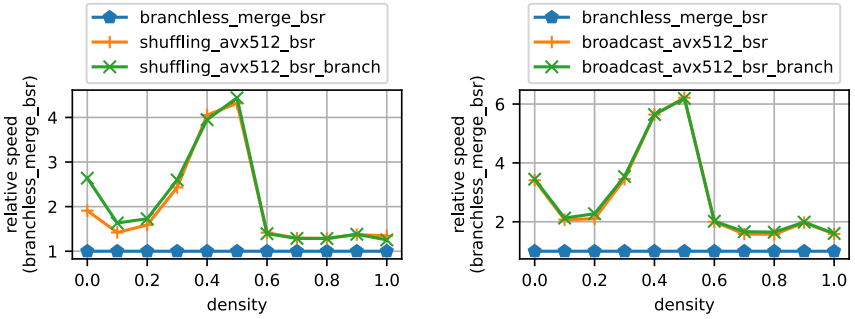


Figure A.5: Branch vs. Branchless (Xeon Gold 6342) (continued)