Integrating Skips and Bitvectors for List Intersection

by

Andrew Kane

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

(Andrew Kane)

Abstract

This thesis examines space-time optimizations of in-memory search engines. Search engines can answer queries quickly, but this is accomplished using significant resources in the form of multiple machines running concurrently. Improving the performance of search engines means reducing the resource costs, such as hardware, energy, and cooling. These saved resources can then be used to improve the effectiveness of the search engine or provide additional added value to the system.

We improve the space-time performance for search engines in the context of in-memory conjunctive intersection of ordered document identifier lists. We show that reordering of document identifiers can produce dense regions in these lists, where *bitvectors* can be used to improve the efficiency of conjunctive list intersection. Since the process of list intersection is a fundamental building block and a major performance bottleneck for search engines, this work will be important for all search engine researchers and developers. Our results are presented in three stages.

First, we show how to combine multiple existing techniques for list intersection to improve space-time performance. We combine *bitvectors* for large lists with *skips* over compressed values for the other lists. When the *skips* are large and overlaid on the compressed lists, space-time performance is superior to existing techniques, such as using *skips* or *bitvectors* separately.

Second, we show that grouping documents by size and ordering by URL within groups combines the skewed clustering that results from document size ordering with the tight clustering that results from URL ordering. We propose a new *semi-bitvector* data structure that encodes the front of a list, including groups with large documents, as a *bitvector* and the rest of the list as *skips* over compressed values. This combination produces significant space-time performance gains on top of the gains from the first stage.

Third, we show how partitioning by document size into separate indexes can also produce high density regions that can be exploited by *bitvectors*, resulting in benefits similar to grouping by document size within one index. This partitioning technique requires no modification of the intersection algorithms, and it is therefore broadly applicable. We further show that any of our partitioning approaches can be combined with *semi-bitvectors* and grouping within each partition to effectively exploit skewed clustering and tight clustering in our dataset. A hierarchy of partitioning approaches may be required to exploit clustering in very large document collections.

Acknowledgements

I would like to thank my supervisor, Frank Wm. Tompa, for his patience, guidance and constructive feedback over these many years. His tireless dedication is both incredible and inspirational. I also thank the remaining members of my committee, Alejandro López-Ortiz, Charles L. A. Clarke, Mark D. Smucker, and Alistair Moffat, for taking the time to review and improve this work. I am thankful to Güneş Aluç and Jeffery Pound for their feedback on my harebrained ideas over the years.

Many researchers have been generous in providing their implementations of intersection related algorithms. In particular, I would like to thank J. Shane Culpepper for providing his *hybrid bitvector* code [Culpepper 10], Alejandro López-Ortiz for providing his uncompressed intersection code [Barbay 09], and the researchers at WestLab, Polytechnic Institute of NYU for providing their block based compression code [Yan 09, Zhang 08].

Preliminary versions of the work presented in this thesis have appeared in the following publications:

- Andrew Kane and Frank Wm. Tompa. Distribution by document size. *In the Workshop on Large Scale and Distributed Systems for Information Retrieval (LSDS-IR)*, 2014.
- Andrew Kane and Frank Wm. Tompa. Skewed partial bitvectors for list intersection. *In Proceedings of the 37th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 263-272. ACM, 2014.

I would like to thank the anonymous reviewers of those papers (and previous submissions) for their constructive feedback.

This research was supported by the University of Waterloo, the Mprime Network of Centres of Excellence, Open Text Corporation, and the Natural Sciences, and Engineering Research Council of Canada.

Motivation

I am motivated by the need to understand how things work. The best way to really ensure you understand how a computer works is to optimize an application running on it. As such, this thesis is a direct demonstration of my understanding of computer systems through the optimization of an in-memory search system.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

A search engine is a system that answers information needs in the form of queries over free form data. For quick query answering over large datasets, the data is preprocessed (offline with respect to the queries) to produce an index. This index is spread across sufficient hardware resources for the system to execute at the desired query rate (throughput) and the desired query runtime (latency). In order to produce good throughput and latency (efficiency) while still producing relevant answers (effectiveness), most search engines restrict the queries to be a string of keywords or phrases, the processing to act on words (tokens), and the answers to be a small, ordered list of data objects that are subsequently displayed to the user.

Modern search engines often store their indexes in the random access memory of their machines for much better runtime performance than if the indexes were stored on disk. In order to scale to large systems in memory, many machines may be required. Distributing the queries among these machines and merging the query results is fast and simple, because the amount of data transferred is small and the data is separable (i.e., a search engine is an "embarrassingly parallel" [Moler 86] system). As a result, a search engine will be fast if it is given enough machines. Nevertheless, improvements to efficiency, such as those examined in this thesis, can reduce the often significant resource costs of a search engine.

To answer a query, the search index provides for each query term a list of documents containing that term, and the engine then intersects and ranks those lists. List intersection and ranking are the major performance bottlenecks for search engines that use large amounts of resources, such as Web search, so any improvement in efficiency in these areas translates into significant cost savings. The work presented in this thesis improves the efficiency of list intersection, then describes how these improvements can be applied to ranking based systems. For a list intersection system these efficiency improvements have no effect on the contents of the result lists, thus giving a definite performance improvement rather than an efficiency-effectiveness tradeoff. Applying these efficiency improvements to

a ranking based system, however, may affect the ranking function and thus change the effectiveness of the search system.

In this thesis, we examine the space-time performance of algorithms that perform in-memory intersection of ordered integer lists. These algorithms are used in search engines, where the integers are document identifiers, and in databases, where the integers are row identifiers. In both cases, the integers map to identifiers that are assigned by the system. We assume that the lists are stored in integer order, which allows for fast merging and for compression using deltas. List indexes (or auxiliary indexes) can be added to improve random access by skipping parts of the compressed lists, with list index values either overlaying or being extracted from the compressed form. If the domain of integers is compact, then *bitvectors* can be used to store some or all of the lists, resulting in compression for large lists. These approaches can be combined in various ways to produce a tradeoff of space vs. time. For example, we can try to make the search faster by adding more *skips* to the list indexes, or we can use more space to store more lists as *bitvectors*. We can also reorder the assignment of identifiers to improve the performance of these systems.

Culpepper and Moffat [Culpepper 10] showed that combining *vbyte* (variable byte) compression [Williams 99] for small lists with *bitvectors* for large lists (*vbyte+bitvectors*) is better, in both space and time, than adding list indexes that allow skipping within the *vbyte* compressed lists (*vbyte+skips*). This raises some fundamental questions that we answer in this thesis:

1. Are *bitvectors* better than *skips* for all compression algorithms?
2. Since *skips* work better with document reordering, do *bitvectors* remain better than *skips* under document reordering?
3. Are there situations in which it is best to use *bitvectors* and *skips* together?
4. Should *bitvectors* be used for subsections of individual lists?

In order to quickly process a query in a large search engine, the processing must be done in parallel across many machines. In practice, this means splitting the dataset into partitions (or shards), which contain non-overlapping subsets of the documents. This process is referred to as document distribution, and usually the documents are randomly placed into the partitions for good load balancing.

Each partition runs the query in parallel, after which the top-k results from each are merged to produce the final query results. The partitions do not need to communicate with each other to process the query, and the amount of communication needed to broadcast the query and merge the results is small, so the speedup from document distribution is nearly linear in the number of partitions. Other techniques, such as term distribution, are not

usable in many search systems because of their difficulties with scalability, load imbalance, and network saturation [Büttcher 10, §14.1.2].

Our experiments use data and queries from the search engine domain, where the lists of integers are usually encoded using *bitvectors* or compressed delta encodings such as variable byte (*vbyte*), PForDelta (*PFD*) [Zukowski 06], Simple-9 (*S9*) [Anh 05a] or Simple-16 (*S16*) [Zhang 08]. List indexes are often added to the compressed values for better runtime performance. We also examine some intersection algorithms that are slow or require the lists be encoded using large amounts of memory, which gives context to the performance characteristics of the other algorithms.

The integers in our lists are document identifiers assigned by the search engine. Typically, these values are assigned based on the order in which the documents are indexed. Reordering the document identifiers can produce both space and runtime benefits, so many approaches to reordering have been examined, such as sorting by document size, clustering by content, applying TSP, sorting by URL, and hybrid orderings. A detailed review of such reordering techniques is presented in Section 2.7. We focus on the original order (*orig*), random ordering (*rand*), terms-in-document ordering (*td*), and URL ordering. Other orderings have performance within the range represented by these four orderings, since URL ordering has been shown to give comparable performance to the best of the other approaches [Silvestri 07].

We examine space and time performance using the TREC *GOV2* dataset and queries, since it is the standard benchmark when considering efficiency-based analysis in the search engine domain.

Our first contributions improve the performance the existing approaches to list intersection:

- We show that *bitvectors* are better than *skips* for more advanced and compact compression algorithms (under the original document order), but the space improvements when using *bitvectors* are less significant than they are when *vbyte* compression is used (Chapter 4).
- We show how document reordering affects the index space and query runtime for list intersection, rather than just the time for list encoding and decoding (Chapter 4).
- We show that *bitvectors* are not always better than *skips* under the URL ordering, in particular when using the more compact compression algorithms. *Bitvectors* still produce space-time performance benefits under URL ordering as compared to the original document order, but *skips* can produce larger runtime gains and then run faster for some space budgets (Chapter 4).

- We find that combining *bitvectors* with large overlaid *skips* results in significant runtime improvements, for all compression algorithms and all orderings, as compared to *bitvectors* or *skips* alone (Chapter 4).

Previous studies have not shown that *bitvectors* are effective when combined with compression algorithms other than *vbyte*, used with document reordering, or combined with *skips*. (The implementation details of the *skips* algorithm can substantially affect the performance of combining *bitvectors* and *skips*, thus explaining why previous results "got no additional gain" from the combination [Moffat 07a].) Because our results are consistent across several compression algorithms, we believe that they will also hold for others not examined here.

Our remaining contributions involve new techniques that improve upon existing approaches to list intersection:

- We introduce a new hybrid ordering approach that groups by terms-in-document size in descending order, and then sorts by URL ordering within each group, resulting in improvements with respect to document clustering and delta compression (Chapter 5).
- To improve the usage of *bitvectors*, we introduce a type of *partial bitvector* called a *semi-bitvector*, and then combine it with our grouped ordering. This data structure allows the front portion of a list to be a *bitvector* while the rest uses normal compression and *skips*, with cutoff points aligned to the group boundaries of our hybrid ordering (Chapter 5).
- To improve the usage of *bitvectors* and data clustering similar to grouping above, we partition by document size. This is broadly usable, since any form of query execution (including ranking) can be used within the partitions. Load balancing in this approach requires manipulation of partition cutoff points (Chapter 6).
- We show how *bitvectors* can be used in a ranking based search engine (Chapter 7).

Neither grouping nor partitioning by document size has previously been examined in the context of search engines or other research domains. We find that grouping or partitioning by document size, combined with appropriate data structures, can produce significant improvements in space and runtime for list intersection on top of the benefits we have shown by tuning the performance of existing techniques. (When compared to algorithms using *skips* or *bitvectors* alone, which is the norm for many search engines, the improvement is very large.) In addition, we believe that similar types of improvement are possible for more general search systems involving ranking and in other research domains such as database systems.

We present related work in Chapter 2, experimental setup in Chapter 3, and *compression+skips+bitvectors* with reordering in Chapter 4. We then consider grouping and *semi-bitvectors* in Chapter 5 and partitioning by document size in Chapter 6, and we draw conclusions in Chapter 7.

Chapter 2

Related Work

In this chapter, we present related work on indexing, searching, ranking, distribution, and efficiency for search systems. While there are many details involved in implementing an efficient search system, we focus on the methods of list intersection, the various list encodings, and the effects of document reordering on the performance of the search algorithms. For a more detailed background on these topics, we recommend the survey of search engine techniques written by Zobel and Moffat [Zobel 06], as well as various textbooks [Baeza-Yates 11, Büttcher 10, Croft 10, Witten 99].

2.1 Indexing

Search engines execute on datasets that are too large to be evaluated by scanning through the data at query time. Instead, the datasets are preprocessed to create indexes that allow fast query processing. Standard text indexing deals with a dataset as a set of data objects (henceforth referred to as documents) consisting of a sequence of words (tokens). The first stage of the indexing process thus involves retrieving the documents, converting them into text, and then splitting the text into sequences of tokens. Even directly processing the dataset as tokens would be too slow, so the tokenized form is converted into an new form that improves the query performance. To create this new form the documents are given

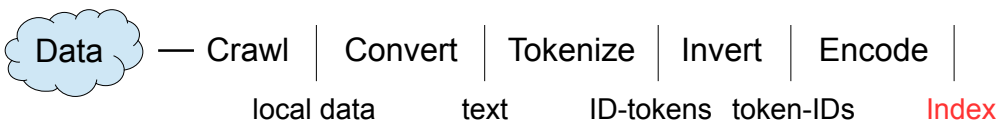

Figure 2.1: Search engine indexing pipeline.

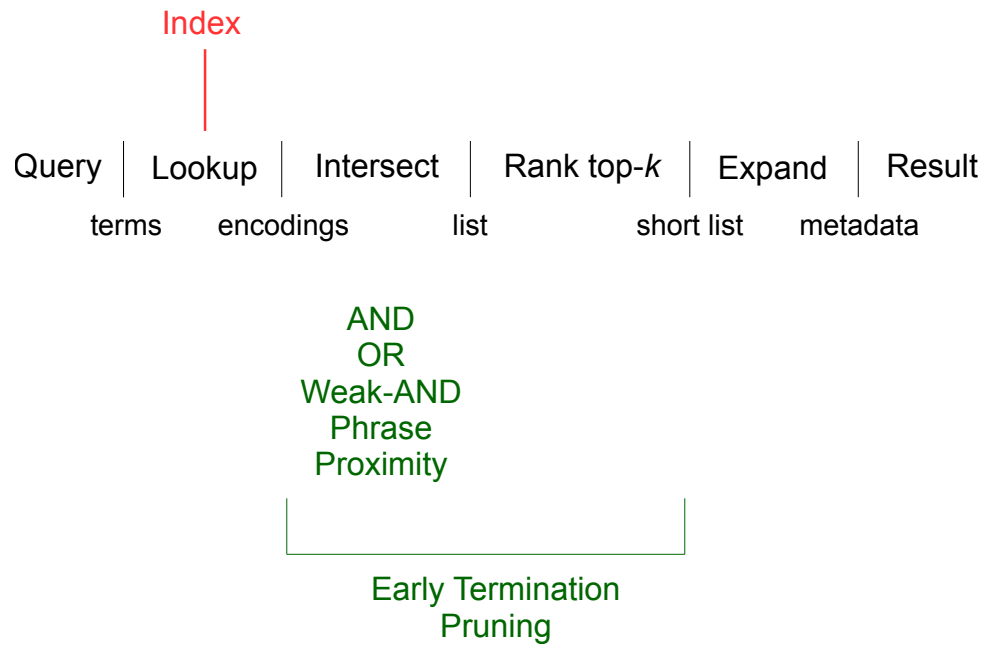

Figure 2.2: Search engine query processing.

identifiers (IDs) and the mapping from ID to token sequence is then inverted to produce for each token a list of IDs called a *postings list*, where the IDs in each list refer to the specific documents (and sometimes their positions, or offsets, in the documents) containing the corresponding token. In addition, a *dictionary* (or lexicon) is produced to map from a token to the corresponding postings list. The data structure comprising the dictionary and the postings lists is referred to as an *inverted index*. The index is then encoded for more efficient storage and processing. This entire process of indexing the data can be done in parallel so that it completes in a reasonable amount of time. This indexing pipeline is summarized in Figure 2.1. The format of the resulting index is:

$$\text{Index: } [\text{ token} \rightarrow \text{<ID, frequency, offsets>}^{+}]^{*}$$

2.2 Query Processing

Queries are issued to the search engine as simple text strings that must be tokenized to produce an ordered list of query terms (or tokens) with associated operations indicating how to combine the terms together. The terms are looked up in the dictionary portion of the inverted index, which contains the encoded postings lists with the document identifiers

of the documents containing those terms (and offsets into the documents). These encoded lists are then intersected using the operators specified in the query, or the default operators for the search engine, to produce a list of query results (i.e., the lists and operators act as a filter over the documents in the dataset to produce a candidate result list). A small number of relevant query results (top-k) are chosen from the candidate list using a ranking algorithm. These few document identifiers are the query results that are expanded using a separate data structure to produce the document metadata, such as the URLs and snippets of the documents, required to produce the final query results that are displayed to the end user. This query processing pipeline is summarized in Figure 2.2.

The intersection and ranking steps of the query processing pipeline are the most time-consuming parts of a large search system. As a result, search systems implement many optimizations of the intersection and ranking steps to produce runtime performance benefits. For example, interleaving these two steps can remove the duplicate memory accesses required in a two pass approach.

If the intersection step can quickly reduce the candidate list, then the ranking of the candidates step may be fast. If the intersection step cannot quickly reduce the candidate set, then other optimizations are needed to pare down the candidate list, such as assuming a stronger intersection operation or including ranking restrictions to reduce the candidate list. These optimizations may be *safe-to-rank-k*, where the top-k results are guaranteed to be the same, or *approximations*, where the top-k results are considered to be good enough.

Combining information about the maximum scores achievable from the terms in a query with the current top-k ranking threshold can allow portions of the postings lists to be *pruned* from the processing.

The *max-score* pruning approach [Turtle 95] calculates the maximum possible score from each query term, orders the query terms by size and determines how many of the largest query terms can be combined to produce a score guaranteed to be below the top-k threshold. Documents containing only these largest query terms can then be pruned from the query processing, because they cannot appear in the top-k results. In addition, a document can be pruned if the remaining query terms to be included in its score cannot increase that score above the top-k threshold. Separating the top scoring documents in each term [Strohman 05] can allow for additional optimizations to the *max-score* approach, since combining the top scoring documents can quickly produce a good top-k threshold and removing the top scoring outliers can reduce the maximum possible score from the rest of the list.

The *Weak-AND* (*WAND*) pruning approach [Broder 03] does more dynamic processing to produce document pruning. The query terms become iterators that are sorted by their current document identifier, then the maximum scores from these query terms can be summed in order until reaching the top-k threshold, which identifies the pivot term. The

document identifiers from query terms before the pivot term's current document identifier cannot produce a score above the top-k threshold, so the earlier query terms are advanced to the pivot's current document identifier. This ordering and advancing process is repeated until enough terms have the same current document identifier which is scored and added to the top-k if the score is high enough. The ordering and advancing process then repeats starting at the next document identifier. Advancing the query terms in this way allows effective pruning of documents, but requires the added overhead of sorting query terms and calculating pivots.

The *block-max WAND* pruning approach [Ding 11] stores the maximum possible score for blocks of consecutive values in postings lists. This approach allows more granular calculations of maximum score, but taking into account the block sizes makes the *block-max WAND* algorithm more complicated. In addition, the size and location of the blocks used when calculating the maximum scores within each postings list must be tuned to choose an appropriate point within the space-time tradeoff.

If the postings lists are ordered to align with the ranking scores, it may be possible to avoid processing the ends of the lists if they are unlikely to produce high enough scores, thus allowing *early termination* of the query. This early termination approach can be implemented at a high level by splitting the documents into *tiers* based on their potential ranking scores. The tiers can then be split onto separate machines allowing separate scaling per tier, or implemented dynamically within each machine. The processing within each tier can use any combination of pruning, early termination, or ordered list merging, though the performance of these techniques may be affected by the characteristics of the data found in each tier.

The *max-score*, *WAND*, *block-max WAND* and early termination approaches can all be implemented as safe-to-rank-k optimizations. For extremely high volume systems, more aggressive optimizations that are approximations of the top-k ranking may be valuable. Although the approaches we have already presented can be improved if they produce only approximations of the top-k ranking, other approaches may be more appropriate.

Defaulting queries to use the AND operator (*default-AND*) allows the intersection step to quickly reduce the candidate set, but the top-k results only approximate full ranking since they will not contain documents with fewer query terms. If the *default-AND* query does not return enough results, this smaller result list can be returned to the user or the query can be re-executed using another approach. *Default-AND* can also be used with tiers, where queries are only executed on lower tiers when there are not enough results returned by the higher ones.

For search engines that contain large amounts of data, such as Web search engines, the *default-AND* approach may produce effective results and be highly efficient. The average number of terms per query is often low for a Web search engine, while the number of

documents is large, so many documents in the engine are likely to contain all the query terms and these documents will be highly ranked, thus dominating the top-k results. Such large search engines can, therefore, answer many queries by intersecting the query term postings lists (using an AND operator) to produce candidates that are then ranked to produce the top-k. Indeed, this approach was used in the first version of Google's Web search engine [Brin 98], where they "scan through the doclists until there is a document that matches all the search terms" and then compute the rank. It has been acknowledged that the *default-AND* approach is commonly used in Web search systems, and may even improve the ranking effectiveness of small queries [Croft 10, p. 172]. Clearly, the efficiency of both the intersection and ranking steps are important for such systems. In fact, the efficiency of conjunctive AND intersection that we are examining in this thesis is of particular importance for these large search systems.

2.3 Functions used for Ranking

Search results are ranked by a function that uses information from the query lists and the overall index to produce a score for a document and a given query. Ranking functions can then be compared, using various metrics based on the order of the scored documents and external opinions on whether the results are relevant. The metrics used to compare ranking algorithms treat the results (the full list or the top-k) of each query as a set, such as with precision, recall, F-measure, and mean average precision (MAP) [Büttcher 10, §2.3], or as an ordered list, such as with mean reciprocal rank (MRR) using the first correct answer [Kantor 96] and discounted cumulative gain (DCG) using all the correct answers [Järvelin 02]. Ranking metrics can also take into account models of the user, as with expected reciprocal rank (ERR) [Chapelle 09], rank-biased precision (RBP) [Moffat 08], and time-biased gain (TBG) [Smucker 12].

Traditionally, ranking functions were static and designed by the search engine developers to produce good results in a reasonable amount of time (e.g., Okapi BM25 [Robertson 95]). Alternatively, a *learning-to-rank* approach can be applied on a large number of features extracted from the documents. This approach uses machine learning techniques, such as gradient descent [Burges 05] or support vector machines (SVMs) [Cao 06], to decide which features are best and how they should be combined. The search engine developers can then tune the system to approximate this combination within a reasonable query runtime.

Multiple ranking functions can be combined in a cascading approach [Wang 11], where fast ranking functions are used to produce a small number of candidate results that are passed to slower, but more effective, ranking functions that re-rank or prune out candidates. This cascading approach can effectively combine the two ranking approaches described above. For example, a fast developer-designed ranking function could produce a candidate

set of results, which are then re-ranked using a feature based ranking function created using machine learning techniques.

2.4 Distributed Search Systems

A search system may be distributed across multiple machines in order to support fault tolerance, scale to a large dataset, or meet query latency and throughput specifications. Query throughput requirements can be met by simply replicating the search engine, and extra replication can also be used to support fault tolerance. Scaling to a large dataset and reducing query latency can be achieved to some extent using high performance hardware, such as large servers or disk systems, but it is much more cost effective to distribute the inverted index across multiple commodity machines, disks, and processors. We refer to these units of inverted index distribution as *partitions* (or *shards*).

The inverted index can be split across multiple partitions using *document distribution*, in which each partition contains a subset of the documents forming a separate sub-index. Queries are broadcast to the partitions, each running in parallel, and the query results from all partitions are merged to produce the final top-k set of query results. Distributing the queries to the partitions and sending back the results requires some network communication, but the message sizes are small. However, any overhead incurred per query term can be expensive, since the overhead applies at each partition. On the other hand, running the partitions in parallel produces low query latency, while the independence of partitions and ease of result merging allows for near linear scaling. As a result, document distribution is used in commercial search engines. In order to produce good load balancing, the documents are usually distributed randomly.

Some versions of document distribution will first cluster the dataset's documents by topic [Kulkarni 10] or top-k result counts [Puppin 06] and then place these clusters into separate partitions. The queries can then be executed on a subset of the partitions and still produce relevant query results. This process is called *selective search* (or *collection selection*), and it results in improved throughput and reduced resource use. Unfortunately, cluster based partitioning with selective search has the potential to degrade ranking effectiveness, which limits its use. These non-random document distribution approaches can also be used without selective search [Feldman 11, Lavee 11], but load balancing is difficult.

As an alternative approach, the inverted index can be split across partitions using *term distribution*, meaning that each term's postings list is placed in a single partition. To process a query, all the postings lists of the query are sent to a single broker machine that intersects the lists and produces the top-k query results. Transferring the postings lists across the network adds latency and can saturate the network, but per query term overheads occur only once. Processing the entire query at a single broker can produce

large query latencies, and not having enough brokers can limit performance. In order to produce good load balancing, the postings lists are usually distributed randomly.

In order to reduce network traffic and lighten the load of the broker when processing queries in a term distributed index, the *pipelined* approach [Moffat 07b] passes partially evaluated query results among partitions. The queries are partially evaluated at each partition containing a postings list of the query, with the partitions ordered by the lowest term frequency of their query terms. Each partition folds its lists into the partial results that are subsequently passed to the next partition, thus preventing full document-at-a-time query processing (see Section 2.5). Partial evaluation involves all the query lists at each partition before processing moves on to the next partition, meaning the lists are not always combined in term frequency order. This pipelined approach does not produce good load balancing, but the load imbalance can be somewhat improved by controlling the distribution of lists into partitions or by placing some lists in multiple partitions [Moffat 06].

Hybrid distribution approaches can combine the benefits, or alleviate the drawbacks, of the standard approaches. One hybrid approach splits postings lists into chunks that are distributed across partitions to help balance the load [Xi 02]. Another hybrid approach uses document distribution to split the index into sub-collections, which are then split into partitions using term distribution, thus reducing the per query term overhead of document distribution and limiting the network communication of term distribution [Kane 09].

For extremely large datasets, the data may be split into tiers [Risvik 03] based on some global order such as PageRank or on past query results. As mentioned earlier, lower tiers are queried only if higher tiers do not return sufficiently good results. Each tier is typically independent and uses document distribution, but the characteristics of the data in each tier could vary significantly from the characteristics of the full dataset. Examining how changes in the per-tier data characteristics affect the choice of query execution technique is beyond the scope of this thesis.

Term distribution and pipelined evaluation may be valuable for systems with large overheads per query term, such as disk based systems with large random access latencies, but document distribution performs better in most systems. For our in-memory search system, the per query term overheads are small, so we use document distribution in our experiments.

2.5 Intersection Approaches

This thesis focusses primarily on the *intersect* step depicted in Figure 2.2, and more specifically conjunctive AND intersection. As explained earlier, using conjunctive intersection gives approximate ranking of results since it does not guarantee the best top-k results, but

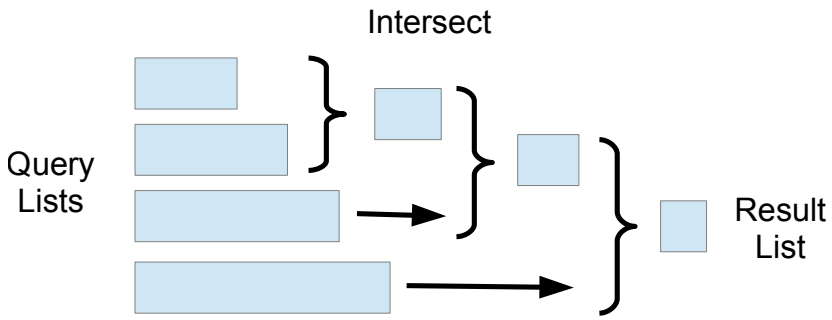

Figure 2.3: Set versus set conjunctive intersection of multiple lists.

this approximation may still be effective and highly efficient for large search systems, such as Web search engines.

We have limited our query execution to conjunctive list intersection without ranking. In databases, this is an equality join on the lists, which can be executed using various methods. If the lists are in sorted order, they can be joined using a merge algorithm, otherwise, additional temporary memory must be used to join the lists [Shapiro 86]. For search engines, the values stored in the temporary join structure are referred to as accumulators, since they are used to accumulate the ranking scores for candidate results.

Intersecting multiple lists can be implemented by intersecting the two smallest lists, then intersecting the result with the next smallest list iteratively, thus producing a *set versus set* (*svs*) or *term-at-a-time* (*TAAT*) [Turtle 95] approach. This *svs* approach for conjunctive list intersection is shown in Figure 2.3. For ordered lists, each step of the *svs* approach uses the *merge* or *MG* algorithm as defined in Algorithm 1, which takes each element in the smaller list M and finds it in the larger list N by executing a forward search, then reports any elements that are found. Various implementations of the forward search subroutine are possible, but the choice of implementation is restricted by the encoding approach used for N. The M list could be encoded differently than the N list, and indeed, after the first *svs* step it is the uncompressed result list of the previous step. The sequential processing and memory accesses of the *svs* approach allows the compiler and CPU to improve the execution, making this approach extremely fast, even though temporary memory is required for intermediate result accumulators. Although these intermediate results are needed in our experiments, they are simply document identifiers since we are not ranking the results. In addition, we avoid the need for join structures by using sorted lists.

To reduce the temporary memory used in the *TAAT* approach, while still maintaining some of the compiler and CPU optimization ability, processing can be split into batches. In such a *batch-at-a-time* approach, the document identifiers are split into batches with the intersection for all terms being processed for one batch before the next batch is processed.

Algorithm 1 *MG* – Intersect Merge

```
1: function MG(M,N)                                          ▷ |M| ≤ |N|
2:     i ← N.iterator(), n ← 0, r ← {}
3:     for all m ∈ M do
4:         n ← i.forwardSearch(m)                            ▷ iterator dependent
5:         if m = n then
6:             r ← r ∪ {m}
7:     return r                                              ▷ list
```

The number of accumulators is, therefore, bounded by the batch size. The batches being processed can contain a fixed number of document identifiers or a range of the document identifier domain. To improve efficiency, the list encoding and list indexes can be aligned with these fixed range batches.

Intersecting multiple lists can also be implemented by intersecting all the lists at the same time. If the lists are not in sorted order, using this approach may require a large amount of temporary space for the join structures and the accumulators. If the lists are sorted, then we call this a *document-at-a-time* (*DAAT*) [Turtle 95] approach, and it requires very little temporary space: just one pointer per list to keep track of its processing location. The order that the lists are intersected could be static, such as ascending term frequency order (as done with *svs*), or it could adapt to the processing. Such adaptive methods are not as fast for real data [Barbay 09] and are not explored here. All of these non-*svs* approaches jump among the lists that are stored at different memory locations, so it is more difficult for the compiler and CPU to improve the execution performance. In addition, the loop iterating over the lists for each result item and the complications when using different list encodings (i.e., the need for dynamic calls, extra checks of list types, or multiple implementations) will slow down non-*svs* implementations. Despite these limitations, many of the optimizations that we demonstrate in this thesis using *svs* list intersection can also be applied to implementations using non-*svs* approaches.

For systems that return the top ranked results, a small amount of additional memory is needed for a top-k heap to keep track of the best results. Instead of adding results that match all the terms into a simple array of results, they are added to the heap. At the end of query processing, the content of the heap is output in rank order to form the final top-k query results.

2.6 List Encodings

We present detailed related work for list intersection and illustrate various algorithms' performance using space-time graphs, for which the experimental setup uses the *GOV2* corpus and query workload as described in Chapter 3. Where possible we have used the authors' original code; otherwise we coded the algorithms to follow as closely as possible their descriptions in the literature. When presenting these algorithms, we specify the name of the intersection algorithm followed by the encoding algorithm and the document identifier ordering as *intersection(encoding)(ordering)*. For example, SK(Vbyte$_1$skip$_X$)(orig) represents the *SK* algorithm with an encoding of *vbyte* values accessed one element at a time and *skips* over a range of X values using the original document ordering.

2.6.1 Uncompressed Lists

Storing the lists of document identifiers in an *uncompressed* format simply means using a sequential array of integers. For fast intersection, the integers in these lists are stored in order, allowing merging of lists without additional temporary memory storage, and also allowing the use of many methods to search for a particular value in a list (i.e., the implementation of the forward search subroutine in Algorithm 1). As a result, there are many fast algorithms available for intersecting uncompressed integer lists (e.g., storing the integers using 32 bits/posting, *U32*, in the native machine word format plus the list length value), but the memory used to store the uncompressed lists is very large and probes into the list can produce wasted or inefficient memory access. All of these uncompressed intersection algorithms rely on random access into the lists, so they are inappropriate for compressed lists. We present only the three best algorithms for our *GOV2* dataset [Barbay 09]:

Galloping *svs* (*MG-gallop*): Galloping forward search [Bentley 76] probes into the list to find a point past the desired value, where the probe distance doubles each time, then the desired location is found using binary search within the last two probe points. The galloping *svs* algorithm, therefore, fits our definition of merge in Algorithm 1 (page 15) with galloping used to implement the forward search subroutine.

Galloping swapping *svs* (*MGSW-gallop*): In the previous galloping *svs* algorithm, values from the smaller list are found in the larger list. Galloping swapping *svs* [Demaine 01], however, finds values from the list with the smaller number of *remaining* integers in the other list, thus potentially swapping the roles of the lists. The galloping swapping *svs* algorithm does not fit the pseudo-code of our merge algorithm, but does execute using a *set-versus-set* approach.

Sorted Baeza-Yates using adaptive binary forward search (*SBY-ab*): The Baeza-Yates algorithm [Baeza-Yates 04] is a divide and conquer approach that finds the median

value of the smaller list in the larger list, splits the lists, and recurses. Adding matching values at the end of the recursion [Barbay 09] produces a sorted result list. The adaptive binary forward search variant uses binary search within the recursed list boundaries when searching in the larger list, rather than using the original list boundaries. The sorted Baeza-Yates algorithm does not fit the pseudo-code of our *merge* algorithm, but does execute using a *set-versus-set* approach.

2.6.2 Compressed Lists

There are a large variety of compression algorithms available for sorted integer lists. The lists are first converted into differences minus one (i.e., deltas or d-gaps) to get smaller values, but this removes the ability to randomly access elements in the list (thus limiting the possible implementations of the forward search subroutine in Algorithm 1). Next, a variable length encoding is used to reduce the number of bits needed to store the values, often grouping multiple values together to allow word or byte alignment of the groups and fast bulk decoding. We examine a variety of compression algorithms below:

Golomb: The *Golomb* algorithm [Golomb 66] uses a tuneable parameter k to split a value x into a quotient $q = \lfloor x/k \rfloor$ and a remainder $r = x - q \cdot k$. The value is encoded as q in unary followed by a binary representation of r. The pivot point $p = 2^{\lfloor \log_2 k \rfloor + 1} - k$ dictates how the remainder is encoded, by using either $\lfloor \log_2 k \rfloor$ bits (from r when $r < p$) or $\lceil \log_2 k \rceil$ bits (from $r + p$ when $r \geq p$). Since the *Golomb* algorithm can encode zero, the values are often reduced by one when encoding, which we have already presented as using deltas minus one.

As an example of *Golomb* encoding, given $k = 100$ meaning $p = 28$, the value 898 (assuming it is already a delta minus one) results in a quotient $q = 8$ and remainder $r = 98$. The remainder is larger than p, so it is encoded using the binary representation of $r + p = 126$ using 7 bits. Combining the unary encoding of the quotient with the binary encoding of the remainder gives the *Golomb* encoding as 00000000 1*1111110*, with the binary remainder marked in italics.

Rice: The *Rice* algorithm [Rice 71] is a restricted version of the *Golomb* algorithm, where the tuneable parameter is of the form $k = 2^m$. Requiring that k be a power of two means the calculations for encoding and decoding are easy, since the quotient and remainder are simply ranges of bits from the original input value (i.e., the lower m bits of the input are r and the higher bits are q). As with the *Golomb* algorithm, the *Rice* algorithm can encode zero, allowing the use of deltas minus one.

As an example of *Rice* encoding, given $m = 7$ and thus $k = 128$, the value 898 (assuming it is already a delta minus one) results in a quotient $q = 7$ and remainder $r = 2$. Combining

the unary encoding of the quotient with the binary encoding of the remainder gives the *Rice* encoding as 0000000 1*0000010*, with the binary remainder marked in italics.

Elias *gamma*: The *gamma* algorithm [Elias 75] stores a value x by encoding it in binary up to its last significant one bit (comprising b bits), then prepends $b-1$ (i.e., $\lfloor \log_2 x \rfloor$) zeros before the binary value. This encoding can also be produced using the unary value of b followed by the lower $b-1$ bits of the binary encoding of the number. This encoding can store only positive integers, so we must use deltas rather than deltas minus one.

As an example of *gamma* encoding, the value 898 contains 10 significant bits, so we prepend 9 zeros giving the *gamma* encoding as 000 00000*11 10000010*, with the binary representation marked in italics.

Elias *delta*: The *Elias delta* algorithm [Elias 75] stores a value x by measuring the size b of its binary encoding, then encoding b using *gamma* encoding and appending the $b-1$ lower bits of the binary encoding. Again, this encoding can store only positive integers, so we must use deltas rather than deltas minus one.

As an example of *Elias delta* encoding, the value 898 (binary encoding 11 10000010) contains 10 significant bits, so we encode 10 using *gamma* encoding (i.e., 0001010) followed by the lower 9 bits of the binary encoding, giving the *Elias delta* encoding as 0001010*1 10000010*, with the lower bits of the binary representation marked in italics.

Variable byte (*vbyte*): The *vbyte* algorithm [Williams 99] stores deltas using a variable length sequence of bytes. This encoding is byte aligned with each byte storing seven bits of a delta and using the remaining bit to indicate whether storing the delta also requires the next byte (i.e., the first bit indicates the layout of the data bits). As a result, a positive integer x can be stored using $\lfloor \log_{128} x \rfloor + 1$ bytes, so that numbers from 0 to 127 use one byte, 128 to 16383 use two bytes, and so on. Even though byte aligned operations are fast, there are many if statements in the code, which can slow down execution on modern CPUs. The *vbyte* algorithm has been shown to be faster than many non-byte aligned approaches [Trotman 03], but this speed is achieved at the expense of compression.

As an example of *vbyte* encoding, the value 898 as a 32-bit number is 00000000 00000000 00000011 10000010, but this value can be stored with two bytes using the *vbyte* encoding as **1**0000111 **0**0000010, where the layout bits are marked in bold.

Simple-9 (*S9*): The *S9* algorithm [Anh 05a] is word aligned (32 bits), using four bits of each word to indicate how the remaining 28 bits are allocated to fit a few deltas, allowing a switch statement on the four layout bits to be used for fast decoding. The same number of bits are used for each delta in a word, giving nine possible combinations for 28 bits, though some have unused bits, as shown in Table 2.1. The *S9* algorithm uses a variable block size (i.e., the number of deltas that fit in the word), but we combine these words into larger blocks having a fixed number of deltas [Zhang 08] for a clearer comparison with *PFD* (see below). The Simple-9 approach of using 4 bits per word for layout can

be extended to 64 bit words for additional performance improvements, as done with the Simple-8b algorithm [Anh 10].

As an example of *S9* encoding, we consider the input containing the value 898 followed by a series of 1s. The value 898 requires at least ten bits, so rule 8 can be used to store the first two values as **0111***0000 11100000 10*0000000 00000001, with layout bits marked in bold and the value 898 marked in italics.

Simple-16 (*S16*): The *S16* algorithm [Zhang 08] is an extension of *S9* where different numbers of bits can be used for different values stored in one word. This version has sixteen possible combinations and none have unused bits, as shown in Table 2.2. As before, multiple words are encoded into larger blocks.

As an example of *S16* encoding, we consider the input containing value 898 followed by a series of 1s. The value 898 requires at least ten bits, so rule 14 can be used to store the first three values as **1101***1110 00001010*0 00000010 00000001, with layout bits marked in bold and the value 898 marked in italics.

Frame of Reference (*FOR*): The *FOR* algorithm [Goldstein 98] stores a block of values sequentially in an array using the same number of bits for each value (i.e., a bit array). If the values have minimum l and maximum h, then the values are stored relative to l using $\lfloor \log_2(h-l) \rfloor + 1$ bits for each value. By using blocks containing multiples of 32 values, the resulting encoding is ensured to be 32 bit word aligned. The *FOR* algorithm is fast for decoding, but even a few large values can restrict the resulting compression.

As an example of *FOR* encoding, we consider the value 898 followed by three 1s. In this batch of four values, the minimum is $l = 1$ and maximum is $h = 898$, meaning the block can be stored with each value using ten bits. The batch of four values (898,1,1,1) are encoded relative to l (897,0,0,0) using *FOR* as *11100000 01*000000 00000000 00000000 00000000, with the bits representing the value 898 marked in italics.

PForDelta (*PFD*): The patched frame of reference over deltas (PForDelta or *PFD*) algorithm [Zukowski 06] is word aligned (32 bits) and combines multiples of 32 deltas into a block (padding lists with zeros to fill out the last block). Lists with fewer than 100 elements use normal *vbyte* encoding (to avoid the overhead of padding a list with zeros to fill the last block), thus producing a hybrid algorithm. When using *PFD*, the "frame of reference" stores an array of the deltas using b bits each as described above in the *FOR* algorithm, with the value b stored in the header. To reduce the number of bits in the frame of reference, up to 10% of the deltas are stored separately as exceptions in the "patch". The exceptions are identified using a linked list by encoding the start of the list in the header and the linking in the frame of reference entries. If the exceptions are farther than 2^b values apart, then additional exceptions are added to allow for the linking. If adding of exceptions results in more than 10% of the values being stored as exceptions, then a larger value of b is used. The values of the exceptions are stored in an array or using

rule	layout (4 bits)	data ($\leq$ 28 bits)	unused (bits)
1	0000	$28\times1_b$	0
2	0001	$14\times2_b$	0
3	0010	$9\times3_b$	1
4	0011	$7\times4_b$	0
5	0100	$5\times5_b$	3
6	0101	$4\times7_b$	0
7	0110	$3\times9_b$	1
8	0111	$2\times14_b$	0
9	1000	$1\times28_b$	0

Table 2.1: Details of *S9* encoding ($28\times1_b$ means 28 values of 1 bit each).

rule	layout (4 bits)	data (28 bits)	*S9* rule
1	0000	$28\times1_b$	1
2	0001	$7\times2_b$, $14\times1_b$	
3	0010	$7\times1_b$, $7\times2_b$, $7\times1_b$	
4	0011	$14\times1_b$, $7\times2_b$	
5	0100	$14\times2_b$	2
6	0101	$1\times4_b$, $3\times8_b$	
7	0110	$1\times3_b$, $4\times4_b$, $3\times3_b$	
8	0111	$7\times4_b$	4
9	1000	$4\times5_b$, $2\times4_b$	
10	1001	$2\times4_b$, $4\times5_b$	
11	1010	$3\times6_b$, $2\times5_b$	
12	1011	$2\times5_b$, $3\times6_b$	
13	1100	$4\times7_b$	6
14	1101	$1\times10_b$, $2\times9_b$	
15	1110	$2\times14_b$	8
16	1111	$1\times28_b$	9

Table 2.2: Details of *S16* encoding ($28\times1_b$ means 28 values of 1 bit each) with equivalent *S9* rules indicated.

another compression routine. The batch size is restricted to a multiple of 32, causing the frame of reference to be word aligned. The header and patch are stored so as to ensure a word aligned encoding. The *PFD* algorithm gives a compact encoding with fast decoding because few *if* statements are used. The process of encoding using the *PFD* algorithm may, however, require trying several values of b before the number of exceptions drops below 10%.

As an example of *PFD* encoding, we consider the input containing the value 898 followed by thirty-one 1s. This block of 32 values can be stored in a frame of reference that is one bit wide ($b = 1$) with few exceptions, but representing the linked list of exceptions would require too many additional exceptions. As a result, b must be increased until $b = 4$ when fewer than 10% of the values are stored as exceptions. With this setting, the frame of reference portion is (**15**,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0, **15**,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0) with each value encoded using four bits and exceptions marked in bold. The header contains $b = 4$, the minimum value $l = 1$, and the head of the linked list of exceptions (i.e., the first entry). The exceptions section contains two values (898,1).

NewPFD: The *NewPFD* algorithm [Yan 09] is a variant of the *PFD* algorithm that stores the lower b bits of the exceptions in the frame of reference portion of the encoding. To identify and restore the exceptions, a list containing the indices of the exceptions and another list containing the higher bits of the exceptions are stored in the patch portion of the encoding, with each list stored using *S16* compression. The header contains the values b and l, but no longer contains the head of the linked list of exceptions. The *NewPFD* variant can more compactly store small values than the original *PFD* algorithm.

As an example of *NewPFD* encoding, we consider the input containing the value 898 (significant bits 11 10000010) followed by a series thirty-one 1s. This block of 32 values can be stored in a frame of reference (encoded relative to the minimum value $l = 1$) that is one bit wide ($b = 1$) with fewer than 10% exceptions, encoded as *1*0000000 00000000 00000000 00000000. The patch contains the exception index list, which is only index zero here and is encoded using *S16* as **1111**0000 00000000 00000000 00000000. The patch also contains the exception higher bits list encoded using *S16* as **1111***0000 00000000 00000001 11000000*. The layout bits are marked in bold and the bits representing the value 898 are marked in italics.

OptPFD: The *OptPFD* algorithm [Yan 09] uses *NewPFD* to store values, but extends the approach by tuning the exception rate (previously fixed at 10% for each block of values) over the entire list rather than for each block, resulting in a range of space-time configurations and better efficiency.

VSEncoding (*VSE*): The *VSE* algorithm [Silvestri 10] is a *FOR/PFD* variant that has no exception mechanism; instead it dynamically varies the block sizes and stores deltas in each block using the same number of bits (i.e., using just a frame of reference). Such dynamic block sizes obstruct the use of *skips*, so we do not examine this approach further.

Some compression algorithms act directly on the original postings lists as monotonically increasing sequences of integers, then build the delta encoding more deeply into their implementation. To prevent decoding the entire list, these approaches can act on subsections of the postings lists (i.e., by splitting the lists into blocks). We examine two approaches to encoding such sequences:

Binary Interpolative Coding (*IPC*): The *IPC* algorithm [Moffat 00] is a divide and conquer approach that stores a monotonically increasing sequence by encoding the value at the midpoint of the list of length n, then recurses on the values above and below this midpoint (with base cases of zero and one element lists). The midpoint value is encoded using enough bits to specify it within the value range (l, h) inclusive, with the elements above and below the midpoint removed, giving the restricted value range $(l + \lceil n/2 \rceil - 1, h - n + \lceil n/2 \rceil)$. For a dataset D, the initial value range can be $(l_0, h_0) = (0, |D| - 1)$, but the list length n must be encoded to determine the restricted value range when decoding. As with the *Golomb* algorithm, when the size s of the restricted value range is not a power of two, some of the values can be encoded with $\lfloor \log_2 s \rfloor$ bits and others with $\lceil \log_2 s \rceil$ bits, but for the IPC algorithm, the values using fewer bits are placed in the middle of the range using a centred minimal binary code [Howard 93]. This binary code encodes $c = 2^{\lceil \log_2 s \rceil} - s$ elements in the centre of the range using one fewer bit.

As an example of *IPC* encoding, we consider a list containing the two elements $(800, 898)$ with $l = 0$ and $h = 999$. The midpoint element 800 is encoded in the range $(0, 998)$ giving $s = 999$ and $c = 25$ entries in the centre. Our element 800 is not in the centre, so it requires $\lceil \log_2 s \rceil = 10$ bits. The recursion below the midpoint contains no elements so nothing is output, but the recursion above the midpoint encodes 898 in the range $(801, 999)$ giving $s = 199$ and $c = 57$ entries in the centre. Our element 898 is in the centre, so it requires $\lfloor \log_2 s \rfloor = 7$ bits. Our two element list can, therefore, be stored in 17 bits using *IPC* encoding.

Elias-Fano (*EF*): The *EF* algorithm [Elias 74] stores a monotonically increasing sequence by splitting each value into upper bits stored as deltas in unary encoding, and lower bits stored in binary format. This separation and encoding is identical to the *Rice* algorithm, except that deltas are used in the upper portion and the two portions are not stored together. The encoding of the upper bits for all the values are concatenated, followed by the lower bits of all the values concatenated together in a bit array. The upper bit structure can be processed by counting the number of 1 bits. If the start of the lower bit structure is known, the lower bits of any value can be accessed directly as an array. For a list of size n with values below a high point h, this encoding uses $b = max(0, \lfloor \log_2(h/n) \rfloor)$ lower bits. For a dataset D, the high point $h = |D|$ is an obvious choice. This storage approach allows fast list intersection and compact storage [Vigna 13], as discussed briefly in Section 2.6.5.

As an example of *EF* encoding, we consider a list containing the two elements $(800, 898)$ with $|D| = 1000$ giving $b = 6$ lower bits. Our two values when split become an upper

portion $(12, 14)$ and a lower portion $(32, 2)$, with the upper portion encoded as deltas $(12, 2)$. As a result, our two element list can be encoded using *EF* as 0000 00000000 1*001* 1000 00*000010*, with the bits produced from the value 898 marked in italics.

Recent work has improved decoding and delta restore speeds for many list compression algorithms using vectorization [Lemire 13]. Additional gains are possible by changing delta encoding to act on groups of values, thus improving runtime at the expense of using more space. While such approaches can be combined with our work, we do not explore this here.

In the remainder of this thesis, we examine the most common *vbyte* algorithm and some of the top performers, in particular the *S9*, *S16*, *PFD* and *OptPFD* algorithms.

2.6.3 List Indexes

List indexes, also known as *skip* structures or auxiliary indexes, can be used to jump over values in the postings lists and thus avoid decoding, or even accessing, portions of the lists. The desired jump points can be encoded inline with the lists, but this causes more memory locations to be access when few values are being searched for in the list. Instead, it is preferable to store the jump points in a separate, contiguous memory location, often without compression, allowing fast scanning through the jump points. These list indexes can be used with compressed lists by storing the deltas of the jump points, but the block based structure causes complications if the jump point is not byte or word aligned, as well as block aligned. The actual list values that are found in the *skip* structures can either be maintained within the original compressed list (overlaid) preserving fast iteration through the list, or the values could be extracted (i.e., removed) from the original compressed lists, giving a space reduction but slower iteration through the list.

A simple list index algorithm ("skipper" [Sanders 07]) groups by a fixed number of elements (a bucket) storing every X^{th} element in a separate array structure, where X is a constant, so we refer to it as *skips(X)* and the corresponding intersection algorithm as *SK*, see Algorithm 2. When intersecting lists, the *skip* structure is scanned linearly to find the appropriate jump point into the compressed structure (lines 7-8), where the decoding can commence (line 9). Since jump points indicate a bucket or range within the encoded structure (line 10), the forward search subroutine can be restricted to this range (line 11), thus allowing for alternate implementations. Some of the costs involved in decoding and searching within each bucket can be shared between all the elements we are searching for within the bucket. As a result, the *skips* algorithm simply iterates though the smaller list, finds which bucket in the larger list might contain the current element, and then searches for the element within the bucket starting at the last decoded point in the bucket.

Using variable length *skips* is possible, perhaps tuning the number of skipped values relative to the list size n, such as using a multiple of $\sqrt{n}$ [Moffat 96] or $\log n$ [Culpepper 10].

Algorithm 2 *SK* – Intersect *Skips*

```
1: function SK(M,N)                                              ▷ |M| ≤ |N|
2:     i ← N.iterator(), n ← 0, r ← {}
3:     b ← N.bucketIterator()
4:     e ← b.endID                                               ▷ end of bucket
5:     for all m ∈ M do
6:         if m ≥ e then                                         ▷ new bucket
7:             while m ≥ b.endID do
8:                 b.next()
9:             i.jumpToBucket(b)
10:            e ← b.endID
11:        n ← i.forwardSearch(m, b.end)                         ▷ within bucket
12:        if m = n then
13:            r ← r ∪ {m}
14:    return r                                                  ▷ list
```

Algorithm 3 *SG* – Intersect *Segment*

```
1: function SG(M,N)                                              ▷ |M| ≤ |N|
2:     i ← N.iterator(), n ← 0, r ← {}
3:     b ← N.bucketIterator()
4:     e ← 1 ≪ N.segmentBits                                     ▷ end of bucket
5:     for all m ∈ M do
6:         if m ≥ e then                                         ▷ new bucket
7:             h ← m ≫ N.segmentBits
8:             b.setBucket(h)
9:             i.jumpToBucket(b)
10:            e ← (h + 1) ≪ N.segmentBits
11:        n ← i.forwardSearch(m, b.end)                         ▷ within bucket
12:        if m = n then
13:            r ← r ∪ {m}
14:    return r                                                  ▷ list
```

We found that *skips* based on $\sqrt{n}$ do not have a good space-time tradeoff, because space is spent disproportionately on small lists where skipping is less valuable and too few skip points are provided on large lists. We leave the exploration of tuning *skips* for future work.

Another type of list index algorithm ("lookup" [Sanders 07]) groups by a fixed size document identifier range by using the top level bits of the value to index into an array storing the desired location in the encoded list, similar to a segment/offset scheme. Each list can pick the number of bits in order to produce reasonable jump sizes. Here we use D as the set of indexed documents and N as the postings list, meaning the list's density is $y = |N|/|D|$. If we assume randomized data and use the parameter B to tune the system, then using $\lceil \log_2 B/y \rceil$ bottom level bits will leave between $B/2$ and B entries per segment in expectation. As a result, we call this approach *segment(B)* and the corresponding intersection algorithm *SG*, see Algorithm 3. The execution of the segment algorithm is almost identically to the *skips* algorithm, except that it defines buckets differently and then can quickly find the bucket containing the current element without scanning through the list index (lines 7-8).

To avoid non-aligned jump points in our implementations, *segment* uses only *vbyte*, and *skips* over block based encodings set the skip size X equal to the block size [Jonassen 11]. We also store the list index separate from the original list for better memory access performance. Further elaborations of our implementations are provided in Chapter 3.

With compressed lists alone, intersection is slow. On the other hand, uncompressed lists are much larger than compressed ones, but random access allows them to be fast. List indexes, when combined with the compressed lists, add little space, and their targeted access into the lists allows them to be even faster than the uncompressed algorithms. This suggests that the benefits of knowing where to probe into the list (i.e., using *skips* rather than a probing search routine) outweighs the cost of decoding the data at that probe location. Due to large space usage, we do not investigate combining *skips* and uncompressed lists, even though this combination knows where to probe into the lists and has no decoding cost.

In order to illustrate the performance of various intersection algorithms, we present space-time graphs where space is the encoded size per posting over all lists in the dataset (bits/posting) and time is the average single threaded query execution time for our workload (ms/query). We use the standard TREC *GOV2* dataset and 5,000 queries randomly selected from the related corpus query workload, as described in Chapter 3. The performance of various intersection algorithms using our main-memory index and the original document ordering is shown in Figure 2.4, where solids represent block based compressed encodings, crosses and stars represent uncompressed encodings, and hollow icons represent *vbyte* and compressed encodings with list indexes. For all these algorithm, we present only the fastest configuration that we tested.

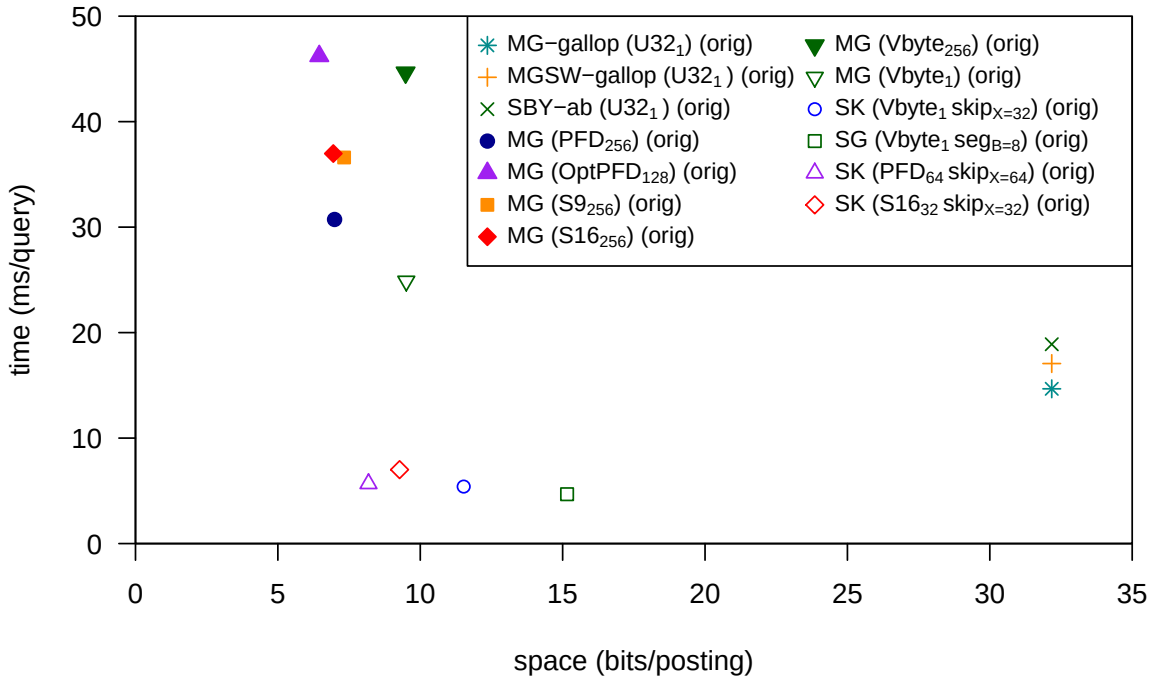

Figure 2.4: Space vs. time for various intersection algorithms executing 5,000 queries on our in-memory index containing the TREC *GOV2* dataset in the original document ordering.

2.6.4 Bitvectors

When using a compact domain of integers, as we are, the lists can instead be stored as *bitvectors*, where the bit number is the integer value and the bit is set if the integer is in the list. For runtime performance benefits, this mapping from the identifier to the bit location can be changed if the same permutation is applied to all *bitvectors*.

The intersection of two lists stored as *bitvectors* can be easily found using a bitwise AND (see *bvand* in Algorithm 4) over the entire length of the *bitvectors* to produce a new *bitvector* with the bits for the intersecting set of document identifiers. The bitwise AND operation acts on machine words (in our case, 32 bit words) using a single CPU instruction, meaning the bitwise AND operations are extremely fast, but these operations must read and write full *bitvectors*. As a result, intersecting using *bvand* may not be the most efficient when the lists are sparse.

We have defined the intersection algorithm to return a list of integers, not a *bitvector*, so the intersection result list stored as a *bitvector* must be converted to a list of integers (see *bvconvert* in Algorithm 5). There are many potential implementations of an algorithm to convert a *bitvector* into a list of integers. The *bvconvert* algorithm checks if any bits are

Algorithm 4 *bvand* – *Bitvector* AND

```
1: function bvand(M,N,s)                                   ▷ |M| ≤ |N|
2:     assert(M.isBitvector), assert(N.isBitvector)
3:     r ← new bitvector(s)
4:     for i = 0 → ⌈s/32⌉ − 1 do
5:         r[i] ← M[i] & N[i]                              ▷ 32 bit AND
6:     return r                                            ▷ bitvector
```

set in a machine word (in our case, 32 bits), then executes a linear scan of the bits in the word when needed and outputs the document identifiers represented by the bits that are set. Since our result lists are typically sparse (in our case, 24,699 results per query vs. 25.2 million documents, which is equivalent to 1 in 980 bits set), it is valuable to check each word for bits being set, but the subsequent linear scan is expensive. Instead of a linear scan, our implementation uses a logarithmic check which uses a mask to check whether each half of the bits contain any that are set, then recurses as needed. Another approach to convert a word of a *bitvector* into a list of integers is to identify a bit that is set, convert the bit to its corresponding integer, remove the bit, and then recurse. Identifying the bits in document identifier order allows the output to be streamed. Various CPU operations (ffs, ctz, clz, $\log_2$) and bitwise calculations ($w \,\&\, -w$ = least significant bit) can be used to identify the appropriate bit that is set in a machine word. For our dataset and query workload, an implementation based on linear scan was slow, but using our logarithmic check code or recursive bit identification code (from Culpepper and Moffat [Culpepper 10]) were both fast and have similar performance.

Algorithm 5 *bvconvert* – *Bitvector* Convert

```
1: function bvconvert(B,s)
2:     assert(B.isBitvector)
3:     r ← {}
4:     for i = 0 → ⌈s/32⌉ − 1 do
5:         b ← B[i]
6:         if b ≠ 0 then
7:             for k = 0 → 31 do
8:                 if b &1 k then                          ▷ bit exists
9:                     r ← r ∪ {32 · i + k}
10:    return r                                            ▷ list
```

When all the lists are stored as *bitvectors*, conjunctive list intersection can be easily implemented by combining the *bitvectors* using *bvand* and converting to a list of results in the final step using *bvconvert*, as shown in Algorithm 6. Note, except for the last step, the

result of each step is a *bitvector* rather than an uncompressed list. In our implementation, the final step combines *bvand* with *bvconvert* in a single pass of the *bitvectors*. For our dataset, encoding all the lists as *bitvectors* results in a large space usage (in our case, 25.2 million bits per list · 49 million lists / 9 billion postings = 137,200 bits per posting), but gives a very good runtime performance (3.67 ms/query using the original ordering).

Algorithm 6 *BV* – Intersect *Bitvector*

```
1: function BV(M,N,isLastStep)                          ▷ |M| ≤ |N|
2:     assert(M.isBitvector), assert(N.isBitvector)
3:     s ← N.bitvSize
4:     if isLastStep then
5:         return bvconvert(bvand(M, N, s), s)          ▷ list (see page 27)
6:     return bvand(M, N, s)                            ▷ bitvector (see page 27)
```

To alleviate the space costs of using *bitvectors*, the lists with list density less than a parameter value F can be stored using normal *vbyte* compression, resulting in a *hybrid bitvector* algorithm [Culpepper 10]. Using D as the set of indexed documents and N as a postings list, the list density is $|N|/|D|$. Note: the list density is also the raw document frequency (or the list size) $|N|$ normalized by the dataset size $|D|$.

The best *hybrid bitvector* algorithm from the original paper, referred to as "method two" or *H2*, is presented in Algorithm 8. This algorithm intersects the *vbyte* lists using merge, then intersects the remainder with the *bitvectors* by checking if the elements are contained in the first *bitvector* (see *bvcontains* in Algorithm 7), repeating this for each *bitvector* in the query, with the final remaining values being the query result.

Algorithm 7 *bvcontains* – *Bitvector* Contains

```
1: function bvcontains(M,N)                             ▷ |M| ≤ |N|
2:     assert(!M.isBitvector), assert(N.isBitvector)
3:     r ← {}
4:     for all m ∈ M do
5:         if N[m/32] &1 (m mod 32) then                ▷ bit exists
6:             r ← r ∪ {m}
7:     return r                                         ▷ list
```

The *hybrid bitvector H2* approach is summarized in Algorithm 9 using a syntactic grammar that symbolically represents how a query is processed against a sequence of term lists. Each grammar rule takes two lists, produces an intersection result, and replaces the input lists by the result. The grammar's rules are used to reduce the input set of lists to an uncompressed result list. The grammar rules process the query lists in ascending order

Algorithm 8 *H2* – Intersect *Hybrid Bitvector*

```
1: function H2(M,N,isLastStep)                          ▷ |M| ≤ |N|
2:     if !N.isBitvector then
3:         return MG(M, N)                              ▷ list (see page 15)
4:     if !M.isBitvector then
5:         return bvcontains(M, N)                      ▷ list (see page 28)
6:     return BV(M, N, isLastStep)                      ▷ list or bitvector (see page 28)
```

of their list size, giving a unique reduction. The rule to use at each step is determined by the two input lists, except when the query contains only *bitvectors* where the *bvand* rule is used until the final step which uses *bvand+bvconvert* to produce an uncompressed integer list.

Algorithm 9 Syntactic Grammar for *H2* – Intersect *Hybrid Bitvector*

(setup)	uncompressed list:	U
	small compressed list:	S
	large list as *bitvector*:	L
	query:	$\{L, S\}^*$
	processing order:	S^*L^* (smallest to largest)
(rules)	intersect:	$SS \Rightarrow U$; $US \Rightarrow U$
	bvcontains:	$SL \Rightarrow U$; $UL \Rightarrow U$
	bvand:	$LL \Rightarrow L$
	bvand+bvconvert:	$LL \Rightarrow U$

The *hybrid bitvector H2* approach fits into our *set-versus-set* query processing framework, since the rules are applied to the smallest lists first. The rules can be applied in other ways and still reduce the input set of lists to an uncompressed result list. If the *bvand* rule has higher priority than the *bvcontains* rule, then the subsequent grammar defines the "method one" or *H1* approach from the original *hybrid bitvector* paper. For clarity, examples of query executions for *H1* and *H2* are presented in Table 2.3, where the subscripts of the list variables indicate their relative size (for example, L_1 is smaller than L_2). We do not present the space-time performance of the *H1* algorithm since it is dominated by the performance of the *H2* algorithm.

While both *hybrid bitvector* algorithms are faster than non-bitvector algorithms, they are not quite as compact as some of the other compression schemes, even with the compression benefits from more compactly storing the large dense lists as *bitvectors*. Combining *bitvectors* with other compression algorithms can produce better results, as we show in Section 4.3. The performance of the original implementation of the *hybrid bitvector* algorithm,

algorithm	query type	query/reduction	operation
H1 & H2	small lists only	$S_2S_3S_1$	
		$\Rightarrow S_1S_2S_3$	sort
		$\Rightarrow US_3$	intersect
		$\Rightarrow U$	intersect
H1 & *H2*	large lists only	$L_3L_1L_2$	
		$\Rightarrow L_1L_2L_3$	sort
		$\Rightarrow LL_3$	*bvand*
		$\Rightarrow U$	*bvand+bvconvert*
H1	small and large lists	$S_3L_5L_6S_1L_4S_2$	
		$\Rightarrow S_1S_2S_3L_4L_5L_6$	sort
		$\Rightarrow US_3L_4L_5L_6$	intersect
		$\Rightarrow UL_4L_5L_6$	intersect
		$\Rightarrow ULL_6$	*bvand*
		$\Rightarrow UL$	*bvand*
		$\Rightarrow U$	*bvcontains*
H2	small and large lists	$S_3L_5L_6S_1L_4S_2$	
		$\Rightarrow S_1S_2S_3L_4L_5L_6$	sort
		$\Rightarrow US_3L_4L_5L_6$	intersect
		$\Rightarrow UL_4L_5L_6$	intersect
		$\Rightarrow UL_5L_6$	*bvcontains*
		$\Rightarrow UL_6$	*bvcontains*
		$\Rightarrow U$	*bvcontains*

Table 2.3: Examples of query executions for the *hybrid bitvector* algorithms.

H2-CM (Culpepper and Moffat), is shown in Figure 2.5, with the fastest configurations of *skips* and *segments* using various compression schemes included for comparison. A more detailed comparison is presented in Chapter 4.

2.6.5 Other Approaches

The Elias-Fano (*EF*) encoding can be used to store postings lists (i.e., quasi-succinct indices [Vigna 13]) with the lower bits of the values in an array and the higher bits as deltas using unary encoding. This structure produces a good space-time tradeoff when the number of higher level bits is limited. The list intersection implementation can exploit the unary encoding of the higher level bits by counting the number of ones in machine

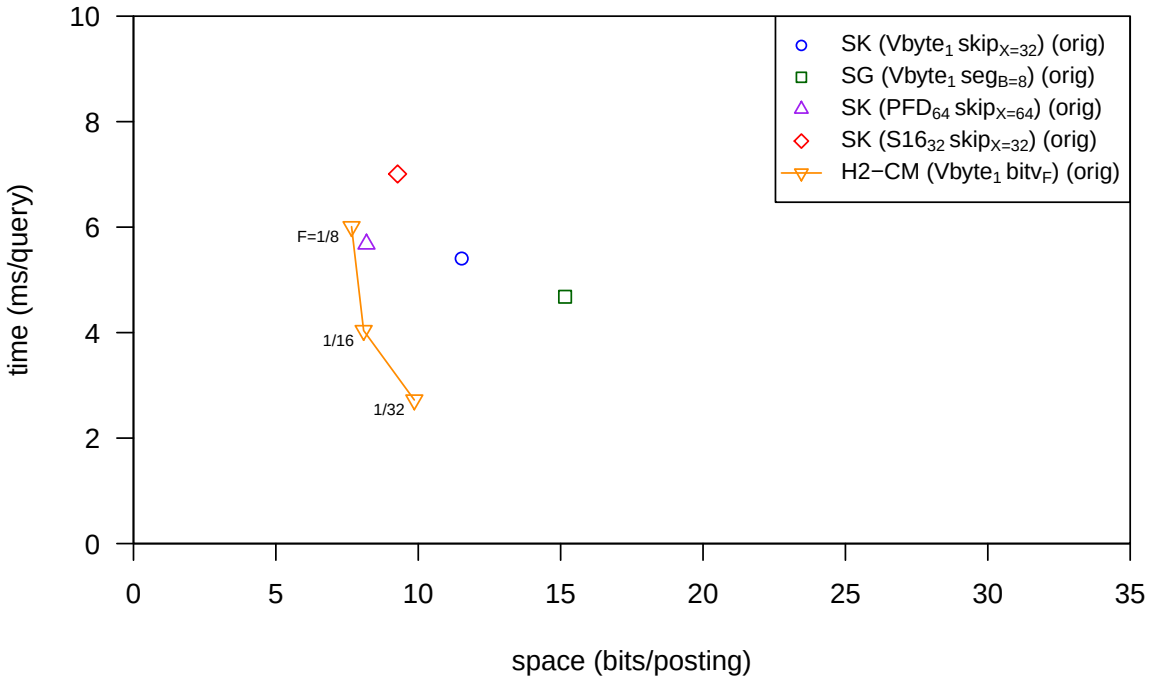

Figure 2.5: Space vs. time for *bitvectors* compared to the fastest configurations of *skips* and *segments* using original ordering. Additional algorithms are presented in Figure 2.4.

words to find values quickly. The higher bits structure can also be combined with *skips* for fast access. The EF encoding allows fast processing in the higher bits structure at the expense of additional memory accesses into the lower bits structure, however, the decoding of values in the lower bits structure is extremely fast and does not require blocks of values to be decoded. The resultant space-time performance is comparable to various *skips* implementations for conjunctive list intersection, though indexing speed may be slower.

The treap data structure [Aragon 89] combines the functionality of a binary tree and a binary heap. This treap data structure can implement list intersection [Konow 13] by storing each list as a treap where the list values are used as the tree order and the term frequencies are used as the heap order. During list intersection, subtrees can be pruned from the processing if the term frequency is too low to produce highly ranked results. In order to make this approach viable, low term frequency values are stored in separate lists using delta compressed lists with *skips*. This treap and term frequency separated index structure can produce some space-time performance improvements compared to existing ranking based search systems.

Wavelet trees [Grossi 03] can also be used to implement list intersection [Navarro 10]. The postings lists are ordered by term frequency and each value is assigned a global location, so that each list can be represented by a range of global locations. A series of bit sequences represent a tree that starts with the term frequency ordered lists of the global locations and translates them into the actual document identifiers, with each level of the tree splitting the document identifier domain ranges in half and encoding the path using the bit sequences (left as 0 and right as 1). Multiple lists can be intersected by following the translation of their document ranges in this tree of bit sequences, only following the branch if it occurs in all of the list translations. After some careful optimization of the translation code, the wavelet tree data structure results in similar space usage, but faster query runtimes than some existing methods for conjunctive queries [Konow 12].

These other approaches to list intersection could be combined with our work, but we do not explore this here.

The performance of many of the list intersection algorithms described in this chapter have been compared previously [Barbay 09, Culpepper 10, Sanders 07, Zhang 08].

2.7 Reordering

2.7.1 Reasons for Reordering

Search engines intersect lists of integers that represent document identifiers. Although these identifiers could be any unique number, in order to make the identifier deltas smaller and more compressible, they are assigned by the system to form a compact domain of values. This assignment of identifiers can be changed to produce space and/or runtime benefits in a process referred to as document *reordering*. In order to include new documents in the index or merge subindexes, the information used to order the documents must be maintained in the index. This ordering information is only used for indexing and not for querying, so the information can be stored on disk rather than in memory. Although some ordering approaches may use significant amounts of information or be slow to calculate, the orderings we consider can be quickly determined using a simple sort routine accessing limited amounts of data.

Search engines typically store their lists as compressed deltas. Reordering can improve space usage by placing documents with similar terms close together in the ordering, thus reducing the size of the deltas, which can then be stored more compactly. This reduces the space being used by the index, as well as the amount of data being accessed per query. Interestingly, reordering can improve compression in areas other than the storage of document identifiers, for example, reordering to improve the compression of term frequencies embedded in the lists [Yan 09].

Reordering can improve runtime performance by producing data clustering within the lists [Yan 09]. This gives large empty ranges in the document domain during query processing which causes list indexes, such as *skips*, to work better. These runtime improvements are seen not just with list intersection, but with systems that use term frequency, ranking, and even dynamic pruning, where knowledge of the ranking algorithm is used to avoid processing some parts of the lists [Tonellotto 11]. Tuning the compression algorithms to a particular ordering can also give better space-time tradeoffs [Yan 09].

In this thesis, reordering of document identifiers is used to improve both space and runtime, but reordering can also be used to improve runtime even if it degrades space usage. For example, reordering by a global page order [Long 03] using a ranking based measure, such as PageRank, allows early termination of the query. This can be much faster than using a non-ranking based order, since the end portions of the lists are essentially pruned from the calculation.

Each postings list could also be ordered independently based on the amount of ranking impact a document has for the associated term [Anh 05b]. Such *impact ordering* allows early termination of the query, but intersecting the lists cannot be done using a simple merge, since the lists do not share the same order. In addition, without document ordering, the impact ordered lists cannot gain runtime benefits from using list indexes such as *skips*. As a result, using impact ordered lists requires some additional memory space and runtime to execute the join on the document identifiers and accumulate the ranking scores. Document identifiers can also be reordered in a system using impact ordered lists, but there is limited compression benefit since the impact ordered lists cannot use delta encoding. A combined approach is possible, such as grouping by impact value within the lists or using tiers (separate documents in each tier) that can be pruned as needed. Both grouping and tiers can support document ordered lists within each group or tier, thus allowing for performance benefits from using delta encoding, list indexes, and merge to intersect the groups/tiers.

2.7.2 Types of Reordering

Below we present various document orders that can be applied to a search engines' postings lists:

Random: If the documents are ordered randomly (*rand*), there are no trends for the encoding schemes or the intersection algorithms to exploit. As a result, random ordering will produce a large index and slow query runtimes, but it is a good basis of comparison for the other orderings.

Original: The dataset comes in an original order (*orig*), which may be the order in which the data was crawled. This ordering could be anything, so it might not a good basis of

comparison. We have found that using the original order of the *GOV2* dataset gives some improvement over random ordering.

Rank: Reordering to approximate ranking allows the engine to terminate early when sufficiently good results are found. A global document order [Long 03], such as PageRank or result occurrence count in a training set [Garcia 04], can be used. This approach requires explicit knowledge of the ranking algorithm to decide when to terminate the query processing. Some implementations guarantee the same results as full query processing, while others provide a tradeoff between query runtime and ranking effectiveness.

Matrix: Reordering by manipulating the document vs. term matrix can produce improvements in space by grouping together documents with high frequency terms [Shi 12], producing a block diagonal matrix [Baykan 08], or creating run-length encodable portions of the matrix [Arroyuelo 13]. Manipulating the matrix for large datasets can be expensive, and merging subindexes can be difficult, so these techniques have not been widely used.

Document Size: The simple method of ordering documents by decreasing number of unique terms in the document (terms-in-document or *td*) has been shown to produce index compression [Büttcher 10, §6.3.7] and some runtime improvements [Tonellotto 11], while requiring no additional information about the documents and very little processing at indexing time. Ordering by terms-in-document is approximately the same as ordering by the number of tokens in the document or by the document size. The improvements obtained from terms-in-document ordering are not as large as those that occur with other orderings, so it has been mostly ignored.

Content Similarity: Ordering by content similarity uses some similarity metric to produce an order. Ordering using normal content clustering techniques [Blandford 02] or through solving a travelling salesman problem (TSP) formulation [Shieh 03] can produce space improvements. However, even with various improvements [Blanco 06, Ding 10, Silvestri 04a, Silvestri 04b], these approaches are too slow to be used in practice. In addition, these techniques must start from scratch when two subindexes are merged, although not for subindex compaction.

Metadata: Ordering lexicographically by URL provides similar improvements in space usage as obtained from ordering by content similarity [Silvestri 07], and it improves runtime substantially when using *skips* [Yan 09]. The URL ordering is exploiting the human-based organization of website authors, which will often group the documents by topic or application, to produce content similarity in the ordering. The effectiveness of metadata ordering can vary greatly based on the dataset and density distribution of the data within the chosen domain. This approach is simple to compute at indexing time and can support fast merging of subindexes. In addition, ordering using metadata other than the URL can make this technique applicable beyond Web indexes.

Hybrid: Ordering by terms-in-document is not as effective as ordering by URL, but these

two can be combined to get slightly better results. For example, one hybrid approach groups the documents by URL server name, then subdivides each into five document size ranges, and finally orders by URL within each subdivided group [Ding 10]. This approach is similar to what we present in Section 5.2, but the reasoning and final result are quite different.

2.7.3 Improvements from Reordering

A summary of the papers on document ordering within search systems is shown in Table 2.4, where the last two columns indicate if the paper examines the *space* or *time* benefits from ordering the documents.

reference	type	data collection and size	space	time
[Blandford 02]	clustering	TREC4-5(1GB); WT2g(2GB)	Y	N
[Long 03]	global page ordering	Web crawl 2002(1.2TB)	N	Y
[Shieh 03]	TSP	PC(small); FBIS(470MB); LATimes(475MB)	Y	N
[Garcia 04]	query results	WT10g(10GB)	N	Y
[Silvestri 04a, Silvestri 04b]	clustering	GPC(2GB)	Y	N
[Blanco 06]	SVD·TSP·clustering	FBIS(470MB); LATimes(475MB)	Y	N
[Silvestri 07]	URL; URL·clustering	WBR99(22GB)	Y	N
[Baykan 08]	block-diagonal matrix	GPC-plus(3GB)	Y	N
[Yan 09]	URL·local tweeks	GOV2(426GB)	Y	Y
[Büttcher 10, §6.3.7]	doc terms; URL	GOV2(426GB)	Y	N
[Ding 10]	TSP·LSH; URL·doc size	Wiki08(50GB); Ireland(160GB); GOV2(426GB)	Y	Y
[Silvestri 10]	URL	WT10g(10GB); WBR99(22GB); GOV2(426GB)	Y	Y
[Tonellotto 11]	doc size; URL	ClueWeb09-CatB(1.5TB)	Y	Y
[Shi 12]	term frequency	FBIS(470MB); LATimes(475MB); WT2g(2GB); Wiki12(17GB)	Y	N
[Arroyuelo 13]	run-length	GOV2(426GB)	Y	Y

Table 2.4: Details of reordering papers.

Ordering documents using various forms of content similarity has been shown to substantially improve compression [Blanco 06, Blandford 02, Ding 10, Shieh 03, Silvestri 04a, Silvestri 04b], but all the techniques are slow to calculate. The simple approach of ordering lexicographically by URL, however, is fast to calculate, just as good as any of the other approaches [Silvestri 07], and especially effective for the *GOV2* dataset. Ordering by URL achieves this performance gain by placing documents with similar terms close together to produce *tight clustering*, which is exactly what happens with the content similarity ordering techniques.

In addition to space savings, URL ordering has been shown to improve the runtime performance of *skips* by approximately 50% for queries when using the *GOV2* dataset [Yan 09]. As a result, for our queries and dataset we expect that URL ordering will be equivalent to the best ordering currently available.

We have found no published results on the runtime performance of *bitvector* based algorithms when reordering document identifiers. It is useful, however, to note that a list stored as a *bitvector* will use the same amount of space irrespective of the document ordering that is used, since each *bitvector* has one bit per document for a compact domain of document identifiers.

In general, a full ranking based search engine is likely to order by global rank or impact order, so that the query can prune portions of the intersection, while URL order is the most common approach for list intersection systems. Some recent work has tried to combine these ideas with a *block-max* algorithm [Ding 11] that uses URL ordering together with the pruning of blocks based on their maximum possible score. With this approach, extra space is required to store the *block-max* values, while the block layout must be tuned to produce a good space-time tradeoff.

Chapter 3

Setup and Validation

In addition to information about the dataset, workload, hardware configuration, and experimental design, we present details of coding style, memory layout, and overall verification to make our results trustworthy and reproducible.

3.1 Experimental Setup

Our goal is to measure the index space usage and query runtime performance of an in-memory list intersection system. Our space and time values ignore the cost of the dictionary, and our algorithms do not encode positional information nor ranking information. For a disk based system, the amount of data in each query is the amount of data transferred from disk, but an in-memory system accesses only some portions of that data. As a result, we do not record the amount of data in each query. We also do not record the amounts of memory accessed in each query, because the transfer costs of moving data from memory to CPU cache in our system are already captured in the query runtime.

The index space is calculated by summing the encoded size of every postings list in the complete dataset. This value is usually divided by the number of (document level) postings in the dataset, so that index space has units of bits/posting.

In order to measure query runtime, our experiments simulate a full in-memory (but out-of-cache) index: we load the postings lists for query batches, encode them, flush the CPU cache by scanning a large array, then measure the time to execute the conjunctive intersection of terms and produce the results. For the experiments in this thesis, each batch contains 10 queries. Within a batch, separate postings lists are created for each query to avoid coincidental caching effects; thus, performance is independent of query order and shared terms. Intersection runtimes per step are recorded, and overall runtimes are the

sums over all steps for all queries. Note, some implementations combine multiple steps to improve code clarity.

We measure query runtime using single threaded execution and assume that the relative performance of the algorithms is proportional to their performance in a multi-threaded environment, where multiple queries or various portions of each query are run concurrently. This assumption is present in most of the related work involving the performance of list intersection, so we do not verify it and leave additional investigation for future work.

Our space-time graphs present an algorithm's performance as a line connecting the performance using various parameter settings. We present the space axis as bits per posting and the time access as milliseconds per query. We directly compare the runtimes of two algorithms t_{base} and t_{new} using *speedup* (t_{base}/t_{new}), where larger is better. Similarly, we compare the space required by two algorithms s_{base} and s_{new} using *compression benefit* or *comp-ben* (s_{base}/s_{new}), where again larger is better.

Our code was run on an AMD Phenom II X6 1090T 3.6Ghz Processor with 6GB of memory, 6MB L3, 512kB L2, and 64kB L1 caches running Ubuntu Linux 2.6.32-43-server with a single thread executing the queries. The gcc 4.4.3 compiler was used with the -O3 optimization settings to produce high performance code. The queries were visually verified to be returning plausible results and automatically verified to be consistent for all algorithms. The CPU cache was flushed before query execution, and Valgrind[1] was used to verify proper memory usage.

3.2 Dataset and Workload

We take the TREC *GOV2* corpus, index it using the Wumpus[2] search engine with no stemming and no stopwords, then extract the document level postings. This corpus contains 426GB of uncompressed text in 25.2 million documents, resulting in 9 billion document level postings and 49 million terms when processed by Wumpus. These values are much larger than in related work, such as the 6.8 billion tokens and 37 million terms reported by Ding et al. [Ding 10], because Wumpus indexes the general markup of regions and the HTTP header information, as well as the document content. Even with this difference in index size, the compression rates are consistent with existing work, as shown in Table 3.1 for *OptPFD* using the original ordering and the (sorted) URL ordering.

Our workload is a random sample of 5,000 queries chosen by Barbay et al. [Barbay 09] from the 100,000 corpus queries, which we have found to produce stable results. The query text is interpreted simply by using Wumpus to tokenize the text without any operator

[1] http://valgrind.org/

[2] http://www.wumpus-search.org/

		URL order		Original order	
	postings (billions)	size (GB)	size (bits/posting)	size (GB)	size (bits/posting)
[Yan 09]	6.797	2.853	3.36	5.903	6.95
Our experiments	9.043	3.748	3.32	7.295	6.45

Table 3.1: Verification of the compressibility of our index compared to existing results using $OptPFD_{128}$ sizes.

interpretation (i.e., phrases, proximity, or Boolean modifiers) giving an average of 4.1 terms per query. Query statistics are summarized in Table 3.2, including averages of the *smallest* list size for each query, the sum of *all* list sizes, and the *result* list size over all queries with the indicated number of terms and applied to the entire corpus. For our runtime per query calculations, we remove the single term queries.

terms	queries	%	smallest	all	result
1	92	1.8	131,023	131,023	131,023
2	741	14.8	122,036	1,520,110	39,903
3	1,270	25.4	194,761	6,203,147	31,730
4	1,227	24.5	199,732	13,213,388	17,879
5	803	16.1	204,093	20,361,435	13,087
6	428	8.6	192,445	29,367,581	15,004
7	206	4.1	205,029	36,346,235	8,240
8	98	2.0	206,277	46,198,187	5,726
≥ 9	135	2.7	198,117	63,406,170	3,308
Total	5,000	100.0	186,070	14,944,683	24,699

Table 3.2: Query information.

Our set of 5,000 queries is a good representation of the 100,000 corpus queries, as shown by the terms per query breakdown in Figure 3.1. These corpus queries are used as a standard benchmark in search engine research, but other workloads can be quite different, as shown by the Excite 1999 queries (original and unique). Throughout this thesis, we also present some of our final results using a terms per query breakdown, so that those results might be more broadly applicable.

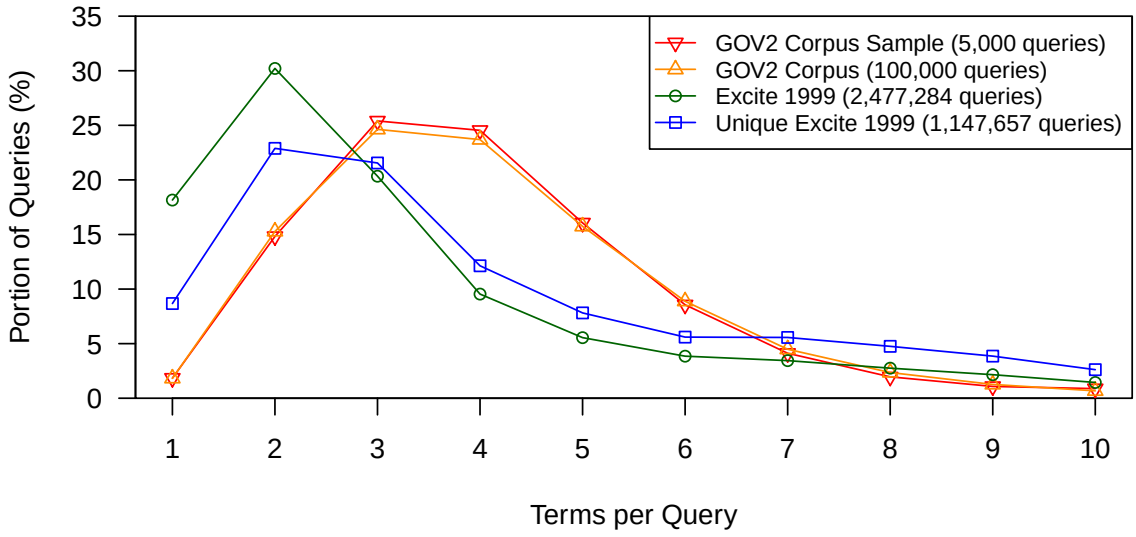

Figure 3.1: Terms per query breakdown for various benchmark query workloads.

3.3 Style of Code

Our code was designed to have good performance while still being maintainable. We used the C++ language and classes for readability, but the core algorithms use only non-virtual inline functions, allowing a large range of compiler optimizations. We also simplified our implementations by using zero to represent an uninitialized iterator. Whenever possible, temporary variables are allocated on the stack rather than the heap for faster creation, better cache line usage, and reduced probability of cache line eviction. (Note, eviction is common, since the data is usually larger than the cache size.) Compression algorithms such as *PFD* were designed to remove branches (*if* statements) in the code, since they can prevent many hardware optimizations such as pipelining, out of order execution, and loop unrolling. In our intersection code, we also minimized the use of branches to improve query runtimes.

Where possible we have used the authors' original code. In particular, the code for the *OptPFD*, *PFD*, *S9*, and *S16* block based compression code was provided by the researchers at WestLab, Polytechnic Institute of NYU. In addition, the uncompressed intersection code was provided by Alejandro López-Ortiz, and the original *hybrid bitvector* code was provided by J. Shane Culpepper. We implemented our own versions of *vbyte* compression, list index intersection (*skips* and *segment*), *hybrid bitvector*, and *semi-bitvector* algorithms. These components were run directly, or combined as needed to produce the resulting intersection algorithms examined in this thesis.

Definitions:	+ refers to concatenation $v(x)$ = *vbyte* encoding of x $a(x)$ = pad to x byte alignment
Require:	$n \leftarrow$ number of integers in list $D_c \leftarrow$ compressed data $D_b \leftarrow$ *bitvector* data $I \leftarrow$ *skips* or *segment* list index $B_s \leftarrow$ *segment* bits

- $BLOCK = v(n) + a(4) + D_c$
- $BLOCK\ \ SKIPS = v(n) + v(|D_c|/4) + a(4) + D_c + I$
- $VBYTE = v(n) + v(|D_c|) + D_c$
- $VBYTE\ \ SKIPS = v(n) + v(|D_c|) + v(|I|) + D_c + a(4) + I$
- $VBYTE\ \ SEGMENT = v(n) + v(|D_c|) + v(|I|) + v(B_s) + D_c + a(8) + I$
- $BITVECTOR = v(n) + a(4) + D_b$

Table 3.3: Layout for storing encoded lists and related structures.

While implementing these algorithms, several key considerations made the results much more stable and often faster than would be otherwise observed. As explained below in more detail, these include data memory layout, single pass processing, and cache coherency of query setup information.

Since data layout affects performance, we have encoded directly into a byte array for each list, and we include decoding time in our results. This ensures that we account for all memory in our size calculations and that the layout is consistent, thus producing more realistic and repeatable results. To avoid copying, our list indexes (arrays of integer pairs {location, delta}) point directly into the original byte array, and we improve read performance by padding these arrays to four or eight byte alignment. For improved cache coherency at query setup time, all the array sizes occur before any of the arrays themselves, as shown in Table 3.3. For example, the *vbyte* encoding of each list of integers stores the number of integers as a *vbyte*, the length of the compressed data as a *vbyte*, and then the compressed data as a delta *vbyte* encoded stream. The *PFD*, *OptPFD*, *S9*, and *S16* algorithms use block based encoding, with lists of length 100 or less using *vbyte* encoding.

For in-memory systems, the memory access time (i.e., loading from memory into CPU cache) can be a large portion of the runtime cost of a query. As a result, multiple passes through the data can be especially expensive when the data is larger than the various CPU cache sizes. To alleviate this cost, we combine multiple operations for a single intersection step so that it acts on the data in a single pass. For example, when intersecting *bitvector* only queries, the final step combines the *bitvectors* using bitwise AND and converts the

results to a list of integers all in a single pass, which does not need temporary storage of the bitwise AND result *bitvector*.

Combining these design considerations with some low level optimizations, we have produced a new *hybrid bitvector* implementation *H2-K* (Kane) that follows the pseudo code of the *H2-CM* (Culpepper and Moffat) implementation but is faster for all configurations. The runtime performance difference between our new *H2-K*($Vbyte_1\ bitv_F$) implementation compared to the original *H2-CM* implementation is shown in Figure 3.2. The runtime performance is calculated as an average runtime per query, but the runtimes for individual queries may have large variance. We find that the standard deviation of the query runtimes is much larger than the average runtime improvement from our new *hybrid bitvector* implementation, as shown in Table 3.4. However, applying a paired t-test to these query runtimes, we find that the p-value is extremely low, meaning that these average runtime performance improvements from using our new *hybrid bitvector* implementation are statistically significant. As a result, we use our new implementation as the basis of subsequent *bitvector* algorithms in this thesis. In addition, since the statistical significance is so strong, we assume that other similarly consistent performance improvements found in this thesis are also statistically significant.

	H2-CM($\text{Vbyte}_1\,\text{bitv}_F$)(orig)		H2-K($\text{Vbyte}_1\,\text{bitv}_F$)(orig)		
F	average	stdev	average	stdev	p-value
1/8	6.01	5.53	5.30	4.88	0.000
1/16	4.04	3.82	3.58	3.33	0.000
1/32	2.72	3.12	2.35	2.46	0.000
1/64	2.35	3.46	1.87	2.43	0.000
1/128	2.62	4.00	1.93	2.69	0.000
1/256	3.14	4.44	2.18	2.91	0.000

Table 3.4: Analysis of statistical significance of runtime improvements using average runtime (ms/query), standard deviation of runtime (ms/query), and paired t-tests of individual query runtimes (p-value).

3.4 Validation

When dealing with in-memory systems, small details of the execution approach can have significant impacts on the observed runtimes. The main cause of this variation is the CPU cache hierarchy, which causes highly varying access times for data dependent on when it was last accessed and what other data has been accessed in between. For simulated systems,

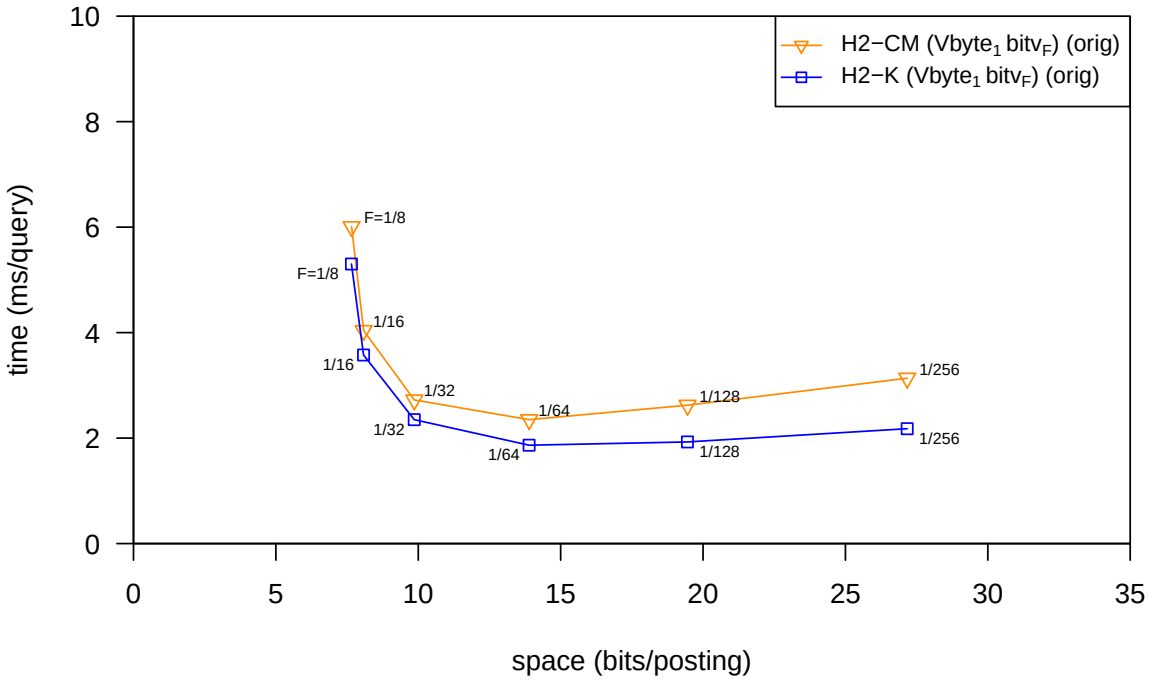

Figure 3.2: Space vs. time for *hybrid bitvector* implementations using original ordering. The three smallest configurations of *H2-CM* were compared to *skips* in Figure 2.5.

the encoding of the data is often done just before the query execution, meaning the encoded data could be in the CPU cache. For any system, if queries are executed multiple times, the data could be in the CPU cache after the first execution. To prevent these interactions, we only execute each query once per test run, and flush the cache between the encode and execute steps. (The encoded size of the 10 queries in a batch will likely be larger than the cache size, causing its own cache flushing.)

We validate our method of query execution by showing stability between multiple query runs. The runtimes of five separate query executions over five different random orders of the documents is presented in Table 3.5 for the *PFD+skips* algorithm using a skip size (and block size) of $X = 128$, and also the *vbyte+bitvectors* algorithm using the recommended setting of $F = \frac{1}{32}$. The resultant standard deviation for the five runs is very small, meaning that our execution runtimes are stable. This demonstrates that our query load produces stable runtime between executions, even though each query is only run once and we are using a subset of the corpus queries. It also demonstrates that our document randomization is indeed random and produces stable performance numbers.

We validate the independence of individual queries by showing that repeating the same

	$\mathrm{SK}(\mathrm{PFD}_{128}\,\mathrm{skip}_{X=128})(\mathrm{rand})$		$\mathrm{H2}(\mathrm{Vbyte}_1\,\mathrm{bitv}_{F=1/32})(\mathrm{rand})$	
	space	time	space	time
random 1	7.8841	7.2994	10.1828	2.4723
random 2	7.8840	7.3034	10.1826	2.4695
random 3	7.8841	7.3037	10.1829	2.4721
random 4	7.8838	7.3003	10.1827	2.4685
random 5	7.8840	7.3012	10.1828	2.4704
average	7.8840	7.3016	10.1828	2.4706
stdev	0.0001	0.0019	0.0001	0.0016

Table 3.5: Validation of random ordering and repeatability of query runs using space (bits/posting) and time (ms/query).

query twice produces the same runtime performance per query as running the query once. We do this by taking our query workload and doubling every query so that the duplicates are run one after the other. Since our system creates independent encoded lists for each query, these duplicates should have their own lists in memory and thus be independent. Indeed, the difference in query runtime of the double query workload compared to the average over five runs for the single query workload is very small, as shown in Table 3.6.

	$\mathrm{SK}(\mathrm{PFD}_{128}\,\mathrm{skips}_{X=128})(\mathrm{rand})$	$\mathrm{H2}(\mathrm{Vbyte}_1\,\mathrm{bitv}_{F=1/32})(\mathrm{rand})$
single queries	7.3016	2.4706
double queries	7.3012	2.4683
difference (abs.)	0.0004	0.0023

Table 3.6: Validation of query independence using query runtime (ms/query).

Our experiments using *OptPFD* compression produce an index size that is consistent with the published results, both when using the original document ordering and the URL ordering, as was shown in Table 3.1. Similarly, the index sizes produced by the other compression algorithms are consistent with published results relative to the compressibility of the data, as shown in Table 3.7. Compressibility is measured using entropy, as described in Section 5.2. The index sizes from using the *hybrid bitvector* encoding show a similar ability to improve the compression of dense lists relative to the *vbyte* compression algorithm, as shown in Table 3.8. However, the amount of compression for our index is larger, because we have more dense lists from including metadata tags and HTML headers.

The uncompressed intersection algorithms and the original *hybrid bitvector* implemen-

	Vbyte_{128}	S9_{128}	S16_{128}	PFD_{128}	Entropy
PolyBot 2002 (2.0 billion postings)	10.05	9.05	8.84	8.24	7.44
GOV2 (9.0 billion postings)	9.50	7.37	6.99	7.00	6.71

Table 3.7: Space (bits/posting) using blocks of size 128 and the original ordering. The first dataset, PolyBot 2002 [Zhang 08], contains Web pages crawled in 2002 by the PolyBot Web crawler [Shkapenyuk 02].

		$\text{Vbyte}_1\,\text{bitv}_F$			
	Vbyte_1	$F = \frac{1}{8}$	$F = \frac{1}{16}$	$F = \frac{1}{24}$	$F = \frac{1}{32}$
GOV2 (6.1 billion postings)	9.74	9.16	9.61	10.58	11.67
GOV2 (9.0 billion postings)	9.51	7.65	8.08	8.90	9.86

Table 3.8: Space (bits/posting) for *hybrid bitvector* algorithm using *vbyte* and the original ordering. The smaller *GOV2* index was produced by Moffat and Culpepper [Moffat 07a] using Zettair[3].

tation (*H2-CM*) use the original authors' code, after being ported to our test harness. The runtime performance using the *PFD* algorithm is faster than *S16*, while the *S9* algorithm is comparable to *S16*, agreeing with published results [Zhang 08]. A more detailed analysis of intersection performance when using *PFD* compression is presented in Section 4.1. When using *vbyte* compression, we found the settings which produced the fastest configurations of the *skips* ($X = 32$) and *segment* algorithms ($B = 8$) were the same as the published results [Sanders 07], validating our implementations for these algorithms. When considering our *skips* implementation, the fastest *vbyte+skips* configuration is faster than Culpepper and Moffat's list intersection implementation ("svs+bc+aux") [Culpepper 10], after normalizing the performance relative to the original *hybrid bitvector H2-CM*($Vbyte_1\, bitv_{F=1/8}$) setup. A detailed analysis of the runtime performance of these list index algorithms is presented in Section 4.4.

As a result of examining our experimental process and comparing our results to previous work, we believe our experiments are producing valid and consistent space-time results.

Chapter 4

Combining Compression, Skips, Bitvectors, and Reordering

Various approaches have been used to improve the space and runtime of list intersection algorithms, as summarized in the related work described in Chapter 2. These techniques include compressing the lists, adding list indexes (*skips*) for fast searching in lists, using *bitvectors* for large dense lists, and reordering to increase term clustering, which improves both space and runtime performance. Because of its size, the design space of combining these techniques has not been carefully explored. In particular, the use of *bitvectors* for large dense lists has only been combined with *vbyte* compression and not *skips* or reordering, while the interaction of *skips* with the choice of compression algorithm and document ordering is not clear. We explore the combination of compression, *skips*, *bitvectors*, and reordering using experiments to give detailed space-time performance results. These results indicate the beneficial combinations of these list intersection approaches, which are then examined in detail to determine the reasons for their superior performance.

4.1 Compressed Intersection

There are three steps involved in intersecting compressed lists: decode the compressed form (decode), undo the deltas (restore), and combine the lists (combine). Many of the published speeds encompass only the decode step, but clearly the implementation of the restore and combine steps could significantly affect overall performance. Some recent work does, however, report decode+restore runtimes [Lemire 13].

The standard *vbyte* algorithm is usually implemented in an incremental fashion, meaning the values are decoded one at a time; thus, the decode, restore, and combine steps are

	runtime
MG(PFD_{256}) decode	7.70
MG(PFD_{256}) decode+restore	16.77
MG(PFD_{256}) decode+restore+combine	30.73
MG-gallop($U32_1$)	14.67

Table 4.1: Breakdown of average runtime (ms/query) for merge intersection using *PFD* compression and original ordering.

all interleaved. This interleaving and the byte aligned nature of the *vbyte* encoding results in moderate performance on modern processors. The lack of random access, however, is a major bottleneck, since it usually requires that the lists be processed in their entirety.

Most other compression algorithms act on blocks of integers, many with adaptive block sizes. Some of these algorithms, such as *S9*, can be efficiently decoded in an incremental fashion, but most either require decoding of the entire block or are faster when decoded as an entire block. As stated earlier, we combine the small word-sized blocks produced by *S9* and *S16* into larger blocks of a fixed size in order to compare them directly with *PFD*. Similarly, we use a block based version of *vbyte* for comparison. We explore block sizes of 32, 64, 128, and 256 for these block based algorithms; however, the *OptPFD* implementation provided to us is hard coded for a block size of 128 and produces only the most compact, but slowest, encoding in its tuneable range.

Using blocks will often require separation of the decode step from the restore and combine steps, preventing some interleaving. Nevertheless, we interleave the restore step with a linear scan for the combine step, while decoding each block as needed. This has the advantage that intermediate forms are never flushed from the CPU cache, which could happen if the decode and restore steps act on the lists before the combine step is executed.

Table 4.1 shows a breakdown of the runtime performance for the *merge* algorithm using *PFD* compression when documents are numbered in the original order. Adding the runtime of the $MG(PFD_{256})$ decode+restore entry to the fastest uncompressed intersection algorithm *MG-gallop*($U32_1$) gives a runtime similar to our implementation, thus further validating our results. It is possible that other implementations of the *restore* and combine steps could be faster, but we leave this exploration for future work.

Using our implementation of *merge*, the resultant runtimes for intersecting compressed lists is much slower than the uncompressed intersection algorithms for the original order, as shown earlier in Figure 2.4. The incremental *vbyte* algorithm is faster than *PFD* because interleaving of the decode, restore, and combine steps allows for hardware optimizations such as instruction reordering. However, the block based implementation of the *vbyte* algorithm prevents some of this interleaving, and it is therefore much slower (44.7 ms/query

for *vbyte* using blocks of 256 values vs. 24.9 ms/query for incremental *vbyte*), as shown earlier in Figure 2.4.

4.1.1 Reordering with Compression

When considering the reordering of document identifiers, one needs a baseline for comparison. The original dataset order could have been produced in any number of ways, so a random ordering is a more stable baseline. Unfortunately, a random order has not been commonly used in previous experiments; instead the original order is the standard ordering used in previous work. When we examined a random order for *GOV2*, we found that our space and runtime performance results were only slightly degraded compared to the original order. As a result, the original order turns out to be an acceptable baseline for *GOV2*, and so we have chosen to use it in the rest of this chapter.

Reordering document identifiers using lexicographic URL order has been shown to significantly improve the space usage for various compression algorithms. The space saving are from the clustering of documents with similar terms, thus producing smaller deltas, which encode more compactly. This smaller space usage could result in runtime savings, assuming that memory transfer rate is the bottleneck.

We find that runtime does not always improve for our compressed algorithms when intersecting using URL order; some algorithms are slightly faster and some slightly slower ($\pm$12%). The runtimes of the decode, restore and combine steps of the *PFD* algorithm with URL order are each similar to the corresponding times with the original order. Clearly, the space benefits do not translate directly into runtime benefits. This lack of consistent runtime improvement suggests that memory transfer is not the bottleneck here. Perhaps the improvements to the decoding step are insignificant, since it is only a small portion of the runtime for these algorithms, as shown earlier in Table 4.1. The reordering could also be producing less predictable execution paths, which would prevent some hardware optimizations, or more exceptions in the *PFD* encoding, which are slower to decode. We leave a more detailed examination of these decoding algorithms for future work.

The URL ordering significantly improves the tight clustering within the lists, which reduces the delta sizes substantially. Approximately 69.8% of the deltas have the value one in URL ordering vs. approximately 46.2% for terms-in-document ordering, 23.5% for original ordering, and 20.4% for random ordering. This suggests that there is limited room for additional improvement from new ordering methods. Our measurement of the percentage of deltas having the value one for URL ordering of the *GOV2* index is higher than reported previously (59% [Ding 10]), because we include more repetitive content such as the HTML headers and document region tags. Compressibility is examined in more detail in Section 5.2.

	orig	url	comp-ben
Vbyte_1	9.51	8.74	1.09
PFD_{256}	6.99	5.26	1.33
OptPFD_{128}	6.45	3.32	1.95
S9_{256}	7.33	4.12	1.78
S16_{256}	6.95	3.95	1.76

Table 4.2: Space (bits/postings) and compression benefit using original and URL ordering.

The space usage also improves significantly for our compression schemes when the data is ordered by URL, as has been shown previously [Silvestri 07, Yan 09]. The rate of improvement, however, is not the same for all the encodings, as shown in Table 4.2.

As expected, the space used by *vbyte* has little improvement under the URL order, since the minimum encoding size is 8 bits/posting. Unlike the original paper on URL ordering [Silvestri 07], where the dataset appears to include many large deltas, we find that *S9* achieves very good compression for *GOV2*. The space used by the *S9* and *S16* encodings improves substantially under the URL ordering, and, in fact, *S16* and *S9* are now much smaller than *PFD*. Surprisingly, the space used by the *S9* algorithm is closer to *S16* when using the URL ordering, even though *S16* is tuned for small deltas (i.e., more layout options for deltas that can be stored in a small number of bits) which are more common under URL ordering.

The *OptPFD* algorithm combines *S16* with parts of *PFD* to be faster and smaller than both; it may even be faster than *S16* and *PFD* when using the most compact setting, as we are. The space improvement is larger for *OptPFD* when using URL ordering than it is for the other compression algorithms, as shown in Table 4.2; clearly this encoding is a good combination of *PFD* and *S16* for space usage. The runtime performance of *OptPFD*, however, is slower for the original order as shown earlier in Figure 2.4, and it is also slower under URL ordering. This could indicate a problem with the implementation or the algorithm could indeed be slower, but regardless of the reason for the slow performance, we do not explore *OptPFD* in the remainder of this thesis. (We expect that *OptPFD* will behave similarly to *S16* when adding list indexes and *bitvectors*.)

The *VSE* algorithm is an offshoot of the ideas found in the *PFD* and *S9*/S16 algorithms that results in a good space-time tradeoff. Although this algorithm may be better than *PFD* and *S16* for compression and decoding speed, the block sizes may interfere with the choice of skip points. Given proper skip points, we expect that *VSE* will produce similar results when *skips* and *bitvectors* are added. We leave further investigation of *VSE* compression for future work.

In all cases, we found that the runtime performance of the *S9* algorithm is similar to

S16, but *S9* uses slightly more space. As a result, we omit *S9* from further comparisons, and in the remainder of this chapter, we explore the performance of *vbyte*, *PFD*, and *S16* when *skips* and *bitvectors* are added. We do not expect variants such as *OptPFD* or VSE, or optimizations such as vectorization, to change the overall conclusions found in this thesis.

4.2 List Indexes

List indexes improve the random access performance for compressed lists by providing entry (skip) points into the lists, so that decoding can start from those points. Clearly, the ability to skip over values in a list is a big advantage when combining two lists with very different sizes. Such size differences are common in our workload, thus explaining why even having skip points every 256 entries results in substantial performance gains over intersecting using only the compressed form. For example, with the original document order, the *vbyte* algorithm goes from 24.9 ms/query (Figure 2.4) to 6.5 ms/query (Figure 4.1) when skips to every 256^{th} posting are stored in the list index. That improvement is a 3.8x speedup simply from adding skips every 256 postings, and it represents 94% of the possible benefits from adding *skips* to the *vbyte* compressed lists for any of the skip sizes we tested.

Adding *skips* to the compression algorithms being examined results in large runtime gains for all skip sizes we examined, as shown in Figure 4.1. The *skips* code, however, degrades in performance if the skip size (X parameter) gets too small. This degradation is to be expected, since processing the skip points will add some runtime overhead that can offset the gains provided by jumping over elements in the compressed form. In addition to the skip sizes presented, our performance curve for *skips* could include the performance when using just the underlying compressed form, since this represents the performance as $X \to \infty$, but we omit this when it is not needed.

When implementing *skips*, there are two additional variants that could be employed: (1) For small X values, the skip structure itself becomes fairly dense and thus scanning the skip structure in a non-linear manner could speed up performance; (2) Removing the skip values from the original list could be used to improve compression at the expense of runtime, especially when iterating over the list. Neither of these variants were found to be beneficial in our experiments.

When the document IDs are in their original order, we found that the performance of the *segment* code is faster than the *skips* code, agreeing with previous results [Sanders 07]. However, for cases when there are few skip points ($X \geq 64$), using *skips* or *segments* produce similar runtime performance. These large skip sizes turn out to include the configurations of interest in this thesis. When using URL ordering, the clustering of postings gives a skew in the size of *segments* and thus a slowdown in runtime performance of the

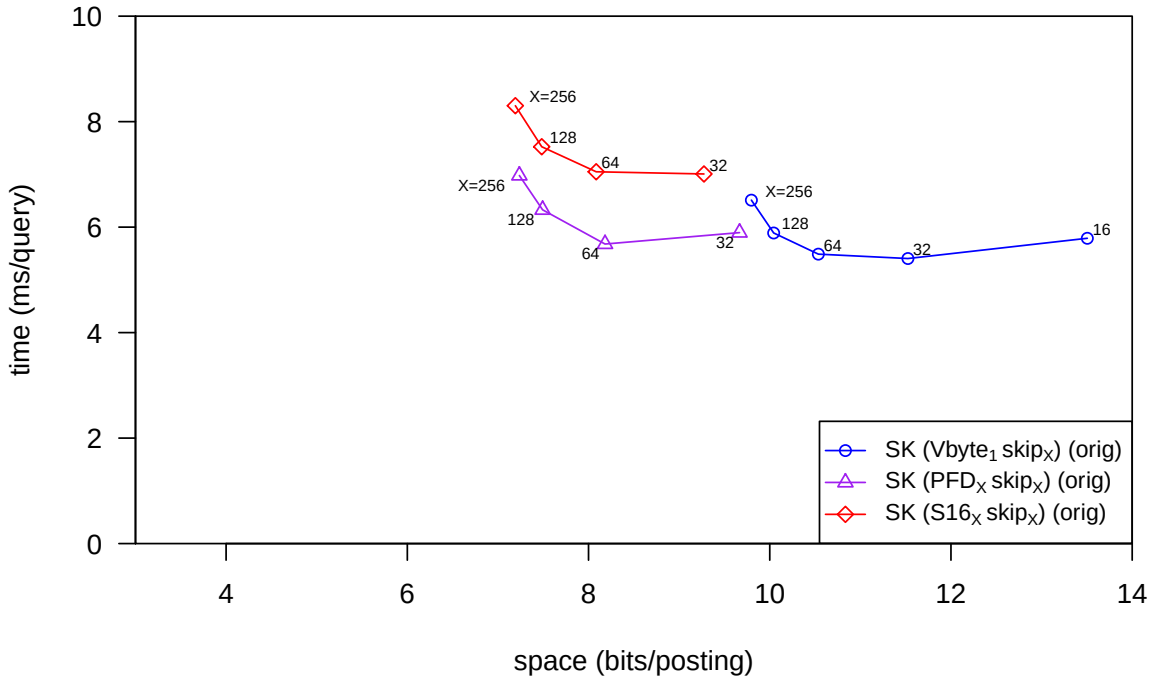

Figure 4.1: Space vs. time for *skips* using original ordering. The fastest configurations of *skips* were presented in Figures 2.4 & 2.5.

segment intersection algorithm. In addition, the *segment* code cannot be applied to the block based compression algorithms without modifications, since *segment* end points do not align with block end points. As a result, the *segment* algorithm gives no additional benefit to our analysis, and we omit it in the rest of this thesis.

It has previously been shown that using a list index reduces the number of decoded blocks and the query runtime by approximately half (i.e., a speedup of 2.0x) when documents are ordered by URL [Yan 09]. Our results also show significant runtime performance gains for *skips* using the URL order, as seen by comparing Figures 4.1 and 4.2, resulting in a similar speedup of approximately 1.9x, as shown in Table 4.3. The additional improvements to *skips* by using URL ordering, however, are not correlated with the space benefits. Instead, the URL ordering increases clustering of document identifiers in lists, which increases the likelihood of multiple results being encoded in the same block and thus amortizing the decoding costs. Viewed from the other direction, this clustering produces larger sequences of integers that do not contain a result and can, therefore, be skipped over. So, *skips* become more beneficial under the URL ordering, and as a result, the fastest skip size becomes larger. For example, the best configuration for *vbyte+skips* under the original

	orig	url	speedup
$\text{SK}(\text{Vbyte}_1\,\text{skip}_{X=64})$	5.49	2.96	1.86
$\text{SK}(\text{PFD}_{64}\,\text{skip}_{X=64})$	5.68	3.04	1.87
$\text{SK}(\text{OptPFD}_{128}\,\text{skip}_{X=128})$	7.35	3.87	1.90
$\text{SK}(\text{S9}_{64}\,\text{skip}_{X=64})$	6.95	3.50	1.98
$\text{SK}(\text{S16}_{64}\,\text{skip}_{X=64})$	7.05	3.54	1.99

Table 4.3: Runtime (ms/query) of *skips* with $X = 64$ (or 128) using original and URL ordering.

order is $X = 32$, but under the URL order it is $X = 64$.

As shown by these space and runtime benefits, there is clearly a large advantage to using URL ordering. The runtime benefits, however, are not proportional to the space benefits and are not solely the result of reading less data from the lists via a more compact structure. Instead, the runtime benefits are predominantly from term clustering causing *skips* to be more efficient and the intersection, therefore, decodes less of the larger list. This means that if a compression approach restricts skipping, it might not be valuable even if it uses less space. As such, space improvement and decoding time should not be the only metrics to be considered when comparing document reordering techniques.

4.3 Skips vs. Bitvectors

The *hybrid bitvector* algorithm proposed by Culpepper and Moffat [Culpepper 10] uses *vbyte* compression for small lists and *bitvectors* for large lists, with the list density cutoff determined by a tuneable value F. The original implementation, *H2-CM*, is much faster and smaller than *vbyte* with *skips*, as shown earlier and in Figure 2.5. We produced an even faster *hybrid bitvector* implementation and use it throughout the rest of this thesis.

The *hybrid bitvector* algorithm combines two small lists using the normal intersection algorithm, a small list with a large list using a containment call on the *bitvector* for each element in the small list (*bvcontains*), and two large lists using bitwise-AND with the last execution converting the results to a list of integers (*bvand+bvconvert*). Note, the result of the combination is temporarily stored in memory during query processing and may be a *bitvector* or an uncompressed list depending on the input. Like all the other intersection algorithms discussed so far, these steps are applied using the *set-versus-set* approach, where lists are intersected in increasing list size (frequency) order. The *hybrid bitvector* approach was summarized in Algorithm 9 (page 29) using a syntactic grammar to symbolically represent how a query is processed.

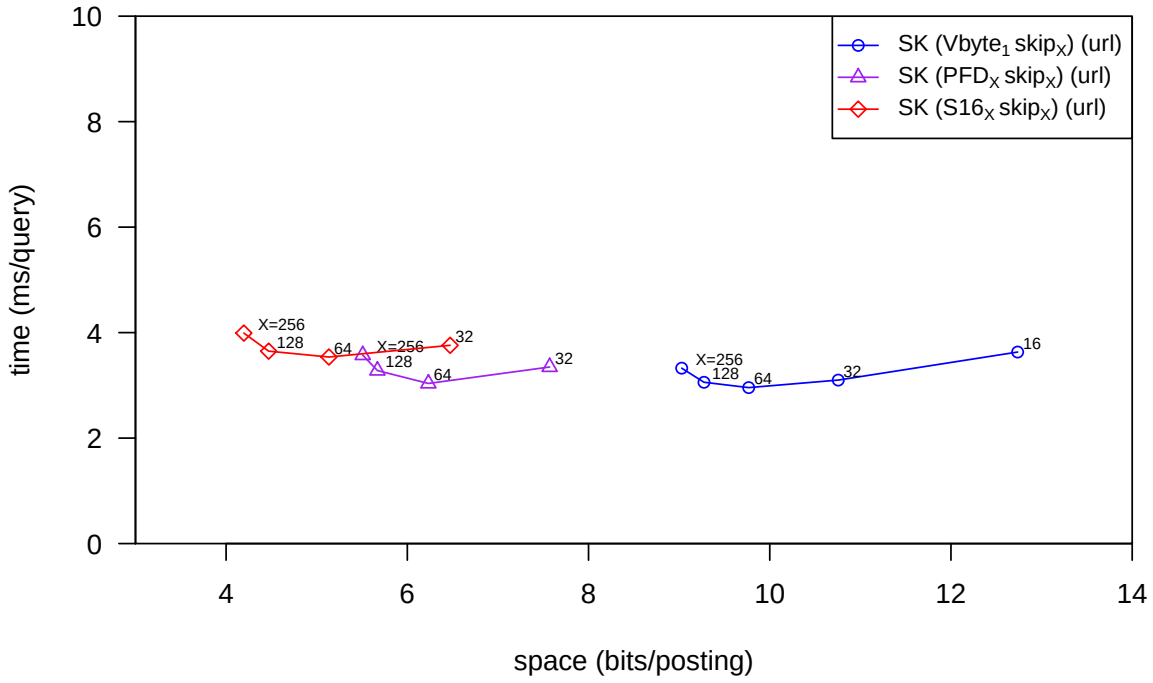

Figure 4.2: Space vs. time for *skips* using URL ordering. The same configurations using original ordering were presented in Figure 4.1.

Culpepper and Moffat showed significant space benefits from combining *vbyte* compression with *bitvectors*, because large lists can be more compactly stored as *bitvectors*. We augment those results by using either *vbyte*, *PFD*, or *S16*, with the last two executing on blocks of size 256. As expected, combining *PFD* or *S16* compression with *bitvectors* results in less space required than when combining *vbyte* with *bitvectors*, as shown in Table 4.4.

The *hybrid bitvector* approach uses a list density cutoff F to determine which lists are stored as *bitvectors* (large lists having higher density than F) and which are stored using delta compression. Since some lists can be more efficiently stored as *bitvectors* than they could be using delta compression, there is a minimum sized configuration at some F setting. In our results, the minimum sized configurations for our three compression algorithms occur at substantially different F values, and the resultant space savings vary. With *vbyte* compression, the minimum encoding size for one value is 8 bits, so any list with density $> \frac{1}{8}$ is smaller when it is stored as a *bitvector*. We tried density cutoffs $F \in \{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{24}, \frac{1}{32}, \frac{1}{48}\}$ and found $F = \frac{1}{8}$ produced the most compact configuration for *vbyte+bitvectors*. The *PFD* and *S16* algorithms have much smaller minimum encoding sizes, so the minimum sized configuration should be at a larger F value. Indeed, for the

	normal	bitvector	F	comp-ben
Vbyte_1	9.51	7.65	1/8	1.24
PFD_{256}	6.99	6.69	1/4	1.04
S16_{256}	6.95	6.84	1/4	1.02

Table 4.4: Space (bits/postings) and compression benefit from using *bitvectors* with original ordering.

same set of choices for density cutoff, *PFD* and *S16* with *bitvectors* require minimum space when $F = \frac{1}{4}$. However, the resultant space saving for the *PFD* algorithm when combined with *bitvectors* is small, and for *S16* it is very small, as shown in Table 4.4. Clearly, the *S16* algorithm does a very good job of compressing large lists.

For all three compression algorithms, there are substantial runtime gains from adding *bitvectors* when using the original order. Culpepper and Moffat showed that combining *vbyte* compression with *bitvectors* (*vbyte+bitvectors*) is faster than using *vbyte* compression with *skips* (*vbyte+skips*) when using the original document order, and our experiments confirm those results. The *PFD* and *S16* algorithms have never before been combined with *bitvectors*. Our experiments show that for both of these compression algorithms, the same result holds: converting some lists to *bitvectors* is better than adding *skips* when using the original order.

In a similar manner, the space and runtime performance of *bitvectors* with document reordering has not previously been explored.

Bitvectors do not change their space usage when documents are reordered, but the compressed lists can use less space under URL ordering. Even with the lack of space benefit for lists stored as *bitvectors*, some lists are still smaller using *bitvectors* than they would be using a compressed form under URL ordering. This change in compression size can affect the minimum encoding size, which now occurs at $F = \frac{1}{2}$ for *S16* compression combined with *bitvectors*, while *PFD* stays at $F = \frac{1}{4}$ and *vbyte* stays at $F = \frac{1}{8}$. The compression algorithms produce the expected relative sizes when combined with *bitvectors* using the URL order, so for similar F values *S16* compression combined with *bitvectors* is smaller than *PFD* with *bitvectors,* which is smaller than *vbyte* with *bitvectors*.

We found that *bitvectors* are significantly faster under URL ordering, as shown for the $F = \frac{1}{32}$ configuration in Table 4.5. This benefit is from documents being clustered in the URL order, producing fewer cache line loads within the *bitvectors*. However, this speedup of approximately 1.25x is not as large as the speedup for *skips* of approximately 1.9x, so the relative performance must be re-examined to see which is best under URL ordering. We found that the space-time performance of *S16+bitvectors* is worse than *S16+skips* for some configurations, and similarly that *PFD+bitvectors* is worse than *PFD+skips* for some

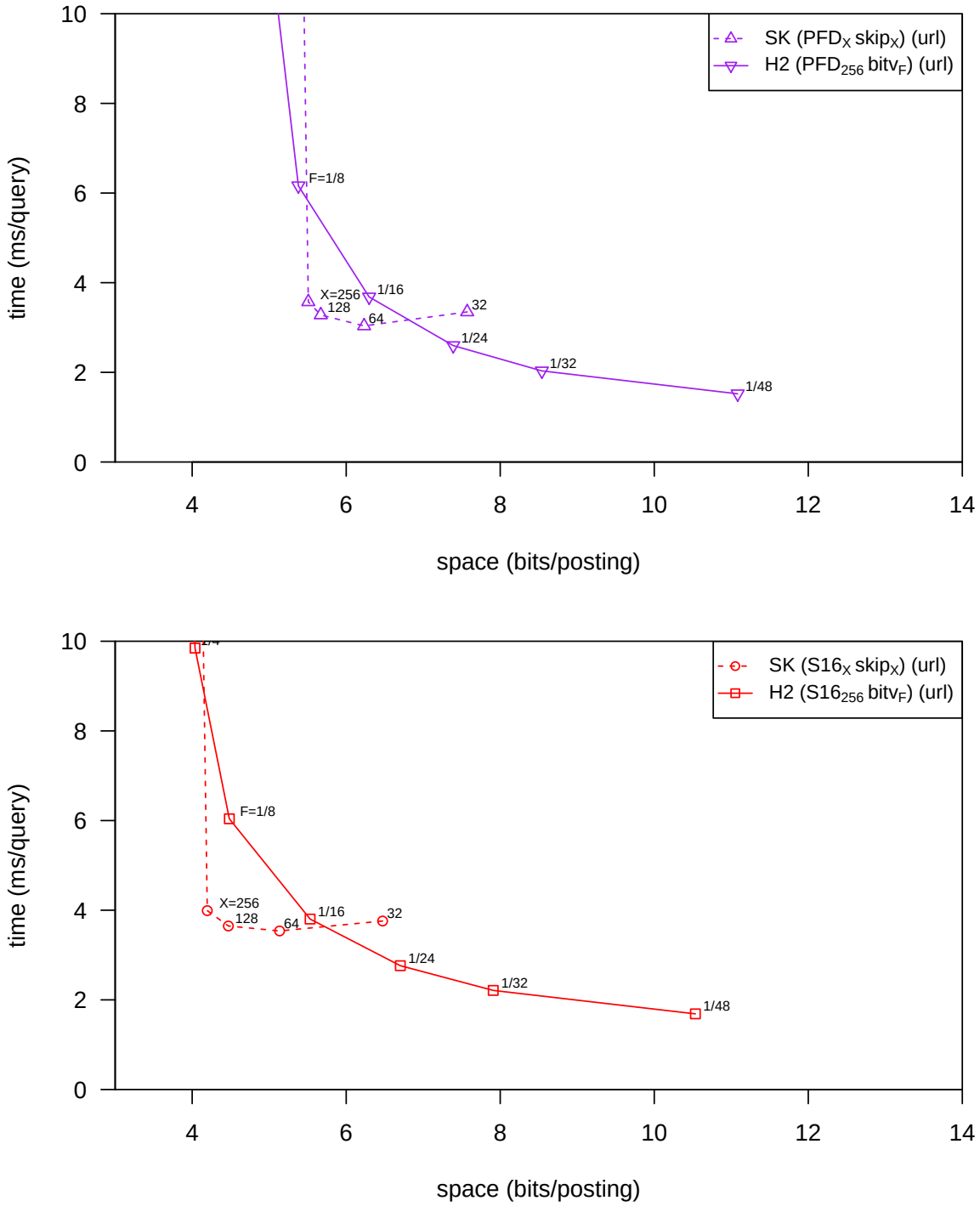

Figure 4.3: Space vs. time for *skips* and *bitvectors* using URL ordering. The same configurations of *skips* were presented in Figure 4.2.

	orig	url	speedup
H2(Vbyte$_1$ bitv$_{F=1/32}$)	2.35	1.81	1.30
H2(PFD$_{256}$ bitv$_{F=1/32}$)	2.40	2.03	1.18
H2(S16$_{256}$ bitv$_{F=1/32}$)	2.85	2.21	1.29

Table 4.5: Runtime (ms/query) of *bitvectors* using original and URL ordering.

configurations, as shown in Figure 4.3. So, previous results for the performance of *hybrid bitvector* schemes [Culpepper 10] do not hold in general, since we have shown that *skips can* outperform *bitvectors* over a set of queries when using advanced compression algorithms and URL ordering, even though *bitvector* structures are faster for dense lists.

The general trends of these algorithms are interesting: switching more lists to *bitvectors* will use more memory, but the runtime continues to improve and eventually is faster than the fastest configuration using *skips*, as shown in Figure 4.3. Adding *skips*, however, improves more quickly when using small amounts of space. In the figure, we emphasize the improvements from adding *skips* by connecting the largest skip setting ($X = 256$) with the point having no *skips* (i.e., the minimum size point). Clearly, there is potential for improvement from allowing both approaches to coexist. As a result, we examine methods to combine *bitvectors* with *skips* in the next section.

4.4 Skips with Bitvectors

Combining *skips* with *bitvectors* for large lists in a single intersection algorithm is a simple extension of our existing implementations, as shown in Algorithm 10. When we combine *vbyte* compression, *bitvectors*, and *skips* using various skip sizes, we see a significant performance gain for many configurations (i.e., choices of values for parameters X and F), as shown in Figure 4.4. Considering various skip sizes, we see that configurations with large skips are slightly slower, but the space savings with large skips are significant for

Algorithm 10 *H2SK* – Intersect *Hybrid Bitvector* and *Skips*

```
1: function H2SK(M,N,isLastStep)                    ▷ |M| ≤ |N|
2:     if !N.isBitvector then
3:         return SK(M, N)                          ▷ list (see page 24)
4:     if !M.isBitvector then
5:         return bvcontains(M, N)                  ▷ list (see page 28)
6:     return BV(M, N, isLastStep)                  ▷ list or bitvector (see page 28)
```

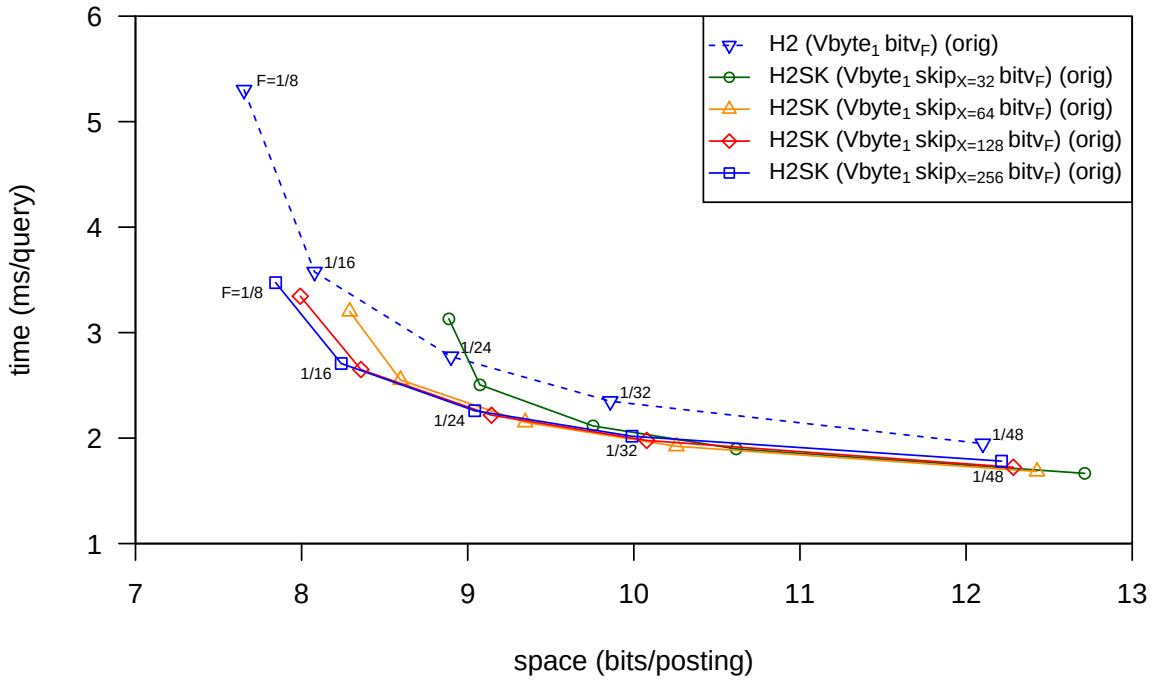

Figure 4.4: Space vs. time for *vbyte* with various sized *skips* and *bitvectors* using original ordering. The *hybrid bitvector H2* algorithm was presented as the *H2-K* implementation in Figure 3.2.

the smaller configurations (i.e., those with higher values for F). Overall, the large skip size ($X = 256$) works best for small configurations, while being competitive for larger configurations, meaning that this skip size produces a good space-time tradeoff. This result confirms our previous analysis, which showed that most of the gains from *skips* can be found with large skips, at which point adding *bitvectors* is more valuable than adding additional skip points. As a result, we use skips of size 256 for all the compression algorithms when combining *skips* with *bitvectors*.

We have picked a static skip size (256), but this could be varied depending on the system configuration. The skip size could also be chosen separately for each list based on the space budget of the system, perhaps by measuring the incremental space-time benefit of using *bitvectors* or adding *skips* to lists within various density ranges, then using a bin packing approach to fill up the space budget. While varying the skip sizes could produce a small benefit, we leave further investigation for future work.

Our implementation shows clear benefits when combining *bitvectors* with auxiliary indexes, but Moffat and Culpepper saw no additional gains from the combination [Moffat 07a].

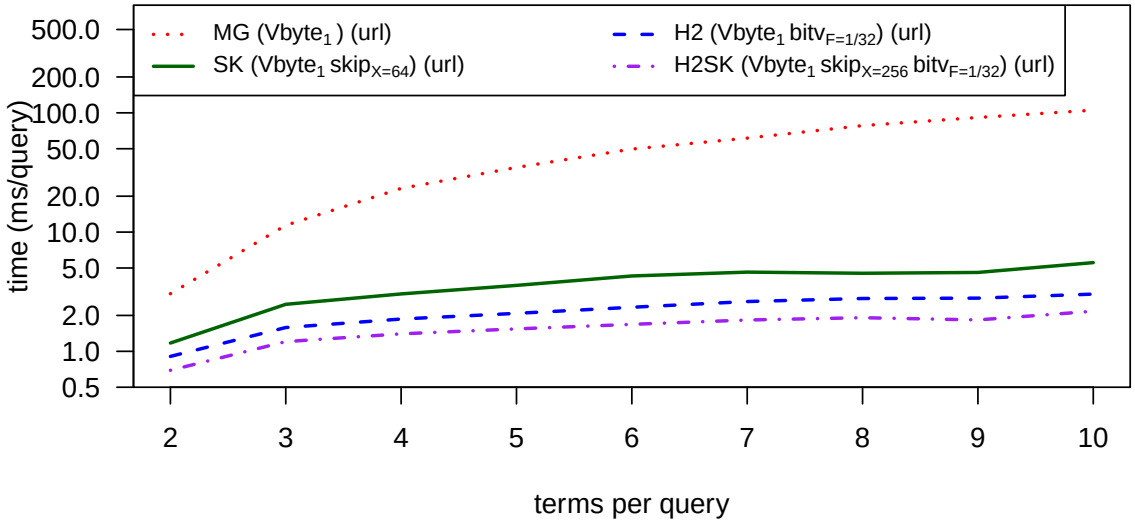

Figure 4.5: Runtime vs. terms per query for various *vbyte* based algorithms.

Their implementation of auxiliary indexes is based on skips of size $k \log n$, which varies with the list size. While their auxiliary indexes are separate from the original lists, they also remove the skip values from the original list (extracted *skips*) producing a more compact structure, but slower iteration over the list values. Our list index implementation has a constant skip size and maintains all the values in the underlying list, resulting in a larger structure, but faster iteration. Since our *bitvectors+skips* implementation is using large skips of size 256, the increase in space from maintaining all the values in the original compressed list (overlaid *skips*) is very small. Iterating over the lists, however, is done in the first step of intersection in the intersect ($SS \Rightarrow U$) and the *bvcontains* ($SL \Rightarrow U$) portions of the *hybrid bitvector* algorithm as defined in Algorithm 9 (page 29), which is significantly faster in our implementation. Indeed, their *skips* implementation ("svs+bc+aux") is the same speed or slower than their compressed intersection implementation ("svs+bc") for two-term queries [Culpepper 10]. Their experimental query set consists of mostly two-term queries (58%), and these were slower using their *skips* implementation, which explains why they found no benefit when combining *bitvectors* with their *skips*. Our *skips* implementation, however, is significantly faster than compressed intersection for any number of terms per query, and we find that combining *skips* with *bitvectors* is faster than *bitvectors* alone for any number of terms per query, as shown in Figure 4.5 using URL ordering.

When we combine *bitvectors* with *skips* of size 256 over our compressed encodings, we see large performance gains with both the original and URL orders, as shown in Figures 4.6 to 4.8. As expected, there are larger gains from combining *bitvectors* with *skips* compared

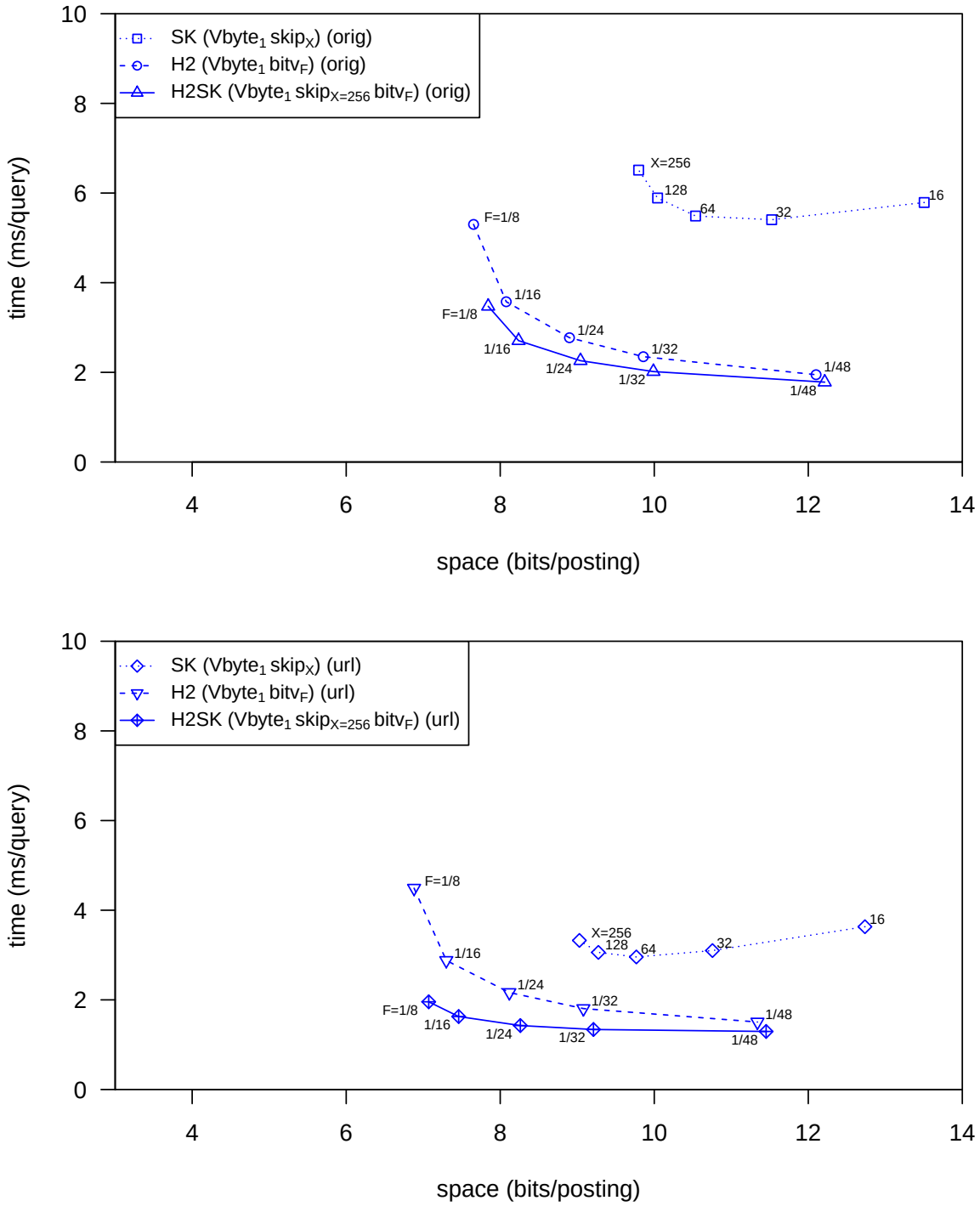

Figure 4.6: Performance of *vbyte* compression, *bitvectors*, and *skips* using original and URL ordering. Portions of these graphs were presented in Figures 4.1, 4.2 & 4.4.

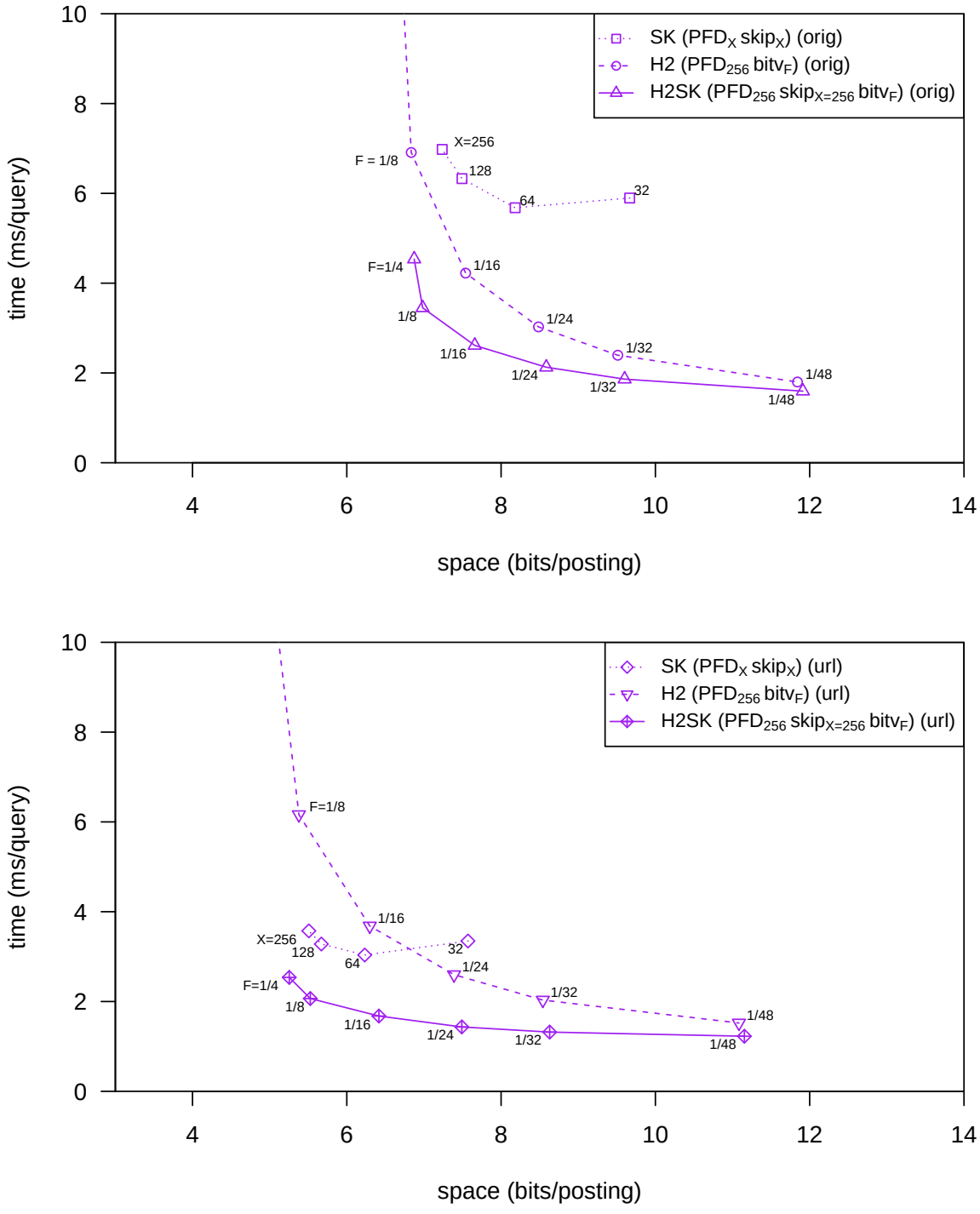

Figure 4.7: Performance of *PFD* compression, *bitvectors*, and *skips* using original and URL ordering. Portions of these graphs were presented in Figures 4.1, 4.2 & 4.3 (top).

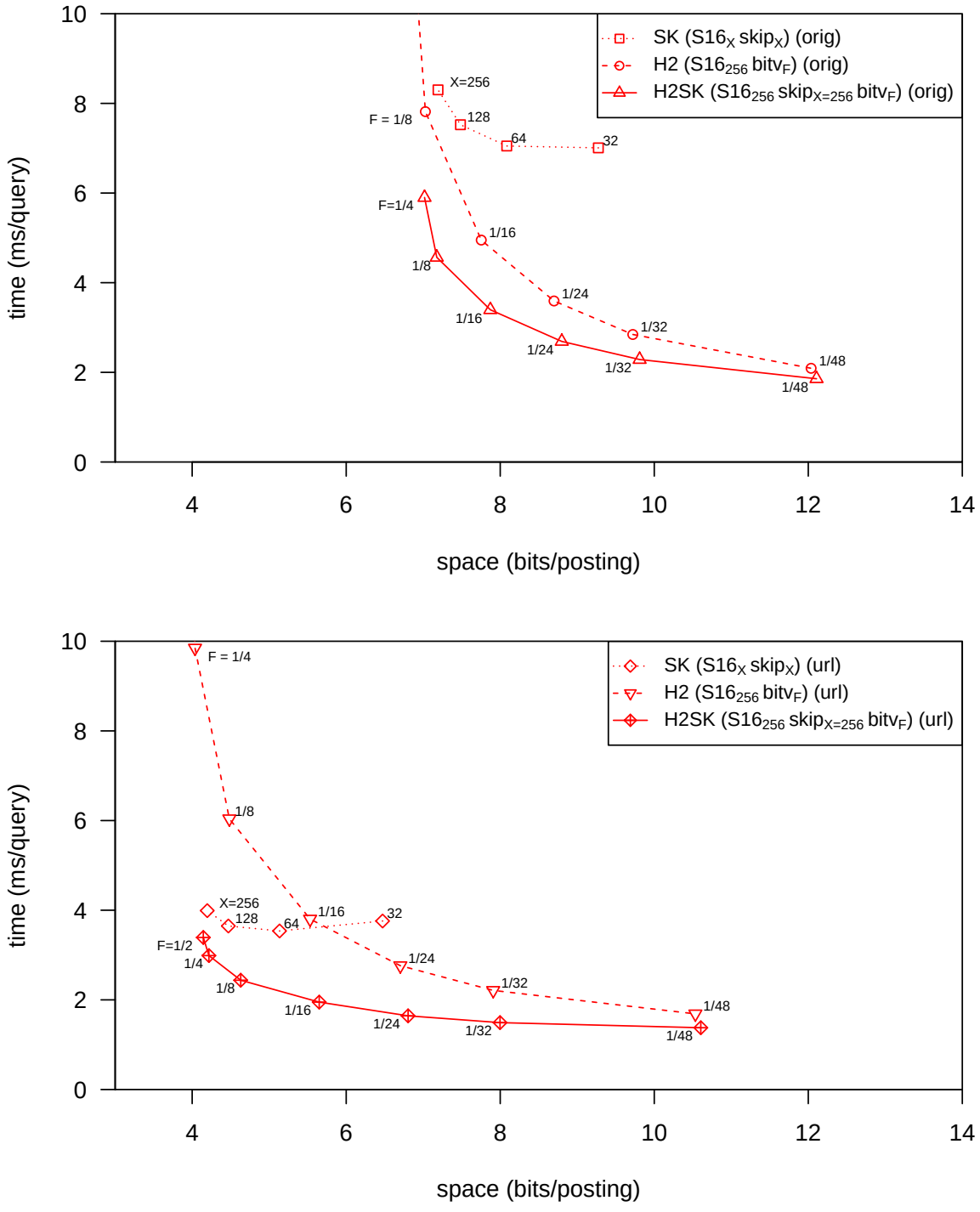

Figure 4.8: Performance of *S16* compression, *bitvectors*, and *skips* using original and URL ordering. Portions of these graphs were presented in Figures 4.1, 4.2 & 4.3 (bottom).

with *bitvectors* alone when using URL ordering, because the *skips* produce even more benefit with this ordering. However, our results are even more robust: we find that in all cases, combining *bitvectors* with *skips* has a better space-time performance than using *skips* alone, thus confirming that *bitvectors* are indeed faster than *skips* for the large lists in which we use them. Although Figure 4.4 shows that adding *skips* of size 256 to *hybrid bitvectors* results in a larger minimum size (see configurations with $F = \frac{1}{8}$), even larger skips could be used to reduce the space required if needed, so our results hold in general.

Clearly, combining *bitvectors*, large overlaid *skips* and URL ordering is a winning combination. It produces the considerable runtime improvement of *skips*, the additional runtime improvement and large list compression of *bitvectors*, and the space and runtime improvement from clustering in the URL ordering.

In addition to space-time graphs, we compare specific configurations as an example to emphasize the benefits of combining *bitvectors+skips* with URL order. In Table 4.6, we compare the $F = \frac{1}{8}$ configuration of *bitvector+skips* using the URL order to the fastest configuration of *skips* and the recommended $F = \frac{1}{32}$ setting for *bitvectors* using the original document order. We find that this combination results in an approximately 2.8x speedup and 1.5x to 2.0x compression benefit compared to the fastest configuration of *skips* using the original document order, and an approximately 1.17x speedup and 1.4x to 2.1x compression benefit when compared to the recommended configuration of *bitvectors* using the original document order.

	SK(__skip$_X$)(orig)		H2(__bitv$_{F=1/32}$)(orig)	
	comp-ben	speedup	comp-ben	speedup
Vbyte$_1$	1.63	2.76	1.39	1.20
PFD$_{X,256}$	1.48	2.75	1.72	1.16
S16$_{X,256}$	2.00	2.87	2.10	1.17

Table 4.6: Improvement for H2SK(__skip$_{X=256}$ bitv$_{F=1/8}$)(url) vs. the fastest *skips* and recommended *bitvectors* configurations using original ordering.

4.5 Conclusions

We have shown that *bitvectors* should be combined with large overlaid *skips* and that the documents should be in URL order to produce the best results in all situations. Thus, the designer of a system to intersect integer lists must first decide on a compression algorithm based on the compression rate and decoding speed. Before compressing the lists,

however, documents should be re-numbered into URL order [Silvestri 07]. Next, list indexes in the form of large overlaid *skips* need to be added to regain the lost random access speed from compression. Finally, extending the recommendation of Culpepper and Moffat [Culpepper 10] to other compression schemes, large lists should be stored as *bitvectors*.

The *bitvectors+skips* combination dominates the use of either *skips* or *bitvectors* separately for all the compression algorithms we examined, and under both document orders we have considered. We believe this property will hold for all compression algorithms and all document orders.

The conclusion is clear, conjunctive list intersection systems should *always* use a *hybrid bitvector* approach with large overlaid *skips* and order the documents appropriately. Effective use of these approaches in full search systems, however, requires additional investigation. Furthermore, since the best algorithms for list intersection in terms of space and runtime use *bitvectors* for large lists, future work in this area should remove large lists from consideration when comparing compression algorithms.

Chapter 5

Partial Bitvectors

We use the term *partial bitvectors* to refer to any list encoding approach that can store some portions of a list in a *bitvector* format with other portions of the list stored in another format. We develop our particular *partial bitvector* approach in two steps. First, we introduce a new *partial bitvector* encoding called *semi-bitvectors* and use this encoding to store grouped lists in terms-in-document ordering. After that, we show that ordering by URL within groups improves the performance of *semi-bitvectors* and outperforms the representations presented in the previous chapter.

5.1 Grouped Terms-in-Document Ordering

The URL and clustering based orderings place documents with similar terms close together, producing *tight clustering* within the postings lists. The terms-in-document ordering, however, does not place documents with similar terms together in the ordering. Instead, through ordering by decreasing number of terms-in-document, postings are packed into lower document identifiers, meaning that the density of the postings lists tends to decrease throughout the lists. This front-packing results in many small deltas, which can be easily compressed. The front-packing also means that values in the postings lists are denser for lower document identifiers than for higher ones, giving *skewed clustering* with more effective use of *skips* at the end of the lists. This skewing of postings to lower document identifiers can be clearly seen in the distribution of terms-in-document values, as shown in Figure 5.1. The dotted lines split the index into three groups containing equal numbers of postings, reflecting that the largest 10.9% of documents contain 33.3% of the postings.

We can exploit the skewed nature of terms-in-document ordering by using *partial bitvectors*. In particular, we use *bitvectors* for the denser front portion of a postings list, and then

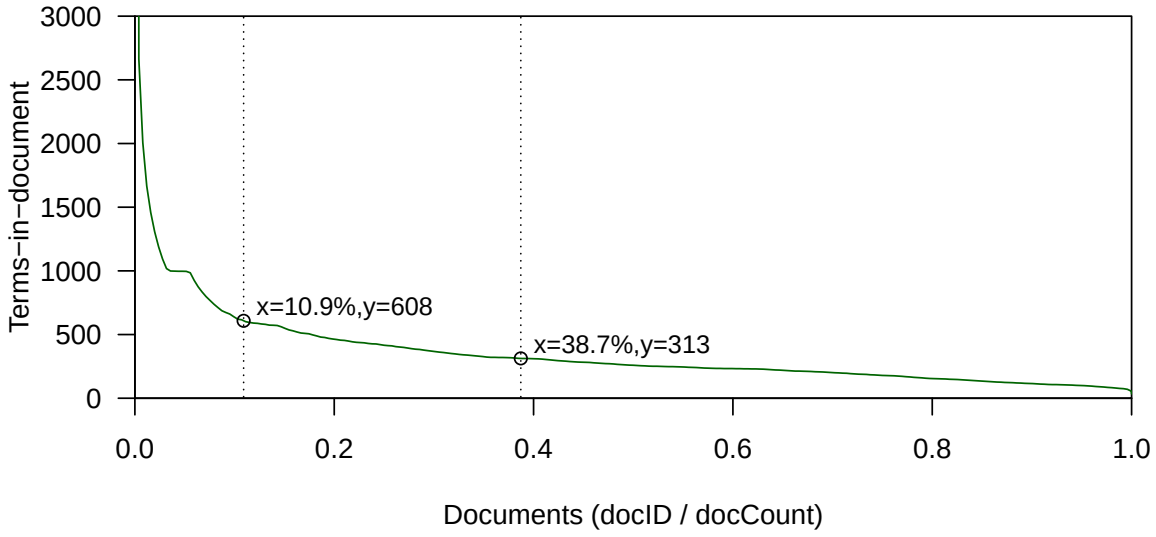

Figure 5.1: Terms-in-document distribution for *GOV2* with three groups marked.

normal delta compression and *skips* for the rest of the list. We call this front *partial bitvector* structure a *semi-bitvector*, and the highest document identifier in the *bitvector* portion of a postings list is called the *cut point*. The front *bitvector* portion of the *semi-bitvector* structure can be accessed quickly by document identifier, while the cost of encoding the single cut point is negligible. As a result, we believe that the *semi-bitvector* format is an efficient form of *partial bitvector*. An example layout of a postings list encoded using either *skips*, *bitvectors*, or *semi-bitvectors* is shown in Figure 5.2. In order to reduce the number of possible combinations of structures when intersecting lists, the *semi-bitvector* intersection algorithm must first order the lists ascending by their cut points, then execute in a pairwise set-versus-set manner. Each pairwise list intersection has three (possibly empty) parts that are executed in order: *bitvector*-to-*bitvector*, sequence-to-*bitvector*, and sequence-to-sequence. In general, the end result contains a *partial bitvector* that must be converted to values, followed by a sequence of values. The intersection of two *semi-bitvectors* is defined in Algorithm 11 using basic intersection subroutines acting on *bitvectors* and sequences of integers.

Our implementation of *semi-bitvector* intersection applies various optimizations: The *bvand* and *bvconvert* algorithms (lines 6 and 8) are executed in a single pass on the last intersection and the *bitvector* b is not created if the query contains only two lists. The restrictions on $M.seq$ applied by the select calls (lines 9 and 10) are executed as a single pass on M. Also, the two conditionals from the select call ($value < t$) and the loop through $M.seq$ (line 9) are combined when possible (i.e., first find the end point t in an

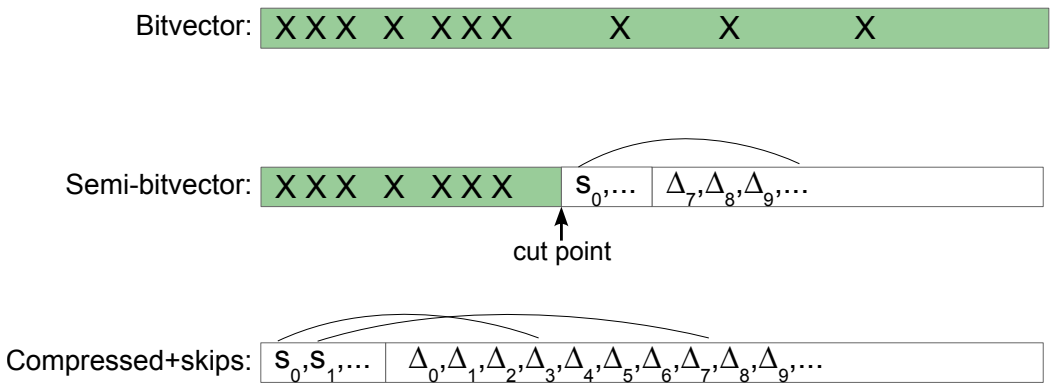

Figure 5.2: Layout of *semi-bitvectors* compared to *bitvectors* and *skips*.

Algorithm 11 *SBV* – Intersect *Semi-Bitvector*

```
1:  function SBV(M,N,isLastStep)                                  ▷ |M| ≤ |N|
2:      assert(M.isSemiBitvector), assert(N.isSemiBitvector)
3:      r ← {}
4:      s ← M.bitvSize
5:      t ← N.bitvSize                                            ▷ assert(s ≤ t)
6:      b ← bvand(M.bitv, N.bitv, s)                              ▷ bitv-to-bitv (see page 27)
7:      if isLastStep then
8:          r ← r ∪ bvconvert(b, s)                               ▷ (see page 28)
9:      r ← r ∪ bvcontains(M.seq.select(value < t), N.bitv, t)    ▷ seq-to-bitv (see page 28)
10:     r ← r ∪ merge(M.seq.select(value ≥ t), N.seq)             ▷ seq-to-seq (see page 15)
11:     if isLastStep then
12:         return r                                              ▷ list
13:     return new semi-bitvector(b, r, s)                        ▷ semi-bitvector
```

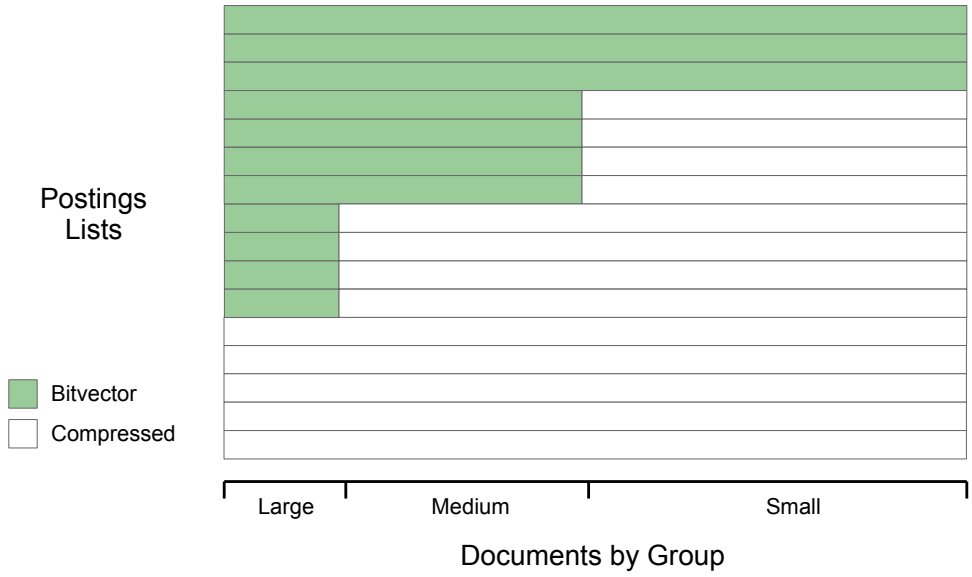

Figure 5.3: Schematic of a three-group *semi-bitvector* index, as defined in Figure 5.1 (three groups) and Figure 5.2 (*semi-bitvector* layout) with cut points aligned to group boundaries.

uncompressed sequence or the nearest skip point before t in a compressed sequence, then use that location as a single conditional check). The memory for the result set r can be reused between pairwise steps (i.e., as input from the last step and output of the current step), except on the final step if there is a *bitvector*-to-*bitvector* portion to include in the result (line 8). Note, while it is easier to write code that intersects the *bitvector*-to-*bitvector* portions for all the lists, converts that result to integers, then intersects the remainder of the lists, this approach produces non-sequential access into the lists and thus slower runtimes.

Our *semi-bitvectors* allow more postings to be stored in *bitvector* structures than *bitvectors+skips* for the same amount of memory used. Since the performance of *bitvectors* is much faster than other approaches, better use of *bitvectors* can produce a significant improvement in runtime performance, allowing the overall system to be more efficient.

We pick *semi-bitvector* cut points so that the *bitvector* portion of each list will have at least density F, similar to how the *hybrid bitvector* algorithm uses F as the density threshold for choosing *bitvectors*, as defined in Section 2.6.4. We make the cut point calculation efficient by splitting the document domain into groups and only allowing *semi-bitvectors* to have cut points at group boundaries. A schematic of a *semi-bitvector* index using three groups (and thus four potential cut points) with lists ordered by their cut points is shown in Figure 5.3. The cut point for a list is the highest group boundary where the group itself is above the density threshold F, and the *bitvector* portion (from the start of

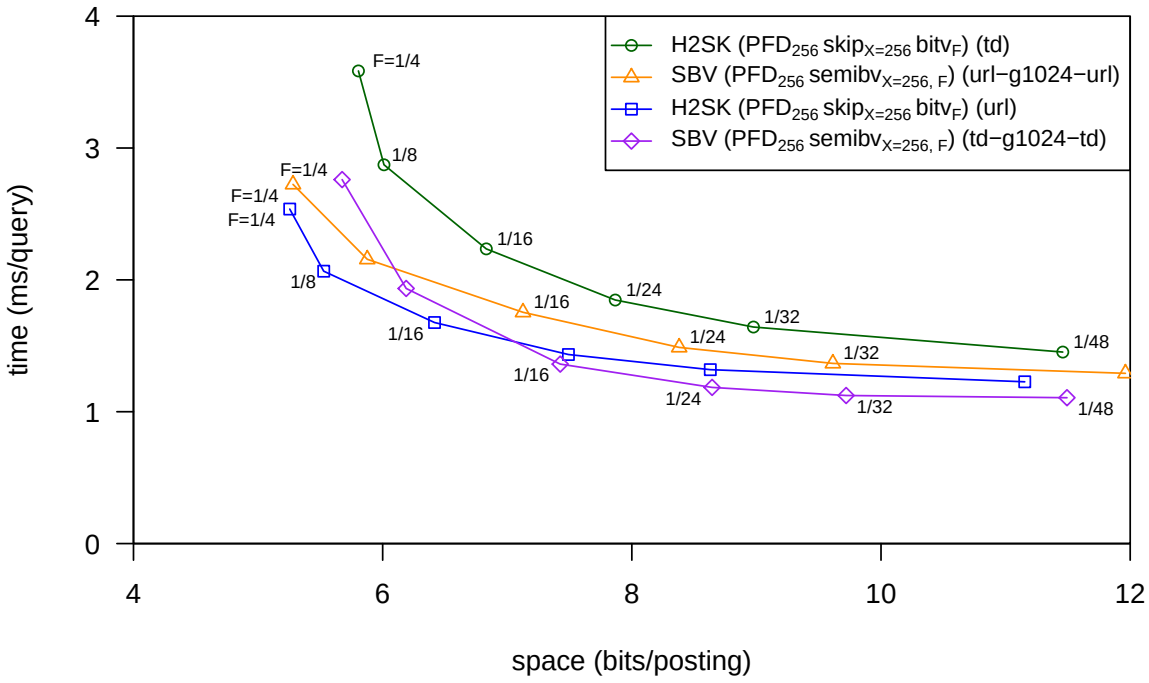

Figure 5.4: Space vs. time for *semi-bitvectors* using td-g1024-td ordering compared to *bitvectors+skips* using either URL or terms-in-document ordering.

the list to the end of the group) is also above F. A dense region in a higher group may thus force a lower group to be stored as a *bitvector* even if the lower group is below the density threshold F, meaning the groups are not independent of each other. This definition allows a large number of groups to be used without degrading the index with too few or too many *bitvector* regions. When *semi-bitvectors* are used with a single group, this approach results in the same choices of *bitvector* as the original *hybrid bitvector H2* algorithm.

The formation of groups within terms-in-document ordering does not change the order of the postings within the index, since the postings within each group remain in terms-in-document order. We use many groups to effectively exploit the skew and choose the group boundaries so that each group contains the same number of postings. (Other approaches could be used to determine the group boundaries, but they are not explored further here.) With this approach, we see a significant performance improvement from using *semi-bitvectors*. In order to demonstrate this performance improvement, we run *semi-bitvectors* for terms-in-document ordering using 1024 groups (td-g1024-td, i.e., order by terms-in-document, split into 1024 groups, keep terms-in-document ordering within each group) as shown in Figure 5.4. Our configuration using *semi-bitvectors* with td-g1024-td ordering dominates *bitvectors+skips* with terms-in-document ordering and performs even

better than our previously best URL based approach, *bitvectors+skips*, for large configurations, though smaller configurations are still slower and no configuration is as small as what can be achieved with URL ordering. Grouping URL ordering using 1024 groups gives a *semi-bitvector* configuration with url-g1024-url ordering, but this configuration degrades performance relative to *bitvectors+skips* using URL ordering.

5.2 Grouped URL Ordering

The terms-in-document ordering gives skewed clustering towards the front of the lists, while URL ordering and other approaches give tight clustering throughout the lists. We would like to combine these two types of ordering into a hybrid ordering to produce the benefits of both skewed clustering and tight clustering.

Terms-in-document ordering was previously combined with URL ordering by Ding et al. [Ding 10] in the form of url.server-td-url, which splits into chunks by url.server, then groups into five parts by terms-in-document, then orders by URL. This hybrid ordering gives slight benefits in terms of space, but the effect on runtime performance was not tested. While the method of determining group separations (i.e., the boundaries for each terms-in-document group) was not specified, the url.server portion of the hybrid ordering will split the index into many small pieces. As a result, the skew from the subsequent terms-in-document ordering is spread out across the entire document range. This means that the skew cannot be easily exploited through grouping as we did in Section 5.1.

For our hybrid combination of terms-in-document and URL ordering, we first group the documents by their terms-in-document value, then reorder within each group using URL ordering. We will refer to this approach as td-g#-url, where # is the number of groups. (Since the *GOV2* dataset covers only one section of the Web, we can relate our new grouping approach to the previous hybrid approach as being in the form of url.server.suffix-td-url.) Increasing the number of groups of documents will reduce the tight clustering from URL ordering but increase the skewed clustering of the data. This means that as the number of groups increases, the performance will trend towards the grouped terms-in-document performance and thus degrade "coherence." As a result, we cannot use a large number of groups, as we did previously with terms-in-document ordering, but must instead find a good choice for our workload and dataset. We now explore the space-time benefits of *semi-bitvectors* using the td-g#-url approach with various numbers of groups.

The td-g3-url ordering results in some skewing of document sizes (and thus postings) towards the front of the lists, as shown in Figure 5.5 (bottom). It also reduces the delta sizes as compared to URL ordering, with approximately 71.9% of the deltas having the value one for td-g3-url ordering compared to 69.8% for URL ordering. This means that space usage can be improved for our dataset when using our grouped td-g#-url approach.

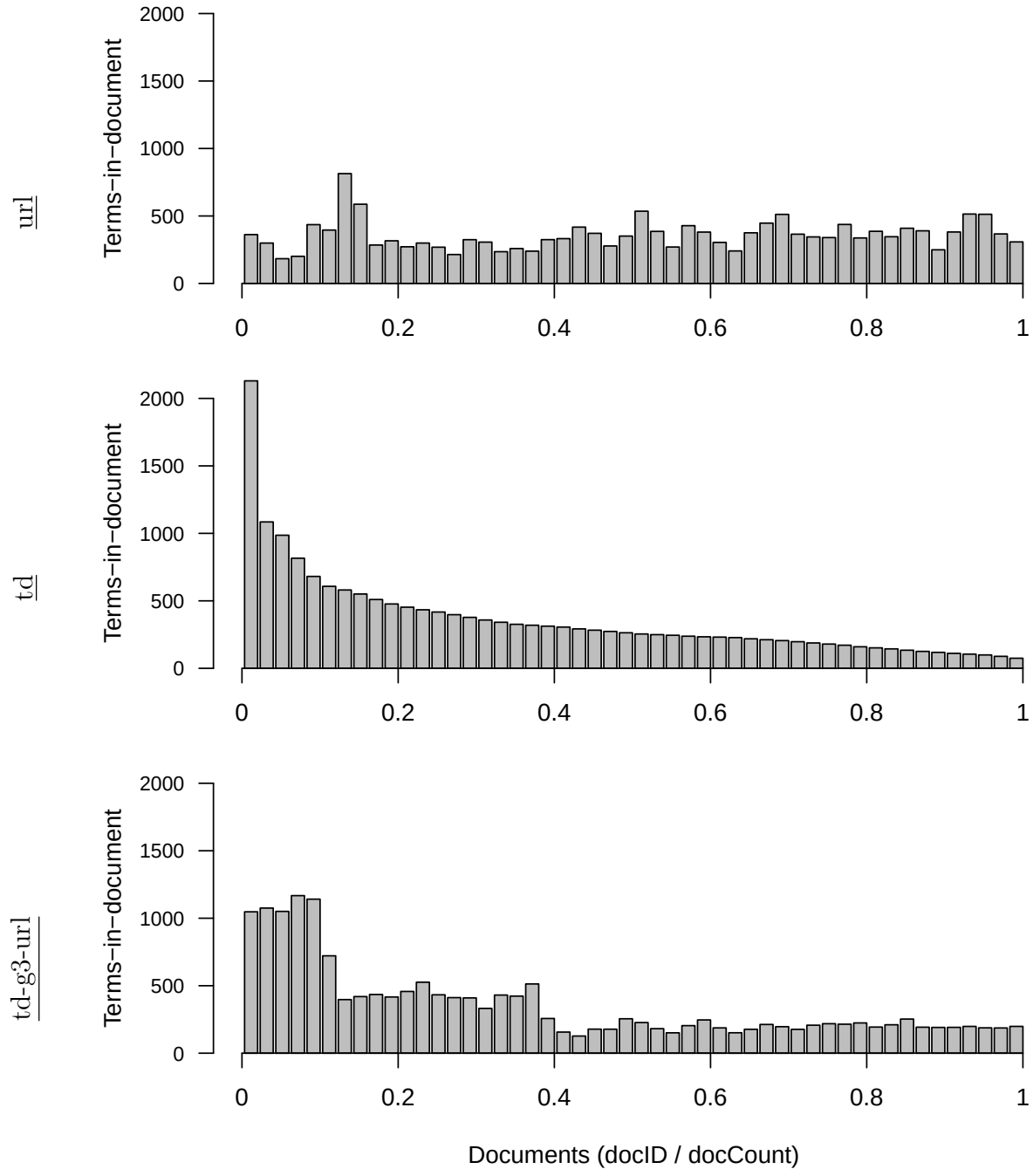

Figure 5.5: Average document sizes over ID ranges for various document orderings.

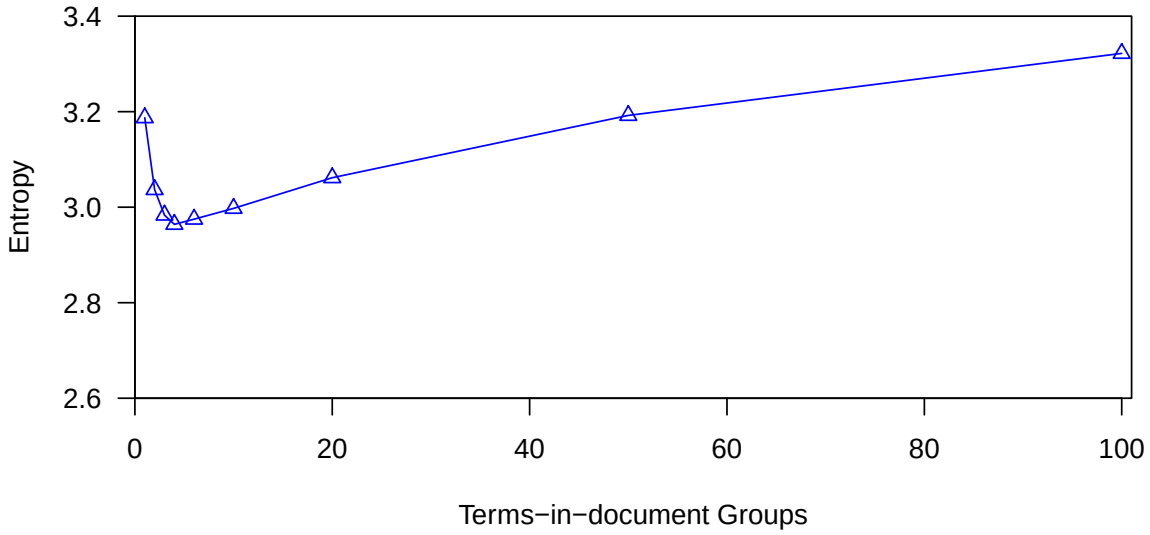

Figure 5.6: Entropy vs. number of terms-in-document groups using td-g#-url ordering.

To explore how the compressibility of our dataset changes over various numbers of groups, we use two measures, where d is a bag of delta values occurring in postings lists and D is the set of unique delta values. First, we measure the zero order Shannon entropy H (which assumes deltas are independent and generated with the same probability distribution), where p_i is the probability of a delta having the value i in the data:

$$H(d) = -\sum_{i \in D} p_i \log_2 p_i$$

Lower values of entropy indicate that more compression is possible. Since entropy deals with probabilities, the actual delta values are not important. Second, we measure the average $\log_2$ of the delta values, which we call the log-delta and denote L:

$$L(d) = \tfrac{1}{|d|} \sum_{d} \log_2 d = \sum_{i \in D} p_i \log_2 i$$

Lower values for the log-delta measure indicate that more compression is possible.

The entropy and log-delta values for the list of deltas found in our basic document orderings are shown in Table 5.1. The td-g#-url approach can improve entropy compared to URL ordering, as shown in Figure 5.6, where four groups gives the best entropy. Surprisingly, even with one hundred groups, the entropy does not significantly degrade, despite the fact that the entropy of the (pure) terms-in-document ordering is 5.07 and we expect that

splitting the index into many groups will degrade performance towards terms-in-document ordering. For the log-delta measure, the plot has a similar shape and four groups again gives the lowest measurement.

	rand	orig	td	url
entropy	7.24	6.71	5.07	3.19
log-delta	4.10	3.70	2.51	1.44

Table 5.1: Entropy and log-delta of the delta values for various document orderings.

The actual space-time performance for various numbers of groups and various F values is shown in Table 5.2. Using four groups produces the smallest configuration with a *bitvector* density cutoff of $F = \frac{1}{8}$, but for other F values, using eight groups is better than using four groups in both space and time. As a result, we use td-g8-url as our preferred configuration, though this configuration may not be the best when using different datasets or query workloads. The performance of td-g8-url is similar to the performance of configurations using between 4 and 16 groups, which suggests that the performance improvements from using *semi-bitvectors* are not particularly sensitive to our tuning of the number of groups.

	$F = \frac{1}{8}$		$F = \frac{1}{16}$		$F = \frac{1}{32}$	
	space	time	space	time	space	time
td-g2-url	5.32	1.77	6.28	1.33	8.11	1.08
td-g4-url	5.16	1.65	6.15	1.22	7.92	0.98
td-g8-url	5.23	1.56	6.14	1.17	7.79	0.96
td-g12-url	5.28	1.54	6.23	1.16	7.94	0.96
td-g16-url	5.34	1.55	6.28	1.15	7.98	0.96

Table 5.2: Detailed space (bits/posting) and runtime (ms/query) performance using various configurations of the SBV($\mathrm{PFD}_{256}\,\mathrm{semibv}_{X=256,\,F}$) algorithm.

When we compare our *semi-bitvector* approach using td-g8-url ordering to the *bitvectors+skips* algorithm using URL ordering, we see a significant improvement in performance. For the same amount of memory, the *semi-bitvectors* produce a speedup of at least 1.4x compared to the best URL ordering approach, and a small reduction in space usage is also possible, as shown in Figure 5.7. There is a performance improvement from using

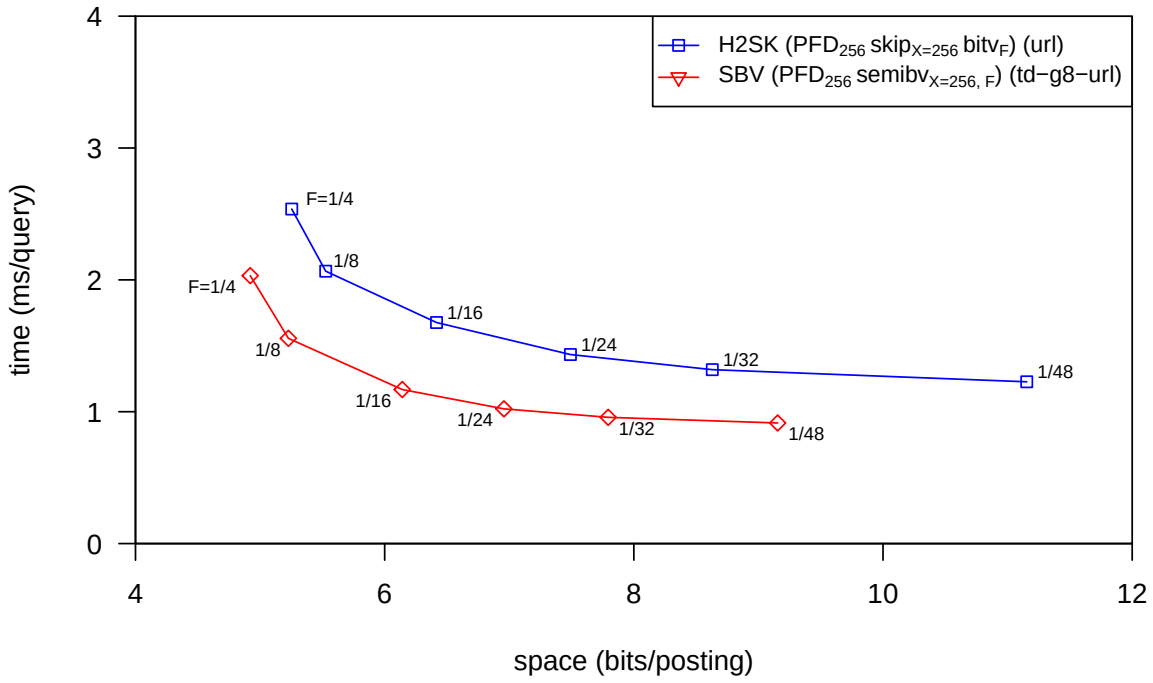

Figure 5.7: Space vs. time for *semi-bitvectors* using td-g8-url ordering compared to *bitvectors+skips* using URL ordering.

semi-bitvectors for all query lengths when using the $F = \frac{1}{32}$ configuration and the relative improvement increases with more terms per query, as shown in Figure 5.8. For the same space budget, the number of postings that can be placed in *bitvectors* with the *semi-bitvector* grouping approach is much higher than the original *hybrid bitvector* approach, as shown in Figure 5.9. Since *bitvectors* are much faster than *skips* over delta compressed structures, the more effective usage of *bitvectors* explains much of our performance improvement.

In order to verify that *semi-bitvectors* are needed to produce the performance improvements found in the td-g8-url *semi-bitvector* approach, we examine the td-g8-url ordering using the *bitvectors+skips* algorithm where *bitvectors* are used only for entire lists. The td-g8-url ordering with the *bitvectors+skips* algorithm produces a small space improvement, but no evidence of runtime improvement over URL ordering. Using fewer groups shows some runtime improvement compared to URL ordering, but they were small compared to the benefits of using *semi-bitvectors*.

The *semi-bitvector* implementation itself is more complicated than the *bitvectors+skips* implementation. As a result, running the *semi-bitvector* implementation without grouping

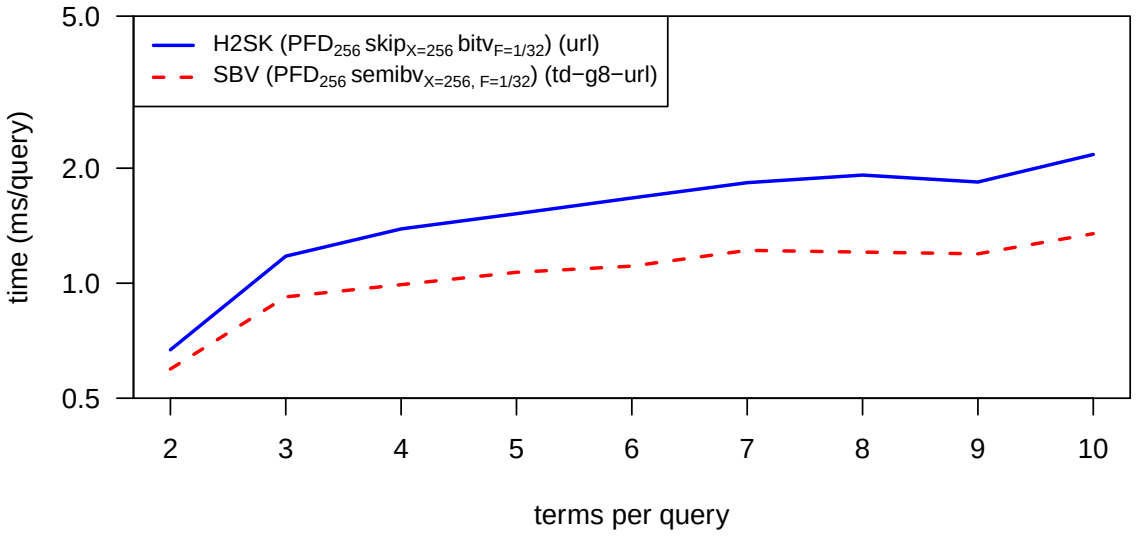

Figure 5.8: Runtime vs. terms per query using *skips+bitvectors* and *semi-bitvectors*.

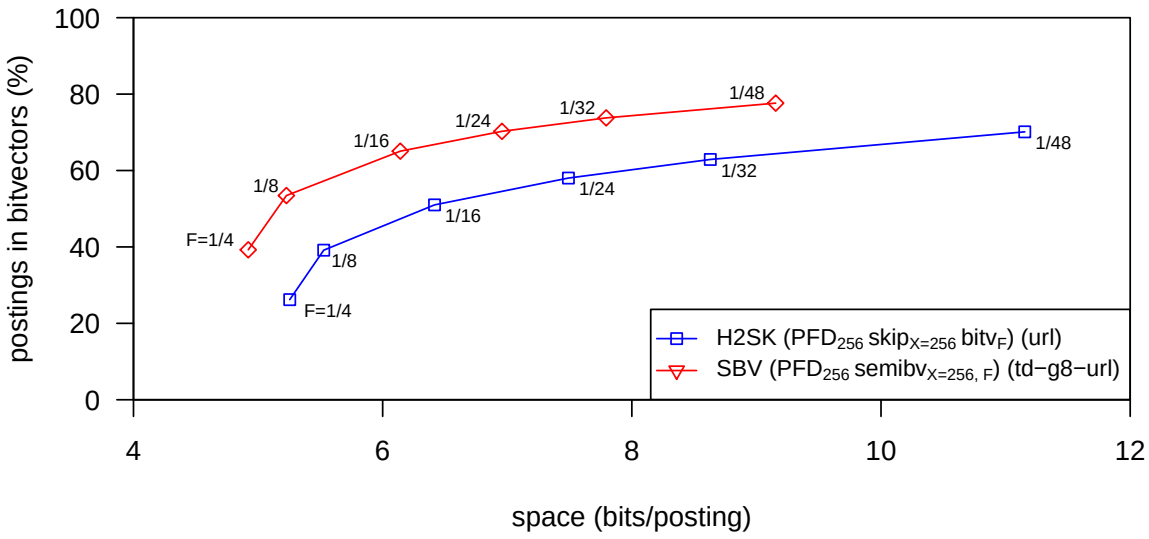

Figure 5.9: Space vs. postings in *bitvectors* for *semi-bitvectors* using td-g8-url.

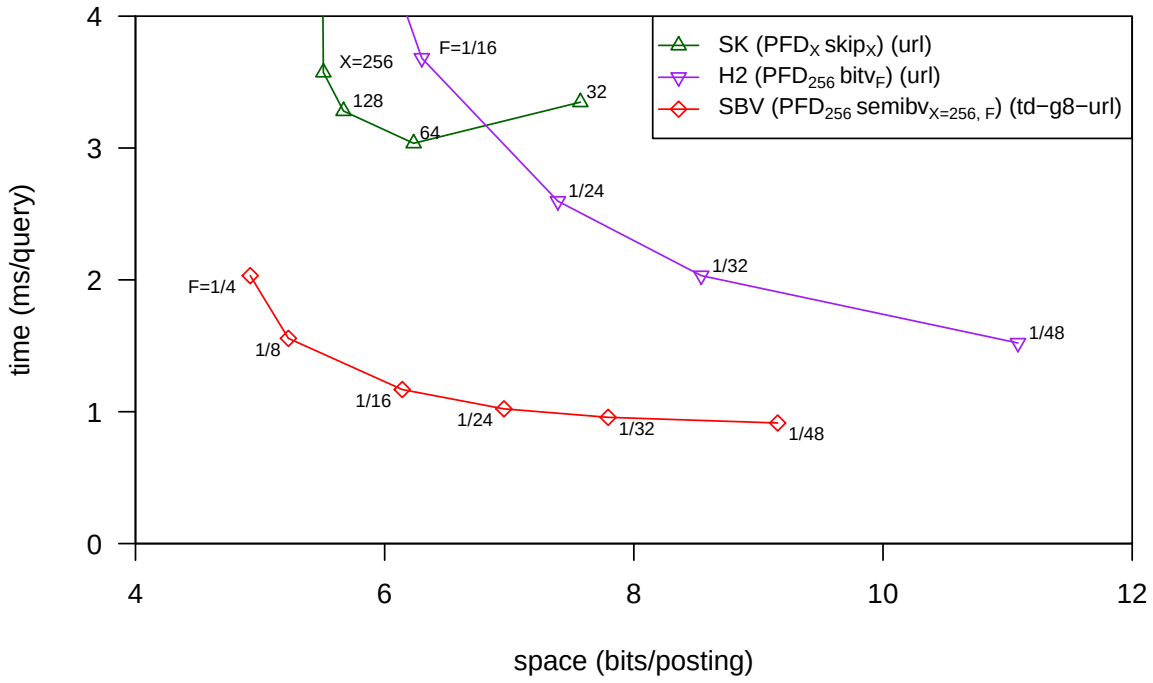

Figure 5.10: Space vs. time for *semi-bitvectors* using td-g8-url ordering compared to *skips* or *bitvectors* alone using URL ordering.

by terms-in-document (i.e., using a single group giving td-g1-url which is the same as url ordering) results in a small degradation in runtime performance. Clearly, both grouping and *semi-bitvectors* are needed to produce the performance improvements in the td-g8-url *semi-bitvector* approach.

In addition, the space-time benefits of *semi-bitvectors* for terms-in-document ordering (td-g1024-td vs. td) are similar to the benefits of *semi-bitvectors* for URL ordering (td-g8-url vs. url). This suggests that our td-g#-url approach is effective in combining the benefits of tight clustering found in URL ordering with the benefits of skewed clustering found in terms-in-document ordering.

The benefits of using *semi-bitvectors* are in addition to the benefits of combining *skips* with *bitvectors*. As a result, a more appropriate comparison is between our *semi-bitvector* implementation and using *skips* or *bitvectors* separately: we found a speedup of at least 2.4x compared to using *skips* and at least 1.6x compared to using *bitvectors*, as shown in Figure 5.10. This comparison also shows that a significant space improvement is possible over using *skips* by themselves. These benefits are in addition to the performance gains from using URL ordering rather than some other ordering. Such inefficient orderings may

be common in existing installations. Incredibly, our *semi-bitvector* approach has a speedup of at least 6.0x compared to *skips* using the original ordering, while using the same amount of space, although significant improvements to space are also possible.

Similar types of runtime improvements would occur with any compression algorithm, since the benefits come from using *bitvectors* in dense regions where they are much faster than any compression algorithm.

5.3 Conclusions

We have shown how groups of documents defined by the skewed terms-in-document ordering, when combined with URL ordering and *partial bitvectors*, can be used to make list intersection more efficient. This is accomplished by forming varying densities within grouped portions of the postings lists, reordering within the groups by URL ordering, and then storing them as *semi-bitvectors*, which encode dense front portions of the lists as *bitvectors*. As a result, we can store more postings in *bitvectors* for a given space budget, and *bitvectors* are much faster than other approaches. This combination gives most of the benefits of tight clustering in URL ordering, while also gaining the benefits of skewed clustering for effective use of *semi-bitvectors*.

This multi-ordered configuration (td-g#-url) gives significant space-time improvements, when combined with *semi-bitvectors*. When compared to a fast and compact configuration that combines *bitvectors*, large overlaid *skips* and URL ordering, we get a speedup of at least 1.4x. When compared to using only *skips* or *bitvectors* with URL ordering, we get a speedup of at least 2.4x compared to *skips* and at least 1.6x compared to *bitvectors*. Although the overall improvement will depend on the size and type of the data, as well as the number of groups used, we expect significant benefits for most large datasets.

Chapter 6

Partitioning by Document Size

In this chapter, we propose that document distribution be done by partitioning the collection based on document size, rather than at random. Our document size distribution causes skew in the location of postings within the document identifier domain, which gives benefits that reduce the overall index size and improve query throughput. This skew is identical to that generated from grouping by document size, but the independence of partitions allows it to be more easily and more fully exploited. The drawbacks over simple grouping within a single index include the potential need to do load balancing for the partitions and the introduction of space-time overheads from duplicate dictionary entries and per-partition query setup times. These overheads also occur with other partitioning approaches, but a larger number of partitions than what is usual practice may be needed to exploit the benefits of skewing.

Partitioning at random (i.e., placing each document into a randomly selected partition) automatically produces good load balancing in general, but other types of partitioning need to be actively load balanced and may still produce unbalanced dynamic loads. Our experiments do not attempt to produce good load balancing, but we do examine the per-partition bottlenecks of the system to understand how the partitioning approaches affect query latency and throughput. We expect that the load balancing of our partitioning by document size and partitioning by URL approaches can be improved by adjusting the range of document sizes that are assigned to each partition, but we leave exploration of such optimizations for future work.

We demonstrate space (i.e., total index size) and time (i.e., latency and throughput) improvements for in-memory list intersection when the partitions use *PFD* compression combined with *skips* and with *bitvectors+skips*. We present various causes for the space-time benefits of document size partitioning by analyzing the performance details from individual partitions. These benefits occur without any changes to the algorithm code.

Finally, we explore the various methods to combine partitioning and grouping for our *GOV2* index using *semi-bitvectors*, which may result in benefits over the use of partitioning with *bitvectors+skips*.

6.1 Terms-in-Document Distribution

We can use the terms-in-document measure to distribute documents into partitions, allowing for the documents to be ordered in a potentially better way within the partitions, but still gaining some of the benefits found with terms-in-document ordering. Such a distribution by document size can be used with any underlying ordering, compression, intersection, or ranking approach within the partitions, so it is broadly applicable. In addition, many benefits of document size distribution could also be gained by splitting a single partition into multiple document sized partitions running on a single machine.

An additional benefit occurs when our document size distribution is used with *bitvectors*. This benefit is not available with simple terms-in-document ordering, so it was not discussed in previous work. Document distribution allows each partition to decide independently which lists to store as *bitvectors*, determined by the list density within the partition. With a random distribution, this decision is fairly consistent because the list densities are likely to be consistent between partitions. For URL based distribution, a small amount of skew in density is produced that can be exploited. In our document size distribution, however, the list densities are highly skewed between partitions, meaning that *bitvectors* can be used much more efficiently than in other settings.

Partitions with large documents have dense lists and use many *bitvectors*, while partitions with small documents have less dense lists and use fewer *bitvectors*. The choice of whether to use a *bitvector* for each term is independent for every partition. This is illustrated in Figure 6.1, where F is the density threshold for using *bitvectors* and the partitioned postings list 'A' demonstrates that the choice to use *bitvectors* is independent in each partition. As a result, the use of partitions allows more actual postings to be stored as *bitvectors*, which improves query runtime since *bitvectors* are much faster than *skips*.

Partitioning by terms-in-document with the *bitvectors+skips* algorithm is very similar to grouping by terms-in-document with *semi-bitvectors* from Chapter 5, but groups are not independent. Using *semi-bitvectors*, each term has only one cutoff point with *bitvectors* used below it and *skips* with delta compression used above it, but the partitions that use *bitvectors* for their fragments of a given postings list need not all be for the largest documents. As a result, partitioning can more effectively use *bitvectors* as compared to grouping when using the same number of partitions and groups. For example, when considering the 'A' list shown in Figure 6.1, the *semi-bitvector* encoding would either use a *bitvector* for

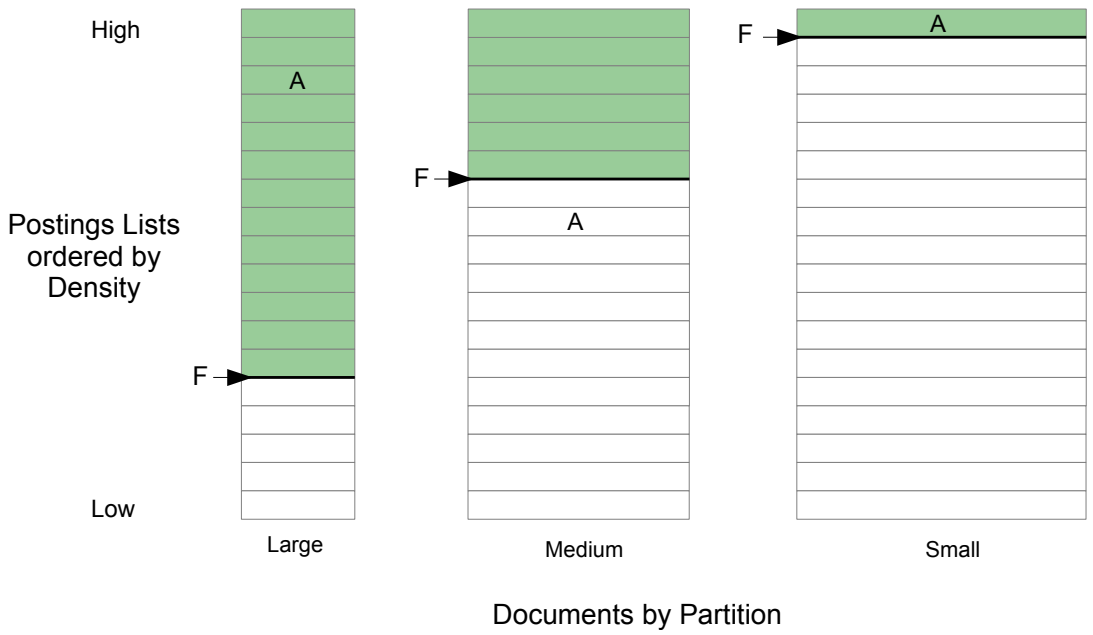

Figure 6.1: Schematic of a terms-in-document three partition *bitvector* index. This is similar to using the three groups and *semi-bitvectors* shown in Figure 5.3.

the entire list or for just the large documents, while the partitioned version uses *bitvectors* for both the large and small documents, but not the medium sized documents.

6.2 Implications for Indexing

We implemented document size distribution (and URL distribution) to place an equal number of postings in each partition. While this does not always produce good load balancing, it is sufficient to demonstrate the benefits available from document size distribution. Placing an equal number of postings in each partition requires knowing the number of postings in each document before deciding which partition should own the document. If the search system does not have a shared storage mechanism, such as a central disk system, then determining the number of postings in each document before distribution may require redistribution of index data among the partitions. With random distribution, the documents are sent to the partitions and indexed locally, meaning the *convert* through *encode* steps in Figure 2.1 happen within each partition. For document size distribution and URL distribution, the documents are first distributed randomly, then the *convert* and *tokenize* steps are executed on each partition's local documents. The document size (and URL information

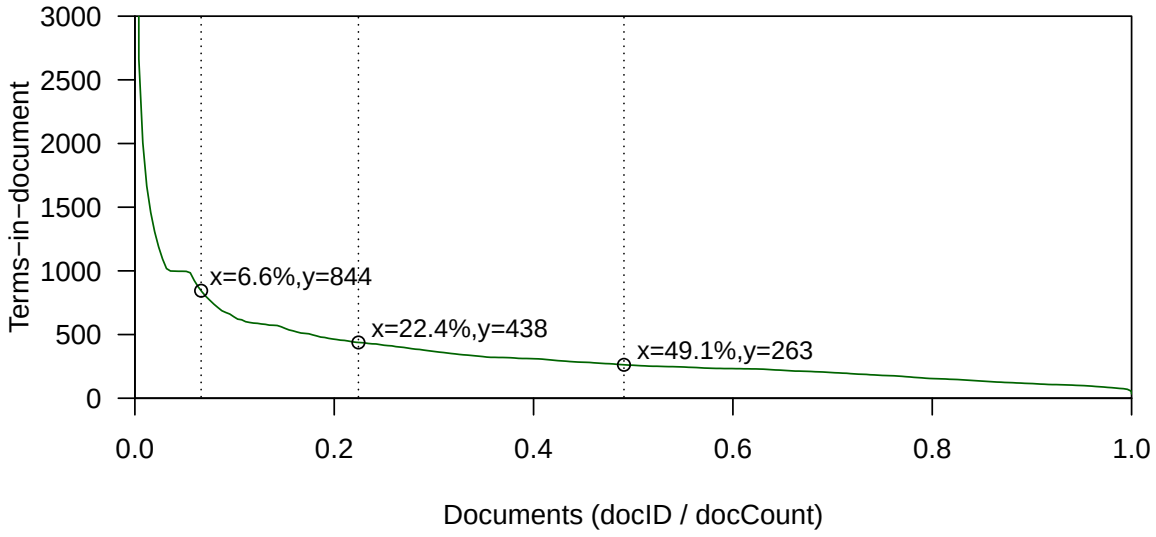

Figure 6.2: Terms-in-document distribution for *GOV2* with four partitions marked.

for URL distribution) is then sorted globally to determine the repartition location for each document. The tokenized form of the documents is then sent to the appropriate partitions, where the *invert* and *encode* steps are executed to produce the desired distributed index. The new *sort* and *repartition* steps in this process are easy to implement, but they will result in a small slow down in indexing speed.

One way to avoid some of the extra processing involved in producing an index distributed by documents size or URL is to approximate the distribution by statically determining which partition owns each document. The document size distribution could be approximated by using static ranges of document sizes, with cutoff points determined using information from a representative set of data (perhaps a subset of the data or a previous version of the data). For example, creating four partitions using our *GOV2* dataset, as shown in Figure 6.2, determines cutoff points at 844, 438 and 263 postings per document, which can be used as approximate cutoff points for some related dataset or another version of the *GOV2* dataset. Using document size cutoffs would avoid the *sort* step, but not the *repartition* step of the indexing pipeline. To approximate our URL distribution, static ranges of URL values or static mappings of URL values can be used to assign documents to partitions. For example, a prefix of the URL values could be hashed to determine the partition that owns the document. This URL prefix hashing is a fast operation that does not require any document tokenization information, so it avoids both the *sort* and *repartition* steps of the indexing pipeline.

Adding new documents to an existing index that is distributed by document size or

URL may require dropping the requirement for an equal number of postings in each document, moving documents between partitions, approximating the distribution approach, or reindexing.

6.3 Skew Using Four Partitions

In our experiments, we choose to use four partitions, since using four groups gave rise to the smallest entropy and log-delta values in Chapter 5. To show the benefits of document size distribution, we examine five configurations: The first two configurations examine the benefits to a system that does not exploit document ordering, by using random ordering within each partition. The first configuration uses four random partitions and orders documents using a random order (rand-p4-rand) and the second configuration partitions by terms-in-document and orders by a random order within each partition (td-p4-rand). These two configurations demonstrate that our improvements are not confounded by another ordering within the partitions and that the benefits are possible for any ordering within the partitions. The last three configurations use URL ordering within each partition, and the index is partitioned using a random order (rand-p4-url), the URL order (url-p4-url), or the terms-in-document order (td-p4-url). These three configurations demonstrate our improvements over a top performing approach to list intersection.

Given the random, URL and term-in-document partitioning approaches, we expect random partitions to have similar document counts, URL partitions to have a small skew, and terms-in-document partitions to have a large skew in document counts. Indeed, our experiments using four partitions have the expected differences in amount of skew between the random, URL, and terms-in-document partitioning approaches, as shown in Table 6.1. Note: we order the names for the random partitions according to their document counts and maintain this ordering in subsequent tables, while for URL partitioning, we maintains the lexicographic ordering of the partitions, and for terms-in-document partitioning, we maintains the document size ordering of the partitions.

	rand-p4	url-p4	td-p4
Partition 1	24.6	24.4	6.6
Partition 2	24.8	28.0	15.8
Partition 3	25.2	24.3	26.6
Partition 4	25.5	23.3	50.9

Table 6.1: Portion of documents (%) in each partition.

Partitioning by the highly skewed terms-in-document measure gives partitions and individual list densities that are highly skewed towards the large document partitions. These

denser large document partitions produce smaller deltas and better compression. In addition, they are more efficient for list intersection through better locality of memory access and shared decoding work for multiple candidate documents, resulting in improved runtime performance for both *skips* and *bitvectors*. In the small document partitions, the less dense lists allow *skips* to work more effectively.

The skew in terms-in-document sizes for the *GOV2* dataset does result in a skew of the individual postings lists used. This skew in postings lists using td-p4 is shown in Table 6.2 by the per-partition differences in the average density of the smallest lists in our workload of queries. In addition, we find that the url-p4 partitions have some skew and the rand-p4 partitions have very little skew.

	rand-p4		url-p4		td-p4	
	portion	density	portion	density	portion	density
Partition 1	25.3	0.76	17.3	0.45	26.9	2.65
Partition 2	25.1	0.75	22.7	0.52	32.7	1.36
Partition 3	24.8	0.73	29.8	0.78	27.1	0.67
Partition 4	24.7	0.72	30.3	0.83	13.3	0.17

Table 6.2: Portion (%) of smallest list and average density (%) of smallest list per query in each partition (full index density = 0.74%).

For conjunctive list intersection systems, combining skewed postings lists gives *result lists* that are even more skewed. This increased skew in result lists occurs because the likelihood of a document occurring in the intersection of multiple lists increases as the number of lists containing the document increases, which is exactly the number of terms in the document. Increased result list skew does not improve the compressibility of the data, but it does compound the benefits of locality of access and the effectiveness of *skips*.

Similar to ordering by document size, the dataset can be ordered by the document usage rate as measured by the number of times a document occurs in the postings lists or result lists for the set of random training queries [Garcia 04]. We found that ordering by document usage in the query term lists or query result lists produces similar performance to terms-in-document ordering, so we do not examine ordering by usage in more detail.

We find that the conjunctive result lists for the td-p4 partitions are highly skewed: the first partition contains 45.4% of the results, even though it contains a quarter of the postings and only 6.6% of the documents, while the last partition contains 6.7% of the results, a quarter of the postings and 50.9% of the documents. The other sizes for td-p4 can be found in Tables 6.1 and 6.3. Those tables also show that the result sizes for the url-p4 partitions have a small amount of skew and the rand-p4 partitions contain no such skew.

	rand-p4		url-p4		td-p4	
	portion	density	portion	density	portion	density
Partition 1	25.5	0.102	23.1	0.093	45.4	0.669
Partition 2	25.3	0.100	20.1	0.070	28.1	0.175
Partition 3	24.8	0.096	26.9	0.108	19.8	0.073
Partition 4	24.4	0.094	29.9	0.126	6.7	0.013

Table 6.3: Portion (%) of results and average density (%) of results per query in each partition (full index density = 0.098%).

We find that the result lists are indeed more highly skewed than the smallest list sizes for the td-p4 partitions, as shown by comparing the relative density values in Table 6.2 to those in Table 6.3. With td-p4 partitioning, the portion of the smallest lists returned in the result lists drops from 25.3% in the first partition (density ratio is approximately 0.669/2.65) down to 7.5% in the fourth partition (density ratio is approximately 0.013/0.17).

We have demonstrated that document size distribution produces skew in the postings lists of our *GOV2* dataset, and even more skew in the conjunctive query result lists for our workload of queries. In the next section we demonstrate how using td-p4 partitioning in our conjunctive list intersection system exploits these types of skew to give five significant benefits, namely: reduced smallest list size, compressible data, locality of access, effective *skips*, and effective *bitvectors*.

6.4 Benefits of Document Size Distribution

In this section, we compare the performance of random document distribution (rand-p4), URL distribution (url-p4) and document size distribution (td-p4) for a conjunctive list intersection system in order to validate the benefits of document size distribution. We measure index space by the sum of the per-partition space. We use three metrics (as detailed in the following paragraphs) to compare the runtimes of our partitioned systems. Our results are, therefore, presented on three separate space-time graphs corresponding to the summed space and the three runtime metrics.

To compare the performance of four partitions with our previous results running the entire index in a single partition, we sum the runtimes from each partition and assume the unaccounted for per-partition overheads, such as dictionary lookup times, are negligible. This *sum of partition times* metric is the the amount of work required to process a query (the total resource usage) on all the partitions and can thus be used to represent a configuration where all the partitions are run on the same machine. The total resource usage

can also be used to give an upper bound on the system throughput by assuming the ideal case where perfect load balancing is achieved.

When the partitions are run on separate machines, the slowest partition becomes the bottleneck. As a result, our *max partition time* metric is the average query runtime on the partition that incurs the maximum total query runtime. Comparing this max partition time to the ideal load balancing case (i.e., dividing the sum of partition times by the number of partitions) gives an indication of the achieved load balancing. Improving this load balancing by adapting the partitioning is, however, left for future work.

The slowest partition might not be the slowest for every query, so the partitoned query runtime is calculated as the runtime of the slowest partition for that query. We expect that most of the queries in a search system must meet a maximum query runtime requirement, often expressed as a *service level agreement* (*SLA*). To represent such an SLA, we take all the partitioned query runtimes and present the 99^{th} *percentile time* as a metric. We expect that other percentiles and other SLA requirements will behave similarly when the proposed partitioning approaches are used.

B1: Reduced smallest list size: Our conjunctive list intersection algorithm first orders the query lists by size and then intersects them iteratively in pairs from smallest to largest. This ordering of query lists is done independently in each partition, so variances in list density between partitions can cause the order to be different between partitions. As a result, density variances can reduce the overall size of the smallest lists used in the query. (The "size of the smallest list" in a partitioned system is the sum of the sizes of the smallest lists used in each partition.) Reducing the smallest list size will reduce the number of searches involved in intersecting subsequent lists, thus making the intersection run faster. The random distribution into four partitions produces almost no change in the smallest list size compared to using a single partition for the entire index, as shown in Table 6.4. Both terms-in-document distribution and URL distribution using four partitions result in significant reductions in overall smallest list size. When compared to the random distribution using four partitions, terms-in-document distribution produces an 11.3% reduction and URL distribution produces a 13.3% reduction in overall smallest list size (all differences are statistically significant as measured by paired t-tests producing very low p-values).

	p1	rand-p4	url-p4	td-p4
full index	186,070	186,040	161,221	164,978

Table 6.4: Average size of smallest list per query over entire index.

B2: Compressible data: Document size distribution produces density skew between partitions, giving smaller and more uniform deltas and better compression. We measure the compressibility of the deltas in each partition by calculating the entropy and log-delta

measures (see Section 5.2), where lower values indicate the deltas are more compressible. As shown in Table 6.5, the terms-in-document partitioning (td-p4) has a lower average entropy, lower average log-delta measure, and is more compressible than random partitioning (rand-p4) when using random ordering within partitions, although it is naturally not as compact as when ordering by terms-in-document within a single partition. Indeed, the actual storage costs using *PFD* compression with block sizes of 256 improves for terms-in-document partitioning (td-p4) with random ordering.

(*order*)	rand			td
	p1	rand-p4	td-p4	p1
entropy	7.24	7.21	6.39	5.07
log-delta	4.10	4.09	3.44	2.51
PFD_{256}	7.37	7.47	6.69	5.86

Table 6.5: Space, entropy, and log-delta values (bits/postings) using partitioning and random ordering.

The terms-in-document partitioning (td-p4) also has lower average entropy, lower average log-delta measure, and is more compressible than random partitioning (rand-p4) when using URL ordering within the partitioning, as shown in Table 6.6. The average entropy and average log-delta measures for URL partitioning (url-p4) are between the corresponding values for terms-in-document and random partitioning. The actual storage costs using *PFD* compression with block of size 256 gives similar results.

(*order*)	url				td
	p1	rand-p4	url-p4	td-p4	p1
entropy	3.19	3.52	3.14	2.92	5.07
log-delta	1.44	1.65	1.43	1.31	2.51
PFD_{256}	5.26	5.67	5.26	5.02	5.86

Table 6.6: Space, entropy, and log-delta values (bits/postings) using partitioning and url ordering.

These space measurements are from a system that stores only the deltas as postings, but for systems that also store term frequencies in the postings we expect that the frequencies will be more compressible using terms-in-document partitioning compared to random or URL partitioning. The frequency values for terms occurring in a document should increase as the document gets larger, which will correlate with our terms-in-document measure.

This reduces the variance of frequency values within each partition, which makes them more compressible.

B3: Locality of Access: The partitions with large documents should get better locality of memory access and more blocks containing multiple values, thus sharing decoding costs. Clearly, the partition with large documents (partition 1 for td-p4) has a much higher density of results, since the number of results is high and the number of documents is low. The increased density of results gives more efficient processing of queries, since the average runtime per result returned is much lower than for other partitions, as shown in Table 6.7 when using either random ordering or URL ordering within the partitions. This efficiency suggests that the density of results has indeed given us better locality of access. Note, the small average runtime per result for the first partition using URL partitioning (url-p4) is caused by very tight clustering within this partition, which also makes this partition much more compressible as shown by an entropy of 2.14 compared to an average of 3.14 for all of the url-p4 partitions.

(*order*)	rand		url		
	rand-p4	td-p4	rand-p4	url-p4	td-p4
Partition 1	251	152	135	82	101
Partition 2	250	224	133	128	106
Partition 3	251	258	131	122	111
Partition 4	254	366	132	132	160

Table 6.7: Runtime (ns) per result for $\mathrm{SK}(\mathrm{PFD}_{64}\,\mathrm{skip}_{X=64})$ using partitioning.

B4: Effective *skips*: The partitions with small documents should get an increased benefit from *skips*. Indeed, the partition of small documents (partition 4 in td-p4) has the fewest results relative to the total list sizes for our query workload. As a result, *skips* are extremely useful in this partition. To directly demonstrate the usefulness of *skips*, we calculate the

(*order*)	rand		url		
	rand-p4	td-p4	rand-p4	url-p4	td-p4
Partition 1	0.43	0.91	0.23	0.13	0.61
Partition 2	0.43	0.46	0.23	0.17	0.22
Partition 3	0.42	0.27	0.22	0.20	0.12
Partition 4	0.41	0.13	0.22	0.28	0.06

Table 6.8: Runtime (ns) per element in all but the smallest query list for $\mathrm{SK}(\mathrm{PFD}_{64}\,\mathrm{skip}_{X=64})$ using partitioning.

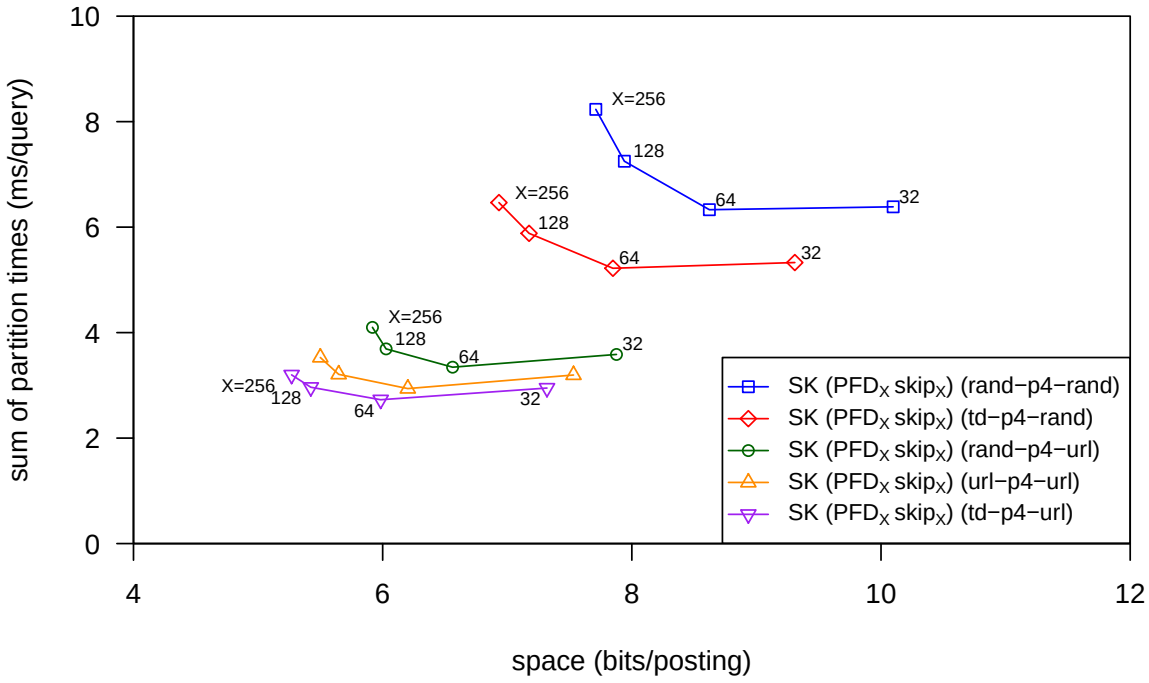

Figure 6.3: Space vs. summation of runtimes using *skips* and partitioning.

amount of runtime per element in the non-smallest lists (i.e., the lists where elements can be skipped over). As shown in Table 6.8, the partition containing only small documents uses the least runtime per skippable element and is thus more dependent on the speed of skipping.

When the first four benefits are combined for the *PFD+skips* algorithm, there is a significant improvement in space from using term-in-document partitioning. In addition, the runtime performance improves for search systems that place all the partitions on the same machine, as shown by the reduction in the overall amount of work required to process the queries (i.e., the summation of the partition runtimes) in Figure 6.3. This space-time improvement occurs with both random ordering and URL ordering, so we expect it to occur when using any other ordering. With URL partitioning, the performance is between the results of random and terms-in-document partitioning. Since URL partitioning with URL ordering is equivalent to running the index in a single partition using URL ordering, the random partitioning shows a degradation in performance. This degradation is related to the reduction of the tight clustering found in the URL ordering, and it will get worse as the number of partitions increases, but even four partitions is enough to see evidence of the degradation for *skips*.

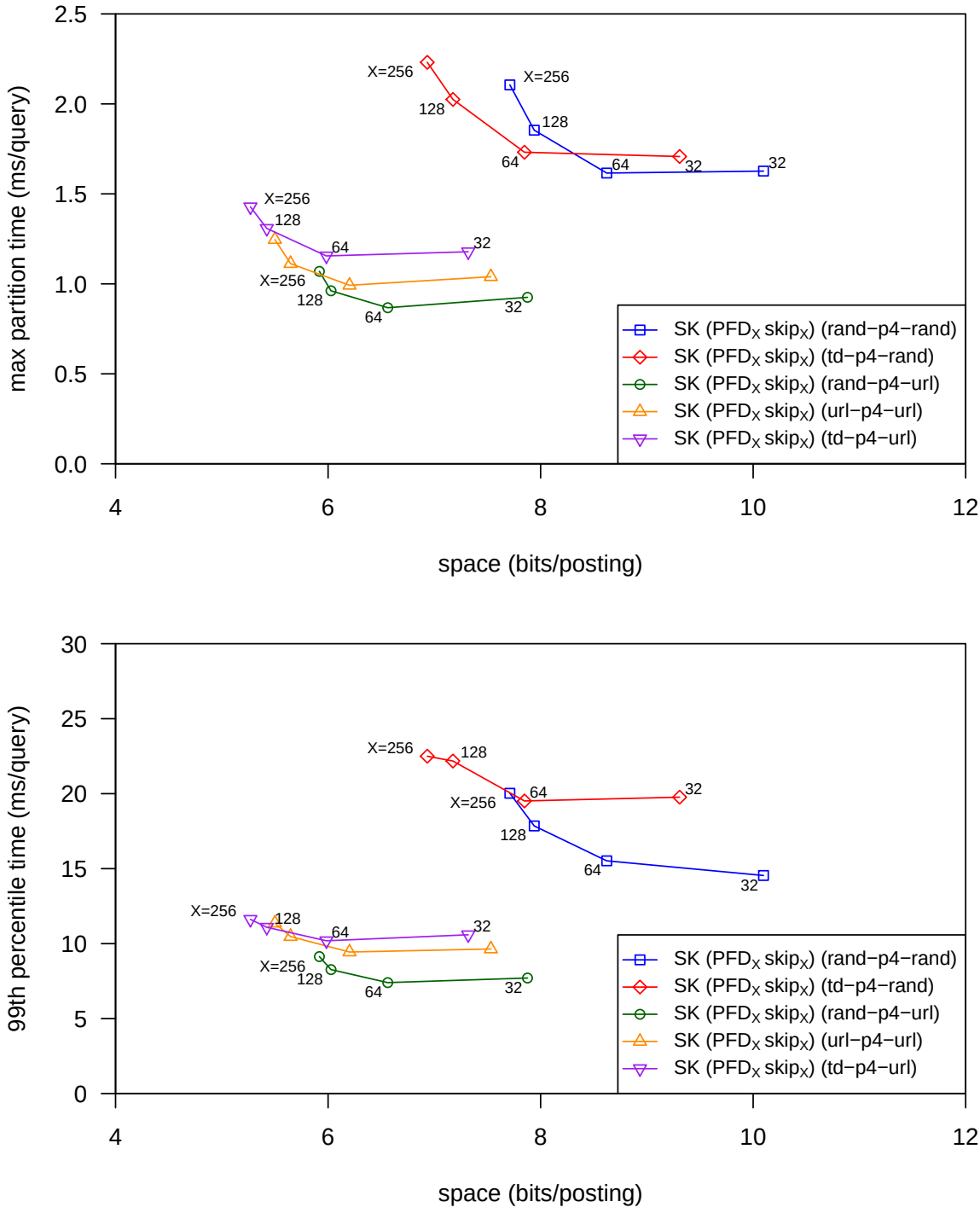

Figure 6.4: Space vs. maximum runtime using *skips* and partitioning.

Most of the benefits from using four terms-in-document partitions for *skips* are also found when using a single partition with four terms-in-document groups. The small performance difference between these two is accounted for by the independent selection of the smallest list size as discussed in B1.

When the partitions are run on separate machines where load balancing affects performance, the runtime performance of using terms-in-document (or URL) partitioning with *skips* degrades. Both the runtime performance of the slowest partition to execute all queries (i.e., the maximum partition runtime) and the runtime of slow queries (i.e., 99^{th} percentile query runtime) degrade, as shown in Figure 6.4. It is possible that moving the cutoff points for determining which documents are assigned to each partition may be able to improve the load balancing enough to exploit the reduction in total processing work required to process queries when using terms-in-document (or URL) partitioning, but we leave this exploration for future work.

B5: Effective *bitvectors*: The independent decision of whether to use *bitvectors* within each partition results in a significant space-time improvement on a single machine for td-p4 partitioning compared to rand-p4 or url-p4 partitioning, even when using random ordering, as shown in Figure 6.5. In fact, the performance of *PFD* compression, *bitvectors+skips*, td-p4 partitioning, and random ordering is close to the performance using url-p4 partitioning and URL ordering for the larger configurations with many *bitvectors*. This means that for large configurations, the skewed clustering in terms-in-document ordering exploited by partitioning and *bitvectors* is more important than the tight clustering in URL ordering. The effectiveness of using *bitvectors* compared to tight clustering may help explain why there was limited performance degradation when using a large number of terms-in-document groups in Chapter 5.

The space-time performance on a single machine degrades for random partitioning compared to URL partitioning when using *PFD* compression, *bitvectors+skips*, and URL ordering, as shown in Figure 6.5. As with *skips*, this degradation results from the reduction in tight clustering, and it will get worse as more partitions are added. The terms-in-document partitioning, however, produces a significant improvement over URL partitioning when using *bitvectors+skips*. This improvement is similar to the gain when using groups and *semi-bitvectors* compared to *bitvectors+skips* as described in Chapter 5. Much of this performance improvement comes from storing more postings in *bitvectors* for the same space budget, as shown for td-p4 partitioning and *bitvectors+skips* in Figure 6.6. Having more *bitvectors* in partitions with large documents also compounds the benefits from locality of access and increased density of results, since more of the query processing is done using *bitvectors*. The number of postings stored in *bitvectors* using four terms-in-document partitions (td-p4) is only slightly higher than using four terms-in-document groups with *semi-bitvectors* (td-g4), though these values may diverge as the number of partitions/groups increase. Performance could be improved by using the query workload

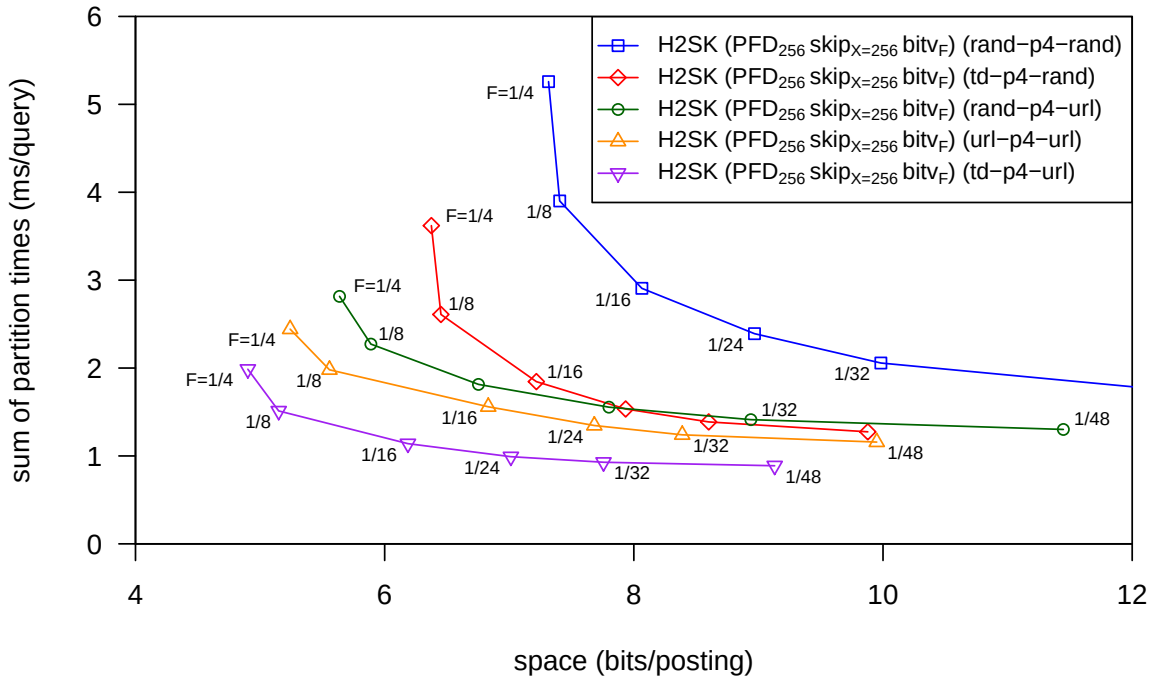

Figure 6.5: Space vs. summation of runtimes using *skips*, *bitvectors*, and partitioning.

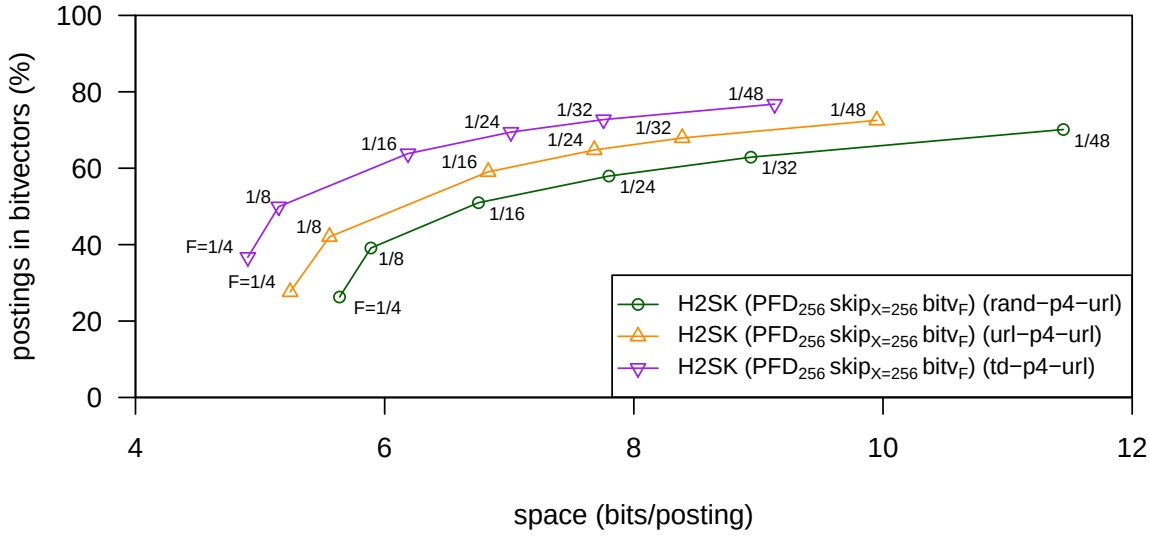

Figure 6.6: Space vs. postings in *bitvectors* using *skips*, *bitvectors*, and partitioning.

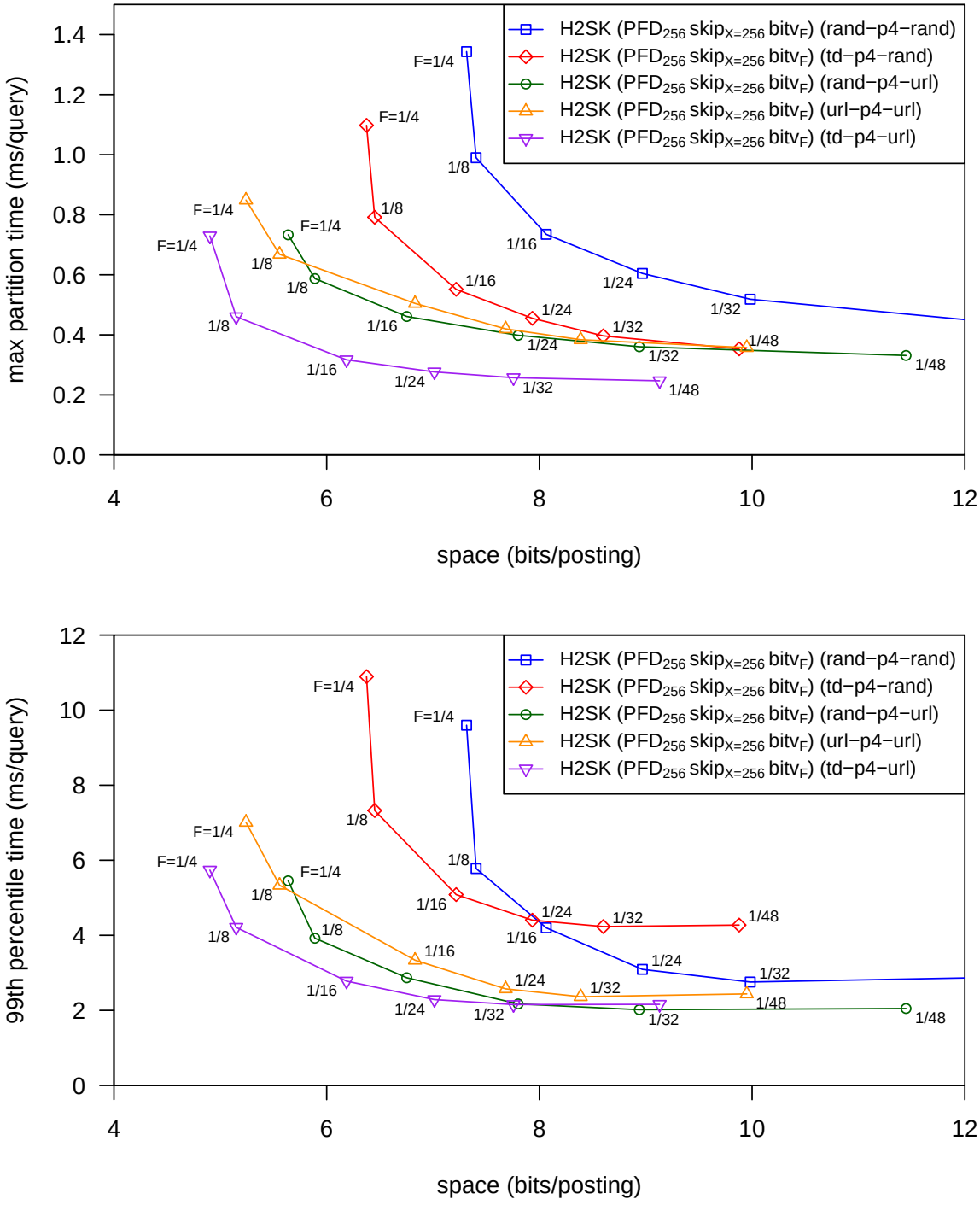

Figure 6.7: Space vs. maximum runtime using *skips*, *bitvectors*, and partitioning.

to decide where to use *bitvectors* in the index, but we leave this exploration for future work.

The space-time improvement on a single machine from using terms-in-document partitioning occurs without any changes to the actual list intersection code, thus proving our thesis that skewed clustering can be exploited to improve space-time performance. It also means that the space-time improvements from groups and *semi-bitvectors* are not caused just by coding differences, biases, or lucky breaks. In addition, the performance of *bitvectors+skips* using td-p4-url is similar to the performance of *semi-bitvectors* using td-g4-url, meaning our *semi-bitvector* approach and the specific implementation are effectively exploiting the term-in-document skew without incurring significant space-time overheads.

When our four partitions are run on separate machines, the maximum partition runtime performance of *PFD* compression with *bitvectors+skips* using td-p4 partitioning is still significantly faster than rand-p4 or url-p4 partitioning, as shown in Figure 6.7 (top). Comparing the average partition runtime to the maximum partition runtime measures the amount of load balancing. We find that rand-p4 has the best load balancing, while td-p4 has better load balancing than url-p4 in most configurations ($F \geq \frac{1}{8}$). For all the partitioning approaches, using more *bitvectors* gives better load balancing. The partition cutoff points used in td-p4 partitioning were defined to place an equal number of postings in each partition. Changing these cutoff points may be able to produce better load balancing, but we leave this exploration for future work.

All the partitioning approaches have similar runtime performance for the slow queries in our workload, as shown in Figure 6.7 (bottom). In all cases, using *bitvectors* greatly improves the runtime performance of the slow queries, as shown by comparing the performance of *bitvectors+skips* to the performance of *skips* in Figure 6.4 (bottom).

6.5 Partitioning vs. Grouping

We have shown that terms-in-document partitioning can improve space-time performance and that grouping by terms-in-document ordering and using *semi-bitvectors* can also improve performance. We now examine how to combine these approaches for the best overall results. For each of our partitioning approaches, we run eight terms-in-document groups with *PFD* compression, *semi-bitvectors*, and URL ordering within groups.

Using eight terms-in-document groups and *semi-bitvectors*, the single machine space-time performance of rand-p4, url-p4, and td-p4 partitioning are much closer together than their performance using *bitvectors+skips*, as shown in Figure 6.8. The best configuration from the previous section was td-p4-url using the *bitvectors+skips* algorithm, but when we add groups and *semi-bitvectors* to td-p4 partitioning the space-time performance degrades

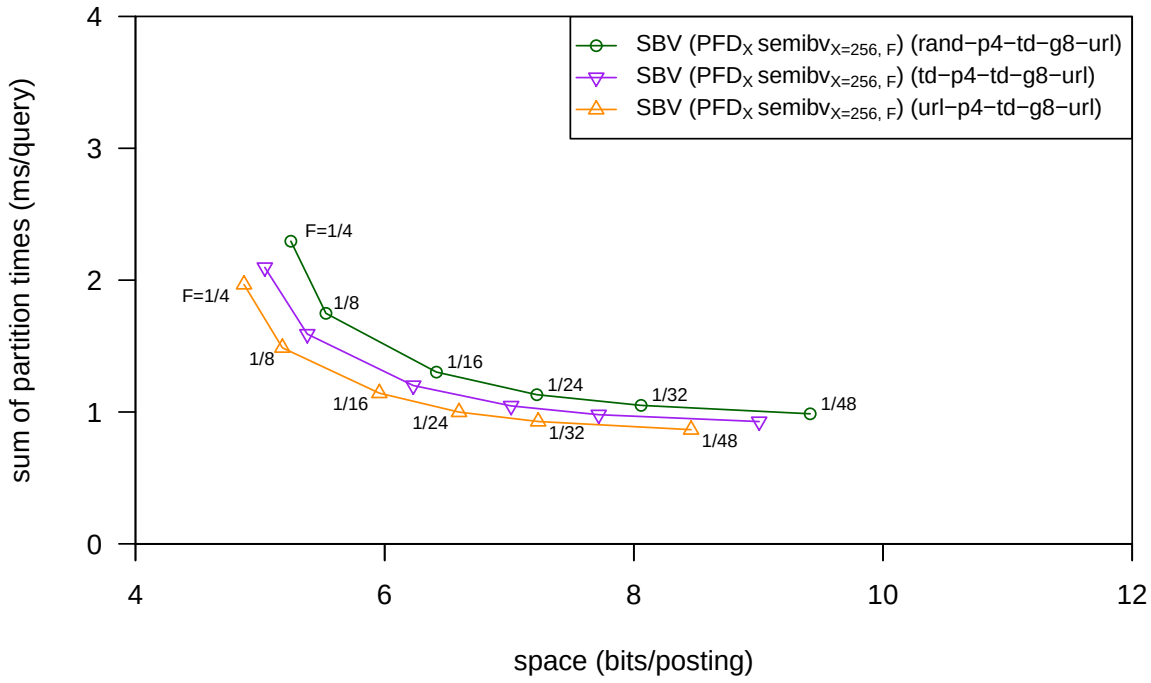

Figure 6.8: Space vs. summation of runtimes using *semi-bitvectors* and partitioning.

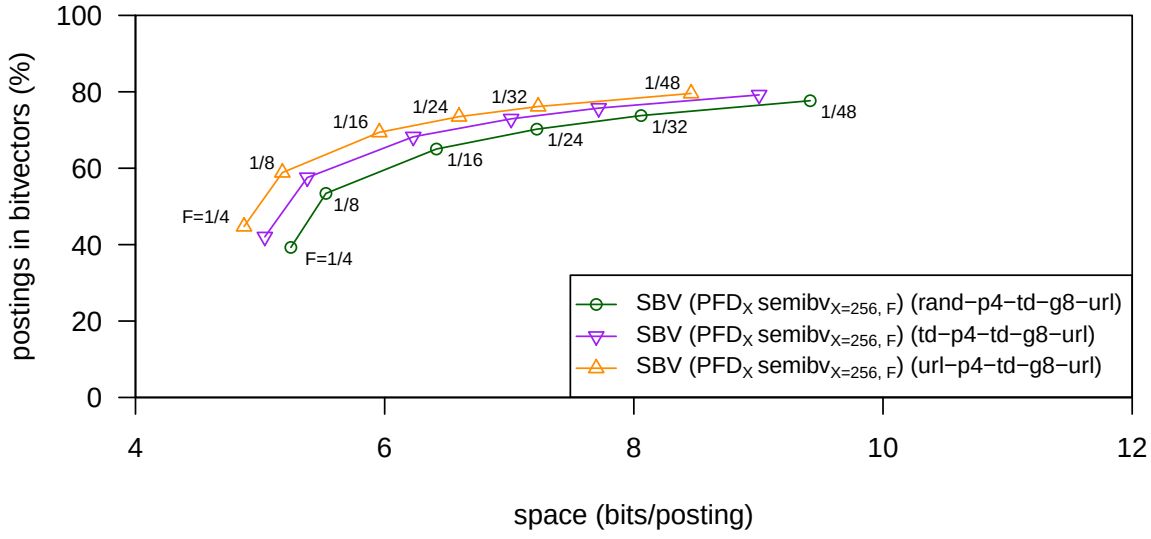

Figure 6.9: Space vs. postings in *bitvectors* using *semi-bitvectors* and partitioning.

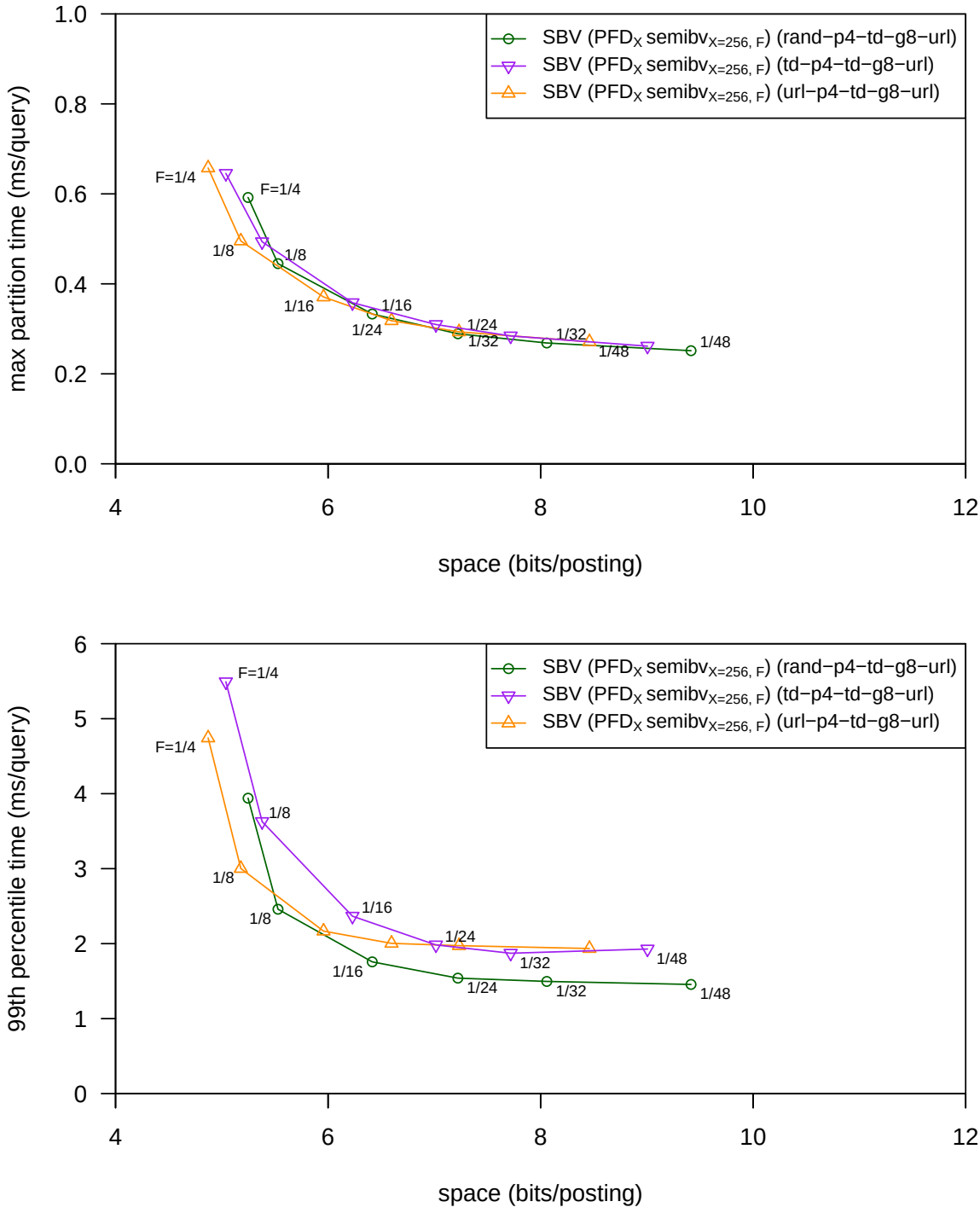

Figure 6.10: Space vs. maximum runtime using *semi-bitvectors* and partitioning.

slightly. For rand-p4 partitioning the space-time performance improves with groups and *semi-bitvectors*, but it is not as good as our td-p4-url configuration. For url-p4 partitioning the space-time performance also improves resulting in slightly better performance than our td-p4-url configuration and the other combinations of partitions and groups.

Using groups and *semi-bitvectors* can effectively exploit the skewed clustering in our index, so the difference in single machine performance between our three partitioning approaches is related to the amount of tight clustering from the URL ordering that is retained in the partitions. This explains why URL partitioning has the best single machine performance. In addition, the single machine performance of terms-in-document partitioning and random partitioning will degrade as more partitions are used, because the tight clustering is reduced inside each partition. The single machine performance of URL partitioning, however, does not degrade as more partitions are added.

All of these partitioning approaches improve the number of postings that are placed in *bitvectors* when using *semi-bitvectors* with grouping, as shown in Figure 6.9. Since the terms-in-document partitions on a single machine slowed down when using *semi-bitvectors* with grouping, the benefits from using more *bitvectors* was outweighed by the reduction in tight clustering and the implementation overheads of using *semi-bitvectors*. Producing significant gains in *bitvector* usage on top of URL partitioning with groups and *semi-bitvectors* seems unlikely, since nearly 45% of the postings are in *bitvectors* for the smallest configuration and nearly 80% are in *bitvectors* for the $F = \frac{1}{48}$ configuration. This is similar to the 1024 group terms-in-document configuration (td-g1024-td), where the small configuration has a similar portion of postings in *bitvectors* and the large configuration has 81.5% of postings in *bitvectors*. Clearly, adding more groups or partitions will not improve *bitvector* usage significantly. However, additional gains in the usage rate of *bitvectors* for specific query workloads may be possible; we leave this exploration for future work.

When we compare the average partition runtime to the maximum partition runtime, we find that using *semi-bitvectors* and grouping improves the load balancing of configurations using few bitvectors ($F \geq \frac{1}{4}$) compared to using *bitvectors+skips*. The space-time performance using *semi-bitvectors* and grouping for all the partitioning approaches on multiple machines as measured by the maximum partition runtime is similar for all the partitioning approaches, as shown in Figure 6.10 (top). The performance of these approaches is also similar to using *bitvectors+skips* with our td-p4-url configuration.

The performance of slow queries is somewhat improved by using partitioning and grouping together, since using *bitvectors+skips* is slower than using *semi-bitvectors* and grouping with any of our partitioning approaches, as shown in Figure 6.10 (bottom). This improvement in the performance of slow queries from using partitioning and grouping together is smaller than the improvement from adding *bitvectors* to a system using only *skips*.

6.6 Conclusions

We have shown how skewed partitioning from document size distribution can be used to improve list intersection systems, allowing more compression and faster query runtimes. The compression improvements are from lowering the size and variance of deltas in postings lists. The runtime improvements when using *skips* are from reduced smallest list sizes, better locality of access, and sparse result regions. The runtime improvements when using *bitvectors* are from reduced smallest list sizes, better locality of access, and more effective adaption of *bitvectors* to varying list densities within partitions. When using *bitvectors*, our document size distribution, as compared to random or URL partitioning, produces runtime improvements for a single machine configuration by reducing the total work done per query, and for a multiple machine configuration by improving the runtime of the slowest machine while producing similar performance for the slowest queries. We demonstrated these benefits using an in-memory conjunctive list intersection system with random ordering and URL ordering within partitions.

Document size distribution is broadly applicable, being usable with any document ordering. Additional improvements to load balancing could be achieved by changing the partition cutoff points, using more smaller partitions on each machine, or increasing replication of slow partitions. Scaling to a large number of nodes [Feldman 11] may require a hierarchy of distribution approaches, such as distributing first by domain (e.g., .gov), and then by document size within each domain.

Overall, the benefits suggest that document size distribution warrants closer examination by the research community.

In addition, we have shown that combining *semi-bitvectors* and groups with any of our partitioning approaches results in similar performance to using document size distribution with *bitvectors+skips*. The performance of all of these configurations comes from allowing both the skewed clustering of term-in-document ordering and the tight clustering of URL ordering to be readily exploited. As above, scaling configurations using *semi-bitvectors* and groups with partitioning may require a hierarchy of distribution approaches.

Chapter 7

Conclusions

We have explored various strategies for representing postings lists and examined their effect on the performance of a conjunctive list intersection system. The strategies encompass data layouts, such as document ordering, grouping, and partitioning, as well as encodings, including the use of compact representations, *skips*, and *bitvectors*. Rather than picking a single best approach, we have presented these techniques as a set of tuneable alternatives that can be applied to a search system in order to improve its performance.

We have shown that *bitvectors* are much faster and more space efficient than other structures for storing dense postings lists. The sparser lists should be stored using delta compressed structures with large overlaid *skips*, and the documents should be in URL order to produce the best results in all situations. This *bitvectors+skips* combination dominates the use of either *skips* or *bitvectors* separately, and we believe this will hold for all compression algorithms and all document orders.

In order to improve performance further, we have examined the effects of document ordering and the selection of where to use *bitvectors* to more effectively exploit the speed of *bitvectors*. Performance improvements can be accomplished by grouping (or partitioning) by the document size, as measured by the number of unique terms in a document, to skew the postings lists and then using *bitvectors* for dense portions of the lists as defined by the groups (or partitions). An appropriate ordering technique, such as URL ordering which gives tight clustering of postings, can then be used within each group (or partition). To handle *partial bitvectors* within a single list, we propose *semi-bitvectors*, a new structure that uses *bitvectors* at the front of the list and *skips* over compressed deltas for the rest of the list, with the transition point picked to improve *bitvector* usage.

We have shown that given a space budget more postings can be placed in *bitvectors* by using either groups with *semi-bitvectors* or partitions with *bitvectors+skips*. The increased use of *bitvectors* combined with other resultant benefits gives a significant space-time performance improvement over existing list representation and intersection techniques. For

example, using our *GOV2* dataset and related workload, eight groups with *semi-bitvectors* resulted in a total query runtime speedup of at least 1.4x over *bitvectors+skips* and a speedup of at least 2.4x over *skips* alone with URL ordering. Surprisingly, using document size distribution with *bitvectors+skips* rather than random or URL distribution resulted in benefits to total query runtime and slowest partition runtime, with similar performance for the slow queries.

In addition, we have shown that any of our partitioning approaches can be combined with *semi-bitvectors* and grouping to effectively exploit both the tight and skewed clustering in the *GOV2* dataset. A hierarchy of partitioning approaches may be required to maintain this effectiveness when scaling to support a very large collection.

Some of the proposed techniques produce performance improvements even without using *bitvectors*, so this work is broadly applicable, however, using *bitvectors* or *semi-bitvectors* is clearly superior. In order to encourage the use of *bitvectors* and *semi-bitvectors*, we present methods for their application to full ranking based search systems in Section 7.2. These proposals warrant further investigation.

7.1 Future Work

Several aspects deserve further investigation before adopting our partitioning and grouping approaches. The two main questions involve applying these approaches to ranking based systems (see next section) and achieving good load balancing over the partitions. It is possible that good load balancing requires the use of random document distribution [Büttcher 10, §14.1.1], but we believe that various techniques could alleviate many of the problems caused by load imbalance. For example, static load differences can be improved by repartitioning or online document routing [Lavee 11], leaving the dynamic load caused by the imbalanced work involved in particular queries. This dynamic load imbalance could be reduced by overlapping partitions and on a per-query basis adaptively moving ownership of documents between partitions that overlap. Note, however, that high frequency load changes may not affect our in-memory system when the query runtimes are much faster than the application requires, because this allows query latency to be traded for system throughput. Exploring the amounts and frequency of dynamic load changes could identify where various distribution techniques can be used. For example, URL partitioning likely has large load changes, but random distribution of URL prefixes into partitions or terms-in-document partitioning might not. Other techniques, such as running multiple small partitions on each machine could also reduce the dynamic load.

Beyond ranking and load balancing, several other considerations deserve further study. In many places throughout the thesis, avenues for further exploration were identified. A few of these are elaborated in the remainder of this section.

Performance results are limited by the hardware, dataset, and workload explored, in our case a single machine running a single dataset (*GOV2*) and using a single workload (5,000 of the 100,000 TREC queries). Although we explored performance for queries with various term counts, more experiments are needed to determine how our optimizations would perform for various system configurations. Our workload consists of only conjunctive queries of single word tokens, but our partitioning and grouping techniques could be applied to other workloads including full Boolean, phrase, proximity, t-threshold, or *Weak-AND* queries. We believe that *bitvectors* could also help with these workloads, but we are not aware of any study that applies *bitvectors* to non-conjunctive queries in search engines. Extending our experiments to include the commonly used *NOT* operator may be especially valuable. Our grouping and partitioning approaches may also improve the performance of systems using data structures other than *skips* and *bitvectors*, such as the Elias-Fano encoding or wavelet trees, or in other portions of a search system, such as list caching in disk based systems. In addition, our performance testing was done through simulation of portions of a search system running single threaded queries with only the relevant index portions in memory. Our performance results should, therefore, be validated on a full concurrent live system with the entire index in memory.

One of the limits to effectively deploying our approach is that the optimum number of partitions and groups must currently be determined by trial and error. In addition to the number of groups, the cutoff points for each group may affect system performance, and the cutoff points for partitions may be selected so as to improve load balancing. Clearly, it would be valuable to have metrics to measure the potential of skewed clustering and tight clustering within a dataset and then to map these metrics into suggestions for a system configuration of partitions and groups.

Our hybrid td-g#-url ordering has been described as grouping by terms-in-document, followed by ordering within each group by URL, but it is equivalent to sieving documents from the URL ordering based on their terms-in-document values. A more detailed combination of URL's tight clustering and terms-in-document's skewed clustering could be used to produce a better combined ordering. For example, a document could be moved to another group if it no longer shares enough terms with its neighbours, thus sacrificing skewed clustering for tight clustering to get a better balance for these approaches.

Alternative hybrid orderings could combine groups using terms-in-document ordering with some other second ordering. This approach could exploit general ordering techniques that are better than URL ordering or tailored orderings that are tuned to the workload and dataset. As such, our grouping by terms-in-document approach acts to boost the performance of other document ordering techniques. In addition, the combination of grouping by terms-in-document with a second ordering could reduce the amount of time needed to calculate the second ordering, because the secondary ordering acts only on the documents within each group, rather than on the documents in the entire index. This could be a

big advantage for ordering techniques that do not scale well, such as approaches based on content similarity.

We have used the same tight URL ordering to partition documents and to order within groups, but these could be different in order to combine the benefits of different clustering techniques or to allow the use of more expensive ordering calculations at the lower level. For example, the partitioning level could use fast term matching [Lavee 11], while the lower ordering level could be based on content similarity.

We have improved the usage of *bitvectors* so that more postings can be stored in *bitvectors* without increasing total space. This causes the runtime performance to improve because such a configuration results in more of the postings used in the query being stored in *bitvectors*, even though our choice of when to use *bitvectors* was agnostic to the query workload. Runtime performance could, therefore, be improved further by using the query workload to decide on where to use *bitvectors* in the index. A more detailed approach could take into account the query usage rate and the difference in space-time performance from various storage methods, perhaps using a bin packing approach to fill up the space budget with the most effective storage methods.

If a search system is unable to use document ordering techniques, perhaps because the system has a very high update rate, the documents could still be grouped (or partitioned) by their terms-in-document size to produce some benefit. Indeed, any partial ordering that can exploit some amount of tight clustering or skewed clustering may have large benefits for such systems. Even a system tuned for range searching performance, such as a system using time based ordering, could exploit range filtering at a high level, such as at the partition or multi-partition level, and still exploit terms-in-document grouping (or partitioning) at a lower level.

7.2 Ranking Based Systems

The most significant unknown to be addressed is how well our approaches can be adapted to work within practical information retrieval systems, which all depend on ranking query results. Ranking based systems include ranking information, such as term frequencies, interleaved with the postings lists, which increases their size and decoding costs. This changes the relative costs of skipping and decoding, so our optimization choices for list intersection systems, such as using a skip size of 256, may be different for ranking based systems. Also, the runtime costs of intersecting lists and ranking results are both significant, so the relative improvements in runtime observed in our list intersection system may be significantly different in a ranking based system.

Ranking based systems will gain many of the benefits we have demonstrated for combining skewed clustering and tight clustering in list intersection systems. Comparable

configurations using terms-in-document grouping and/or partitioning will gain benefits including reduced smallest list size, compressible data, and locality of access. For ranking based systems that use *skips*, adding terms-in-document grouping and/or partitioning will make them more effective.

In addition, terms-in-document partitioning may be especially valuable for ranking based systems, because the differences among partitions will allow results from a subset of the partitions to more effectively improve subsequent execution. For example, results from a subset of the partitions could be used to distribute top-k ranking information which can improve pruning/early termination or to choose query execution types such as AND or *Weak-AND* [Broder 03]. Note, the skew in the list intersection result values toward large documents may also occur in ranking based results, but more investigation is need to verify this property for ranking based systems.

Although *bitvectors* cannot be easily interleaved with the term frequency and posting offset information used in ranking, they can still be valuable in ranking based search systems because they are much faster for the list intersection portion of the query processing.

For comparison, we provide performance numbers in Table 7.1 from various papers using in-memory query execution on the *GOV2* dataset alongside our own performance results. The runtime performance differences presented in these entries are much bigger than any hardware or query differences could produce. Indeed, our performance results are at a disadvantage, because we have indexed much more data, including the HTTP header information (our index has 9.0 billion document level postings vs. 6.8 billion reported elsewhere [Ding 10]).

algorithm/system	time	space	data	structures	type	reorder	reference
Lucene (*vbyte*) (AND)	26.0	42.1	text	ID+offsets	AND+cnt	N	[Vigna 13]
Quasi-succinct indices (QS*) (AND)	11.9	36.9	text	ID+offsets	AND+cnt	N	[Vigna 13]
Exhaustive AND	6.56	4.5	text	ID+freq.	BM25	Y	[Ding 11]
Hierarchical *Block-Max* (HIER 10G)	4.29	14.5	text	ID+freq.	BM25	Y	[Dimopoulos 13]
SK($\text{PFD}_{64}\,\text{skip}_{X=64}$)(url)	3.04	7.0	text+meta	ID	AND	Y	-
SBV($\text{PFD}_{256}\,\text{semibv}_{X=256,\,F=1/32}$)(td-g8-url)	0.96	8.8	text+meta	ID	AND	Y	-

Table 7.1: Published time (ms/query) and space (GB) performance numbers from various papers using the *GOV2* dataset.

The system configurations presented in Table 7.1 use different query execution approaches, but we believe the comparison is still valuable. The first two entries in the table reference systems that index the document identifiers and offsets without reordering the document identifiers, so the performance is slow and the index is large, even though the systems do not incorporate ranking calculations. The third and fourth entries refer to systems that index document identifiers and frequencies, and reorder the document

identifiers, so the performance is fast and the index is relatively small, even though the systems include BM25 ranking calculations. The fifth and sixth entries are our results for the fastest configuration of *skips* using *PFD* and a *semi-bitvector* configuration, where no ranking information is stored and no ranking calculations are executed. The fifth entry is equivalent to the third entry with ranking removed, resulting in much faster execution. The sixth entry using *semi-bitvectors* indicates a large runtime performance improvement over the fifth entry using only *skips*, while using little additional index space.

Clearly, our approach using *bitvectors* can answer a conjunctive query much faster than the existing ranking based systems, while using much less space than a full index. While some of this runtime performance difference is from the lack of ranking calculations, a significant portion of the improvement is from algorithm improvements.

We have demonstrated that executing conjunctive queries with s*emi-bitvectors* can be done using small amounts of space to produce extremely fast runtimes compared to ranking based search systems. These characteristics suggest that ranking based systems can benefit from judiciously incorporating *semi-bitvector* data structures. We introduce five possible approaches below:

Pre-filter: Use *semi-bitvectors* to produce the conjunctive results, then process the ranking structures restricted to these results, as suggested in previous work [Culpepper 10]. This may require reordering of conjunctive results if the ranking structures use a different document ordering. The ranking structures could use non-query based information, such as PageRank, or normal ranking structures, thus duplicating some postings information. Having the conjunctive results can make the ranking process more efficient by exploiting *skips* in the first list, or limiting the number of accumulators. Using conjunctive results to pre-filter proximity or phrase queries is the natural implementation approach. Using this type of pre-filtering, however, prevents the use of non-AND based processing such as *Weak-AND* [Broder 03].

Sub-document pre-filter: Use *semi-bitvectors* in a pre-filtering step as a heuristic to limit the results to high quality or highly ranked documents by exploiting the correlation of query term proximity to query relevance [Büttcher 06]. This is accomplished by splitting the documents into (potentially overlapping) sub-document windows, then building the *semi-bitvector* structures over these windows. This reduces the number of results that must be ranked, while the results being ranked will be highly relevant because the query terms appear close together. The conjunctive results will need to be mapped from window IDs to document IDs before executing the ranking step. The windows could be implemented as half-overlapping windows to guarantee proximity of query terms within half the window size. This approach needs more examination to determine if significant filtering can be achieved without adversely affecting ranking effectiveness. If this approach can produce significant filtering, the ranking step could be implemented by directly storing the tokens of each window for quick ranking/proximity/phrase processing.

High density filters: High density terms have low value for ranking, with the extreme case being stopwords. However, they can still act as a filter and be processed more efficiently using *semi-bitvectors*. In fact, high density regions of a postings lists may act similarly, but a constant ranking value may be needed to smoothly integrate filtering regions with ranking regions in a single postings list. Based on our results, even using *semi-bitvectors* with density cutoff $F \geq \frac{1}{8}$ can result in significant performance benefits. In systems that have multiple stages of ranking, the exact order of the candidate results produced by the first stage may not be important, so using *semi-bitvectors* with a high density cutoff might not degrade the ranking effectiveness of the final stage.

Query specific filter: The terms that could be implemented as filters may be query specific. To improve the processing efficiency of these filtering terms, duplicate structures can be introduced: a *semi-bitvector* structure for filtering and a separate structure suitable for ranking. In fact, additional information about the user from past system usage and *collaborative filtering*, such as topics of interest, can be included in the ranking algorithm. Adding these user specific terms may reduce the effect of certain query terms in the ranking, allowing more query terms to be executed as filters using *semi-bitvectors*.

Guided processing: *Semi-bitvector* structures can be used to produce conjunctive results that will provide statistics on the query, and these statistics can guide subsequent processing of the query. For example, the statistics can indicate whether ranking should be done using conjunctive processing or some form of non-conjunctive processing, such as a *Weak-AND* implementation. These statistics can also indicate how to adapt this processing to the specific query terms, perhaps by identifying the specific query term that causes the conjunctive processing to be overly restrictive. Processing the conjunctive results for a subset of the documents may be enough to produce effective statistics. Such adaptive query processing techniques deserve close examination.

Given these methods for applying our techniques to ranking based systems, we believe that our work will be broadly applicable in practice.

References

[Anh 05a] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, vol. 8, no. 1, pages 151–166, 2005. 3, 18

[Anh 05b] Vo Ngoc Anh and Alistair Moffat. Simplified similarity scoring using term ranks. In *Proceedings of the 28th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 226–233. ACM, 2005. 33

[Anh 10] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Software: Practice and Experience (SPE)*, vol. 40, no. 2, pages 131–147, 2010. 19

[Aragon 89] Cecilia R Aragon and Raimund G Seidel. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 540–545. IEEE, 1989. 31

[Arroyuelo 13] Diego Arroyuelo, Senén González, Mauricio Oyarzún and Victor Sepulveda. Document identifier reassignment and run-length-compressed inverted indexes for improved search performance. In *Proceedings of the 36th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 173–182, 2013. 34, 35

[Baeza-Yates 04] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Combinatorial Pattern Matching*, pages 400–408. Springer, 2004. 16

[Baeza-Yates 11] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern information retrieval: the concepts and technology behind search, second edition.* Addison-Wesley, 2011. 7

[Barbay 09] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu and Alejandro Salinger. An experimental investigation of set intersection algorithms for text

searching. *Journal of Experimental Algorithmics (JEA)*, vol. 14, no. 1:3.7, pages 1–24, 2009. vii, 15, 16, 17, 32, 38

[Baykan 08] Izzet Cagrı Baykan. *Inverted index compression based on term and document identifier reassignment*. PhD thesis, Bilkent University, 2008. 34, 35

[Bentley 76] Jon Louis Bentley. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, vol. 5, no. 3, pages 82–87, 1976. 16

[Blanco 06] Roi Blanco and Alvaro Barreiro. TSP and cluster-based solutions to the reassignment of document identifiers. *Information Retrieval*, vol. 9, no. 4, pages 499–517, 2006. 34, 35

[Blandford 02] Dan Blandford and Guy Blelloch. Index compression through document reordering. In *Proceedings of the Data Compression Conference (DCC)*, pages 342–351. IEEE, 2002. 34, 35

[Brin 98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, vol. 30, no. 1, pages 107–117, 1998. 11

[Broder 03] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 12th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 426–434. ACM, 2003. 9, 103, 104

[Burges 05] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, pages 89–96. ACM, 2005. 11

[Büttcher 06] Stefan Büttcher, Charles L. A. Clarke and Brad Lushman. Term proximity scoring for ad-hoc retrieval on very large text collections. In *Proceedings of the 29th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 621–622. ACM, 2006. 104

[Büttcher 10] Stefan Büttcher, Charles Clarke L. A. and Gordon V. Cormack. *Information retrieval: Implementing and evaluating search engines*. The MIT Press, 2010. 3, 7, 11, 34, 35, 100

[Cao 06] Yunbo Cao, Jun Xu, Tie-Yan Liu, Hang Li, Yalou Huang and Hsiao-Wuen Hon. Adapting ranking SVM to document retrieval. In *Proceedings of the 29th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 186–193. ACM, 2006. 11

[Chapelle 09] Olivier Chapelle, Donald Metlzer, Ya Zhang and Pierre Grinspan. Expected reciprocal rank for graded relevance. In *Proceedings of the 18th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 621–630. ACM, 2009. 11

[Croft 10] W. Bruce Croft, Donald Metzler and Trevor Strohman. *Search Engines: Information Retrieval in Practice*. Addison-Wesley, 2010. 7, 11

[Culpepper 10] J. Shane Culpepper and Alistair Moffat. Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)*, vol. 29, no. 1:1, pages 1–25, 2010. vii, 2, 23, 27, 28, 32, 45, 53, 57, 59, 64, 104

[Demaine 01] Erik D. Demaine, Alejandro López-Ortiz and J. Ian Munro. Experiments on adaptive set intersections for text retrieval systems. *Algorithm Engineering and Experimentation (ALENEX)*, pages 91–104, 2001. 16

[Dimopoulos 13] Constantinos Dimopoulos, Sergey Nepomnyachiy and Torsten Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proceedings of the 6th International Conference on Web Search and Data Mining (WSDM)*, pages 113–122. ACM, 2013. 103

[Ding 10] Shuai Ding, Josh Attenberg and Torsten Suel. Scalable techniques for document identifier assignment ininverted indexes. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pages 311–320. ACM, 2010. 34, 35, 38, 49, 70, 103

[Ding 11] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 993–1002. ACM, 2011. 10, 36, 103

[Elias 74] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, vol. 21, no. 2, pages 246–260, 1974. 22

[Elias 75] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, vol. 21, no. 2, pages 194–203, 1975. 18

[Feldman 11] Moran Feldman, Ronny Lempel, Oren Somekh and Kolman Vornovitsky. On the impact of random index-partitioning on index compression. *CoRR*, vol. abs/1107.5661, pages 1–9, 2011. 12, 98

[Garcia 04] Steven Garcia, Hugh E. Williams and Adam Cannane. Access-ordered indexes. In *Proceedings of the 27th Australasian Conference on Computer Science*, pages 7–14. Australian Computer Society, Inc., 2004. 34, 35, 84

[Goldstein 98] Jonathan Goldstein, Raghu Ramakrishnan and Uri Shaft. Compressing relations and indexes. In *Proceedings of the 14th International Conference on Data Engineering (ICDE)*, pages 370–379. IEEE, 1998. 19

[Golomb 66] Solomon W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, vol. 12, no. 3, pages 399–401, 1966. 17

[Grossi 03] Roberto Grossi, Ankur Gupta and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850. SIAM, 2003. 32

[Howard 93] Paul G. Howard and Jeffrey Scott Vitter. Fast and efficient lossless image compression. In *Proceedings of the Data Compression Conference (DCC)*, pages 351–360. IEEE, 1993. 22

[Järvelin 02] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pages 422–446, 2002. 11

[Jonassen 11] Simon Jonassen and Svein Erik Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *Advances in Information Retrieval*, pages 530–542. Springer, 2011. 25

[Kane 09] Andrew Kane. Simulation of distributed search engines: Comparing term, document and hybrid distribution. *University of Waterloo Technical Report CS-2009-10*, pages 1–7, 2009. 13

[Kantor 96] Paul B. Kantor and Ellen M. Voorhees. Report on the TREC-5 confusion track. In *Proceedings of the 5th Text REtrieval Conference (TREC)*, pages 65–74, 1996. 11

[Konow 12] Roberto Konow and Gonzalo Navarro. Dual-sorted inverted lists in practice. In *String Processing and Information Retrieval (SPIRE)*, pages 295–306. Springer, 2012. 32

[Konow 13] Roberto Konow, Gonzalo Navarro, Charles L. A. Clarke and Alejandro López-Ortiz. Faster and smaller inverted indices with treaps. In *Proceedings of the 36th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 193–202. ACM, 2013. 31

[Kulkarni 10] Anagha Kulkarni and Jamie Callan. Topic-based index partitions for efficient and effective selective search. In *the 8th Workshop on Large-Scale Distributed Information Retrieval (LSDS-IR)*, 2010. 12

[Lavee 11] Gal Lavee, Ronny Lempel, Edo Liberty and Oren Somekh. Inverted index compression via online document routing. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*, pages 487–496. ACM, 2011. 12, 100, 102

[Lemire 13] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience (SPE)*, 2013. (accepted online). 23, 47

[Long 03] Xiaohui Long and Torsten Suel. Optimized query execution in large search engines with global page ordering. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 129–140. VLDB Endowment, 2003. 33, 34, 35

[Moffat 96] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems (TOIS)*, vol. 14, no. 4, pages 349–379, 1996. 23

[Moffat 00] Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, vol. 3, no. 1, pages 25–47, 2000. 22

[Moffat 06] Alistair Moffat, William Webber and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *Proceedings of the 29th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 348–355. ACM, 2006. 13

[Moffat 07a] Alistair Moffat and J. Shane Culpepper. Hybrid bitvector index compression. In *Proceedings of the 12th Australasian Document Computing Symposium*, pages 25–31, 2007. 4, 45, 58

[Moffat 07b] Alistair Moffat, William Webber, Justin Zobel and Ricardo Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, vol. 10, no. 3, pages 205–231, 2007. 13

[Moffat 08] Alistair Moffat and Justin Zobel. Rank-biased precision for measurement of retrieval effectiveness. *ACM Transactions on Information Systems (TOIS)*, vol. 27, no. 1:2, pages 1–27, 2008. 11

[Moler 86] Cleve Moler. Matrix computation on distributed memory multiprocessors. *Hypercube Multiprocessors*, vol. 86, pages 181–195, 1986. 1

[Navarro 10] Gonzalo Navarro and Simon J. Puglisi. Dual-sorted inverted lists. In *String Processing and Information Retrieval (SPIRE)*, pages 309–321. Springer, 2010. 32

[Puppin 06] Diego Puppin, Fabrizio Silvestri and Domenico Laforenza. Query-driven document partitioning and collection selection. In *Proceedings of the 1st International Conference on Scalable Information Systems*, no. 34, pages 1–8, 2006. 12

[Rice 71] Robert Rice and James Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology*, vol. 19, no. 6, pages 889–897, 1971. 17

[Risvik 03] Knut Magne Risvik, Yngve Aasheim and Mathias Lidal. Multi-tier architecture for Web search engines. In *Proceedings of the 1st Latin American Web Congress (LA-WEB)*, pages 132–143, 2003. 13

[Robertson 95] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline M. Hancock-Beaulieu and Mike Gatford. Okapi at TREC-3. In *Proceedings of the Third Text REtrieval Conference (TREC)*, pages 109–109, 1995. 11

[Sanders 07] Peter Sanders and Frederik Transier. Intersection in integer inverted indices. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 71–83. SIAM, 2007. 23, 25, 32, 45, 51

[Shapiro 86] Leonard D Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems (TODS)*, vol. 11, no. 3, pages 239–264, 1986. 14

[Shi 12] Liang Shi and Bin Wang. Yet another sorting-based solution to the reassignment of document identifiers. In *Information Retrieval Technology*, pages 238–249. Springer, 2012. 34, 35

[Shieh 03] Wann-Yun Shieh, Tien-Fu Chen, Jean Jyh-Jiun Shann and Chung-Ping Chung. Inverted file compression through document identifier reassignment. *Information Processing & Management*, vol. 39, no. 1, pages 117–131, 2003. 34, 35

[Shkapenyuk 02] Vladislav Shkapenyuk and Torsten Suel. Design and implementation of a high-performance distributed web crawler. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 357–368. IEEE, 2002. 45

[Silvestri 04a] Fabrizio Silvestri, Salvatore Orlando and Raffaele Perego. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proceedings of the 27th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 305–312. ACM, 2004. 34, 35

[Silvestri 04b] Fabrizio Silvestri, Raffaele Perego and Salvatore Orlando. Assigning document identifiers to enhance compressibility of web search engines indexes. In *Proceedings of the 19th ACM Symposium on Applied Computing (SAC)*, pages 600–605. ACM, 2004. 34, 35

[Silvestri 07] Fabrizio Silvestri. Sorting out the document identifier assignment problem. *Advances in Information Retrieval*, pages 101–112, 2007. 3, 34, 35, 50, 64

[Silvestri 10] Fabrizio Silvestri and Rossano Venturini. VSEncoding: efficient coding and fast decoding of integer lists via dynamic programming. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1219–1228. ACM, 2010. 21, 35

[Smucker 12] Mark D. Smucker and Charles L. A. Clarke. Time-based calibration of effectiveness measures. In *Proceedings of the 35th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 95–104. ACM, 2012. 11

[Strohman 05] Trevor Strohman, Howard Turtle and W Bruce Croft. Optimization strategies for complex queries. In *Proceedings of the 28th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 219–225. ACM, 2005. 9

[Tonellotto 11] Nicola Tonellotto, Craig Macdonald and Iadh Ounis. Effect of different docid orderings on dynamic pruning retrieval strategies. In *Proceedings of the 34th ACM International Conference on Research and Development*

in Information Retrieval (SIGIR), pages 1179–1180. ACM, 2011. 33, 34, 35

[Trotman 03] Andrew Trotman. Compressing inverted files. *Information Retrieval*, vol. 6, no. 1, pages 5–19, 2003. 18

[Turtle 95] Howard Turtle and James Flood. Query evaluation: strategies and optimizations. *Information Processing & Management*, vol. 31, no. 6, pages 831–850, 1995. 9, 14, 15

[Vigna 13] Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the 6th International Conference on Web Search and Data Mining (WSDM)*, pages 83–92. ACM, 2013. 22, 30, 103

[Wang 11] Lidan Wang, Jimmy Lin and Donald Metzler. A cascade ranking model for efficient ranked retrieval. In *Proceedings of the 34th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 105–114. ACM, 2011. 11

[Williams 99] Hugh E Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, vol. 42, no. 3, pages 193–201, 1999. 2, 18

[Witten 99] Ian H. Witten, Alistair Moffat and Timothy C. Bell. *Managing gigabytes: compressing and indexing documents and images, second edition*. Morgan Kaufmann, 1999. 7

[Xi 02] Wensi Xi, Ohm Sornil, Ming Luo and Edward A Fox. Hybrid partition inverted files: Experimental validation. In *Research and Advanced Technology for Digital Libraries*, pages 422–431. Springer, 2002. 13

[Yan 09] Hao Yan, Shuai Ding and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*, pages 401–410. ACM, 2009. vii, 21, 32, 33, 34, 35, 39, 50, 52

[Zhang 08] Jiangong Zhang, Xiaohui Long and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web (WWW)*, pages 387–396. ACM, 2008. vii, 3, 18, 19, 32, 45

[Zobel 06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, vol. 38, no. 2:6, pages 1–56, 2006. 7

[Zukowski 06] Marcin Zukowski, Sandor Heman, Niels Nes and Peter Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, no. 59, pages 1–12. IEEE, 2006. 3, 19