An Experimental Analysis of Skipping Join Algorithms

by

Ian J. Rae

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2014

Date of final oral examination: 02/21/2014

The dissertation is approved by the following members of the Final Oral Committee:

Jeffrey F. Naughton, Professor, Computer Sciences
Jignesh M. Patel, Professor, Computer Sciences
AnHai Doan, Associate Professor, Computer Sciences
David J. DeWitt, Professor Emeritus, Computer Sciences
Kristin R. Eschenfelder, Professor, Library & Information Studies

To my wife, Irene.

Acknowledgments

First and foremost, I thank my advisor, Jeff Naughton. Over my career in graduate school, Jeff provided invaluable guidance, shared numerous amusing stories, and exhibited tremendous patience with me, even as my description of my results always managed to sound like the Monty Python penguin sketch.[1] I would never have been successful without Jeff's incredible tolerance and willingness to let me explore my own way, and I am forever indebted to him for taking me on as a student.

My wife, Irene, has been amazingly supportive and wonderful throughout this long and arduous process. She has never once complained about long hours spent at the lab or late nights when a deadline was near, proofread papers she had no interest in, and suffered through numerous practice talks. Furthermore, after taking my advice to enroll in graduate school and pursue a Ph.D. of her own, she hasn't blamed me despite encountering a great deal of long hours and late nights herself.

I am, of course, incredibly blessed to have a loving, considerate family that has encouraged me to pursue my dreams. My parents, James and Judi, have always had my best interests at heart, and they have never done anything less than their very best to enable me to succeed. Despite a few differences growing up, my sisters, Jennifer and Julie, have been my staunchest allies, and they only suggested that I "go get a real job" once or twice (just kidding...). I wouldn't trade you guys for the world.

My graduate studies have been supported in large part by the Microsoft Gray Systems Lab, led by David DeWitt. David, thank you for providing financial support, guidance, and lending your expertise whenever I needed it, as well as for showing me that it is possible to be highly successful while never losing your acerbic wit and quick tongue. Furthermore, the staff at GSL gave me their time and their knowledge on several occasions, and I thank Alan Halverson, Eric Robinson, Rimma Nehme, Nikhil Teletia, Willis Lang, Srinath Shankar, Melody Bakken, Donghui Zhang, Hideaki Kimura, and Dimitris Tsirogiannis for all of their assistance.

Jignesh Patel, AnHai Doan, and Chris Ré taught me many things about data management and life as a researcher, and I greatly appreciate their innumerable insights. Kristin Eschenfelder, Alan Rubel, and Anuj Desai broadened my perspective on ethics, law, and theories regarding the use of information, and Kristin in particular gave me the ability to say that I've read Bruno Latour, which is worth a lot of points in some circles.

[1] http://www.montypython.net/scripts/penguins.php

Wisconsin wouldn't be the same without its students, and I thank Danielle Albers, Sean Andrist, Akanksha Baid, Jeff Ballard, Spyros Blanas, Craig Chasseur, Yueh-Hsuan Chiang, Jaeyoung Do, Avrilia Floratou, Chaitanya Gokhale, Brian Kroth, Yeye He, Chien-Ming Huang, Arun Kumar, Jiexing Li, Kwanghyun Park, Erik Paulson, Allison Sauppé, Chong Sun, Dan Szafir, Khai Tran, Mark Wellons, Wentao Wu, and Chen Zeng for their friendship and support.

Last, but certainly not least, I thank Radek Vingralek, Eric Rollins, Jeff Shute, Ben Handy, Chad Whipkey, Stephan Ellner, Bart Samwel, Shivakumar Venkataraman, and the rest of the engineers I worked with at Google for being great colleagues and excellent friends.

Contents

List of Tables

List of Figures

Abstract

Joins are a fundamental operation in relational database management systems (RDBMSs), and join processing has been a focus of both researchers and system implementors for decades. In this work, we are concerned with the amount of input data processed by join algorithms, since this has a direct impact on join processing performance. In an ideal situation, a join algorithm would be able to read only the input that directly contributed to the output, and no more.

In practice, however, join algorithms typically consume many more tuples than necessary; hybrid hash join, for example, consumes the entirety of both of its inputs, even if the output of the join is empty. In some sense, this is quite unsatisfying—clearly it seems that we should be able to do better, if we have the right information. To address this, we evaluate a particular category of join algorithms that we term *skipping join algorithms*.

Skipping join algorithms utilize secondary data structures and metadata to reduce the amount of data they must process in order to evaluate a join. The simplest skipping join algorithms are well known, as both index nested loops join, which relies on an index, and merge join, which relies on sort order, can be viewed as skipping data. In this work, we investigate two more complicated algorithms, ZigZag merge join and Minesweeper, that utilize the combination of indexes and sort order to skip more data.

In some cases, ZigZag merge join can be arbitrarily faster than traditional merge join, and in other cases, Minesweeper can be arbitrarily faster than ZigZag merge join. While this can be shown in theory, the central issue of our work is whether such skipping is useful in practice. We demonstrate that skipping join algorithms are broadly applicable, improving performance on full-text search queries in an RDBMS as well as on queries inspired by general-purpose database benchmarks. Furthermore, we show that although Minesweeper has better theoretical complexity than ZigZag merge join, in the general relational queries we considered, ZigZag merge join provides better performance. Finally, we provide insight into how to estimate the performance of a skipping join algorithm prior to execution.

It appears the database community has now reached the point where experiments are no longer an option. There was a time when experiments were not normal in a computer-science research paper. There was even a term in the database community—a "Wisconsin-style paper"—for a paper in which a proposed algorithm was implemented and experiments regarding its performance were described. Kudos to the faculty at Wisconsin for seeing the value of using experimentation as a way to demonstrate the worth of certain ideas.

…

[T]here is a more damaging effect of the seriousness with which we take experimental results. It encourages the writing of papers that really shouldn't be written, because they are so incremental and specialized that their use in practice is unlikely. There are many areas of database research where the nature of the data can vary greatly, and performance of different algorithms will vary with the data. Think of multidimensional indexes, or clustering, or even join algorithms. In research areas of this kind, it is very easy to find a special form of data and invent an algorithm that works well in this narrow special case. You then run your experiments on data for which your algorithm is best suited and compare it with others, which—surprise surprise—do not work as well. But were you to run your algorithm on the common cases, or random cases, you would do less well or not well at all. It doesn't matter; you can still publish yet another paper about yet another algorithm for doing this or that.

— Jeffrey D. Ullman (2013)

1 | Introduction

Joins are a core operation in relational database management systems (RDBMSs), and the problem of how to efficiently implement joins has been studied almost since the inception of the relational model. The goal of this thesis is to show that, even after 40 years of work, there is still something new to say about join algorithms. Database researchers and implementors have come up with myriad different algorithms and optimizations over the years, but by and large, there are three representative algorithms used to evaluate joins: index nested loops join, sort–merge join, and hash join [42]. Each of these join algorithms can excel in different situations, and by combining them, RDBMSs are able to offer good performance over a variety of queries.

However, in some circumstances, each of these algorithms performs dramatically more work than necessary. Consider a natural join on between two relations, R and S. Each relation has a single attribute, A, and both relations have a clustered index of some kind (e.g., a B-tree) on this attribute. Now, suppose the A value of the last tuple in R is larger than the A value of the first tuple in S. All three join algorithms will consume the entirety of at least one relation—index nested loop join will read all tuples in its outer relation, merge join will read all of R, and hash join will read all of both relations.

This result is somewhat unsatisfying. Since both relations are sorted and the last tuple in R is smaller than the first tuple in S, the join output is clearly empty; in fact, since both relations are indexed, this could be discovered with two simple index seeks. Instead, the traditional join algorithms process many more tuples than needed, and can in fact be made to do arbitrarily more work by manipulating the number of tuples in R and S.

Accordingly, this thesis focuses on *skipping join algorithms*—algorithms that do not need to

read (i.e., can *skip*) some portion of the input data while still producing correct output. The central premise of this thesis is that skipping join algorithms are *broadly applicable*, *efficient*, and *important* for overall database performance. We support our premise by examining three main areas: *why* we should skip data, *how* we should skip data, and *when* we should skip data.

1.1 Why we should skip data

The toy example we presented above is artificial and contrived, and it represents a corner case. While such examples are useful for showing the impact skipping can have on some queries, the benefit of skipping is meaningless if it cannot provide an appreciable performance improvement on a number of real-world workloads. To address this, the first part of this thesis demonstrates the importance of skipping join algorithms in a specific real-world setting: in-RDBMS full-text search.

Many major RDBMSs provide some support for full-text search integrated with the structured query engine [32, 44, 64, 84]. Efficient full-text search is commonly implemented using an inverted index (a set of terms with associated lists indicating which documents contain that term) [5, 102], and database researchers have shown that inverted indexes can be stored in relations and queried with SQL [46]. On the other hand, some RDBMS vendors have chosen instead to implement inverted indexes using specialized techniques from the Information Retrieval (IR) community [48].

These specialized inverted indexes implement their own operators and compression techniques that are not shared with the general-purpose relational engine within the RDBMS, and their integration with the general RDBMS storage layer treats it as little more than a glorified filesystem. Although RDBMSs are not typically used for text search within the IR community, we found it surprising that even commercial RDBMS developers would opt for a separate engine. After all, a common case for keyword search is conjunctive queries (e.g., find documents containing both "dog" and "cat"), and these queries can be implemented using standard relational joins.

Given that RDBMSs are highly optimized for join processing, one would expect that the relational engine would be able to provide good performance for this case. However, our preliminary

experiments showed that the specialized text search engine in Microsoft SQL Server was 10–20x faster than SQL Server's relational engine for text queries. After investigating this difference, we found that the performance of the specialized engine was not due to sophisticated compression algorithms, boutique storage formats, or vectorized operator execution, but rather the result of its skipping join algorithm.

We show that a variant of merge join called *ZigZag merge join* [7] can enable the general relational engine to match the performance of the specialized text search engine with no change in storage format. ZigZag merge join assumes that both of its input relations are indexed; when a mismatch between tuples is encountered, instead of performing a linear scan in the smaller input, ZigZag merge join executes an index seek to jump directly to the next possibly matching tuple. Each index seek can save a large amount of join processing, and use of ZigZag merge join improves performance on some text queries by an order of magnitude or more.

Given the ubiquity of text search on the Web, this indicates that skipping join algorithms can be important beyond simple toy examples, but text is just one possible workload. Accordingly, our next contribution extends ZigZag merge join to more general relational queries by evaluating it over queries from two standard database benchmarks, Star Schema Benchmark [76] and TPC-H [93], and we use this opportunity to compare with other skipping join algorithms proposed by the database theory community.

1.2 How we should skip data

ZigZag merge join is not the only skipping join algorithm; the efficacy of skipping algorithms has been known in the context of set intersection for some time. As a result, different algorithms exist with a variety of trade-offs, and consequently, the question of *how* to skip data is important. The basic idea behind how to skip is the amount of effort expended in attempts to avoid reading data, and whether this effort constitutes a net savings.

To illustrate this point, traditional merge join can be viewed as a very simple skipping join

algorithm. It avoids reading some tuples in one relation after finding the last tuple in another, since it knows that no later tuple could in the first relation could contribute to the output. ZigZag merge join extends this by performing index seeks within each relation in an attempt to skip some tuples. In the worst case, each index seek returns the very next tuple in the relation (i.e., the tuple that traditional merge join would have read next anyways). As a result, any time spent in the index is wasted, and ZigZag merge join would actually perform *worse* than traditional merge join.

Furthermore, database theoreticians have recently investigated a class of (near) worst-case optimal skipping join algorithms [73, 74, 95]. These optimal skipping join algorithms are more complex than ZigZag merge join, maintaining more state and, subsequently, potentially skipping more data. However, as these algorithms have not been evaluated in practice, no results exist that indicate whether their additional overhead is beneficial for general relational workloads.

In this thesis, we implement Minesweeper [73], a near-optimal skipping join algorithm, as a representative algorithm from this work. Minesweeper can be thought of as a variant of index nested loops join. It operates by repeatedly probing all relations in a join and noting the gaps between successive tuples; each gap indicates a region of the join space where no output can exist, and Minesweeper encodes these gaps as *constraints*. Since each gap is recorded, Minesweeper ensures that it never probes for a point that has already been ruled out, and over time, it produces the output of the join by systematically eliminating all non-matching regions of the join space.

Ngo et al. [73] provide a theoretical framework for reasoning about the complexity of skipping join algorithms, and we use this framework to show that while ZigZag merge join is optimal for simple two-relation joins, it is not optimal for more complex queries. We then compare the performance of ZigZag merge join and Minesweeper over a variety of workloads, including text search, Star Schema Benchmark, and TPC-H.

We demonstrate that, although Minesweeper has better theoretical complexity than ZigZag merge join, ZigZag merge join provides the best performance on real-world workloads. This result is due to two factors. First, Minesweeper must spend time maintaining its list of constraints as it explores the join space. Second, Minesweeper eliminates potential probe points only when it has

definitively ruled them out or output them; as a result, portions of a probe point may be probed repeatedly, causing unneeded index seeks. However, we also show that in queries with specially constructed data, Minesweeper can be arbitrarily faster than ZigZag merge join.

Consequently, depending on the query and the data distribution, either ZigZag merge join or Minesweeper may be better, and as we discussed earlier, in some instances, it may even be best not to skip at all. This gives rise to the need to determine *when* to skip data and which skipping join algorithm should be used, if any.

1.3 When we should skip data

The performance of skipping join algorithms is sensitive to two factors: how much data can be skipped and how much effort must be expended to skip it. The first factor, how much data can be skipped, can be cast as a cardinality estimation problem. If the size of the join output is small relative to the size of the input relations, then implicitly, a large number of tuples do not contribute to the output and can be skipped. Cardinality estimation is an established research problem that has a wide range of applications in query optimization, and researchers have developed a number of approaches to address it. In this thesis, we instead focus on the second factor—how much effort must be expended to skip.

Both ZigZag merge join and Minesweeper skip based on gaps in the data formed by adjacent tuples, and they do some fixed amount of work per skip. A gap in one relation means that any tuples whose join key falls within that gap in the other relation can be safely eliminated with no further processing. Accordingly, to estimate how many skips will occur, we need to determine how many gaps in one relation eliminate tuples in the other (and vice versa).

Unfortunately, standard statistics collected by RDBMSs, like histograms, are unlikely to capture gaps. In the case of equi-depth histograms, each bin will have a number of values equal to the depth, and as a result, *no* gaps will be captured. Equi-width histograms, in contrast, can capture gaps, but only if the bin size is small enough. Many RDBMSs use relatively few bins for equi-width

histograms; for example, Microsoft SQL Server 2008 uses a fixed maximum of 200 bins, even for relations with hundreds of millions of tuples [88]. As a result, it is very unlikely that any bin would be empty.

To address this, we propose a variant of equi-width histograms that we call a *gap map*. Gap maps are equi-width histograms that represent each bin as a single bit, which indicates whether any value in the bin was present in the relation. We can combine gap maps from different relations to estimate the number of skips needed to join them—any run of zero bits in one map where some bit is set in the other map indicates a skip.

Consequently, gap maps are effective at calculating the number of skips for a join; however, the number of gap maps required for perfect accuracy can be impractical. For example, consider a relation $D(id, year, month, day)$ that stores one year's worth of dates from January to December, with id as a sequential identifier. A single gap map on id would show that it has the values 1–365, with no gaps. However, if a query is posed with a predicate on $month$, the appropriate gap map on id is now much sparser; as a consequence, for complete accuracy, the RDBMS would need to store *twelve* gap maps, one for each month.

As a result, the number of gap maps that a system may need to store depends on the type of queries that are posed as well as the cardinality of the data. For complex queries or large datasets, this can quickly become intractable. Consequently, we do not claim that our work on gap maps is a definitive solution for estimating the number of skips required by a skipping join algorithm. Instead, we feel that gap maps are useful in providing insight on the complexities of this issue, and we think that further investigation into certificate size estimation would be interesting future work.

1.4 Thesis organization

The remainder of this thesis is organized as follows. Chapter 2 explores the performance of in-RDBMS full-text search and shows that skipping join algorithms, ZigZag merge join in particular, are important for high performance in this type of workload. We extend ZigZag merge join to

arbitrary relational data and we compare it with Minesweeper, a state-of-the-art worst-case optimal skipping join algorithm, in Chapter 3. In Chapter 4, we introduce statistics for skipping join algorithms, gap maps, and discuss methods for constructing and querying them. Finally, we discuss related work in Chapter 5, and we conclude in Chapter 6.

2 | Skipping join algorithms and in-RDBMS full-text search

2.1 Introduction

Full-text search is an important issue for relational database management systems (RDBMSs), with all major commercial and open-source RDBMSs providing some support for accelerating full-text searches with inverted indexes. A common way of implementing this feature is to integrate a specialized IR engine into the RDBMS, which optimizes storage and query execution for text search [48, 79]. IR engines designed in this way take advantage of the algorithms and techniques developed by the IR community [5], but these implementations raise an interesting question: why not use the standard relational engine for text queries?

The research community has demonstrated that text indexes can be expressed naturally in the relational model [37, 39, 46], and modern RDBMSs contain a great deal of machinery for storing, indexing, and processing relational data efficiently. By implementing support for full-text searches using a specialized IR engine, RDBMS developers are throwing away decades of research, development, and innovation on the core RDBMS query stack. These special-purpose engines do not use the relational query engine's operators for execution, and they treat the RDBMS as little more than a glorified filesystem.

Although RDBMSs are not typically used for text search in the IR community [17, 21], we found it surprising that even commercial RDBMS developers would opt for a separate engine. Accordingly, we felt compelled to investigate the differences between these specialized, in-RDBMS

IR engines and the standard relational query engine. Our preliminary performance tests were quite startling—we found that the specialized IR engine inside Microsoft SQL Server 2012 was 10–20x faster than SQL Server's relational engine when answering full-text queries.

While it might not be unexpected that a specialized system would outperform a more general one, we did not believe the magnitude of the performance gap would be so large. Given that inverted index lookups can be expressed as SQL queries, this indicates that SQL Server's IR engine is very effective at processing a particular class of SQL query—yet it is used only for text searches! In effect, the developers of Microsoft SQL Server have implemented an accelerator that offers incredibly high performance for certain SQL queries, but they have locked this functionality away in a "text-only" box.

Now, both the specialized in-RDBMS IR engine and the relational query engine are just computer programs. Either of them can be modified to include any feature the other possesses, which means there can be no formal argument that one engine cannot provide the performance delivered by the other. This plainly suggests the path of moving the IR performance-enhancing functionality to the SQL engine. However, these specialized IR engines are complicated and implement many features—the IR engine within SQL Server provides column-oriented, integer-specific compression; vectorized execution between operators; multi-way, multi-predicate join algorithms; and a block-based storage format with skip lists, just to name a few.

This is a large bag of tricks, and it is not immediately apparent how important each feature is to the overall performance of the system. If we are to unlock the functionality of the IR engine and bring it to a non-text world, we must first understand the "secret sauce" that makes the in-RDBMS IR engine so effective. In existing systems, however, all of these orthogonal features are packaged together. Furthermore, some of the features the IR engine provides—such as vectorized execution and column-oriented compression—have recently been applied to relational engines in the form of column-oriented databases [20, 58, 59, 60, 90].

This suggests that a column-oriented relational system may already have narrowed the performance gap between row-oriented relational systems and IR engines. However, we found in our

experiments that this was not the case, again raising the question of why this is so. Answering all of these questions requires a comparison of the performance of row-oriented RDBMSs, column-oriented RDBMSs, and special-purpose IR engines for IR workloads, as well as an analysis of the reasons for any observed differences.

In order to evaluate all three designs in a common code base, we built a system that implements an inverted index in three representative settings: an in-RDBMS IR engine in the style of a commercial RDBMS [48], a traditional ("textbook") row-oriented RDBMS, and a column-oriented RDBMS in the manner of Lamb et al. [58]. By having a standardized environment for these three approaches, we can identify which features are important for performance and what kind of changes need to be made to the relational engines, if any. After developing this framework, we discovered that the main reason for the IR engine's high performance is not its storage format nor its overall design, but the algorithm used to evaluate queries.

When the cache is warm, the IR engine is able to provide an order of magnitude better performance than the relational engines on conjunctive and phrase queries. As it computes posting list intersections, the in-RDBMS IR engine utilizes a two-level skip list structure to seek past postings that cannot match the query. This enables it to answer these queries with many fewer operations than the relational engines.

We show that the most important factor to narrow this performance gap is changing the RDBMS's traditional merge join algorithm to ZigZag merge join. ZigZag merge join uses the auxiliary data already present in the RDBMS, like B-trees and block-level metadata, to skip over irrelevant postings; this idea has been explored by researchers in a variety of contexts [7, 47, 50, 56, 99], but to the best of our knowledge, its importance to in-RDBMS IR has not yet been discussed. We demonstrate that ZigZag merge join can improve performance by as much as two orders of magnitude over traditional linear merge join with no change to the underlying storage format.

This chapter provides the following contributions:

- An outline of the storage formats and algorithms used when expressing an inverted index in an RDBMS using a special-purpose in-RDBMS IR engine, a traditional row-oriented

RDBMS, and a column-oriented RDBMS.

- Design guidance on which engine-level features are important for storing and querying inverted indexes efficiently in an RDBMS.
- A description and discussion of ZigZag merge join and its application to full-text search inside an RDBMS.
- An in-depth analysis of the differences in performance for conjunctive and phrase queries between a specialized IR engine, a row-oriented relational engine, and a column-oriented relational engine.

The rest of the chapter is organized as follows. We briefly describe the logical design of an inverted index in Section 2.2, and we detail the resulting storage format and layout of the inverted index in each of the three settings in Section 2.3. We discuss the algorithms involved in RDBMS inverted index lookups and extensions to them in Section 2.4, and we present our experimental performance results and analysis in Section 2.5. Finally, we conclude in Section 2.6.

2.2 Background

Logically, inverted indexes have two components: a set of *posting lists* that record which documents a term appears in, and a *dictionary* that maps terms to posting lists. Posting lists contain a series of integer document identifiers (*docids*) as well as associated ranking statistics. The exact statistics collected depend on the scoring formula used to rank a document's relevance—in principle, inverted indexes can be designed to store arbitrary supporting information.

In this work, we include statistics necessary for the Okapi BM25 scoring function [83], which include the following:

- **Term frequency**: the number of times a term occurs in a given document, abbreviated as *tf*.
- **Document frequency**: the number of times a term occurs in the entire corpus, abbreviated as *df*.

- **Document length**: the number of words in a document, abbreviated as *doclen*.
- **Item count**: the total number of documents in the corpus.
- **Average document length**: the average document length in the corpus.

In addition to these statistics, we support phrase and proximity queries by storing position offsets for each word occurrence.

2.3 Storage formats

Many different organizations can be used for storing the posting lists and the dictionary. These design choices affect both the storage footprint of the index as well as the runtime performance of queries. In the following sections, we describe designs for storing an inverted index in an RDBMS across three settings: a specialized in-RDBMS IR engine, a row-oriented relational engine, and a column-oriented relational engine.

2.3.1 Specialized IR storage

Although an in-RDBMS IR engine has more knowledge about the queries it is intended to execute than a relational query engine, it must still live within the constraints of the RDBMS. Since the RDBMS is also expected to execute non-text queries, the in-RDBMS IR engine cannot assume that its data is permanently memory resident, and it must format the inverted index into pages that are compatible with the RDBMS buffer manager. Consequently, the architecture of an in-RDBMS IR engine is not perfectly identical to a standalone IR engine; however, these engines implement many of the ideas from the IR community and adapt them to an RDBMS environment by treating the RDBMS primarily as dumb storage [79].

For our experiments, we evaluate a representative approach similar to that found in Microsoft SQL Server [48]. In this design, the inverted index is divided into a set of tuples that contain compressed blocks of hundreds of postings. Each tuple has three components: metadata about its

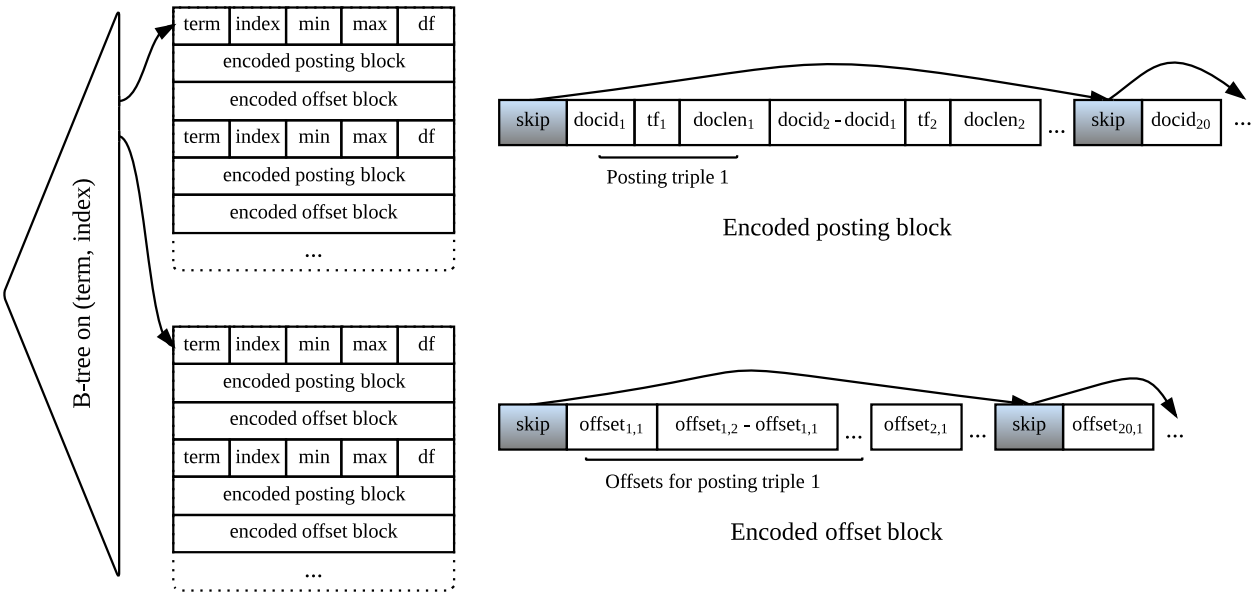


Figure 2.1: Physical design for a specialized, IR inverted index inside an RDBMS.

contents, including the range of docids within the tuple and the document frequency of the term; a set of postings that comprises (potentially a portion of) the posting list for the term; and a set of offset lists that correspond to the positional information for those postings. Because the posting list for a single term may be stored in multiple tuples, each tuple also has a zero-based index to indicate which partition of the posting list it stores.

We store these tuples in a row-oriented storage engine as a traditional table (see Figure 2.1). We represent the metadata portion of each tuple using explicit columns, and we include the posting lists and offsets as binary blob columns (bytes that are not interpreted by the relational storage engine). This allows us to use a clustered B-tree index to serve as the dictionary, and since tuples are stored on standard pages in the RDBMS, we can reuse the existing buffer manager for caching.

Accordingly, a standard index seek operator can be used to locate a term's posting list; however, the format of the posting and offset blocks is not natively understood by the RDBMS and must be decoded by special operators.

Posting block. The posting block stores the list of postings which indicate which documents in the corpus contain a term, sorted by docid. In addition, each posting stores information necessary to score matching documents for relevance. As a result, in our case, a single posting is actually a

triple of $\langle docid, tf, doclen \rangle$.

Because postings consist only of integers, they can be compressed with a variety of variable-length encodings to store them more efficiently [62, 98, 103]. For simplicity, in this work, we encode the integers using VByte [98]. As these encodings are biased toward efficient storage of small numbers, we store only the δ gaps between docids; each docid after the first is stored as the difference between it and the previous docid in the sequence. Note that, while all elements of a posting may be encoded using variable-length integers, only the docids may be stored as gaps, since the remaining elements are not sorted.

Offset block. Each posting has a list of offsets that indicate where that term appeared in the document, necessary for phrase and proximity queries. Accordingly, the offset block stores the set of offset lists corresponding to the postings in the posting block, with one list per posting.

These offset lists are sorted, δ-gapped, and compressed using the same variable-length encoding applied to docids. Offset lists are stored in the same order as the postings in the posting block, allowing the IR operators to implicitly join each offset list to its corresponding posting by position. Each offset list is terminated by a sentinel value to delimit it from the next list in the block.

Skip lists. When computing conjunctive, phrase, and proximity queries, the ability to rapidly seek within a posting list is important for performance [27]. A common way of handling this is to take advantage of the ordered posting list by executing a binary search; however, since postings are stored using δ gaps, they cannot be decoded without computing the sum of all prior postings. As a result, a standard binary search cannot be used.

To address this, we include a skip list in each block of postings. The skip list divides the posting list at evenly spaced, fixed-length intervals, and the posting stored immediately after each skip is stored as an explicit value instead of a gap. The first skip is always written to the head of the posting block, and it contains the byte offset of the following skip. This continues until the final skip, which contains the value 0 to indicate that there are no further skips available.

Because offset lists are stored in the offset block in the same order as postings in the posting block, when we insert a skip in the posting block, we also insert a skip in the corresponding location

in the offset block. This enables fast parallel iteration and seeking of both posting and offset lists.

Consequently, this design has two options for skipping unneeded postings. Entire blocks can be skipped by utilizing the block-level metadata stored in each tuple; the minimum and maximum docid values indicate whether a search key can be present in the block. Once the appropriate block is located, its skip list can be used to search for the desired posting.

2.3.2 Row-oriented storage

The entirety of the inverted index can also be stored in relational tables using a traditional row-oriented RDBMS. Because the data is organized in rows, compression possibilities are more limited; postings are generally not stored as δ gaps since data is accessed row-by-row. Additionally, schema design can have a large impact on performance.

Previous work has suggested a normalized index representation, with three tables in total: one for posting-level information $\langle term, docid, tf \rangle$, another for term-level information $\langle term, df \rangle$, and a third for document-level information $\langle docid, doclen \rangle$ [46]. This representation is space efficient and allows for relatively inexpensive index updates, but it has poor read performance for ranked queries since all tables must be joined to produce ranked results.

Accordingly, we implemented a denormalized design with all five columns in one table: $\langle term, docid, tf, df, doclen \rangle$. This design can be thought of as a materialized view of the normalized design where all statistics needed for relevance ranking are pre-joined. This increases the performance of reads at the cost of storage space, since many statistics are repeated in the denormalized table. We call this table the *main index table*.

To support proximity and phrase queries, we stored word offsets in a separate table as $\langle term, docid, offset \rangle$ tuples, which we call the *word offset table*. Due to the large number of word offsets found in a typical corpus, we opted not to include them in the denormalized main index table. Consequently, this organization requires an additional join against the main index table to return ranked results for queries involving offsets.

We stored both the main index table and the word offset table in clustered B-trees, with the former clustered on $\langle term, docid \rangle$ and the latter clustered on $\langle term, docid, offset \rangle$.

2.3.3 Column-oriented storage

We also implemented the denormalized schema from Section 2.3.2 in a column-oriented layout as described by Lamb et al. [58]. As before, we store the main index table sorted on $\langle term, docid \rangle$ and the word offsets table sorted on $\langle term, docid, offset \rangle$.

Column-oriented database engines typically support many different types of compression, including run-length encoding (RLE), δ gaps (as in the IR index), and variable-length integer encodings. In the case of a column-oriented inverted index, RLE is a natural fit for compressing termids, docids (in the word offset table), and repeated document-level statistics like document frequency. δ gaps seem like a natural fit for docids (in the main index table); however, although docids are sorted within a posting list, most column-oriented storage layouts compress entire pages of column values together without regard to semantic breaks in associated columns.

Accordingly, δ-gap compression does not work properly when a break between posting lists is encountered, since the first docid of the new posting list is likely to be smaller than the last docid of the preceding list. This results in the δ gap becoming a potentially large negative value, which is not encoded efficiently by most variable-length integer schemes. To address this, we used common delta compression [58] for docids, where all docids within a page are encoded as differences from the minimum docid in that page, regardless of posting list.

Rather than using B-trees to accelerate seeks for column values, the column store relies on a *metadata file*. The metadata file stores information about each page in the column's data file, including the position in the file, the minimum value in that page, and the maximum value in that page. Because the metadata file stores only a small amount of information about each page in the data file, and each page is a megabyte of data, the metadata file is much smaller than the data file. Even though the metadata file is essentially a flat structure, it is small enough to cache in memory, and it can be searched rapidly.

2.3.4 Updates

These physical designs are optimized for read-mostly workloads, as is common for inverted indexes. The assumption behind this is that the inverted index is primarily bulk updated, and in this case, the index can be maintained in multiple fragments in a manner similar to that found in standalone IR systems [17]. Each fragment is effectively a new inverted index that covers the most recent set of bulk-loaded documents; when a query is submitted, it must be executed against all index fragments, and the results must be merged before being returned. Since this impacts performance, fragments are periodically merged by a background worker.

We do not consider updates in this work. However, since some of the physical designs—the row-oriented engine in particular—may be more amenable to in-place updates than others, it would be interesting future work to evaluate update performance in these settings.

2.4 Query processing

The choice of whether to re-use the relational processing engine for inverted indexes or to integrate a special-purpose IR engine has a substantial effect on the algorithms used to evaluate queries. In particular, the specialized IR engine is free to implement operators optimized for a specific purpose (text lookups) with relatively few supported data types. Additionally, the specialized IR engine can make use of a small set of fixed query plans, and consequently does not need to integrate with the query optimizer.

The relational query engines, in contrast, are designed to be more general, supporting a wide range of data types and operating as part of trees of operators that can be generated from a vast space of query plans. Accordingly, the interface between relational operators is well-established and more loosely coupled than in a specialized IR engine. This architecture is expressive and powerful, but as implemented in traditional relational query engines, it can have a negative effect on IR performance.

In this section, we describe the algorithms used by our representative in-RDBMS IR engine for various types of queries, and we compare them with the plans generated by a commercial RDBMS

query optimizer.

2.4.1 Single-keyword and disjunctive queries

All three approaches are quite similar when considering single-keyword and disjunctive queries. Single-keyword queries map directly to index lookups. In the case of the specialized IR index, this means decoding a single posting list and returning all values, while in the relational engines, it means an index seek (either through a B-tree, metadata file, or other auxiliary data structure as appropriate).

Disjunctive queries are somewhat more complex, but not very. Both the IR engine and the relational query engine provide a multi-way union operator that computes the union of its children with a standard heap-merge algorithm [45]. This algorithm is quite straightforward; to union k posting lists, the union operator maintains a min-heap of k elements, populated with the head of each posting list. The smallest element in the heap can then be output and replaced with the next element from its corresponding posting list; after re-heapifying, the process repeats until all posting lists are exhausted.

Since neither single-keyword nor disjunctive queries perform any filtering on the posting lists involved, there is little room for optimization. Therefore, we focus the bulk of our attention on conjunctive and phrase queries.

2.4.2 Conjunctive queries

Conjunctive queries involve the intersection of one or more posting lists corresponding to the terms in the query. The subject of set intersection has been well-studied by the IR community, which has proposed and compared a number of approaches [8, 13, 27, 85]. Much of this work can be applied in an RDBMS setting, although some assumptions may not always hold; for example, because an RDBMS is not a dedicated IR engine, it is impractical to assume that the entirety of a posting list is in main memory. As a result, some techniques, such as binary search over the full range of a posting list, are not efficient.

The specialized inverted index uses a modified version of sequential multi-way intersection [13], shown in Algorithm 1. This algorithm intersects a set of k posting lists sorted in ascending cardinality; when a candidate value is determined not to be present in a later posting list, the head of that posting list is used as an eliminator in the first set. This allows later intersections to give feedback to the first intersection, which can substantially reduce the number of comparisons necessary to compute a query when combined with efficient seeks.

Algorithm 1 Multi-way algorithm used by the IR engine to intersect k posting lists.

```
min_docid ← 0
while all posting lists have values do
    i ← 0
    while i < k do
        if SEEK(l_i, min_docid) then
            docid ← GETNEXT(l_i)
            if i = 0 or docid = min_docid then
                i ← i + 1
            else
                i ← 0
            min_docid ← docid
        else
            return results
    add docid to results
    min_docid ← min_docid + 1
return results
```

The relational query engine, in contrast, uses a tree of binary merge join operators combined with index seeks (see Figure 2.2), implementing an interleaved form of "small vs. small" intersection [27]. Traditionally, these operators implement a Volcano-style pull-based iterator interface [43] with a GetNext function that returns either the next matching tuple (in the case of a traditional row-oriented engine) or block of tuples (in the case of vectorized execution). This naturally implements a form of Document-at-a-Time processing [94], since the pipeline of GetNext calls will iterate over all posting lists simultaneously. Unfortunately, using this interface means that operators that are higher in the tree cannot give feedback to operators lower in the tree; there is no way for a value from a higher posting list to be used as an eliminator in a lower posting list.

However, we can modify the Volcano-style interface to include a Seek function that accepts a

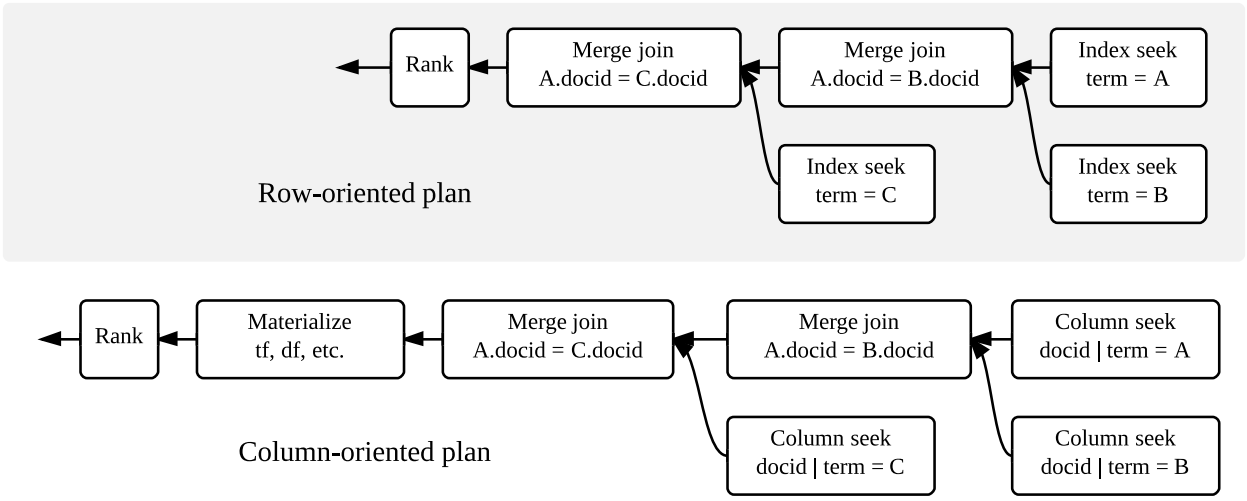


Figure 2.2: Plans for evaluating the 3-keyword conjunctive query "a AND b AND c" in row- and column-oriented relational inverted indexes.

search key as input and moves the operator's cursor to the first tuple greater than or equal to that key. Seek functionality can be integrated with a tree of binary operators in a straightforward manner. If an operator is able to process the seek itself (e.g., the operator is a scan), it does so; otherwise, it simply passes the search key on to all of its children through their SEEK functions. This is similar to a proposal by Graefe [40], where seek values may be bound for the inner input of a nested query iteration, but we enable seeking on both the outer and inner join inputs.

With this new interface, we can implement a *ZigZag merge operator*. The basic idea behind ZigZag merge is that when a mismatch of join keys is encountered, instead of simply advancing the cursor of the child whose value is smaller, the ZigZag merge operator first executes a seek for the larger value—potentially skipping over many non-matching results. The resulting pseudocode is shown in Algorithm 2.

To make ZigZag merge effective, the relational query engine requires an efficient way to process seeks. As described in Section 2.3.1, the IR engine relies on a two-level skip list to accelerate searches for values. The row-oriented query engine can use the B-tree on $\langle term, docid \rangle$ to perform the search; however, accessing the B-tree for every SEEK can be wasteful, since traversing the upper levels of the B-tree can require numerous comparisons. Furthermore, in the worst case, the result of the B-tree search would in fact be the next tuple on the page, which was due to be returned by the

Algorithm 2 ZigZag merge join algorithm for intersections.

```
r ← GetNext(R)
s ← GetNext(S)
while r ≠ End(R) and s ≠ End(S) do
    if r.docid = s.docid then
        add r.docid to results
    else if r.docid < s.docid then
        Seek(R, s.docid)
        r ← GetNext(R)
    else
        Seek(S, r.docid)
        s ← GetNext(S)
return results
```

next GetNext call regardless.

To reduce this overhead we conduct a gallop search [15] for the key on the current page prior to invoking the B-tree index. Gallop search is a form of exponential search where the offset between examined values begins at one and doubles with each iteration (accessing elements 1, 3, 7, …). Gallop search may overshoot the target value, in which case we apply a binary search over the remaining interval. Finally, if the gallop search indicates the search key is not on the current page, only then do we resort to a search in the B-tree index. As a result, there is at most one wasted index lookup per page of data.

Although the B-tree index serves as a good solution for the row-oriented relational engine, the column-oriented engine we described in Section 2.3.3 does not store its data in a B-tree. Instead, it maintains a small metadata file that contains information about each page, such as the minimum and maximum value on that page. In principle, this metadata file can be used to skip blocks based on search key values.

However, recall that page boundaries in the column-oriented engine are not divided along the boundaries of posting lists. Accordingly, the tail of one posting list and the head of the next posting list may be in the same page; as a result, the minimum and maximum values in the metadata file are not very discriminative, since the tail of one posting list is likely much larger than the head of another. This is compounded by the fact that the column-oriented engine uses large pages and

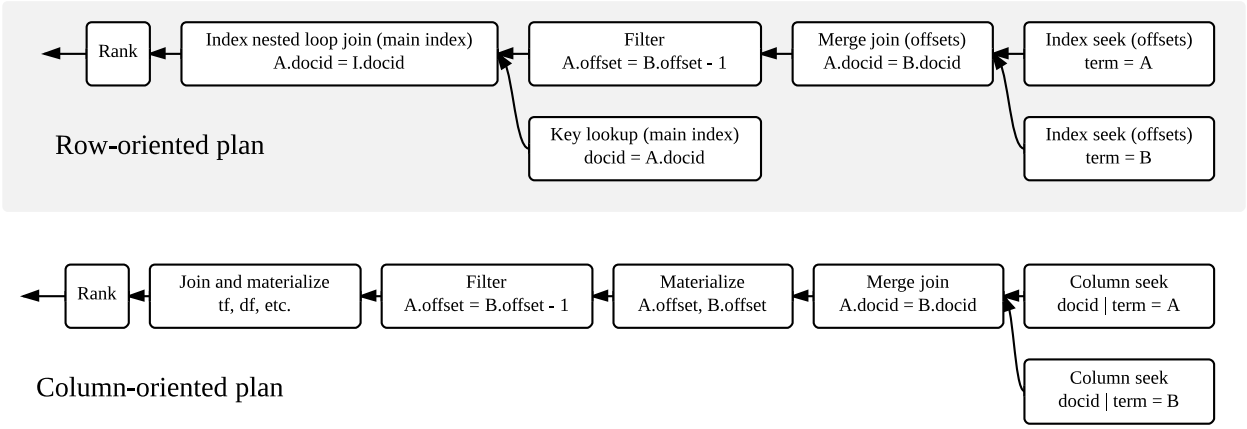


Figure 2.3: Plans for evaluating the 2-keyword phrase query "'a b'" in row- and column-oriented relational inverted indexes.

heavily compresses its data, packing many values into a single block. Therefore, even if the metadata information was more fine-grained, it would be unlikely to filter out a given page.

The column-oriented query engine's ability to skip within a page is also limited. Because the column-oriented query engine stores integers with variable-length encoding, algorithms like gallop search and binary search cannot be used; it is not possible for these algorithms to determine exactly how many elements they have skipped with each iteration. To address this, these algorithms could be modified to perform "fuzzy" jumps (for example, by skipping based on number of bytes instead of values). In this case, it would be difficult to incorporate the result with the rest of the column-oriented query processing, since the exact position of a row in the table is necessary for late materialization of additional columns.

2.4.3 Phrase queries

Phrase queries are similar to conjunctive queries, but they involve an additional level of processing. After a set of postings is intersected, their corresponding offsets must be checked to ensure that the search terms appear in the appropriate positions in the document. Additionally, once results have been located in the offsets table, the main index table must be joined to access ranking statistics. The manner in which these steps are performed has a dramatic effect on performance.

The in-RDBMS IR engine processes phrase queries using a multi-predicate merge join algo-

rithm [101]. Multi-predicate merge join takes advantage of the fact that both the posting and offset lists are sorted by merging on them in order. Effectively, the algorithm first computes a merge join on the postings to find an intersection, then computes a second merge join on the associated offset lists.

In contrast, both the row-oriented and column-oriented relational engines use only a single-predicate merge join operator, producing the plans shown in Figure 2.3. This operator computes a merge join on the postings to find an intersection, but it evaluates the condition on word offset as a residual. Consequently, it produces the cross product of all occurrences from two matching postings and applies a predicate to each of them, completely ignoring the sort order of the word offsets. This can substantially increase the number of comparisons required to evaluate phrases where one or more of the search terms occurs frequently in the same document.

The IR engine and the column-oriented relational engine both have an advantage over the row-oriented relational engine when it comes to materializing data. Our schema for the row-oriented relational engine stores offsets as $\langle term, docid, offset \rangle$ triples; since the data is organized in rows, all offsets are read from disk and processed by the join operator. Because the IR engine stores word offsets in a separate compressed block on the same page, it does not need to decode word offsets for postings that do not match. The column-oriented relational engine takes this one step further—as it stores each column in its own file, it can avoid reading unused offset data from disk.

However, we can partially address both of these issues by combining the multi-predicate merge join algorithm with the skipping merge join we introduced in Section 2.4.2 to create a *ZigZag multi-predicate merge join* operator (see Algorithm 3). As before, when the merge operator encounters a mismatch between docids, we utilize the SEEK function to search for the larger key in the other posting list, and we also apply the same logic when merging offset lists in the same operator. This produces query plans identical to those shown in Figure 2.2 with the merge join implemented by a ZigZag multi-predicate merge operator.

This ameliorates the largest problem with the row-oriented relational inverted index, namely the repetition of $\langle term, docid \rangle$ pairs. When two docids fail to intersect, the resulting seek skips a large

Algorithm 3 Multi-predicate ZigZag merge join algorithm for a 2-keyword phrase query.

```
r ← GetNext(R)
s ← GetNext(S)
while r ≠ End(R) and s ≠ End(S) do
    if r.docid = s.docid then
        if r.offset = s.offset − 1 then
            add r.docid, r.offset to results
        else if r.offset < s.offset − 1 then
            Seek(R, s.docid, s.offset)
            r ← GetNext(R)
        else
            Seek(S, r.docid, r.offset)
            s ← GetNext(S)
    else if r.docid < s.docid then
        Seek(R, s.docid)
        r ← GetNext(R)
    else
        Seek(S, r.docid)
        s ← GetNext(S)
return results
```

number of offset tuples, eliminating many unproductive GetNext calls and comparisons, and a further reduction is achieved by seeking when comparing offsets.

2.5 Experimental results

We performed a variety of experiments with a focus on conjunctive and phrase queries. Our preliminary results showed that all three approaches produced similar performance for single-keyword and conjunctive queries, and so we excluded them from the bulk of our investigation.

Since no existing system supports all of the algorithms and storage formats we are examining, any experiment using existing projects would necessarily involve cross-system comparisons where many variables changed simultaneously. Therefore, in the interest of maintaining a controlled environment, we developed the various inverted index designs in a standalone application written in C++. This application was meant to be as "DB-like" as possible, with files of slotted pages, a simple buffer manager that implements the Clock replacement algorithm, and a Volcano-style iterator

interface with query plans composed of generic operators that support tuple-at-a-time iteration for the row-oriented engine and block-at-a-time iteration for the IR and column-oriented engines.

While our implementation attempts to be fair to all approaches, it is of course true that it, like all programs, could be optimized further. These optimizations may improve the performance of any of the approaches we consider, and they may even change their relative position in the results when the performance of the approaches does not differ dramatically. However, our intent with this work is not to argue that specific implementations are good or bad; it is instead to identify broad trends and to determine which features of the storage engine and algorithm are important. These insights remain true regardless of what additional optimizations are applied.

Furthermore, although we believe that cross-platform experiments are less useful than our common-platform experiments, for a sanity check, we did run tests on a commercial row-oriented RDBMS, a special-purpose IR engine in a commercial RDBMS, and two commercial column-oriented RDBMSs. These tests confirmed the general trends identified in our common-platform experiments.

2.5.1 Experimental setup

We executed our experiments on a single server running Microsoft Windows Server 2012. The server was equipped with 2 quad-core, 2.16 GHz Intel Xeon processors and 32 GB of RAM. Data was stored on 12 10,000 RPM 146 GB SAS hard disks, with one disk storing the OS and the remaining disks, combined into an array with RAID 0, storing the inverted indexes. We used 8 kB pages for the IR and row-oriented inverted indexes, and 1 MB pages for the column-oriented inverted index. We sized our buffer pool to use 28 GB of memory, leaving 4 GB for the OS.

For the main index table in the column-oriented layout, we used run-length encoding (RLE) for terms; common delta encoding for docids, term frequencies, and document lengths; and RLE for document frequencies. As for the word offsets table, we used RLE for terms and docids and common delta encoding for offsets themselves. These choices are identical to what was produced by a commercial column-oriented RDBMS's physical design advisor when we provided it with our

inverted index schema and data.

In general, our results show the performance of queries executed in isolation with a warm cache. Each query was run once to warm the cache, and the results of that run were discarded. The query was then run five more times with the same parameters, and the mean execution time of these runs is reported as the execution time of the query.

Where explicitly noted, some experiments test cold-cache performance. To ensure that the cache was truly cold between each run, we flushed the cache on the disk controller, invalidated the Windows filesystem cache, and flushed the buffer pool before each execution.

Other than the unranked experiments mentioned in Section 2.5.7, all of our benchmark queries request documents ranked by relevance using the Okapi BM25 scoring function [83] with $k_1 = 1.2$ and $b = 0.75$. We omitted the time spent sorting the result by score, since all implementations have the same sorting cost. Because we are focused on raw performance, our benchmark queries return all results.

2.5.2 Corpus and keywords

We used a corpus of 21 million English-language USENET posts from Westbury Lab [86] that comprised 123 GB of text. Each document was assigned a unique, sequential integer docid, and we constructed the base inverted index data using the specialized IR engine in Microsoft SQL Server. Common words were removed using the IR engine's built-in English stop list, and we exported the contents of the inverted index into our experimental testbed application.

We divided the keywords in the corpus into three buckets based on their frequency using the method described in TEXTURE [33]. Before classifying the keywords, we sorted them by their total number of occurrences in descending order. Then, we called the words that comprised the first 90% of all occurrences *high* frequency, the words which comprised the next 5% (90%–95%) of all occurrences *medium* frequency, and the remainder *low* frequency.

2.5.3 Index sizes

The different storage layouts produced inverted indexes of varying sizes. As one might expect, the uncompressed row-oriented storage format used the most space; although we were able to store the clustered B-tree index with no free space in its pages since we are optimizing for read performance, the row-oriented index occupied 362 GB (137 GB for postings and 225 GB for offsets). The main reason for this large size is because the row-oriented index repeats the $\langle term, docid \rangle$ pairs for each individual occurrence of a word, dramatically increasing the size of the offset table.

The column-oriented storage format, in contrast, is able to remove much of this redundancy through RLE, and it also encodes integers more compactly. However, even with RLE, the $\langle term, docid \rangle$ pairs in the offset table still required 81 GB. Accordingly, the column-oriented inverted index required 162 GB (44 GB for postings and 118 GB for offsets), using less than half the space of the row-oriented index.

Finally, the IR engine's inverted index was the smallest of the three, using 71 GB to store both offsets and postings. The IR engine benefits from storing offsets in the same block as postings, which means that it does not need to store any additional $\langle term, docid \rangle$ pairs (not even pairs compressed with RLE). Its compression is otherwise quite similar to the column-oriented engine, with the exception that because the IR engine is aware of the semantic breaks between posting lists, it can utilize δ gaps more effectively than the column-oriented engine.

2.5.4 Conjunctive queries

We tested conjunctive query performance with queries that included two to five medium-frequency keywords. For each keyword cardinality, we selected 100 sets of keywords at random and verified that all keywords in a set co-occurred; any sets that had no results were reselected. Results reported are the mean execution time of all 100 queries. The average posting list for our selected keywords had 930,686 postings, and the average query returned 31,620 documents.

Our results are shown in Figure 2.4. When compared to approaches using traditional merge

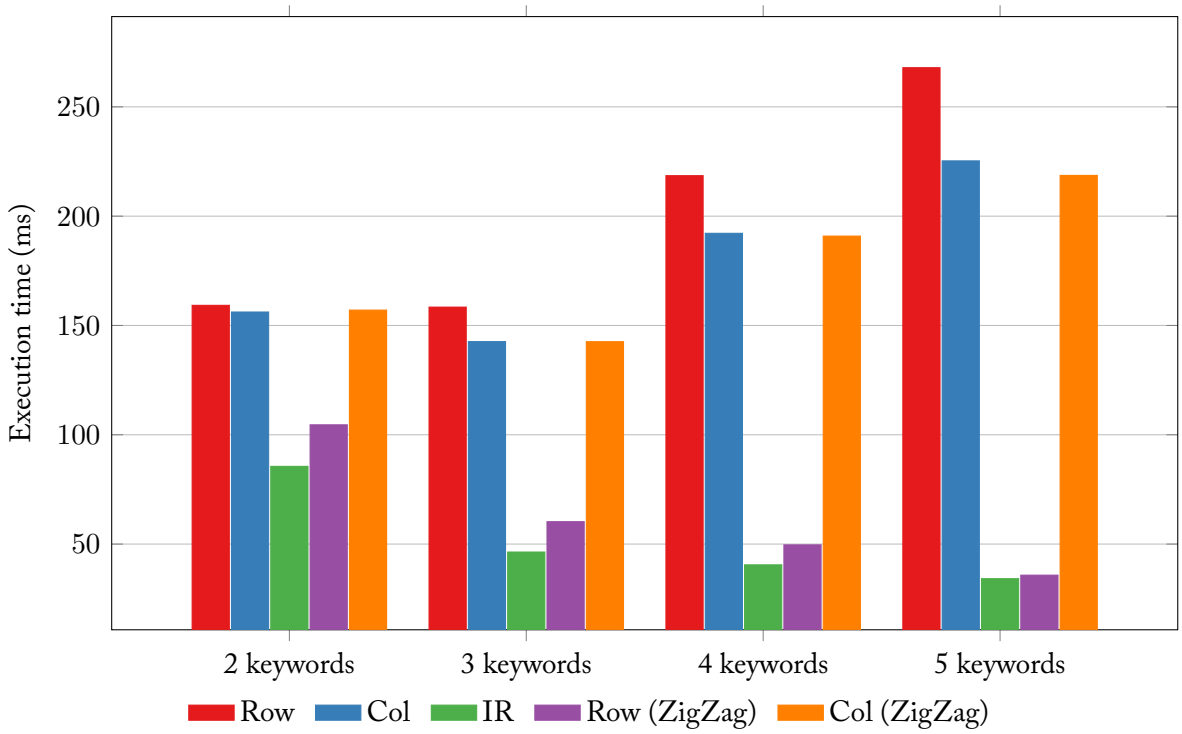


Figure 2.4: Conjunctive query performance with two to five keywords.

join, the IR engine has a tremendous performance advantage, offering performance that is up to eight times faster than both the row- and column-oriented relational engines. The column-oriented engine, although slower than the IR engine, outperforms the row-oriented engine with traditional merge join due to vectorized execution combined with late materialization.

Vectorized execution improves data locality and reduces function call overhead, while late materialization means that columns with ranking statistics need to be loaded only for documents that match the query. As the number of keywords in the query increases, the query becomes more selective, and the performance benefit from late materialization grows accordingly.

However, when using ZigZag merge, the row-oriented engine provides performance that is at worst 20% slower than the IR engine and comes within 5% of its performance at five keywords. The main reason for the performance improvement is the change in the number of comparisons required to answer the query, shown in Table 2.1. The warm-cache response time of conjunctive queries and the number of comparisons required to answer the queries were positively correlated, Pearson's $r(1200) = 0.94$, $p < 0.001$. As a result, since the row-oriented engine is able to reduce

		Comparisons (k)		
	Type	Traditional	ZigZag	Savings
2 keywords	Row	6443.30	3279.85	49%
	Col	7878.03	7434.83	6%
	IR	4199.51	3384.76	19%
3 keywords	Row	7463.84	2413.20	70%
	Col	7968.60	6574.00	18%
	IR	4962.57	2325.26	53%
4 keywords	Row	10,600.90	1828.28	83%
	Col	11,290.00	8216.25	26%
	IR	7064.65	1870.74	74%
5 keywords	Row	13,275.50	1435.54	89%
	Col	13,760.80	8269.73	40%
	IR	8859.64	1532.83	83%

Table 2.1: Number of comparisons needed to answer conjunctive queries.

the number of comparisons needed to answer 5-keyword conjunctive queries by almost 90%, it enjoys a corresponding reduction in execution time.

The column-oriented engine, however, is not able to benefit as much from ZigZag; although it can make use of tuples being filtered out at lower levels of the tree by information from higher levels, it cannot skip tuples as effectively as the row-oriented engine. While the column-oriented engine can use its metadata file to skip pages if possible, the large size of its pages means that blocks are unlikely to be skipped—in practice, every page contains at least one value of interest. Furthermore, its ability to seek within a page is limited, since its variable-width encoding means that it cannot use algorithms like binary or gallop search, and it does not have any auxiliary index on the contents of a block like the skip list structure used by the IR engine. Finally, vectorized execution is not as effective since many of the tuples in a block do not match the query and should instead be skipped.

2.5.5 Phrase queries

For generating our benchmark phrase queries, we used a modified version of the approach used in TEXTURE [33]. We selected 100 random high-frequency keywords, then enumerated all two-word

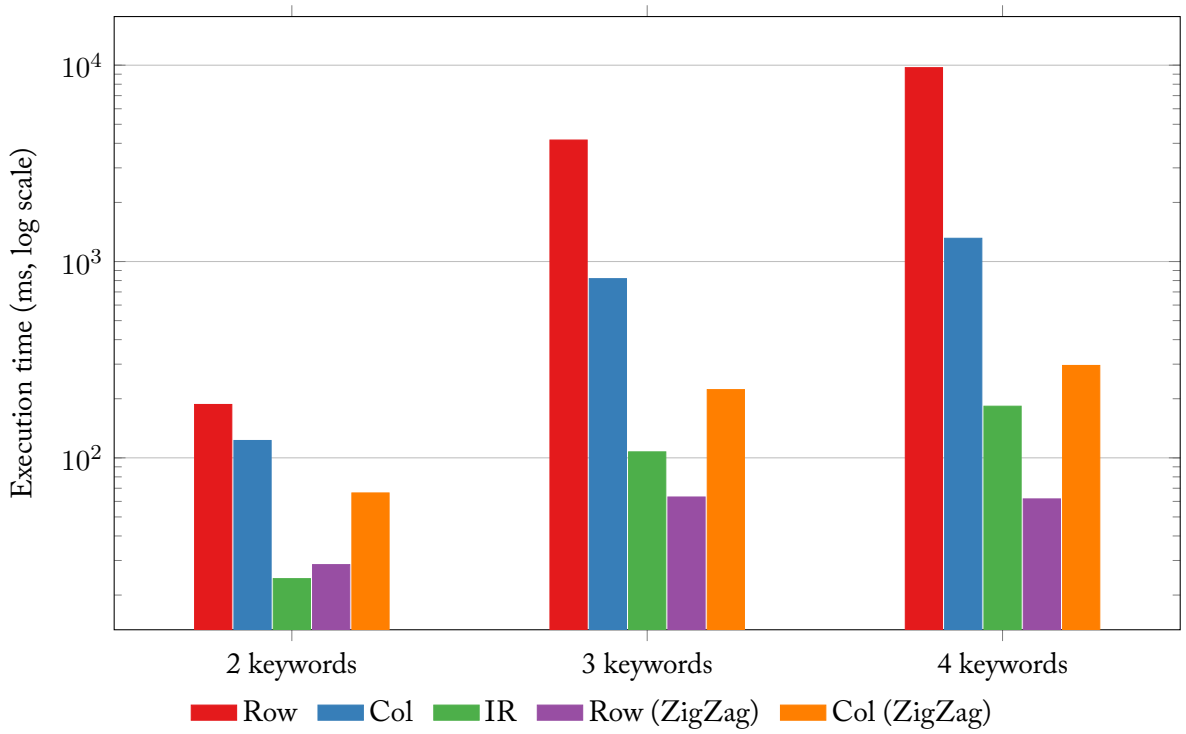


Figure 2.5: Phrase query performance with two to four keywords using multi-predicate merge join operators.

phrases that occurred in the corpus starting with one of the selected keywords. Finally, we selected 100 phrases at random from this set.

We then used these phrases as seeds for 3- and 4-keyword phrases. To generate n-word phrases, we enumerated all phrases with n words that started with the $(n-1)$-keyword phrases we had already generated. As before, we selected 100 phrases at random from this set. The average keyword had 3,809,125 postings, and the average query returned 6.9 documents.

Our results are shown in Figure 2.5, with the number of comparisons shown in Table 2.2. For the relational engines, these queries involve accessing the word offsets table to find matching documents, then joining the result set against the main index table to rank the final output. Because phrase queries involve a conjunction, much of the discussion from Section 2.5.4 still applies; however, there are several additions.

When computing a conjunctive query using the traditional merge join, the row- and column-oriented engines offered roughly the same performance. In contrast, when computing a 4-keyword

		Comparisons (k)		
	Type	MP Merge	MP ZigZag	Savings
2 keywords	Row	9382.84	1184.42	87%
	Col	2750.20	1913.67	30%
	IR	2391.19	1160.87	51%
3 keywords	Row	231,127.30	2664.24	99%
	Col	20,807.70	6480.21	69%
	IR	20,323.61	2676.56	87%
4 keywords	Row	499,260.70	2576.19	99%
	Col	34,582.35	9160.00	74%
	IR	33,991.12	2578.48	92%

Table 2.2: Number of comparisons needed to answer phrase queries.

phrase query, the column-oriented engine can offer almost an order of magnitude better performance, even without using a ZigZag operator. This is because the column-oriented engine is able to evaluate the conjunctive portion of the query directly on compressed data.

Recall that the word offsets table consists of $\langle term, docid, offset \rangle$ tuples. Our setup for the column-oriented engine uses RLE for the docid column, so when a pair of docids does not result in a match, the column-oriented engine simply avoids materializing the corresponding offsets. The row-oriented engine, on the other hand, stores offset tuples explicitly, and as a consequence, when a mismatch is encountered, the row-oriented engine will still iterate through all offsets for that docid, resulting in unnecessary comparisons.

When using ZigZag merge join, the row-oriented relational engine uses the B-tree index to skip past these offset tuples, improving its performance by two orders of magnitude for 4-keyword phrases. The column-oriented engine also benefits from ZigZag merge, but as with conjunctive queries, its limited seeking abilities do not allow it to match either the row-oriented or IR engines. Finally, we also compared the performance of multi-predicate merge join with single-predicate merge join that evaluates the offset condition as a residual.

Interestingly, we found that multi-predicate merge join improved performance only by a small amount. The reason for this is that, although the keywords we used in the phrases appear in many

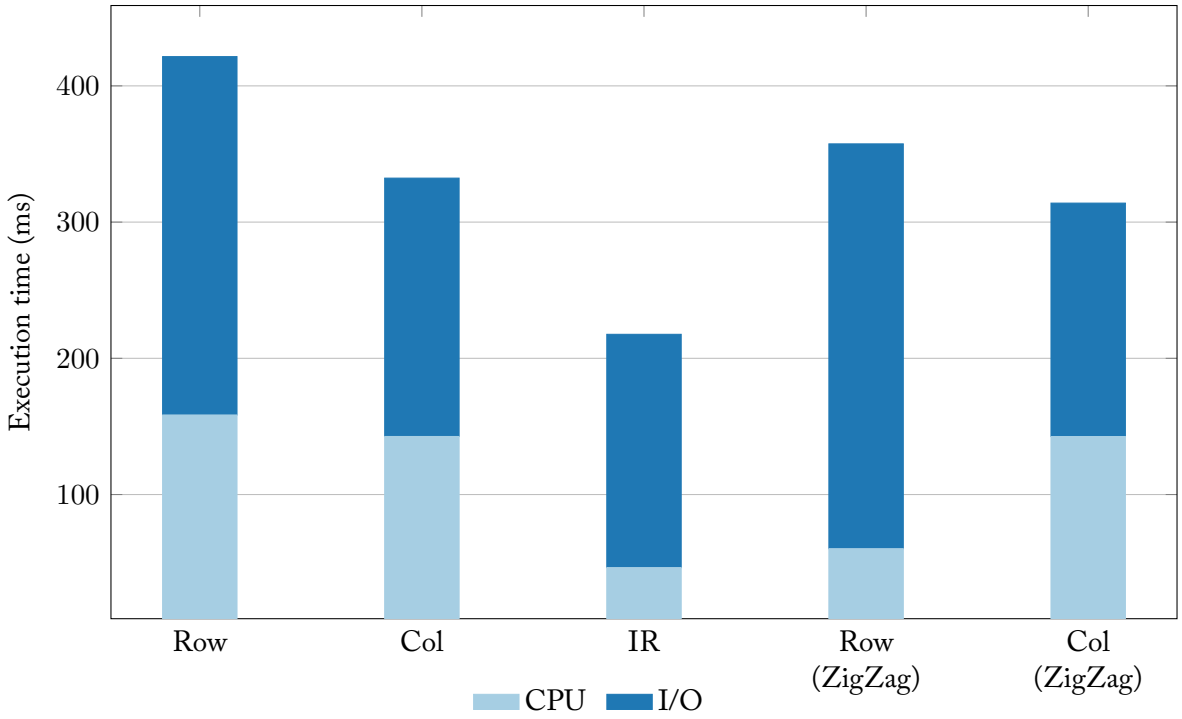


Figure 2.6: Cold-cache query performance with 3-keyword ranked conjunctive queries.

documents in the corpus, they do not occur too frequently within the same document; this is exacerbated by the fact that common words are removed as stop words. Accordingly, although the single-predicate merge join operator forms the cross product of all offset tuples in the matching documents, this product is relatively small. Since the total number of comparisons required to evaluate these queries is in the millions, the extra comparisons to filter the cross product impact performance by only 5–10%.

2.5.6 Cold-cache performance

We show the cold-cache performance of ranked conjunctive queries in Figures 2.6. We include only the results for 3-keyword queries here; other queries are similar.

The IR engine and column-oriented relational engines are heavily compressed. This gives them an advantage in execution with a cold cache, since they need to read less data from disk. However, this also means that every page of a posting list has at least one value of interest, and so every page must be read.

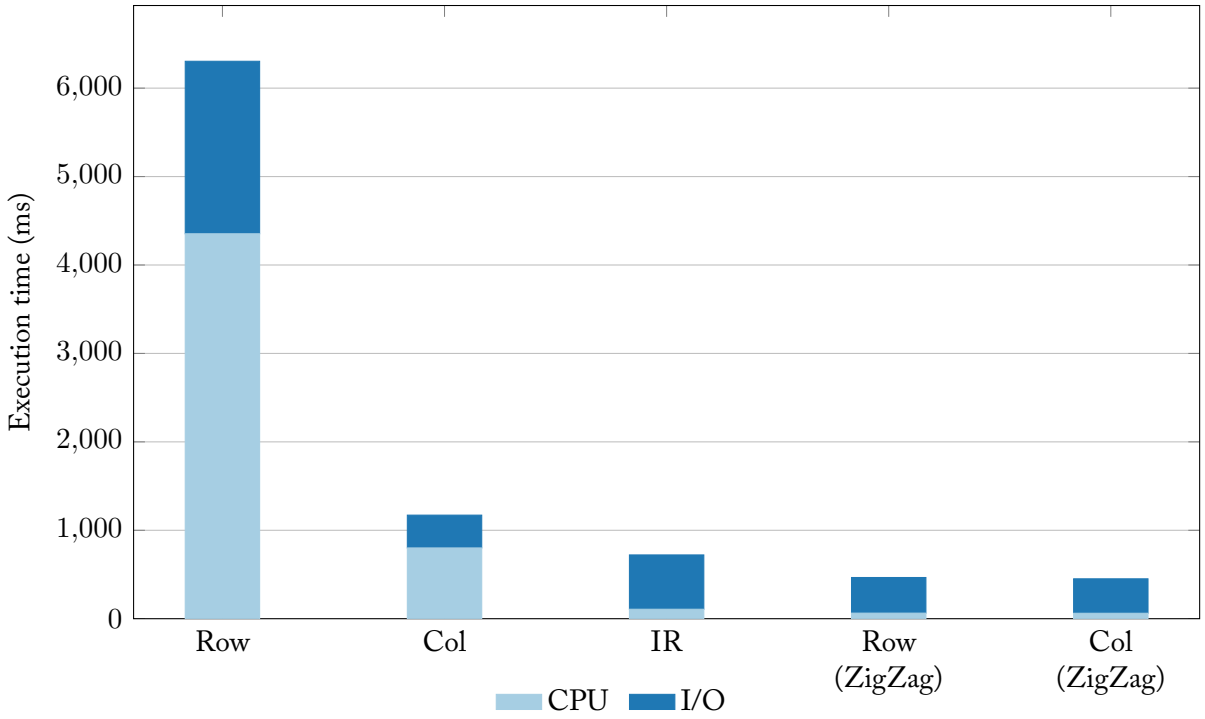


Figure 2.7: Cold-cache query performance with 3-keyword ranked phrase queries.

In contrast, the row-oriented ZigZag merge join is able to avoid reading some pages off of disk entirely by using the B-tree to skip unneeded pages. We expected this to provide an improvement to ZigZag merge's cold run time when compared to traditional merge join in a row-oriented engine. Unfortunately, in practice, the average B-tree lookup performed by ZigZag merge join for a conjunctive query skips only a small number of pages.

For 2-keyword queries, only 1 page was skipped on average, and this increased to 4 pages for 5-keyword queries. As each page is 8 kB, this results in a 32 kB jump at best. Since our data is laid out sequentially on disk, the cost of seeking past that data is similar to the cost of retrieving it, and furthermore, the jump is small enough that read-ahead mechanisms in both the OS and the RDBMS are likely to fetch the data regardless.

The story changes, however, when evaluating phrase queries. Here, ZigZag merge is able to have a substantial impact on the cold-cache performance of the row-oriented engine (see Figure 2.7). Since the word offsets table contains a tuple for each occurrence of a word, when two postings fail to intersect, the resulting index seek skips over a great deal of tuples. As a result, the number of

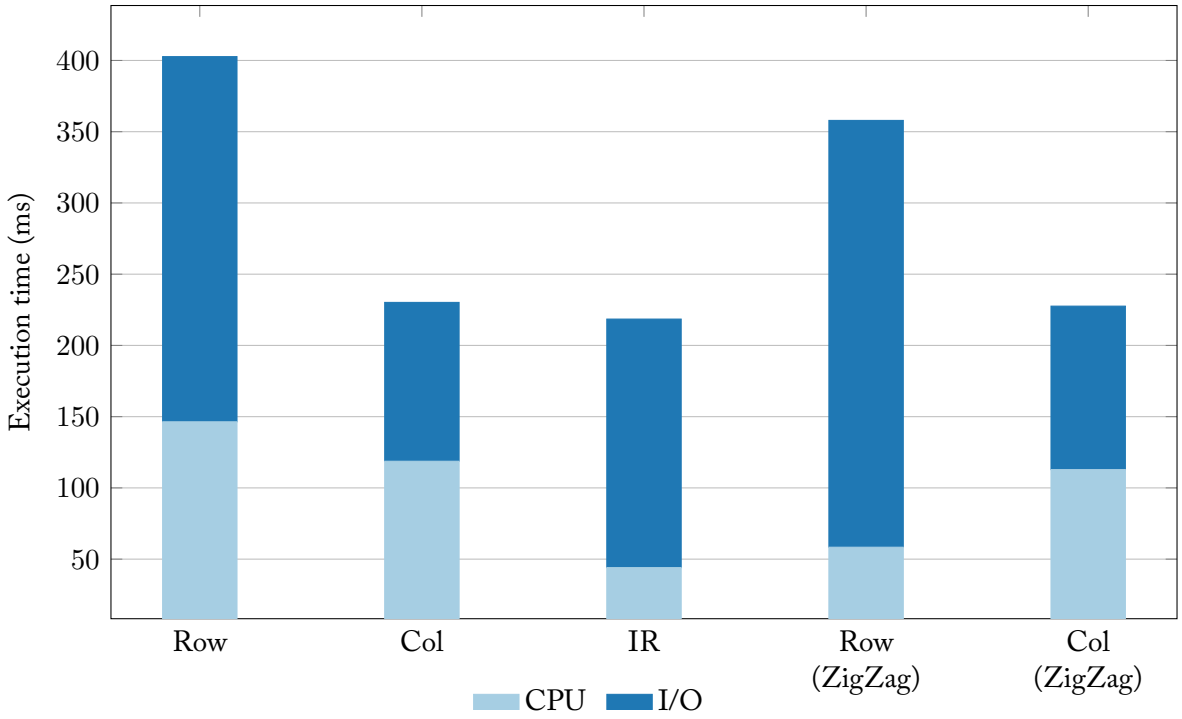


Figure 2.8: Cold-cache query performance with 3-keyword unranked conjunctive queries.

pages skipped by each index lookup is much larger for phrase queries than for conjunctive queries: 11.6 pages for 3-keyword phrases and 26.9 pages for 4-keyword phrases.

Similarly, the column-oriented engine is able to effectively skip offsets through late materialization, giving it excellent cold-cache performance. While the IR engine does not need to decode offsets for postings that fail to intersect, it stores both postings and offsets on the same page, and therefore must still read them from disk, increasing its I/O burden.

2.5.7 Unranked performance

We also evaluated conjunctive queries without ranking results. If the cache is warm, the relative performance of unranked queries is similar to that of ranked queries, since the dominant factor—number of comparisons—is unchanged. With a cold cache, however, the results are more dramatic (see Figure 2.8).

As I/O becomes a factor, data volume becomes more important. The row-oriented engine is uncompressed, and therefore has the largest amount of I/O; the IR engine is compressed, but it

stores ranking statistics in its postings, meaning it must read them from disk even if the query is unranked. In contrast, the column-oriented engine is able to avoid reading columns containing ranking statistics entirely. This allows it to perform less I/O than the IR engine, but since it requires more CPU time for comparisons, its overall performance is roughly identical to the IR engine.

2.6 Conclusion

In this chapter, we examined in-RDBMS inverted indexes in three representative settings: a specialized in-RDBMS IR engine, a row-oriented relational engine, and a column-oriented relational engine. We found that, in their default states, the specialized IR engine was able to outperform both the row- and column-oriented engines by more than an order of magnitude when processing conjunctive and phrase queries. Our results demonstrated that, when the cache is warm, the bottleneck in answering these queries is the number of comparisons.

The IR engine, due to its skip list and skipping merge operator, is able to compute an intersection with fewer comparisons than both the row- and column-oriented engines using traditional merge join. This advantage can be brought to relational database systems by using an existing algorithm, ZigZag merge join, that has not previously been applied to RDBMS text search in the research literature. We show that, by using the RDBMS's indexes more effectively, ZigZag merge join allows the row-oriented relational engine to offer performance that matches the specialized IR engine with no change in the underlying storage format. However, the performance of the column-oriented relational engine in this case—while improved by ZigZag merge join—is limited by its poor seek performance, the result of minimal secondary indexes and encoding formats that prevent fast forward searches.

When the cache is cold, data volume is the dominant factor. The IR engine and the column-oriented engine are heavily compressed compared to the row-oriented engine, and this gives them a noticeable performance advantage for cold queries. For ranked queries, the IR engine provides the best cold-cache performance, since it is compressed and stores all columns necessary for ranking

together, while the column-oriented engine must perform extra work to materialize the ranking columns. In contrast, for unranked queries, the IR and column-oriented engines offer similar performance; the column-oriented engine does less I/O, but requires more CPU. While ZigZag merge join is effective at reducing the CPU cost of both conjunctive and phrase queries, it is able to improve I/O performance only for phrase queries, since only phrase queries result in index seeks that skip enough pages.

Although our results are most applicable to processing text workloads in an RDBMS, we feel they may also have broader implications. Since the operations the specialized IR engine performs can be expressed as SQL queries, the IR engine is obviously capable of providing extremely high performance for a certain class of queries. In current systems, this functionality is locked away—used only for text search. In the next chapter, we evaluate the effect that this functionality could have if extended to more general relational queries.

3 | Extending skipping join algorithms to relational data

3.1 Introduction

In Chapter 2, we showed that the idea of skipping join algorithms, and ZigZag merge join in particular, can have a large impact on the performance of full-text search queries in an RDBMS. While full-text search queries are useful, they are a small subset of all the queries that one might pose to an RDBMS. Furthermore, they have a specific structure—a series of self-joins with predicates on each relation. As a result, it raises a natural question: are ZigZag merge join and other skipping join algorithms effective when applied to general relational queries?

Now, certainly, the idea of skipping join algorithms in general relational settings is not new. Even simple algorithms like traditional merge join and index nested loops join perform a degree of skipping in one relation, and ZigZag merge join itself has been implemented in at least one commercial RDBMS [7]. However, apart from the baseline merge and index nested loops joins, there has been no thorough examination of the impact a more sophisticated skipping join algorithm can have on performance.

Accordingly, in this chapter, we extend our implementation of ZigZag merge join to handle general relational queries, and we compare its performance with traditional join algorithms on two standard RDBMS benchmarks, Star Schema Benchmark (SSB) [76] and TPC-H [93]. We show that ZigZag merge join can meet or exceed the performance of traditional join algorithms on many SSB and TPC-H queries, sometimes improving performance by an order of magnitude or more. By

themselves, these performance results provide empirical evidence that skipping join algorithms can be effective, but they do not provide a rigorous foundation for why this is the case.

To address this, we incorporate new ideas from recent work in the database theory community. Inspired by the concepts of *certificates* [11] and *proofs* [30] in set intersection, database theoreticians have proposed *join certificates* as a measure of an algorithm's complexity [73]. Join certificates are a set of comparisons between tuples that certify the output of a join is correct; i.e., each comparison in the certificate rules out a portion of the join key value space.

Every comparison-based join algorithm, like merge join, implicitly creates a certificate as it executes, and there can be many different certificates for a single join. However, the *optimal certificate* is the smallest possible certificate, i.e., the certificate with the fewest comparisons. Optimal certificates are at most linear in the input size, and in some cases, they can be constant—consequently, a join algorithm that operates in time proportional to the size of the optimal certificate can potentially be very fast even for large inputs.

We analyze ZigZag merge join using this framework and show that, for a join between two sorted lists of join key values (i.e., two relations), ZigZag merge join executes one index seek per comparison in the optimal join certificate. This result explains ZigZag merge join's performance advantage over traditional merge join; however, we also show that for joins involving more than two relations or multiple sorted lists of join key values, ZigZag merge join is not optimal. This is due to the fact that multiple ZigZag merge joins are implemented as a tree of binary ZigZag merge join operators.

Information from joins later in the tree may eliminate a large portion of the join key space, but those joins will only be accessed after a match is found in an earlier join. In extreme cases, later joins may eliminate the *entire* join key space, which means that any work done to find a match in earlier joins is completely wasted. In contrast, a newly proposed join algorithm, Minesweeper [73], is optimal for these cases.

Minesweeper extends ZigZag merge join by adding multi-way operation and the ability to remember regions of the join space that have been ruled out, potentially saving work if a relation

is accessed repeatedly in the course of executing a query. In theory, these two features enable Minesweeper to handle some queries more efficiently than ZigZag merge join, but there are no published empirical results that investigate whether Minesweeper is effective in practice. Accordingly, we implemented a version of Minesweeper and evaluated its performance relative to ZigZag merge join.

Our results on SSB and TPC-H show that while Minesweeper can outperform traditional join algorithms in some cases, it never outperforms ZigZag merge join in the generic database benchmarks we tested. On the other hand, we demonstrate that on specially designed queries and data, Minesweeper can be arbitrarily faster than ZigZag merge join. This suggests that, although there may be queries where Minesweeper provides better performance than ZigZag merge join, these cases are not tested by standard benchmarks, and they may not be common in practice.

This chapter provides the following main contributions:

- A description of changes needed to extend ZigZag merge join to relational queries without requiring sorting of intermediate results.
- An analysis of ZigZag merge join's complexity in terms of optimal join certificates.
- An discussion of how to implement Minesweeper, a theoretical skipping join algorithm, in practice.
- A comparison of ZigZag merge join and Minesweeper that includes a discussion of the cases where Minesweeper is expected to outperform ZigZag merge join.
- An analysis of the performance of ZigZag merge join, Minesweeper, and traditional join algorithms on generic relational queries and special cases.

The rest of the chapter is organized as follows. We describe our problem setting and the notion of certificates in Section 3.2. We discuss the details of extending ZigZag merge join to relational queries and analyze its complexity in Section 3.3, and we give a brief overview of Minesweeper and our implementation of it in Section 3.4. We compare ZigZag merge join and Minesweeper's design

in Section 3.5, and we show their performance in a variety of situations in Section 3.6. Finally, we conclude in Section 3.7.

3.2 Background

In this work, we consider queries consisting of a series of natural joins. Each query has a set of *join attributes* $A = \{A_1, \ldots, A_n\}$, and a set of relations $R = \{R_1, \ldots, R_n\}$, each of which contains some subset of join attributes. We use the notation $R(A_1, \ldots, A_k)$ to describe a relation R with attributes $\{A_1, \ldots, A_k\}$, and for brevity, we omit any attributes that do not participate in the join. We call a list of ordered values a *sorted run*, or simply a *run*.

When describing queries, we use standard relational algebra notation. $R \bowtie S$ denotes a natural join between relations R and S, $\pi_{A_1}(R)$ denotes the projection of attribute A_1 from relation R, and $\sigma_{A_1=a}(R)$ denotes the selection of all tuples from R where A_1 has the value a.

Order and indexes. We assume that all relations are sorted and stored in a *global attribute order* [73], and more specifically, the global attribute order is $A_1, \ldots, A_n$ (i.e., a relation $R(A_1, A_2)$ is sorted on (A_1, A_2)). Furthermore, we assume that each relation has an appropriate index on its data consistent with the global attribute order, such as a B-tree, that enables efficient seeks. These are the same assumptions made by Ngo et al., but note that they do place large restrictions on when these algorithms can be used.

Obviously, if data is unsorted with no indexes, it is very difficult to skip data efficiently, and skipping join algorithms are likely to be ineffective. On the other end of the spectrum, the join result may be precomputed and stored, which clearly enables "optimal" skipping in that only output tuples will be accessed. The interesting ground for a discussion about skipping is in between these two extremes, and we focus on a read-optimized case where appropriate indexes exist.

Certificates. Traditional worst-case guarantees do not accurately capture the efficiency of skipping join algorithms. The ability of a join algorithm to skip is dependent on the input data, but in the worst case, no skipping may be possible. As a result, in a traditional worst-case runtime

analysis, skipping join algorithms are just as bad as those that *never* skip, but clearly this fails to consider cases where the algorithm may return the result without reading all of the data.

To address this, Ngo et al. introduce the notion of a *certificate* for joins [73]. Join certificates consist of a set of comparisons between tuples that certify the output of a join is correct. The size of a certificate is the number of comparisons it contains, and the *optimal certificate* for a given join is the smallest certificate. We denote the optimal certificate for a given join $R_1 \bowtie \cdots \bowtie R_k$ as $C(R_1 \bowtie \cdots \bowtie R_k)$, and we use $|C|$ to indicate the size of the optimal certificate.

Join certificates are described using the relative order of tuples in relations, and to simplify our later discussion, we borrow Ngo et al.'s notation, which we describe informally here for completeness. Given a relation $R(A_1, \ldots, A_k)$, an *index tuple* x for R is a tuple of positive integers $x = (x_1, \ldots, x_j)$ such that $j \leq k$. If x consists of a single attribute, say $x = (x_1)$, then $R[x]$ refers to the x_1'th smallest A_1 value in R. If x consists of multiple attributes, say $x = (x_1, x_2)$, then $R[x]$ refers to the x_2'th smallest A_2 value in $\sigma_{A_1=R[x_1]}(R)$, and so on for larger x tuples.

Ngo et al. also define three special cases. Given an index tuple $x = (x_1, \ldots, x_{j-1})$, $R[x, 0]$ is $-\infty$, $R[x, *]$ is the set of A_{j+1} values in $\sigma_{A_1=R[x_1],\ldots,A_j=R[x_1,\ldots,x_{j-1}]}(R)$, and $R[x, |R[x, *]|+1]$ is $+\infty$. For example, suppose we have the relation $R(A_1, A_2) = (1, 1), (1, 3), (2, 7), (2, 10)$. In this case, $R[1] = 1$, $R[*] = \{1, 2\}$, $R[1, *] = \{1, 3\}$, $R[1, 3] = +\infty$, $R[2] = 2$, and $R[2, 1] = 7$.

To make this concrete, we apply this notation to an actual join. Consider a join between two sorted relations, $R(A) \bowtie S(A)$, and suppose that R contains values $[1, \ldots, N]$ and S contains values $[N + 1, \ldots, 2N]$. The output of the join is empty, and a certificate for this output would be a single comparison: $R[N] < S[1]$ or, in other words, the last value in R is smaller than the first value in S. Note that this certificate is independent of the size of the input—N could be arbitrarily large, but the certificate would still be a constant size.

In the worst case, join certificates can be linear in the input size. Consider the same join as above, but now suppose that R contains values $[1, 3, \ldots, 2N + 1]$ (all odd values) and S contains values $[2, 4, \ldots, 2N]$ (all even values), resulting in this minimum-size certificate: $R[1] < S[1], S[1] < R[2], R[2] < S[2], \ldots, R[N - 1] < S[N], S[N] < R[N]$. Again the output of the join is empty,

but the certificate for this join has $2N - 1$ elements, since the perfect interleaving of values means that no comparison eliminates more than a single value.

As a result, join certificates capture a finer notion of complexity than traditional worst-case analysis. We now examine how ZigZag merge join can be applied to relational data and how its processing relates to the optimal certificate for a given join.

3.3 ZigZag merge join

In this section, we consider a binary ZigZag merge join operator. This operator executes over two relations, an *outer* relation and an *inner* relation. Because ZigZag merge join seeks in both of its children, the distinction between the outer and inner relations is not as pronounced as in other join algorithms, and we introduce it mostly for clarity of discussion.

In order to apply ZigZag merge join to general queries over relational data, we must make some modifications to the algorithm presented in Algorithm 2. First, when computing an intersection, all join key values are unique in each relation; however, this is not true for all queries, and ZigZag merge join must accommodate this fact. Second, intersections have common join attributes across all tables (e.g., for conjunctive full-text search queries, the only join attribute is docid), while many queries (e.g. queries over many-to-many relationships) have some join attributes that appear only in a subset of relations. We discuss each of these issues in turn.

Non-unique join key values. Multiple tuples within a relation may have the same join key value, forming a subset. When a subset of tuples in the outer relation joins with a subset of tuples in the inner relation, ZigZag merge join must output the Cartesian product of the two subsets. One approach to handling this is to buffer the subsets in memory, then emit the Cartesian product, but this can require large amounts of memory in the case of large subsets.

Instead, we opted to implement ZigZag merge join such that it rewinds iterators as needed. Consequently, when a match is found, ZigZag merge join marks the beginning of the subset in the inner relation. For each tuple in the outer relation's subset, ZigZag merge join outputs all tuples in

the inner relation's subset, rewinding to the beginning of the inner subset at the beginning of each iteration. This continues until all tuples in the outer relation's subset have been processed.

Distinct join attributes. When processing a conjunctive full-text query, all relations in the join have a single join attribute: docid. As a result, any value learned from any relation participating in the join can be used to seek in any other relation. However, in practice, many queries have join attributes that do not appear in all relations.

For example, consider a three-relation join: $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2)$. We implement this using ZigZag merge join as a tree of pipelined binary joins, effectively computing $(R(A_1) \bowtie S(A_1, A_2)) \bowtie T(A_2)$. In this case, T is only accessed after finding a match in $R \bowtie S$, and join key values read from T can be used only for seeks on S, not on R.

Additionally, only the initial join between R and S is a join between two sorted runs; it produces a series of sorted runs of A_2 values, one per matching A_1 value. Since the join with T is on A_2, we are left with two options: sort the intermediate result of $R \bowtie S$ on B or modify ZigZag merge join to accommodate multiple sorted runs.

Sorting the intermediate result of $R \bowtie S$ is undesirable for multiple reasons. First, it introduces a blocking operator into what was formerly a streaming pipeline, increasing the time necessary to produce the first result to the query. Second, it limits ZigZag merge join's ability to seek within the intermediate result unless a temporary index is constructed while performing the sort—further increasing the work needed to evaluate the join. Finally, given that the entire intermediate result must be materialized before ZigZag merge join can execute, any benefit that may be derived from skipping over non-matching tuples in the result is offset by the fact that all tuples were read at least once during the sorting phase. Consequently, we chose to modify ZigZag merge join so that it could execute over multiple sorted runs in-place.

In order to process a series of sorted runs, we make use of a simple observation. In the case of $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2)$, for a given value of A_1, the join between S and T has exactly one sorted run of A_2 values from S. Accordingly, ZigZag merge join can be run on these values without modification. When a new A_1 value is encountered, this marks the beginning of a new sorted run,

Method	Purpose
HasLeftRun(*operator*)	Returns true if *operator* has set a flag indicating it has left its run of join values or false otherwise.
SetLeftRun(*operator*)	Sets the flag on *operator* that indicates it has left its run of join values; after this is called, subsequent calls of HasLeftRun(*operator*) will return true.
ClearLeftRun(*operator*)	Clears the flag on *operator* that indicates it has left its run of join values; after this is called, subsequent calls of HasLeftRun(*operator*) will return false.

Table 3.1: Utility methods used by ZigZag merge join.

and the iterator on T must be rewound to the beginning of the table before ZigZag merge join can continue.

To support this, the ZigZag merge join over T must be aware of when a new run has been encountered. In our case, we add three methods to the ZigZag merge join operator, HasLeftRun, SetLeftRun, and ClearLeftRun (see Table 3.1). When a ZigZag merge join operator leaves a run, it calls SetLeftRun. This communicates to the parent ZigZag merge join operator (if any), that a rewind is necessary. When the parent operator sees that flag, it performs the rewind and calls ClearLeftRun to ensure the rewind is not repeated unnecessarily.

If there are multiple many-to-many joins in a single query (e.g., $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2, A_3) \bowtie U(A_3)$), then multiple rewinds may be necessary. This is required because a change in A_2 value will result in a rewind on U, and a change in A_1 value will result in a rewind on both T and U. Accordingly, we keep one rewind flag per ZigZag merge join operator.

We show the full algorithm for ZigZag merge join over arbitrary relational queries in Algorithm 4, with supporting functions described in Appendix A.

ZigZag merge join and certificates. For a two-relation join, $R(A_1) \bowtie S(A_1)$, ZigZag merge join executes $O(|C|+Z)$ comparisons, where C is an optimal certificate and Z is the size of the output. The reasoning for this is straightforward.

Algorithm 4 ZigZag merge join algorithm for $R \bowtie S$.

```
in_group ← false
left_seeked ← false
right_seeked ← false
done ← false
results ← ∅
repeat
    output ← false
    if in_group then
        ProcessGroup
    else
        r ← GetNext(R)
        s ← GetNext(S)
    if not output then
        RewindIfNeeded
        while not output and r ≠ End(R) and s ≠ End(S) do
            if r = s then
                add r ⋈ s to results
                output ← true
            else if r < s then
                Seek(R, s)
                GetNext(R)
            else
                Seek(S, r)
                GetNext(S)
            if not output then
                RewindIfNeeded
        if not output then
            done ← true
until done
return results
```

Every tuple in R and S must be accounted for either by a comparison in C that rules it out or by the fact that it is in the output. ZigZag merge join begins by inspecting the first tuple in R and comparing it with the first tuple in S. If they are equal, they are output tuples and they are charged to Z. If they are not equal, ZigZag merge join performs an index seek for the larger of the two values. This seek corresponds to a comparison in C, i.e., if $R[1] > S[1]$, the index seek must return some other value, say $S[x]$. The semantics of an index seek are such that it returns tuple in S with the smallest join key value that is greater than $R[1]$, and thus, the index seek eliminates as many

S values as possible. Therefore, $R[1] > S[x-1]$ must be in the optimal certificate, and a similar argument applies for subsequent index seeks.

For joins involving multiple relations, ZigZag merge join can do arbitrarily more work than the optimal certificate. For example, consider a three-relation join, $R(A_1) \bowtie S(A_1) \bowtie T(A_1)$. Suppose the relations as populated as follows:

- R: $[1, 3, 5, \ldots, 2N+1]$ (i.e., odd numbers)
- S: $[2, 4, 6, \ldots, 2N]$ (i.e., even numbers)
- T: $[2N+2, 2N+3, \ldots, 3N]$

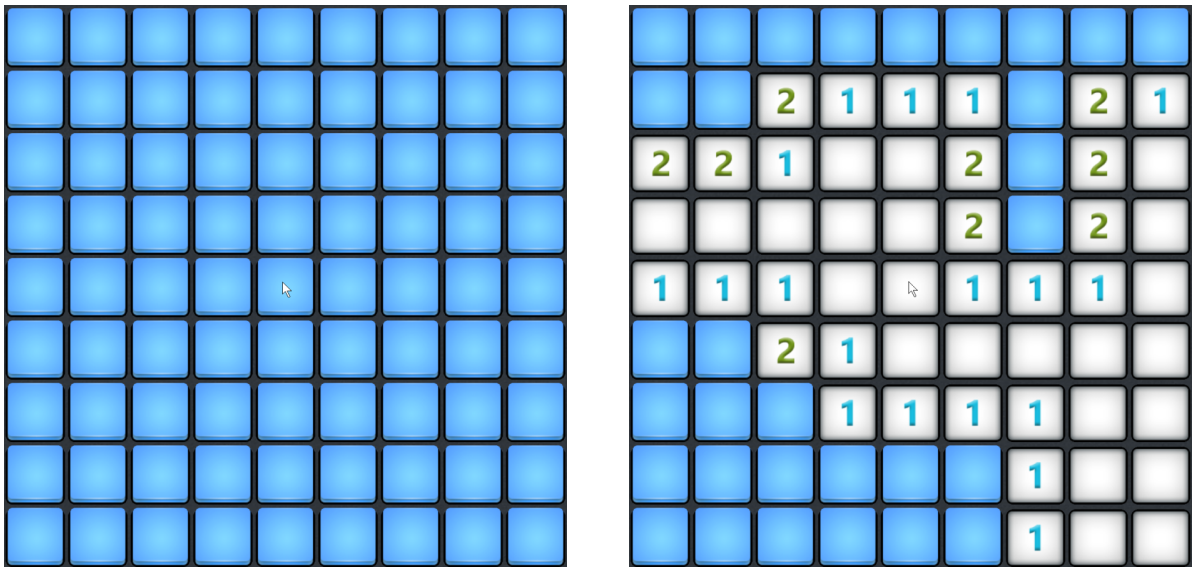
The optimal certificate for this instance is $T[1] > S[N]$, but ZigZag merge join will compute the entire certificate of $R \bowtie S$ without ever attempting to join T.

Astute readers will note that ZigZag merge join could address this issue by performing $S \bowtie T$ first, instead of $R \bowtie S$. This is correct, but it is not immediately clear how to determine the optimal join ordering a priori. Traditional metrics for determining join order, such as cardinality, are ineffective since T can be arbitrarily large without affecting the result of the query. We revisit this issue in Chapter 4.

3.4 Minesweeper

Ngo et al. introduce Minesweeper, a state-of-the-art skipping join algorithm [73]. Minesweeper is a multi-way algorithm that processes all relations in a query simultaneously, even for queries with multiple join attributes, and it provides strong worst-case guarantees in terms of the size of the optimal certificate for an instance. The key idea behind Minesweeper is similar to the principle in how ZigZag merge operates.

Namely, adjacent tuples in a sorted relation form *gaps* where no possible output tuples exist. For example, given a relation $R(A_1) = \{1, 2, 3, 7\}$, the 3 and 7 tuples indicate that there cannot be any output tuples with $A_1 \in (3, 7)$. When joining R with another relation S, ZigZag merge join



(a) Before probe. **(b)** After probe.

Figure 3.1: A "probe" on the center square in Microsoft Minesweeper.

exploits this information to perform an index seek in S when 7 is encountered, potentially skipping many tuples.

In contrast, Minesweeper approaches the problem from a different direction. Minesweeper repeatedly probes all relations in a query simultaneously, performing index seeks for arbitrarily chosen join attribute values. Each seek tells Minesweeper one of two things. First, the join attribute value may be present in the relation; if all seek values are present in all relations, then the tuple is a match and is output. Alternately, the value may be absent, in which case the index seek returns the join key values from the tuples immediately preceding and following the seek value, indicating a gap.

Minesweeper then remembers every gap it has encountered for each join key attribute, allowing it to rule out regions of the join key space. Therefore, with each iteration, Minesweeper rules out more and more of the output space until it can determine that it has explored all possibilities, at which point the algorithm terminates. Accordingly, Minesweeper can be viewed as a variant of index nested loops join that never executes an index seek that it has already ruled out. This is analagous to the gameplay of Microsoft Minesweeper [70], where players probe squares on a grid to find gaps where no mines exist (see Figure 3.1).

For the following description, we assume a natural join query with join attributes $A = \{A_1, \ldots, A_n\}$.

Constraints. When Minesweeper starts executing a query, it knows nothing about the distribution of data in the join relations. However, as it accesses each relation, it observes gaps in the data, and it encodes these gaps as *constraints*. Constraints are n-dimensional vectors with one component per join attribute. Each component in a constraint is associated with the corresponding (pairwise) join attribute, and each component in a constraint consists of one of three types: an equality, a wildcard, or an interval.

Equalities are literal values, and they denote that the constraint is only valid when the corresponding join attribute is equal to that value. Wildcards (denoted as $*$) place no restriction on their corresponding join attribute. Finally, intervals indicate regions of the join space that have been eliminated, given the equality conditions previously noted in the constraint.

Each constraint has exactly one interval component. All components after the interval components are wildcards, and when describing a constraint, we omit them. Components prior to the interval component may be either literals or wildcards, depending on how broadly the constraint applies.

Although one might think that Minesweeper needs only to track simple intervals, much of what Minesweeper learns is conditioned on some value of the join attribute. For example, consider a relation $R(A_1, A_2)$ in an arbitrary query. Suppose we probe R for $(1, 4)$ and find a gap denoted by two tuples, $(1, 2)$ and $(1, 9)$. We have now learned an interval on A_2, namely $(2, 9)$, but recall that R consists of a series of sorted runs of A_2. Our observation is valid only for the *current* run of A_2 values, where $A_1 = 1$, and the gap cannot be used to rule out values where $A_1 \neq 1$. The corresponding constraint would consist of $\langle 1, (2, 9)\rangle$.

Efficiently storing and combining constraints is a critical part of Minesweeper's performance. To facilitate this, Ngo et al. present an optimized constraint data structure (CDS) that stores constraints using a prefix tree [73], optimized for compact storage and fast lookup performance, and our implementation follows their design. Minesweeper must also accumulate potential new constraints

as it conducts probes, and we use a similar prefix tree structure for storing these intermediate constraints.

Probe points and index seeks. On each iteration, Minesweeper must determine the join attribute values to seek for. Given a query with join attributes $A = \{A_1, \ldots, A_n\}$, Minesweeper creates a tuple with literal values $x = x_1, \ldots, x_n$ called a *probe point*. Probe points are generated such that Minesweeper never examines a point in the join space that has been ruled out by an existing constraint, ensuring it will not waste time repeatedly probing an area of the space that has already been eliminated.

Given a probe point, Minesweeper executes a number of index seeks. Recall that a probe point contains values for each unique join attribute in the query, but a specific relation in the query may contain only a subset of these attributes. As a result, Minesweeper must decompose the probe point values into seek keys appropriate for each individual relation.

For example, suppose we are computing $R(A_1) \bowtie S(A_1, A_2, A_3) \bowtie T(A_2) \bowtie U(A_3)$. The join attributes for this query are $\{A_1, A_2, A_3\}$, and Minesweeper must execute at least three index seeks for each probe point it investigates in this case, one for each relation. Seeks against R use A_1; seeks against S use A_1, A_2, and A_3; seeks against T use A_2; and seeks against U use A_3. Accordingly, a probe point $x = x_1, x_2, x_3$ will seek for x_1 in R, (x_1, x_2) in S, (x_2, x_3) in T, and x_3 in U.

Performing only one seek per relation is sufficient to produce correct and complete output, but it may not eliminate much of the join space per iteration. To find more gaps per iteration, Minesweeper performs multiple index seeks based on prefixes of a relation's join attributes. If a relation has only one join attribute, Minesweeper will naturally execute only a single index seek. However, for relations with multiple join attributes, Minesweeper will initially perform a seek with only the first join attribute, finding the low and high key surrounding the first attribute (i.e., the gap). The low and high key may be equal in the case where the first attribute of the join key is present in the relation. For each unique key discovered by the first seek, Minesweeper executes a second seek using the second join attribute prefixed by the key from the first seek. This process repeats for all join attributes, resulting in a total of $2^k - 1$ seeks for a relation with k join attributes.

To illustrate this, consider a probe point for the above query, $(2, 7, 4)$. Minesweeper will issue an index seek on S for (2); suppose this seek returns the gap $(1, 3)$. Minesweeper will then issue two more seeks on S, one for $(1, 7)$ and a second for $(3, 7)$, and the results from these seeks will be used to form the final 3-attribute seeks. This behavior allows Minesweeper to potentially eliminate very large regions of a relation in a single iteration at the cost of possibly performing more seeks than necessary.

Probe point selection. In principle, probe points can be arbitrarily chosen and checked against the CDS. However, for efficiency reasons, Minesweeper populates values for a probe point from left to right, starting with the first attribute in the global attribute order. In effect, Minesweeper finds the smallest value for A_1 that is not ruled out by a constraint, then the smallest value for A_2 not ruled out by a constraint given the value chosen for A_1, and so on, until the probe point is fully populated.

There are cases where, given the values selected for earlier join attributes, no value can be found for a later join attribute. Assuming that there are other possible values for the earlier attribute, this means that Minesweeper has not explored the entire space, and therefore cannot assume the join is empty. In this instance, Minesweeper inserts a new constraint that eliminates this path from future consideration, and it then backtracks to the most recently chosen attribute value and selects a new value given the new constraint. This ensures that Minesweeper will continue making forward progress without wasting a great deal of effort exploring dead-end paths.

Implementation. We show the pseduocode for our implementation of Minesweeper in Algorithm 5 with related utility functions described in Table 3.2. Our implementation is largely based on the description provided by Ngo et al., extended to output the Cartesian product of matching tuples as needed.

We also encapsulate the process of performing index seeks in a function, FindBounds. FindBounds executes all appropriate seeks against each relation, and it stores any intermediate constraints it discovers in the process. These constraints are not used if the probe point finds an output tuple, but they must be formed at the time the seeks are performed. For efficiency reasons, we utilize a

Method	Purpose
GetProbePoint	Returns a probe point that has not been ruled out by any constraints in the CDS or $\varnothing$ if no such probe point exists.
MatchesProbe($probe, tuple$)	Returns true if *tuple*'s join attributes are equal to the corresponding values in $probe$ or false otherwise.
FindBounds($probe, relation, constraints$)	Returns the two largest (smallest) tuples in $relation$ whose join attribute values are smaller (larger) than the corresponding values in $probe$ and populates $constraints$ with all constraints learned.

Table 3.2: Utility methods used by Minesweeper.

prefix tree structure similar to the CDS to store these constraints.

3.5 Comparing ZigZag merge join and Minesweeper

Minesweeper adds two features over ZigZag merge join: *multi-way operation*, where all relations are joined simultaneously, and *constraint memory* of gaps (and consequently tuples) it has explored, stored in the CDS. Accordingly, we expect Minesweeper to do well in situations where these features are important, and less well where these features are unnecessary, since there is some overhead in maintaining the CDS. The importance of these features varies depending on the query and the data, and we now examine these cases in more detail.

When neither feature is important. Consider a join between two relations, $R(A_1) \bowtie S(A_1)$ with a single join attribute, A_1. Multi-way operation is obviously not helpful, since the join has only two relations, and constraint memory is not helpful since neither relation is ever rewound—each tuple is processed (or skipped) one time and never seen again. In fact, for this join, ZigZag merge join and Minesweeper have a similar pattern of index seeks.

ZigZag merge join begins by reading the first tuple of R and S and comparing their join key values. If the values are equal, it will output the tuple and advance both iterators, while if the join

Algorithm 5 Minesweeper algorithm for $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$.

```
in_group ← false
done ← false
results ← ∅
repeat
    output ← false
    if in_group then
        i ← k
        while i > 0 and not output do
            r_i ← GetNext(R_i)
            if r_i ≠ End(R_i) and MatchesProbe(probe, r_i) then
                add r_1 ⋈ r_2 ⋈ ··· ⋈ r_k to results
                output ← true
            else
                rewind R_i to the beginning of the group
                r_i ← GetNext(R_i)
            i ← i − 1
        if not output then
            in_group ← false
    if not output then
        probe ← GetProbePoint
        if probe is empty then
            done ← true
        else
            constraints ← ∅
            is_match ← true
            i ← 1
            while i ≤ k do
                r_i ← FindBounds(probe, R_i, constraints)
                if is_match then
                    is_match ← MatchesProbe(probe, r_i)
            if is_match then
                add r_1 ⋈ r_2 ⋈ ··· ⋈ r_k to results
                output ← true
                in_group ← true
                add constraint that excludes probe to the CDS
            else
                for each c in constraints do
                    add c to the CDS
until done
return results
```

key values are different, it will seek for the larger value in the relation whose tuple has the smaller value. All seeks are based on the largest value that has been encountered so far, and this process continues until one iterator is exhausted, at which point ZigZag merge join stops.

In contrast, Minesweeper will begin by seeking for $-\infty$ in R and S, which will return the first tuple of each relation. If the tuples have equal join key values, Minesweeper will output the tuple and insert a constraint to rule out that specific join key value. If the tuples have different join key values, Minesweeper will insert two constraints into the CDS, ruling out any join key values smaller than those it just read. Since there is only one join attribute, these constraints combine, and consequently Minesweeper's next probe point will be the larger of the two values, identical to the seek ZigZag merge join performed.

Therefore, Minesweeper performs largely the same index seeks as ZigZag merge join with three exceptions. First, when accessing the initial tuple in each relation, Minesweeper performs two index seeks for $-\infty$, whereas ZigZag merge join merely reads the first tuples from the iterators over R and S. Second, when performing a seek for the larger of two values, Minesweeper will seek for that value in *both* R and S, even though one relation must already have its iterator positioned appropriately. Third, when a match has been found, instead of simply advancing both iterators, Minesweeper will execute a seek for the next largest value after the match.

The overhead from these additional seeks can be removed by virtue of an intelligent index implementation that recognizes when an index seek is not necessary (e.g., if the iterator is already positioned at the seek key). However, Minesweeper also stores constraints for all tuples encountered in the CDS, and maintaining the CDS has an impact on performance. Accordingly, we would expect that ZigZag merge join would perform better in this case.

When multi-way operation is important. An n-relation query ($n > 2$) with a single join attribute places more importance on multi-way operation. Binary join operators, like ZigZag merge join, begin by joining two relations, and they attempt to join additional relations only after first finding a matching tuple from the initial join. In contrast, multi-way operation allows a join algorithm to look into "the future," potentially using information from later relations to filter earlier

relations regardless of whether a match has been found.

This ability allows a multi-way join operator to skip significantly more data than a binary join operator in some cases. For example, consider a three-relation join, $R(A_1) \bowtie S(A_1) \bowtie T(A_1)$, and suppose R contains $[1, 3, 5, \ldots, 2N + 1]$, S contains $[2, 4, 6, \ldots, 2N]$, and T contains $[2N + 2, 2N + 3, \ldots, 3N]$. Here, Minesweeper will recognize after two iterations that the join output is empty, since its probe in T will exclude any value smaller than $2N + 2$. ZigZag merge join will attempt to first join R and S, waiting to probe T until a match is found; since no match exists, it will consume the entirety of R and S before terminating. Consequently, Minesweeper can be arbitrarily better than ZigZag in this scenario.

Note that multi-way operation is only effective when there are multiple relations joined on a single attribute. If a join has multiple join attributes such that a given attribute is shared by a maximum of two relations (e.g., $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2)$), then multi-way operation has no benefit by itself. In this example, values learned from $S \bowtie T$ provide information only about the domain of A_2, which does not apply to $R \bowtie S$.

When constraint memory is important. An n-ary join (with $n > 2$) with multiple join attributes benefits from constraint memory. Consider a three-relation join, $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2)$, with two join attributes, $\{A_1, A_2\}$. The presence of multiple join attributes means that rewinds will be necessary—in particular, the iterator on T will be rewound for each unique A_1 value. As a result, constraint memory can have an impact, as tuples in T will be processed multiple times.

For each A_1 value, Minesweeper sees a list of A_2 values from S that explore some portion of the space in T. In the best case, the A_2 values from S cover a range of T that has been explored previously (i.e., all gaps in that region are entered in the CDS). This allows Minesweeper to avoid performing index seeks in T, since it knows exactly which tuples are of interest, and it simply needs to seek in S to determine if they are present. If the A_2 values from S cover only a portion of the already explored space in T, only some index seeks will be omitted, and in the worst case, where none of the A_2 values have been seen before, Minesweeper will have to perform the same index seeks as ZigZag merge join.

Note that Minesweeper's constraint memory can never save any index seeks on S. All of the constraints that Minesweeper learns over S are conditioned on particular A_1 values, and since the iterator on S is never rewound, each A_1 value is seen a maximum of one time.

Now consider a four-relation join, $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2, A_3) \bowtie U(A_3)$, with three join attributes, $\{A_1, A_2, A_3\}$. In this case, the above logic still applies—a portion of $S \bowtie T$ will be processed once per A_1 value, and constraint memory will save seeks on T—but this join introduces another level of rewinds; $T \bowtie U$ will be processed once per A_2 value found in $R \bowtie S$. When initially evaluating $T \bowtie U$ for a given A_2 value, Minesweeper's performance will be dependent on the size of the certificate.

However, after being processed once, Minesweeper will have recorded constraints that encode the exact output of $T \bowtie U$ for a given A_2 value. This means that the performance of subsequent evaluations will not be dependent on the size of the certificate, but instead on the size of the *output*. If the certificate for $T \bowtie U$ given a particular A_2 value is large, but the output is small, ZigZag merge join can end up wasting a great deal of effort compared to Minesweeper, since ZigZag merge join will re-evaluate the entire join on each rewind.

When combining multi-way operation and constraint memory is important. In some cases, the combination of multi-way operation and constraint memory can dramatically improve performance. Consider the four-relation join described above, $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2, A_3) \bowtie U(A_3)$. Suppose R contains $[1, \ldots, N]$, S contains $[1, \ldots, N] \times [1, \ldots, N]$, T contains $\{(2, 2), (2, 4)\}$, and U contains $\{1, 3\}$.

Minesweeper will conduct probes in T and U on each iteration, and it will quickly discover that both $\sigma_{A_2=2}(T) \bowtie U$ and $\sigma_{A_2 \neq 2}(T)$ are empty. By combining these two inferences, Minesweeper can conclude that the output of the join must be empty, and it can reach this conclusion *independent of any processing on R or S*. ZigZag merge join, since it cannot detect this fact, will repeatedly execute $R \bowtie S \bowtie T$, performing arbitrarily more work than Minesweeper.

This result is dependent on the global attribute order. One might complain that ZigZag merge join is unnecessarily hobbled in this case, since if it executed $T \bowtie U$ first, it would be able to

detect that join as empty much sooner. This is correct, but recall that the global attribute order is determined by which indexes have been created. If the only indexes that exist have a global attribute order such that $T \bowtie U$ must be executed last, ZigZag merge join has no choice but to do so.

3.6 Experimental results

Minesweeper has substantial theoretical performance advantages over ZigZag merge join, but these advantages are based on a worst-case runtime analysis. It is not clear whether these advantages materialize outside of specific special cases, and consequently, what their real-world impact might be. To address this, we performed two sets of experiments that explore a wide range of queries and data.

The first set of experiments is intended to be representative of these algorithms' performance on general, common cases. In this set of experiments, we evaluate performance over two general relational benchmarks: Star Schema Benchmark and TPC-H. Each of these benchmarks represents a different class of queries, and therefore, each benchmark has varying numbers of join relations and attributes that stress different aspects of each join algorithm.

The second set of experiments is meant to highlight advantages provided by specific features of Minesweeper. For these experiments, we use hand-crafted data and queries that emphasize various parts of Minesweeper's functionality. Our intent with these results is to show that, if data and queries fit a certain format, Minesweeper is indeed the clear choice.

We begin by describing the setup used for all of our experiments in this chapter, and then we describe each of our experiments and results in turn.

3.6.1 Experimental setup

We implemented ZigZag merge join and Minesweeper in the same stand-alone C++ application we used in Section 2.5. This application was designed to be "database-like," including files of slotted pages, a buffer pool that uses the Clock replacement algorithm, and a Volcano-style iterator interface

with a tree of binary operators (in ZigZag merge join's case) or a single multi-way operator (in Minesweeper's case). All benchmark binaries were generated using Visual Studio 2013 in 64-bit Release mode with full optimizations and link-time code generation enabled.

We executed our experiments on a single server running Microsoft Windows Server 2012. The server was equipped with 2 quad-core, 2.16 GHz Intel Xeon processors and 32 GB of RAM. For the majority of experiments, we used 25 10,000 RPM 146 GB SAS hard disks, with one disk storing the OS and the remaining 24 disks, combined into an array with RAID 0, storing the database. When experimenting with solid state disks, we used 12 256 GB Crucial C300 RealSSD solid state disks, combined into an array with RAID 0, to store the database. We allocated 24 GB of memory to the buffer pool unless otherwise noted.

In general, our results show the performance of queries executed in isolation with a warm cache. Each query was run once to warm the cache, and the results of that run were discarded. The query was then run five more times with the same parameters, and the mean execution time of these runs is reported as the execution time of the query.

For results with a cold cache, we also execute each query five times. However, in between each execution, we clear both the buffer pool and the operating system file cache. As before, we report the mean execution time of these five runs as the execution time for the query.

We generated data for our benchmarks using standard tools. For TPC-H, we used the official dbgen utility from TPC [92]. For SSB, we used PDGF, a parallel data generation framework [80]. Data for both TPC-H and SSB were uniformly distributed.

We assume that appropriate indexes have already been created on the data prior to execution, and so our execution time results do not include time spent to sort relations and construct the indexes.

3.6.2 Star Schema Benchmark and TPC-H

In order to investigate performance on general relational queries, we utilize two standard database benchmarks: Star Schema Benchmark and TPC-H. These benchmarks contain a suite of queries that illustrate different types of common joins with a variety of predicates, and as such, we feel they

represent a good proxy for "average case" queries.

To provide baseline performance for standard join algorithms, we also implemented hybrid hash join and index nested loops join. Since our C++ testbed application does not have a query optimizer, for queries with multiple joins, we determined the join order based on cardinality assuming perfect cardinality statistics. All join algorithms use the same join order, and we do not investigate using multiple different join algorithms in a single plan (e.g., executing one join with index nested loops and a later join with hash join). Finally, for hash join, we implement hybrid hash join, and we always use the smaller input relation as the build side and the larger input relation as the probe side.

In addition, for some results, we show data for ZigZag merge join with no heuristic (denoted ZigZag-NH or simply NH). When executing a seek, our implementation of ZigZag merge join will first search the current page of the relation prior to accessing the index. This reduces the number of index seeks that occur for small gaps, and consequently allows ZigZag to execute faster in some cases. ZigZag-NH does not use this heuristic, and instead accesses the index for every seek; it functions identically to ZigZag in all other respects.

Finally, we created appropriate indexes for each query, potentially using multiple indexes per relation. This allows us to ensure that all join algorithms we test use the same join order, but it also creates a favorable scenario for skipping join algorithms.

Star Schema Benchmark

Star Schema Benchmark (SSB) tests the performance of the various join algorithms when evaluating a star join. Star joins consist of a single large relation, called the fact table, that has many join attributes; this relation joins with a set of smaller relations, called dimension tables, that each have a single join attribute. As a result, we expect that the most important factor for performance in this benchmark is the ability to skip over tuples in the fact table, since it is by far the largest relation in the query.

Although there are a great deal of join algorithms specialized for star joins, our intent in this benchmark is not to claim that skipping join algorithms are ideal for this case. Instead, our goal is

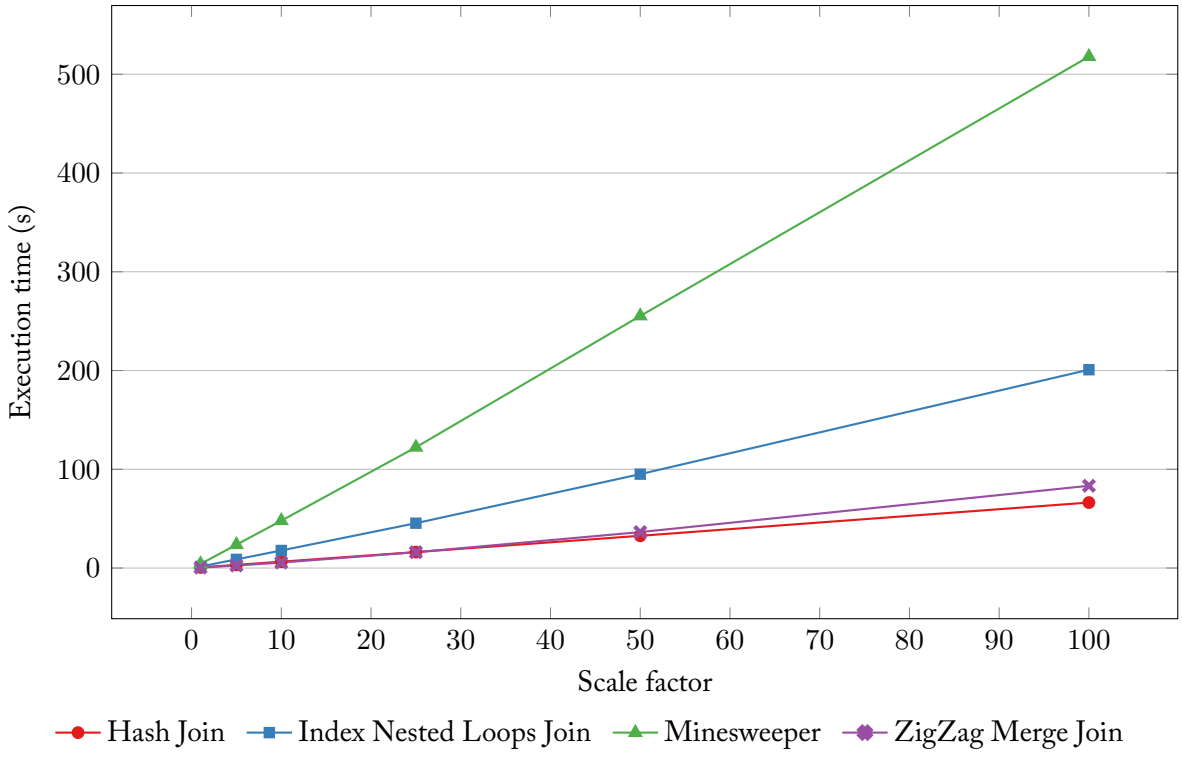


Figure 3.2: Warm-cache Star Schema Benchmark performance over various scale factors.

to show their performance on a representative class of queries as a baseline indicator, and as a result, we do not compare with join algorithms that work only for star join. However, we do note that our hash join implementation builds hash tables on all of the dimension tables and streams tuples from the fact table through all hash tables in succession, which is similar to the plan used by Microsoft SQL Server to execute star joins [36].

Warm-cache queries. We show mean warm-cache execution time across all queries at various scale factors in Figure 3.2. On average, ZigZag merge join and hash join provide the best performance, with hash join outperforming ZigZag merge join by a small margin at larger scale factors. The performance of index nested loop join suffers since the initial join with the fact table produces many tuples, all of which cause index seeks in a later dimension table. Finally, Minesweeper actually provides the worst performance of all join algorithms tested, and the difference becomes more pronounced as the data volume increases. In order to explain these differences, we dive deeper into performance at scale factor 100, which has roughly 55 GB of raw data (see Figure 3.3).

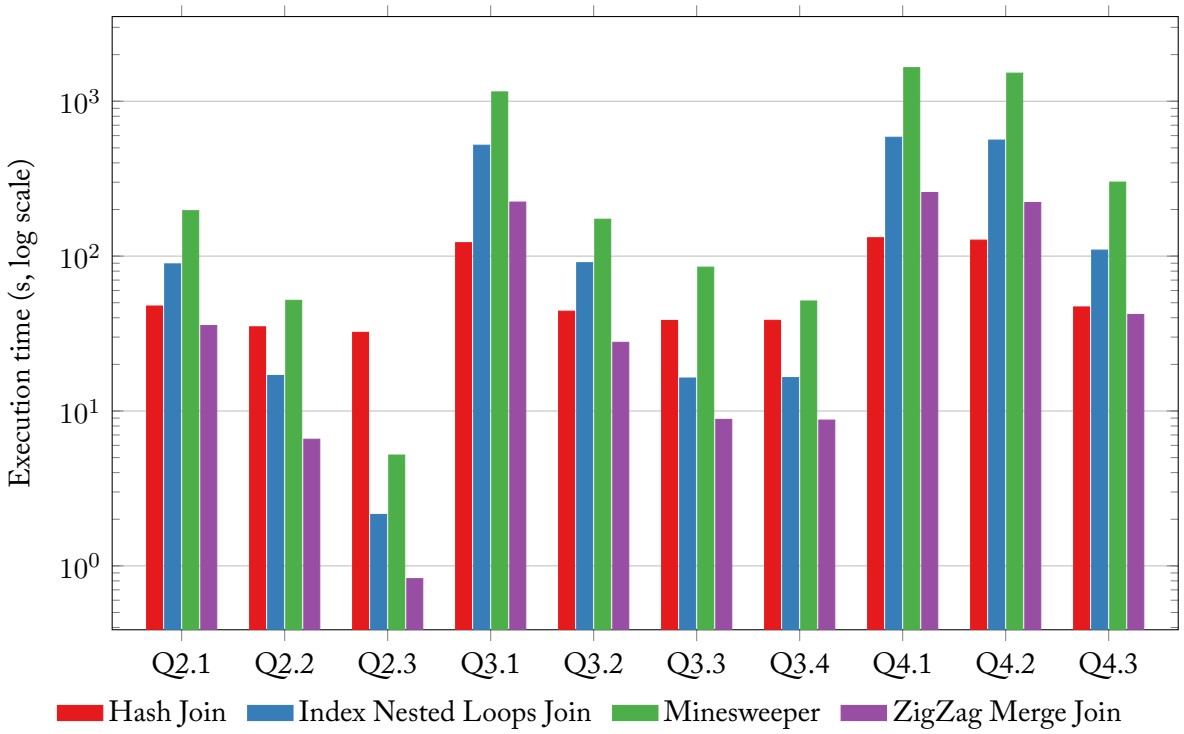


Figure 3.3: Warm-cache Star Schema Benchmark performance at scale factor 100.

At scale factor 100, ZigZag merge join outperforms hash join on a number of queries, sometimes by an order of magnitude or more, but provides worse performance on others. This is because ZigZag merge join processes many fewer tuples than hash join does (see Table 3.3), but at a higher per-tuple cost; in the queries where ZigZag merge join has the largest performance advantage, Q2.3 and Q3.3, it processes 99.8% and 99% fewer tuples, respectively. Minesweeper is also able to process fewer tuples than hash join in many cases, but it always processes more tuples than both ZigZag merge join and index nested loops join.

The reason for this discrepancy is because Minesweeper seeks in every relation on each iteration, even if the values for the probe point have not changed since the last iteration. While this may seem strange, recall that probe points in Minesweeper can be generated arbitrarily—there is no guarantee that they explore the space in any particular order. Accordingly, Minesweeper is not designed to take advantage of exploring the space in a systematic way, and since it remembers only gaps and not found values, it may read the same tuple multiple times.

Furthermore, Minesweeper seeks in all relations simultaneously. While this is a good way to

	Tuples read (k)			
	ZigZag merge	Minesweeper	Hybrid hash	Index nested loops
Q2.1	38,523.89	134,168.48	600,094.55	62,372.24
Q2.2	7720.09	34,251.12	601,438.63	12,516.77
Q2.3	956.11	3604.78	600,039.77	1555.19
Q3.1	233,869.75	781,095.31	600,638.30	357,529.76
Q3.2	19,071.01	105,017.12	600,126.81	53,151.24
Q3.3	5527.25	43,512.11	603,198.61	19,901.66
Q3.4	5189.54	25,526.20	603,196.08	19,788.30
Q4.1	181,920.10	943,692.21	602,039.27	366,169.99
Q4.2	117,688.80	878,792.58	602,039.28	364,747.37
Q4.3	15,154.03	170,074.45	600,663.21	58,941.25

Table 3.3: Number of base data tuples read to answer SSB queries at scale factor 100.

	Index seeks (k)			
	ZigZag merge	ZigZag merge (NH)	Minesweeper	Index nested loops
Q2.1	5075.53	12,032.07	56,915.88	28,784.80
Q2.2	980.09	2416.87	14,985.84	5777.64
Q2.3	125.06	301.47	1552.18	718.25
Q3.1	30,164.01	66,463.71	358,309.63	168,096.36
Q3.2	3370.00	12,298.86	58,432.27	26,144.40
Q3.3	1298.73	3833.57	25,021.30	9933.30
Q3.4	1291.03	3824.11	13,691.77	9893.88
Q4.1	36,820.59	74,657.40	479,940.01	178,266.59
Q4.2	32,738.90	74,477.93	475,232.01	181,127.14
Q4.3	6368.14	13,389.33	88,627.06	29,445.34

Table 3.4: Number of index seeks needed to answer SSB queries at scale factor 100.

rapidly learn about more of the join space, if relations that are earlier in the global attribute order are effective at pruning large portions of the space, seeks in later relations may not prove useful in the end. In this scenario, ZigZag merge join and index nested loops join may accomplish the same effective pruning simply by examining relations in a pair-wise manner.

To illustrate this, we show the number of index seeks required to answer queries at scale factor 100 in Table 3.4. We include index nested loops join as a baseline figure, and we also include ZigZag

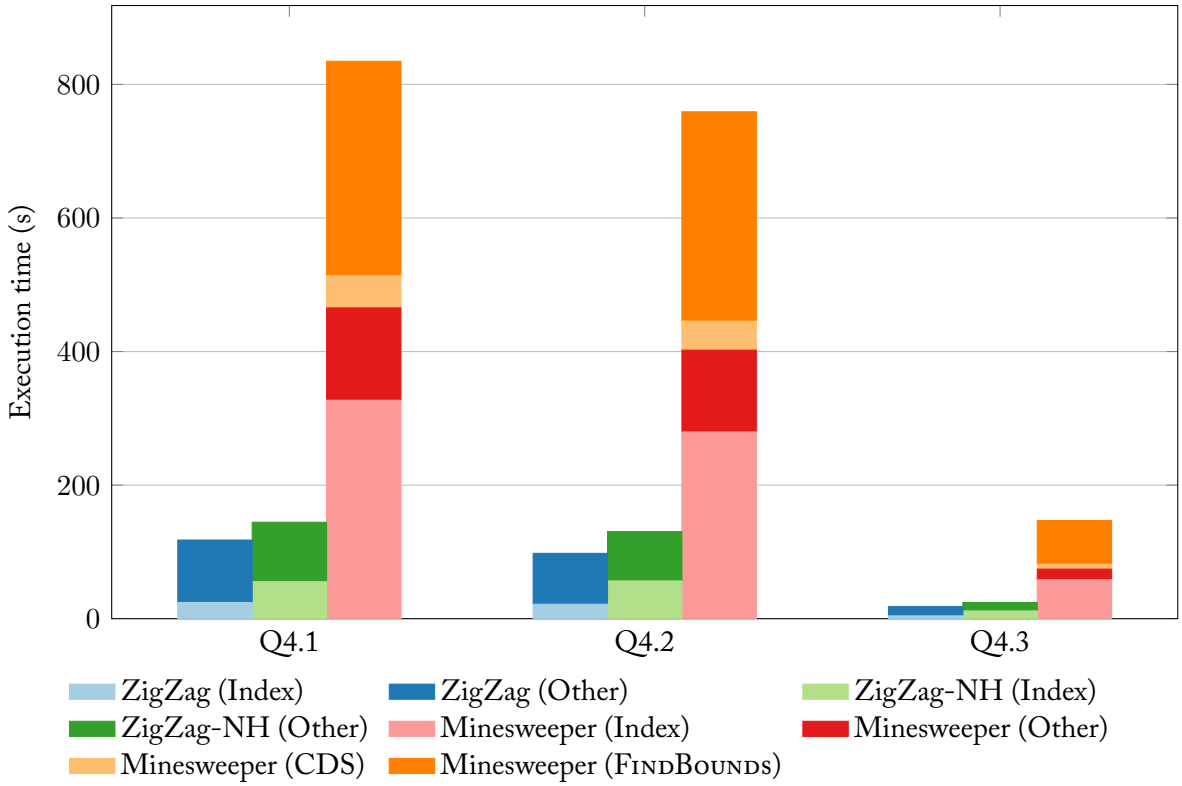


Figure 3.4: Skipping join algorithm time breakdown for SSB query flight 4 at scale factor 50 with a warm cache.

merge join with its index seek heuristic disabled (ZigZag NH). Here, we can see that Minesweeper performs 10–20x more seeks than ZigZag merge join and 2–3x more seeks than index nested loops join.

Additional index seeks are not the only factor that affects Minesweeper's performance, as Minesweeper also spends time learning and storing constraints. We show a time breakdown for Minesweeper and ZigZag merge join in Figure 3.4. For brevity, we show only queries from SSB flight 4—results for other flights are similar. We denote time spent executing index seeks as *index* and time spent checking if tuples are matches, copying tuples to output buffers, etc., as *other*. For Minesweeper, we denote time spent inserting constraints into the CDS as *CDS* and time spent generating new seeks and intermediate constraints based on probes as *FindBounds*.

At first glance, it may seem that Minesweeper spends an inordinate amount of time executing FindBounds, and that perhaps our implementation of FindBounds is poor. However, each

execution of FindBounds is reasonably fast. On query 4.1, scale factor 50, FindBounds executed 126,868,632 times with an average execution time of $2.50\,\mu s$. Similarly, on average, each index seek took $1.40\,\mu s$, and each call to the CDS took $0.56\,\mu s$.

Supposing $seek\_time$ is the time to execute one index seek and $proc\_time$ is the time to process (i.e., compare, copy, etc.) a tuple, we can roughly describe ZigZag merge join's performance with the following:

$$(\#\text{ index seeks} \times seek\_time) + (\#\text{ tuples processed} \times proc\_time)$$

Similarly, supposing $bounds\_time$ is the time for one call to FindBounds and cds_time is the time to insert a constraint into the CDS, Minesweeper's performance can be described as:

$$(\#\text{ index seeks} \times seek\_time) + (\#\text{ tuples processed} \times proc\_time) +$$
$$(\#\text{ FindBounds calls} \times bounds\_time) + (\#\text{ CDS calls} \times cds\_time)$$

Our implementation could always be improved, but some costs are shared—any improvement in index seek or tuple processing time will benefit both algorithms. Consequently, for Minesweeper to provide better performance, it must perform fewer seeks and read fewer tuples than ZigZag merge join does, since it always has some overhead for calls to FindBounds and the CDS. For example, consider query 4.1.

When executing query 4.1 at scale factor 100, Minesweeper examines 41,991,813 probe points. Suppose we have an ideal version of Minesweeper that executes only a single index seek per probe, but learns equivalent information to our current implementation that executes multiple seeks per probe; we call this idealized implementation *Minesweeper-I*. Reducing the number of index seeks will also reduce the number of data tuples read, and for a baseline estimate, we use two data tuples per seek.[1] Accordingly, Minesweeper-I will execute 41,991,813 index seeks and examine 83,983,626 tuples; without the gallop search heuristic, ZigZag merge join executes 74,657,403 seeks and examines

[1] This is the ratio that our implementation of Minesweeper shows on query 4.1.

181,920,102 tuples.

From our empirical results, $seek\_time$ is 1.40 µs and $proc\_time$ is 0.90 µs. Using these times, Minesweeper-I will spend 134.37 s seeking and processing tuples, while ZigZag merge join will spend 268.25 s, giving Minesweeper-I a buffer of 133.87 s it can use for calls to FindBounds and the CDS. Each probe must generate at least one FindBounds call and one constraint inserted into the CDS, and with our earlier figures, this requires 104.98 s and 23.52 s, respectively, for a total of 128.50 s and an overall execution time of 262.87 s. Therefore, in this idealized case, Minesweeper-I would be slightly faster than ZigZag merge join.

This analysis assumes ZigZag merge join is not using the heuristic of gallop searching in the current page before accessing the index. With the gallop search heuristic, ZigZag merge join converts 37,836,816 index seeks into gallop searches at a cost of 1.07 µs per search, effectively reducing its average index seek cost to 1.23 µs. This reduces ZigZag merge join's overall execution time to 255.82 s. If we assume that the gallop search heuristic could be applied to Minesweeper-I and achieves similar improvements, Minesweeper-I's total execution time would be reduced to 254.47 s, making ZigZag merge join and Minesweeper roughly equal.

Minesweeper-I receives less benefit from the gallop search heuristic since our idealized version performs fewer seeks than ZigZag merge join. As a result, even with the gallop search heuristic, Minesweeper-I would need to reduce time spent in calls to FindBounds and the CDS to offer any substantial performance improvement over ZigZag merge join on query 4.1. This suggests that, at least for Star Schema Benchmark data, Minesweeper would outperform ZigZag merge join only with an extremely optimized implementation that both dramatically reduced the number of index seeks as well as improved the performance of FindBounds and the CDS.

Finally, a similar argument can be made when comparing ZigZag merge join with hybrid hash join. For SSB queries, ZigZag merge join takes 1.30 µs per tuple on average (including all seeks and other processing time), while hybrid hash join takes 0.18 µs, roughly 7x faster. ZigZag merge join makes up for its slower per-tuple processing time by accessing many fewer tuples than hash join; if it can read at least 7x fewer tuples than hybrid hash join, it can be more efficient. Of course,

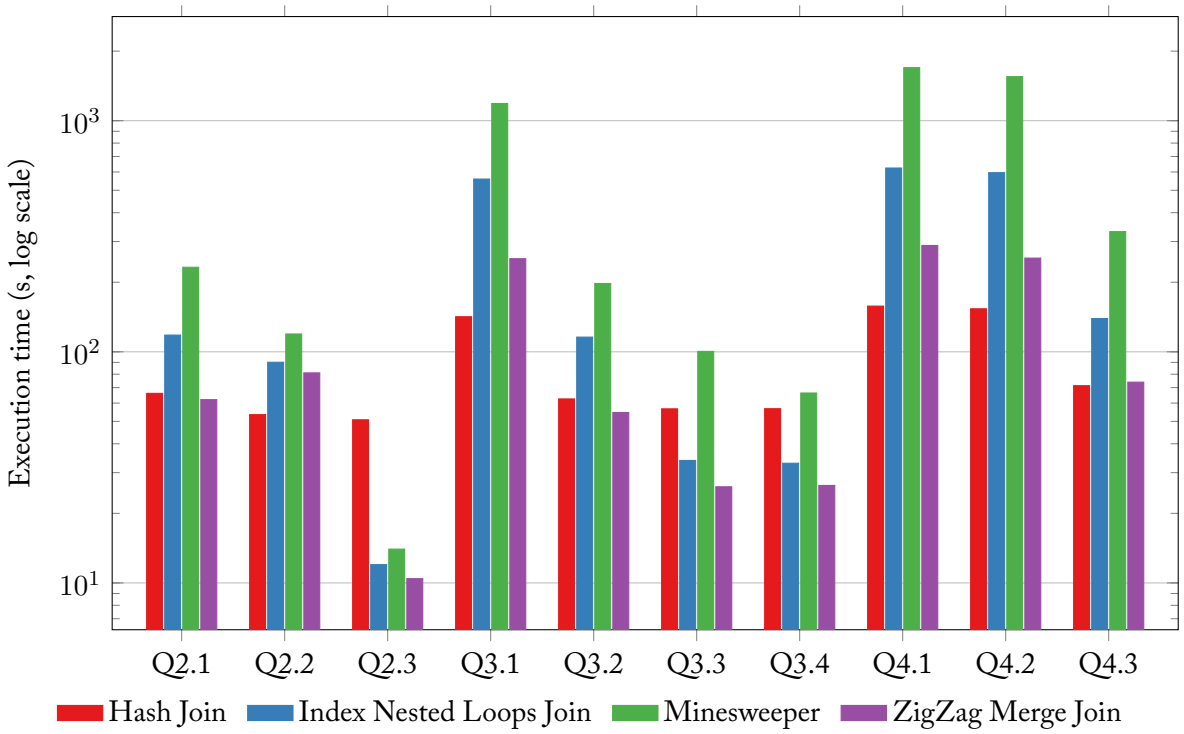


Figure 3.5: Cold-cache Star Schema Benchmark performance at scale factor 100.

given different hardware, ZigZag merge join's per-tuple processing time may increase, and this will change the amount of skipping required to make ZigZag merge join attractive.

Cold-cache queries. Hash join, ZigZag merge join, and Minesweeper all have different I/O access patterns. Hash join performs primarily sequential I/O, while the skipping join algorithms may randomly jump among relations. Accordingly, skipping join algorithms will win in this case only if they skip enough data to outweigh the cost of random I/O.

We present cold-cache query results at scale factor 100 in Figure 3.5. When the cache is cold, the effective cost of a seek increases, since it incurs a random I/O. This changes the amount of data that must be skipped in order to make ZigZag merge attractive when compared to hybrid hash join.

As a result, for queries where ZigZag merge join is able to read only a small portion of the data—like queries 2.3, 3.3, and 3.4—it remains the best choice even when the cache is cold. On other queries, where the reduction in data read by ZigZag merge join is less dramatic, the increased cost of seeks outweighs the savings from skipping. Minesweeper remains the worst of the three skipping join algorithms, since it always performs the most index seeks and consequently suffers the

largest I/O penalty.

However, for some queries, the gap between Minesweeper and hash join is smaller than when the cache is warm. This is because there are several cases (see Table 3.3) where Minesweeper reads many fewer tuples than hash join. When I/O is a factor, reading fewer tuples saves more time, and consequently the extra time spent in FindBounds and the CDS is more worthwhile.

TPC-H

TPC-H uses a normalized schema with more relations than SSB, which means that its queries involve many more joins than SSB's. Additionally, each relation in a query typically has at most two join attributes, and as a result, TPC-H provides us with insight into how the various algorithms perform on long sequences of joins. However, we are interested only in the performance of join algorithms, and as a result, some TPC-H queries are either not applicable (e.g., query 1 has no joins) or they have substantial non-join components (such as aggregates).

Accordingly, we use modified versions of a subset of TPC-H queries (see Appendix C for a full description). These modifications pare the queries down to only join operations with simple predicates, maximizing the differences between join algorithms. Therefore, our results in this section should not be considered as full runs of the TPC-H benchmark, but rather a baseline for performance on a set of representative joins from a common class of queries.

Warm-cache queries. We show mean warm-cache execution time across all queries at various scale factors in Figure 3.6. In this case, ZigZag merge join provides the best performance of the join algorithms we tested, while Minesweeper performs the worst. Compared to Star Schema Benchmark queries, TPC-H queries are more complex, and this complexity has a large impact on Minesweeper's execution time.

More specifically, Minesweeper is impacted by the number of relations in the query as well as the number of join attributes in those relations. Because Minesweeper is constantly probing all relations, the number of seeks it issues is affected by how many relations are in the query. In Star Schema Benchmark, there are relatively few relations—at most four—while our version of TPC-H has

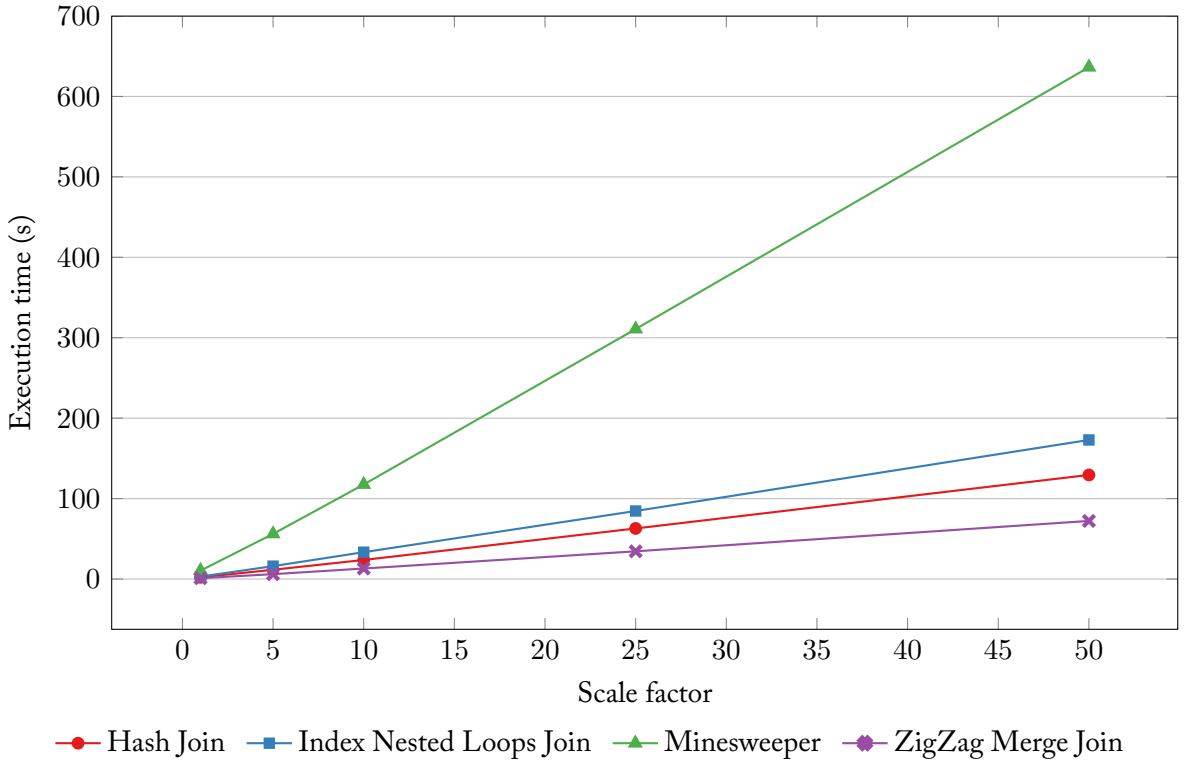


Figure 3.6: Warm-cache TPC-H query performance at various scale factors.

potentially eight relations in a single query. Furthermore, as the number of join attributes increases, the size of constraints increases, which affects the amount of time required by FindBounds and the CDS. These issues combine to result in Minesweeper's poor performance on TPC-H queries.

As before, we further illustrate these issues by diving into our results at scale factor 50, which has roughly 110 GB of raw data (see Figure 3.7). Minesweeper offers the worst performance of all tested join algorithms for all queries except Q2, Q11, Q14, and Q16. Q11, Q14, and Q16 are some of the simplest queries in our benchmark, with only 2–3 relations and a single equality predicate, and Q2 also features relatively few relations with selective predicates. Consequently, Minesweeper is able to rapidly eliminate large portions of the join key space on these queries; however, it is not able to do so in other cases.

We show a time breakdown for three of Minesweeper's worst-performing queries in Figure 3.8; other queries are similar. As in the SSB case, Minesweeper spends a substantial amount of time performing index seeks and generating intermediate constraints. On each iteration, Minesweeper

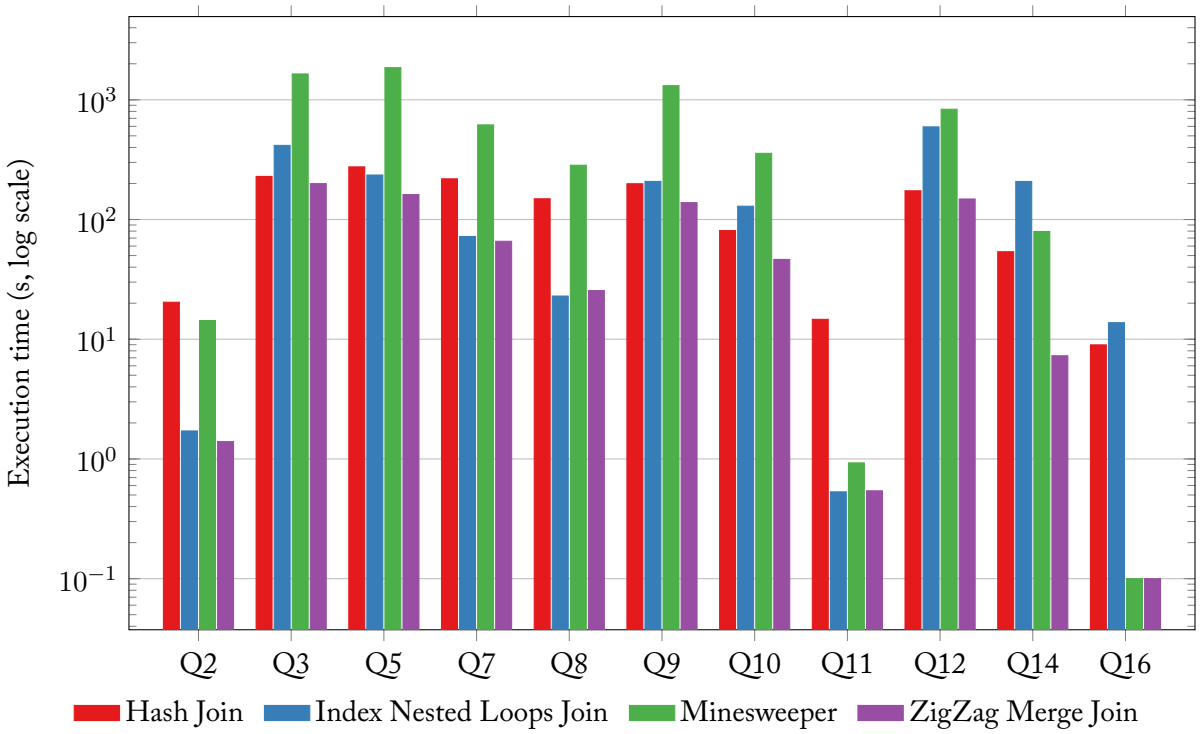


Figure 3.7: Warm-cache TPC-H query performance at scale factor 50.

calls FindBounds once per relation, and each call to FindBounds executes up to $2^k - 1$ index seeks, where k is the number of join attributes in the relation. Compared to SSB queries, Minesweeper needed more iterations (22,164,602 vs. 7,421,093) and executed more index seeks (175,711,750 vs. 79,819,210) on average. Tables showing the number of index seeks and data tuples read for each query are available in Appendix C.

Cold-cache queries. We present results for cold-cache TPC-H queries at scale factor 50 in Figure 3.9. Here, both ZigZag merge join and Minesweeper are much slower than hash join for several queries. This occurs because, unlike in SSB, the largest relations in TPC-H—Orders and Lineitem—are rewound in the course of query processing.

These rewinds are required due to predicates on relations in the query. For example, in query 3, there is a range predicate on Orders that selects all tuples whose order date is prior to March 15, 1995. Consequently, the join must effectively be evaluated once for each order date prior to March 15, 1995, causing a large number of rewinds.

This manifests as a much larger per-tuple processing cost for ZigZag merge join versus hash join.

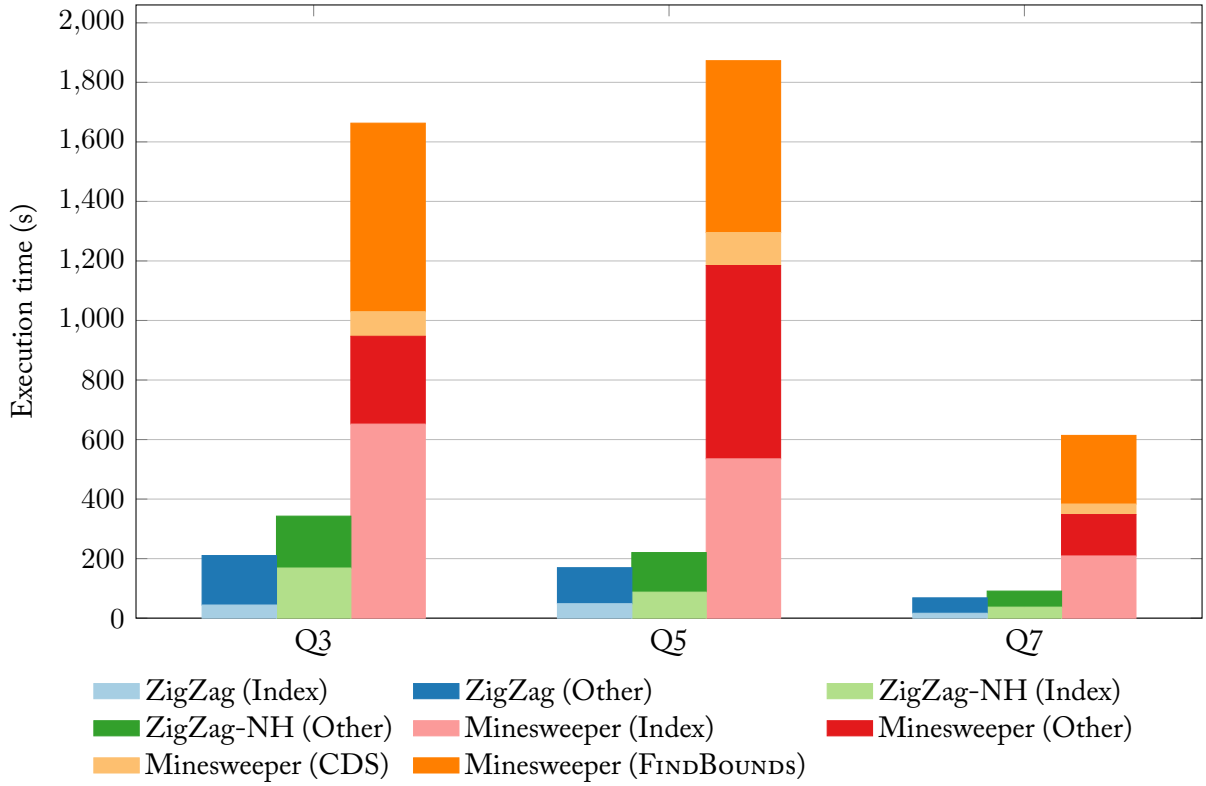


Figure 3.8: Skipping join algorithm time breakdown for selected TPC-H queries at scale factor 50 with a warm cache.

On average, for cold-cache queries, hash join processes individual tuples 25x faster than ZigZag merge join, and in some cases, hash join is nearly 80x faster per tuple. Accordingly, ZigZag merge join needs to skip a large number of tuples in order to outperform hash join, but on average, ZigZag merge join reads only 11x fewer tuples than hash join. This causes hash join to outperform ZigZag merge join for most cold-cache TPC-H queries.

Low-memory experiments. We also experimented with running a single query, query 3, at scale factor 100 under various amounts of available memory (see Figure 3.10a). When the cache is warm and the machine has at least 16 GB of RAM available, ZigZag merge join and hybrid hash join offer roughly equivalent performance on query 3. However, when the cache is cold, ZigZag merge join is an order of magnitude slower than hybrid hash join, and this performance difference is exacerbated as the amount of available memory decreases, eventually causing ZigZag merge join to be almost two orders of magnitude slower than hybrid hash join.

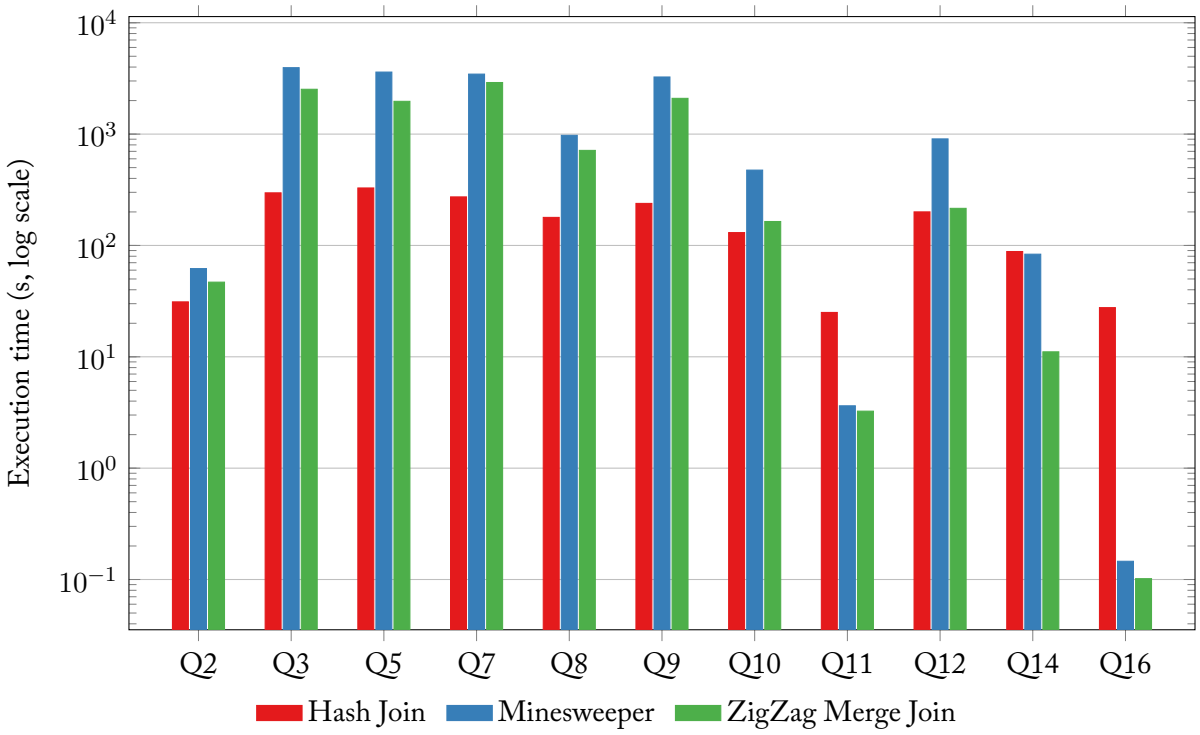
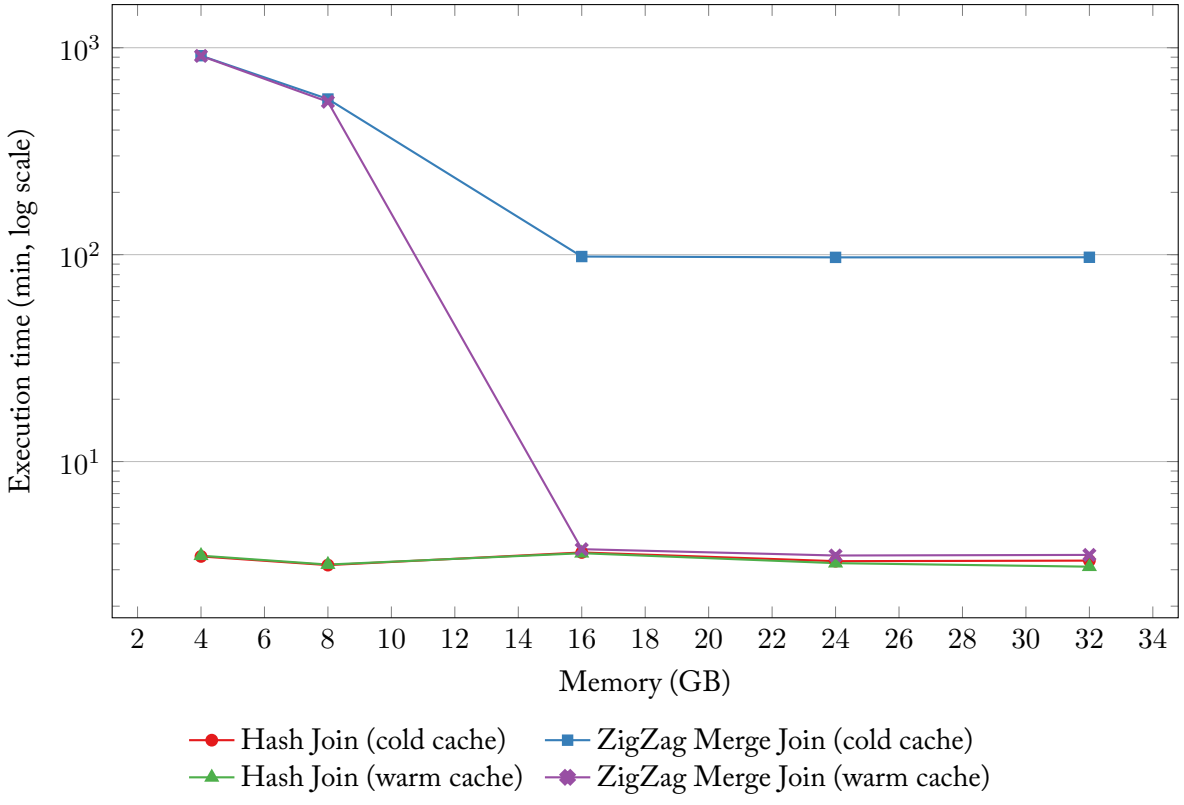


Figure 3.9: Cold-cache TPC-H query performance at scale factor 50.

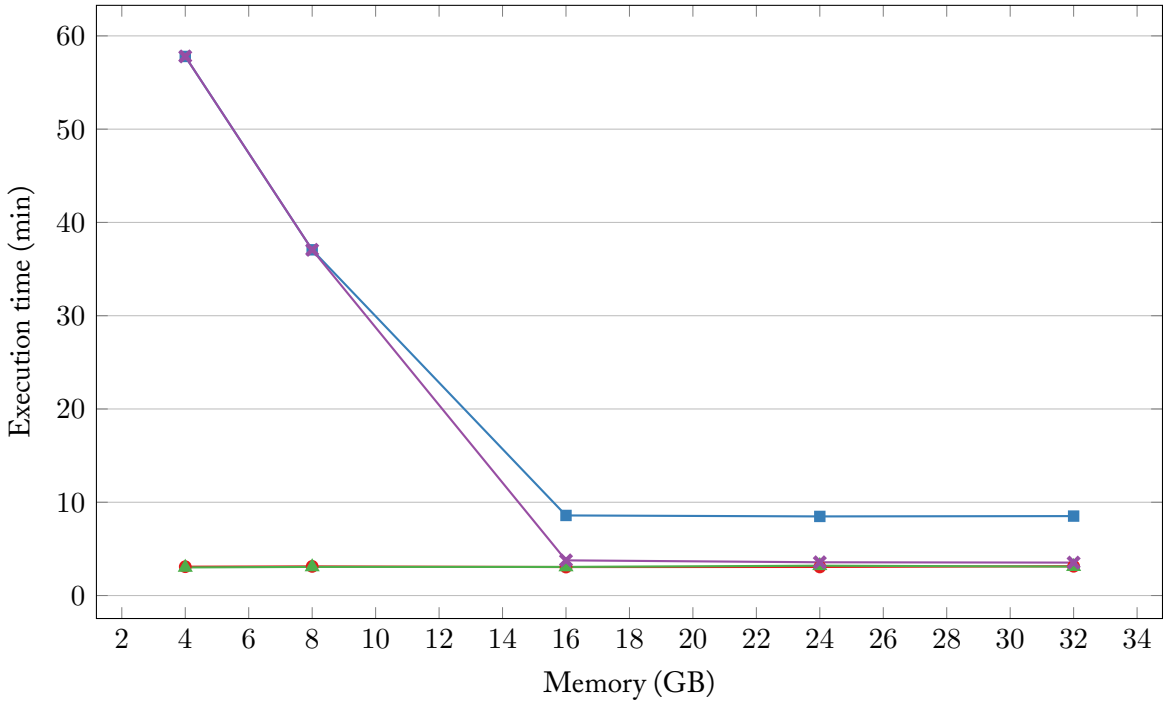
This performance degradation is due to ZigZag merge join's random seeks. When sufficient memory is available, smaller relations involved in the join are cached, causing their random seeks to be eliminated. As the amount of memory shrinks, more seeks access the disk instead of memory, and ZigZag merge join's warm-cache performance eventually equals its cold-cache performance.

Note that these results are dependent on hardware characteristics. The server we used for our experiments has a large number of fast hard disks, giving it a large amount of sequential read bandwidth but limited random read performance. This favors hash join in cold-cache experiments; however, if we used hardware with better random read performance, it is possible that ZigZag merge join would be more attractive.

To test this, we ran the low-memory experiment using solid state disks (SSDs) instead of traditional hard disks (see Figure 3.10b). Solid state disks have substantially better random read performance than traditional hard disks, and this reduces the penalty that ZigZag merge join pays when the cache is cold or memory is unavailable. Unfortunately, SSDs do not completely



(a) Hard drives.



(b) Solid state disks.

Figure 3.10: Effect of memory on TPC-H query 3 at scale factor 100.

eliminate the performance degradation for random I/O, and consequently, ZigZag merge join is still outperformed by hybrid hash join on this query unless the cache is warm.

Some of the random I/Os that ZigZag merge join incurs are due to rewinds on later relations. These rewinds can be removed by sorting intermediate results, allowing ZigZag merge join to perform skip-sequential I/Os instead of random I/Os for seeks. However, sorting intermediate results introduces a blocking operator, and it limits the ability of ZigZag merge join to skip, since the sorted result set will not have an index.[2] We do not explore the interaction of ZigZag merge join and sorting intermediate results in this work, but we feel it could be an interesting avenue for future investigation if cold-cache performance is a concern.

3.6.3 Specially crafted benchmarks

In Section 3.5, we mentioned that Minesweeper added two features over ZigZag merge join: multi-way operation and constraint memory. Although these features did not improve performance in generic database benchmarks like SSB and TPC-H, there are certainly queries and datasets where they could make a difference. To explore these scenarios, we conducted several special-purpose benchmarks using tailored data to show the impact of these features on performance in a controlled manner.

Multi-way operation

We discussed multi-way operation in Section 3.3 with a simple three-relation join, $R(A_1) \bowtie S(A_1) \bowtie T(A_1)$, populated as follows:

- R: $[1, 3, 5, \ldots, 2N + 1]$ (i.e., odd numbers)
- S: $[2, 4, 6, \ldots, 2N]$ (i.e., even numbers)
- T: $[2N + 2, 2N + 3, \ldots, 3N]$

[2] Some skipping is still possible, since binary or gallop search could be used without an index.

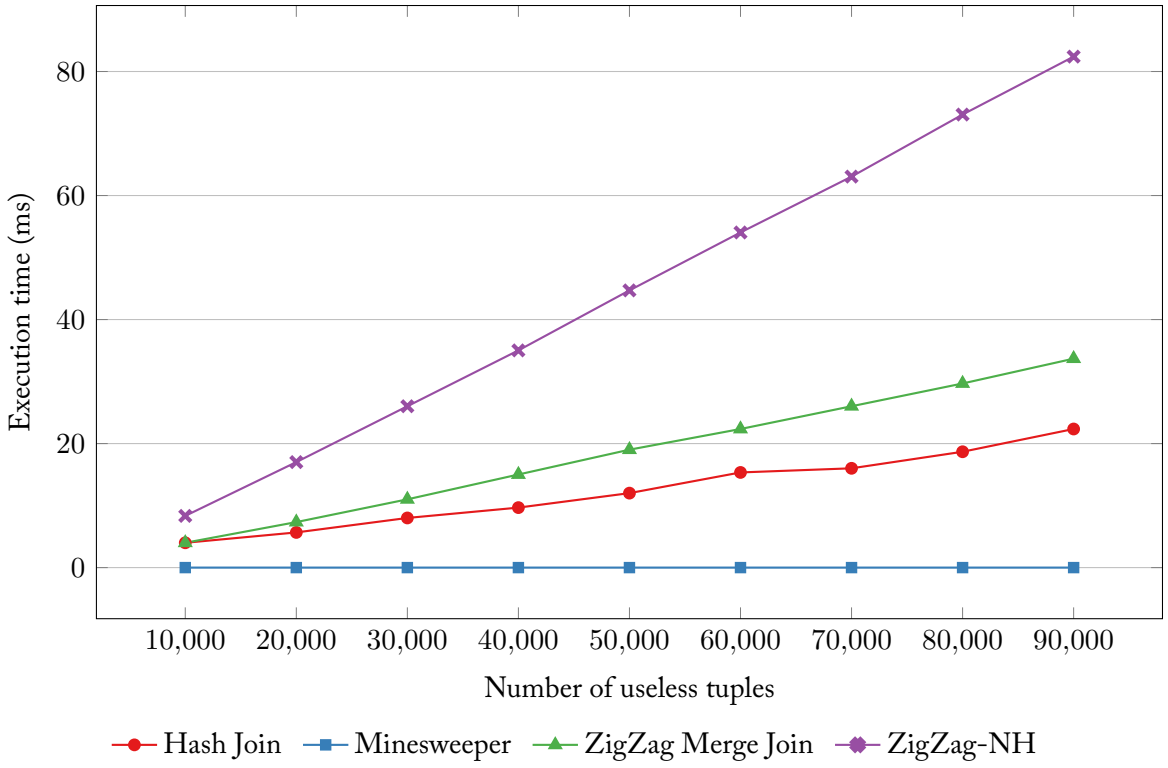


Figure 3.11: The effect of multiway operation with a warm cache.

The optimal certificate for this instance is $T[1] > S[N]$, but ZigZag merge join will compute the entire certificate of $R \bowtie S$ without ever attempting to join T.

We show this result, for varying values of N, in Figure 3.11. As expected, Minesweeper's execution time remains constant as N increases, since the amount of work it must do to determine that the join is empty is fixed. In contrast, all other algorithms, including ZigZag merge join and hash join, perform more and more work, making them arbitrarily slower.

First-time execution constraint caching

In principle, when performing a many-to-many join for the first time, Minesweeper may be able to use knowledge gained from previous sorted runs of join key values to reduce seeks when joining later runs, an effect we call *first-time execution constraint caching*. For example, consider the following query: $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2)$. For each sorted run of A_2 values Minesweeper processes, it explores some amount of T and stores its findings as constraints. As a result, when a previously

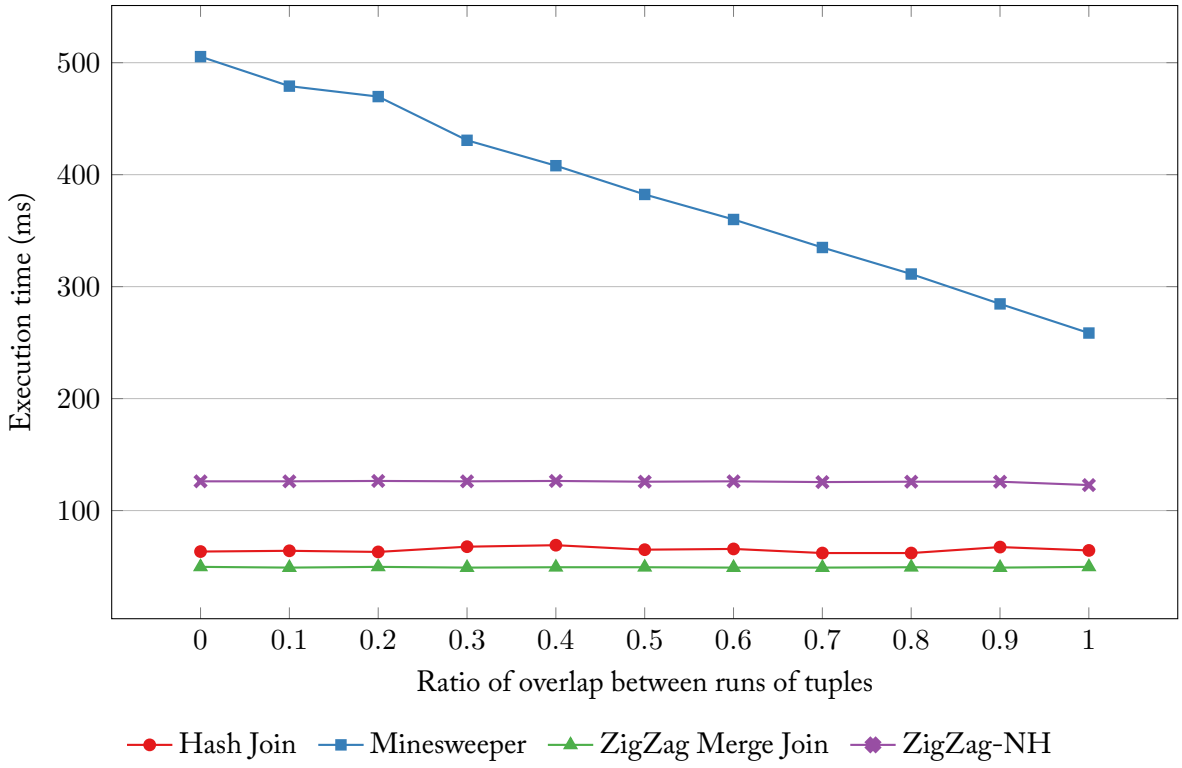


Figure 3.12: The effect of first-time execution constraint caching with a warm cache.

explored region of T is encountered for a second time (i.e., when processing a later run of A_2 values), Minesweeper can rely on the constraints it has already examined rather than performing new seeks.

More specifically, the effect of first-time execution constraint caching is affected by how much overlap there is between sorted runs of A_2 values. In the worst case, all sorted runs of A_2 values are disjoint—as a result, knowledge learned from joining previous runs is useless, since it explored a different portion of T. In the best case, all sorted runs of A_2 values are the same, in which case the relevant portion of T will be explored after joining the first run, and no subsequent run will perform unnecessary seeks in T.

To test the effect of first-time execution constraint caching, we populate the relations as follows:

- $R(A_1)$: $[1, \ldots, N]$
- $S(A_1, A_2)$: $\bigcup_{k=1}^{N}([k] \times [1 + (k-1)(I-J), \ldots, kI - (k-1)J])$
- $T(A_2)$: $[1, \ldots, NI]$

N determines the number of sorted runs of A_2 values that are present in S, I determines the number of values in each sorted run, and J determines the number of values that overlap from one run to the next.

We are not interested in manipulating the performance of $R \bowtie S$, as no constraints are useful when joining two sorted runs a single time. Accordingly, we load R with a single contiguous run of A_1 values, and every A_1 value in R joins with S. Finally, we vary the amount of overlap in A_2 runs from completely disjoint ($J = 0$) to completely overlapping ($J = I$) with $N = 100$ and $I = 1000$.

Our results are shown in Figure 3.12. Minesweeper performs one seek on T per unique value in $\pi_{A_2}(S)$; as the number of unique values decreases, Minesweeper performs fewer seeks. Therefore, its performance improves as runs overlap.

However, both ZigZag merge join and hash join outperform Minesweeper, even when runs are completely overlapping. This is because, although Minesweeper has effectively determined all relevant values in T after processing a single A_2 run, it is not aware of this fact.

All Minesweeper knows is that the relevant values in T have not been ruled out by any constraint so far—it does not remember that they are *present*. Consequently, Minesweeper seeks in every relation, *including* T, for each probe point it evaluates. Thus, Minesweeper actually saves no seeks versus ZigZag merge join.

Repeated execution constraint caching

When repeatedly performing a many-to-many join (as in the case of rewinds), Minesweeper encodes the exact output of the join in the CDS. This enables subsequent executions of the join to retrieve all output tuples directly by using index seeks, an effect we call *repeated execution constraint caching*. To test the impact of repeated execution constraint caching, we use the following query: $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2, A_3) \bowtie U(A_3)$. In this query, $T \bowtie U$ is executed repeatedly, and Minesweeper can use its constraint memory to more efficiently process later executions of the join. Accordingly, there are two factors that influence Minesweeper's performance advantage.

The first factor that affects the size of Minesweeper's advantage is the amount of work saved

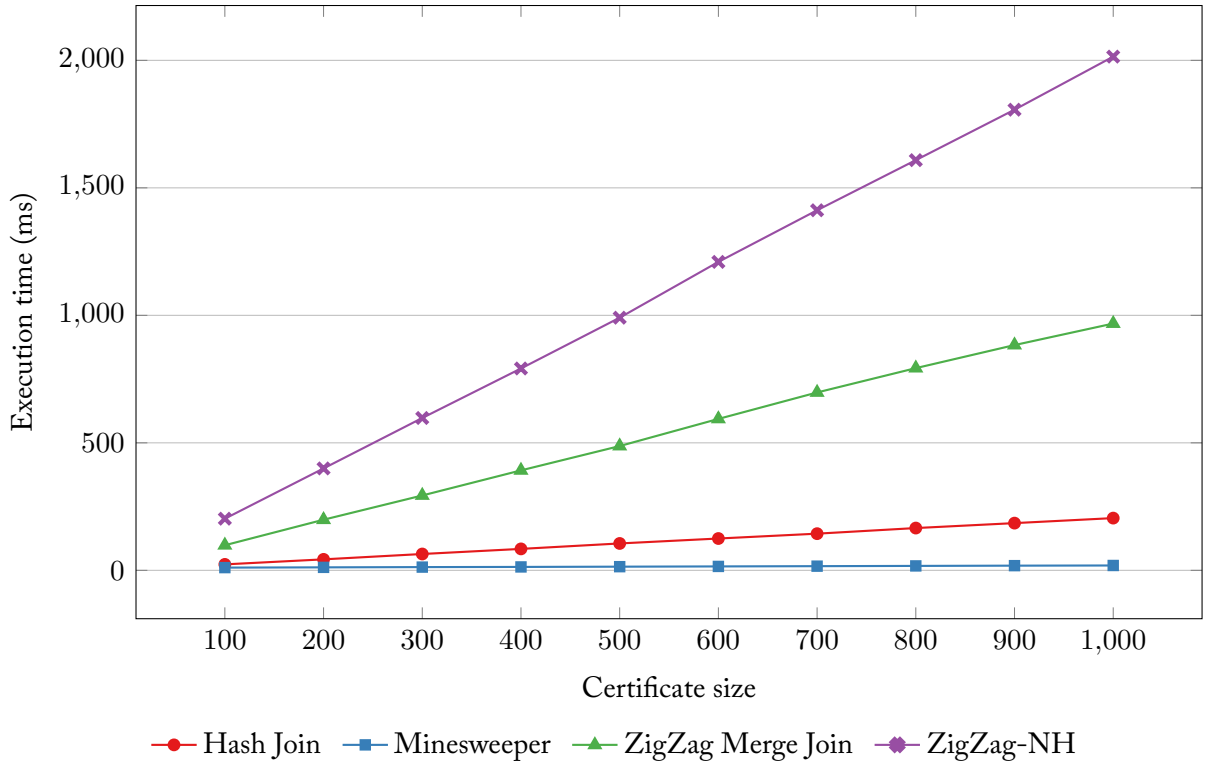


Figure 3.13: The effect of repeated execution constraint caching with a warm cache ($N = 1000$).

on a repeated execution of $T \bowtie U$ compared with ZigZag merge join. Minesweeper's execution of $T \bowtie U$ for a given A_2 value is sensitive to the size of the optimal certificate on the first iteration, but once all constraints are learned, Minesweeper's performance depends only on the size of the output. In contrast, ZigZag merge join performs a number of index seeks equal to the size of the certificate on all iterations. Consequently, by varying the size of the certificate for $T \bowtie U$, we can make ZigZag merge join "waste" more time relative to Minesweeper.

Minesweeper's advantage is also affected by the number of times $T \bowtie U$ is executed, and this is determined by the number of rewinds necessary. This query rewinds once for each A_1 value in R that joins with S. By increasing the number of A_1 values, we can increase Minesweeper's advantage; the total amount of work Minesweeper saves is $\sum_{x \in \pi_{A_2}(T)} |C(\sigma_{A_2=x}(T) \bowtie U)| \times |R \bowtie S|$.

We populate the relations as follows:

- $R(A_1)$: $[1, \ldots, N]$

- $S(A_1, A_2)$: $[1, \ldots, N] \times [1]$
- $T(A_2, A_3)$: $[1] \times ([1, 3, \ldots, 2M + 1] \cup [2M + 2])$
- $U(A_3)$: $[1] \times ([2, 4, \ldots, 2M] \cup [2M + 2])$

For simplicity, all values in R join with S, we use only a single A_2 value, and the output from $T \bowtie U$ is always exactly one tuple. Since A_2 has only one value, N controls the number of times $T \bowtie U$ is executed, and M determines the size of $C(T \bowtie U)$.

Our results with $N = 1000$ and varying values of M are shown in Figure 3.13. As the certificate size increases, Minesweeper achieves more and more savings. This query is particularly bad for ZigZag merge join, since it continually executes useless seeks on each iteration, but hash join, which never attempts to seek, is more resilient as the data volume increases.

Combined multi-way inference

Minesweeper is able to combine the information it learns from constraints across multiple tables to form inferences about the output of the join. This is most important when the output from one of the joins in a query is empty, because as soon as this fact is discovered, join execution is complete. To test this, we use the following query: $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2, A_3) \bowtie U(A_3)$.

We populate the relations as follows:

- $R(A_1)$: $[1, \ldots, N]$
- $S(A_1, A_2)$: $[1, \ldots, N] \times [1, \ldots, N]$
- $T(A_2, A_3)$: $\{(2, 2), (2, 4)\}$
- $U(A_3)$: $\{1, 3\}$

Note that $T \bowtie U$ is empty; Minesweeper determines this on its first evaluation of the join, but ZigZag merge join will execute $T \bowtie U$ N times. Therefore, by varying N, we can control how much wasted work is performed by ZigZag merge join.

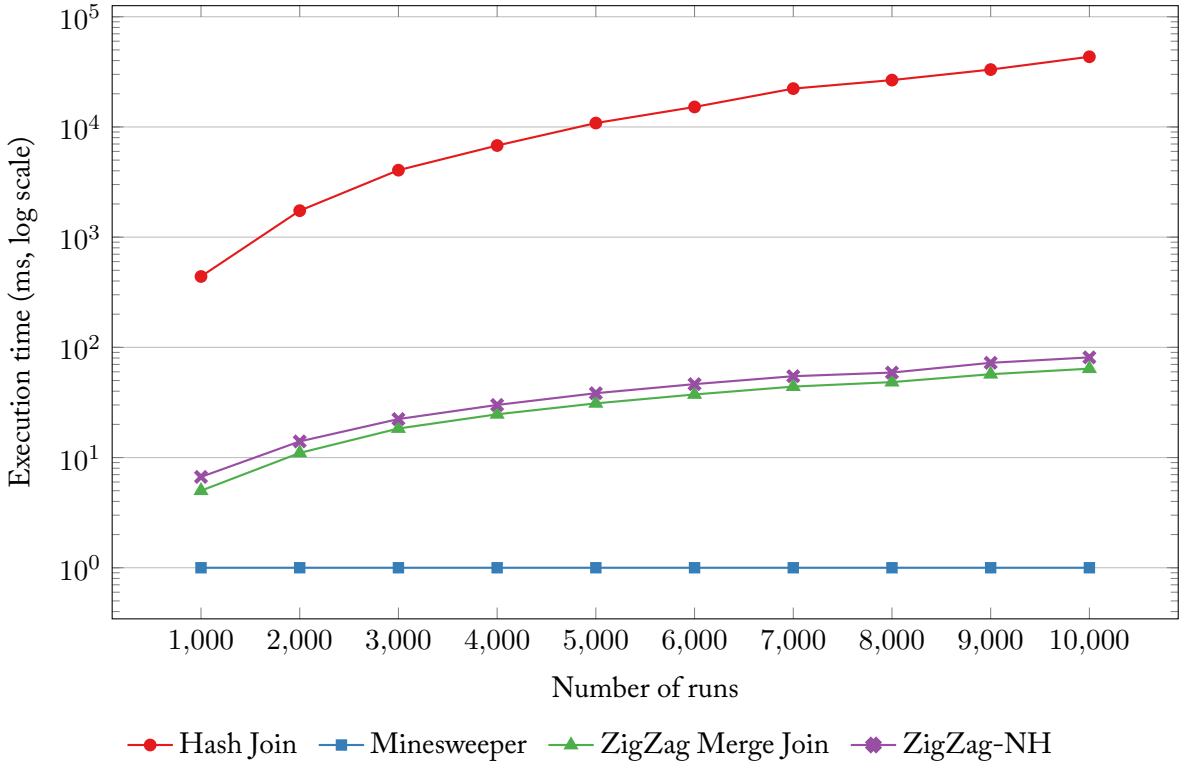


Figure 3.14: The effect of combined multi-way inference with a warm cache.

Our results are shown in Figure 3.14, and again, Minesweeper performs a constant amount of work while both ZigZag merge join and hash join increase linearly.

3.7 Conclusion

In this chapter, we extended ZigZag merge join to operate over general relational queries, and we analyzed its operation using a theoretical framework based on join certificates by Ngo et al. [73]. We showed that ZigZag merge join is optimal in terms of the certificate for a single join between two sorted lists, but it is not optimal for joins involving multiple relations or multiple sorted lists due to its binary nature. We then compared ZigZag merge join with a state-of-the-art theoretical skipping join algorithm, Minesweeper.

Minesweeper adds two main features over ZigZag merge join: multi-way operation and constraint memory. Multi-way operation enables Minesweeper to use information from later joins in a

query without having to find a match in earlier joins, and constraint memory allows Minesweeper to be more efficient when repeatedly executing the same join (as in the case of rewinds). These features give Minesweeper substantial advantages in the worst case, but may not be useful on average.

To evaluate this, we performed two general benchmarks using Star Schema Benchmark and TPC-H. When the cache is warm, our results showed that both ZigZag merge join and Minesweeper can outperform traditional, non-skipping join algorithms in certain cases, and ZigZag merge join provides the best performance overall, outperforming Minesweeper by an order of magnitude in some cases. This performance gap is due to the fact that Minesweeper performs more index seeks than ZigZag merge join; ZigZag merge join is aware of the position of the iterators on each relation, while Minesweeper inherently executes some index seeks that do not move the iterator. Additionally, although Minesweeper's constraint memory allows it to save some index seeks, the overhead of maintaining the constraints means this does not translate into a performance advantage.

When the cache is cold, I/O access patterns have more of an impact. This did not affect our results for SSB queries, since most tuples are read from the fact relation, and ZigZag merge join processes this relation in a mostly sequential manner. However, for TPC-H queries, ZigZag merge join and Minesweeper both rewind the largest two relations (Orders and Lineitem) a number of times, causing substantially more random seeks. This manifests as a much higher per-tuple processing cost than hash join (up to nearly 80x in some cases), and this higher cost means that ZigZag merge join and Minesweeper outperform hash join on cold-cache queries only when a very large number of tuples are skipped.

Furthermore, when the cache is cold and memory is limited, the amount of random I/O that ZigZag merge join performs increases. In this situation, hash join outperforms ZigZag merge join by almost two orders of magnitude on some TPC-H queries when using traditional hard disk drives. The cost of random I/O can be reduced by using solid state disks, but our results show that, while solid state disks improve ZigZag merge join's cold-cache performance by an order of magnitude, this is not sufficient to close the gap entirely.

In addition to general-purpose relational benchmarks, we also evaluated specific cases where

Minesweeper's unique features can substantially affect performance. In these cases, Minesweeper can be arbitrarily faster than ZigZag merge join and the traditional join algorithms we tested. Accordingly, for some situations, Minesweeper's advanced functionality is key for efficient processing.

Regardless of the specific differences between ZigZag merge join and Minesweeper, our results indicate that advanced skipping join algorithms have the potential to improve performance over the standard suite of join algorithms. Furthermore, although these algorithms can be arbitrarily faster in some special cases, the performance advantages these algorithms offer are in fact broadly applicable to a wide range of queries and data. In essence, this chapter makes that case that the standard join algorithms—index nested loops, traditional merge join, and hash join—are not sufficient, and we argue that more sophisticated skipping join algorithms are needed for performance.

These algorithms offer multiple new challenges that we have not discussed in this chapter. In particular, it is not immediately clear when to use ZigZag merge join or Minesweeper over a more traditional join algorithm; there are cases where hash join is the best choice, even when appropriate indexes exist. This decision is based on two factors: how much data can be skipped and how much work (i.e., how many index seeks and constraint updates) are required to skip that data. Our work in the next chapter is intended to provide a baseline method for estimating these factors prior to running a query.

4 | Statistics for estimating certificate sizes

4.1 Introduction

Chapters 2 and 3 have shown that skipping join algorithms can have a substantial impact on performance in a variety of situations. However, it is not always clear when these algorithms should be applied. Certainly, one approach is to always use a skipping join algorithm if the appropriate indexes exist, only falling back to traditional join algorithms when there is no other choice.

Unfortunately, as we saw in Chapter 3, skipping join algorithms do not always provide the best performance. As a result, always blindly using a skipping join algorithm will leave potential performance improvements on the table. However, it is not straightforward to determine what the performance of a skipping join algorithm will be prior to running it—RDBMSs often collect statistics about the data, but it is not clear whether common statistics, like histograms, are of any use.

The efficacy of skipping join algorithms is determined by two factors: how much data can be skipped, and how much time is required to skip that data. The first factor, how much data can be skipped, is closely related to the size of the join output—if the join produces very few tuples relative to the size of the input relations, then most tuples do not contribute to the output and can be skipped. However, the second factor, how much time is required, is the product of the number of seeks and the time per seek; if it is too large, the extra work required to perform the seeks may outweigh the savings from skipping data.

Standard techniques for join cardinality estimation [14, 65, 72] can be used to predict how much data can be skipped, and the average amount of time per seek can be estimated using, e.g., some

sort of offline calibration process or the optimizer's cost model. The number of seeks required, on the other hand, is proportional to the size of the join certificate, and to the best of our knowledge, no techniques exist for certificate size estimation.

More specifically, the size of the join certificate is related to the number of gaps where data exists in one relation but not in another—the more individual gaps there are, the more seeks will be required to exploit those gaps. Consequently, we need a method to determine where these gaps exist (i.e., the range of missing values). One obvious approach is to utilize statistics that RDBMSs already collect, but the standard histograms that RDBMSs maintain over their data are not helpful in this case.

Consider two common types of histograms: equi-depth and equi-width [77]. An equi-depth histogram will never store any missing values by definition (assuming the histogram has a nonzero depth), since all bins have the same number of values. Equi-width histograms, on the other hand, could detect missing values assuming a small enough bin size. In practice, however, RDBMSs use a small number of bins for a large number of rows; Microsoft SQL Server 2008, for example, uses a maximum of 200 bins even for relations with hundreds of millions of tuples [88]. As a result, it is very unlikely that a bin will be empty unless the overall join key value distribution is extremely skewed.

Accordingly, in this chapter, we introduce a new type of histogram called a *gap map*. Gap maps are equi-width histograms that store each bin as a single bit; instead of storing the cardinality of values in a bin, gap maps simply indicate the presence or absence of *any* value within the bin. By using standard bitmap compression techniques, gap maps can store many more bins than regular histograms, and assuming appropriate alignment across relations, they can be combined cheaply using bitwise operations. This enables them to track missing values accurately, and therefore they provide a method for estimating the size of a join certificate.

However, gap maps have several issues that prevent them from being a general-purpose solution. With a bin size of 1, gap maps are able to exactly determine the size of the optimal certificate for a two-relation join with one attribute and no predicates, but as the bin size increases, gap map

accuracy degrades sharply. Additionally, for more complex queries, use of gap maps becomes more difficult.

When a query contains predicates, for accurate certificate estimation, the gap map must be filtered to only include join key attributes that belong to tuples which pass the predicate, either on-the-fly or as a precomputed map. Similarly, as the number of join attributes grows, more gap maps are needed; relations with multiple join attributes must store one gap map per combination of prefix attribute values. Since the number of gap maps required for exact estimation is dependent on the complexity of the query and the distribution of the data, this can be impractical in many cases. Therefore, we do not present gap maps as a crisp solution to join certificate size estimation but rather as a way to explore the problem and shed light on the issues that can be encountered along the way.

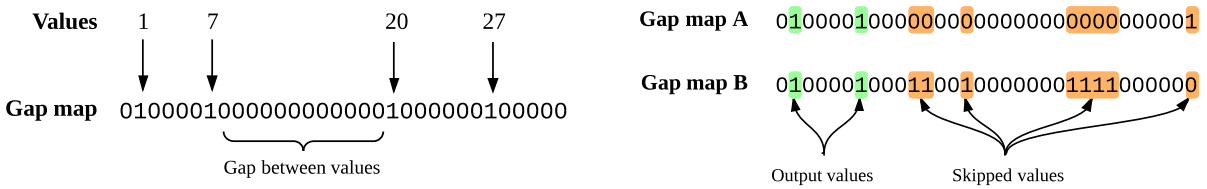
This chapter provides the following contributions:

- The design of gap maps, a histogram variant for tracking absent values that can be easily combined to estimate the size of a join certificate.
- Algorithms for constructing gap maps efficiently and for using them to estimate certificate sizes without decompression.
- An in-depth discussion of when gap maps are effective and the challenges involved in scaling them to arbitrary datasets and queries.

The rest of this chapter is organized as follows. We introduce gap maps, and how to construct and use them, in Section 4.2. We perform an empirical analysis of gap map performance and space requirements over synthetic data in Section 4.3, and we discuss the impact of query and data complexity on the number of gap maps required in Section 4.4. Finally, we conclude in Section 4.5.

4.2 Gap maps

A gap map is a bitmap where each bit represents a possible join key value, built over a join attribute in an input relation. We assume that the domain of join key values is finite and discrete; in the case



(a) Gap map for a list of unsigned integer values.

(b) Information gained by combining gap maps.

Figure 4.1: A simple gap map example.

of continuous types, such as decimal numbers, we assume that they are quantized to a discrete set at the cost of some precision. We align all gap maps to a common starting value (e.g., the minimum possible value), and then we set each bit in the gap map to 1 if the corresponding join key value is present in at least one tuple in the input relation (see Figure 4.1a). Each run of 0 bits in the map indicates a *gap* in the input data where no join key values exist.

By combining the gap maps of different relations, we can estimate the size of the optimal certificate for joining those relations by determining which join key values will be skipped (see Figure 4.1b). In this section, we describe how to construct gap maps efficiently, and we propose two algorithms for using them to estimate join certificate sizes that provide varying levels of accuracy with differing performance. The bulk of our discussion focuses on a simple case with two relations and a single join attribute; we discuss more complex queries in Section 4.4.

4.2.1 Building gap maps

In principle, a gap map requires one bit per possible join key value. While this is feasible for data types with rather limited ranges, like 32-bit integers, it is not feasible for data types with a large number of possible values, like 64-bit integers (an uncompressed gap map over a 64-bit integer would require 2 EB to store). However, in practice, gap maps are much smaller than this worst-case scenario, and this happens due to two reasons.

First, the join key values that are actually present in a relation are highly likely to be only a small fraction of the possible values. Since gap maps are only sensitive to *unique* join key values, not repeated values, to actually fill the entire range of a 64-bit integer would require 2^{64} tuples, which

themselves would be difficult to store. As a result, gap maps compress very well using standard bitmap compression techniques like run-length encoding (RLE) [49].

Second, gap maps do not need to use a one-to-one mapping between values and bits. We say that each gap map has a *bin size* b, which indicates how many values each bit represents. This decreases the size of the uncompressed gap map by a factor of b, at the cost of a loss of precision when estimating the certificate size. The estimated certificate size for a given gap map may be off by a factor of $2b - 1$, since combining two bins with b values each may contribute $2b - 1$ comparisons to the certificate if the values in the bins are perfectly interleaved. Furthermore, because gap maps are intended to be combined with others, gap maps with different bin sizes must be lossily converted to have the same bin size; therefore, in an ideal situation, all gap maps in a database have the same bin size.

All gap maps over a given relation's join attributes can be built in a single scan. Because uncompressed gap maps can be impractically large, it is not feasible to build a gap map and compress it after the fact. Instead, we incrementally construct a word-aligned hybrid (WAH) compressed bitmap [100]. In a WAH bitmap, there are two types of words: fill words and literal words.

Fill words encode runs of repeated bits (either 0 or 1), while literal words store bits as-is. When constructing a gap map, we instantiate literal words as needed, and we add fill words only after all the input has been consumed. This allows us to take advantage of the fact that join key values occupy only a small portion of the value space, since we implicitly store zero fills during construction, and we never have to break a fill word into literals.

To accomplish this, we must be able to determine if a literal word has already been instantiated, and we need to be able to determine how many fill words should separate each literal word. Accordingly, we store literal words in a tree indexed by their offset from the minimum value in the gap map. Since each word represents a fixed number of values based on the bin size, it is easy to calculate a given word's offset. Thus, when we read a new input value for the gap map, we look for its corresponding literal word in the tree, instantiate it if needed, and set the appropriate bit. When all input values have been read, we "stitch" the literal words together with appropriate zero-fill words,

and we also combine repeated words of set bits into one-fill words as we encounter them.

We show the pseudocode for this algorithm in Algorithm 6. Note that our implementation stores values in order from the least significant bit to the most significant bit. This decision is arbitrary, but it does affect the algorithms we use to combine gap maps later.

Algorithm 6 Algorithm to build a gap map for attribute A_1 in relation R with bin size b and w-bit words.

```
words = ∅
r ← GetNext(R)
while r ≠ End(R) do
    a ← ⌊r.A_1 / b⌋
    word_pos ← ⌊a / (w−1)⌋
    offset ← a − (word_pos × (w − 1))
    if no word exists in words with position word_pos then
        words[word_pos] ← 0
    set bit offset in words[word_pos] to 1
last_word_pos ← −1
run_count ← 0
output ← ∅
for word_pos, word in words do
    distance ← word_pos − last_word_pos
    if distance > 1 then                          ▷ There is an implicit run of 0 bits.
        if run_count > 0 then
            add a one-fill word containing run_count runs to output
            run_count ← 0
        add a zero-fill word containing distance − 1 runs to output
    last_word_pos ← word_pos
    if word has all bits set to 1 then
        run_count ← run_count + 1
    else                                          ▷ Current run of 1 bits is broken.
        if run_count > 0 then
            add a one-fill word containing run_count runs to output
            run_count ← 0
        add word to output
if run_count > 0 then                             ▷ May have ended with a run of 1 bits still in progress.
    add a one fill word containing run_count runs to output
    run_count ← 0
```

4.2.2 Querying gap maps

For this section, we assume a query $R(A_1) \bowtie S(A_1)$ with gap maps $M_{A_1}(R)$ and $M_{A_2}(S)$, and we introduce two algorithms for estimating the size of the join certificate.

The first algorithm, which we call *Bitwise*, uses bitwise operations to combine the gap maps. These bitwise operations are fast, but this algorithm tends to overestimate the size of the certificate. The second algorithm, which we call *Merge*, emulates the behavior of ZigZag merge on the bits of the gap maps. This algorithm is slower, but provides much more accurate estimates.

Bitwise

In Bitwise, we break our join certificate size estimation into two parts: first, we estimate the number of skips that will result from gaps in S, and second, we estimate the number of skips that result from R. We can estimate the number of skips that result from gaps in S by computing a new gap map $M = M_{A_1}(R) \wedge \neg M_{A_1}(S)$, which indicates the values that are in R but fall into some gap in S.

More specifically, we know that each *run* of set bits in M indicates a single skip, since all of those values fall into the same gap. Consequently, our challenge with Bitwise is to compute the number of runs of set bits, and this is indicated by the number of transitions from 0 to 1 in M (i.e., the number of 1 bits whose preceding bit is 0). We now show how to compute this without decompressing the gap map.

Since we use WAH compression for gap maps, we must consider how to handle both fill words and literal words. Fill words are straightforward. Each fill word represents a single contiguous run of either 0 or 1 bits. Accordingly, we care only about runs of 1 bits, and we simply need to increment our count by one if the last bit in the preceding word was a 0 (bits preceding the first word in a gap map are implicitly 0).

Literal words are somewhat more complicated. To count the number of runs within a literal word w, we need to compute $\textsc{PopulationCount}((w \oplus (w \ll 1)) \wedge w)$ (where $\textsc{PopulationCount}$ is a function that counts the number of set bits in a bitmap and $\ll$ is a bitwise left shift operator).

Algorithm 7 Algorithm to count the number of runs in a gap map consisting of w-bit *words*.

```
count ← 0
prev_bit ← 0
for each word in words do
    if word is a fill word then
        current_bit ← the fill bit in word
        if last_bit ≠ current_bit then
            count ← count + 1
        prev_bit ← current_bit
    else
        shifted_word ← (word ≪ 1) ∨ prev_bit
        count ← count + PopulationCount((shifted_word ⊕ word) ∧ word)
        prev_bit ← word ∧ (1 ≪ (w − 2))
return count
```

We propagate the last bit from the previous word (0 if no previous word exists) as the least significant bit in $w \ll 1$. The XOR between w and $w \gg 1$ gives us an initial count for the number of runs, but it can potentially count a single run multiple times—a run of set bits that is entirely contained within the word will be counted twice, once for the beginning of the run and once for the end. To address this, we AND the result with w, which ensures that each run will be counted at most one time. We show the pseudocode for this algorithm in Algorithm 7.

M captures how many runs in R will be skipped by gaps in S, but both ZigZag merge join and Minesweeper skip on both sides of the join. Accordingly, we must also compute $M_{A_1}(S) \wedge \neg M_{A_1}(R)$ and count the number of runs of set bits it contains. This technique provides an upper bound on the size of the optimal certificate, but it can overestimate the certificate size in many cases.

This occurs because of the interaction between seeks from R and seeks from S. Some values in R that may cause seeks in S may end up skipped themselves, and as a result, they do not actually cause a skip. Since we decomposed certificate size estimation into two discrete steps, we have no way of handling this case using the bitwise algorithm, and we introduce a second algorithm to address this.

Merge

Our second algorithm for estimating the size of a join certificate emulates the behavior of ZigZag merge join. We use two iterators over $M_{A_1}(R)$ and $M_{A_1}(S)$. Each iterator tracks its current bit

Algorithm 8 Algorithm to estimate the size of the certificate between two gap maps, $left$ and $right$.

Input: $left\_it$ and $right\_it$, iterators over the bits of $left$ and $right$, respectively
```
advance left_it to the first set bit
advance right_it to the first set bit
count ← 0
while left_it ≠ End(left) and right_it ≠ End(right) do
    if Position(left_it) = Position(right_it) then
        advance left_it to the next set bit
        advance right_it to the next set bit
    else if Position(left_it) < Position(right_it) then
        advance left_it to Position(right_it)
        if left_it is not pointing to a set bit then
            advance left_it to the next set bit
        count ← count + 1
    else
        advance right_it to Position(left_it)
        if right_it is not pointing to a set bit then
            advance right_it to the next set bit
        count ← count + 1
return count
```

position, and it provides a method for skipping to the next set bit.

Because the bitmaps are compressed, finding the next set bit can be quite efficient. Zero-fill words enable skipping of entire runs of zeros very quickly, and for literal words, many processors provide instructions that compute the next set bit directly. Therefore, we can implement a ZigZag merge-style algorithm over the gap maps with reasonable performance.

The overall logic is straightforward. We begin by placing each iterator at the first set bit, and then we compare their positions. If their positions are equal, we advance both iterators to the next set bit. If $M_{A_1}(R)$'s position is less than $M_{A_1}(S)$'s, we increment our count and advance it to the first set bit whose position is greater than or equal to $M_{A_1}(S)$'s position, and vice versa if $M_{A_1}(S)$'s position is less than $M_{A_1}(R)$'s. We show the pseudocode for this algorithm in Algorithm 8.

This algorithm produces exact certificate size estimates when the bin size is 1, since it is essentially equivalent to ZigZag merge join.

4.3 Experimental results

We implemented gap maps in a C++ application and examined the performance of building and querying them over synthetic data. Our datasets covered four different distributions, each with N tuples. Specifically, we generated uniform random data, Zipfian data (with $s = 1$), normally distributed data (with $\mu = \frac{N}{2}$ and $\sigma = \frac{N}{100}$), and data that followed the ClusterData model [3]. We store each dataset in its own relation, and each relation has only a single join attribute.

In this section, we construct 32-bit gap maps exclusively. Gap maps can also be stored in 64-bit words, but this causes them to require substantially more space, and our preliminary testing did not show a significant performance improvement over 32-bit gap maps.

These results are intended to provide a flavor for gap maps' efficacy in various situations, and they are not intended to be a comprehensive exploration of the entire space.

4.3.1 Building gap maps

Constructing gap maps is sensitive to the number of tuples that need to be processed and the distribution of the join attribute values in those tuples. When join attribute values are repeated, gap maps take less time to construct, since they do not need to instantiate new words as often. Similarly, when join attribute values are clustered around a certain part of the space, particular words are accessed repeatedly, making them more likely to be already instantiated and cached.

We show the performance of building gap maps with bin size 1 as the number of tuples increases in Figure 4.2. Even at 100 million tuples, gap maps take less than a minute to construct. Uniformly distributed data is the slowest to construct, since it has the most unique values and covers the broadest range of the space. ClusterData and Zipfian data both have more redundancy in their values, and so take less time. Finally, our normally distributed data is tightly clustered and has the most redundancy, and therefore is constructed the fastest.

The size of a gap map is also sensitive to the join value distribution, since repeated values do not require additional space. We show the size of a gap map (in words) in Figure 4.3. Here we can see

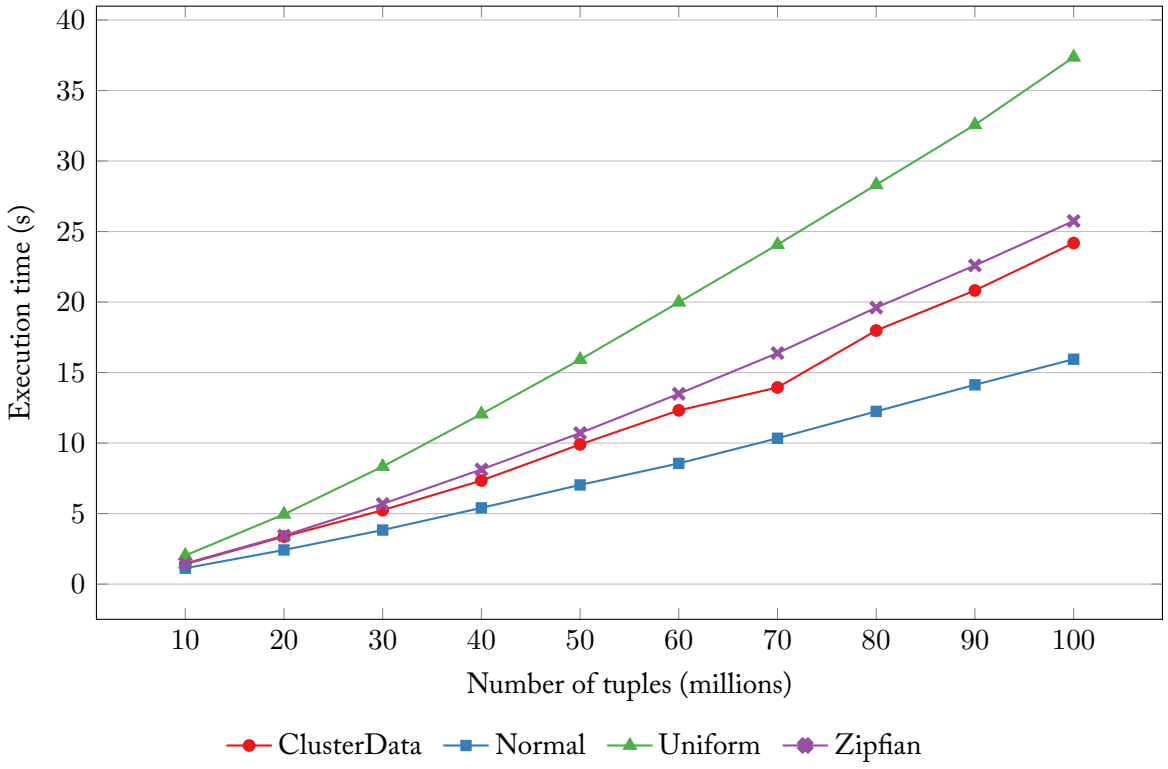


Figure 4.2: Gap map build performance as the number of tuples increases with bin size 1.

that, although we constructed a gap map over data from the ClusterData model in about the same time as a gap map over Zipfian data, the ClusterData gap map required roughly double the space. This is because, although ClusterData exhibits more locality than uniformly distributed data, it still covers more of the space than Zipfian data does. Lastly, as expected, our tightly clustered normally distributed data required the least amount of storage space.

Overall, gap maps are reasonably compact. For highly redundant data, like our normally distributed data, the gap map for 100 million tuples can be stored in about 1 MB. Even for uniformly distributed data with little redundancy, a 100-million tuple gap map requires only 24 MB. Thus, we feel it is reasonable to assume that an entire gap map could be cached in memory.

4.3.2 Querying gap maps

We compared our two algorithms for certificate size estimation, Bitwise and Merge, on gap maps with 50 million tuples. We estimated the size of the certificate with bin size 1 over all combinations

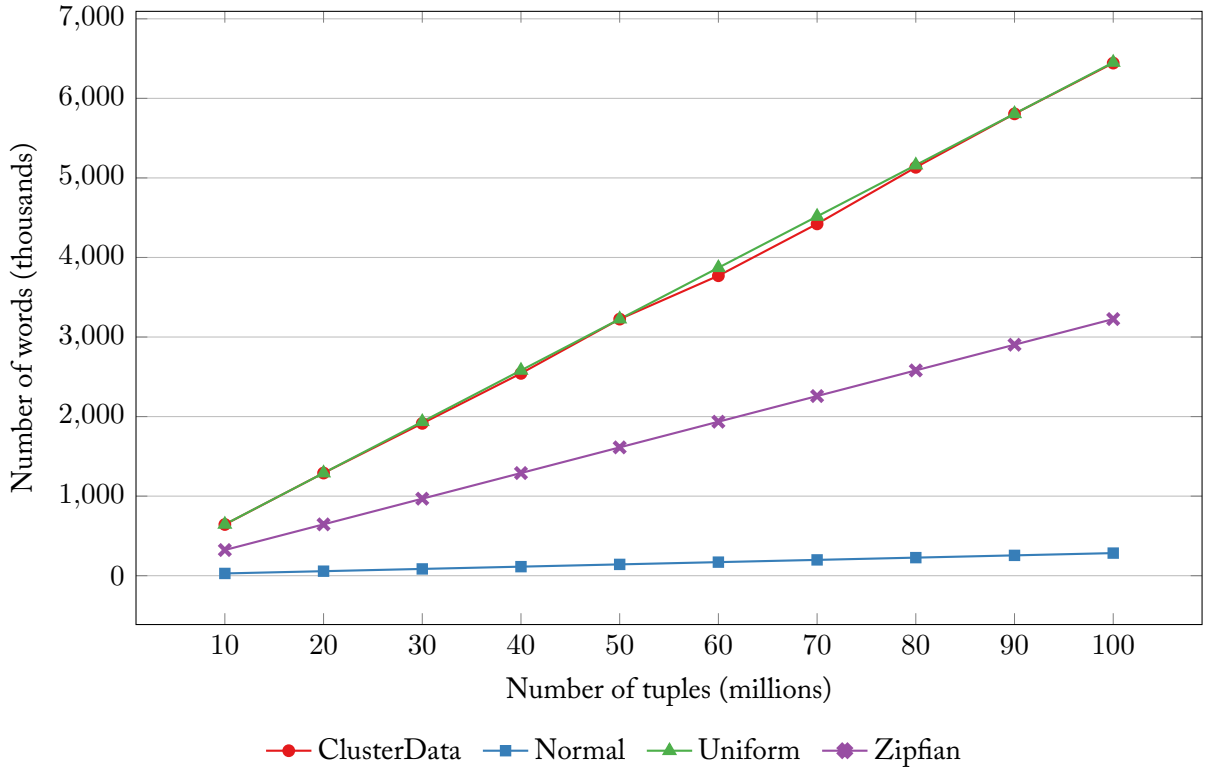


Figure 4.3: Gap map size as the number of tuples increases with bin size 1.

of distributions, and we show our results in Table 4.1. In this experiment, Merge produces the exact certificate size, while Bitwise often overestimates the size of the certificate.

Bitwise provides the closest estimate when data is uniformly distributed or distributed according to the ClusterData model, offering estimates that are 15–30% off from the true value. However, when data is normally distributed, Bitwise can be off by an order of magnitude or more. This is because our normal distribution is tightly clustered, and therefore, it produces large skips in the relation it joins with. These skips eliminate many of the seeks that Bitwise expected from that relation, which Bitwise does not account for.

On the other hand, Bitwise does provide better performance than Merge in several cases. This is because Bitwise performs operations on a per-word basis. Every fill and literal word has the same bitwise operations applied to it, and therefore Bitwise's per-word processing cost is fixed. In contrast, Merge jumps from set bit to set bit, and some words take longer to process than others. Accordingly, for gap maps with a large number of set bits, Merge will do much more processing

		Estimated cert. size (k)		Execution time (ms)	
R distribution	S distribution	Merge	Bitwise	Merge	Bitwise
ClusterData	Normal	1673.97	11,842.44	37.99	37.99
	ClusterData	26,242.91	33,401.36	447.90	99.98
	Uniform	28,153.67	34,826.42	487.89	100.98
	Zipfian	9095.07	14,375.96	140.97	53.99
Uniform	Normal	1695.98	13,299.00	38.99	36.99
	Uniform	29,712.96	35,689.35	514.89	93.98
	Zipfian	9845.71	15,406.89	150.97	55.99
Zipfian	Normal	547.10	7277.41	13.00	25.99
	Zipfian	5922.11	8097.52	136.97	50.99
Normal	Normal	690.69	859.90	42.99	7.00

Table 4.1: Certificate estimation results depending on distribution for 50 million tuples with bin size 1.

than Bitwise, but provide a more accurate answer.

Our next experiment evaluates the accuracy of gap maps as the bin size increases. We expect that accuracy will degrade with larger bins, since binning values has the potential to remove some gaps from the map, and we show results for uniformly distributed data in Figure 4.4; other distributions are similar. Unfortunately, gap map accuracy quickly degrades as the bin size increases. Even a bin size of 2 is sufficient to introduce more than 40% error, and by a bin size of 6, gap maps are effectively useless.

This is because the size of the certificate is very sensitive to each individual gap. A bin size of 6 means that any gap of 12 values or less will be eliminated. In our synthetic datasets, most of the gaps fall within this range, and as a result, our estimation techniques will dramatically underestimate the size of the certificate.

Interestingly, although Bitwise overestimates the size of the certificate when the bin size is 1, as the bin size increases, its error rate is almost identical to Merge. Recall that Bitwise does not account for the fact that a single skip from one relation may eliminate several skips in the other relation. However, the eliminated skips tend to be close together, and as the bin size increases, those skips merge into a single run, and end up being counted correctly. As a result, when operating over

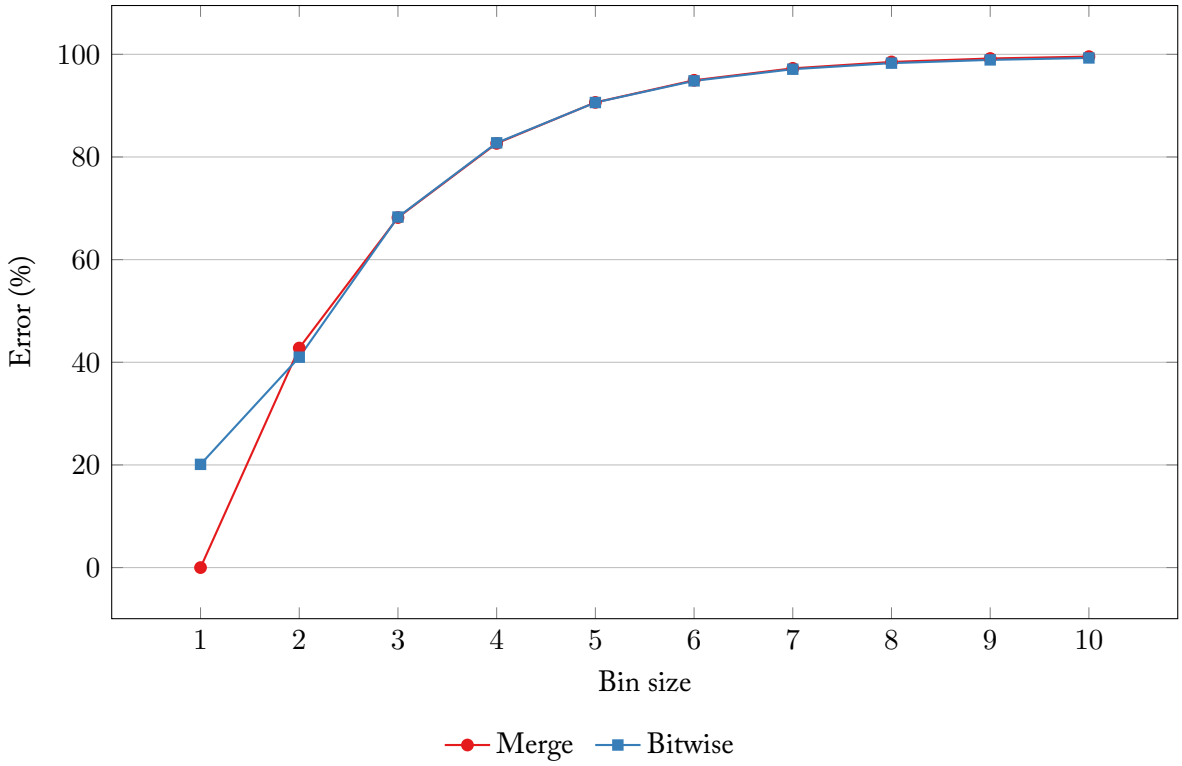


Figure 4.4: Certificate size estimation accuracy for uniformly distributed data with 50 million tuples as bin size increases.

gap maps with larger bin sizes, the choice of algorithm is less important than when exact gap maps (i.e., bin size 1) can be used.

Finally, we investigate how gap maps can be applied to two generic relational queries, specifically our versions of Star Schema Benchmark Q2.1 and Q2.3. Our results in Chapter 3 showed that ZigZag merge join offered almost no performance advantage over hybrid hash join for Q2.1, while ZigZag merge join provided much better performance than hybrid hash join on Q2.3. While we do not have a full cost model, we can use gap maps to provide an intuition as to why ZigZag merge join's performance differs for the two queries.

To do this, we first construct a gap map over the partkey attribute in the LineOrder relation. Then, we construct two additional gap maps over the Part relation. For Q2.1, we construct a gap map on partkey for all tuples with category equal to "MFGR#12," and for Q2.3, we construct a gap map on partkey for all tuples with brand equal to "MFGR#2239."

By combining these gap maps with the gap map on LineOrder, we can estimate the size of the certificate for the first join in the query. Using Merge, Q2.1's gap maps estimate the first join requires 45,902 seeks to find 47,850 matching values. This is obviously many fewer seeks than required to answer the entire query, but it indicates that there is little skipping in the first join, since the number of seeks is very close to the number of matching values.

In contrast, running Merge on Q2.3's gap maps indicates that its join requires only 1,185 seeks to find 4,154 matching values. As a result, not only does it require fewer seeks in absolute terms than Q2.1, its ratio of seeks to matching values is much better, indicating that more data is being skipped. This shows that gap maps can provide some evidence as to whether a query would be a good fit for ZigZag merge join, at least in simple cases.

To accurately estimate the size of the certificate for the entire query would require combining gap maps across multiple joins, which poses problems both in terms of the amount of computation and the amount of storage space required. We discuss these issues briefly in the next section.

4.4 Query complexity

Our results in this chapter address a two-relation join with a single join attribute. However, this is the ideal case, and when queries are more complex, gap maps become more difficult to use effectively. More specifically, the number of gap maps that must be stored and combined for an arbitrary query can increase dramatically in some situations.

For example, suppose we we have a query $\sigma_{A_1=x}(R(A_1, A_2) \bowtie S(A_2))$, where x is an arbitrary A_1 value. To effectively determine the size of the certificate for this query, we need to use a gap map built over $\pi_{A_2}(\sigma_{A_1=x}(R))$; since x is an arbitrary value, to accommodate any possible query we will require one gap map per unique A_1 value. This means the number of gap maps is dependent not only on the complexity of the query, but also on the distribution of the *data*.

This is exacerbated for queries with more than two relations. Suppose we have a three-relation join, $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2)$. The certificate for $R \bowtie S$ can be determined with just two gap

maps; however, the certificate for $S \bowtie T$ is dependent on the distribution of each run of A_2 values. Accordingly, we need a gap map per unique A_1 value that appears in S, even though we do not have any predicates in the query.

Furthermore, when estimating the certificate size, we need to combine the gap maps for each A_2 value in $\pi_{A_2}(R \bowtie S)$ with the gap map for T. This means we may end up doing a lot of work to determine which gap maps need to be combined, and this grows as queries involve more joins. Therefore, for large relations or complex queries, the sheer number of gap maps needed can quickly become unmanageable and impractical.

One approach for dealing with this might be to combine multiple gap maps into a single approximate gap map. However, any approximation of a gap map will suffer from some degree of inaccuracy, and we showed in Section 4.3 that certificate estimation is very sensitive to approximation errors. As a result, it is not clear how to combine multiple data distributions in a way that limits error strongly enough to be useful in practice.

4.5 Conclusion

In this chapter, we introduced gap maps, a first attempt at statistics for estimating the size of a join certificate. Although gap maps can be constructed and queried efficiently for a two-relation join without predicates, there are several challenges in scaling gap maps to more complex queries, in particular the large number of gap maps required to produce accurate estimates. Furthermore, while gap maps can in principle represent multiple values in a single bit, the error in their estimated certificate size grows dramatically as the bin size increases, even for just two values per bin.

These issues combined suggest that gap maps, and perhaps binning synopsis structures in general, are not an effective solution to this problem. Instead, it may make more sense to rely on strategies to correct poor skipping decisions after the fact. For example, if an RDBMS notices that a skipping join algorithm is not skipping effectively, it may make a runtime decision to revert to a simpler variant (e.g., convert from Minesweeper to ZigZag merge join, or from ZigZag merge join to

traditional merge join) or in some cases simply stop execution and reoptimize, as in progressive optimization [63, 66].

Additionally, although we focused on certificate size estimation in this chapter, the optimal certificate depends on the global attribute order. Therefore, a related area of work is to advise RDBMS administrators on the best indexes to construct to optimize the performance of skipping join algorithms and to integrate this feedback into a database engine tuning advisor [1, 24, 28]. Gap maps may be better suited to this area than to query-time optimization, since an offline advisor process has more time to construct gap maps, and after tuning is complete, gap maps would no longer need to be stored, alleviating storage issues for complex queries.

5 | Related work

5.1 Full-text search in databases

The integration of RDBMSs and IR systems has been discussed repeatedly over the years. Biller [18] and Stonebraker et al. [89] argued for the inclusion of text storage features in an RDBMS as early as 1982. More recently, several members in the database community have noted that the functionality of RDBMSs and IR systems has been converging [25, 96] and that some new RDBMS features could have a large impact [19].

Heman et al. [51] and Cornacchia et al. [26] demonstrated that inverted indexes can be processed efficiently in an open-source column-store database. They achieved high query performance through the use of vectorized execution and novel compression techniques with a PAX-like layout [2]; however, their work used the traditional merge join algorithm. Accordingly, our work is complementary to theirs, since we demonstrate a way to further increase performance through algorithmic improvements.

Whang et al. [97] implemented Odysseus, an ORDBMS with a number of IR features. The overall design of Odysseus is similar to the specialized design we discussed in Section 2.3.1, with the addition of a full B-tree index over each posting list rather than a simple skip list. This work further highlighted the possibilities of a specialized design, but did not compare it to a pure relational approach.

DeFazio et al. [29] proposed cooperative indexing as a way to provide some of the storage-level optimizations used by IR engines when storing the inverted index in relations. This work investigated scalability, rather than raw performance, and given its focus on the storage layer, it did not include a

discussion of the join algorithms used.

The pure relational approach can also be more complex than we studied in this thesis. Holmes [52] showed that SQL can be used to parse keyword queries, and several researchers have demonstrated relational IR in parallel settings [37, 39, 68]. The main results of this work are demonstrations of the flexibility and power of the relational approach, rather than performance.

Set-containment joins could also be used to evaluate conjunctive queries by viewing each document as a set of terms and checking to see if that set contained the set of keywords in the query. This would enable the use of algorithms such as those proposed by Ramasamy et al. [82] and Melnik et al. [69], which utilize hash-based signatures of sets and partitioning, and it could possibly allow for better performance in systems that support set-valued attributes. We do not compare against these algorithms in this work, but this represents an interesting area for future exploration.

Some previous work has compared relational IR systems with specialized IR engines. Kaufmann et al. [57] and Grossman et al. [46] showed that relational IR can be competitive with special-purpose IR engines, and Ercegovac et al. [33] performed a general benchmark of full-text search features across RDBMSs. Because these benchmarks were cross-system, they highlighted general trends and did not dive deeply into the reasons for the performance differences, and thus did not expose the algorithmic differences that our single-system experiments showed.

Finally, a related area of research is keyword search over relational databases, such as that done by Bahlotia et al. [16] and Hristidis et al. [53]. This work aims to allow users to pose keyword queries to query structured data instead of SQL, and as such, it is primarily concerned with enumerating possible joining networks that may match a given search; however, inverted indexes are used peripherally in this work to aid in determining which joining networks are of interest or to locate previously stored auxiliary information, like form templates (as in Baid et al. [10]).

5.2 Skipping algorithms

Skipping in the IR community. Skipping among posting lists has frequently been proposed for high performance in the context of set intersection, and the primary issue in this domain is how to determine which set to draw eliminator values (i.e., values used to seek in other lists). Demaine et al. [30] described an algorithm called Adaptive that uses the value from the set with the fewest remaining elements as the eliminator. They later showed that Adaptive can be outperformed in practice by a simpler algorithm that intersects two sets at a time (called Small vs. Small or SvS), and they proposed a hybrid algorithm, Small Adaptive, to address this issue [31]. Barbay et al. [11] suggested rotating among sets, drawing eliminators from each in turn, producing an algorithm called Sequential, and Baeza-Yates [8] proposed an adaptive algorithm that combines the properties of a binary search-based variant of nested loops and merge intersection. Barbay et al. [12] extended this work by comparing different algorithms for seeking in sets, showing that interpolation search can outperform binary and gallop search.

This work is largely complementary to this thesis. Trees of binary ZigZag merge join operators are equivalent to an interleaved version of Sequential, and Minesweeper is analogous to a multi-way intersection algorithm. In principle, many of the algorithms proposed for set intersection can be applied to joins; however, our focus is less on particular algorithms and more to show that the general concept of skipping join algorithms is important. Furthermore, our experimental results showing that ZigZag merge join can provide better performance than Minesweeper in practice is similar to studies of set intersection algorithms that show simpler algorithms outperforming more complicated algorithms that have better theoretical bounds [9, 13, 27, 31].

Skipping in the RDBMS community. The RDBMS community has also explored the benefits of skipping in a number of different ways. These algorithms include many different methods, including both indexes and hashing, and we discuss a selection of them here.

Graefe [40] discussed efficient ways of implementing nested queries using nested iterators. These nested iterators can take advantage of indexes in some cases, and can be made more efficient by

using spool operators, but they bind (i.e., seek) only on one side of a join. Consequently, ZigZag merge join differs in that it binds on *both* sides; additionally, many of the optimizations discussed by Graefe can be applied to the ZigZag merge context in a complementary fashion.

Raman et al. [81] investigated intersection of RIDs in the context of index ANDing. They use an adaptive, probabilistic algorithm for determining the order in which sets should be intersected, and they store the high RID value on each page as a way to quickly eliminate pages with no matching values. However, they do not use the upper levels of the B-tree to skip, and consequently have to binary search in a list of all leaf pages when probing for a RID.

Similarly, Bucket-Skip Merge Join (BSMJ) [56] divides input relations into buckets and stores a high and low key for each bucket; all buckets are stored in sorted order by the low key. This allows BSMJ to skip buckets of tuples at a time, since if two buckets have disjoint key–value ranges, they cannot possibly contain any matching tuples, but they do not make use of any secondary indexes.

Diag-Join [50] exploits time-of-creation clustering to enable skipping on one-to-many joins. It uses a sliding window across its input relations to limit the amount of tuples joined at any given time, and it achieves high performance by building a hash table over the window to accelerate probes. This strategy works only when time-of-creation clustering is present and the join is one-to-many—it is not applicable to arbitrary data and queries.

Criss-Cross Hash Join (CCHJ) [38] arranges pages into contiguous zones with one zone per hash bucket. It then processes relations zone-by-zone, alternating hashing over both relations ("criss-crossing"). This strategy allows CCHJ to outperform hybrid hash join in some situations, but it does not permit arbitrary skipping of tuples.

Cao et al. [22] introduced SCALE, a variant of sort–merge join optimized for self joins. SCALE assumes that the self join is on two separate attributes (i.e., $R \bowtie R$ on $R.A = R.B$). We do not consider this case in our work, as the self join used in text processing is on a single attribute.

ZigZag merge join has been applied in several areas. Halverson et al. [47] used a ZigZag algorithm to efficiently handle inverted index queries for XML processing in an RDBMS, and Oracle Rdb implemented ZigZag merge join for general queries [7]. However, neither of these

pieces of work has discussed the practical performance impact of ZigZag merge join on full-text queries and relational processing.

Leapfrog Triejoin [95] is a multi-way skipping join algorithm that is closely related to ZigZag merge join. Leapfrog Triejoin divides a query into levels, with one level per join attribute; each level contains a single Leapfrog join operator that performs a multi-way join between all relations in the query that have that join attribute. When a match has been found at one level, Leapfrog Triejoin advances to the next level and attempts to find a match for the next join attribute given the values found for all previous attributes. As a result, when each join attribute appears in a maximum of two relations, Leapfrog Triejoin is identical to a binary tree of ZigZag merge join operators, and therefore our results from Section 3.6.2 apply to Leapfrog Triejoin as well.

Graefe mentions that skipping could be combined with a generalized join (g-join) algorithm [41]. However, the intent of this work is to achieve predictable performance with a single general-purpose join algorithm (rather than an optimizer-based choice among a set of algorithms), and skipping is mentioned only in passing. In contrast, the purpose of this thesis is to explore the performance impact skipping can have on a variety of situations.

Ngo et al. [75] discuss worst-case performance for Leapfrog Triejoin and an earlier algorithm by Ngo et al. [74], and they create a general join framework that shows both of these joins are special cases of a single join algorithm. This work considers optimality in terms of the input size of an instance, and its results are extended by Ngo et al.'s description of join certificates and analysis of Minesweeper [73]. As ZigZag merge join is closely related to Leapfrog Triejoin, our experimental evaluation in this thesis is a companion to the theoretical analysis in Ngo et al.'s work.

5.3 Statistics for skipping

Although the notion of a certificate is present in both set intersection [11, 30] and join algorithms [73], to the best of our knowledge, no work has been done on a way to estimate the size of a certificate prior to running the join. However, there has been a great deal of work on statistics for

join cardinality estimation, which is closely related.

Histograms are a very common structure used for database statistics [54]. However, many histograms assume a uniform distribution within bins (though this is sometimes relaxed [14, 71, 78]), and they construct bins according to specified rules, e.g., equi-width, equi-depth, or some other optimality metric [55]. These constructions typically fail to capture join key attribute values that are missing from the data, and the same is true for alternative synopsis structures like wavelets [67].

While it is reasonable in the case of join cardinality estimation to misestimate missing values—after all, these synopsis structures are likely to state that the value is *low*—skipping join algorithms are very sensitive to this property. Whether a tuple is present or not can make the difference between skipping or not skipping, and the cumulative error caused by these misestimations can dramatically change the expected performance of a skipping join algorithm.

A similar argument applies when considering sampling-based techniques [34, 61, 65, 72, 91]. A sample of the data is naturally going to have many gaps, and it is not clear which gaps are "natural" (i.e., are in the real data) and which gaps were introduced by the sampling process.

Gap maps are somewhat similar to bitmap indexes [23]. In fact, if the bin size is 1 and the join key attribute is unique, a gap map will store essentially the same content as a bitmap index. However, gap maps support larger bin sizes, and additionally, since they are not used to answer queries, gap maps may be inaccurate and do not need to be constantly maintained.

Gap maps are not the first bitmap-oriented histogram. Furfaro et al. [35] introduced Grid Hierarchical Binary Histograms (GHBHs). GHBHs partition the value space into regions of varying detail; a region where values are far from uniformly distributed will be subdivided into smaller regions, and each summary statistic (e.g., the count of values in a region) is stored efficiently along the hierarchy of regions. However, GHBHs are still not designed to capture missing values, and because they store full summary statistics, they require somewhat more space than gap maps.

We used Word-Aligned Hybrid (WAH) compression from Wu et al. [100] for gap maps. However, many other types of bitmap compression techniques have been proposed [4, 6, 62, 87], and these schemes could also be used for gap maps.

6 | Conclusion and future work

In this thesis, we demonstrated that skipping join algorithms can provide better performance versus traditional join algorithms over a variety of workloads, including in-RDBMS full-text search and general relational queries like those used in Star Schema Benchmark and TPC-H. We compared two skipping join algorithms, ZigZag merge join and Minesweeper, and found that, while Minesweeper has better worst-case performance, ZigZag merge join often performs better in practice. Finally, we introduced gap maps, a first attempt at statistics for estimating the amount of work a skipping join algorithm will perform prior to execution, and we showed that while they are efficient to construct and query, the number of gap maps required for complex queries and data is impractical.

A central issue in this work is what input is reasonable for a join algorithm, or in other words, what assumptions can be made about how the data is stored and indexed. Clearly, if no information is known about how the data is stored, it is difficult to find optimize for performance. In this case, it seems likely that, lacking a better option, we would fall back on hash join.

On the other hand, if we were given more information, we could optimize further. If the input is sorted, merge join may be faster than hash join; if one input is small and the other is indexed, index nested loops join may be the best choice. These are commonly known scenarios in the database context, and our work has focused on an even more particular case where all inputs are indexed according to a global attribute order. Here, we have shown that skipping join algorithms like ZigZag merge join and Minesweeper can provide better performance on a variety of workloads and datasets, including both in-RDBMS full-text search and general relational benchmarks like Star Schema Benchmark and TPC-H. On some level, this is an intuitive result—these algorithms make the best use of the additional constraints on the input data, and so provide the best performance.

Nonetheless, this does not mean that we have explored the entire space. To suggest a few options, one could imagine that the input data may have fewer assumptions (e.g., perhaps only half of the relations are indexed appropriately), or there may be detailed auxiliary statistics available on the populated range of join key values. Join algorithms that are able to adapt to these situations may be able to provide better performance.

In the limit, of course, the result of the join is precomputed, and the optimal "join algorithm" for this data simply reads it out. While this may sound a bit absurd, consider that this is essentially a materialized view, a feature already supported by a large number of relational database management systems. As a result, we feel that it is not too outlandish to expect that there may be a wide range of assumptions that may hold for a given input, and different assumptions will provide unique opportunities.

For example, Minesweeper utilizes only a single global attribute order, and this can impact its performance. Consider a query like $R(A_1) \bowtie S(A_1, A_2) \bowtie T(A_2)$, and suppose we populate it as follows:

- $R(A_1)$: $[1, \ldots, N]$
- $S(A_1, A_2)$: $[1, \ldots, N] \times [1, \ldots, N]$
- $T(A_2)$: $[N + 1, \ldots, 2N]$

The output of this query is empty—no value in $\pi_{A_2}(S)$ exists in T—but Minesweeper cannot quickly determine this fact. Since Minesweeper assumes that S is sorted on (A_1, A_2), any constraints Minesweeper learns on $\pi_{A_2}(S)$ are conditioned on some A_1 value, and Minesweeper must evaluate every run of A_2 values in $R \bowtie S$ to see if it joins with T. This makes sense if there is only a single global attribute order, but suppose we have an additional index I on $S(A_2)$.

Index I is not consistent with the global attribute order, and in fact, it is not generally suitable for evaluating this query as it provides no efficient way to join S with R. Nonetheless, it can provide useful information; in this case, two seeks in I could be used to determine the minimum and

maximum A_2 value in S, and so establish that the query is empty very rapidly. Thus, an algorithm that takes advantage of this index can be arbitrarily faster than one does not.

We can integrate this processing very simply into Minesweeper. Minesweeper already performs multiple seeks in each relation along the global attribute order; for the above query, Minesweeper will seek in S for some (A_1) tuple and some (A_1, A_2) tuple. Consequently, all Minesweeper must do is check for the presence of an additional index on A_2 (or, more generally, any join attribute in the relation) and probe it as well.

One may be wondering how this modified algorithm relates to the theory behind Minesweeper. After all, Minesweeper is optimal in terms of the minimum-size certificate, but we have just shown a variant of Minesweeper that can be arbitrarily better. To understand this, recall that Ngo et al. showed that the minimum-size certificate is dependent on the global attribute order. Consequently, one can see that if the order was $T \bowtie S \bowtie R$, Minesweeper would be just as good as our proposed variant.

However, the global attribute order is not selected arbitrarily—input data must be indexed appropriately to support it. Furthermore, an entirely new global attribute order is not required, since I is a *partial order* that still provides useful feedback. Because the definition of a certificate does not allow for combinations of multiple (possibly partial) orders, the minimum-size certificate cannot capture this information, and therefore Minesweeper is minimum-size certificate optimal despite not covering this case.

This suggests that the definition of a certificate could be broader, and new (or modified) algorithms may be required to be optimal for new definitions and different assumptions on input data. We feel that this is an interesting area of research with a number of possibilities from both a theoretical and a practical perspective, and we look forward to further developments in the future.

Finally, an issue that we have not addressed is how skipping join algorithms can be implemented in real-world systems. Most relational database management systems already support merge join and secondary indexes, and in this case, ZigZag merge join is only a small modification. In fact, our prototype system implements ZigZag merge join and traditional merge join in the same

operator—changing between them is a matter of removing only a few lines of code.

In contrast, Minesweeper's implementation is much more complicated. Unlike secondary indexes, data structures like the CDS are not shared by any other part of the RDBMS, and Minesweeper's multi-way design is not consistent with the binary nature of most traditional join algorithms. As a result, it may make more sense to implement ZigZag merge join as a baseline skipping join algorithm, and consider Minesweeper only if motivated by a specific workload or use case.

A | ZigZag merge join supporting functions

Algorithm 9 Algorithm for outputting the cross product of matching groups in $R \bowtie S$.

```
function ProcessGroup
    if left_seeked then
        r ← GetNext(R)
    if right_seeked or not left_seeked then
        s ← GetNext(S)
    rewound ← false
    while r ≠ End(R) and not output do
        if r = s then
            add r ⋈ s to results
            output ← true
        else
            if rewound then SetLeftRun(true)
                in_group ← false
                left_seeked ← false
                right_seeked ← false
            else
                rewind S to the beginning of the run
                r ← GetNext(R)
```

Algorithm 10 Algorithm for rewinding iterators over operators as needed.

```
function RewindIfNeeded
    if r ≠ End(R) and (s = End(S) or HasLeftRun(R)) then
        rewind S to the beginning of the run
        s ← GetNext(S)
        if not HasLeftRun(R) then
            advance R to the next run
        else
            ClearLeftRun(R)
```

Algorithm 11 Algorithm for seeking in a ZigZag merge join or a base relation.

```
function SEEK(operator, key)
    if operator is a ZigZag merge join then
        if operator.R has join attributes in key then
            operator.left_seeked ← true
            SEEK(operator.R, key)
        if operator.S has join attributes in key then
            operator.right_seeked ← true
            SEEK(operator.S, key)
    else                                          ▷ operator must be a base relation
        perform an index lookup in operator's relation for key
```

B | Star Schema Benchmark queries

Query 2.1

```
SELECT lo_partkey, lo_suppkey, lo_orderdatekey, d_datekey, p_partkey,
   s_suppkey
FROM lineorder, date, part, supplier
WHERE lo_orderdatekey = d_datekey
AND lo_partkey = p_partkey
AND lo_suppkey = s_suppkey
AND p_category = 'MFGR#12'
AND s_region = 'AMERICA'
```

Query 2.2

```
SELECT lo_partkey, lo_suppkey, lo_orderdatekey, d_datekey, p_partkey,
   s_suppkey
FROM lineorder, date, part, supplier
WHERE lo_orderdatekey = d_datekey
AND lo_partkey = p_partkey
AND lo_suppkey = s_suppkey
AND p_brand BETWEEN 'MFGR#2221' AND 'MFGR#2228'
AND s_region = 'ASIA'
```

Query 2.3

```
SELECT lo_partkey, lo_suppkey, lo_orderdatekey, d_datekey, p_partkey,
   s_suppkey
FROM lineorder, date, part, supplier
WHERE lo_orderdatekey = d_datekey
AND lo_partkey = p_partkey
AND lo_suppkey = s_suppkey
AND p_brand= 'MFGR#2239'
AND s_region = 'EUROPE'
```

Query 3.1

```
SELECT lo_custkey, lo_suppkey, lo_orderdatekey, d_datekey, s_suppkey,
   c_custkey
FROM customer, lineorder, supplier, date
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_orderdatekey = d_datekey
AND c_region = 'ASIA'
AND s_region = 'ASIA'
AND d_year >= 1992
AND d_year <= 1997
```

Query 3.2

```
SELECT lo_custkey, lo_suppkey, lo_orderdatekey, d_datekey, s_suppkey,
   c_custkey
FROM customer, lineorder, supplier, date
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_orderdatekey = d_datekey
AND c_nation = 'UNITED STATES'
AND s_nation = 'UNITED STATES'
AND d_year >= 1992
AND d_year <= 1997
```

Query 3.3

```
SELECT lo_custkey, lo_suppkey, lo_orderdatekey, d_datekey, s_suppkey,
   c_custkey
FROM customer, lineorder, supplier, date
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_orderdatekey = d_datekey
AND (c_city = 'UNITED KI1' OR c_city = 'UNITED KI5')
AND (s_city = 'UNITED KI1' OR s_city = 'UNITED KI5')
AND d_year >= 1992
AND d_year <= 1997
```

Query 3.4

```
SELECT lo_custkey, lo_suppkey, lo_orderdatekey, d_datekey, s_suppkey,
   c_custkey
FROM customer, lineorder, supplier, date
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_orderdatekey = d_datekey
AND (c_city = 'UNITED KI1' OR c_city = 'UNITED KI5')
AND (s_city = 'UNITED KI1' OR s_city = 'UNITED KI5')
AND d_yearmonth = 'Dec1997'
```

Query 4.1

```
SELECT lo_custkey, lo_suppkey, lo_partkey, lo_orderdatekey, c_custkey,
   s_suppkey, p_partkey, d_datekey
FROM date, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_partkey = p_partkey
AND lo_orderdatekey = d_datekey
AND c_region = 'AMERICA'
AND s_region = 'AMERICA'
AND (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')
```

Query 4.2

```
SELECT lo_custkey, lo_suppkey, lo_partkey, lo_orderdatekey, c_custkey,
   s_suppkey, p_partkey, d_datekey
FROM date, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_partkey = p_partkey
AND lo_orderdatekey = d_datekey
AND c_region = 'AMERICA'
AND s_region = 'AMERICA'
AND (d_year = 1997 OR d_year = 1998)
AND (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')
```

Query 4.3

```
SELECT lo_custkey, lo_suppkey, lo_partkey, lo_orderdatekey, c_custkey,
   s_suppkey, p_partkey, d_datekey
FROM date, customer, supplier, part, lineorder
WHERE lo_custkey = c_custkey
AND lo_suppkey = s_suppkey
AND lo_partkey = p_partkey
AND lo_orderdate = d_datekey
AND c_region = 'AMERICA'
AND s_nation = 'UNITED STATES'
AND (d_year = 1997 OR d_year = 1998)
AND p_category = 'MFGR#14'
```

C | TPC-H queries and additional results

Query 2

```
SELECT p_partkey, ps_partkey, ps_suppkey, s_suppkey, s_nationkey,
   n_nationkey, r_regionkey
FROM part, supplier, partsupp, nation, region
WHERE p_partkey = ps_partkey
AND s_suppkey = ps_suppkey
AND p_size = 15
AND p_type like '%%BRASS'
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'EUROPE'
```

Query 3

```
SELECT c_custkey, o_custkey, o_orderkey, l_orderkey
FROM customer, orders, lineitem
WHERE c_mktsegment = 'BUILDING'
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate < '1995-03-15'
AND l_shipdate > '1995-03-15'
```

Query 5

```
SELECT c_custkey, o_custkey, l_orderkey, o_orderkey, l_suppkey, s_suppkey,
   s_nationkey, n_natonkey, r_regionkey
FROM customer, orders, lineitem, supplier, nation, region
WHERE c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND l_suppkey = s_suppkey
AND c_nationkey = s_nationkey
AND s_nationkey = n_nationkey
AND n_regionkey = r_regionkey
AND r_name = 'ASIA'
AND o_orderdate >= '1994-01-01'
AND o_orderdate < DATEADD(YY, 1, CAST('1994-01-01' AS DATE))
```

Query 7

```
SELECT s_suppkey, l_suppkey, o_orderkey, l_orderkey, c_custkey, o_custkey,
   s_nationkey, c_nationkey, n1.n_nationkey, n2.n_nationkey
FROM supplier, lineitem, orders, customer, nation n1, nation n2
WHERE s_suppkey = l_suppkey
AND o_orderkey = l_orderkey
AND c_custkey = o_custkey
AND s_nationkey = n1.n_nationkey
AND c_nationkey = n2.n_nationkey
AND (n1.n_name = 'FRANCE' AND n2.n_name = 'GERMANY')
AND l_shipdate BETWEEN '1995-01-01' AND '1996-12-31'
```

Query 8

```
SELECT p_partkey, l_partkey, s_suppkey, l_suppkey, l_orderkey, o_orderkey,
   c_nationkey, n1.n_nationkey, r_regionkey, s_nationkey, n2.n_nationkey
FROM part, supplier, lineitem, orders, customer, nation n1, nation n2, region
WHERE p_partkey = l_partkey
AND s_suppkey = l_suppkey
AND l_orderkey = o_orderkey
AND o_custkey = c_custkey
AND c_nationkey = n1.n_nationkey
AND n1.n_regionkey = r_regionkey
AND r_name = 'AMERICA'
AND s_nationkey = n2.n_nationkey
AND o_orderdate BETWEEN '1995-01-01' AND '1996-12-31'
AND p_type = 'ECONOMY ANODIZED STEEL'
```

Query 9

```
SELECT p_partkey, s_suppkey, l_partkey, l_suppkey, ps_suppkey, ps_partkey,
   o_orderkey, l_orderkey, s_nationkey, n_nationkey
FROM part, supplier, lineitem, partsupp, orders, nation
WHERE s_suppkey = l_suppkey
AND ps_suppkey = l_suppkey
AND ps_partkey = l_partkey
AND p_partkey = l_partkey
AND o_orderkey = l_orderkey
AND s_nationkey = n_nationkey
AND p_name LIKE '%%green%%'
```

Query 10

```
SELECT c_custkey, o_custkey, l_orderkey, o_orderkey, c_nationkey, n_nationkey
FROM customer, orders, lineitem, nation
WHERE c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate >= '1993-10-01'
AND o_orderdate < DATEADD(mm, 3, CAST('1993-10-01' AS DATE))
AND l_returnflag = 'R'
AND c_nationkey = n_nationkey
```

Query 11

```
SELECT ps_suppkey, s_suppkey, s_nationkey, n_nationkey
FROM partsupp, supplier, nation
WHERE ps_suppkey = s_suppkey
AND s_nationkey = n_nationkey
AND n_name = 'GERMANY'
```

Query 12

```
SELECT o_orderkey, l_orderkey
FROM orders, lineitem
WHERE o_orderkey = l_orderkey
AND l_shipmode IN ('MAIL','SHIP')
AND l_receiptdate >= '1994-01-01'
AND l_receiptdate < DATEADD(mm, 1, CAST('1995-09-01' AS DATE))
```

Query 14

```
SELECT l_partkey, p_partkey
FROM lineitem, part
WHERE l_partkey = p_partkey
AND l_shipdate >= '1995-09-01'
AND l_shipdate < DATEADD(mm, 1, '1995-09-01')
```

Query 16

```
SELECT p_patkey, ps_partkey
FROM partsupp, part
WHERE p_partkey = ps_partkey
AND p_brand <> 'Brand#45'
AND p_type NOT LIKE 'MEDIUM POLISHED%%'
AND p_size IN (49, 14, 23, 45, 19, 3, 36, 9)
AND ps_suppkey NOT IN (SELECT s_suppkey FROM supplier WHERE s_comment LIKE
   '%%Customer%%Complaints%%')
```

	Index seeks (k)			
	ZigZag merge join	ZigZag merge (NH)	Minesweeper	Index nested loops
Q2	205.74	804.11	4743.86	718.17
Q3	13,467.55	87,372.16	468,746.60	147,883.10
Q5	20,128.02	52,088.87	465,433.20	93,880.25
Q7	7777.67	26,372.42	172,227.30	22,566.32
Q8	3235.87	7739.43	84,733.72	5948.07
Q9	19,060.24	19,802.29	338,226.80	43,191.20
Q10	5159.14	8176.83	96,827.14	89,477.90
Q11	17.34	20.11	157.68	20.11
Q12	20,958.81	22,685.78	272,160.70	402,302.10
Q14	443.02	3719.41	29,572.05	303,748.00
Q16	0.03	0.06	0.27	14,800.80

Table C.1: Number of index seeks needed to answer TPC-H queries at scale factor 50.

	Tuples read (k)			
	ZigZag merge	Minesweeper	Hybrid hash	Index nested loops
Q2	1458.53	9822.22	40,700.28	1468.44
Q3	109,841.90	863,734.80	376,508.40	297,262.90
Q5	164,521.10	1,043,033.00	382,506.20	233,286.10
Q7	41,284.01	339,855.10	383,006.60	45,278.66
Q8	18,949.01	155,656.70	382,573.90	12,510.00
Q9	106,860.90	815,736.20	386,049.80	102,705.90
Q10	52,044.47	222,591.50	156,526.50	184,747.00
Q11	1668.64	2004.10	40,500.00	1648.53
Q12	181,804.80	669,284.70	375,006.50	827,335.20
Q14	22,430.34	77,483.95	310,005.80	611,238.20
Q16	0.08	0.42	50,500.04	29,601.58

Table C.2: Number of base data tuples read to answer TPC-H queries at scale factor 50.

Bibliography

[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. "Database tuning advisor for Microsoft SQL Server 2005: demo." In: *SIGMOD*. 2005, pp. 930–932.

[2] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. "Weaving Relations for Cache Performance." In: *VLDB*. 2001, pp. 169–180.

[3] Vo Ngoc Anh and Alistair Moffat. "Index compression using 64-bit words." In: *Software: Practice and Experience* 40.2 (2010), pp. 131–147.

[4] Vo Ngoc Anh and Alistair Moffat. "Inverted Index Compression Using Word-Aligned Binary Codes." In: *Information Retrieval* 8.1 (2005), pp. 151–166.

[5] Vo Ngoc Anh and Alistair Moffat. "Structured Index Organizations for High-Throughput Text Querying." In: *SPIRE*. 2006, pp. 304–315.

[6] Gennady Antoshenkov. "Byte-aligned bitmap compression." In: *DCC*. 1995, p. 476.

[7] Gennady Antoshenkov and Mohamed Ziauddin. "Query processing and optimization in Oracle Rdb." In: *VLDB* 5.4 (1996), pp. 229–237.

[8] Ricardo Baeza-Yates. "A Fast Set Intersection Algorithm for Sorted Sequences." In: *Combinatorial Pattern Matching*. 2004, pp. 400–408.

[9] Ricardo Baeza-Yates and Alejandro Salinger. "Experimental analysis of a fast intersection algorithm for sorted sequences." In: *SPIRE*. 2005, pp. 13–24.

[10] Akanksha Baid, Ian Rae, Jiexing Li, AnHai Doan, and Jeffrey Naughton. "Toward Scalable Keyword Search over Relational Data." In: *VLDB* 3.1 (2010), pp. 140–149.

[11] Jérémy Barbay and Claire Kenyon. "Adaptive intersection and t-threshold problems." In: *SIAM*. 2002, pp. 390–399.

[12] Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. "Faster adaptive set intersections for text searching." In: *Experimental Algorithms*. 2006, pp. 146–157.

[13] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger. "An Experimental Investigation of Set Intersection Algorithms for Text Searching." In: *ACM JEA* 14.3.7 (2009), pp. 1–24.

[14] D.A. Bell, D.H.O. Link, and S. McClean. "Pragmatic estimation of join sizes and attribute correlations." In: *ICDE*. 1989, pp. 76–84.

[15] Jon Louis Bentley and Andrew Chi-Chih Yao. "An almost optimal algorithm for unbounded searching." In: *Information processing letters* 5.3 (1976), pp. 82–87.

[16] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. "Keyword Searching and Browsing in Databases using BANKS." In: *ICDE*. 2002, pp. 431–440.

[17] Andrzej Białecki, Robert Muir, and Grant Ingersoll. "Apache Lucene 4." In: *SIGIR Workshop on Open Source Information Retrieval*. 2012, pp. 17–24.

[18] Horst Biller. "On the Architecture of a System Integrating Data Base Management and Information Retrieval." In: *SIGIR*. 1982, pp. 80–97.

[19] Truls A Bjørklund, Johannes Gehrke, and Øystein Torbjørnsen. "A Confluence of Column Stores and Search Engines: Opportunities and Challenges." In: *USETIM*. 2009.

[20] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. "MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine." In: *SIGMOD*. 2006, pp. 479–490.

[21] Eric A. Brewer. "Combining Systems and Databases: A Search Engine Retrospective." In: *Readings in Database Systems*. Ed. by Joseph M. Hellerstein and Michael Stonebraker. 4th ed. 2005.

[22] Yu Cao, Yongluan Zhou, Chee-Yong Chan, and Kian-Lee Tan. "On optimizing relational self-joins." In: *EDBT*. 2012, pp. 120–131.

[23] Chee-Yong Chan and Yannis E Ioannidis. "Bitmap index design and evaluation." In: *ACM SIGMOD Record* 27.2 (1998), pp. 355–366.

[24] Surajit Chaudhuri and Vivek Narasayya. "Self-tuning database systems: a decade of progress." In: *VLDB*. 2007, pp. 3–14.

[25] Surajit Chaudhuri, Raghu Ramakrishnan, and Gerhard Weikum. "Integrating DB and IR Technologies: What is the Sound of One Hand Clapping?" In: *CIDR*. 2005.

[26] Roberto Cornacchia, Sándor Héman, Marcin Zukowski, Arjen P de Vries, and Peter Boncz. "Flexible and efficient IR using array databases." In: *VLDB* 17.1 (2008), pp. 151–168.

[27] J. Shane Culpepper and Alistair Moffat. “Efficient set intersection for inverted indexing.” In: *ACM TOIS* 29.1 (2010), pp. 1–25.

[28] Benoit Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zait, and Mohamed Ziauddin. “Automatic SQL tuning in Oracle 10g.” In: *VLDB*. 2004, pp. 1098–1109.

[29] Samuel DeFazio, Amjad Daoud, Lisa Ann Smith, and Jagannathan Srinivasan. “Integrating IR and RDBMS Using Cooperative Indexing.” In: *SIGIR*. 1995, pp. 84–92.

[30] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. “Adaptive Set Intersections, Unions, and Differences.” In: *SODA*. 2000, pp. 743–752.

[31] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. “Experiments on adaptive set intersections for text retrieval systems.” In: *Algorithm Engineering and Experimentation*. 2001, pp. 91–104.

[32] Paul Dixon. “Basics of Oracle Text Retrieval.” In: *IEEE Computer Society Data Engineering Bulletin* 24.4 (2001): *Special Issue on Text and Databases*, pp. 11–14.

[33] Vuk Ercegovac, David J. DeWitt, and Raghu Ramakrishnan. “The TEXTURE Benchmark: Measuring Performance of Text Queries on a Relational DBMS.” In: *VLDB*. 2005, pp. 313–324.

[34] C. Estan and J.F. Naughton. “End-biased Samples for Join Cardinality Estimation.” In: *ICDE*. 2006.

[35] Filippo Furfaro, Giuseppe M Mazzeo, Domenico Saccà, and Cristina Sirangelo. “Hierarchical binary histograms for summarizing multi-dimensional data.” In: *SAC*. 2005, pp. 598–603.

[36] Cesar A Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, and Shin Zhang. “Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server.” In: *ICDE*. 2008, pp. 1190–1199.

[37] Carlos Garcia-Alvarado and Carlos Ordonez. “Information Retrieval from Digital Libraries in SQL.” In: *WIDM*. 2008, pp. 55–62.

[38] Ram D. Gopal, Ram Ramesh, and Stanley Zionts. “Criss-cross hash joins: design and analysis.” In: *ICDE TKDE* 13.4 (2001), pp. 637–653.

[39] Torsten Grabs, Klemens Böhm, and Hans-Jörg Schek. “PowerDB-IR: Information Retrieval on Top of a Database Cluster.” In: *CIKM*. 2001, pp. 411–418.

[40] Goetz Graefe. “Executing Nested Queries.” In: *BTW*. Vol. 26. 2003, pp. 58–77.

[41] Goetz Graefe. “New algorithms for join and grouping operations.” In: *Computer Science Research and Development* 27.1 (2012), pp. 3–27.

[42] Goetz Graefe. “Query Evaluation Techniques for Large Databases.” In: *ACM CSUR* 25.2 (1993), pp. 73–169.

[43] Goetz Graefe. “Volcano—An Extensible and Parallel Query Evaluation System.” In: *IEEE TKDE* 6.1 (1994), pp. 120–135.

[44] Luis Gravano, ed. *IEEE Computer Society Data Engineering Bulletin* 24.4 (2001): *Special Issue on Text and Databases*.

[45] William A. Greene. “k-way merging and k-ary sorts.” In: *ACM Southeast Conference*. 1993, pp. 127–135.

[46] David A. Grossman, Ophir Frieder, David O. Holmes, and David C. Roberts. “Integrating Structured Data and Text: A relational approach.” In: *JASIST* 48 (2 1997), pp. 122–132.

[47] Alan Halverson, Josef Burger, Leonidas Galanis, Ameet Kini, Rajasekar Krishnamurthy, Ajith Nagaraja Rao, Feng Tian, Stratis D Viglas, Yuan Wang, Jeffrey F Naughton, et al. “Mixed Mode XML Query Processing.” In: *VLDB*. 2003, pp. 225–236.

[48] James R. Hamilton and Tapas K. Nayak. “Microsoft SQL Server Full-Text Search.” In: *IEEE Computer Society Data Engineering Bulletin* 24.4 (2001): *Special Issue on Text and Databases*, pp. 7–10.

[49] Gilbert Held and Thomas Marshall. *Data Compression; Techniques and Applications: Hardware and Software Considerations*. John Wiley & Sons, Inc., 1991.

[50] Sven Helmer, Till Westmann, and Guido Moerkotte. “Diag-Join: An Opportunistic Join Algorithm for 1:N Relationships.” In: *VLDB*. 1998, pp. 98–109.

[51] S. Héman, M. Zukowski, A. de Vries, and P. Boncz. “Efficient and Flexible Information Retrieval Using MonetDB/X100.” In: *CIDR*. 2007, pp. 96–101.

[52] David Holmes. “SQL Text Parsing for Information Retrieval.” In: *CIKM*. 2003, pp. 496–499.

[53] Vagelis Hristidis and Yannis Papakonstantinou. “DISCOVER: Keyword Search in Relational Databases.” In: *VLDB*. 2002, pp. 670–681.

[54] Yannis Ioannidis. “The history of histograms (abridged).” In: *VLDB*. 2003, pp. 19–30.

[55] HV Jagadish, Nick Koudas, S Muthukrishnan, Viswanath Poosala, Kenneth C Sevcik, and Torsten Suel. “Optimal histograms with quality guarantees.” In: *VLDB*. Vol. 98. 1998, pp. 24–27.

[56] Mohan Kamath and Krithi Ramamritham. *Bucket Skip Merge Join: A Scalable Algorithm for Join Processing in Very Large Databases using Indexes*. Tech. rep. 20. Amherst, MA: University of Massachusetts, 1996.

[57] Helmut Kaufmann and Hans-Jörg Schek. "Text Search Using Database Systems Revisited: Some Experiments." In: *BNCOD*. BNCOD '95. Springer, 1995, pp. 204–225.

[58] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. "The Vertica Analytic Database: C-Store 7 Years Later." In: *PVLDB* 5.12 (2012), pp. 1790–1801.

[59] Per-Åke Larson, Cipri Clinciu, Eric N Hanson, Artem Oks, Susan L Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. "SQL Server Column Store Indexes." In: *SIGMOD*. 2011, pp. 1177–1184.

[60] Per-Åke Larson, Eric N Hanson, and Susan L Price. "Columnar Storage in SQL Server 2012." In: *IEEE Computer Society Data Engineering Bulletin* 35.1 (2012), pp. 15–20.

[61] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. "Cardinality Estimation Using Sample Views with Quality Assurance." In: *SIGMOD*. 2007, pp. 175–186.

[62] Daniel Lemire and Leonid Boytsov. "Decoding billions of integers per second through vectorization." In: *Software: Practice and Experience* (2013).

[63] Quanzhong Li, Minglong Shao, Volker Markl, Kevin Beyer, Latha Colby, and Guy Lohman. "Adaptively Reordering Joins during Query Execution." In: *ICDE*. 2007, pp. 26–35.

[64] Albert Maier and David Simmen. "DB2 Optimization in Support of Full Text Search." In: *IEEE Computer Society Data Engineering Bulletin* 24.4 (2001): *Special Issue on Text and Databases*, pp. 3–6.

[65] Michael V. Mannino, Paicheng Chu, and Thomas Sager. "Statistical Profile Estimation in Database Systems." In: *ACM CSUR* 20.3 (1988), pp. 191–221.

[66] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. "Robust Query Processing Through Progressive Optimization." In: *SIGMOD*. 2004, pp. 659–670.

[67] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. "Wavelet-based histograms for selectivity estimation." In: *ACM SIGMOD Record*. Vol. 27. 2. 1998, pp. 448–459.

[68] M. Catherine McCabe, David O. Holmes, David A. Grossman, and Ophir Frieder. "Parallel, Platform-Independent Implementation of Information Retrieval Algorithms." In: *PDPTA*. 2000.

[69] Sergey Melnik and Hector Garcia-Molina. "Adaptive Algorithms for Set Containment Joins." In: *ACM TODS* 28.1 (2003), pp. 56–99.

[70] Microsoft Corporation. *Microsoft Minesweeper*. 2012. URL: http://apps.microsoft.com/windows/en-us/app/microsoft-minesweeper/45ac18d7-e742-494f-a1b1-009aa412a179 (visited on 02/05/2014).

[71] Tommaso Mostardi. "Estimating the size of relational SPθJ operation results: an analytical approach." In: *Information Systems* 15.5 (1990), pp. 591–601.

[72] James K. Mullin. "Estimating the size of a relational join." In: *Information Systems* 18.3 (1993), pp. 189–196.

[73] Hung Q Ngo, Dung T Nguyen, Christopher Ré, and Atri Rudra. "Removing the Haystack to Find the Needle(s): Minesweeper, an Adaptive Join Algorithm." In: (2013). arXiv: 1302.0914.

[74] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. "Worst-case Optimal Join Algorithms." In: *PODS*. 2012, pp. 37–48. ISBN: 978-1-4503-1248-6.

[75] Hung Q Ngo, Christopher Re, and Atri Rudra. "Skew Strikes Back: New Developments in the Theory of Join Algorithms." In: *ACM SIGMOD Record*. Vol. 42. 4. 2013, pp. 5–16.

[76] Pat O'Neil, Betty O'Neil, and Xuedong Chen. *Star Schema Benchmark*. 2009. URL: http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[77] Gregory Piatetsky-Shapiro and Charles Connell. "Accurate Estimation of the Number of Tuples Satisfying a Condition." In: *ACM SIGMOD Record*. Vol. 14. 2. 1984, pp. 256–276.

[78] Viswanath Poosala and Yannis E Ioannidis. "Selectivity estimation without the attribute value independence assumption." In: *VLDB*. Vol. 97. 1997, pp. 486–495.

[79] Steve Putz. *Using a Relational Database for an Inverted Text Index*. Tech. rep. P91-00158. Xerox PARC, Jan. 1991.

[80] Tilmann Rabl, Meikel Poess, Hans-Arno Jacobsen, Patrick O'Neil, and Elizabeth O'Neil. "Variations of the star schema benchmark to test the effects of data skew on query performance." In: *ICPE*. 2013, pp. 361–372.

[81] Vijayshankar Raman, Lin Qiao, Wei Han, Inderpal S. Narang, Ying-Lin Chen, Kou-Horng Yang, and Fen-Lin Ling. "Lazy, Adaptive RID-List Intersection, and Its Application To Index Anding." In: *SIGMOD*. 2007, pp. 773–784.

[82] Karthikeyan Ramasamy, Jignesh M Patel, Jeffrey F Naughton, and Raghav Kaushik. "Set Containment Joins: The Good, The Bad and The Ugly." In: *VLDB*. 2000, pp. 351–362.

[83] S.E. Robertson, S. Walker, M. Beaulieu, and Peter Willett. "Okapi at TREC-7: automatic ad hoc, filtering, VLC and interactive track." In: *TREC*. Vol. 21. 1999, pp. 253–264.

[84] Roger Ford. *Oracle Text*. 2007. URL: http://www.oracle.com/technetwork/database/enterprise-edition/11goracletexttwp-133192.pdf (visited on 01/26/2013).

[85] Peter Sanders and Frederik Transier. "Intersection in Integer Inverted Indices." In: *ALENEX*. Vol. 7. 2007, pp. 71–83.

[86] C. Shaoul and C. Westbury. *A USENET corpus (2005–2010)*. Edmonton, AB: University of Alberta. 2011. URL: http://www.psych.ualberta.ca/~westburylab/downloads/usenetcorpus.download.html.

[87] Michał Stabno and Robert Wrembel. "RLH: Bitmap compression technique based on run-length and Huffman encoding." In: *Information Systems* 34.4 (2009), pp. 400–414.

[88] *Statistics Used by the Query Optimizer in Microsoft SQL Server 2008*. 2009. URL: http://technet.microsoft.com/en-us/library/dd535534(v=sql.100).aspx (visited on 01/22/2014).

[89] Michael Stonebraker, Heidi Stettner, Nadene Lynn, Joseph Kalash, and Antonin Guttman. "Document processing in a relational database system." In: *ACM Trans. Inf. Syst.* 1 (2 Apr. 1983), pp. 143–158. ISSN: 1046-8188.

[90] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. "C-Store: A Column-oriented DBMS." In: *VLDB*. 2005, pp. 553–564.

[91] Arun Swami and K.Bernhard Schiefer. "On the estimation of join result sizes." In: *EDBT*. Vol. 779. 1994, pp. 287–300.

[92] Transaction Processing Performance Council. *TPC-H*. 2014. URL: http://www.tpc.org/tpch/ (visited on 01/22/2014).

[93] Transaction Processing Performance Council. *TPC-H benchmark specification*. 2008. URL: http://www.tpc.org/tpch/.

[94] Howard Turtle and James Flood. "Query evaluation: Strategies and optimizations." In: *Information Processing & Management* 31.6 (1995), pp. 831–850.

[95] Todd L Veldhuizen. "Leapfrog Triejoin: a worst-case optimal join algorithm." In: (2013). arXiv: 1210.0481.

[96] Gerhard Weikum. "DB&IR: Both Sides Now." In: *SIGMOD*. 2007, pp. 25–30.

[97] Kyu-Young Whang, Min-Jae Lee, Jae-Gil Lee, Min-Soo Kim, and Wook-Shin Han. "Odysseus: a High-Performance ORDBMS Tightly-Coupled with IR Features." In: *ICDE*. IEEE. 2005, pp. 1104–1005.

[98] Hugh E. Williams and Justin Zobel. “Compressing Integers for Fast File Access.” In: *The Computer Journal* 42.3 (1999), pp. 193–201.

[99] Joel L Wolf, Daniel M Dias, and Philip S Yu. “An Effective Algorithm for Parallelizing Sort Merge Joins in the Presence of Data Skew.” In: *DPDS*. 1990, pp. 103–115.

[100] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. “Optimizing Bitmap Indices with Efficient Compression.” In: *ACM TODS* 31.1 (2006), pp. 1–38.

[101] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. “On Supporting Containment Queries in Relational Database Management Systems.” In: *SIGMOD*. 2001, pp. 425–436.

[102] Justin Zobel and Alistair Moffat. “Inverted Files for Text Search Engines.” In: *ACM CSUR* 38.2 (2006), pp. 1–56.

[103] Marcin Zukowski, Sandor Héman, Niels Nes, and Peter Boncz. “Super-Scalar RAM-CPU Cache Compression.” In: *ICDE*. 2006.