Surprising Interactions Between Filters In Equi-Join Processing

Mihir Khare

CMU-CS-25-152

December 2025

Computer Science Department
School of Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jignesh Patel, Chair
Andrew Pavlo

Submitted in partial fulfillment of the requirements for the Master's degree in Computer Science.

Keywords: database management systems, join pre-filtering, robust query processing, adaptive query processing, bloom filters

Abstract

The current era of data storage is defined by the widespread adoption of data lakes, and the disaggregation of storage and compute hardware. Modern database management systems (DBMSs) are often operating on large volumes of data stored in object stores (like Amazon's S3), open file formats (like Apache's Parquet), or otherwise have outdated or nonexistent statistics. In join-heavy analytical workloads, the traditional approach of optimizing query plans to minimize the cost of joins breaks down if the available information to estimate cardinalities and costs is inaccurate. In recent years, a class of techniques known as "join pre-filtering" has gained focus as an attempt to reduce the reliance on a good optimizer for minimizing join costs by reducing the inputs of joins to the minimum set of tuples needed to produce the output. This thesis explores the current state-of-the-art in pre-filtering, and concludes that more work must be done to create a strategy that is applicable across a wide range of workloads. First, we provide an overview of the fundamental concepts of pre-filtering, and describe the key design decisions of and differences between currently studied techniques. We then evaluate two pre-filtering methods, min-max filters and Bloom filters, using a modified implementation of Dynamic Predicate Transfer (RPT+), a leading contemporary technique for join pre-filtering. Our analysis focuses on the performance interactions between these filter types. Finally, we discuss the implications of the results for modern DBMSs, and future directions of study in this area.

Acknowledgments

First, I thank my advisor, Jignesh Patel, whose continuous support made this thesis possible. He, along with Andy Pavlo, instilled in me a love for databases and the necessary knowledge to understand how modern systems function and interact.

I thank Peter Boncz for bringing the RPT+ work to our attention in the summer of 2025, as well as the authors of RPT+, in particular the key GitHub contributor, Yiming Qiao, for making their implementation publicly available.

I also thank Angy Malloy, Peter Steenkiste, and again Andy Pavlo for their work in running the 5th Year Master's program.

Finally, I thank all of the members of the Carnegie Mellon University Database Group who have provided assistance, feedback, and friendship during the thesis process, especially Chris Laspias, Yuchen Liang and Wan Shen Lim.

Contents

Figures

Chapter 1

Introduction

Database management systems (DBMSs) are a core component of modern computing. Essentially all software will somehow interact with or manage a database, from media streaming, to banking, to social media, and beyond. One common function of many databases is to support performant and efficient data analysis queries. Such analytical workloads consist of queries that may scan from multiple tables, join tables together according to some conditions, filter out individual tuples based on some set of constraints, aggregate information across tuples, or more – often many of each of these operations happen within a single query!

Joining two tables into a single output, where the tuples it contains are some combination of tuples from both inputs, is a fundamental operation for any analytical DBMS. However, it is also slower and more complex to execute than most other operations — many workloads will spend significant portions of their runtime simply processing joins. Some join execution strategies (e.g., hash joins) also incur a large memory overhead for intermediate data structures. Due to these factors, improving the performance of joins can often substantially improve the overall performance of the query.

One main contributor to join performance is the order in which joins are evaluated. More concretely, two different join orders can have vastly different intermediate join results compared

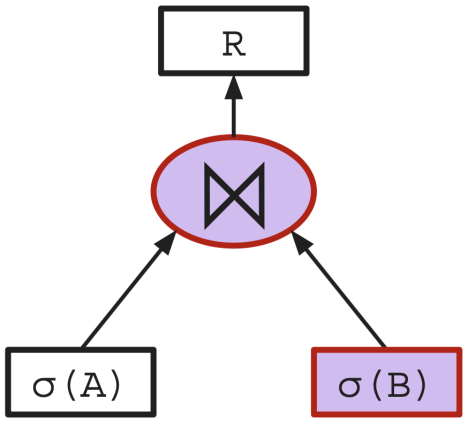

Figure 1.1: A query plan containing a single join on two filtered tabled, `A` and `B`.

to the final result after all joins. A poor join order could incur large performance penalties to process these extra tuples, only to result in the same output as a better join order.

For decades, the standard approach to solving this problem has been **join order optimization**, which is a subset of query optimization that focuses on determining the best order and strategies to execute joins in a query. It makes use of pre-computed table statistics, cardinality estimates, and many heuristics in order to create a join execution plan. While optimzation is known to be an NP-complete problem, and it is infeasible to fully explore all plans for queries with a large number of joins, a good optimizer generally can do pretty well! As optimizer architecture and estimation quality has improved over the past decades, essentially every production database now involves some amount of join order optimization.

Traditionally, analytical SQL databases have stored and managed data with strict schemas and structures, often with data ingestion pipelines that transform raw data into a cleaner version that the database can use. However, the past decade has seen many significant shifts in how databases are organized and used. As storage and compute hardware continue to be further disaggregated, many workloads now make heavy use of object storage for unstructured data where tables are often stored in open file formats that are not managed by the DBMS itself. In many cases, this leaves the query optimizer with little information to base decisions on, while the expectations of performant and consistent query execution still remain high.

1.1 Performant Joins in a Statistics-Light World

With such changes in the landscape of analytical databases, there is a renewed focus on how to performantly execute complex queries without relying on accurate statistics and estimates. If we are unable to create an optimal query plan through optimization, the problem must be solved either before or after. Here, we detail three types of approaches that address this issue in different ways.

1.1.1 Query Reoptimization

One idea is that if we determine during runtime that a join order is incorrect, the query optimizer may be able to produce a better plan given the new, more accurate information we have after running a portion of the query. Work in this area has historically involved adding checkpoints to the initial query plan, where key statistics are collected and used to inform whether the optimizer should reoptimize the remainder of the query [5].

Such methods generally operate on the physical plan constructed by the initial optimization or the previous reoptimization. However, this can cause problems: the initial plan may be extremely different from the optimal, and choosing where checkpoints should be set based on the physical plan may not be the correct strategy. Zhao et. al. have recently developed a reoptimization technique that uses the logical plan and join graph, rather than the physical plan and heuristics on the join types [20]. They identify subqueries within the logical plan based on primary/foreign key relationships within joins, and then check whether or not to reoptimize between each subquery. This technique addresses both of the described issues by making smarter choices of subqueries

to execute, and ensuring that reoptimization overhead is not incurred unnecessarily while still allowing for sufficient ways to near optimality.

1.1.2 Plan Hints and External Optimization

Another idea is that the client may know more about how to optimally execute their queries than the optimizer, and therefore the client should be able to directly influence the optimizer in certain cases. Traditionally, this has manifested itself through "hints" provided by the client within the query text that are treated as optimizer directives. Various production databases, such as Oracle, have supported this and related features for many decades, and allow clients to provide direction to all levels of the optimizer [11].

In recent years, there has been a growth in techniques that use automated learning models to write plan hints, rather than clients doing it manually. This, in effect, works as an external optimizer that works at a higher level than traditional single-query optimizers (e.g., optimizing using data collected over multiiple runs). Techniques of this type, such as HERO and COOOL, have good results when comparing performance to their baselines [23] [17].

Other recent work retains the human component, but aims to better assist with improving the quality of plan hints. HintQPT, for example, provides an interface for clients to visualize the impact of different plan hints, and select their preferred option, with a focus on being robust to inaccuracies in cardinality or cost estimation [16].

1.1.3 Join Pre-Filtering

A third idea is that if we can shrink the cardinalities of all join inputs so that they do less work, then perhaps the chosen join order does not matter. This class of techniques is known as **join pre-filtering**, and is the focus of the remainder of this thesis. Formally, we define join pre-filtering as a set of methods that reduce the input relations of joins towards the minimum set of tuples necessary to produce the correct output. In the next two sections, we will provide a classification of filters used for pre-filtering, as well as a description of some of the foundational techniques in this area.

1.2 Thesis Contributions

In this thesis, we first introduce a novel categorization of filter types, and provide a thorough characterization of modern join-prefiltering techniques. Then, we introduce an extension to Dynamic Predicate Transfer, a leading contemporary join pre-filtering implementation, allowing for both independent and combined use of the two filter types it supports. Finally, we evaluate all four combinations of filter configurations on standard benchmarks, specifically highlighting the surprising interactions and how this depends on workload and query specifics. After this analysis, we detail potential future directions for join pre-filtering based on the observations made in this thesis.

Chapter 2

Characterization of Filters

Join pre-filtering has been implemented in many different fashions over the history of query execution. All pre-filtering strategies, however, follow the same three principles:

1. All pre-filtering strategies **must** not filter out tuples needed for the final result.
2. Good pre-filtering strategies shall reduce the overall runtime of the query.
3. Good pre-filtering strategies shall reduce the overall memory usage of the query.

The first of these is a strict requirement, while the second and third are behaviors that should in practice occur most of the time. Following these basic principles, pre-filtering strategies make use of different types of filters and fall into different categories of algorithms.

2.1 Types of Filters

At a fundamental level, the filters used in pre-filtering are very similar to those used to process required filter clauses written in queries. The biggest differences are that pre-filters are created at runtime based on the data being processed, rather than statically by the client, and that if a pre-filter is not enforced the query result will still be correct. This second difference means that, unlike with required/static filters, pre-filters can make use of both exact and approximate data structures.

2.1.1 Exact Filters

Exact filters will ensure that the output is as minimal as possible, but in general they will be expensive to create and/or apply. In certain cases, where the accepted set is fairly small (DuckDB, a leading analytical DBMS, has a default threshold of 50 [13]), an `IN` predicate can be used. Another case of interest is when the range of accepted values is small, where a concise bitarray can represent the accepted set and allow for very efficient probing.

In general, an exact filter will be a traditional set-like data structure, needing to track all unique values accepted by the filter. Such data structures are often expensive to create and apply, and requiring memory that is linear in terms of the number of unique values present. This means that the benefits of pre-filtering with an exact filter can be fairly limited, with much of the

improvement occuring through indirect cascading effects of pre-filtering on later portions of the query, or through better cache/memory utilization of the smaller data structure.

2.1.2 Approximate Filters

Approximate filters are generally cheaper and smaller than exact filters, with the tradeoff being generally lower effectiveness. However, the intention is that the savings outweigh the extra runtime caused by letting extra tuples into the joins. While no pre-filter should reject tuples that are needed for the final join result, it is particularly important for approximate filters that the strategy used does not allow any false negatives. There are a variety of approximations that are possible, but they roughly fall into two categories: **uniform** and **non-uniform**.

If an approximate filter is uniform, then any true negative probe will have a roughly equal chance of being a false positive. The classic example of this is a **Bloom filter**, which is a hash-based set-membership data structure [2]. Both the insert and probe methods involve applying one or more hash functions to the data, and then setting or checking the relevant bits in the array respectively. When probing, if all of the relevant bits had been set, the Bloom filter outputs a "yes". This results in false positives in the case of enough hash collisions, but with proper choices of hash function these are roughly evenly distributed across the input space. The expected false positive rated is also tunable based on the number of hash functions and amount of memory used, and generally is a low single-digit percentage (e.g., Apache Arrow has a default rate of 2% [1]).

If it is non-uniform, then some sets of true negatives will have higher chances of being false positives than others. There are many different ways to create such a filter, but the simplest kind are range-based filters, most commonly in the form of **min/max filters**. These filters are created by tracking the minimum and maximum values of the relevant data, and then when applying the filter, all values within the range are accepted and all other values are rejected. This results in a very cheap and selective filter if much of the data it is applied on is outside of the range, but if the data is largely within the range, or if the range is extremely large, then it may not be very effective.

Within these categories, different strategies will still be better or worse in terms of expected false positive rate, memory footprint, cache access pattern, creation and probe costs, actual false positive rate, and more. An important observation is that the false positive distribution for uniform approximate filters is very different from that of non-uniform filters, and even two non-uniform filters may have very different distributions. Therefore, it may be useful for a single pre-filtering strategy to use multiple approximate filters in order to get close to an exact filter while still being cheaper and lighter.

Chapter 3

Overview of Pre-Filtering Techniques

Over the decades of research in query processing, many pre-filtering techniques have been developed, all presenting different ideas for how to improve execution time by reducing the input size of joins. Here, we detail the two most common categories of pre-filtering in use today, and describe a selection of fundamental work in these categories.

3.1 Probe-Side Pre-Filtering

One of the most common methods of processing equi-joins is as a **hash join**, where one input is used to construct a hash table that later gets probed by the other table to determine which tupes from each side should be combined to produce an output tuple. This process can be quite efficient in many scenarios as it poses no constraints on the shape or order of the inputs, but the process of building and probing the hash table may be quite expensive. Especially if many tuples present in the inputs are ultimately not used for the output of the join, or if there happen to be many hash collisions, we end up performing lots of unnecessary work on the probe side and take up more of the limited memory capacity on the build side. This makes hash joins a good target for pre-filtering, as the potential savinhs can be quite high in many scenarios.

With hash joins, an observation is that one of the inputs is already going to be involved in building a data structure, and therefore it is relatively cheap to build another one at the same time. This insight is the key behind **probe-side pre-filtering**, a class of pre-filtering techniques that applies to hash joins where filters are constructed using the build input for the join, and are probed prior to the hash table probes. This doesn't necessarily mean that these pre-filters are probed immediately before probing the hash table (e.g., within the hash join operator itself), but rather that the probes happen somewhere in the query plan rooted at the probe side of the join.

Probe-side pre-filtering techniques are especially interesting because they are all examples of **single-pass** techniques, meaning that only one pass over the table data is needed to pre-filter the data. Specifically, this means that there is no extra scanning needed to pre-filter compared to just executing the base query itself. This is especially important when tables are too large to fit in memory, or when memory capacity is limited, as keeping disk I/O costs at a minimum is vital for executing queries efficiently.

There have been many different probe-side pre-filtering strategies developed over time, and

we describe the most important here.

3.1.1 Bloom Joins

The most basic probe-side pre-filtering algorithm is the **Bloom join**, first proposed in 1986 by Mackert et. al. [9]. They describe using a Bloom filter as a "hashed semi-join", building the filter on the join column while building the hash table, and then probing the filter immediately before probing the hash table. While this may seem as if we are just trading one hash calculation when probing the table for a hash calculation when probing the filter, upon accounting for the cost of managing collisions within a hash table it becomes clear how a Bloom join may substantially increase performance if the pre-filter is highly selective.

In the paper, the authors simulate the performance of a Bloom join with a basic heuristic, and show that the performance when using a Bloom filter with a 30% false positive rate is significantly improved compared to the regular baseline, and a larger improvement is seen when a 0% false positive rate is simulated. It is important to note that this analysis is not comprehensive, and doesn't account for considerations such as memory usage of the filter, or detail performance in different data distributions or with different selectivities of the join condition.

This pre-filtering technique specifies that we use Bloom filters, although the algorithm to create and pass the filter can be used with any type of filter, exact or approximate.

3.1.2 Probe-Side Pre-Filter Pushdown

A fairly general rule of thumb in query optimization is that filters should be pushed down below joins whenever possible, as the cost of reducing the table before the join is very often less than the cost of joining first. We can apply this same idea to Bloom joins, where instead of filtering the probe input immediately before processing the join, we instead send the filter as far down as possible, and pre-filter at the `Scan` operator itself. This class of techniques had been in use in distributed query processing prior, but was first written about by Graefe in 1993 [4]. It limits the applicable settings slightly, as it requires that the join column is present in some base table, but this turns out to be fairly common. As with Bloom joins, this paper specifies that Bloom filters are used rather than the more expensive semi-join, but in practice any filter can be applied using this algorithm.

An interesting consequence of sending every pre-filter to its source is that in complex queries, a single table may recieve multiple pre-filters. This, in effect, means that we are maximally pre-filtering every table involved in a hash join probe prior to executing any of them. This can significantly increase the impact of pre-filtering, as discarding a single tuple here potentially prevents a large number of hash table probes down the line.

Bitvector-Aware Query Optimization

Since 2012, Microsoft has implemented this technique into SQLServer, with a few additions [3]. The first is that only a single filter will be created at each hash join operator, even if there are multiple conjuctive join predicates present. This presents an interesting scenario when a hash join's predicates reference more than one base table on the probe side, as it is not possible for this

single filter to be pushed down to any individual `Scan`. Instead, SQLServer will send the filter to the lowest possible point in the query where information from all referenced tables is present, which is generally either the `Scan` or some intermediate hash join operator. While this doesn't pre-filter tuples as early as it could, having a single filter incorporate multiple join conditions reduces the memory usage of the technique and still shows significant benefits.

The other major innovation stems from the realization that in many cases, naively adding pre-filtering to a query plan does not actually give you the optimal execution plan for that query. Just like with regular query selection filters, pre-filtering and especially pre-filter pushdown shrink the cardinalities of their inputs, and therefore may significantly affect what the optimal join order and execution strategies are. Without accounting for the effects of pre-filtering in the cost model, any DBMS that uses it may choose highly sub-optimal execution plans for queries. Microsoft implemented pre-filtering in SQLServer as a transformation rule in the optimizer, and detailed how it can be added to any Volcano- or Cascades-style optimizer. They show that for any star or snowflake schema query, the optimal plan when including pre-filtering can be computed in linear time, and can be extended to other query shapes by breaking it down into multiple connected snowflake plans.

Pre-Filter Pushdown in DuckDB

Newer versions of DuckDB include a version of pre-filter pushdown, but using min/max filters rather than Bloom filters [13]. The main reason for this is that min/max filters are extremely cheap to create and probe and require very little memory compared to other types of filters. Further, DuckDB implements an optimization known as **row group skipping** for range-based filters within `Scan` operators, which can avoid testing a condition entirely if a row group's (by default 2048 tuples) metadata says the group falls completely inside or completely outside of the filter range [13]. Both of these factors mean that we can often pre-filter significantly for very little overhead.

3.1.3 Lookahead Information Passing (LIP)

While a goal of pre-filtering algorithms is to increase performance of join processing, another goal is that the use of pre-filtering should minimize the effects of selecting a poor join order. Lookahead Information Passing (LIP) was developed by Zhu et. al. to answer this question and to present an optimal execution strategy for star schema queries [22]. In this setting, the optimal execution plan will always be a **left-deep** plan, where there is one table that is joined with multiple other tables in a row. In the star schema context, this will be the large fact table joining with multiple small dimension tables. Here, the only choice the optimizer must make is what order the dimension tables get probed in, which significantly affects the runtime of the query.

A key insight is that this scenario is the ideal case for pre-filter pushdown. Since only one table is probing all others, every pre-filter can be evaluated before any join occurs, meaning the number of probes that occurs is at its minimum. However, with multiple filters needing to be probed, if we do so in a poor order we can still have sub-optimal performance. The main innovation of LIP is to adapt the order of pre-filters based on selectivities at runtime, with the

assumption that all filter conditions are independent. The authors proved that under this independence assumption, left-deep star schema queries can be evaluated both optimally and **robustly** when using LIP. More formally, this means that the maximum normalized spread in execution cost across all join orders is typically narrower when using LIP than the minimum normalized spread without LIP.

In some cases, the performance with LIP is actually better than with any plan where LIP is not applied. This is again due to its adaptive reordering, which means that the filter order can optimally handle changes in the data distribution, rather than being stuck with a fixed, sub-optimal join order.

3.2 Both-Sides Pre-Filtering

While probe-side pre-filtering techniques can often be implemented without significant overhead, their primary focus is to reduce the amount of work done during the probe phases of hash joins. However, this is only half of the full problem — having unnecessary tuples on the build sides of joins also has a negative impact on both memory usage and overall performance. One example of a case when pre-filtering the build side is important is if the optimizer incorrectly assigns the build and probe sides of a hash join, but many more cases exist in more complex query plans.

The notion of a **full reduction** defines a complete solution to our problem, formally meaning that all input tables are reduced to only the set of tuples required to produce the correct join output. It is in general not possible to achieve this with simply probe-side pre-filtering, as not every tuple built into a hash table is necessarily required for the final result after all joins. This brings forward the class of techniques known as **both-sides pre-filtering**, which aim to get closer to a full reduction with varying levels of approximation.

3.2.1 Yannakakis Algorithm

The foundational work in this space was done by Yannakakis in his 1981 paper detailing various algorithms to improve join processing on acyclic queries [19]. The main algorithm given, now known as the **Yannakakis algorithm**, splits join processing into two phases: the reduction phase, and the join phase. In the reduction phase, the algorithm first creates a "join tree" from the query's join graph, and does two passes of semi-joins across each join edge in the tree: one "upwards" from the leaves to the (arbitrarily chosen) root, followed by one "downwards" from the now reduced root down to the leaves. At each step of each pass, the next semi-join is constructed after applying the filter from the previous step. Then, the join phase executes on all reduced tables by performing a join for every edge in this tree in a bottom-up fashion.

The reduction phase guarantees that all tables are fully reduced. After the upwards pass, a table is pre-filtered based on its children in the join tree, which ends with the root table being fully reduced. Then, the downwards pass supplies this reduction information back to all tables, ensuring that each table is given information about all other tables it is connected to in the join graph.

This algorithm is the first full-reduction algorithm developed, and allows for acyclic queries to be evaluated in linear time with respect to the final result size and the total input sizes. The

algorithm is also not specific to hash joins, and can be applied to any equi-join query. However, its structure essentially requires that the data is read twice, which can add a significant overhead in terms of disk I/O. Further, its reliance on the expensive semi-join operation made its adoption infeasible for DBMSs, and as such this algorithm saw little use over the next decades.

3.2.2 Parachute

While a full reduction is the ideal outcome of pre-filtering from an algorithmic perspective, the seeming requirement for a second pass over the data in order to achieve this goal is a major roadblock for such techniques to see use in production systems. A recent technique called Parachute, developed by Stoian et. al., aims to take a step in solving this by enabling some bi-directional pre-filtering with only a single pass of the data [14]. The core contributions are twofold: they first formalize the idea of information flow in a query, and then details the algorithms and storage modifications needed to implement Parachute.

With probe-side filtering, information naturally flows "upwards" through the probe pipelines, but information flow does not happen in the "downwards" direction. Parachute aims to enable this reverse direction information flow for the the restricted case of primary/foreign key joins, where the foreign key table is "below" the primary key table. The key insight is that a big source of blocked information flow is when a primary key table in a later probe pipeline has a filter on it, which could be used to filter the foreign key table in an earlier probe pipeline. In order to do so while still requiring only a single pass over the data, it moves the burden onto the storage and table insertion/update rather than filter construction. If probe-side filtering is not enabled, Parachute is also able to detect that the "upwards" information flow is missing and can handle that accordingly.

Parachute enables the storage of small approximations of relevant columns of the primary key table as new **parachute columns** in the foreign key table, and at runtime allows for translated filters from the primary key table to be applied to foreign key tables. These approximations are created to be much smaller than the original column, making use of approxmations such as hashing, histograms, string fingerprints, and other techniques, in order for the additional overhead when scanning to remain minimal. As it is infeasible to construct parachute columns as needed while executing a query, these must be pre-computed based on the expected workload. Maintaining the accuracy of these values through updates/insertions is slow, especially if the primary key table is being updated, but for workloads where the frequency of these operations is low the overhead is not substantial.

Although Parachute does not achieve a true full reduction, the results still show significant benefits when enabling it on DuckDB both with and without its probe-side filtering enabled. A large portion of unnecessary tuples were removed on both the JOB and CEB benchmarks resulting in reduced overall runtime. With probe-side filtering enabled, the results show a smaller improvement, but still show that efficiently removing additional dangling tuples can be useful.

3.2.3 Robust Predicate Transfer (RPT)

Robust Predicate Transfer (RPT) is a recent technique created by Zhao et. al. that builds on the ideas of the Yannakakis algorithm and a previous technique called Predicate Transfer in a modern

and performant fashion [21] [19] [18]. Predicate Transfer addressed a key issue with Yannakakis' technique by using cheaper Bloom filters rather than semi-joins to perform the full reduction, and RPT further improves upon this work with three key contributions and an implementation in DuckDB v0.9.2.

The first contribution is the `LargestRoot` algorithm to create the pre-filter **transfer schedule**, which is the order in which filters are passed in each of the two Yannakakis-style passes (here denoted "forwards" and "backwards" in place of upwards and downwards respectively). Specifically, while Predicate Transfer did not use a join tree for its transfer schedule, RPT returns to the Yannakakis algorithm in order to guarantee a full reduction. This algorithm constructs a maximum spanning tree on the join graph with the largest table as the root. The reasoning behind this specific choice of root is that it is generally preferable to pre-filter larger tables as much as possible before constructing filters from it, so that these resulting filters can be as small and selective as possible.

While in the Yannakakis algorithm, joins are executed only as present in the join tree, in order to guarantee performance complexity linear in terms of the inputs and outputs, this prevents the optimzer from choosing join orders which may achieve better performance. However, we still want to ensure tight bounds on any intermediate results of joins in order to achieve this complexity. The second contribution is the `SafeSubjoin` algorithm to identify query plans with such a "safe" join order. It is shown that a subset of acyclic queries known as "γ-acyclic" guarantee that any plan chosen by the query optimizer will be safe. For all other acyclic queries, the optimizer must ensure that the chosen join order is safe.

The third contribution is empirical robustness of RPT regardless of the chosen join order. It was demonstrated that when applying RPT on many randomly chosen join orders, the performance for γ-acyclic queries is generally within a very narrow bound. It is important to note that this notion of robustness is not the same as is guaranteed by LIP, since there is no mathematical guarantee on the performance spread with different query plans [22]. Ultimately, this combined with the overall performance improvements after applying RPT show that the join order problem may be closer to solved for γ-acyclic queries.

Dynamic Predicate Transfer (RPT+)

Recently, there has been an updated version of RPT under development known as Dynamic Predicate Transfer (RPT+) in a newer version of DuckDB [6]. This technique aims to improve the performance of RPT with three changes: removing passes determined not to be useful from the transfer schedule, early termination of Bloom filter creation and probing pipelines in certain cases, and additionally creating and passing min/max filters.

The first two changes aim to address the issue of pre-filter passes that do not filter many tuples, by removing them at optimization time and execution time respectively. While this may result in useful passes being skipped, overall the change aims to improve the performance of this technique compared to RPT. The third change is less structural, but may provide significant benefits for minimal overhead. By passing min/max filters alongside the existing Bloom filters, we can increase the selectivity (in some cases significantly) and still maintain the existing guarantees of the full reduction algorithm. Min/max filters are created within `CreateBF` operators, and to take advantage of DuckDB's row group skipping, are always passed down to the `Scans` below

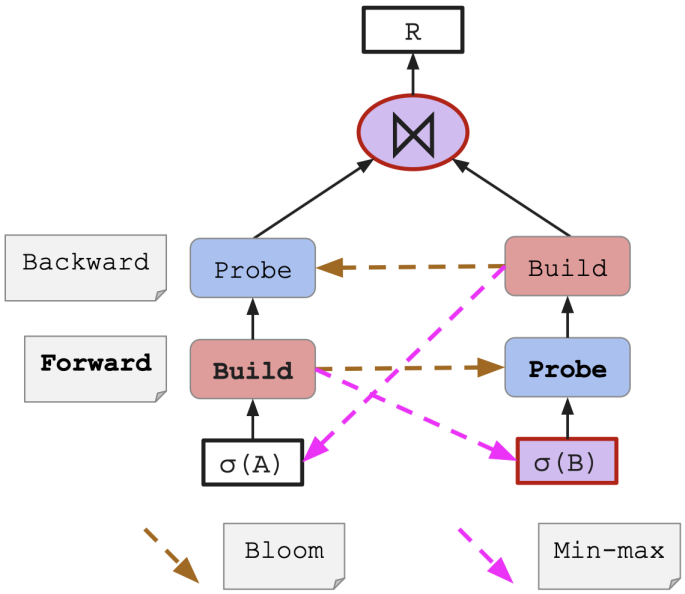

Figure 3.1: Filter passing schedule for a single-join query with RPT+.

the corresponding `UseBF`. This, to the best of our knowledge, is the first pre-filtering technique that explicitly calls for applying multiple types of filters on all tables.

3.3 Missing Passes in RPT+

Currently, RPT+'s min/max filter passes are not working as intended. The pushdown of min/max filters presents some issues upon the the backwards pass of RPT+, as seen in the example in Figure 3.1. Specifically, since the forwards pass must complete before the backwards pass, requiring the `Scan` to use a filter created during the backwards pass causes a cyclic dependency in the algorithm.

In the current implementation, this issue is manifested as the backwards pass min/max filters getting created, but never getting used as their destination must already have been executed. We propose and implement a simple change to correct this issue, where min/max filters on the backwards pass are not pushed down and instead are evaluated with the `UseBF`, as in Figure 3.2.

This "missing pass" means that until now, there have not been any pre-filtering techniques that correctly use multiple filters to fully reduce the tables. There are also no prior studies done on the performance of a full reduction using non-uniform approximate filters. We aim to investigate both of these gaps in the following experiments.

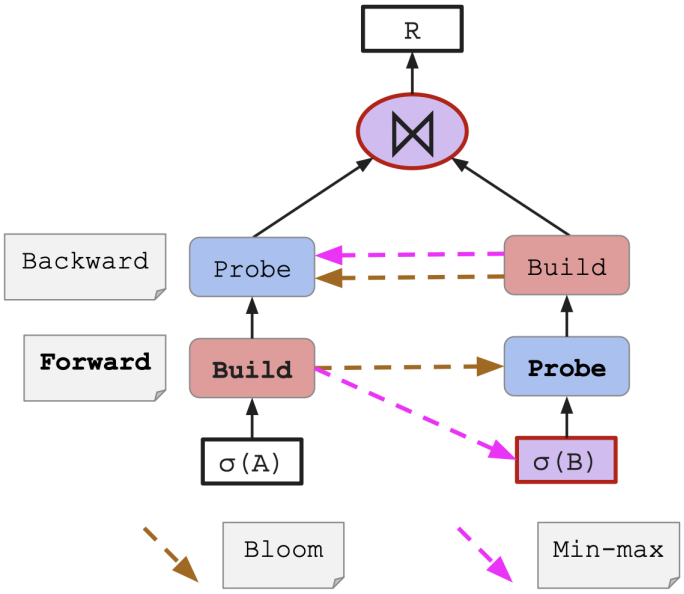

Figure 3.2: Corrected filter passing schedule for a single-join query with RPT+.

Chapter 4

Exploring Filter Interactions in RPT+

Explicitly using multiple types of approximate filters on a single table is a very new development in pre-filtering. Aside from RPT+ [6], most techniques have explicitly stated the single type of filter to use in any given circumstance. In this new space, there are a few natural questions to ask:

- What effects do different filter types have on each other?
- Is there a clear best filter combination to apply?
- What workloads and data are the best case for each filter type?

We aim to investigate these questions for the specific case of Bloom filters and min/max filters, two of the most commonly used uniform and non-uniform approximate filters in pre-filtering. Specifically, we evaluate their efficacy in DuckDB v1.3.0 [13] using the modified `LargestRoot` transfer schedule used in RPT+ in order to fully reduce all input tables with respect to the applied filter configuration [6]. The specific implementation used is our modified version of RPT+ which correctly passes and applies min/max filters [1]. We further implemented an additional extension that allows either filter type to be enabled or disabled independently. Notably, this work is the first known evaluation of a pre-filtering algorithm making use of only min/max filters to approximate a full reduction.

This base implementation was chosen for two main reasons: it uses optimized implementations of the pre-filtering data structures (Bloom filters from Apache Arrow and lightweight, vectorized min/max filters), and the use of DuckDB ensures that our data reflects a highly performant baseline and therefore has realistically interpretable results [1].

4.1 Workloads and Experimental Setup

In this work, we evaluate the performance of four different pre-filter configurations: no pre-filtering, Bloom filters only, min/max filters only, and both filters. We ran three sets of benchmarks, commonly used to measure the performance of analytical DBMSs.

[1]Our modified implementation can be found at `https://github.com/mihirkhare/transfer-research`.

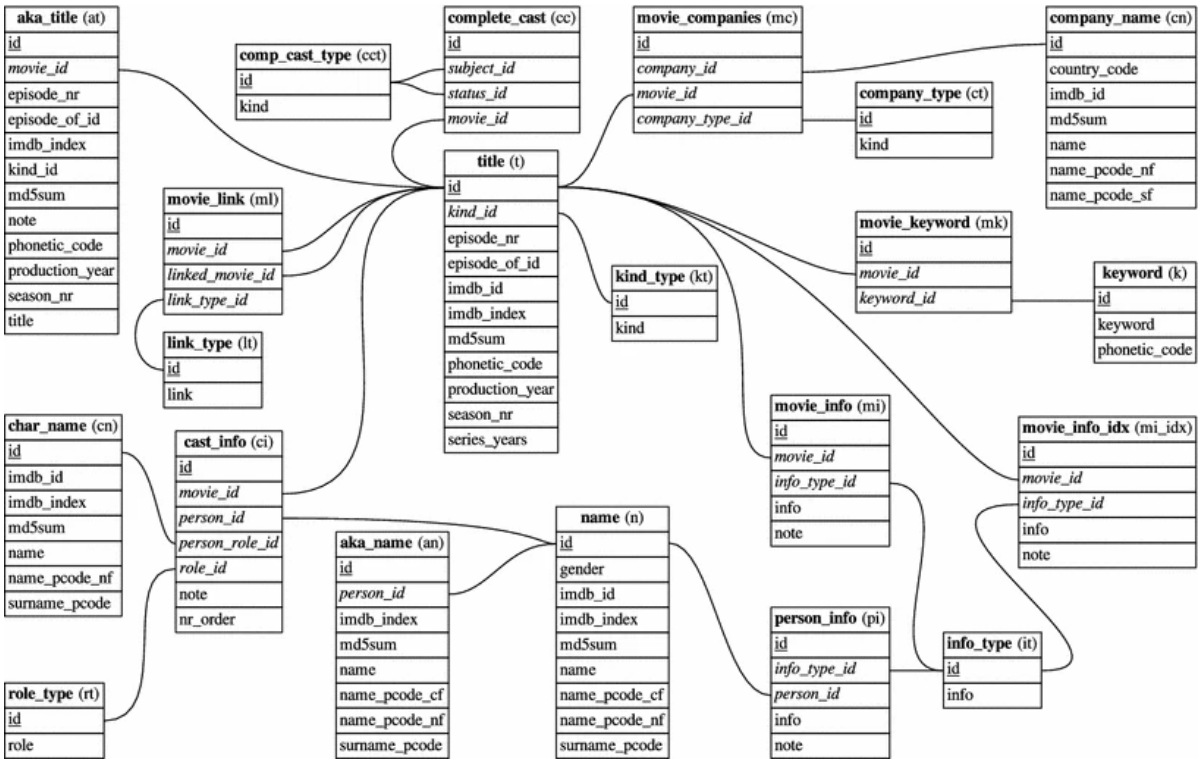

Figure 4.1: Schema of the IMDB dataset in JOB [8].

4.1.1 Workloads

The first is the Join Order Benchmark (JOB), developed by Leis et. al. to measure the quality of join order optimization and cardinality estimation in a query optimizer [7]. It is built on a real-world IMDB dataset (schema in Figure 4.1), with 113 join-heavy queries designed to emulate realistic workloads on this data. These queries are divided into 33 "groups" of 2-6 queries each, where each group follows roughly the same query structure with minor differences (e.g., different filter constants). In our case, the DuckDB query optimizer is already fairly high quality, so we aim to see how much of an impact different pre-filtering configurations can have on top of the high baseline.

Second is the TPC-H benchmark, a decision support benchmark that has seen widespread use in literature over the past 25 years [15]. Its dataset consists of 8 tables related by primary and foreign keys (schema in Figure 4.2), and is generated from a uniform distribution. TPC-H allows for a "scale factor" to be specified when generating the data, which roughly corresponds with the size of the dataset in gigabytes. Here, we used scale factors 1, 10, and 100, roughly corresponding to 1GB, 10GB, and 100GB respectively. The workload consists of 22 queries that vary in complexity, designed to test all parts of an analytical DBMS. The main distinctions between this and JOB are the data distributions in the workload, as well as a smaller focus on join processing specifically.

The third benchmark is TPC-DS, which is a newer decision support benchmark designed to be much more complex than TPC-H [12]. Its dataset follows a snowflake schema consisting

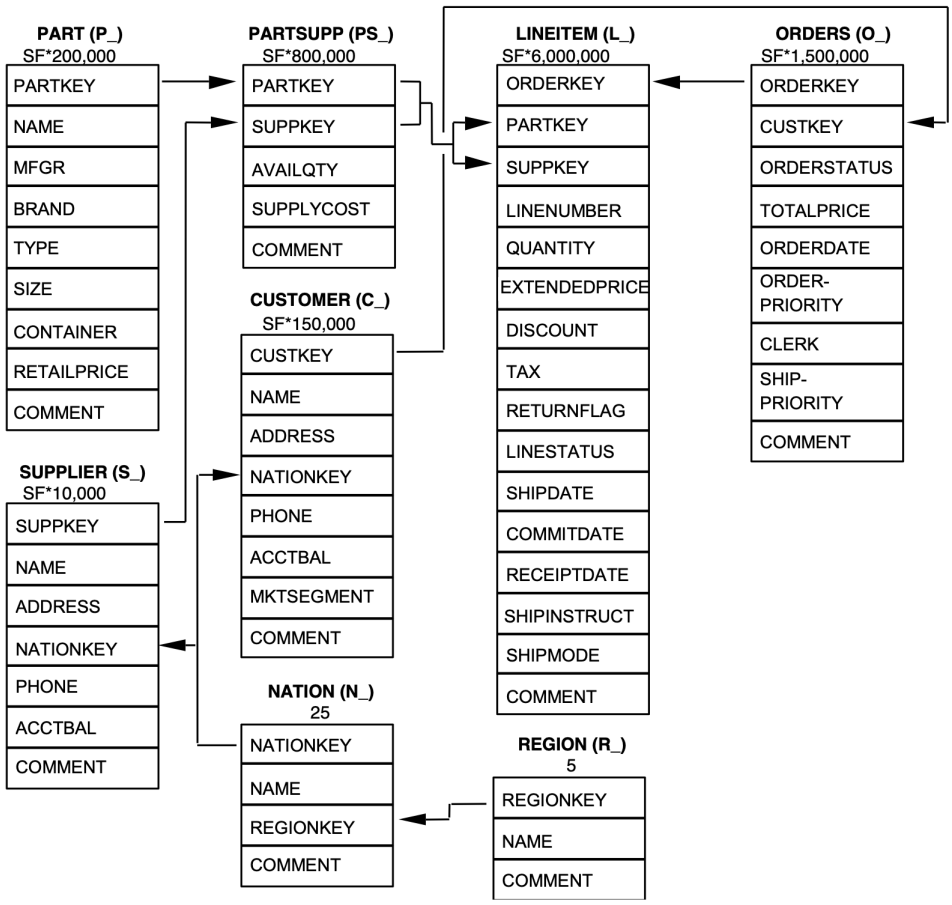

Figure 4.2: Schema and cardinalities of the TPC-H dataset [15].

of 25 tables (compared to 8 in TPC-H), and a hybrid data generation approach that uses both synthetic and real-world sources to better model modern data analytics workloads. The workload consists of 99 queries aiming to cover the wide range of requests encountered by decision support systems. Although this benchmark is somewhat similar to TPC-H, many modern DBMSs still use this as a valuable measure of performance, and as such we will utilize it here as well.

4.1.2 Experimental Setup

All evaluations were performed on a 10-core Intel Xeon E5-2630 v4, with a 5MiB L2 cache and a 50MiB L3 cache, on top of 128GiB of DDR4 RAM. In order to investigate the behavior of and interactions between Bloom and min/max filters, we integrated our implementation into DuckDB's built-in benchmarking harness, which we used in all evaluations to accurately determine the total query runtime.

For each query within each benchmark, the results are reported for 5 runs, preceded by a single unmeasured "warmup" run to populate memory and the cache with the necessary table data. The key datapoint here is the average across the 5 runs for each query, which is what is used to compare between the different filter configurations.

Every experiment was run in a single-threaded environment to simplify the space of analysis, although we don't expect there to be significant differences in interactions if multiple threads are used.

4.1.3 Note on Robustness

In this work, we do not evaluate the effects of different filter configurations on the "robustness" of the transfer schedule used, as has been previously studied with LIP and RPT [22] [21]. We instead seek to understand the separate and combined effects of Bloom and min/max filters on only the plan created by DuckDB, for the following reasons:

1. The goal of this work is not to analyze the underlying pre-filtering algorithm, but rather to better understand the interactions between the two types of filters used.
2. Varying the join order does not affect when and how pre-filtering occurs or the efficacy of each filter.
3. The plans generated by DuckDB reflect realistic execution strategies, which provides an indicator of how different filter configurations may perform in real-world settings.

The current robustness arguments for LIP and RPT rely on significant assumptions about the shape and distribution of the data, and the behavior of the hash functions. It seems reasonable to expect that min/max filters can exhibit robustness under a different set of assumptions, although there is currently no known work that has studied this.

4.2 JOB

With JOB, we see a few key observations from the overall data in Figures 4.3 and 4.4:

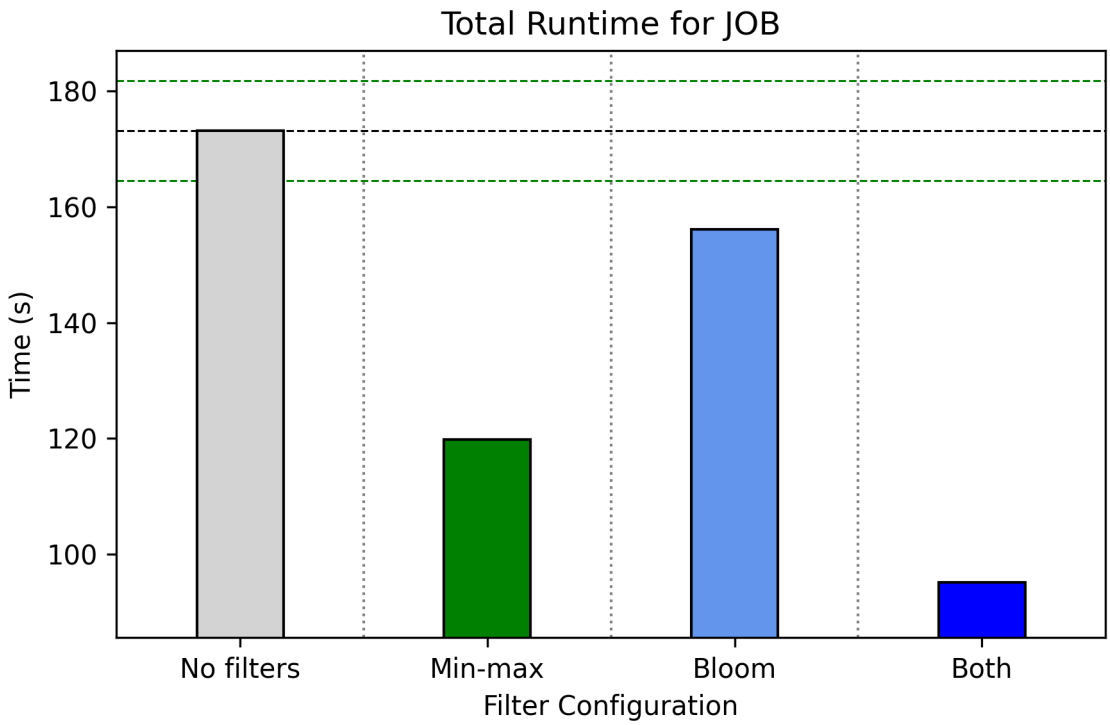

Figure 4.3: Query runtime for all filter configurations on JOB, with 95% and 105% of the baseline marked by a dashed green line. The y-axis is limited to fit the data.

Best Config	Min-max off	Min-max on
Bloom off	10	18
Bloom on	5	80

Figure 4.4: Number of JOB queries which are fastest for each filter configuration.

- All pre-filtering configurations look to have an overall positive effect on the total runtime of JOB.
- The min/max only configuration is significantly more effective than the Bloom only configuration.
- The speedup of applying both types of filters is roughly the sum of the speedups of each type individually.

One interesting statistic is the number of queries which have a significant spread in performance between their best and worst performing filter configurations. Here, we chose 20% as the threshold for "significance". For JOB, 81% of queries tested have this significant spread in performance across configurations. Most of these, as expected from the total runtime in Figure 4.3, involve the configuration with both filters, with 71 of those 80 queries showing a significant spread in performance.

The data seems to indicate that pre-filtering with both filters is the clear best option. However, 16 out of the 18 queries where min/max only is best, all 5 of the queries where Bloom only is best, and 6 of the 10 queries where no filtering is best show a significant spread. Further, for 8 of the queries where applying both filters is not the best choice, their best option is over 20% faster than applying both! It is clear that no single choice of pre-filter configuration is truly correct for this workload without causing significant loss in performance for a portion of the queries.

For many query groups, we observed that each query within them shows roughly similar performance trends between the different configurations. However, this is not universal — there are quite a few query groups where the specific constants within the template significantly changed the resulting data and therefore the effectiveness of pre-filtering. This indicates that while the static query shape may be a good initial indication of the pre-filtering configuration to use, it is still quite dependent on the specific query instance being evaluated.

With JOB, the primarily noticable trend in the total data for all queries is that the min/max filter configuration seems to have around a 30% speedup from the baseline of no pre-filtering, and applying both filters is also around a 30% speedup from applying only Bloom filters, which itself is about a 10% speedup from the baseline. So why is pre-filtering effective on JOB, and why are min/max filters especially useful?

4.2.1 Why Pre-Filtering, and Why Min/Max?

Much of these effects can be attributed to the schema and data distribution of the IMDB dataset. Specifically, the tables are related by a web of primary and foreign keys (as seen in Figure 4.1), which the query set makes heavy use of. Many queries involve tables with filter predicates that result in only small range of tuples or even just a single tuple, which are the ideal cases for both Bloom and min/max filters. The real-world sourcing of the data, with its natural skew and correlations between columns and tables, is also an important factor. Both of these traits can cause join columns to fall in tigher clusters and have fewer unique keys involved in joins, which improves the performance of both Bloom and min/max filters, and more generally of any approximate filter.

One big reason why min/max filters are especially effective may be because of DuckDB's row group skipping, which can significantly reduce the work needed to evaluate filters when

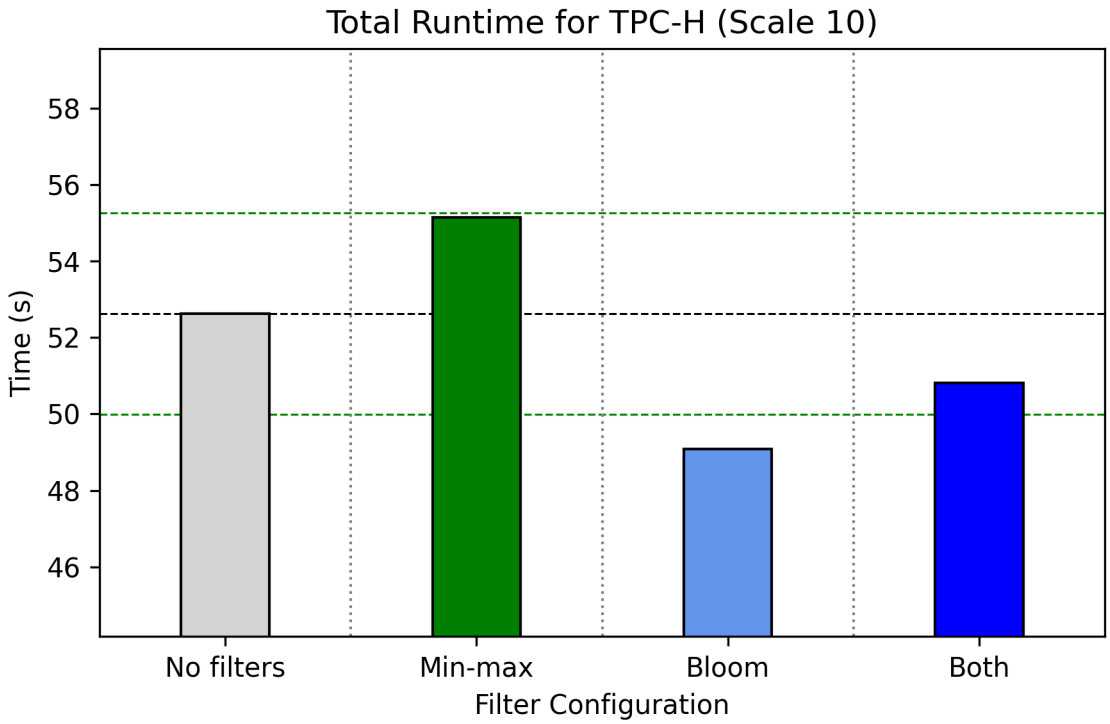

Figure 4.5: Query runtime for all filter combinations on TPC-H @ SF=10. The y-axis is limited to fit the data.

they are narrower as they are here. Since many of the min/max filters in our implementation are evaluated in `Scans`, this can result in significant speedup if the filtering range is narrow, as is often the case with queries in JOB.

4.3 TPC-H

For TPC-H, we evaluated the performance of all pre-filter configurations on scale factors 1, 10, and 100, roughly corresponding to 1GB, 10GB, and 100GB datasets respectively. For the following analyses, we will focus specifically on the 10GB dataset. We found that the general trends were roughly similar between all three scales, and found that the 10GB size is less prone to

Best Config	Min-max off	Min-max on
Bloom off	8	1
Bloom on	9	4

Figure 4.6: Number of TPC-H queries @ SF=10 which are fastest for each filter configuration.

noise than the 1GB size, while still ensuring that all tables can reside in memory on our evaluation machine. The overall data for the 1GB dataset can be found in the appendix in Figures A.1 and 4.6, and for the 100GB dataset in the appendix in Figures A.3 and A.4.

Looking at the data in Figures 4.5 and 4.6, our initial observations here are quite different from what we saw with JOB:

- There is no clear best pre-filtering configuration for this workload.
- Bloom pre-filtering appears somewhat useful, while adding min/max pre-filtering decreases performance in both cases.
- The overall effectiveness of pre-filtering is fairly weak in total.

Our first observation is perhaps the most interesting — with JOB, an okay strategy would have been to choose the configuration with both filters, but with TPC-H no single configuration is best on even half of the queries. 4 of the 8 queries where no pre-filtering is best, 6 of the 9 where Bloom is best, and the query where min/max is best all have significant spreads across their configurations (at least a 20% spread as before). Further, comparing only the no filter and Bloom filter configurations, there are 3 queries where Bloom pre-filtering is at least 20% better than no pre-filtering, and 1 query in the reverse comparison. As with JOB, making a single static choice for the workload is not the optimal strategy, and for TPC-H it would cause regressions on a much larger proportion of the query set.

The results from TPC-H show a very muted impact of any pre-filtering, with only half of the queries showing significant spreads between the configurations. No configuration shows more than about a 7% difference from the baseline in the total runtime. We aim to analyze what characteristics of the workload cause this.

4.3.1 Why Bloom, Somewhat?

One of the main differences between JOB and TPC-H is the method of creating the dataset. Compared to the real-world source of JOB's data, TPC-H generation of data from a uniform distribution results in very different relationships between different columns and across relations. In effect, this means that if columns `a` and `b` of a table `X` have no correlation, then performing a join on column `a` would tell you little information about what values would be matched on a future join on column `b` within a query.

With min/max filters, the width of theaccepted range directly affects how useful the filter is. With uncorrelated column values, the iterative filtering effects of multiple joins may do relatively little to shrink the ranges of keys involved in each join. Taking the previous example, the first join on column `a` may make table `X` half the original size, but the range of column `b` may be unchanged. The schema (seen in Figure 4.2) shows that there is a strong primary and foreign key structure that queries utilize effectively but without relationships between columns there is limited range shrinking that will occur through joins. A Bloom filter, however, can still remain effective in such scenarios — if the number of unique values of column `b` has shrunk, then the Bloom filter will have stronger filtering ability regardless of the range of values present.

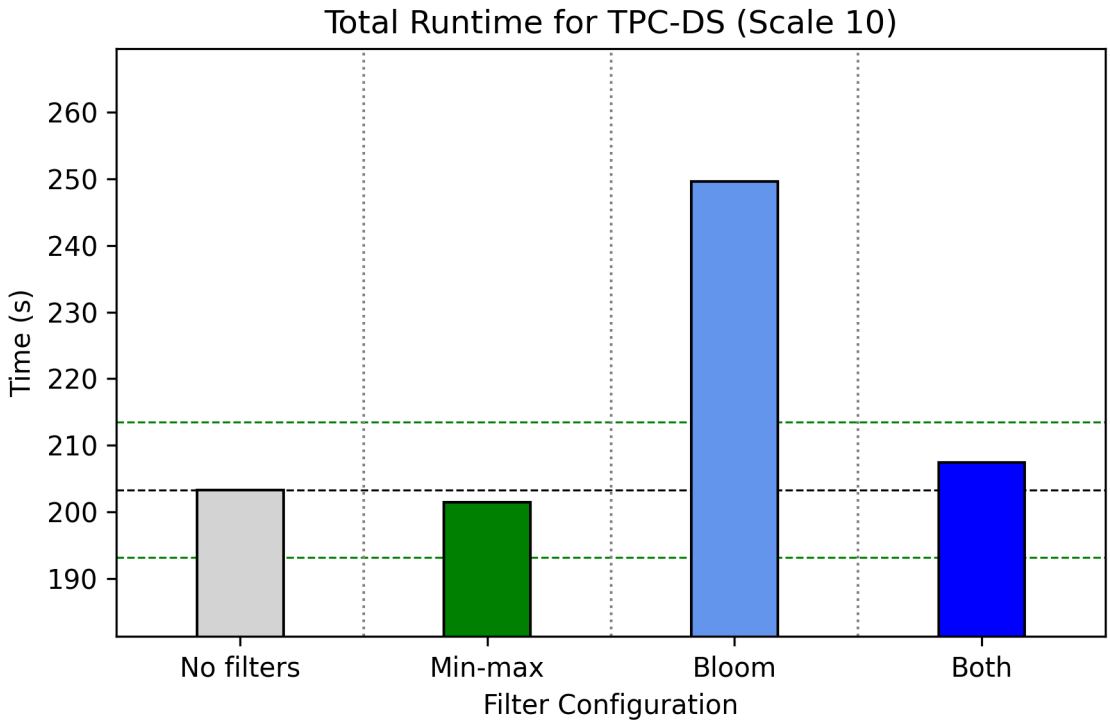

Figure 4.7: Query runtime for all filter combinations on TPC-DS @ SF=10. The y-axis is limited to fit the data.

4.4 TPC-DS

When running TPC-DS, we again evaluated all filter configurations at scale factors 1, 10, and 100, again corresponding roughly with dataset sizes of 1GB, 10GB, and 100GB. Like with TPC-H, we saw that the general trends between the different sizes, and therefore we chose the 10GB dataset for the same reasons. A caveat with the results of the 10GB and 100GB dataset is that query 18 was unable to run successfully due to an issue with the RPT+ implementation used for this work, so the results presented here (in Figures 4.7 and 4.8) and for the 100GB dataset do not include this query. All other queries worked as expected. The overall data for the 1GB dataset can be found in the appendix in Figures A.5 and 4.8, and for the 100GB dataset in the appendix

Best Config	Min-max off	Min-max on
Bloom off	55	25
Bloom on	1	17

Figure 4.8: Number of TPC-DS queries @ SF=10 which are fastest for each filter configuration.

in Figures A.7 and A.8.

Again, we have very different total results from either of the prior workloads. We can observe the following:

- No pre-filtering configuration has a strong positive impact on the total runtime.
- Bloom pre-filtering has a large negative impact on total runtime.

Like with TPC-H, we observe a fairly limited usefulness of any pre-filtering on the total runtime of the 98 queries evaluated (excluding query 18). The big exception to this is the Bloom filter only configuration, where the total performance was over 20% worse than any of the other configurations.

The minimal differences between the other three configurations is interesting, and is especially surprising considering that almost all queries show significant spreads across the four configurations — 49 out of the 55 no pre-filtering, the sole Bloom filter, 18 of the 25 min/max filter, and 13 of the 17 both filter queries meet our 20% threshold. Some of these queries show extremely significant impacts from even Bloom filters, with query 95 showing a reduction from around 7 seconds to less than 1 after applying Bloom pre-filtering.

4.4.1 Why No Pre-Filtering?

A big contributor to the effect we see in the overall data for this benchmark is the quantity of queries being run, and the large variety of differences between the tables and values involved. For such a diverse workload, it is difficult to gain significant information from aggregate data such as the total. Another reasons why pre-filtering is not very effective here may be that the cost of reducing the tables is not made up for by the savings in the joins.

However, at an individual level, many queries do show that pre-filtering, especially when min/max filters are used, is quite effective. Unlike with TPC-H, the data generation for TPC-DS incorporates some non-uniform data from external sources, which means that there is likely some amount of clustering and correlation present. As we observed with JOB, it seems that this data distribution is contributing to the relative effectiveness of min/max filters compared to Bloom filters.

4.5 Pre-Filtering with Skewed Data

A phenomenon we have consistently observed throughout the JOB, TPC-H, and TPC-DS workloads is that when the data distribution has non-uniform data, min/max filters seem to increase in effectiveness. But as both the dataset and query set are different across each workload, it is not clear to what degree the data distribution alone is contributing to this behavior. We want to determine if this effect is generalizable beyond this single example.

To distinguish between the effects of the data and query sets, we once again turn to the TPC-H benchmark, and more specifically its variant that incorporates skewed data generation for many of its tables [10]. This benchmark allows the same query set to be used as in the regular TPC-H, and the same scale factor selection, but samples its data from a Zipfian distribution rather than uniform. This both allows for a more realistic dataset with higher clustering and potentially

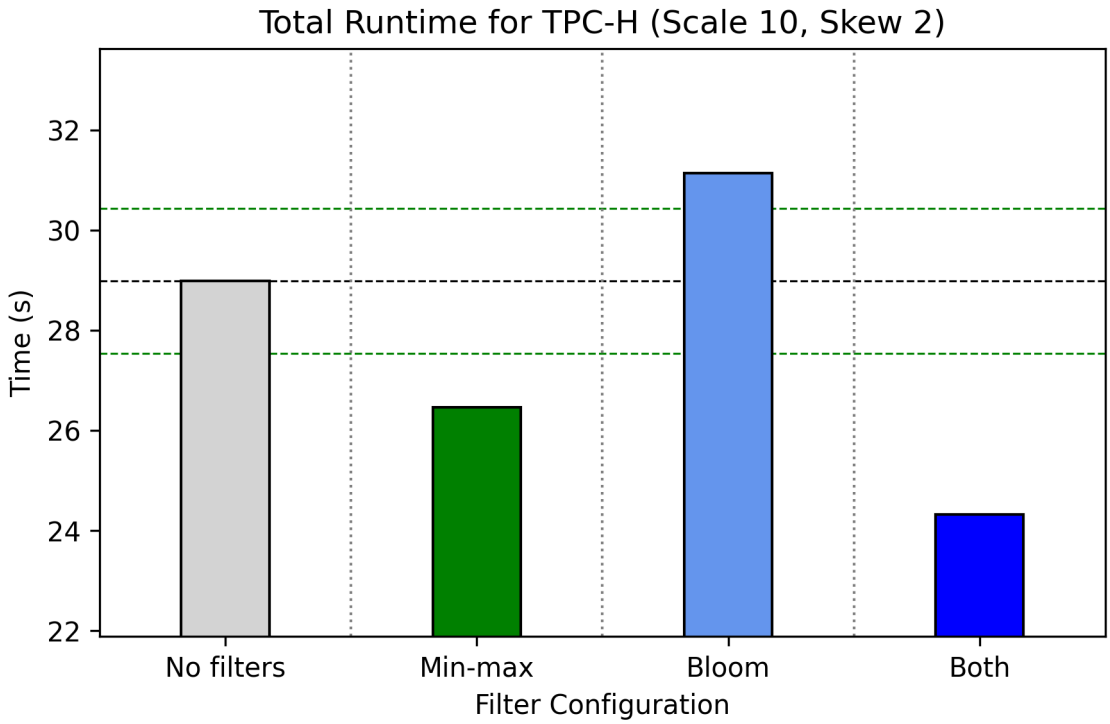

Figure 4.9: Query runtime for all filter combinations on TPC-H @ SF=10, Z=2. The y-axis is limited to fit the data.

higher correlations between columns, and gives us an ideal workload to observe the effects of the data distribution on the performance of min/max filters.

For these experiments, we focus only on the 10GB dataset size, and evaluated with skew factors (z value) 1, 2, 3, and 4, with 0 being the uniform distribution used previously. We analyze the results for skew factor 2 (in Figures 4.9 and 4.10).The overall data for the skew 1 dataset can be found in the appendix in Figures A.9 and A.10, for the skew 3 dataset in the appendix in Figures A.11 and A.12, and for the skew 4 dataset in the appendix in Figures A.13 and A.14. We have two main observations from these overall results:

- The overall data shows this workload is amenable to min/max pre-filtering.

Best Config	Min-max off	Min-max on
Bloom off	6	5
Bloom on	1	10

Figure 4.10: Number of TPC-H queries @ SF=10, Z=2, which are fastest for each filter configuration.

- Bloom filters are poor alone, but are quite effective when combined with min/max filters.

The first result seems to align with our hypothesis that non-uniform data can improve the performance of min/max pre-filtering, even when the query set remains fixed. 12 queries which had either no pre-filtering or Bloom pre-filtering as their best configuration in the original dataset now have min/max or both filters as their best, and only 2 queries show the opposite change.

4.5.1 What Changed?

We want to understand what about this skewed data generation is causing these results on the TPC-H query set. An interesting observation is that many query outputs are smaller with this skewed dataset than with the original uniformly distributed dataset, which may be a contributing factor to the usefulness of min/max filters. This is an example of the compounding shrinking effects of skew on data ranges across filters and joins, which means that the use of range-limiting filters such as min/max will make future data ranges narrower.

For Bloom filters, a big reason we see a performance regression when used alone is because this compounding effect does not occur to the same degree. Even if a Bloom filter represents data within a narrow range, its false positives have a uniform probability of occuring anywhere in the probe space, meaning that the data skew does not help much. The second major factor is the lack of row group skipping on highly selective or highly unselective filters. Even Bloom filters that only accept a single key will incur the cost of probing every single input, which is significantly more overhead than the equivalent min/max filter. In the more general case, the overhead of Bloom filters can significantly outweigh the effects of their filtering.

Conversely, when both filters are applied, we get the best overall performance. This may be because the use of min/max filters provides the compounding range shrinking behavior with row group skipping where applicable, and then Bloom filters are able to remove the unnecessary tuples within these narrower ranges.

Chapter 5

Conclusion and Future Work

In this work, we provided a new implementation for a full reduction min/max pre-filtering scheme, a corrected implementation of RPT+, and an analysis of how these different configurations of Bloom and min/max filters perform and interact on 4 commonly studied workloads. From our analysis of JOB and TPC-H with skewed data, we have clearly seen that min/max pre-filtering is highly effective on non-uniform datasets. We also saw that there were not many clear indicators as to when Bloom filters would be more or less effective for a given query or workload.

Overall, it is clear that fully understanding how and why Bloom and min/max filters interact in the observed manners is challenging. With JOB, for example, there were many cases where queries within a query group all had similar behavior across all four pre-filter configurations. But there were also many cases where a portion of the queries in a group had drastically different results, simply by changing a small portion of the filter conditions with the same overall query shape. We observed general trends for when each filter may be more or less effective individually, but we also observed that the combined effects are not simply compounding.

The results we have seen are fairly surprising: we showed that simply static information about a query and the data is not enough to determine the optimal pre-filtering configuration, and therefore that more work needs to be done to further explore this space.

5.1 Future Work: Adaptivity

An overarching observation across all workloads evaluated here is that no single choice of pre-filter configuration is generally optimal. As with all optimization problems, it is highly likely that an accurate algorithmic solution to this problem is infeasible to compute efficiently, and therefore would rely on inaccurate heuristics and estimates. There is also no evidence that the optimal configuration is one of the four studied here, and not something in the middle. Further, the addition of a second type of filter to the transfer schedule means that we can have a large number of filters to evaluate for each table in each pass. The order these filters are evaluated in can greatly impact the overall performance, but is also infeasible to accurate determine. One solution to these is 2 forms of runtime adaptivity: **adaptive reordering** of multiple filters at a single table, and **adaptive dropout** of minimally useful filters.

Adaptive reordering is quite similar to what is performed by LIP, and would allow the filters for each table at each pass to converge to the optimal order that minimizes the cost of pre-filtering (through selectivity measurements, execution time comparison, etc.) [22]. Similarly, this strategy would also be robust to shifts in the data distribution and can outperform a static order if the overhead for adaptivity is minimized. While generally it is optimal to evaluate the lightweight min/max filters before the more expensive Bloom filters, in queries with wide min/max ranges but sparse data, an adaptive algorithm could detect that some Bloom filters should be evaluated first.

Adaptive dropout would be the main new contribution to allow for dynamically converging on the best filter configuration for a query based on runtime information. This would involve some process of identifying filters that are not sufficiently useful (again through selectivity measurements, execution time comparisons, etc.) and occasionally disabling them. In order to make this strategy robust to shifts in the data distribution, there would similarly have to be a mechanism for previously disabled filters to be reenabled, in case they are now useful.

Like with the results of LIP, it is likely an adaptive solution could outperform a purely static implementation because it can adjust to be optimal at a more fine grained level than the aggregate over the whole query [22]. The proposed algorithm is unable to prevent minimally useful filters from being built in the first place, so further study into this area is necessary.

5.2 Future Work: Robustness

In this thesis, we did not study the effects of adding min/max filters, or only applying min/max filters, to the empirical robustness of RPT and RPT+. Given that there are such performance differences between the filter configurations on only the plan produced by DuckDB, it is reasonable to think that the performance spreads across different query plans could also change. Understanding how the underlying query plan affects the performance of each configuration may also help with our understanding of how Bloom and min/max filters interact and the types of data they will filter alone and together.

Appendix A

Additional Evaluation

Here, we provide the overall data for the other evaluated workloads that were not discussed in prior sections. These include:

- TPC-H @ SF=1 (Figures A.1 and A.2)
- TPC-H @ SF=100 (Figures A.3 and A.4)
- TPC-DS @ SF=1 (Figures A.5 and A.6)
- TPC-DS @ SF=100 (Figures A.7 and A.8)
- TPC-H @ SF=10, Z=1 (Figures A.9 and A.10)
- TPC-H @ SF=10, Z=3 (Figures A.11 and A.12)
- TPC-H @ SF=10, Z=3 (Figures A.13 and A.14)

Of note is that TPC-DS at scale factor 100 also was unable to successfully complete running query 18, so the data for it is for the remaining 98 queries only.

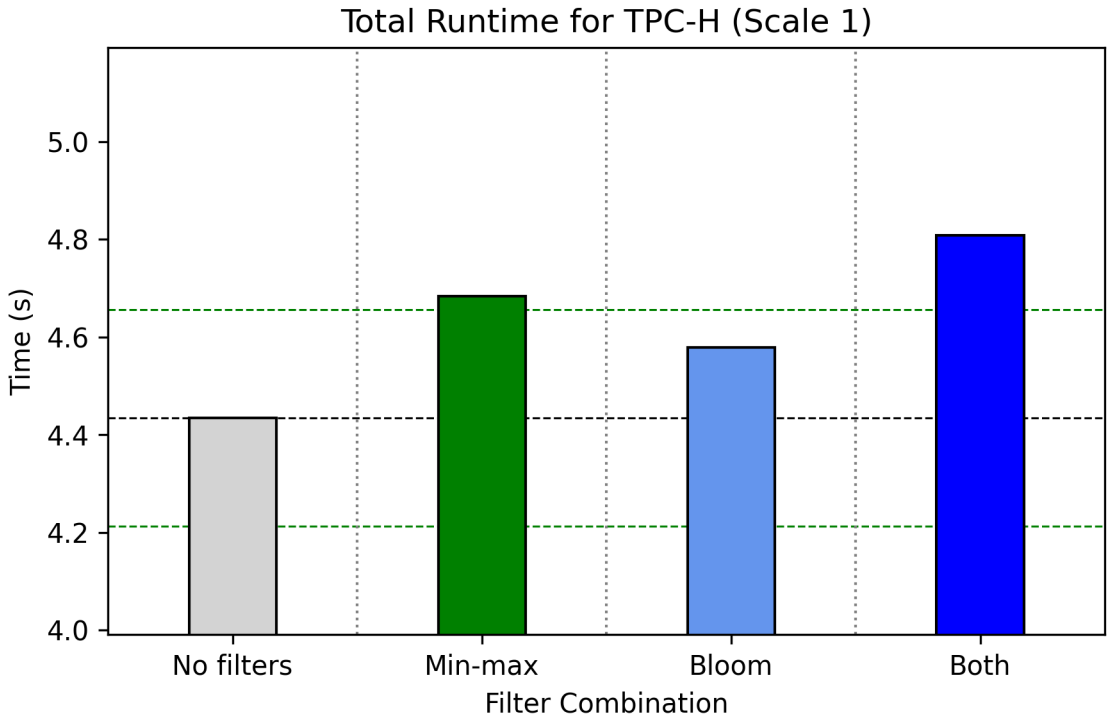

Figure A.1: Query runtime for all filter combinations on TPC-H @ SF=1. The y-axis is limited to fit the data.

Best Config	Min-max off	Min-max on
Bloom off	10	5
Bloom on	4	3

Figure A.2: Number of TPC-H queries @ SF=1, which are fastest for each filter configuration.

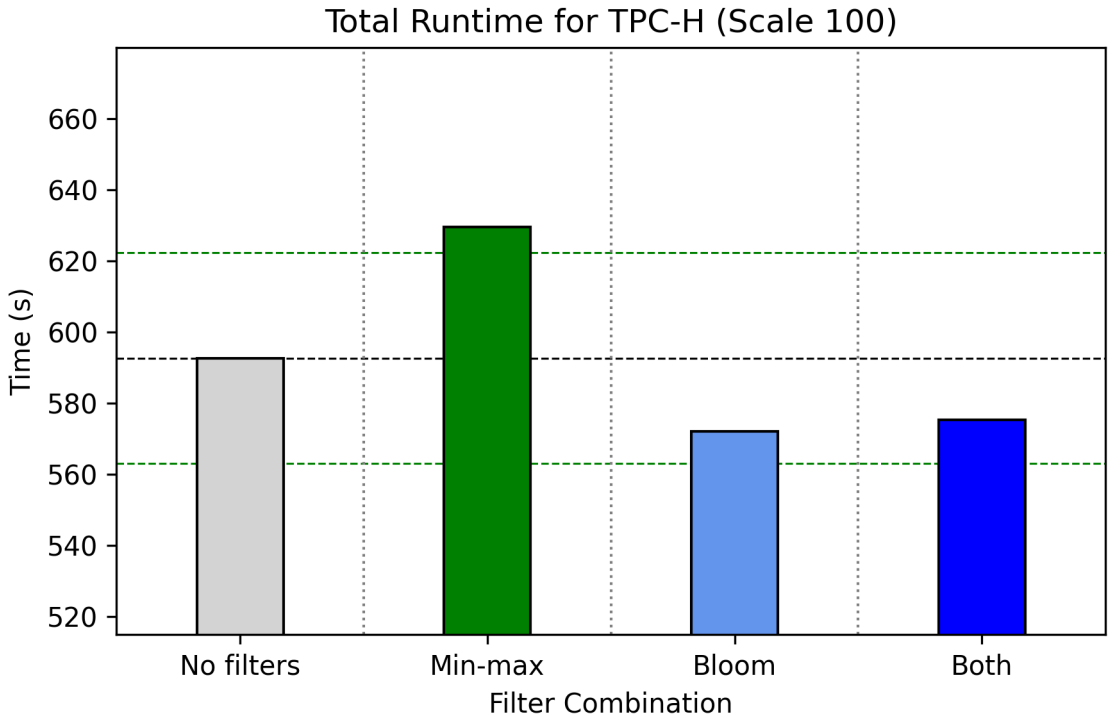

Figure A.3: Query runtime for all filter combinations on TPC-H @ SF=100. The y-axis is limited to fit the data.

Best Config	Min-max off	Min-max on
Bloom off	7	1
Bloom on	8	6

Figure A.4: Number of TPC-H queries @ SF=100, which are fastest for each filter configuration.

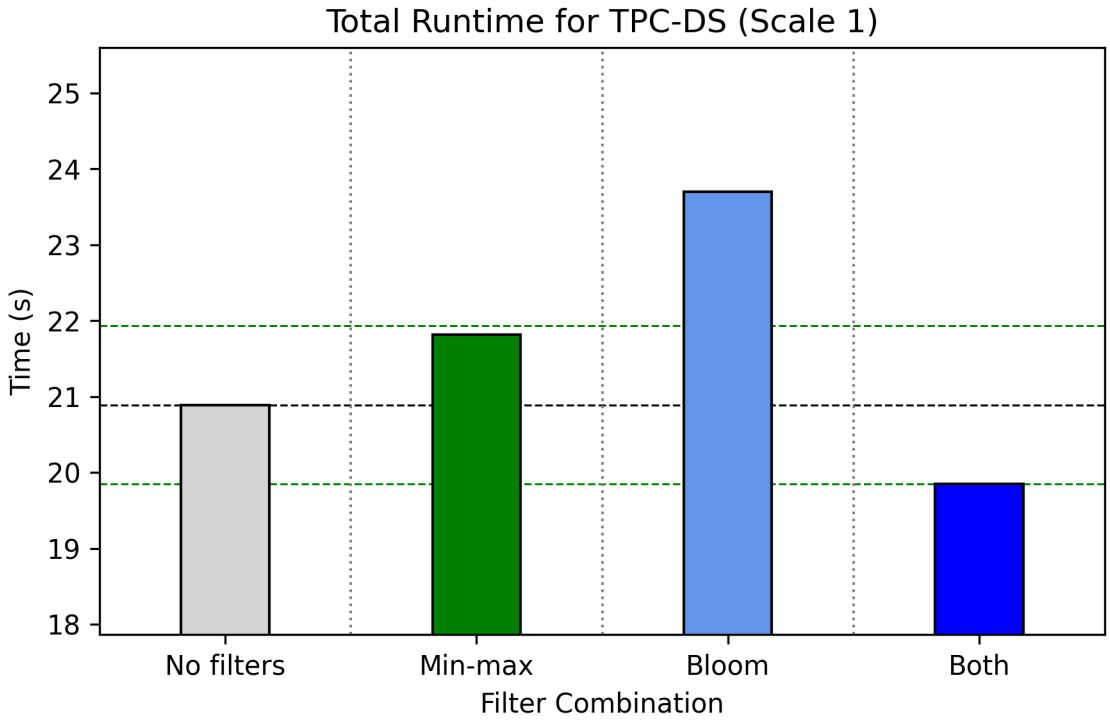

Figure A.5: Query runtime for all filter combinations on TPC-DS @ SF=1. The y-axis is limited to fit the data.

Best Config	Min-max off	Min-max on
Bloom off	62	14
Bloom on	4	19

Figure A.6: Number of TPC-DS queries @ SF=1, which are fastest for each filter configuration.

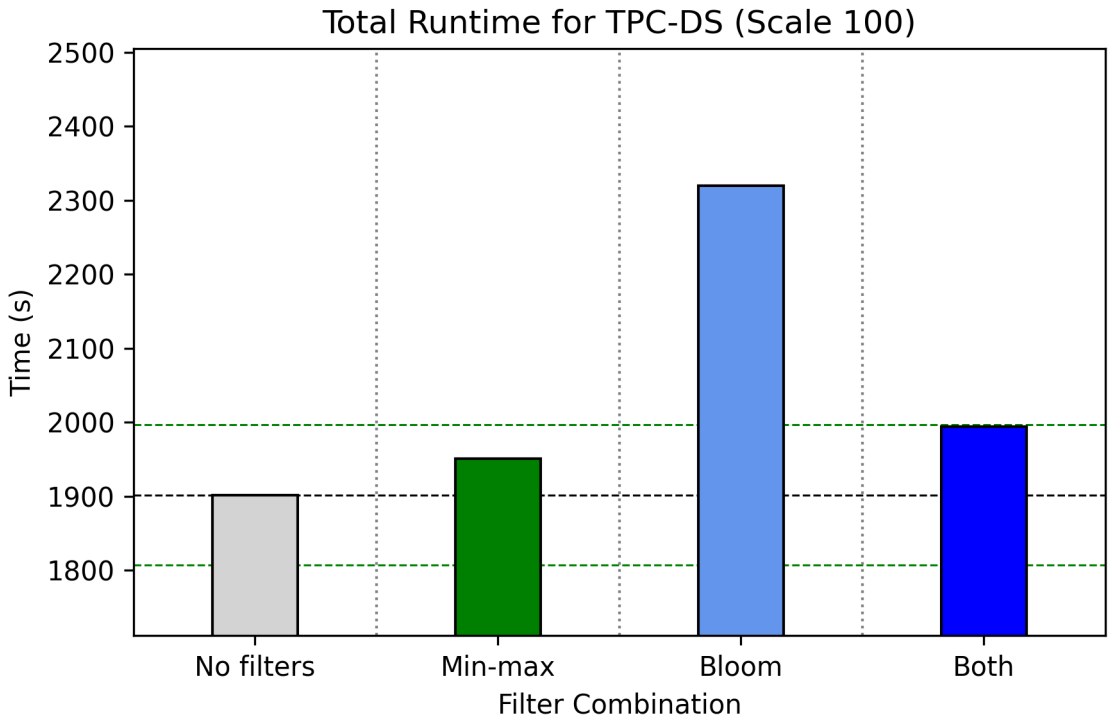

Figure A.7: Query runtime for all filter combinations on TPC-DS @ SF=100. The y-axis is limited to fit the data.

Best Config	Min-max off	Min-max on
Bloom off	59	26
Bloom on	1	12

Figure A.8: Number of TPC-DS queries @ SF=100, which are fastest for each filter configuration.

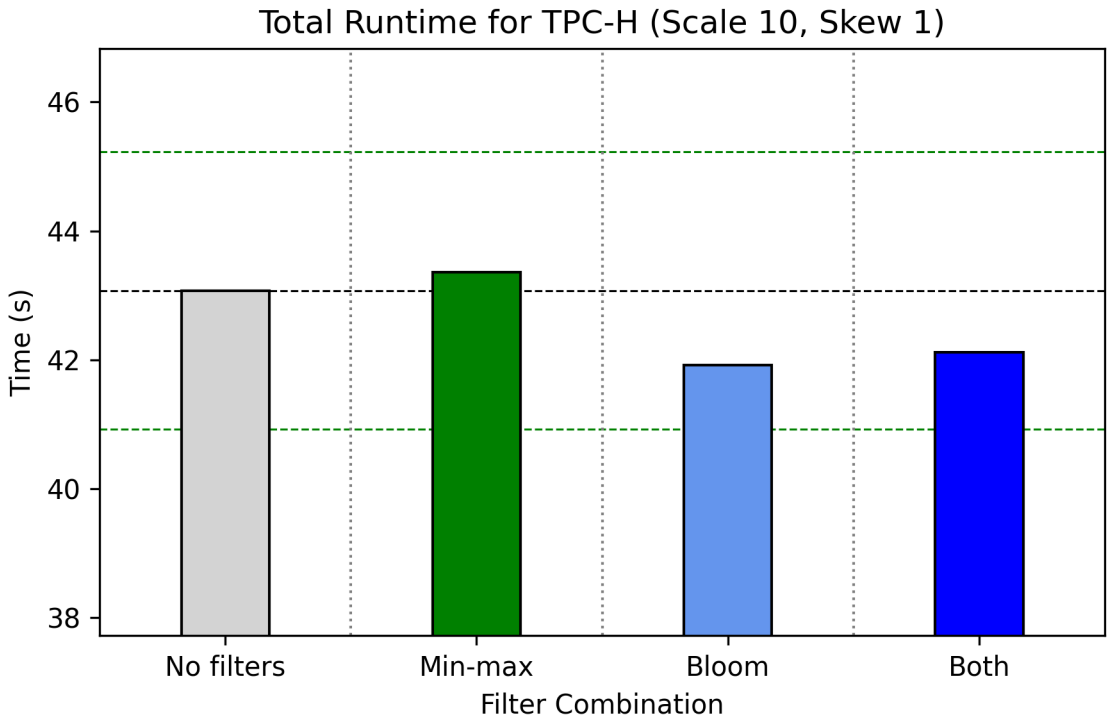

Figure A.9: Query runtime for all filter combinations on TPC-H @ SF=10, Z=1. The y-axis is limited to fit the data.

Best Config	Min-max off	Min-max on
Bloom off	8	3
Bloom on	5	6

Figure A.10: Number of TPC-H queries @ SF=10, Z=1, which are fastest for each filter configuration.

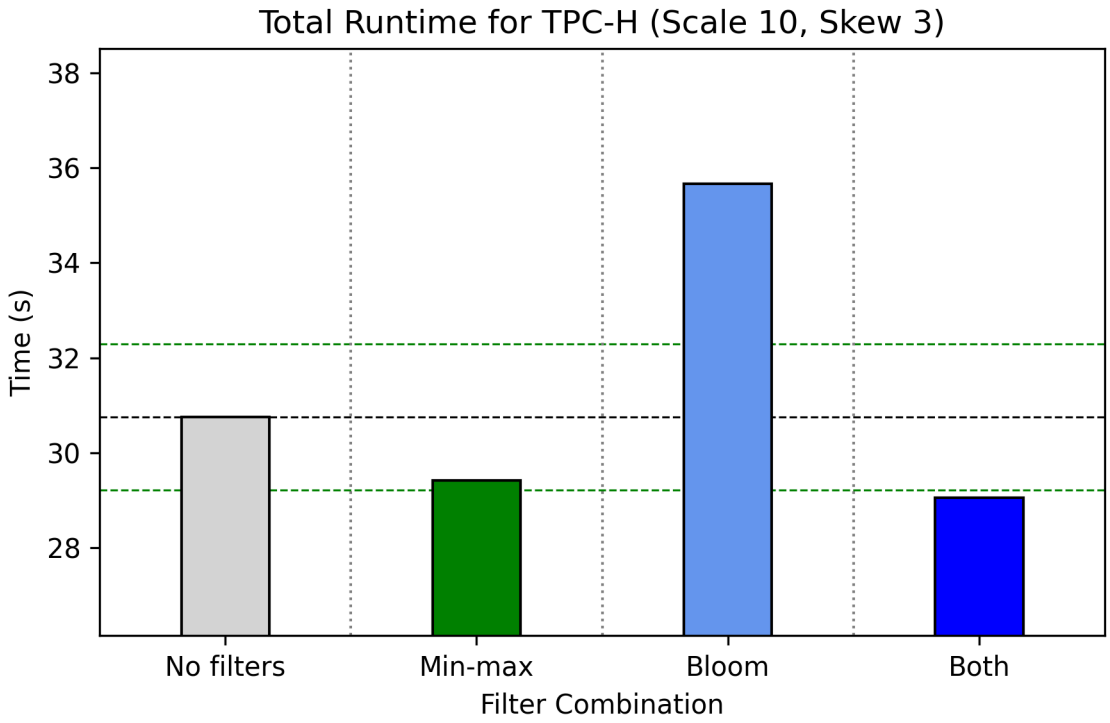

Figure A.11: Query runtime for all filter combinations on TPC-H @ SF=10, Z=3. The y-axis is limited to fit the data.

Best Config	Min-max off	Min-max on
Bloom off	7	2
Bloom on	1	12

Figure A.12: Number of TPC-H queries @ SF=10, Z=3, which are fastest for each filter configuration.

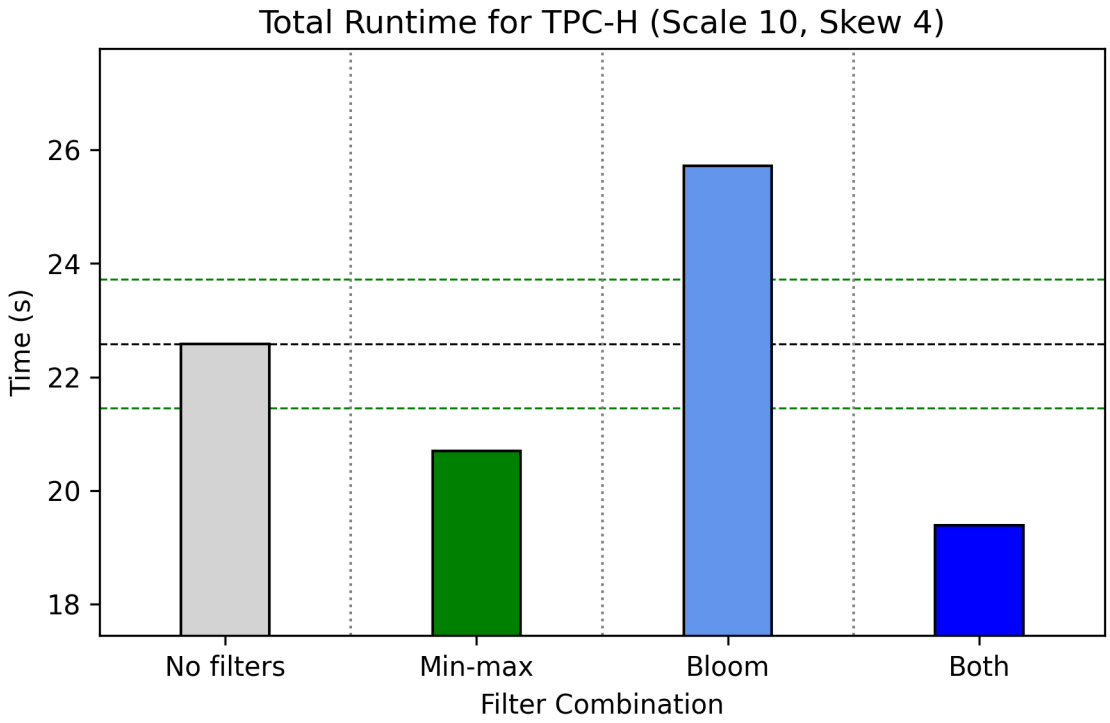

Figure A.13: Query runtime for all filter combinations on TPC-H @ SF=10, Z=4. The y-axis is limited to fit the data.

Best Config	Min-max off	Min-max on
Bloom off	7	3
Bloom on	4	8

Figure A.14: Number of TPC-H queries @ SF=10, Z=4, which are fastest for each filter configuration.

Bibliography

[1] Apache Arrow Developers. Apache arrow. https://arrow.apache.org/, 2016. 2.1.2, 4

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692. URL https://doi.org/10.1145/362686.362692. 2.1.2

[3] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. Bitvector-aware query optimization for decision support queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 2011–2026, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389769. URL https://doi.org/10.1145/3318464.3389769. 3.1.2

[4] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25 (2):73–169, June 1993. ISSN 0360-0300. doi: 10.1145/152610.152611. URL https://doi.org/10.1145/152610.152611. 3.1.2

[5] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, page 106–117, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919955. doi: 10.1145/276304.276315. URL https://doi.org/10.1145/276304.276315. 1.1.1

[6] Embryo Labs. Dynamic Predicate Transfer. URL https://github.com/embryo-labs/dynamic-predicate-transfer. 3.2.3, 4

[7] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, November 2015. ISSN 2150-8097. doi: 10.14778/2850583.2850594. URL https://doi.org/10.14778/2850583.2850594. 4.1.1

[8] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643–668, October 2018. ISSN 1066-8888. doi: 10.1007/s00778-017-0480-7. URL https://doi.org/10.1007/s00778-017-0480-7. (document), 4.1

[9] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for local queries. *SIGMOD Rec.*, 15(2):84–95, June 1986. ISSN 0163-5808. doi: 10.1145/16856.16863. URL https://doi.org/10.1145/16856.16863. 3.1.1

[10] Microsoft. Program for tpc-h data generation with skew. https://www.microsoft.com/en-us/download/details.aspx?id=52430, 2016. 4.5

[11] Oracle Corporation. *Optimizer Hints*. Oracle Corporation, 2002. URL https://docs.oracle.com/cd/B10500_01/server.920/a96533/hintsref.htm. Oracle9i Database Performance Tuning Guide and Reference, Release 2 (9.2), Part Number A96533-02. 1.1.2

[12] Meikel Poess, Raghunath Othayoth Nambiar, and David Walrath. Why you should run tpc-ds: a workload analysis. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 1138–1149. VLDB Endowment, 2007. ISBN 9781595936493. 4.1.1

[13] Mark Raasveldt and Hannes Muehleisen. DuckDB. URL https://github.com/duckdb/duckdb. 2.1.1, 3.1.2, 4

[14] Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Latyr Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. Parachute: Single-pass bi-directional information passing. *Proc. VLDB Endow.*, 18(10):3299–3311, June 2025. ISSN 2150-8097. doi: 10.14778/3748191.3748196. URL https://doi.org/10.14778/3748191.3748196. 3.2.2

[15] Transaction Processing Performance Council. TPC-H Benchmark Specification. Technical Report Revision 2.17.1, Transaction Processing Performance Council (TPC), 2014. URL http://www.tpc.org/tpch/. (document), 4.1.1, 4.2

[16] Haibo Xiu, Yang Li, Qianyu Yang, Weihang Guo, Yuxi Liu, Pankaj K. Agarwal, Sudeepa Roy, and Jun Yang. Hint-qpt: Hints for robust query performance tuning. *Proc. VLDB Endow.*, 18(12):5327–5330, August 2025. ISSN 2150-8097. doi: 10.14778/3750601.3750663. URL https://doi.org/10.14778/3750601.3750663. 1.1.2

[17] Xianghong Xu, Zhibing Zhao, Tieying Zhang, Rong Kang, Luming Sun, and Jianjun Chen. Coool: A learning-to-rank approach for sql hint recommendations, 2023. URL https://arxiv.org/abs/2304.04407. 1.1.2

[18] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. Predicate transfer: Efficient pre-filtering on multi-join queries. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org, 2024. URL https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf. 3.2.3

[19] Mihalis Yannakakis. Algorithms for acyclic database schemes. 01 1981. 3.2.1, 3.2.3

[20] Junyi Zhao, Huanchen Zhang, and Yihan Gao. Efficient query re-optimization with judicious subquery selections. *Proc. ACM Manag. Data*, 1(2), June 2023. doi: 10.1145/3589330. URL https://doi.org/10.1145/3589330. 1.1.1

[21] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. Debunking the myth of join ordering: Toward robust sql analytics. *Proc. ACM Manag. Data*, 3(3), June 2025. doi: 10.1145/3725283. URL https://doi.org/10.1145/3725283. 3.2.3, 4.1.3

[22] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. Looking ahead makes query plans robust: making the initial case with in-memory star schema data ware-

house workloads. *Proc. VLDB Endow.*, 10(8):889–900, April 2017. ISSN 2150-8097. doi: 10.14778/3090163.3090167. URL https://doi.org/10.14778/3090163.3090167. 3.1.3, 3.2.3, 4.1.3, 5.1

[23] Sergey Zinchenko and Sergey Iazov. Hero: Hint-based efficient and reliable query optimizer, 2024. URL https://arxiv.org/abs/2412.02372. 1.1.2