Parachute: Single-Pass Bi-Directional Information Passing

Mihail Stoian
University of Technology Nuremberg
Nuremberg, Germany
mihail.stoian@utn.de

Andreas Zimmerer
University of Technology Nuremberg
Nuremberg, Germany
andreas.zimmerer@utn.de

Skander Krid
University of Technology Nuremberg
Nuremberg, Germany
skander.krid@tum.de

Amadou Latyr Ngom
Massachusetts Institute of Technology
Cambridge, Massachusetts
ngom@mit.edu

Jialin Ding
Amazon Web Services
Boston, Massachusetts
jialind@amazon.com

Tim Kraska
Massachusetts Institute of Technology
Cambridge, Massachusetts
kraska@mit.edu

Andreas Kipf
University of Technology Nuremberg
Nuremberg, Germany
andreas.kipf@utn.de

ABSTRACT

Sideways information passing is a well-known technique for mitigating the impact of large build sides in a database query plan. As currently implemented in production systems, sideways information passing enables only a *uni-directional* information flow, as opposed to instance-optimal algorithms, such as Yannakakis'. On the other hand, the latter require an additional pass over the input, which hinders adoption in production systems.

In this paper, we make a step towards enabling *single-pass bi-directional* information passing during query execution. We achieve this by statically analyzing between which tables the information flow is blocked and by leveraging precomputed join-induced fingerprint columns on FK-tables. On the JOB benchmark, Parachute improves DuckDB v1.2's end-to-end execution time without and with semi-join filtering by 1.54x and 1.24x, respectively, when allowed to use 15% extra space.

PVLDB Reference Format:
Mihail Stoian, Andreas Zimmerer, Skander Krid, Amadou Latyr Ngom, Jialin Ding, Tim Kraska, and Andreas Kipf. Parachute: Single-Pass Bi-Directional Information Passing. PVLDB, 18(10): 3299 - 3311, 2025.
doi:10.14778/3748191.3748196

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/utndatasystems/parachute.

1 INTRODUCTION

A database system's task is to answer user queries as fast as possible. As simple as that might sound, getting to the edge of performance is hard and sometimes requires brilliant research ideas to emerge. One such optimization is pruning tuples and partitions of base tables. An interesting problem appears when one wishes to do so not only based on base table filters, but also via *join-induced* filters; this is what is known as sideways information passing (SIP) [6, 11]. The key idea is to pass *information* between tables as an exact or approximate representation of the key space of a joining table. However, doing this optimally remains a challenge on its own. What production systems now implement is, if the reader will, a rudimentary application of SIP, namely *probe-side filtering* (PSF): At query execution time, build sides and materializing pipeline sinks will compute, if appropriate, a bloom filter which is passed to the upcoming table scans in the query plan, i.e., in the current or later probe pipelines [24, 31]. The main intuition behind this approach is that pre-filtering the tables before the join occurs saves on expensive hash table lookups. This is a valid point, as inaccurate cardinality estimates can lead to incorrect build-side choices, resulting in excessively large hash tables during query execution. As we will argue next, this kind of information passing is *uni-directional*.

In theory, what a system actually *wants* to strive for is a full input reduction, i.e., the filtered base tables eventually contain exactly

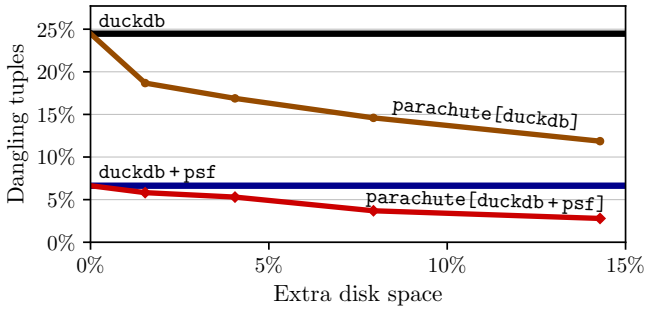

Figure 1: Number of *dangling* tuples in JOB [12] when running DuckDB v1.2 (`duckdb`) [23], probe-side filtering (`duckdb + psf`), and Parachute on both variants. Dangling tuples are those that do not survive full semi-join reduction and user-provided predicates. The numbers are relative to the total input size, excluding the non-dangling part.

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748196

the tuples that are used in the later join operators. We refer to the other tuples as *dangling* tuples. The type of algorithms capable of achieving zero dangling tuples are instance-optimal algorithms, such as Yannakakis' [27] for α-acyclic query graphs. Fig. 1 shows the percentage of dangling tuples in JOB [12] produced by DuckDB v1.2 (`duckdb`) and our PSF implementation (`duckdb+psf`), described in Sec. 4.1. Note that, for JOB, Yannakakis' algorithm removes all dangling tuples.

However, one of the reasons why instance-optimal algorithms do not make it into production systems is the overhead they incur, as the constant hidden behind the $\mathcal{O}$-notation can be rather large. Indeed, several papers report that Yannakakis' algorithm can be up to 5x slower [2, 26]. The underlying issue is that the algorithm itself requires *multiple* passes over the data. This is also true of the more recent approaches, such as Predicate Transfer [26], or its very recent robust version [29], which aim for an approximate semi-join reduction phase. Since these algorithms enable *bi-directional* information passing, requiring multiple passes seems understandable.

Research Question. Inspired by the wide adoption of PSF in production systems, the question is whether it is possible to enable bi-directional information passing through the query plan during query execution, without the need of the additional pass over the input, as required by instance-optimal algorithms.

Contribution. We make a step towards answering this question by first formalizing how information is passed in a PSF-enhanced query plan, showing that it only guarantees a uni-directional information flow, and, secondly, introducing precomputed join-induced fingerprint columns, called `parachute` columns, that are stored on FK-tables and enabled during query execution time to ensure a *bi-directional* information flow through the query plan. Unlike regular bitmap join indexes [4, 20], `parachute` columns generalize to high-cardinality columns over non-dimension tables and enjoy a monotonic performance-space tradeoff as shown in Fig. 1.

Note that Parachute not only supports probe-side pruning, but also build-side pruning if the query plan contains a FK-table on the build side. In a nutshell, Parachute trades off space and insertion performance for improved sideways information passing and hence query performance. We leave it up to the user to control this tradeoff, depending on workload characteristics. In future work, we plan to answer this tradeoff automatically. We show that, on JOB, Parachute achieves a reduction of 8.79x and 2.38x in the total number of dangling tuples over vanilla DuckDB and DuckDB with PSF, respectively. This reduction results in end-to-end execution speedup of 1.54x and 1.24x respectively, with only 15% space overhead and a 3.9x increase in load time, with further potential for optimization. For CEB, the increase in load time is amortized in a single run, whereas for JOB it takes 15 runs (of 113 queries) to amortize.

In summary, our contributions are as follows:

(1) We formalize the information flow induced by PSF during query execution.
(2) With Parachute, we achieve **single-pass bi-directional** information passing by enabling the missing information flow direction in PSF.
(3) We evaluate Parachute on the widely-studied JOB [12] and CEB [17] benchmarks.

Organization. The paper is structured as follows: In Sec. 2, we introduce pipelined execution (Sec. 2.2), probe-side filtering (Sec. 2.3), and formalize information flow (Sec. 2.4). In Sec. 3, we outline Parachute's design, focusing on the way how we represent the join-induced fingerprint columns. We continue with the evaluation on JOB and CEB in Sec. 4, and then with a discussion in Sec. 5. Afterwards, we outline related work (Sec. 6) and conclude in Sec. 7.

2 PRELIMINARIES

In this section, we introduce notations and formalize pipelined execution (Sec. 2.2) and probe-side filtering (Sec. 2.3), concluding with the formalization of the information flow (Sec. 2.4).

2.1 General Notation

Given a table (or relation) T, we refer to $\mathcal{A}(T)$ as the set of attributes of T (the attributes are grouped in equivalence classes, such that keys like `movie_keyword.movie_id` are equivalent to `title.id`).

Query Plan. A query plan is a binary tree in which the tables lie at leaf nodes, and the join operators at the inner nodes. In the following, we assume a hash-join implementation of the join operator, as this is currently the most efficient, with recent improvements in the last year [3, 10]. We follow the established convention that the build side is on the left, and the probe side is on the right of the (hash) join operator.

Semi-Join Reduction. In the following, by semi-join reduction we mean that a table S can be filtered by R, since it holds that $(S \ltimes R) \bowtie R = S \bowtie R$. The tuples that have been filtered out from S are called dangling tuples. By an approximate semi-join reducer we mean an approximate representation of the semi-join operator, yet with no false negatives, such that the correctness of the join operator is still guaranteed. This means that S might still contain dangling tuples. A *full* semi-join reduction of a join-query Q is one in which all the tables present in the query have no dangling tuples.

2.2 Pipelined Execution

In the following and in the rest of the paper, we assume that query plans are executed via pipelined execution, as currently adopted in state-of-the-art systems [18, 19, 23]. In this setting, the query plan is split into multiple execution pipelines. Precisely, a pipeline consists of a probe table and multiple hash tables that are the materializations of other incoming pipelines. To this end, consider the query plan in Fig. 2 (a), where we show the query plan of JOB-4a on the IMDb database [12]; the query graph of the same query is shown in Fig. 3. To ease the later formalization, we will ignore the (single-table) build pipelines, i.e., we will only consider *probe* pipelines. As a consequence, we uniquely identify a pipeline with its probe table, and say that a pipeline's (single-table) build sides have the same pipeline number as the probe table itself, e.g., `pipeline(info_type)` = `pipeline(movie_info_idx)`. With this, the query plan consists of four probe pipelines, which are marked with dotted lines in Fig. 2 (a). For instance, in the last pipeline, `pipeline(keyword)`, `keyword` probes the hash-table built over the output of `pipeline(movie_keyword)`.

Formalization. This way of executing queries naturally incurs a precedence relation between the (probe) pipelines. Namely, if a

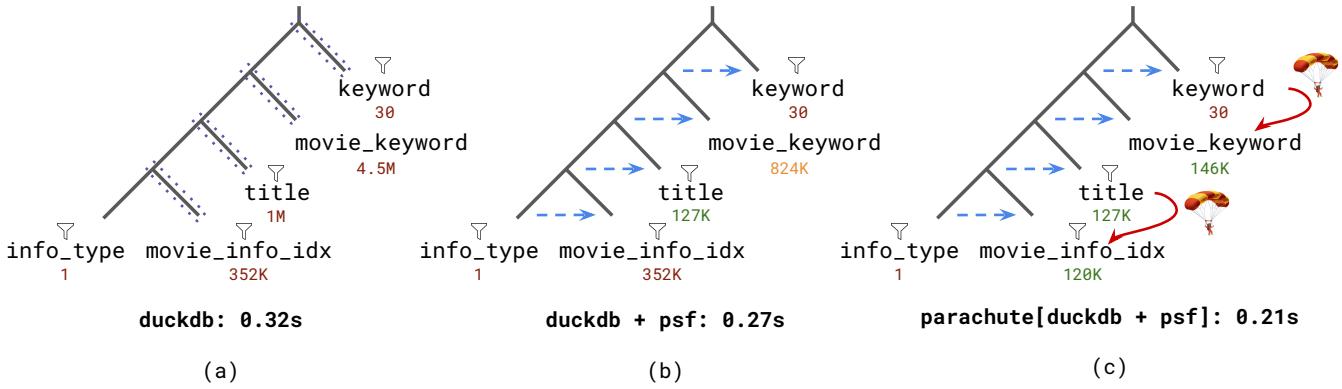

Figure 2: Visualizing how Parachute activates the downwards information flow in DuckDB's query plan of JOB-4a (v1.2.0, single-threaded). We show (a) the query plan with probe pipelines highlighted, (b) the query plan with the PSF-induced, upwards information flow, and (c) the enabled downwards information flow with `parachute` predicates.[1] Note: We follow the convention that the build side is on the left side.

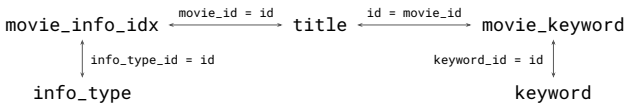

Figure 3: Query graph of JOB-4a with PK-FK relationships.

pipeline P_2 depends on the materialized output of P_1 (i.e., P_1 needs to execute before P_2), then $P_1 < P_2$. For instance, in our query plan, we have that `pipeline(title)` < `pipeline(movie_keyword)`. While common query plans tend to have this precedence relation as a total one—as is indeed the case with our query plan—this formalization allows for arbitrary ones between pipelines.

2.3 Probe-Side Filtering (PSF)

It is known that choosing the wrong build side can lead to over-sized hash tables. Probe-side filtering is a rather pragmatic way employed by production systems to deal with this issue [24, 31]: The idea is building bloom filters during *build* pipelines and pushing them into the table scans of subsequent *probe* pipelines to enable early pre-filtering. Hence, PSF mitigates the effect of having to probe large hash-tables. Indeed, consider Fig. 2 (b) where we show how PSF pushes bloom filters along the query plan. Note that DuckDB v1.2 already supports a restricted form of PSF: For small tables, such as `info_type`, it pushes an `IN`-predicate containing the qualified keys to the probe side. Our implementation of PSF in the same version enables all the other cases (see Sec. 4.1 for more details).

2.4 Information Flow

In the following, we formalize how information flows in a PSF-enhanced query plan. Intuitively, table R sends information to table S through the query plan if R is on a pipeline that eventually contributes to the build side of S's pipeline, i.e., `pipeline`(R) < `pipeline`(S), *and* if R can eventually join with S. Here, "eventually" refers to the transitive case. To understand this, note that, for instance, in the query graph of JOB-4a (Fig. 3), `info_type` cannot *directly* send information to `title`, or vice-versa: it needs `movie_info_idx` to do so. We aim to formalize this in the following.

Precedence Relation. We first define a precedence relation $\prec$ on tables, induced by the pipelined execution. Let R and S be the aforementioned tables. Then,

$$R \prec S :\Longleftrightarrow \begin{array}{l}(\texttt{pipeline}(R) < \texttt{pipeline}(S)) \\ \vee\ (\texttt{pipeline}(R) = \texttt{pipeline}(S) \wedge \texttt{is\_probe}(S)).\end{array}$$

Note that we also have to account for the case where R and S lie on the same pipeline, and S is the probe side, in which case R implicitly sends information to S (by a pipeline's definition).

Is-Joinable. Next, let $\leftrightarrow$ define the *is-joinable* relation, namely:

$$R \leftrightarrow S :\Longleftrightarrow \mathcal{A}(R) \cap \mathcal{A}(S) \neq \emptyset,$$

where $\mathcal{A}(\cdot)$ is the set of attributes of a given table (Sec. 2.1). For instance, in the same Fig. 3, it holds that `movie_keyword` $\leftrightarrow$ `keyword`, but also `title` $\leftrightarrow$ `movie_keyword`.

Information Flow. With this notation in place, we are ready to introduce the information flow relation, which we denote by $\rightsquigarrow$:

$$R \rightsquigarrow S :\Longleftrightarrow (R \prec S) \wedge (R \leftrightarrow S) \wedge \texttt{is\_probe}(S), \tag{1}$$

$$R \rightsquigarrow^{n+1} S :\Longleftrightarrow \exists T.\ R \rightsquigarrow^{n} T \rightsquigarrow S, \tag{2}$$

$$R \rightsquigarrow^{*} S :\Longleftrightarrow \exists n > 0.\ R \rightsquigarrow^{n} S. \tag{3}$$

Let us explain the last part in Eq. (1), namely `is_probe(S)`: Since PSF only pushes the bloom filters into the probe side, this is a natural addition. Yet, a more advanced PSF could in principle also push to the build sides of upcoming pipelines. In such a case, one can remove this addition. We will see yet another form of SIP, namely LIP [30], and its formalization in Sec. 6.1.

Note that the information flow is, by definition, not bound to a single hop. Indeed, we can define its transitive closure $\rightsquigarrow^{*}$, as above. Intuitively, R passes information to S if there is another T in between such that R passes information to T, and T in turn passes

[1] Note that, in Fig. 2 (c), for the cardinalities of `title` and `keyword`, we ignore the effect of DuckDB v1.2.0's internal optimizations that reduce the cardinalities of these tables further, due to the new `parachute` predicates.

information to S. Let us see two concrete examples of information flow in the query plan visualized in Fig. 2.

Simple Example. Consider tables `movie_info_idx` and `title`. It holds that `pipeline(movie_info_idx)` $\prec$ `pipeline(title)`, thus we have `movie_info_idx` $\prec$ `title`. Moreover, since table `movie_info_idx` can join with `title`, i.e., `movie_info_idx` $\leftrightarrow$ `title`, we obtain that `movie_info_idx` $\rightsquigarrow$ `title` by Eq. (1).

Transitive Example. In the same manner, `info_type` also sends information to `title`. To see why this is the case, note that `info_type` is on the same execution pipeline with `movie_info_idx`. As a result, the pushed bloom filter to `title` from the hash-table resulting from this pipeline also contains information from `info_type`.

Let us see how this works under our formalization. We want to show that `info_type` $\rightsquigarrow^2$ `title`, i.e., two-hop information passing. To this end, we fix T = `movie_info_idx`, as specified in Eq. (2). We thus have

$$\texttt{info\_type} \rightsquigarrow^2 \texttt{title} \iff \texttt{info\_type} \rightsquigarrow \texttt{movie\_info\_idx} \wedge \texttt{movie\_info\_idx} \rightsquigarrow \texttt{title}.$$

The second term has already been shown in the previous example, so we are left with showing `info_type` $\rightsquigarrow$ `movie_info_idx`. Note that `info_type` $\prec$ `movie_info_idx` since `pipeline(info_type)` = `pipeline(movie_info_idx)` and `movie_info_idx` is the probe side on this pipeline, i.e., `is_probe(movie_info_idx)`. Finally, we have `info_type` $\leftrightarrow$ `movie_info_idx`, since `movie_info_idx` has a FK on `info_type`.

PSF = Upwards Information Flow. These examples show us that the information flow induced by PSF is only in the *upward* direction. This naturally limits the query engine's ability to filter dangling tuples in the opposite, *downward* direction. With Parachute, we will also enable this very direction by leveraging precomputed join-induced fingerprint columns.

3 PARACHUTE

We have seen that PSF enables an *upwards* information flow. While this can sometimes be enough, and can indeed be achieved in a single pass over the input, with Parachute we make the case that we can also enable the reverse direction, while maintaining the single-pass guarantee.

Motivation. To motivate Parachute, consider again the query plan in Fig. 2 (b). Note that `title` has a base-table filter that could have been, in principle, used to filter `movie_info_idx`. However, this is not possible in PSF (while still being single-pass), since `title` is on a probe side. The same is true also for the last probe side in the query plan, namely `keyword`, which also has a base-table filter. However, in a PSF-enhanced query engine, this cannot be exploited. This is exactly the missing *downwards* information passing which PSF fails to induce. This is where Parachute comes in.

In this case, Parachute recognizes that the information flow is blocked between `movie_info_idx` and `title`, and `movie_keyword` and `keyword`, respectively, and introduces join-induced predicates onto these tables. These are run on auxiliary columns, which are dynamically maintained. The effect of these predicates on the base table sizes is visualized in Fig. 2 (c), showing between which pairs of base tables Parachute is active.

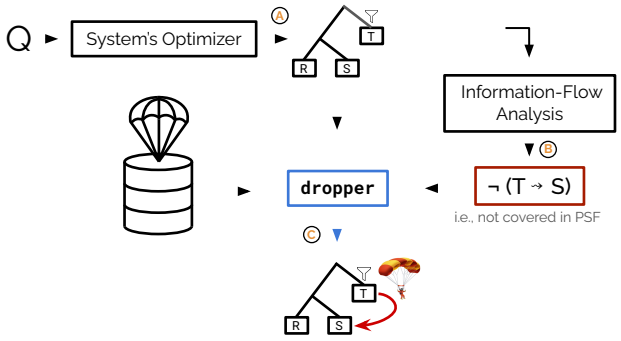

Figure 4: Overview of Parachute's workflow: The system, which is assumed to have PSF enabled, optimizes the query by outputting a query plan (Ⓐ). Our static information-flow analysis (Sec. 2.4) identifies the information flow that is not enabled by PSF (Ⓑ). We use the pre-computed parachute columns in the database to "drop" the corresponding parachute predicates on the base tables (Ⓒ).

Overview. In Fig. 4, we overview Parachute's workflow. Query Q is sent to the default system's optimizer, which outputs a query plan (Ⓐ). System's query engine is responsible to inform Parachute whether it supports any instantiation of SIP, such as PSF (see Sec. 6.1 for more variants). Then, Parachute runs a static information-flow analysis which outputs a list of ordered table-pairs $(source, target)$ through which information is not flowing in the backwards direction (Ⓑ). Using this list and the pre-computed and dynamically maintained parachute columns on FK-tables, if the *source* table—in our case, table T—has a base filter, it "drops" a parachute predicate on the *target* table, S (Ⓒ). The updated query plan can then be run by the system's engine.

Before we delve into Parachute's design, let us consider a simple example that will motivate the following subsections.

Example. Let us assume Q is the following query:

```
select k.keyword
from keyword k, movie_keyword mk, title t
where k.id = mk.keyword_id and mk.movie_id = t.id
and k.keyword like 'Family%'
and t.production_year between 1980 and 2010;
```

and set R := `keyword`, S := `movie_keyword`, and T := `title` in the query plan of Fig. 4. In this setting, the static information-flow analyzer will inform Parachute that `title` $\not\rightsquigarrow$ `movie_keyword`. If `movie_keyword` has a pre-computed `parachute` column for `title`'s `production_year` column, we can translate the above base filter on `title.production_year` in the above query to one adapted for this parachute column. This will be the focus of Sec. 3.2.

3.1 Static Information-Flow Analysis

Our static information-flow analyzer decides between which tables the information flow is not activated in the *downwards* direction. In particular, we will focus on cases such as PK-table $\not\rightsquigarrow$

FK-table pairs, since the parachute columns are present on FK-tables only. Moreover, we can activate a parachute predicate on the FK-table only if the PK-table had one. For instance, in the query plan of JOB-4a (Fig. 2), this was the case for the table-pairs (title, movie_info_idx) and (keyword, movie_keyword).

The analyzer takes as input the query plan and builds the precedence relation, as explained in Sec. 2.4. Then, based on schema (or workload) information, it also builds the *is-joinable* relation (recall Sec. 2.4). Subsequently, it constructs the information flow relation, and its transitive closure $\rightsquigarrow^*$. These constructs can be represented as a binary matrix indexed by the (unique) aliases in the query plan. Finally, it reports the pairs $(source, target)$ for which $source \not\rightsquigarrow^* target$, and $target$ has a FK on R.

3.2 Parachute Column Representation

We now come to the way how we translate columns of arbitrary data types into parachute columns. Note that, for sake of simplicity, we focus for now on the representation of a parachute column. We will come to how to "attach" this representation to joining tables in Sec. 3.3. In the following, it simply suffices to mention that we target a representation in pbw-many bits, where pbw is the bit-width of the parachute column (independent of the data type of the original column). The goal is to have a condensed representation of a column $C_i \in \mathcal{C}$ such that the translation P' of any predicate P on this column does not generate any false negatives; of course, the condensed column should also take less space. Formally, let I be the set of tuples that qualify for the predicate P; similarly, I' for P'. Then, it should hold that $I \subseteq I'$, i.e., no qualifying tuples should be skipped. This is particularly important when we scan the table that the parachute column is attached to.

In the following, we make the differentiation between numeric- and string-valued columns and show how to support each type. Indeed, numeric columns are easy to handle, e.g., using histograms. On the other hand, supporting string-valued columns, in particular LIKE is much more challenging. We will denote the condensed column as C_i'.

Distribution Estimation. Before we start, let us understand how we get an estimate for the value distribution of an *attached* parachute column, e.g., title.production_year on cast_info. This will be helpful in modelling the later histograms, which is one possible representation. To this end, we take a random sample of $m = 10^4$ tuples and join it with the PK-table. This helps Parachute understand which values should have, for instance, their own histogram bin, and which can share their bin with other values. This has the ultimate goal of reducing the false positive rate. For example, it is unwise to place two frequent values into the same bin, since when filtering for one, we will not skip the tuples of the other, leading to more (unwanted) false positives. We will thus assume that we have this distribution sample provided for low-cardinality columns. Moreover, values not represented in the sample are assign a default frequency of 1.

3.2.1 Supporting Numeric Columns. Assuming we use histograms for numeric columns, let us start with their predicate translation, since this will motivate the *type* of histograms we will be using.

Numeric Predicate Translation. Let P be a predicate of type $C_i = a$, where a is a numeric constant. Using histograms, this can be translated to $C_i' = \texttt{bin}(a)$, where bin maps a value to the corresponding bin. A similar approach can be applied for range predicates like C_i between a and b. The corresponding translation of this predicate is C_i' between bin(a) and bin(b). Let us exemplify these translations.

Example: Numeric Predicate Translation

Consider the range predicate production_year < 2025. If constant 2025 falls into the 7th bin, then the translated predicate reads production_year' ≤ 7. Note that we *need* a "≤", since the 7th bin may contain years that are less than 2025.

Value Representation. The type of histogram we choose has to be responsive to a variety of predicates. To this end, we use *equi-depth* histograms and optimize them directly on the sample using an optimized dynamic programming in $\mathcal{O}(2^{\text{pbw}} m \log m)$-time.

As always, we also need to take into account NULL values (if the respective column C_i is nullable). This can be done by reserving the smallest bin in the histogram for them.

Numeric UDFs. Naturally, this histogram representation does not allow for arbitrary predicates, e.g., is_leap(year). Note, however, that if the number of distinct values in the column is small, we can assign each distinct value to a bin and support arbitrary predicates. Indeed, this can still be represented by (equi-depth) histograms: we simply set the bin-boundaries to the distinct values.

3.2.2 Supporting String Columns. Parachute's design cannot be regarded as realistic if string columns are not supported. The temptation of discarding (or postponing) string support in research is great, however, it is now known that string data is the most prominent in production workloads [25]. In the following, we make the distinction between support for low-cardinality and high-cardinality string columns.

Low-Cardinality Regime. Assume that we have a low-cardinality string column. If its cardinality is $\leq 2^{\text{pbw}}$, we can assign a histogram bin to each unique value (including NULL, if the column is nullable). When translating a predicate, we simply look up the value in the histogram and take the corresponding bin index. Note that this case is usually covered by SIP / PSF, since this corresponds to a predicate on a dimension table.

A more interesting case, still in the low-cardinality regime, is when the cardinality is slightly larger than 2^{pbw}; this happens for small pbw. We can still apply this technique, yet in an approximate sense. Namely, we build an equi-depth histogram over the value distribution and store a mapping indexing the values to their assigned bins. In this case, a bin will contain more values, yet a frequent one has the chance to be assigned its own bin.

String UDFs. By the design of a parachute column, we can indeed support UDFs for the low-cardinality regime. Consider, for instance, the following predicate (assuming a low-cardinality title column, for the sake of the example) [21]:

```
where LLM(f'Is {t.title} a boring movie?')
```

We can call the LLM on each distinct value, gather the bin indexes corresponding to an affirmative answer, and build an IN-predicate with the set of bin indexes selected.

High-Cardinality. We show how to support translations of predicates on string-valued columns. This is in particular interesting to the research line on partition skipping. In particular, Parachute supports predicates such as C_i `like '%pattern%'`, and, by extension, `'%pattern1%pattern2%'`. Naturally, this also covers prefix- and suffix-based predicates such as `'%pattern'` and `'pattern%'`, respectively. The next subsections offer a glimpse on Parachute's support for arbitrary LIKE predicates.

String Fingerprints. The main observation is that a pattern has the chance to match a string value if the (multi-)set of characters present in the pattern is a subset of that of the string itself. To understand this, let us consider the pattern `'%utn%'` and the following string values of a column: `'utn'`, `'nutella'`, `'tone'`. When computing the sets of characters of each string, we obtain {'n', 't', 'u'}, {'a', 'e', 'l', 'n', 't', 'u'}, and {'e', 'n', 'o', 't'}, respectively. A first observation is that the first and second strings *could* qualify for the pattern, while the latter cannot, since the character set of the pattern {'u', 't', 'n'} is not a subset of {'t', 'o', 'n', 'e'}. On the other hand, even though the second string value qualifies, it is indeed a false positive. In particular, note that this representation cannot generate false negatives.

The next question is *how* to compactly represent such sets of characters, while supporting arbitrary languages. If we were to support English only, a 128-sized bitmap per word would suffice: We simply use the ASCII value of a letter and set the corresponding bit to 1. Before we come to how to reduce the space usage even further, let us first discuss how string predicates are actually translated to this representation. This will motivate the upcoming paragraphs.

String Predicate Translation. Let P := C_i `LIKE '%pattern%'`, where `pattern` is a string constant (recall that this is just a primitive; see the above discussion for the predicate types we support). We first convert the pattern to its character set representation (without the SQL-specific characters `'%'` and `'_'`), represented as a 128-sized bitmap `pmask`. As argued before, a string value has a chance to qualify for this pattern if C_i' & `pmask = pmask`, i.e., subset check at fingerprint level. To support `ILIKE` predicates, build `pmask` by OR-ing out the masks corresponding to `pattern.lower()` and `pattern.upper()`, respectively. We will come back with an example when our more compact representation is in place.

Byte Partitions. Our simplistic 128-bit bitmap would naturally lead to an increase in the space usage of a parachute column. Indeed, we would require an extra of 16 bytes per tuple. Ideally, we would have a mechanism to further reduce the space usage of a string parachute column to `pbw` bits, where `pbw` is the bit-width we also fixed for the numeric ones. Probably not enough, but do recall that we have to support arbitrary UTF-8 codes as found in current benchmark data, e.g., the IMDb dataset naturally contains names of actors with non-English characters. Understandably, such codes cannot be covered anymore by our 128-bit bitmap. Let us first fix this issue. Note that if we switch to the UTF-8 byte-representation of a string, we are again in a reduced byte space of $[0, 255]$. Assuming *valid* UTF-8 codes, we can apply the same logic from above on this very byte-representation. In other words, a byte, even if part of a UTF-8 code, is treated as a standalone byte that can be added to the fingerprint. This allows to work solely within the byte space $[0, 255]$. Next, we fix the space usage issue.

We argue that partitioning the byte space into cleverly-selected `pbw`-many clusters solves our problem. Namely, a byte is assigned to a unique cluster. When fingerprinting a string, we (a) iterate its UTF-8 byte-representation, (b) lookup its cluster, (c) set a 1 to the corresponding cluster-bin. Let us exemplify this construction.

Example. Consider the example illustrated in Fig. 5 for the string value `'nutella'`. In the case of `pbw = 4`, there are four bins available, corresponding to the clusters which the $[0, 255]$ byte space has been partitioned into. The set of letters appearing in this word are {'a', 'e', 'l', 'n', 't', 'u'}. The letters 'a', 'e', and 'u' fall into the first cluster, while the others in the third cluster. Hence, the final fingerprint reads `1010`.

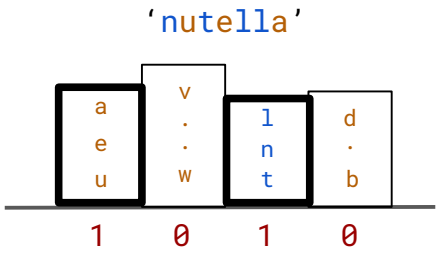

Figure 5: Parachute's string fingerprints: A string value is transformed into its corresponding fingerprint. The byte space—such as the ASCII table—is clustered into four bins, corresponding to four *disjoint* byte clusters.

The previous example already highlights an aspect that we have not yet optimized for: We have to ensure that a fingerprint does not have too many ones, otherwise the pattern mask `pmask` would not skip any tuples. This leads us to a rather intriguing optimization problem that we entitle BytePartitioning, defined below.

Definition 3.1 (BytePartitioning). Given a list of $[0, 255]$-valued strings and a parameter k, partition the byte space into k clusters such that the number of ones in the fingerprints is minimized.

We leave it as future work to build these partitions optimally. Currently, we assume a uniform distribution over the bytes, i.e., we use a Round-Robin strategy to build the partitions.

REGEX-Predicates. We can also support a limited number of regex classes via this design. Consider the following example:

```
where t.title ~ 'house(keeping|work)?'.
```

In this (admittedly very restrictive) setting, we can resort to an IN-predicate with three values, namely `'house'`, `'housekeeping'`, and `'housework'`. Hence, such regular expressions can be translated by enumerating the induced string values.

3.3 Attaching to Joining Tables

So far we only discussed how to *represent* parachute columns, at least at a conceptual level – notably, we did not discuss how these are stored and, probably more important, *where*.

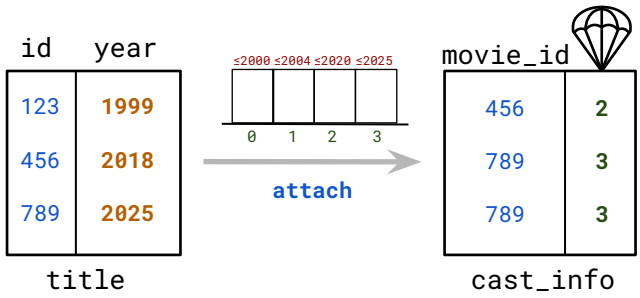

Figure 6: Attaching `title.production_year` to `cast_info`. In this case, we have a parachute bit-width of two, hence four (equi-depth) histogram bins over the domain.

Indeed, Parachute stores its join-induced fingerprints on FK-tables. Let us take a simple example from the IMDb schema [12], namely, attaching a parachute on `cast_info` from `title` on its `production_year` column, visualized in Fig. 6. Table `cast_info` stores the casting information and has a FK from `title`, which stores the actual movies. Since `production_year` is a numeric column, we will build according to Sec. 3.2.1 an equi-depth histogram for it. As each tuple from `cast_info` corresponds to exactly one tuple from `title`, we can encode the histogram bin in `pbw` bits.

Example. Concretely, in the example of Fig. 6, the first tuple from `cast_info` has `movie_id` = 456, so it will take the `production_year` of the second tuple in `title`. Its value is mapped to the third bin in the histogram (hence, 2 in 0-based notation), and set in `cast_info`.

Note that we assumed that PK-FK-constraints are preserved. We show how to allow for looser restrictions, as found in today's data warehouses such as Redshift, in Sec. 5.

General Case. In fact, this very example already showcases how we deal with attaching the parachute column of C_i to FK-tables. We can perform a join between the two tables, during which we introduce a new column that represents the corresponding value of the condensed column C_i'. For instance, the `attach` operation of `title.production_year` onto `cast_info` reads as follows:

```
update cast_info
set cast_info.parachute_title_production_year = (
  case
    when title.production_year <= 2000 then 0
    when title.production_year <= 2004 then 1
    when title.production_year <= 2020 then 2
    else 3
  end)
where title.id = cast_info.movie_id;
```

Note that if we have to attach multiple parachute columns at once, we will do so in the same `UPDATE` statement.

This construction readily applies to both numeric- and string-valued columns. To slightly boost this construction for strings, note that we rely on "helper" columns: We first build the parachute "locally", i.e., on the base table, and then attach it to the FK-table as is. This saves time due to the fact that we do not build the string fingerprints on the larger table (the FK-table), but rather on the smaller one.

3.3.1 Transitivity. A feature of Parachute is that it can be extended to go beyond neighboring, joining tables, and even on *different* join keys. This indeed happens in the IMDb schema, where we have a PK-chain going from `cast_info` → `title` → `kind_type`. In such cases, we can have a `parachute` column attached to `cast_info` which comes from `kind_type`. To support this in the build phase, we have to perform a topological sort on the schema-induced DAG, such that the parachute column on `title` is already available when building for `cast_info`. Hence, the `attach` operation may consider already present parachutes on the PK-table. However, since such PK-chains start at dimension tables, which are intrinsically covered by PSF, such parachute columns never get to be used.

3.4 Inserts and Updates

We outline Parachute's support for inserts and updates.

Inserts. If a batch of new data B is inserted into the FK-table, we need to compute the parachute column only for B. To this end, we perform a join of B with the PK-table to attach the respective parachute column on the new slice of the table. We evaluate this overhead in Sec. 4.4. Note that since the join intrinsically performs histogram lookups, in case the histogram is not updated, we will map the values larger than the maximum histogram boundary into to the last bin; similarly for the smaller values than the minimum histogram boundary. In case the histogram lost its equi-depth guarantee by a given threshold, we only need to re-attach, hence re-compute, the corresponding parachute column on the FK-table. In case an insert happens on the PK-table, this is a noop.

Updates. The more interesting case are updates, since these may happen on the PK-table. Let us assume that we modified the title of a movie in table `title`, and there is a corresponding parachute column on `cast_info`. Then, we have to re-compute for the join-partners in `cast_info` the parachute values.

4 EVALUATION

Setup. We conduct the experiments on a single node Intel® Xeon® Gold 5318Y CPU (24 cores, 48 hyper-threads). The machine is equipped with 128GB DDR4 main memory and runs Ubuntu 24.04. All experiments are run single-threaded due to DuckDB v1.2's limited supported for multi-threaded `UPDATE` statements.

Benchmarks. We evaluate on the JOB [12] and CEB [17] benchmarks. JOB has 113 queries of 33 templates on the IMDb database, while CEB has a total of 13,646 queries on the same database, yet with a much more variety of base table filters. All benchmarks are run on hot data, following the guidelines in Ref. [22].

Competitors. Our PSF implementation (`duckdb+psf`) is done on DuckDB v1.2.0 (`duckdb`) and is described thoroughly in Sec. 4.1. Moreover, we also consider the recent Robust Predicate Transfer (`duckdb+rpt`) [29], which has been implemented on top of DuckDB v0.9.2.[2] Therefore, for a complete comparison, we also run Parachute on DuckDB v0.9.2 when comparing with RPT.[3] Note that, starting with v1.2, DuckDB indeed adopted a lightweight SIP

[2] https://github.com/embryo-labs/Robust-Predicate-Transfer (retrieved June 1, 2025)
[3] We are aware of an ongoing work to integrate RPT in v1.2.

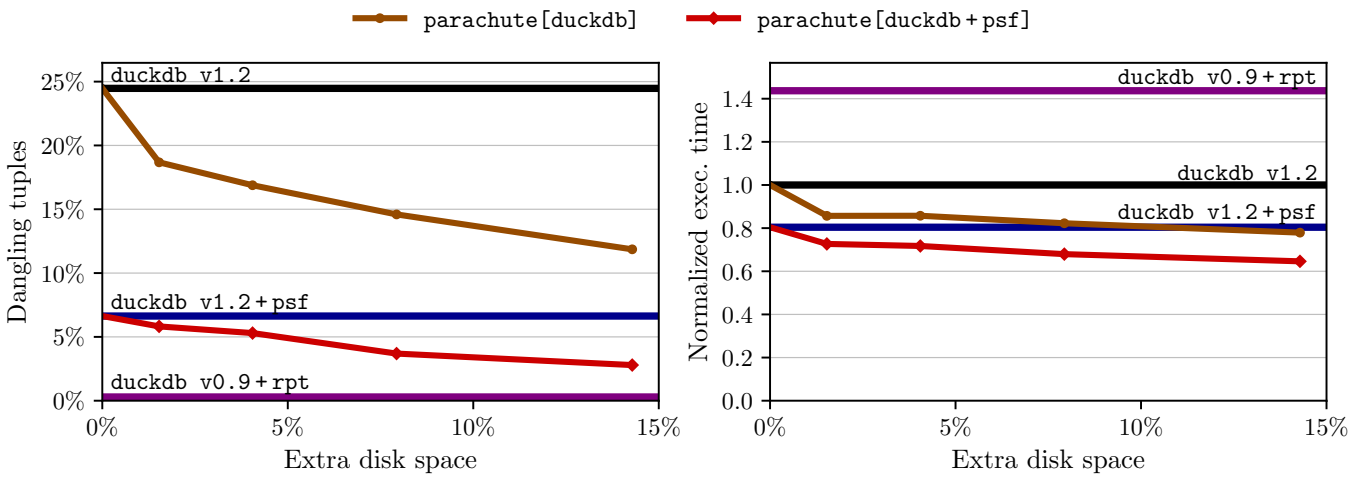

Figure 7: The effect of using Parachute on JOB [12]. We compare with DuckDB v1.2 (`duckdb v1.2`), our PSF implementation in the same version (`duckdb v1.2 + psf`), and the recent RPT (`duckdb v0.9 + rpt`) [29]. While `parachute[duckdb]` enables both the up- and downwards (↕) information flow, `parachute[duckdb+psf]` enables only the missing downwards (↓) direction, since PSF already guarantees the upwards one (cf. Sec. 2.4).

implementation, as described in Sec. 4.1. For brevity, we will omit version numbers when the context makes them clear.

Let us describe how Parachute is integrated with `duckdb` and `duckdb+psf`: Each `parachute` column is a non-nullable integer column of up to `pbw` bits. String fingerprints are computed using `duckdb`'s support for user-defined Python functions. To keep the query plans unchanged if a parachute predicate occurs in the SQL query, we modified `duckdb` to ignore cardinality estimates for base tables and non-pushed base-table filters. Then, a Python framework maintains a catalog with the existing parachute columns and, based on the result of the static analyzer (Sec. 3.1), drops the corresponding `parachute` predicates in the SQL query text. We set `pbw` $\in \{2, 4, 8, 16\}$ and consider three types of applying Parachute: `parachute`↑, `parachute`↓, and `parachute`↕, which corresponds to enabling the upwards, downwards, and bi-directional information flow, respectively. We only use one-hop parachute columns in the experiments, i.e., no transitive parachutes.

4.1 Sideways Information Passing in DuckDB

DuckDB v1.2 [23] implements some form of fundamental sideways-information-passing (SIP) for filtering tuples on the probe sides of hash-joins (PSF). In the general case, DuckDB transfers the min/max values of the key column(s) to the probe side and uses these values for coarse-grained pruning and filtering. Additionally, two forms of SIP are only applied only if the number of distinct join-keys on the build side is small. More specifically, if there is only one value on the build side, an equality predicate on the key-column(s) of the join is generated and transferred to the probe side's base table. Similarly, if the number of distinct keys on the build side is below a certain threshold (default: 50), an `IN`-list filter is pushed to the probe side's base table for partition-level pruning, but is not used for row-level filtering. For build sides exceeding the threshold for the number of distinct keys, only min/max transfer is used.[4]

To establish a state-of-the-art baseline for PSF, we implemented bloom joins [14, 16] into DuckDB to perform probe-side filtering on a row-level. Our implementation provides an end-to-end speedup of 1.26x over vanilla DuckDB on the JOB benchmark.

Implementation. We implemented a custom bloom filter that is tightly integrated into DuckDB's query processing framework. To guarantee fast builds and lookups *without regressions*, the bloom filter's size is capped at 8 KiB, ensuring it fits into the processors L1 cache. Further, we use the 64-bit hash values from the join's hash table to generate two bit positions in the bloom filter that should be set by splitting them into their upper and lower 32 bit and applying fast modulo reduction [13] with respect to the bloom filter's size. This design corresponds to a bloom filter with $m = 2^{16}$ and $k = 2$, supporting about 5000 distinct keys with a 2% false-positive rate (*fpr*). To maintain the *fpr*, the filter is discarded after construction if more than 34% of bits are set. During construction, hash values from the hash table are reused, avoiding extra hash calculations. On the probe side, the bloom filter is evaluated inside the table scan operator directly after simple table filters. We chose to evaluate the bloom filter after table filters based on the assumption that table filters are faster to evaluate. If the Bloom filter is insufficiently selective, i.e., filtering less than 60% of rows after 4000 processed rows, it is automatically disabled.

4.2 JOB & CEB Results

In principle, Parachute can infer from the schema which columns can be attached. However, using workload information, one can restrict the number of columns used. Given that query repetitiveness in real workloads is high [25], one can enable Parachute once a set

[4] We refer the reader to the references in the original blog post: https://duckdb.org/2025/02/05/announcing-duckdb-120.html#optimizations (retrieved June 1, 2025).

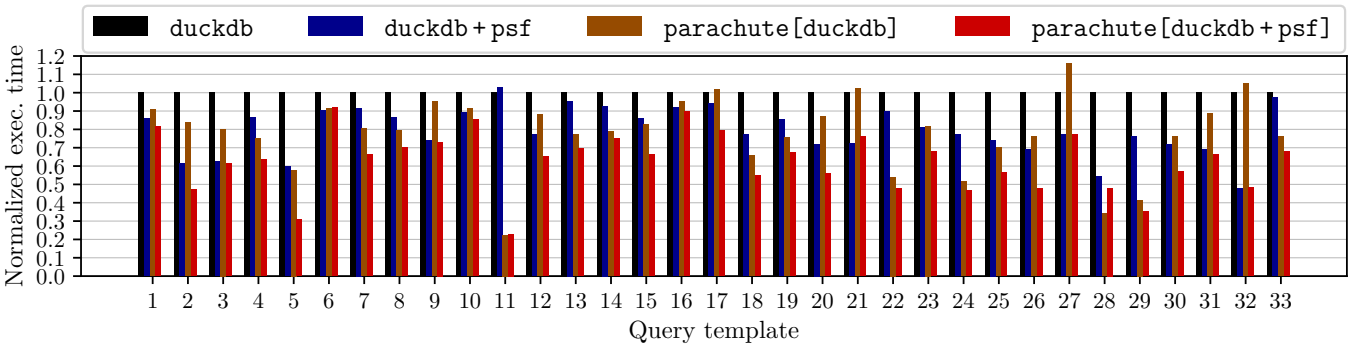

Figure 8: JOB: Normalized execution time in DuckDB v1.2 across query templates. We use a `parachute` bit-width (`pbw`) of 16 bits.

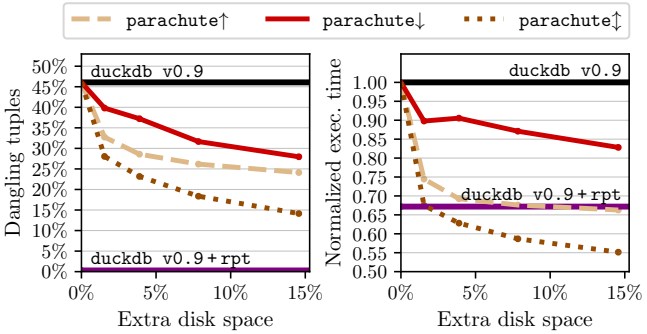

Figure 9: JOB: Evaluation of RPT [29] and different Parachute configurations (↑, ↓, ↕) on DuckDB v0.9 (`duckdb v0.9`).

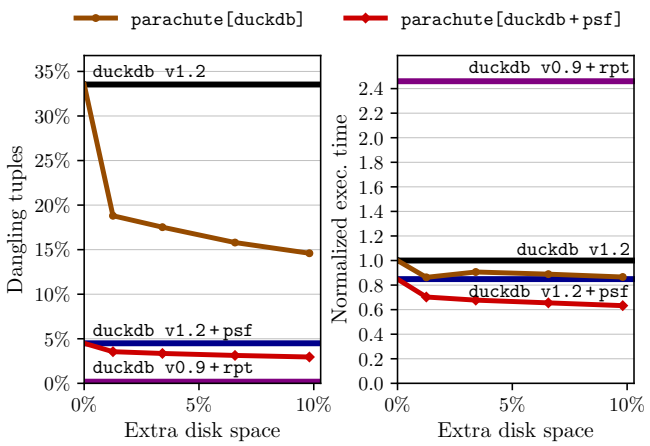

Figure 10: The effect of enabling the downwards information flow in probe-side filtering (`duckdb + psf`) on CEB [17].

of query templates is already available; we will continue this discussion in Sec. 5. To this end, we analyze JOB's and CEB's queries and build parachutes only for columns that are also filtered in combination with the respective table.

JOB. We show the number of dangling tuples (relative to total input size, excluding the non-dangling part) and the normalized execution time for the entire JOB in Fig. 7 (see Fig. 9 for an evaluation on v0.9). Let us explain what the competitors stand for: `parachute[duckdb]` refers to `parachute↕` (without PSF activated) since DuckDB v1.2 only implements a rather simplistic PSF, hence the upwards information flow is rather scarce, while `parachute[duckdb + psf]` refers to `parachute↓`, since PSF already guarantees the upwards information flow.

The first observation is that we obtain a monotonic behavior of the number of dangling tuples with increasing `pbw`. In particular, `parachute[duckdb + psf]` reaches a number of dangling tuples of 2.79%. In constrast, vanilla PSF achieves only 6.63%. The same holds for the normalized execution time (Fig. 7, right), where `parachute[duckdb + psf]` achieves a speedup of 1.54x and 1.24x over `duckdb` and `duckdb + psf`, respectively, for 14.35% extra space, or 1.47x and 1.18x, respectively, for 7.94% extra space.

Parachute vs. RPT. Notably, due to its forward and backward passes of the bloom filters, RPT (`duckdb v0.9 + rpt`) remains with the fewest number of dangling tuples (0.29%). The reason is that a `parachute` column has a higher false positive rate compared to a bloom filter due to its narrow bit-width (`pbw`). For a fair comparison, we show Parachute's effect on DuckDB v0.9.2 in Fig. 9. RPT indeed improves over vanilla DuckDB by 1.48x, while Parachute achieves a 1.81x speedup when allowed to use 16-bit `parachute` columns.

JOB Templates. In Fig. 8, we show the normalized execution across query templates in JOB with respect to DuckDB v1.2 (`duckdb`). Each query template's execution time is the sum of the execution times of its instantiations. We see that `parachute` alone is not sufficient to achieve the best performance: it needs the upwards pass guaranteed by PSF. The regressions in `parachute[duckdb]`, e.g., JOB-27*, are due to DuckDB's current limited support to push complex predicates, such as `col & pmask = pmask`, into the table scan, forcing a materialization of the filtered parachute columns into the projection operator. Recall that such predicates are coming from translated `LIKE` predicates on high-cardinality columns (Sec. 3.2.2).

CEB. The same trend can be observed in CEB, the results of which are shown in Fig. 10. Since there are less parachute columns to maintain (see the next paragraph for the exact numbers for JOB and CEB), we can achieve the same numbers with less extra space: With 9.82% extra space, `parachute[duckdb + psf]` reaches 2.96% dangling tuples, and a speedup of 1.56x and 1.33x over `duckdb`

Table 1: Building `parachute` columns on the IMDb database for JOB and CEB: We report for each parachute bit-width `pbw` $\in \{2, 4, 8, 16\}$ how much time it takes to compute the parachute columns for the entire database and the extra space needed. Note that DuckDB v1.2 performs `UPDATE` statements single-threaded.

IMDb duckdb's `load`: 91.67s				
	JOB: 32 parachutes		CEB: 20 parachutes	
pbw	Attach time	Extra space	Attach time	Extra space
2	244.70s	+1.53%	184.38s	+1.25%
4	248.44s	+4.03%	187.11s	+3.42%
8	247.61s	+7.94%	188.47s	+6.55%
16	265.85s	+14.35%	190.18s	+9.82%

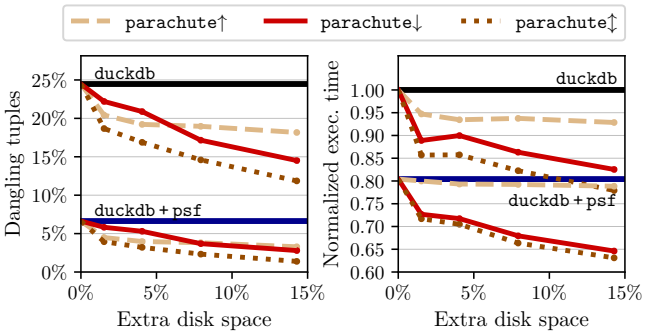

Figure 11: Analyzing different information flow directions (↑, ↓, ↕) for Parachute on `duckdb` and `duckdb + psf` in JOB.

and `duckdb + psf`, respectively. In contrast, RPT's strategy removes almost all dangling tuples on this workload (0.18%).

Size & Build Time. Naturally, computing parachute columns incurs some overhead. We report the total time to compute them, for various values of `pbw` $\in \{2, 4, 8, 16\}$ for both JOB and CEB in Tab. 1. For comparison, we also report the time is takes to load the entire IMDb database into DuckDB v1.2 (single-threaded, to ensure a fair comparison). Note that classical FK-indexes on all FK-tables would require an extra space of 62.03%, assuming no index overhead and 8-byte TIDs.

4.3 Micro-Benchmark

Next, we take Parachute under the lens to understand whether our theoretical formalization is reflected in practice. Namely, we want to answer the following two questions:

Q1. Is PSF enough to guarantee upwards information flow?
Q2. Is downwards information flow what we should strive for?

To this end, consider Fig. 11, in which we enable Parachute in different scenarios: The symbol ↑ represents the upwards information flow (as in PSF), while ↓ represents the downwards information flow (the one missing in PSF), and ↕ the activation of both directions.

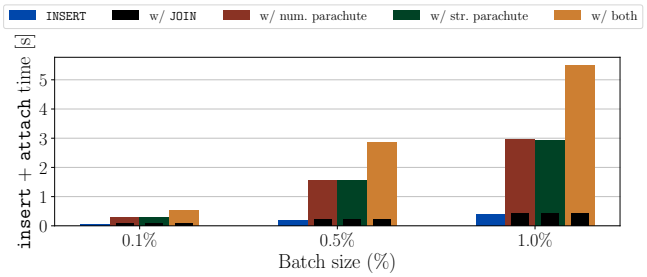

Figure 12: Inserts: We insert batches of various size percentages $\in \{0.1\%, 0.5\%, 1\%\}$ of the original table size, in this case, `cast_info` (36M tuples). We benchmark on the Parachute-enhanced database with `pbw` = 4 and string helper columns. We stress that DuckDB v1.2 performs `UPDATE` statements single-threaded.

Q1. We observe that `parachute↑`, while it still decreases its number of dangling tuples, it *cannot* compete with PSF, due to the low false-positive rate achieved by bloom filters. This is also reflected in the plot of the normalized execution time, where it plateaus after a certain point.

Q2. This question is more interesting, since it also implies the question whether enabling *both* directions for Parachute, even in the presence of PSF, may provide a much better pruning rate, in this case 1.38% remaining dangling tuples for `pbw` = 16. In part, this is natural, since Parachute may indeed help the bloom filter further prune dangling tuples that turn up to be false positives. However, if this is not the case, then the additional parachute predicates only slow down the query execution. In this particular case, the difference in terms of execution time between `parachute↓[duckdb + psf]` and `parachute↕[duckdb + psf]` is minimal, showing that, indeed, the downward direction is what we should eventually optimize for.

4.4 Inserts

Recently, researchers from industry urged the database community to consider how inserts and updates affect the proposed solutions [25]. Since Parachute maintains its columns on FK-tables, an insert is simply a lookup by key in the respective PK-table (which, in the absence of an index, can be done by fetching first the corresponding block and then scanning it). We showcase this overhead for batched inserts in Fig. 12, where we insert batches of various size percentages $\in \{0.1\%, 0.5\%, 1\%\}$ into IMDb's `cast_info` (36M tuples). The referenced table in this case is `title`, and the parachute columns are on `production_year` and `title` for the numeric and string case, respectively. Moreover, for the string case, we use the string-helper column maintained on the PK-side. We see that, indeed, this construction blends the difference between numeric and string columns.

Since DuckDB v1.2 runs `UPDATE` statements single-threaded—enabling multi-threading even slows them down, we also show how much the actual join between the *new* added batch and `title` actually incurs. Indeed, we see that the overhead comes from the update itself.

5 DISCUSSION & FUTURE WORK

Is Extra Space an Issue? One could argue that the extra space required by Parachute to store the parachute columns would impede its adoption in production OLAP systems. This may seem understandable at first glance, but we are actually witnessing a trend in OLAP systems to ignore the storage footprint due to its low cost. For example, Amazon Redshift recently added support for MDDL [7] as a new paradigm for sorting the table based on workload predicates. This is well motivated by the high query repetition rate seen in real customer workloads. To this end, MDDL maintains a *new* wide bitmask column, where each bit corresponds to a predicate on the base table. Hence, we do indeed see the openness of an OLAP system towards accepting new columns in its table files.

Parachute in the Cloud. There are many ways to prepare Parachute for the cloud. Notably, it comes with a clear advantage over pushed bloom filters: Since a parachute column essentially acts as a "built-in" bloom filter into the table, i.e., the parachute column is already attached to the table to be pre-filtered, one can exploit this as part of the *partition pruning* phase using zonemaps [15]. While this naturally works for numeric columns, we can apply the same strategy for string fingerprints as well (Sec. 3.2.2), e.g., by storing min/max-zonemaps and applying MDDL's skipping technique on its bitmask-columns (see Ref. [7, Lst. 3]).

PK-FK Relationships. One challenge that Parachute will need to address in the future is the lack of support for PK-FK constraints in current production systems [25]. Since we envision `parachute` columns being attached on the fly as the workload runs—e.g., building `parachute` columns for those columns accessed with a relatively high number of distinct predicates—we propose the following solution: First, possible joins across the schema can be inferred from the workload itself as Parachute tracks possible parachute column placements. What remains is to determine whether we are dealing with a PK-FK relationship or not.

A lightweight check is to compute an approximation of the exact number of distinct values of both keys using a HLL sketch [8]. To this end, note that, in principle, Parachute can easily handle *inexact* PK-FK relationships if (a) a key appears in more than one tuple on the PK-side, or (b) a key on the FK-side has no associated key on the PK-side. Case (a) can be solved by `or`-ing out the column fingerprints coming from the tuples associated with the same key. In this case, the translation logic will require minor modifications. To accommodate case (b), `parachute` columns can initially store a `NULL` value by default. The problem arises when the matching tuple is eventually inserted on the PK-side. In this case, we would maintain a set of non-matching keys on the FK-tables and update them when the join condition is met.

Parachute-Aware Optimizer. Currently, Parachute acts on the "frozen" query plan, as output by the system's optimizer. This guarantees that we can maintain the efficient upwards information flow by PSF, which also acts on the frozen query plan. In principle, it is possible to integrate parachute column estimates into the optimizer. Indeed, we can (over-)estimate the size of a PK-FK join by analyzing the translated parachute predicate on the FK-table:

$$\left|\texttt{movie\_keyword} \bowtie \sigma_{\texttt{production\_year} < 2025}(\texttt{title})\right|$$
$$\leq \left|\sigma_{\texttt{parachute\_title\_production\_year} \leq \texttt{bin}(2025)}(\texttt{movie\_keyword})\right|,$$

where `bin(2025)` is the corresponding bin of the constant 2025 in the attached parachute column (recall Fig. 6). Indeed, by using the statistics of the parachute column on `movie_keyword`, we can *overestimate* how many join partners we have from `title` with the respective table predicate. While this is trivial for a single join, it is natural to ask how it develops with multiple joins, especially when integrating with PSF; we leave this as an interesting future work. To estimate cardinalities on string fingerprints, we have to sum up the cardinalities of all supersets of the pattern fingerprint, `pmask`; formally, this is known as a (reversed) zeta transform [5].

User Experience. Currently, Parachute has a singl, `pbw`. Note that each `parachute` column, regardless of its data type, is reserved exactly `pbw` bits (which may or may not be fully used, depending on its cardinality). However, as of now, there is no upper-bound on the ultimate extra space needed. Hence, an improvement in terms of usability is to replace `pbw` with an *upper-bound* on the extra space allowed, e.g., 10%. The challenge is therefore to find optimal `pbw`-values for each individual parachute column, while staying within the upper-bound and minimizing the false positive rate.

Join-Induced Sorting. Parachute can also be considered a data-driven variant of MDDL on *join-induced* filters. Hence, although primarily designed for query processing, `parachute` columns can also be used for sorting (even jointly, i.e., considered together as a column by itself). A promising future work is to have hybrid, workload- *and* data-driven parachute columns, i.e., have bits for repeating predicates, and otherwise the standard layout for columns with a large set of distinct predicates.

Alternative Representations. Another way of representing the implicit index induced by Parachute is to use row-ranges, as done in Amazon Redshift [24]. Concretely, we will still keep the histogram representation, yet each bin has a list of row-ranges that cover the tuples of that bin. To bound the space overhead (in case the row-ranges are not clustered), we can employ a similar strategy as in predicate caching [24], to have an over-approximation of the row-ranges, at the cost of having more false positives.

6 RELATED WORK

Given the recent (and understandable) focus of the database community, both research- *and* industry-oriented, on robust query processing, we warn the reader that the related work will be rather verbose. We will not simply enlist prior work, but we will also showcase how Parachute nicely fits into the respective context. The section is split into the following subthemes: sideways-information passing (Sec. 6.1), join indexes (Sec. 6.2), instance-optimal algorithms (Sec. 6.3), and caching mechanisms (Sec. 6.4).

6.1 Sideways-Information Passing

Sideways-information passing (SIP) [6, 11] is a practical way to deal with the issue of expensive hash-tables probes, by building bloom filters on the build sides and pushing them into the upcoming pipelines. Across the paper, we argued that production systems implement a more restrictive form of SIP, namely probe-side filtering (PSF) [24]. However, there is another extension, which aims to optimize the order in which the pushed bloom filters are probed.

LIP. Lookahead-Information-Passing (LIP) [30] makes the case that the order in which the pushed bloom filters are probed does indeed make a difference. Initially, LIP was designed to work on a star schema, yet it is now employed more generally within a single pipeline [28] and has already been adopted in SQLite [9]. Let us see a formalization of this paradigm for the case where only build sides push to the probe side of the same pipeline. We have:

$$R \rightsquigarrow S :\Leftrightarrow (\texttt{pipeline}(R) = \texttt{pipeline}(S)) \wedge \texttt{is\_probe}(S),$$

since $R \leftrightarrow S$ holds by the definition of an execution pipeline. Hence, Parachute is also applicable when LIP is used. In addition, Zhang et al. [28] describe yet another form of LIP, in which build sides can prune themselves. This still fits into our formalization, yet one needs to introduce a $\texttt{rank}(\cdot)$ function on the build sides of a pipeline to match their definition; we omit the details.

6.2 Join Indexes

Parachute can be regarded as a space-flexible bitmap join index [20] or, more specifically, as a generalization of the dimension-join index [4]: The dimension-join index has a bitmap on the FK-table for each value of a low-cardinality dimension column. Parachute is a generalization of this index to (a) non-dimension tables—we can support `parachute` columns coming from `title` or `keyword`, (b) high-cardinality columns: this is achieved via equi-depth histograms, which can be interpreted as a binning together the values in the dimension-join index, and (c) string support.

6.3 Instance-Optimal Algorithms

Instance-optimal algorithms also guarantee a bidirectional information flow by performing (approximate) semi-join reduction on the tables. As a consequence, they significantly reduce the number of dangling tuples. However, this strategy involves multiple passes over the data. Next, we outline several such approaches.

Yannakakis' algorithm. A prominent example is Yannakakis' algorithm [27], which achieves full semi-join reduction, assuming an α-acyclic query. It is composed of two phases: (a) the semi-join phase, and (b) the join phase. The full semi-join reduction is achieved in phase (a). In turn, it consists of a forward and a backward pass. In the forward pass, the algorithm traverses the join tree bottom-up, by performing a semi-join on the current table with its (already reduced) children. In the backward pass, the algorithm iterates the join tree in the reversed direction. Finally, it is guaranteed that the base tables have no dangling tuples. Hence, the join phase can be executed using vanilla binary joins, as the intermediate sizes are guaranteed to not exceed their input's size.

(Robust) Predicate Transfer. Admittedly, Yannakakis' algorithm has the expensive *exact* semi-join phase. Indeed, having an *approximate* semi-join reduction is enough. A first effort in this direction Predicate Transfer (PT) [26], which mimics Yannakakis' forward-backward information flow paradigm yet in an approximate sense, by resorting to bloom filters. Namely, PT introduces a transfer schedule of the bloom filters before the actual query plan is executed. However, despite its promising results, PT fails to provide robustness guarantees for α-acyclic queries [29]. This issue was recently addressed in RPT [29], which ensures a robust transfer schedule. Nevertheless, RPT still performs multiple passes over the data: the forward and backward passes of the bloom filters, and the (implicit) execution of the hash-joins. Parachute bypasses the need for the two additional passes by moving the overhead to storage.

Lookup & Expand. Another promising paradigm is Lookup & Expand (L&E) by Birler et al. [2], and the later refinement by Bekkers [1] for a more restricted class of α-acyclic queries. The main idea is to split the hash-join operator into two intrinsically distinct operators, lookup and expand, that can be freely reordered in the query plan. L&E may still benefit from `parachute` columns in the case of non-α-acyclic queries.

6.4 Caching Mechanisms

In recent work, it has been pointed that queries in real workloads are highly repetitive [24, 25]. This led many systems rethink how they approach such queries. Until recently, the state-of-the-art approach was a rather heavyweight materialization of query results, materialized views. Redshift shifted towards considering a predicate cache [24] instead, where the first-class citizen is not the data itself anymore, as in the case of materialized views, but a lightweight representation of the row indexes that are to be scanned. This makes updates much more easy-to-handle. The main primitive is a cache that holds table predicates and even pushed semi-join filters. This integrates well with Parachute since one can now also store the *translated* parachute predicates. Since `parachute` predicates come from PK-tables, which are less frequently updated than FK-tables, this increases the chance that they remain in the predicate cache.

7 CONCLUSION

In this work, we have made the case that current implementations of sideways information passing lack a bi-directional information flow, which instance-optimal algorithms de facto guarantee. In turn, the latter, including old proposals such as Yannakakis' algorithm [27], and even more recent refinements [26, 29], incur several passes over the data, which impedes their adoption in real systems. Motivated by the openness of production OLAP systems towards storing succinct auxiliary columns to improve query performance, e.g., MDDL [7], we introduce Parachute as precomputed join-induced fingerprint columns that enable bi-directional information flow in a single pass. To achieve this, we propose several column representations for common data types, particularly focusing on string data, which is known to be the most frequent data type in data warehouses [25].

Outlook. Parachute has a strong advantage over dynamically pushed bloom filters, which makes it suitable to the cloud setting: Given that parachute columns are attached directly to the table, they can be leveraged during partition pruning. Moreover, Parachute can also blend well with recent caching paradigms, such as Predicate Caching [24]: Apart from caching semi-join filters, one can now also store the translated parachute predicates, enabling build-side pruning if the query plan contains a FK-table on the build side. In the face of workload changes where the cache has to be invalidated, we envision Parachute's data-driven design as a robust mechanism for dealing with workload changes.

REFERENCES

[1] Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. 2024. Instance-Optimal Acyclic Join Processing Without Regret: Engineering the Yannakakis Algorithm in Column Stores. arXiv:2411.04042 [cs.DB] https://arxiv.org/abs/2411.04042

[2] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *Proc. VLDB Endow.* 17, 11 (2024), 3215–3228. https://doi.org/10.14778/3681954.3681995

[3] Altan Birler, Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2024. Simple, Efficient, and Robust Hash Tables for Join Processing. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*, Carsten Binnig and Nesime Tatbul (Eds.). ACM, 4:1–4:9. https://doi.org/10.1145/3662010.3663442

[4] Pedro Bizarro and Henrique Madeira. 2001. The Dimension-Join: A New Index for Data Warehouses. In *XVI Simpósio Brasileiro de Banco de Dados, 1-3 Outubro 2001, Rio de Janeiro, Brasil, Anais/Proceedings*, Marta Mattoso and Geraldo Xexéo (Eds.). COPPE/UFRJ, 259–273.

[5] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. 2007. Fourier meets möbius: fast subset convolution. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, David S. Johnson and Uriel Feige (Eds.). ACM, 67–74. https://doi.org/10.1145/1250790.1250801

[6] Ming-Syan Chen, Hui-I Hsiao, and Philip S. Yu. 1993. Applying Hash Filters to Improving the Execution of Bushy Trees. In *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, Rakesh Agrawal, Seán Baker, and David A. Bell (Eds.). Morgan Kaufmann, 505–516. http://www.vldb.org/conf/1993/P505.PDF

[7] Jialin Ding, Matt Abrams, Sanghita Bandyopadhyay, Luciano Di Palma, Yanzhu Ji, Davide Pagano, Gopal Paliwal, Panos Parchas, Pascal Pfeil, Orestis Polychroniou, Gaurav Saxena, Aamer Shah, Amina Voloder, Sherry Xiao, Davis Zhang, and Tim Kraska. 2024. Automated Multidimensional Data Layouts in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 55–67. https://doi.org/10.1145/3626246.3653379

[8] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *DMTCS Proceedings (DMTCS Proceedings)*, Philippe Jacquet (Ed.), Vol. DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07). Discrete Mathematics and Theoretical Computer Science, Juan les Pins, France, 137–156. https://doi.org/10.46298/dmtcs.3545

[9] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. 2022. SQLite: Past, Present, and Future. *Proc. VLDB Endow.* 15, 12 (2022), 3535–3547. https://doi.org/10.14778/3554821.3554842

[10] Paul Gross, Daniël ten Wolde, and Peter Boncz. 2025. Adaptive Factorization Using Linear-Chained Hash Tables. In *Proceedings of the Conference on Innovative Data Systems Research*.

[11] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). IEEE Computer Society, 774–783. https://doi.org/10.1109/ICDE.2008.4497486

[12] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[13] Daniel Lemire. 2016. *A fast alternative to the modulo reduction.* https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/ Retrieved: March 1, 2025.

[14] Lothar F. Mackert and Guy M. Lohman. 1986. R* Optimizer Validation and Performance Evaluation for Local Queries. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, Carlo Zaniolo (Ed.). ACM Press, 84–95. https://doi.org/10.1145/16894.16863

[15] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 476–487. http://www.vldb.org/conf/1998/p476.pdf

[16] James K. Mullin. 1990. Optimal Semijoins for Distributed Database Systems. *IEEE Trans. Software Eng.* 16, 5 (1990), 558–560. https://doi.org/10.1109/32.52778

[17] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.* 14, 11 (2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[18] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[19] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[20] Patrick E. O'Neil and Goetz Graefe. 1995. Multi-Table Joins Through Bitmapped Join Indices. *SIGMOD Rec.* 24, 3 (1995), 8–11. https://doi.org/10.1145/211990.212001

[21] Liana Patel, Siddharth Jha, Carlos Guestrin, and Matei Zaharia. 2024. LOTUS: Enabling Semantic Queries with LLMs Over Tables of Unstructured and Structured Data. *CoRR* abs/2407.11418 (2024). https://doi.org/10.48550/ARXIV.2407.11418 arXiv:2407.11418

[22] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, Alexander Böhm and Tilmann Rabl (Eds.). ACM, 2:1–2:6. https://doi.org/10.1145/3209950.3209955

[23] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[24] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 347–359. https://doi.org/10.1145/3626246.3653395

[25] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC Is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[26] Yifei Yang, Hangdong Zhao, Xiangyao Yu, and Paraschos Koutris. 2024. 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. https://www.cidrdb.org/cidr2024/papers/p22-yang.pdf

[27] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 82–94.

[28] Yunjia Zhang, Yannis Chronis, Jignesh M. Patel, and Theodoros Rekatsinas. 2023. Simple Adaptive Query Processing vs. Learned Query Optimizers: Observations and Analysis. *Proc. VLDB Endow.* 16, 11 (2023), 2962–2975. https://doi.org/10.14778/3611479.3611501

[29] Junyi Zhao, Kai Su, Yifei Yang, Xiangyao Yu, Paraschos Koutris, and Huanchen Zhang. 2025. Debunking the Myth of Join Ordering: Toward Robust SQL Analytics. arXiv:2502.15181 [cs.DB] https://arxiv.org/abs/2502.15181

[30] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M. Patel. 2017. Looking Ahead Makes Query Plans Robust. *Proc. VLDB Endow.* 10, 8 (2017), 889–900. https://doi.org/10.14778/3090163.3090167

[31] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. 2012. Vectorwise: A Vectorized Analytical DBMS. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 1349–1350. https://doi.org/10.1109/ICDE.2012.148