Wred: Workload Reduction for Scalable Index Tuning

MATTEO BRUCATO, Microsoft Research, USA
TARIQUE SIDDIQUI, Microsoft Research, USA
WENTAO WU, Microsoft Research, USA
VIVEK NARASAYYA, Microsoft Research, USA
SURAJIT CHAUDHURI, Microsoft Research, USA

Modern database systems offer index-tuning advisors that automatically identify a set of indexes to improve workload performance. Advisors leverage the optimizer's *what-if* API to optimize a query for a hypothetical index configuration. Because what-if calls constitute a major bottleneck of index tuning, existing techniques, such as workload compression, help reduce the number of what-if calls to speed up tuning. Unfortunately, even with small workloads and few what-if calls, tuning can still take hours due to the complexity of the queries (e.g., the number of joins, filters, group-by and order-by clauses), which increases their optimization time. This paper introduces *workload reduction*, a new complementary technique aimed at expediting index tuning by decreasing individual what-if call time without significantly affecting the quality of index tuning. We present an efficient workload reduction algorithm, called **Wred**, which *rewrites* each query in the original workload to eliminate column and table expressions unlikely to benefit from indexes, thereby accelerating what-if calls. We study its complexity and ability to maintain high index quality. We perform an extensive evaluation over industry benchmarks and real-world customer workloads, which shows that **Wred** results in a 3× median speedup in tuning efficiency over an industrial-strength state-of-the-art index advisor, with only a 3.7% median loss in improvement—where improvement is the total workload cost as estimated by the query optimizer—and results in up to 24.7× speedup with 1.8% improvement loss. Furthermore, combining **Wred** and **Isum** (a state-of-the-art workload compression technique for index tuning) results in higher speedups than either of the two techniques alone, with 10.5× median speedup and 5% median improvement loss.

CCS Concepts: • **Information systems → Autonomous database administration**; *Query operators.*

Additional Key Words and Phrases: Index tuning, workload compression, workload reduction, query rewriting.

ACM Reference Format:
Matteo Brucato, Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2024. Wred: Workload Reduction for Scalable Index Tuning. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 50 (February 2024), 26 pages. https://doi.org/10.1145/3639305

1 INTRODUCTION

Indexing is a crucial technique in database systems to accelerate query processing. However, identifying the optimal set of indexes for a workload is a challenging task. To address this, database management systems provide *index advisors* [9, 10, 21, 38] to automatically recommend an appropriate index *configuration* (i.e., set of indexes) for a given workload, subject to constraints such as the number of indexes and available storage space [26, 34].

Authors' addresses: Matteo Brucato, Microsoft Research, USA, mbrucato@microsoft.com; Tarique Siddiqui, Microsoft Research, USA, tasidd@microsoft.com; Wentao Wu, Microsoft Research, USA, wentwu@microsoft.com; Vivek Narasayya, Microsoft Research, USA, viveknar@microsoft.com; Surajit Chaudhuri, Microsoft Research, USA, surajitc@microsoft.com.

ACM 2836-6573/2024/2-ART50
https://doi.org/10.1145/3639305

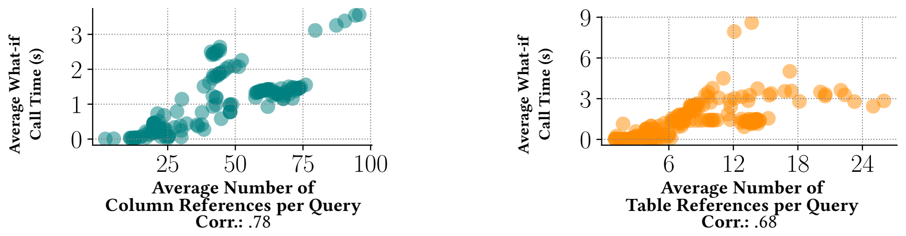


Fig. 1. The average what-if call time (y-axis) grows with the query complexity, estimated (x-axis) by the average number of column references (left) and table references (right). Each point is a workload that has been tuned with an industrial-strength index advisor. The Pearson product-moment correlation coefficients between the two axes are high, with negligible p-values smaller than 0.05.

Index advisors leverage the *what-if* call API, which is supported in most query optimizers of existing database systems [11, 21]. A what-if call enables the advisor to submit a hypothetical index configuration and a query, and optimize the query as though the indexes in the configuration had already been constructed, *without* actually constructing them. By utilizing what-if calls, the advisor is able to estimate the potential cost improvement that can be achieved for a given configuration. Unfortunately, most of the tuning time is spent on executing what-if calls [9, 30, 34, 35, 43], for two main reasons: (1) what-if calls are slow since they are similar to regular query optimizer calls and (2) the advisor typically issues a large number of what-if calls during index tuning.

Since the number of what-if calls grows with the number of candidate configurations and the number of queries, index advisors use heuristics to reduce both. For example, they restrict the set of candidate configurations to only include columns that are most likely to benefit from indexes [1]. Additionally, they can reduce the number of queries by only tuning for queries that, together, can represent most of the workload—an approach referred to as *workload compression* [7, 12, 34].

While existing approaches can speed up index tuning significantly by reducing the total number of what-if calls, each what-if call remains expensive. Even small workloads can have complex queries—e.g., with many joins, filter predicates, group-by and order-by clauses—which typically require long optimization times, and tuning these workloads can still take hours [43]. A simple approximation of the query complexity is given by the number of column and table references in each query, as depicted in Figure 1. The figure, created with our synthetic benchmarks and real-world customer workloads using an industrial-strength index advisor [1], shows that the average what-if call time grows proportionally to the average number of column references (left figure) and table references (right figure) per query. This indicates that index tuning could benefit from a method capable of simplifying each individual query, thereby accelerating what-if calls.

To this end, we propose *workload reduction*, which rewrites queries to reduce their complexity, e.g., by eliminating some filters, joins, etc. In comparison to prior techniques [1, 7, 12, 34, 35, 43] that only reduce the *number* of what-if calls, workload reduction attacks a different dimension of the complexity of index tuning, i.e., the *optimization complexity* of each individual query, thereby making what-if calls *faster*. Workload reduction is *complementary* to prior techniques, including workload compression [34], and can be used in conjunction with them. Similar to workload compression, workload reduction can be used as a pre-processing step for index tuning, requiring no changes to the optimizer or the index advisor. Unlike workload compression, workload reduction can also modify queries, rather than just keeping or eliminating them entirely. Workload reduction can speed up tuning even when eliminating queries is not a viable option—e.g., in small but complex workloads—while still retaining high index quality, as the following example illustrates.

Illustrative example. Figures 2 and 3 present an example of workload reduction in action. In this example, eliminating either of the original queries from Figure 2, Q_1 or Q_2, would make the

Q_1:

```
   SELECT S_Name,
          COUNT(*) AS numwait
     FROM LineItem, Supplier, Nation
    WHERE L_SuppKey = S_SuppKey
      AND S_NationKey = N_NationKey
      AND N_Name = 'INDIA'
 GROUP BY S_Name
```

Q_2:

```
SELECT SUM(L_ExtendedPrice * (1 - L_Discount))
    AS revenue
  FROM LineItem, Orders
 WHERE L_OrderKey = O_OrderKey
   AND O_OrderDate >= '1994-05-01'
   AND O_OrderDate < DATEADD(mm,3,'1994-05-01')
   AND L_ReturnFlag = 'R'
```

Best configuration for $\{Q_1, Q_2\}$ improves total cost by 95%:

```
(1) LineItem(L_ReturnFlag, L_OrderKey, L_ExtendedPrice, L_Discount)
(2) LineItem(L_SuppKey)
(3) Orders(O_OrderDate, O_OrderKey)
(4) Supplier(S_NationKey, S_SuppKey, S_Name)
```

Best configuration for Q_1 improves Q_1 by 97%, but both queries by only 44%:

```
(1) LineItem(L_SuppKey)
(2) Supplier(S_NationKey, S_SuppKey, S_Name)
```

Best configuration for Q_2 improves Q_2 by 95%, but both queries by only 51%:

```
(1) LineItem(L_ReturnFlag, L_OrderKey, L_ExtendedPrice, L_Discount)
(2) Orders(O_OrderDate, O_OrderKey)
```

Fig. 2. Example workload on the TPC-H schema. Tuning on Q_1 alone (Q_2, resp.), causes a great loss in improvement: from 95%, with both queries tuned, to 44% (51%, resp.).

improvement over the original queries drop substantially, from 95% to either 44% or 51%, using an industrial-strength index advisor [1]. Instead, workload reduction (Figure 3) keeps both queries, but it rewrites them to eliminate references to the Nation and Orders tables and their columns. This reduces the complexity of the queries, and results in a much higher 86% improvement than eliminating queries. Notice how the new queries do not retain the semantics of the original queries, but only some of the original expressions (e.g., projections, aggregates, filter predicates, joins, group-by's), and they can include new expressions that were not part of the original query (e.g., the new group-by clause in Q_2').

Challenges of workload reduction. While the above example shows the potential of workload reduction to simplify queries and still retain high improvement for the original queries, achieving this presents several challenges. First, it is unclear how to *define a space* of possible query rewritings that is expressive enough to contain good rewritten queries, while being, at the same time, amenable to efficient solutions. Prior work on query rewriting focuses on maintaining the query semantics [6, 31], which is not a requirement in our setting and does not offer a framework for simplifying queries by eliminating expressions. To the best of our knowledge, this paper is the first to introduce a framework for workload *reduction*, and we show its NP-hardness (Section 2). Second, workload reduction must be *extremely fast* and, in particular, it must avoid any costly operations such as optimizer calls or computing complex statistics, otherwise we would lose the very purpose of reducing a workload in the first place. Third, workload reduction must preserve index *quality* by ensuring that the best configuration for the new reduced workload is also good for the original workload. This, in itself, introduces further challenges: (1) identifying columns and tables that can greatly benefit from indexes (e.g., LineItem) without executing costly what-if calls (Section 3.1); (2) avoiding inadvertent introduction of cross-products that could inflate query cost and divert tuning resources from other queries (Section 4.1); (3) ensuring that columns potentially benefiting from indexing (e.g., L_OrderKey in Q_2) are not overlooked by the index advisor when others in the same join predicate (i.e., O_OrderKey) are eliminated (Section 4.2). Finally, the amount of reduction—i.e., how many expressions to remove from the queries—has a major impact on the index tuning quality: removing too many expressions would result in poor configurations, and removing too few would not provide any major speedups. Unfortunately, we do not know the exact effect of reduction without executing expensive what-if calls (Section 5).

Q_1':
```
  SELECT S_Name, S_NationKey,
         COUNT(*) AS numwait
    FROM LineItem, Supplier
   WHERE L_SuppKey = S_SuppKey
GROUP BY S_Name, S_NationKey
```

Q_2':
```
  SELECT L_OrderKey,
         SUM(L_ExtendedPrice * (1 - L_Discount))
      AS revenue
    FROM LineItem
   WHERE L_ReturnFlag = 'R'
GROUP BY L_OrderKey
```

Best configuration for $\{Q_1', Q_2'\}$ improves $\{Q_1, Q_2\}$ by 86%**:**
```
(1) LineItem(L_ReturnFlag, L_OrderKey, L_ExtendedPrice, L_Discount)
(2) LineItem(L_SuppKey)
(3) Supplier(S_NationKey, S_Name, S_SuppKey)
```
Fig. 3. Example of workload reduction in action: (1) keep both queries, but remove some table and column expressions; (2) keep tables and columns likely to need indexes in the original queries. The improvement over the original workload jumps to 86%.

Our proposed technique. We present **Wred**, a technique for efficiently reducing a workload to speed up index tuning while ensuring that the quality of the resulting configuration is not affected significantly compared to tuning the original workload directly. **Wred** is guided by a *column selection* phase, which identifies a set of important *indexable* columns (i.e., columns that are likely to benefit from indexes) from the *entire* original workload. **Wred** then rewrites each query, retaining only references to these columns and associated tables. While index advisors perform a similar column selection to limit the number of candidate configurations (e.g., *column-group restriction* in Microsoft SQL Server's DTA [1]), they seek to minimize both false positives and negatives, and they achieve this through query optimizer interrogation. However, because this approach is too costly in our setting, we optimize our column selection to include all possibly relevant columns, limiting only false negatives, and delegate the task of eliminating irrelevant ones to the advisor. **Wred** efficiently finds relevant columns by only using readily available statistics, such as table sizes and frequency of table references in the queries.

To rewrite queries, **Wred** modifies the Abstract Syntax Tree (AST) of the original query via a set of rules that eliminate unwanted expressions (nodes in the AST) that correspond to the non-selected columns, while ensuring the SQL validity of the resulting query. Because the new queries are never executed, but only used during tuning to identify good indexes for the original queries, **Wred** does not need to preserve query semantics, i.e., the query generated by **Wred** is not required to produce the same result as the original query. **Wred** avoids the creation of new cross-products in the queries with an efficient algorithm that gradually eliminates nodes from the join graph, ordered by increasing number of edges, as long as the elimination does not create new connected components. This algorithm proves very effective in our experimental evaluation. Furthermore, **Wred** mitigates the loss of join keys when eliminating join predicates with a simple yet effective heuristic: *adding the keys to the group-by clause*, which proves effective at improving the tuning quality in all our experiments.

Wred automatically decides on the right amount of reduction by identifying the *knee* [33] of a curve that estimates improvement based on a low-overhead statistical analysis of the workload, without requiring input from the user. We show that this technique is effective across all our workloads.

We perform a comprehensive evaluation of workload reduction with both synthetic benchmarks and real-world customer workloads. We show that **Wred** is able to speed up the Microsoft SQL Server index advisor DTA by 3× (median across workloads) with only a 3.7% loss in total improvement, compared to using the advisor directly on the original workload. If **Wred** is tuned optimally, it can speed up index tuning by up to 24.7× with 1.8% loss. For large and complex workloads, such as a customer workload with more than 300 complex queries, where the advisor reaches the specified time limit of 8 hours, **Wred** can also help the advisor find configurations with

18.8% better improvement than tuning the original workload directly in the same amount of time. When used in conjunction with workload compression techniques, such as the state-of-the-art **Isum** [34], **Wred** achieves higher speedups than **Wred** *alone* or **Isum** *alone* with comparable loss, or higher improvements with the same speedup, obtaining median speedup of 10.5× with median loss of 5%. In this paper, we consider a simple sequential combination of **Isum** with **Wred**, and discuss limitations in section 6.5.

In summary, this paper makes the following contributions:

(i) To the best of our knowledge, we are the first to speed up index tuning by speeding up what-if calls rather than by reducing the number of what-if calls. To this end, we propose workload reduction, and study its complexity.
(ii) We propose **Wred**, an efficient algorithm that produces reduced workloads of high quality. **Wred** breaks computation into two phases: *column selection* and *query rewriting*. We study the hardness of each and design efficient algorithms.
(iii) We perform a comprehensive experimental evaluation over industry benchmarks as well as real-world customer workloads, using industrial-strength [1] and open-source [22] index tuners.

Paper organization. In Section 2, we formulate the workload reduction problem formally and discuss its complexity. Section 3 presents **Wred**, while Section 4 provides efficient improvements. Section 5 presents a technique to automatically set the reduction amount. Section 6 presents our experimental evaluation, and Section 7 discusses related work. In Section 8, we discuss limitations and opportunities for future work. The paper concludes in Section 9.

2 WORKLOAD REDUCTION

In this section, we introduce workload reduction. We first establish its design goals, and then formally define the problem (Section 2.1), as well as its complexity and challenges (Section 2.2).

Design goals. At a high level, workload reduction rewrites each query in the original workload to speed up index tuning. To do so, it must strike a balance between the following three goals:

(G_1) **Tuning Speedup:** Workload reduction should reduce tuning time substantially compared to tuning the original workload.
(G_2) **Low Quality Loss:** The reduced workload should let the advisor find a configuration that also improves the performance (measured using optimizer estimated cost) for the original queries.
(G_3) **Reduction Efficiency:** Reduction must operate fast. Using reduction in combination with compression (Section 6.5) imposes even more stringent requirements on the efficiency of reduction. To this end, our goal is to never rely on costly operations such as optimizer calls, or computation of complex statistics and histograms.

2.1 Problem Definition

We first recap the index tuning problem. Let $W := \{Q_1, \ldots, Q_N\}$ be a workload consisting of N queries. Let $\mathbb{I}_W := \{I_1, \ldots, I_M\}$ be the M *syntactically-relevant* indexes for W, i.e., indexes applicable to columns referenced in one or more queries in W. The search space of index tuning, denoted by $\Gamma(W)$, is the set of all possible index configurations, i.e., all subsets of $\mathbb{I}_W$. Let $C(Q_i)$ be the *original* cost of Q_i prior to tuning, obtainable via an optimizer call, and $C(Q_i, C)$ the cost of Q_i if (only) configuration $C \in \Gamma(W)$ is used, obtainable with a what-if call. A what-if call, which is supported in most database systems, enables the advisor to submit a hypothetical index configuration and a query, and optimize the query as though the indexes in the configuration had already been constructed, without actually constructing them. By utilizing what-if calls, the advisor is able to estimate the potential cost improvement that can be achieved for a given configuration. What-if calls are typically as expensive as normal optimizer calls. The *improvement* given by configuration C on Q_i is $\mathcal{I}(Q_i, C) := C(Q_i) - C(Q_i, C)$, and index tuning looks for a configuration that maximizes the sum of improvements for all queries in W, subject to a set of constraints such as the maximum configuration size and the limit on the storage space.

Workload reduction speeds up optimizer calls by rewriting each query Q_i into a reduced Q'_i with fewer column and table references. The space of possible rewritings is very large. For instance, we could write new queries from the ground up by combining columns across tables, including them in various operations such as filters, group-bys, order-bys, etc., in all possible ways. This would make the problem space unmanageable. Instead, we design rewriting to *eliminate* expressions from the original query. The choice of which expressions to keep is guided by a *column selection* phase.

Column selection identifies a set of columns referenced in the workload that are most likely to need indexes. We select a single set of columns for the entire workload to obtain a manageable search space and allow for efficient solutions. A more general approach that can select a different set of columns per query might include better solutions, but its search space would be exponentially larger in N, the number of queries. We compare these two approaches in the extended version [3] and show that selecting a single set of columns for the entire workload can offers the same improvements as the general approach under some conditions on the query cost model.

Query rewriting transforms the Abstract Syntax Tree (AST) of the original query into another AST that includes at least the previously selected columns, while still ensuring SQL validity. Query rewriting applies the minimal transformations to the original AST that are necessary to achieve these goals. The transformations include: elimination of nodes, modifications, or addition of new nodes. For example, query rewriting may remove *all* expressions referring to columns that were not selected. We discuss the caveats and challenges of query rewriting in Section 2.2.

Formally, if Q is a SQL query, denoted by $Q \in$ SQL, let $\pi(Q)$ be the set of columns referenced in Q. A reference to a column can appear in any part of the query, including sub-queries, common table expressions, etc. For a subset of columns $K \subseteq \pi(Q)$, let $\rho(Q, K)$ be a rewriting algorithm that rewrites Q into a new valid SQL query that includes, at least, all columns in K. A *reduced version* of Q is a valid SQL query, produced by ρ, that refers to a subset of the columns of Q:

Definition 1: Reduced Query

Let $Q \in$ SQL. Q' is a *reduced version* of Q, denoted by $Q' \sqsubseteq_\rho Q$, iff $\exists K : K \subseteq \pi(Q') \subseteq \pi(Q)$, $Q' = \rho(Q, K)$, and $Q' \in$ SQL.

A reduced query can also reference no columns, i.e., when $\pi(Q') = \emptyset$. This corresponds to *eliminating* the original query, like workload compression might do. While workload compression can only either keep or eliminate a query, reduction can create many reduced versions of the same query by eliminating different subsets of its columns. Since we do not replace the original queries for execution, but only for improving the index tuning efficiency, we do not pose any restrictions on query semantics.

We extend our definitions to a workload in the obvious way: $W' \sqsubseteq_\rho W$ iff $\forall i \in [1..N], Q'_i \sqsubseteq_\rho Q_i$; $\pi(W) := \cup_{i=1}^{N} \pi(Q_i)$ is the set of columns referenced in W; $L := |\pi(W)|$ its size; similarly, for W', $\pi(W') := \cup_{i=1}^{N} \pi(Q'_i)$ and $L' := |\pi(W')|$; we have that if $W' \sqsubseteq_\rho W$, then $\pi(W') \subseteq \pi(W)$. The objective of workload reduction is to identify the optimal set of columns and rewriting algorithm that will maximize the sum of improvements achieved on the original queries when the advisor is applied to the reduced workload.

Problem 1: Workload Reduction for Index Tuning

Given a configuration size limit k, an advisor $\mathcal{A}$, and an upper bound l on the total number of columns referenced in the reduced workload, $1 \leq l \leq L$, the optimal reduced workload $W^{\mathcal{A}}_{l,k}$ is the solution to the following constrained optimization problem:

$$W^{\mathcal{A}}_{l,k} \coloneqq \underset{\boldsymbol{W'} \sqsubseteq_\rho W}{\arg\max} \ \sum_{i=1}^{N} \mathcal{I}(Q_i, C^{\mathcal{A}}_{\boldsymbol{W'},k})$$
$$\text{s.t. } \ |\pi(\boldsymbol{W'})| \leq l$$
$$\text{and } \ \pi(\boldsymbol{Q'_i}) \supseteq \pi(\boldsymbol{W'}) \cap \pi(Q_i), \quad \forall i \in [1..N],$$

where $C^{\mathcal{A}}_{W',k}$ is the configuration given by $\mathcal{A}$ for a reduced W'.

The parameter l serves as a trade-off between tuning speed and the quality of the solution, which are two of our primary goals, i.e., (G_1) and (G_2). When l equals the maximum value L, there is no reduction, leading to no improvement loss, but also no speedup, where L is the total number of columns referenced in the original workload. Conversely, when l equals the minimum value of 1, the speedup is maximal, but the loss may be unacceptably high. Setting this parameter is hard for a user. In Section 5, we propose a technique for automatically determining an appropriate value of l based on the input workload.

The constraints $\pi(Q'_i) \supseteq \pi(W') \cap \pi(Q_i)$ enforce a *workload-level* column selection. They work in the following manner: if a column c is included in the reduced workload (i.e., $c \in \pi(W')$), then c must also be included in all queries that originally referred to it, i.e., c must be included in $\pi(Q'_i)$ for all i such that $c \in \pi(Q_i)$.

Notice that an alternative formulation of Problem 1 that minimizes $|\pi(W')|$ s.t. $\sum_{i=1}^{N} \mathcal{I}(Q_i, C^{\mathcal{A}}_{W',k}) = \sum_{i=1}^{N} \mathcal{I}(Q_i, C^{\mathcal{A}}_{W,k})$ is not viable because constraining the improvement is not possible without running the index advisor.

2.2 Complexity and Challenges

Claim 1: Problem 1 is NP-hard.

We prove the claim in the extended version [3], with a reduction from minimum k-median, following a similar approach to proving the NP-hardness of workload compression [7]. There are exponentially many candidate subsets of columns to choose from, and we cannot exactly measure the improvement each set would yield without expensive what-if calls; instead, we must select columns by *estimating* the improvement. For our efficiency goal (G_3), selection must also be extremely fast. Therefore, we only want to use statistics readily available in the database or quickly computable at runtime, and avoid optimizer calls altogether.

Query rewriting is also non-trivial. Eliminating a table reference that acts as a bridge between two tables in the join graph may introduce a cross-product between the remaining tables; new cross-products may excessively increase the cost of the query and mislead a cost-based index advisor to focus on tuning that particular query, ignoring the other queries. Additionally, in order to maintain valid SQL syntax, the elimination of expressions from a query may cascade to other expressions. For example, eliminating column `C1` from a join predicate `C1` = `C2` requires the *whole* predicate to be eliminated, cascading the elimination to column `C2`, and the advisor would not be able to recommend indexes for `C2`.

3 WRED: EFFICIENT WORKLOAD REDUCTION

In this section, we introduce **Wred**, our efficient algorithm for workload reduction that addresses its challenges. **Wred** has two phases, shown in Figure 4: first, it selects a set of columns to keep globally across the entire workload; then, it processes each query one at a time, creating a new reduced query that includes at least the columns selected in the previous phase. Algorithm 1 depicts these steps in pseudo-code. Following our efficiency goal (G_3), we present our efficient techniques

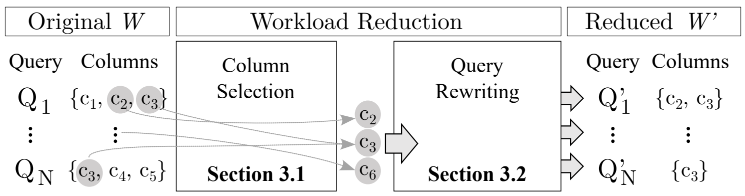


Fig. 4. Phases of Workload Reduction. Column selection (Section 3.1) selects the index-needy columns, and Query Rewriting (Section 3.2) rewrites each query to only reference those columns.

Algorithm 1 WRED

```
Input: Input workload $W$, max number of columns $l \le L$
Output: Reduced workload $W'$
1: $W' \leftarrow \emptyset$
2: $K \leftarrow$ ColumnSelection($W, l$)          ▷ Target set of columns in $W'$
3: for all $i \in [1..N]$ do
4:     $Q'_i \leftarrow$ QueryRewriting($Q_i, K$)
5:     $W' \leftarrow W' \cup \{Q'_i\}$
6: return $W'$
```

for ColumnSelection (Section 3.1) and QueryRewriting (Section 3.2) to produce good results in practice (Section 6), leveraging our understanding of index tuning.

3.1 ColumnSelection

The goal of ColumnSelection is to quickly identify a good set of index-relevant columns to include in the reduced queries. Advisors also perform column selection as part of their search strategy to generate candidate indexes for the workload. As their objective is to find an optimal set of columns, i.e., with both few false negatives and few false positives, they use optimizer calls to obtain the cost of the queries [2]. Our objective is to perform an efficient column selection without invoking the optimizer. For this reason, we only focus on reducing false negatives in our selected columns, which we can achieve, as we show later, by only relying on readily-available statistics, such as the table sizes and the frequency of table references in the workload. This has two advantages: (1) it simplifies our ColumnSelection without the need to interrogate the optimizer, and (2) it also speeds up the advisor's column selection procedure, since it reduces the set of columns the advisor must consider. With this objective in mind, we design our ColumnSelection to exclude columns that may not need indexes. We observe that these are usually columns from smaller tables (as noted by [2]), or tables that are accessed very rarely in the workload:

> **Observation 1:** *"The more rows, the more an index can help"*
>
> The number of rows in a column is generally proportional to the improvement an index can provide for that column.

> **Observation 2:** *"The more frequently a table is accessed to answer a query, the more an index on the same table can help"*
>
> The number of times a table is referenced in the queries is generally proportional to the improvement an index can provide.

Based on these observations, we define *indexable data* to capture the importance of a column for index tuning.

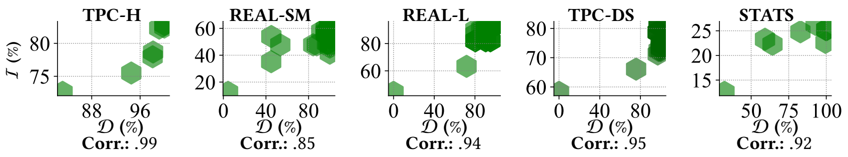


Fig. 5. Percentage improvement (y-axis) obtained by reducing a workload with **WRED** using the corresponding indexable data (x-axis). Indexable data is a good estimate for improvement as indicated by the high correlation between the two.

Algorithm 2 COLUMNSELECTION

```
Input: Input workload $W$, max number of columns $l \in [0..L]$
Output: Subsets of columns $K \subseteq \pi(W)$ s.t. $|K| \leq u$
 1: $K \leftarrow \emptyset$
 2: $T \leftarrow$ GETTABLES($W$)                    ▷ Array of tables referenced in W
 3: $D \leftarrow$ ARRAY(count : $|T|$)              ▷ Array of indexable data scores
 4: for all $i \in [1..|T|]$ do
 5:     $D_i \leftarrow T_i$.ROWCOUNT $\times$ $T_i$.NUMREFS($W$)
 6: $I \leftarrow$ ARGSORT($D$)                       ▷ Indexes of sorted D (desc. order)
 7: for all $i \in [1..|I|]$ do
 8:     $t \leftarrow T_{I_i}$                          ▷ i-th best table
 9:     $C \leftarrow t$.GETCOLUMNS($W$)               ▷ Columns of t referenced in W
10:     if $|K| + |C| \leq l$ then
11:         $K \leftarrow K \cup C$
12:     else break
13: return $K$
```

Definition 2: INDEXABLE DATA

Given a column $c \in \pi(W)$, let $|c|$ be its "*size*", i.e., the number of rows of the table where c belongs; let $f(c, W)$ be its "*frequency*" in W, i.e., the number of times c's table is referenced in W.

$$\mathcal{D}(c) := |c| \cdot f(c, W)$$

The indexable data of a column estimates the *total amount of data*, measured in the number of rows, that the system needs to access in order to answer all the queries in the workload involving that column if no indexes were involved, i.e., the total cost of *scanning* the table where the column belongs to, for as many times as the table is referenced in the workload. The higher this measure, the more rows the system accesses, and therefore the more rows may benefit from indexing. Therefore, this measure captures the opportunity for indexes to reduce access time of tables. Computing it is extremely efficient, with readily available statistics such as table sizes, and counting the frequency of references in the workload. COLUMNSELECTION computes $\mathcal{D}(c)$ for each column c in the input workload, sorts columns by this measure, and picks the first l with the highest scores. Algorithm 2 depicts these steps in detail.

This measure does not take into account the selectivity of predicates. However, since we are only interested in constructing new queries, and letting the advisor do the hard job of checking selectivity, we only need to construct queries that have a good chance to be indexed. If some unselective predicates are also included, we leave to the advisor the job of filtering them out. This has the advantage of simplifying workload reduction, while maintaining the quality of the resulting configuration. For similar reasons, this measure also ignores whether data needs to be sorted for group-by's, order-by's, or merge joins. However, as we discuss later in Section 3.2, **WRED**'s rewriting keeps these operations if they involve selected columns, giving them a chance to be considered for indexing.

Algorithm 3 QueryRewriting

```
Input: Query $Q \in$ SQL, target columns to keep $K$
Output: New query $Q' \sqsubseteq Q$ s.t. $\pi(Q') \supseteq K$
1: $(C, T) \leftarrow$ MarkForRemoval$(Q, K)$    ▷ Column and table expressions marked for removal according to K
2: return RewriteQuery$(Q, C, T)$
```

We further support our intuition as to why indexable data well captures improvement in Figure 5. We run **Wred** on our evaluation workloads (Section 6) with increasing values of l, and report the workload-level indexable data ($\mathcal{D}$) percentage on the x-axis, computed as $\mathcal{D}(\%) := \mathcal{D}(W')/\mathcal{D}(W) * 100\%$, where $\mathcal{D}(W) := \sum_{c \in \pi(W)} \mathcal{D}(c)$ and $\mathcal{D}(W') := \sum_{c \in \pi(W')} \mathcal{D}(c)$, such that $\mathcal{D}(\%) = 100\%$ means that the reduced workload is identical to the original workload (no reduction). We then run the advisor on the reduced workload obtained for each $\mathcal{D}$ value, and report on the y-axis the percentage improvement ($\mathcal{I}$) obtained on the original workload. We show the Pearson product-moment correlation coefficient between indexable data and improvement on the bottom (the associated p-value is always smaller than 0.05). All correlations are high, not due to chance, showing that indexable data is a good estimate for improvement.

Given the high correlation between $\mathcal{D}$ and $\mathcal{I}$, a natural question is whether $\mathcal{D}$ is a good estimate for $\mathcal{I}$ in Problem 1 to produce optimal reduced workloads, where we replace $\mathcal{I}(Q_i, C^{\mathcal{A}}_{W',k})$ with $\mathcal{D}(Q'_i) := \sum_{c \in \pi(Q'_i)} \mathcal{D}(c)$. A sufficient condition for $\mathcal{D}(Q'_i)$ to produce optimal results for Problem 1 is that for any two reduced workloads $W' = \{Q'_1, \ldots, Q'_N\}$ and $W'' = \{Q''_1, \ldots, Q''_N\}$, $\forall i \in [1..N]$: $\mathcal{I}(Q_i, C^{\mathcal{A}}_{W',k}) > \mathcal{I}(Q_i, C^{\mathcal{A}}_{W'',k}) \Rightarrow \mathcal{D}(Q'_i) > \mathcal{D}^*(Q''_i)$. Indexable data only satisfies these conditions if W' has strictly more columns than W'' (proof in the extended version [3]), which only constitutes a small fraction of the cases. While $\mathcal{D}$ is not a good estimate for $\mathcal{I}$ if used directly in Problem 1, it is powerful for identifying important columns for workload reduction, as Figure 5 shows. Devising an optimal or approximate estimate is challenging due to the hardness of index tuning.

3.2 QueryRewriting

Given a query Q and a set of columns K computed by ColumnSelection, QueryRewriting constructs $Q' \sqsubseteq_\rho Q$ by removing table and column expressions according to K. The steps of the algorithm are shown in Algorithm 3. First, the algorithm marks columns and tables in Q *for removal*, according to K. For now, we assume that a column is marked for removal if it is *not* included in K, and that a table is marked for removal if none of its columns is included in K. Later, in Section 4, we discuss the drawbacks of blindly removing all columns and tables according to K, as doing so may introduce new cross-product in the resulting query, or prevent certain columns from being considered for candidate indexes, thereby worsening the results produced by an advisor; we will then modify the algorithm to avoid these pitfalls and improve the results. After this step, the algorithm calls the RewriteQuery method to produce a valid SQL reduced query in output. RewriteQuery operates syntactically on the query, by traversing its AST.

The AST for the original query is obtained by executing the SQL language parser on the SQL string. The specific form the AST takes depends on the SQL dialect and DBMS being used. While we base our implementation around the T-SQL parser of Microsoft SQL Server, the algorithm we present is not tied to a particular dialect and easily extendable to other dialects (e.g., PostgreSQL, see Section 6.8). The method works in three main phases:

(1) Parse obtains the AST of Q, denoted by T;
(2) Rewrite creates a new AST tree T' from T, such that T' only references columns and tables *not* marked for removal;
(3) Print outputs Q' in the target SQL dialect, by simply traversing T' in the obvious way.

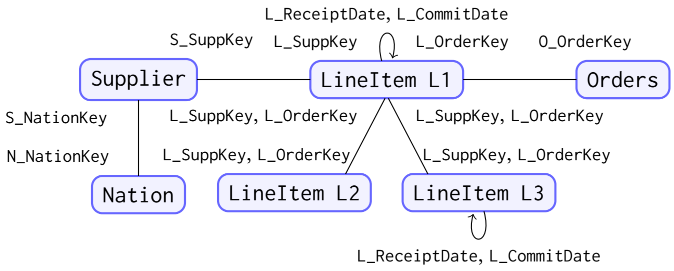


Fig. 6. Join graph of TPC-H Q_{21}. Supplier acts as a bridge between LineItem L1 and Nation. Removing it would introduce a cross-product between LineItem and Nation.

PARSE is provided as part of the DBMS library; PRINT is trivial, and thus we omit its details. REWRITE traverses the AST T, and applies a set of *rules* that depend on the specific node that the algorithm is currently visiting. These rules determine which node to produce in the output AST T'.

AST Nodes. For brevity, we only describe some of the main AST node types. The root of T is a QueryBatch node, i.e., a list of Query nodes. Each Query node contains: a list of CTE nodes, a list of Select nodes, a list of From nodes, a list of GroupBy nodes, a list of OrderBy nodes, a Where node, a Having node, a Topk node, a Distinct Boolean value. CTE includes a CTE name and a Query node. Select is an expression combining Column nodes, Function nodes, primitive types, etc. From is an expression including Table nodes. Function includes a function name and the parameters being passed to the function. Column represents either a column from a table, a view, a CTE, or a sub-query. Table represents either a table, view, CTE or sub-query with an alias name.

The other nodes follow the standard SQL syntax in the obvious way. For example, Where is a Boolean expression with leaves Column, strings, numbers, etc. Having is a Boolean expression that also allows for Function that represent aggregates. Query can also be a Union, Intersect or Except of two SubQuery nodes.

REWRITE Rules. REWRITE traverses the AST T depth-first, and for each node $\mathcal{N}$, it outputs a new node $\mathcal{N}'$ for the resulting AST T'. There are three types of decisions: (1) **keep**: $\mathcal{N}' = \mathcal{N}$; (2) **remove**: do not output any node; and (3) **modify**: $\mathcal{N}' \neq \mathcal{N}$. REWRITE performs different operations based on the type of $\mathcal{N}$. Here, we only show some of the major operations and list the remaining ones in the extended version [3].

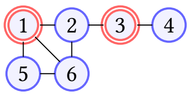
- Column. If $\mathcal{N}$ is a reference to a column in C, **remove**; if it is a reference to some other expressions (e.g., alias or column in a CTE or sub-query), and if all such expressions were removed, **remove**; else, **keep**.
- Table. If $\mathcal{N}$ is a reference to a table in T, **remove**; else, **keep**.
- Function. REWRITE each argument of $\mathcal{N}$. Depending on the function name (some functions can be called with fewer arguments), if too many arguments are removed to form a valid call to the function, **remove**; else **keep**.

4 IMPROVING REDUCTION QUALITY

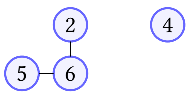
In this section, we discuss the drawbacks of removing all expressions marked for removal, and present techniques to address them.

4.1 Preventing Creation of Cross-products

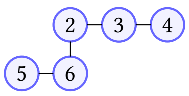
In Section 3.2, we developed a rewriting algorithm that marks for removal *every* column and table expressions related to K, the set of columns picked by the column selection step. However, removing



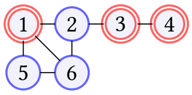
(a) Two nodes marked for removal.



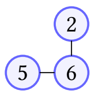
(b) Removing both adds a connected component.



(c) Optimal: remove highest-degree node.



(d) Same join graph as (a), where node 4 is also marked for removal.



(e) In this case, removing all marked nodes is feasible and optimal.

Fig. 7. Hard instances for MaxRemoval. (a–c): optimal solutions can require removal of nodes with higher degrees; (d–e): more removal nodes may make previously infeasible solutions feasible.

a set of tables from a query without considering the structure of the join graph can introduce cross-products in the query, which may affect the quality of index tuning.

Consider Q_{21} from TPC-H [37] and its *join graph* as depicted in Figure 6. The join graph includes a node for each individual table reference in the query; if a table is accessed multiple times throughout the query, either in the same `FROM` clause, or in sub-queries and CTEs, the graph includes one individual node for each such reference. The graph includes an edge between two nodes if the respective table references have a join predicate.

Notice that `Supplier` acts as a *bridge* between `LineItem L1` and `Nation` (similar to our example query Q_1 from Figure 2). This means that, although `LineItem` and `Nation` are not explicitly joined by a join predicate, there is no cross-product between the two tables. A possible query plan for this query may first join `LineItem` with `Supplier`, and then join the result of this join with `Nation`.

If we removed `Supplier` from the query, we would introduce a cross-product between `LineItem` and `Nation`. As `LineItem` is large, introducing a cross-product may excessively increase the cost of the query. Because index advisors typically use query costs to guide the search [1, 22], this may inadvertently cause the advisor to focus on tuning this particular query and ignore many others. For reduction to be effective for index tuning, the distribution of the costs of the new queries should be similar to that of the original queries, with the higher-cost queries still be the higher-cost ones in the reduced workload.

In this section, we develop an improved version of the MarkForRemoval procedure, which only marks tables for removal if doing so does not introduce a new cross-product. Formally, we say that two table expressions are in a cross-product if there is no *path* between them in the join graph. Notice that we do not require a direct join predicate (i.e., a direct edge in the graph) between two tables to conclude that they are not in a cross-product. Detecting whether our transformations introduce new cross-products is equivalent to checking whether the number of connected components in the graph increases after eliminating the nodes and edges corresponding to the table expressions to remove. This is because: (1) our transformation can only remove nodes (and edges) from the graph, thus only creating sub-graphs; (2) if the number of connected components in a sub-graph is the same as in the original graph, then for every pair of nodes, there is still a path if there was a path in the original graph. Therefore, our goal is to allow the removal of nodes (and edges) from the graph subject to the constraint of not introducing new connected components.

Algorithm 4 shows how to achieve this objective. The algorithm first marks nodes in the join graph for removal according to K. Then, in Line 3, it computes the *largest subset* of marked nodes that, if removed, do not introduce new connected components in the resulting join graph. Finally, it removes those nodes, to produce a new join graph that includes all table expressions that can be safely removed from Q.

Algorithm 4 MARKFORREMOVAL avoiding cross-product creation

```
Input: Query $Q \in$ SQL, target columns to keep $K$
Output: Columns and tables, $C$ and $T$ resp., to remove from $Q$
1: $G \leftarrow$ JOINGRAPH$(Q)$
2: $N_K \leftarrow$ REMOVALNODES$(G, K)$    ▷ Nodes marked for removal in G according to K
3: $G' \leftarrow$ MAXREMOVAL$(G, N_K)$    ▷ Maximal removal that avoids introduction of new cross-products (i.e., new connected components)
4: $(C, T) \leftarrow$ COLUMNSANDTABLES$(Q, G')$
5: return $(C, T)$
```

Given a graph $G = (V, E)$, let $\nu(G)$ be the number of connected components in G. MAXREMOVAL finds the largest subset of removal nodes $N' \subseteq N_K$ such that the sub-graph G' obtained from G by removing N' has at most the same number of connected components as G, i.e., $\nu(G') \leq \nu(G)$, and then returns G'.

Claim 2: MAXREMOVAL is NP-hard.

We prove the claim in the extended version [3], with a reduction from the Steiner tree problem [23, 24]. This problem is only easy if all removal nodes have one edge, in which case they can all be removed as they can never "split" the graph and create new connected components. In general, there are up to $2^{|N_K|}$ possible feasible sets of removal nodes, and to find the optimal set, we may have to test them all by removing the nodes and counting the resulting number of connected components, which is clearly impractical. One may think that removing nodes with lower degrees (number of edges) is always better than removing nodes with higher degrees, as lower degrees are less likely to split the graph. Unfortunately, Figure 7 shows why this is not the case: in the top example, the optimal solution is to remove the node with the highest degree. Another factor that makes MAXREMOVAL hard is shown in the bottom example of the figure: adding more removal nodes may make previously infeasible solutions feasible, as removing node 3 is now feasible thanks to the removal of node 1. Hard instances of MAXREMOVAL are not rare in practice. For example, in TPC-DS queries, more than half of attempted node removals would introduce new cross-products and, thus, need to be prevented.

We propose a greedy technique, called GREEDYMAXREMOVAL, that quickly outputs a feasible subset of nodes whose removal does not introduce new cross-products, but may not be the largest. While we do not provide approximation guarantees, we observe its good behavior in all our experiments (Section 6). The procedure first removes all nodes marked for removal that have fewer than two edges—these are safe to remove as their removal cannot create new connected components. Then, the algorithm tries to remove one node at a time, in increasing order of degree (ties can be broken at random or with the indexable data metric of the corresponding table), checking whether its removal creates new connected components in the resulting graph; if it does not, the removal proceeds, otherwise the algorithm greedily terminates without removing other nodes. The algorithm exploits the intuition that the fewer edges a node has, the less likely removing it will create new connected components. Algorithm 5 shows these steps in detail, where we use $e(n)$ to denote the number of edges incident to a node n.

4.2 Mitigating Loss of Join Keys

In some cases, the elimination of expressions may lead to unwanted losses of join keys, preventing the advisor from considering them for candidate indexes. Suppose, for instance, that the original query has a join predicate `C1 = C2`, and we want to eliminate `C1`. We then have to eliminate the whole predicate to maintain valid SQL syntax. This elimination forces `C2` to also disappear, preventing indexes for it to be recommended by the advisor—since `C2` was selected to stay, eliminating it may

Algorithm 5 GreedyMaxRemoval

```
Input: Join graph $G = (V, E)$, target set of removal nodes $N_K \subseteq V$
Output: Sub-graph $G' = (V', E')$ of $G$ s.t. $V' \supseteq V \setminus N_K$, $v(G') \leq v(G)$
 1: $G' \leftarrow G$;   $N' \leftarrow N_K$
 2: repeat
 3:     $S \leftarrow \{n \in N' \mid e(n) < 2\}$          ▷ Removal nodes with less than 2 edges
 4:     $G' \leftarrow$ RemoveNodes$(G', S)$               ▷ Remove all nodes in S
 5:     $N' \leftarrow N' \setminus S$
 6: until $S = \emptyset$
 7: Invariant: all remaining removal nodes have 2 or more edges in $G'$   ◁
 8: repeat
 9:     $S^* \leftarrow$ Sorted$(N', e(\cdot))$            ▷ Sort removal nodes by incr. num. of edges
10:     $n \leftarrow$ Deque$(S^*)$                        ▷ Removal node with fewest edges
11:     $\hat{G}' \leftarrow$ RemoveNodes$(G', \{n\})$
12:     if $v(\hat{G}') \leq v(G)$ then                    ▷ Accept removal iff constraint is satisfied
13:         $G' \leftarrow \hat{G}'$;   $N' \leftarrow N' \setminus \{n\}$
14:     else break                                         ▷ Greedily stop when first removal node is rejected
15: until $S^* = \emptyset$
16: return $G'$
```

cause a significant quality loss. We apply our understanding of index tuning to develop a technique that reduces the impact of losing join keys.

Our algorithm proceeds as follows. During the creation of the new AST T', we collect every join key that gets eliminated. After RewriteQuery, we add these *dangling* join keys back to the query, in places where the advisor may consider them as candidates for indexes—note that we are not interested in the semantics of the new queries, but only in using the new queries for index tuning. We experimented with variants of this idea (e.g., adding the keys to the order-by or select clauses), and found that adding these columns to the group-by clause offers the best results. This simple heuristic works because advisors consider group-by columns for potential candidate indexes, even if these columns do not appear in a selection or join predicate, and our experiments show that, in many cases, if an index can improve the cost of a group-by, it can also improve the cost of a join on the same column.

We could potentially add each dangling join key to the group-by independently (i.e., separated by commas). However, this approach may excessively increase the cardinality of the query result, thereby excessively increasing the cost of the query. As we mentioned, skewing the costs too much may degrade the advisor's performance. Instead, we add a `CASE` clause that combines all dangling join keys in a single expression. For example, for three numeric dangling join keys `C1`, `C2`, and `C3`, we write: `GROUP BY CASE WHEN C1 ≤ C2 THEN C3`. The advisor will still consider each column appearing in the `CASE` clause for candidate indexes, and the estimated cardinality of the result will not grow excessively. If the query already has a group-by clause, the new expression is simply appended at the end; otherwise, a group-by clause is added to the query. We discuss additional improvements in the extended version [3], which we tested in our preliminary experiments. These additions do not have a significant impact in our experiments.

5 PARAMETER-FREE REDUCTION

So far, we have assumed that the number of columns in the reduced workload is given. However, setting this number is itself a challenge in practice. To address this, we introduce an automatic technique for setting the amount of reduction, based on the correlation between our indexable data estimate (developed in Definition 2) and the improvement given by reduced workloads.

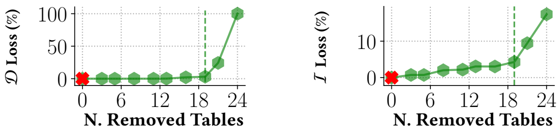


Fig. 8. Indexable data loss (left) used to estimate improvement loss (right) in TPC-DS.

The left-hand side of Figure 8 shows the indexable data *loss* curve, computed on our experimental benchmark TPC-DS. It is efficiently computed, without constructing any reduced workload, as follows: let T be the number of tables referenced in W; let $T_{\mathcal{D}} := [t_1, \dots, t_T]$ be a ranking of the T tables, based on $\mathcal{D}$, where the score of table t is $\mathcal{D}(t) := \sum_{c \in t} \mathcal{D}(c)$, and $c \in t$ means that c belongs to t; for each $x = 0, \dots, T$ on the x-axis:

$$\mathcal{D}\text{ Loss}(x) := \textstyle\sum_{t \in T_{\mathcal{D}}[:x]} \mathcal{D}(t) / \sum_{t \in T_{\mathcal{D}}} \mathcal{D}(t) * 100\%, \tag{1}$$

where $T_{\mathcal{D}}[: x]$ are the top-x tables in $T_{\mathcal{D}}$. The curve is obtained with all the 24 TPC-DS tables.

In the right-hand side of the figure, we plot the improvement *loss* curve. This curve is very expensive to compute. For each point on the x-axis: we reduce the workload, setting l to the number of columns in the kept tables; we tune the reduced workload; we compute the improvement $\mathcal{I}(x)$ on the original workload; we compute the improvement loss as $\mathcal{I}(0) - \mathcal{I}(x)$. Despite its cost, this curve contains all the information necessary to make an informed decision, as we observe that reducing at the *knee* of the curve (dashed line) provides enough reduction without incurring in an excessively high loss. Our goal is to estimate the knee of the improvement loss curve, *without* computing the curve.

Thanks to the high correlation between indexable data and improvement (Section 3.1), we observe that the knee of the indexable data loss curve is a good approximation for the knee of the improvement loss curve. We exploit this observation by using the knee of the easily computable indexable data loss curve at runtime as an estimate. We compute the knee using off-the-shelf knee-point detection algorithms [33], excluding the point with 100% indexable data loss as it is too aggressive for reduction. The algorithm identifies the data point that has the maximum perpendicular distance to a straight line drawn from the first to the last point of the normalized data. If the curve does not have a knee or the detection algorithm fails, we revert to no reduction (in our experiments, we always find a knee). In Figure 8, the knee is at $x = 19$ (dashed line), corresponding to 19 tables (and their corresponding columns) to remove from the workload. In our experiments, this results in 3× speedup for TPC-DS with a 4.3% improvement loss, which is also the best speedup obtainable by reduction where loss is within 5%. In Section 6, we compare this method with using a fixed threshold for indexable data loss (e.g., 5%), instead of computing the knee, and show that using the knee is never worse and often better than the fixed threshold approach.

6 EXPERIMENTAL EVALUATION

In this section, we present our comprehensive evaluation of workload reduction with both synthetic benchmarks and real-world customer workloads.

Workloads. We run our experiments on two standard benchmarks, TPC-H [37] and TPC-DS [36], a benchmark constructed from real-world data, STATS [18], and two real-world customer workloads REAL-L and REAL-M. For TPC-H, we have two variants: single-instance and multi-instance, where the latter, called TPC-H×100, includes 100 instances per template. We also generate a smaller version of REAL-M, called REAL-SM, with $1/10^{\text{th}}$ of the queries randomly chosen. Our AST implementation covers all queries in all workloads, except on TPC-DS, where we excluded 19 out of 99 queries that our parser cannot currently handle. Table 1 summarizes the workloads we use in our experiments.

Workload	Queries	Unique Tables	Unique Columns
TPC-H	22	8	55
REAL-SM	31	61	206
REAL-L	32	19	101
TPC-DS	80	24	237
STATS	146	8	37
REAL-M	316	126	485
TPC-H×100	2,200	8	55

Table 1. The seven workloads used in our evaluation.

Index advisors. We use Microsoft SQL Server DTA for the majority of the experiments, and PostgreSQL Dexter [20, 22] on certain experiments to showcase the applicability of **Wred** to other systems and advisors.

Methods. We evaluate the following methods: (*i*) **Orig** takes as input the original workload and directly tunes it using the index advisor; (*ii*) **Isum** [34], a state-of-the-art workload compression technique, first compresses the original workload selecting k queries, and then tunes the compressed workload; (*iii*) **Wred-auto** applies **Wred** using our auto auto-tuning method from Section 5 to set l, then tunes the reduced workload; (*iv*) **Wred-fixed** tunes a reduced workload obtained by **Wred** using a fixed setting for l (i.e., 5% indexable data loss); (*v*) **Wred-opt** simulates an optimal setting for l under a quality constraint: we first run all settings of l, and then pick the one with the highest speedup such that the loss in improvement is not greater than 5%.

Quality metrics. We evaluate the quality of these methods using the following metrics. (*i*) **Absolute Percentage Improvement** measures the percentage decrease in the total cost of the *original* workload, as estimated by the optimizer, given by the configuration returned by the index advisor. State-of-the-art index advisors use the same metric to measure the quality of recommended indexes [26]. (*ii*) **Percentage Improvement Loss** measures the decrease in Absolute Percentage Improvement of a method wrt. **Orig** (e.g., if **Orig** gives 80% improvement and a method gives 75%, loss is 5%). A *negative* loss means that the method provides a *better* configuration than **Orig**, which happens because advisors use complex heuristics. In our experiments, we see better configurations only on our real customer workloads REAL-SM and REAL-M. For brevity, we will refer to these metrics as (*i*) *absolute improvement* and (*ii*) *improvement loss.*

Efficiency metrics. We measure time efficiency using the following metrics. (*i*) **Compression time** measures the time to compress a workload. Notice that **Isum** uses the costs of the original queries in its computation; we assume that they are given as input (e.g., available from historical workload information [34]), and we do not include the time to compute the costs of the queries. (*ii*) **Reduction time** measures the time to reduce a workload. (*iii*) **Tuning time** measures the time taken by the advisor to tune a workload. (*iv*) **Speedup** is the tuning time of **Orig** divided by the sum of compression (reduction, resp.) and tuning time of a compressed (reduced, resp.) workload (e.g., a speedup of 2x means that a method tunes the workload twice as fast as **Orig**).

6.1 End-to-end Workload Reduction

In the first experiment, we run **Wred** end-to-end, using the three variations **Wred-fixed**, **Wred-auto**, and **Wred-opt**. We report speedup and improvement loss for each workload. Notice that the speedup depends on the complexity and scale of the workload as well as potential for applying indexes; hence it can vary across workloads, as observed in prior work as well [34].

Our results are presented in Table 2. Overall, we observe that **Wred** can provide significant speedup over **Orig**, with only a small loss in quality. In particular, even the simplest approach,

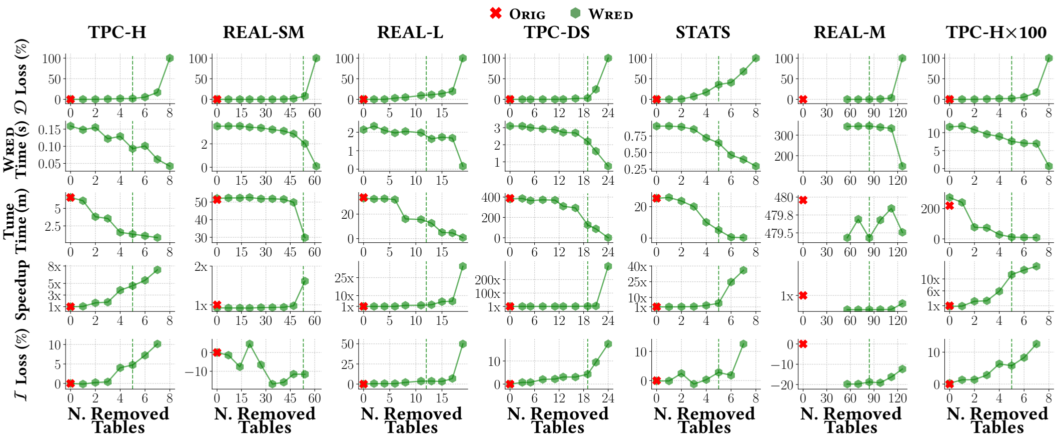


Fig. 9. Effect of degree of reduction, obtained by removing increasingly more tables from W. With more reduction, tuning time decreases but improvement loss increases. The knee (dashed vertical line) of the indexable data loss curve (top row) is always a good speedup/loss trade-off.

Workload	Wred-fixed		Wred-auto		Wred-opt	
	Speedup	$\mathcal{I}$ Loss	Speedup	$\mathcal{I}$ Loss	Speedup	$\mathcal{I}$ Loss
TPC-H	4.6x	4.8%	4.6x	4.8%	4.6x	4.8%
REAL-SM	1.6x	-11.6%	1.6x	-11.6%	2.8x	-7.9%
REAL-L	1.9x	2.4%	2.4x	3.7%	5.6x	3.9%
TPC-DS	3x	4.3%	3x	4.3%	3x	4.3%
STATS	1x	2.5%	4.4x	2.8%	24.7x	1.8%
REAL-M	0.6x	-16.3%	0.6x	-18.8%	0.8x	-12.4%
TPC-H×100	11.4x	5.8%	11.4x	5.8%	2.7x	2.8%

Table 2. End-to-end results of **Wred** variations: **Wred-fixed** sets the indexable data threshold to 95%; **Wred-auto** uses our auto-tuning setting; **Wred-opt** runs all settings and picks the one with highest speedup such that loss is at most 5%.

Wred-fixed, which uses a simple default parameter setting for l, performs well in all workloads, except on STATS and REAL-M where no speedup is obtained. The median speedup across all workloads is 1.9x, with a median improvement loss of only 2.5%. On REAL-M, while there is no speedup, as the advisor always reaches the 8-hour time limit, the configuration is even better than **Orig**, with a gain of 16.3% in improvement, showing that **Wred** is able to make the advisor more effective at finding better configurations in the same amount of time. On TPC-H×100, while the speedup is high (11.4x), the loss is slightly higher (5.8%), above a more desirable 5%. This shows that while **Wred** can substantially speedup a workload made from parameterized queries, the speedup comes at a slightly higher cost. In Section 6.5, we show that the best results for TPC-H×100 are obtained when **Wred** is combined with **Isum** [34].

Wred-auto is never worse than **Wred-fixed**, and it improves the speedup on REAL-L and STATS, with similar losses. The median speedup and loss of **Wred-auto** are 3x and 3.7%, respectively. In TPC-H, TPC-DS, and REAL-M, the performance of **Wred-auto** is comparable to **Wred-opt**, showing that our auto-tuning can pick a near-optimal parameter. For STATS, **Wred-opt** can achieve a significantly higher speedup of almost 25x within 1.8% loss from **Orig**. This shows the existence of a much better reduced workload than what **Wred-auto** can find, highlighting the power of workload reduction as a tool for enhancing the performance of an

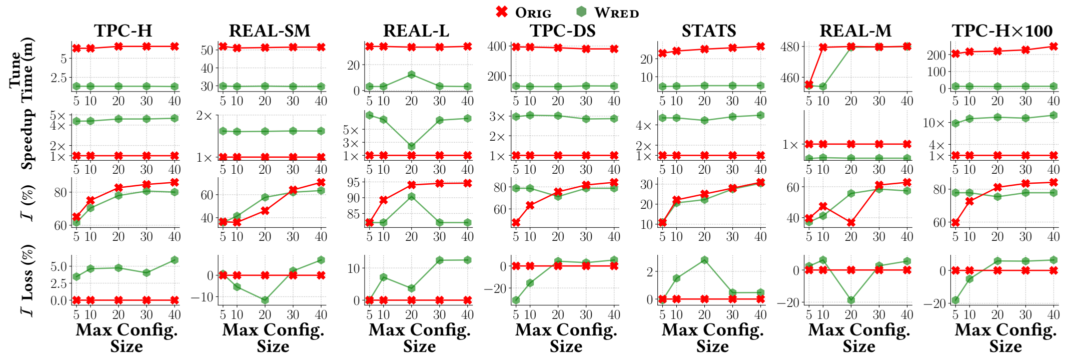


Fig. 10. Effect of increasing the maximum configuration size (number of indexes) on **Wred-auto**. The method scales well with more indexes, maintaining good speedups and low improvement losses, with REAL-L being the hardest case for **Wred-auto** at larger configuration sizes.

index advisor with minimal losses, while suggesting that there is potential to improve our auto-tuning method **Wred-auto** (Section 5). We will explore these improvements in future work. For TPC-H×100, **Wred-auto** obtains a 2.7x speedup at a 2.8% loss.

6.2 Effects of Degree of Reduction

We study the effects on quality and efficiency of increasing the amount of reduction in **Wred**. We increase reduction by removing increasingly more tables (and, subsequently, more columns) from the original workload W. Figure 9 shows our results.

In the top row of the figure, we show the indexable data loss curve, computed using Equation (1) (Section 5), which **Wred-auto** uses to set the reduction degree automatically. We show the value picked by **Wred-auto**, i.e., the knee of the curve, as a dashed vertical line. We then run the advisor on each removal level, fixing the number of tables to remove, and plot: the reduction time incurred by **Wred**, the tuning time for running the advisor on the reduced workload, and the speedup and improvement loss compared to **Orig** (i.e., no reduction).

Reduction time is in the order of seconds, which is negligible compared to the tuning time. This is because **Wred** only parses queries syntactically, requires no optimizer calls, and quickly manipulates queries at the AST level. The plot also shows that more aggressive reduction also requires less time, as the produced queries have fewer expressions.

As one would expect, the tuning time decreases with more reduction, and so the speedup increases, while the loss becomes worse. At the knee point of the indexable data loss curve (dashed line), speedup and loss are at a good trade-off in all workloads. Furthermore, the indexable data loss curve and the improvement loss curve have similar trends, as we discussed in Section 5. REAL-SM and REAL-M both have negative loss, i.e., an absolute improvement *higher* than **Orig**. REAL-M always reaches the 8-hour time limit we set for tuning. For this workload, **Wred** offers no speedup, but a better absolute improvement (indicated by a negative loss).

6.3 Effects of Maximum Configuration Size

In this experiment, we study how the maximum configuration size affects the runtime and quality of **Wred-auto** compared to **Orig**. We show the results in Figure 10. Increasing the maximum configuration size affects both the tuning time and the absolute improvement obtained by **Orig**: larger configurations generally result in higher improvements as they include more indexes. For this reason, we also include in our plots the absolute improvement, in addition to the improvement loss, to better show this trend. The only case when **Orig** does not provide monotonically increasing

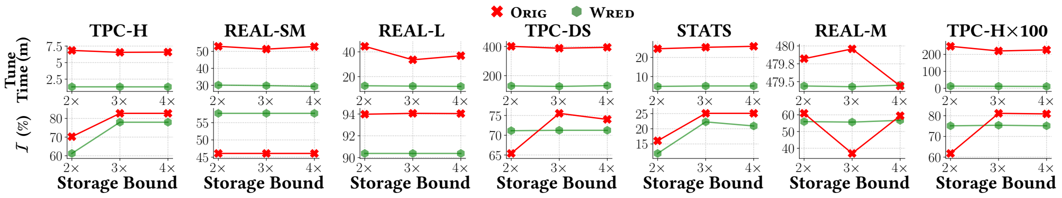


Fig. 11. Effect on **Orig** and **Wred-auto** of increasing storage bounds for the recommended configuration. On the x-axis, the storage bound is expressed as a multiplier of the raw data size in Mb, where 3× is the default used by the advisor. The method scales well with more available storage, maintaining good speedups and high improvement.

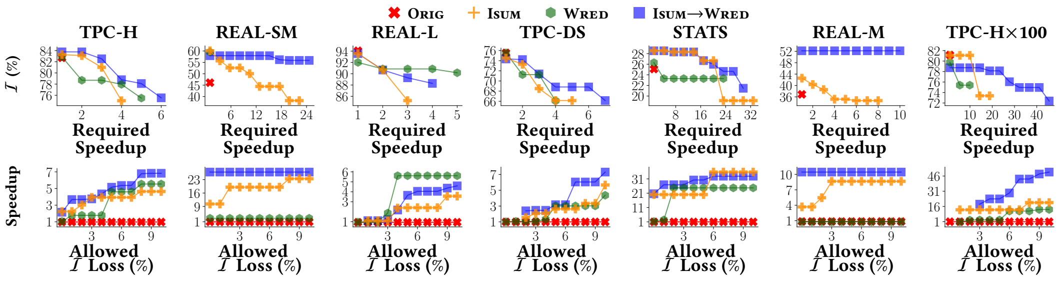


Fig. 12. Comparison and combination with state-of-the-art compression **Isum** [34]. While **Wred** and **Isum** in isolation have comparable results, combining the two in sequence, with **Isum→Wred**, always gives the best results, especially at higher speedups.

improvement is REAL-M because, on this workload, the advisor always reached the 8-hour time limit. **Wred-auto** scales well with increasing configuration sizes, maintaining good speedups. The loss either stays the same with more indexes, or increases, due to the fact that index tuning with more indexes has more candidate solutions and it becomes a harder problem. REAL-L with 20 indexes was a particularly hard case for **Wred-auto**, with speedup going from above 5× down to below 2×. The results on REAL-M show that the configuration size 20 was a particularly hard case for **Orig**, explaining why we observed better improvements when using **Wred** instead. Using different configuration sizes lead to small positive losses, in line with the results obtained on the other workloads.

6.4 Effects of Storage Bounds

In this experiment, we vary the storage bound constraint for the indexes in the recommended configuration, from 2 up to 4 times the raw data size (DTA's default is 3×). The results, in Figure 11, show a similar trend to varying the configuration size (Section 6.3).

6.5 Combination with Compression

While reduction is great at reducing the tuning time by speeding up optimizer calls, workload compression can speed it up by eliminating redundant queries. This can be effective for workloads constructed from parameterized queries, such as TPC-H×100. Existing workload compression techniques (e.g., **Isum** [34], k-medoids [7], and GSUM [12]) are able to efficiently find a small set of queries for tuning. We use the current state of the art **Isum** [34].

We can obtain the benefits of both compression and reduction by chaining the two in sequence. With **Isum→Wred**, we first compress the workload with **Isum** and then reduce it with **Wred**, which is possible given that **Isum** uses the costs of the original queries. Although it is also possible

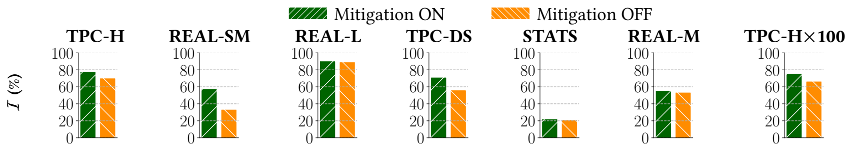


Fig. 13. Effect of our join key loss mitigation strategy. Adding the dangling join keys to the group-by clause never worsens the results and can lead to significant improvements in some workloads.

to first reduce the original workload and then compress it, existing compression approaches, such as **Isum**, prevent the swapping due to practical issues. In fact, **Isum** assumes that the costs of all input queries are provided for free from the logs and can be used to get their benefits. Therefore, if we used **Isum** after **Wred**, we would also need to compute the costs of all *reduced* queries, which is prohibitively expensive as it requires one optimizer call per reduced query. Thus, we only consider **Isum→Wred** as a viable combination option. **Isum** requires an input parameter for the target compression size (number of queries in compressed workload), but it does not offer an auto-tuning method for picking the best parameter. Thus, we sweep the compression size parameter of **Isum** to identify the best results, and then we show how much **Wred** can improve over **Isum**.

In the first row of Figure 12, we report the maximum absolute improvement (y-axis) that each method obtains given a constraint on the minimum speedup required (x-axis). As expected, as the minimum required speedup increases, from left to right, all methods (except **Orig**) gradually lose improvement. Comparing **Wred** with **Isum**, **Isum** is generally better than **Wred** at lower speedups, and the outcome changes in some workloads (TPC-H, REAL-L, STATS) when the required speedup gets higher, showing the ability of reduction to increase the speedup higher than compression. For REAL-SM and REAL-M, **Wred** alone cannot produce high speedups. On TPC-H×100, **Wred** is always worse than **Isum**, as compression eliminates many redundant instances, while **Wred** maintains all instances. In general, except for REAL-L, where **Wred** is consistently better than **Isum** at speedups higher than 2, **Wred** and **Isum** in isolation have comparable results.

The results change dramatically for **Isum→Wred**: combining reduction and compression generally gives the best results, especially at higher speedups. On REAL-SM, **Isum→Wred** is able to produce results with higher improvement than **Orig** itself, with a 24x speedup, and 10x speedup on REAL-M. On TPC-H×100, **Isum→Wred** can reach a 20x speedup with a minimal drop in improvement. The only case where **Wred** is better than **Isum→Wred** is for REAL-L, because **Isum** loses a lot of improvement quickly and combining with **Isum** is detrimental.

In the second row of Figure 12, we look at the results from a different angle: we report the maximum speedup (y-axis) that each method obtains subject to a constraint on the maximum allowed improvement loss (x-axis). With higher allowed losses, from left to right, each method (except **Orig**) is able to provide higher speedups. Again, while comparing **Isum** and **Wred** in isolation does not yield a clear winner across workloads, combining the two is generally better. In particular, for REAL-SM and REAL-M, **Isum→Wred** is always better across the spectrum. Across all workloads, **Isum→Wred** obtains a 10.5× median speedup with a 5% median improvement loss.

6.6 Effects of Reduction Improvements

Cross-product prevention (Section 4.1) is of paramount importance. Without it, some reduced queries in TPC-DS have an explosion in cost by several orders of magnitude, making them, by far, the most costly queries in the workload. The advisor wastes all its time tuning these queries, for which no index can help because of the cross-products, resulting in 0% improvement.

The technique presented in Section 4.2 helps mitigate the effects of losing join keys due to the elimination of expressions from a query. In Figure 13, we show the improvements obtained by

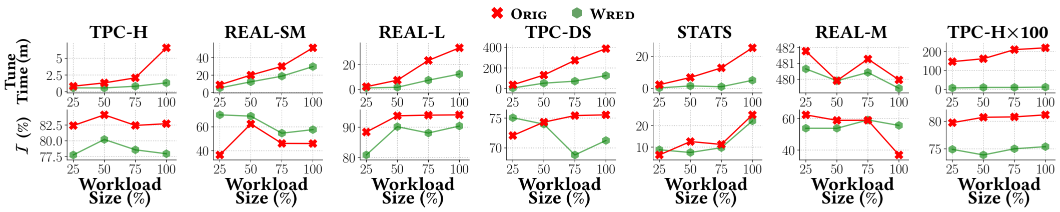


Fig. 14. Scalability of **WRED-AUTO** with increasing workload size (% of original workload). Our method scales well with more queries.

WRED-AUTO with and without mitigation. In all workloads, the mitigation strategy never worsens the results, and in some cases (e.g., REAL-SM, TPC-DS) it improves the results substantially.

6.7 Scalability with Workload Size

In this experiment, we test the ability of workload compression to scale with the number of queries in the workload. We construct workloads of increasing sizes by randomly sub-sampling queries from original workloads, and we run **ORIG** and **WRED-AUTO**. Figure 14 shows that **WRED-AUTO** scales well with workload size, with better tuning time than **ORIG** while maintaining high improvement.

6.8 Applicability to a Different Advisor

To showcase the applicability of workload reduction to a different system and advisor, we run **WRED-AUTO** on PostgreSQL Dexter [20, 22], with default settings, on the TPC-H, TPC-H×100, and TPC-DS workloads. On TPC-H, **WRED** obtains a 1.5x speedup with a 4% improvement loss. The loss is still within an acceptable range, but the speedup is less than that on Microsoft SQL Server. However, the configuration produced by Dexter only gives 17% improvement when tested on Microsoft SQL Server, whereas DTA yields 81%—to make it a fair comparison, we created the indexes recommended by Dexter as hypothetical indexes inside Microsoft SQL Server; we then made what-if calls to the Microsoft SQL Server query optimizer to obtain estimated costs of the queries under the hypothetical configuration given by Dexter. On TPC-H×100, **WRED** obtains a 2.2x speedup with a higher 8% loss, indicating a greater struggle of **WRED** on Dexter to tune an instance-heavy workload. Finally, on TPC-DS, we observe a 3.4x speedup with a 3% loss in improvement, also in line with DTA.

6.9 Improvements in Query Execution Time

We conclude our experiments by analyzing the ability of **ORIG** and **WRED** to improve actual execution time of the queries, rather than their estimated costs. Notice, however, that neither **ORIG** nor **WRED** aims at ensuring that the execution-based and the cost-based improvements are the same. A discrepancy between the two is likely, due to well-known challenges in accurate cost modeling (e.g., cardinality estimation errors) [29, 32, 41, 42]—fixing this discrepancy is beyond the scope of this work. The purpose of this experiment is to show that **WRED** does not drastically diverge from **ORIG** in terms of improvement over query execution time.

In Figure 15, we measure the *absolute improvement per query* (see $\mathcal{I}(Q_i, C)$ from Section 2.1), in seconds, using query *execution* CPU time instead of optimizer estimated cost. We sort queries by the absolute difference between the **ORIG** and **WRED** execution times, where queries with the most similar times are on the left of the figure. For larger workloads such as STATS (which includes 146 queries), we only plot queries for which the absolute difference in execution time is above the 5-th percentile of all the execution times. For example, for STATS, we omit queries where **ORIG** and **WRED** differ by less than 62 ms. We present results for five of our workloads. The results show

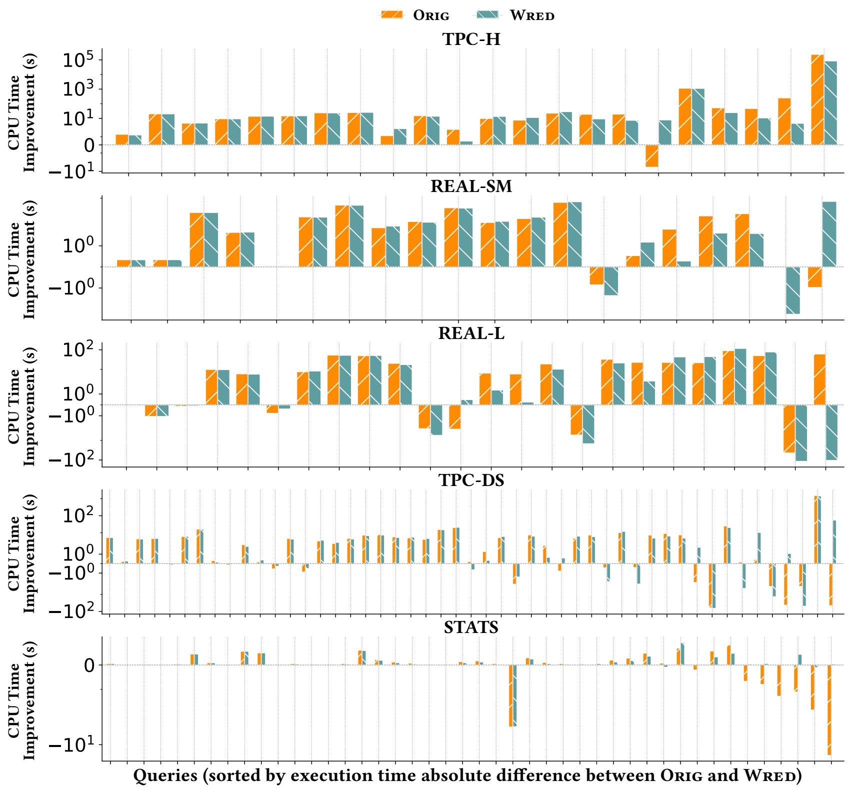


Fig. 15. Query execution absolute improvement (in seconds) of **Orig** and **Wred**. A negative value indicates regression. Queries are sorted by the absolute difference between **Orig**'s and **Wred**'s execution times (most similar on the left).

that **Wred** provides similar improvements as **Orig** for most queries, in line with the results using estimated cost. The median improvement loss across all workloads is 3%, in line with the loss in terms of estimated cost. When regressions occur (seen as negative improvements), **Wred** and **Orig** have generally similar regressions; there are cases where **Wred** has less regression than **Orig**, or vice versa; there are also a few queries where **Orig** regresses and **Wred** improves, or vice versa. These cases are mostly fortuitous, since neither **Orig** nor **Wred** implements techniques that mitigate regressions [14].

7 RELATED WORK

There has been substantial prior work on workload compression, both general and indexing-specific. Among general techniques for SQL workloads, GSUM [13] uses an efficient greedy algorithm that maximizes feature coverage while maintaining a similar distribution to the entire workload. However, GSUM is not effective for index tuning since it ignores the features more relevant to index tuning and potential performance improvement of queries. Jain et al. proposed a machine learning approach [19] for workload compression by training a model specifically for SQL queries. However, this technique requires expensive preprocessing to train the models.

Among indexing-specific, prior work has proposed clustering-based approaches [1, 8, 27] that group similar queries and samples from each cluster. However, these approaches do not scale to large workloads and the distance measures used for comparing queries are less effective in characterizing similarity between queries with varying templates. To address these issues, **Isum** [34] uses an efficient technique that greedily selects queries using indexing-specific featurization and comparison mechanisms. We use **Isum** as a baseline for our work in this paper. As discussed earlier, **Isum** does not rewrite the selected queries, and therefore their what-if call times can still be high. **Wred**, instead, speeds up each and every what-if call by simplifying the queries. As we show in our experiments, combining **Isum** and **Wred** offers the best results.

There have been other mechanisms to improve the scalability of index advisors, which are orthogonal to our line of work. [17, 30] reduce optimizer calls by caching and reusing costs of sub-expressions across queries. [4, 5] compute the bounds on costs of queries based on query optimization of past configurations, which can be used for pruning optimizer calls. There have been recent machine learning techniques that prune candidate configurations and estimate costs based on what-if calls over similar query-configuration pairs [35] as well as ones that use machine learning techniques to improve the search [43]. In this work, we focus on a complementary solution that can be used as a pre-processing step before index tuning.

Query synthesis is a well-studied problem: the goal of *NLP-to-SQL* [25, 44] is to create queries from natural language; *workload synthesis* attempts to create queries that are similar to other queries in the workload benchmarking [39]; in *query by example* [15, 28, 40], the goal is to create queries from input-output examples; *query rewriting* [6, 31, 45] creates queries with identical semantics. While the objectives are different than ours and these techniques typically have extra constraints such as maintaining the query semantics, the challenges related to adding, removing, and transforming sub-expressions in complex SQL queries are shared. There is also work on simplifying the SQL syntax to help natural-language-to-SQL translation models [16]; these methods do not simplify the queries by removing expressions and aim at maintaining query semantics.

8 DISCUSSION AND LIMITATIONS

In this section, we discuss possible future improvements of **Wred** to use column statistics generated internally by some index advisors, to recommend materialized views in addition to indexes, and to seamlessly generalize workload compression by simultaneously performing both workload reduction and workload compression, rather than in sequence.

Dependency on the index advisor internals. The internals of the index advisor play an important role in the success of workload reduction. In particular, workload reduction relies on the fact that the advisor employs query optimizer estimated costs to determine good candidate indexes and guide the search for the best index configuration. This is not a stringent requirement, as most of the modern index tuners follow this paradigm [1, 22, 26].

Moreover, most modern advisors use a notion of *indexable columns* to guide candidate index selection, where only columns that are likely to benefit from indexes are included [1, 26]. Workload reduction is well-suited to help these advisors as it uses a similar notion in its column selection step (Section 3.1). Some advisors, such as DTA, further use column statistics information to distinguish the importance of indexable columns when generating candidate indexes. Such information is currently not used by **Wred**, which may be useful for further pruning the columns selected. This may lead to more reduction of the queries, e.g., by omitting columns that are not crucial based on the statistics information. We leave this investigation of improving **Wred** by incorporating deeper knowledge of index advisor internals as interesting future work.

Extension to materialized views. There is a potential to expand the scope of workload reduction in conjunction with other physical design optimizations, such as the selection of materialized views. For example, DTA, which supports recommending both indexes and materialized views, relies on a similar cost-based approach to find candidate sub-expressions (i.e., views) in a query and across queries in the workload for materialization [1, 2]. Instead of keeping important indexable tables/columns (as **Wred** currently does), the focus needs to be shifted to identifying important *subsets* of tables/columns at workload-level where view materialization on top of these subsets can significantly reduce cost of query processing. Moreover, to ensure that the materialized views can be used by the original queries (not only the reduced queries), one needs to preserve more query semantics when reducing from the original queries. These new challenges require making tangible changes to the query rewriting algorithms presented in Section 3.2. Therefore, we leave such an extension of **Wred** for future work.

Generalization of workload reduction. We sketch a more general formulation of workload reduction that can subsume workload compression in the extended version [3]. Additionally, we study the complexity and optimality compared to the formulation presented in this paper, and pave the way for potential opportunities for algorithms that simultaneously perform both workload compression and reduction.

9 CONCLUSION

In this paper, we introduced *workload reduction* to rewrite complex queries into simpler ones that speed up index tuning by decreasing individual what-if call times. Our workload reduction method, **Wred**, efficiently reduces a workload by keeping the expressions that most need indexing. Our experiments on a variety of benchmarks and real-world customer workloads using state-of-the-art index tuners show that **Wred** is able to significantly speed up the index tuning process with minor losses in index quality, and that combining it with workload compression offers even higher speedups with comparable losses.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, Arvind Arasu, Bailu Ding, Anshuman Dutt, and Arnd Christian König for their valuable and constructive feedback.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 930–932.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R Narasayya. 2000. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, Vol. 2000. 496–505.

[3] Matteo Brucato, Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2024. Wred: Workload Reduction for Scalable Index Tuning (extended version). https://matteo-brucato.github.io/files/wred_extended.

[4] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*. 227–238.

[5] Nicolas Bruno and Surajit Chaudhuri. 2006. To tune or not to tune? A Lightweight Physical Design Alerter. In *Proceedings of the 32nd international conference on Very large data bases*. Citeseer, 499–510.

[6] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. 2000. What is query rewriting?. In *Cooperative Information Agents IV-The Future of Information Agents in Cyberspace: 4th International Workshop, CIA 2000, Boston, MA, USA, July 7-9, 2000. Proceedings 4*. Springer, 51–59.

[7] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. 2002. Compressing SQL workloads. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 488–499.

[8] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek R. Narasayya. 2002. Compressing SQL workloads. In *SIGMOD*. 488–499.

[9] Surajit Chaudhuri and Vivek Narasayya. 2020. Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server. (June 2020). https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server/

[10] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB 1997*. 146–155. http://www.vldb.org/conf/1997/P146.PDF

[11] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. 367–378.
[12] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey Naughton, and Stratis Viglas. 2020. Comprehensive and efficient workload compression. *arXiv preprint arXiv:2011.05549* (2020).
[13] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey F. Naughton, and Stratis Viglas. 2020. Comprehensive and Efficient Workload Compression. *Proc. VLDB Endow.* 14, 3 (2020), 418–430.
[14] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD*. 1241–1258.
[15] Anna Fariha, Sheikh Muhammad Sarwar, and Alexandra Meliou. 2018. SQuID: Semantic similarity-aware query intent discovery. In *Proceedings of the 2018 International Conference on Management of Data*. 1745–1748.
[16] Yujian Gan, Xinyun Chen, Jinxia Xie, Matthew Purver, John R Woodward, John Drake, and Qiaofu Zhang. 2021. Natural SQL: Making SQL easier to infer from natural language specifications. *arXiv preprint arXiv:2109.05153* (2021).
[17] Antara Ghosh, Jignashu Parikh, Vibhuti S Sengar, and Jayant R Haritsa. 2002. Plan selection based on query clustering. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 179–190.
[18] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proc. VLDB Endow.* 15, 4 (2021), 752–765. https://doi.org/10.14778/3503585.3503586
[19] Shrainik Jain, Bill Howe, Jiaqi Yan, and Thierry Cruanes. 2018. Query2Vec: An Evaluation of NLP Techniques for Generalized Workload Analytics. *arXiv preprint arXiv:1801.05613* (2018).
[20] Andrew Kane. 2017. The automatic indexer for Postgres. https://github.com/ankane/dexter.
[21] Andrew Kane. 2017. Introducing Dexter, the Automatic Indexer for Postgres. (June 2017). https://medium.com/@ankane/introducing-dexter-the-automatic-indexer-for-postgres-5f8fa8b28f27 last accessed November 2022.
[22] Andrew Kane. 2017. Introducing Dexter, the Automatic Indexer for Postgres. https://medium.com/@ankane/introducing-dexter-the-automatic-indexer-for-postgres-5f8fa8b28f27.
[23] Richard M Karp. 1972. Reducibility among combinatorial problems, Complexity of computer computations (RE Miller and JW Thatcher, editors).
[24] Richard M Karp. 2010. *Reducibility among combinatorial problems*. Springer.
[25] Hyeonji Kim, Byeong-Hoon So, Wook-Shin Han, and Hongrae Lee. 2020. Natural language to SQL: Where are we today? *Proceedings of the VLDB Endowment* 13, 10 (2020), 1737–1750.
[26] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic Mirror in My Hand, Which is The Best in The Land? An Experimental Evaluation of Index Selection Algorithms. *VLDB* 13, 12 (2020), 2382–2395. http://www.vldb.org/pvldb/vol13/p2382-kossmann.pdf
[27] Gokhan Kul, Duc Luong, Ting Xie, Patrick Coonan, Varun Chandola, Oliver Kennedy, and Shambhu Upadhyaya. 2016. Ettu: Analyzing query intents in corporate databases. In *Proceedings of the 25th international conference companion on world wide web*. 463–466.
[28] Hao Li, Chee-Yong Chan, and David Maier. 2015. Query from examples: An iterative, data-driven approach to query construction. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2158–2169.
[29] Jiexing Li, Arnd Christian König, Vivek Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries Using Statistical Techniques. *Proc. VLDB Endow.* 5, 11 (jul 2012), 1555–1566. https://doi.org/10.14778/2350229.2350269
[30] Stratos Papadomanolakis, Debabrata Dash, and Anastasia Ailamaki. 2007. Efficient use of the query optimizer for automated physical design. In *Proceedings of the 33rd international conference on Very large data bases*. 1093–1104.
[31] Yannis Papakonstantinou and Vasilis Vassalos. 1999. Query rewriting for semistructured data. *ACM SIGMOD Record* 28, 2 (1999), 455–466.
[32] Debjyoti Paul, Jie Cao, Feifei Li, and Vivek Srikumar. 2021. Database Workload Characterization with Query Plan Encoders. *Proc. VLDB Endow.* 15, 4 (dec 2021), 923–935. https://doi.org/10.14778/3503585.3503600
[33] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. 2011. Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*. 166–171. https://doi.org/10.1109/ICDCSW.2011.20
[34] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. In *SIGMOD 2022*. 660–673. https://doi.org/10.1145/3514221.3526152
[35] Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2022. DISTILL: low-overhead data-driven techniques for filtering and costing indexes for scalable index tuning. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2019–2031.
[36] The TPC Benchmark™DS 2023. http://www.tpc.org/tpcds/.
[37] The TPC Benchmark™H 2023. http://www.tpc.org/tpch/.

[38] G. Valentin, M. Zuliani, D.C. Zilio, G. Lohman, and A. Skelley. 2000. DB2 advisor: an optimizer smart enough to recommend its own indexes. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073).* 101–110. https://doi.org/10.1109/ICDE.2000.839397

[39] Chengcheng Wan, Yiwen Zhu, Joyce Cahoon, Wenjing Wang, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Alexandra Ciortea, Konstantinos Karanasos, et al. 2023. Stitcher: Learned Workload Synthesis from Historical Performance Footprints. (2023).

[40] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive query synthesis from input-output examples. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 1631–1634.

[41] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F. Naughton. 2013. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *Proc. VLDB Endow.* 6, 10 (aug 2013), 925–936. https://doi.org/10.14778/2536206.2536219

[42] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *2013 IEEE 29th International Conference on Data Engineering (ICDE).* 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899

[43] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. 2022. Budget-Aware Index Tuning with Reinforcement Learning. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22).* Association for Computing Machinery, New York, NY, USA, 1528–1541. https://doi.org/10.1145/3514221.3526128

[44] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 63:1–63:26. https://doi.org/10.1145/3133887

[45] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System Using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (sep 2021), 46–58. https://doi.org/10.14778/3485450.3485456

Received July 2023; revised October 2023; accepted November 2023