GEqO: ML-Accelerated Semantic Equivalence Detection

Brandon Haynes
brandon.haynes@microsoft.com
Microsoft Gray Systems Lab
USA

Rana Alotaibi
ranaalotaibi@microsoft.com
Microsoft Gray Systems Lab
USA

Anna Pavlenko
anna.pavlenko@microsoft.com
Microsoft Gray Systems Lab
USA

Jyoti Leeka
jyoti.leeka@microsoft.com
Microsoft
USA

Alekh Jindal*
alekh@smart-apps.ai
SmartApps
USA

Yuanyuan Tian
yuanyuantian@microsoft.com
Microsoft Gray Systems Lab
USA

ABSTRACT

Large scale analytics engines have become a core dependency for modern data-driven enterprises to derive business insights and drive actions. These engines support a large number of analytic jobs processing huge volumes of data on a daily basis, and workloads are often inundated with overlapping computations across multiple jobs. Reusing common computation is crucial for efficient cluster resource utilization and reducing job execution time. Detecting common computation is the first and key step for reducing this computational redundancy. However, detecting equivalence on large-scale analytics engines requires *efficient* and *scalable* solutions that are fully *automated*. In addition, to maximize computation reuse, equivalence needs to be detected at the *semantic* level instead of just the *syntactic* level (i.e., the ability to detect semantic equivalence of seemingly different-looking queries). Unfortunately, existing solutions fall short of satisfying these requirements.

In this paper, we take a major step towards filling this gap by proposing GEqO, a portable and lightweight machine-learning-based framework for efficiently identifying *semantically* equivalent computations at scale. GEqO introduces two machine-learning-based filters that quickly prune out nonequivalent subexpressions and employs a semi-supervised learning feedback loop to iteratively improve its model with an intelligent sampling mechanism. Further, with its novel *database-agnostic* featurization method, GEqO can *transfer* the learning from one workload and database to another. Our extensive empirical evaluation shows that, on TPC-DS-like queries, GEqO yields significant performance gains—up to 200× faster than automated verifiers—and finds up to 2× more equivalences than optimizer and signature-based equivalence approaches.

1 INTRODUCTION

Modern data-driven enterprises fundamentally rely on large-scale analytics engines (e.g., Spark [7], SCOPE [52], Synapse [38], BigQuery [20], Redshift [23]) to derive business insights and drive actions. Concretely, engines such as SCOPE process exabytes of data and execute millions of jobs, with trillions of operators [31] per cluster [56]. Computational redundancy within these analytics engines is strikingly common [31, 51], where intermediate results are duplicated across different queries (i.e., they contain *equivalent subexpressions*). According to Jindal et al. [32], about 40% of the jobs in SCOPE contain equivalent subexpressions (i.e., at least one subexpression is equivalent to a subexpression in another job).

Because of this pervasive redundancy, identifying and reusing common computation has long been recognized as a critical technique to improve query performance and reduce operational costs. For example, a wide range of tools and approaches for leveraging materialized views have been developed, including CloudViews [30], Google Napa [2], and Redshift AutoMV [5]. Common computation reuse has also been exploited for multi-query optimization [46, 48] in the context of multiple-query-at-a-time systems.

For all these tools and techniques, detecting equivalent subexpressions is the first and crucial step. For example, view selection algorithms (e.g. [3]) maximize the benefit of materializing computation that is most redundant in cost or frequency of use, under a storage or maintenance cost constraint. Similarly, view matching relies on detecting and leveraging equivalent views to improve query performance. At the query level, identifying equivalence is also a crucial step in efficient rewriting (either automatically by an optimizer or manually by a DBA), where a query is transformed into an equivalent—but better-performing—variant [21, 22]. Finally, determining query equivalence is also important in generating functional or performance tests for database implementations [34, 50].

In this paper, we focus on the problem of *detecting subexpression equivalence at scale*.[1] There are a number of distinct challenges in doing so. First, the detection process must be *automatic* due to the sheer number of developers and jobs involved. Second, scalability is crucial as quadratic pairwise comparison over trillions of subexpressions is intractable in most current solutions. Third, to maximize computation reuse, equivalence detection needs to be sufficiently *general* to identify common computation expressed in different ways by different users. This means that a detection algorithm should go beyond merely "judging a book by its cover" (i.e., only identifying superficially- or syntactically-equivalent subexpressions) but rather "look beneath the surface" to detect *semantic equivalence* between subexpressions with dissimilar structures. Figure 1 shows such an example, where the highlighted subexpressions differ syntactically but are nonetheless semantically equivalent.

Existing approaches to detecting subexpression equivalence do not address all of the above challenges. Optimizer-based approaches, which are used by many classical materialized view selection and matching algorithms [3, 18], defer to the query optimizer to detect equivalence. This approach lacks generality, given that even

---

[1]This work does not propose a novel view selection or rewriting algorithm. Rather, it presents a framework designed to accelerate equivalence detection, which is considered a fundamental step for these and other algorithms.

```
Q1: SELECT y, AVG(x) FROM (
        SELECT A.x, B.y FROM A, B
        WHERE A.joinKey = B.joinKey
          AND A.val > B.val + 10
          AND B.val > 10
    ) GROUP BY B.y

Q2: SELECT SUM(x), SUM(y) FROM (
        SELECT A.x, B.y FROM B, A
        WHERE B.joinKey = A.joinKey
          AND B.val + 10 < A.val
          AND B.val + 10 > 20
          AND A.val > 20
    )
```

Figure 1: Two queries that contain semantically-equivalent subexpressions highlighted by shaded boxes.

highly-mature optimizers such as SQL Server are missing equivalence rules necessary to identify common scenarios [50]. It is also inefficient given cloud-scale volumes of complex queries, where the query optimizer quickly becomes a bottleneck. Manual approaches, commonly used in many relational OLAP databases—including state-of-the-art cloud-based analytics systems like Snowflake [12], BigQuery, and NAPA— require users to manually identify common computations and create materialized views, which is error-prone, tedious and simply does not scale. Signature-based view materialization approaches, like CloudViews [30], use Merkle tree-like signatures for efficient identification of syntactically-identical subexpressions. However, this approach sacrifices completeness as it may miss opportunities for identifying semantically-equivalent subexpressions, as illustrated in Figure 1. At the other end of the spectrum, verification-based approaches, such as Cosette [10] and SPES [54], formally prove the semantic equivalence of queries using automated proof assistants or SMT solvers. While these approaches are highly effective, they suffer from scalability issues. Exhaustively evaluating all pairs of subexpressions over a single day of jobs at cloud-scale would require over a trillion expensive formal verifications and more than a century of compute time!

In this paper, we introduce *GEqO* (a **G**eneral **Eq**uivalence **O**ptimizer), which addresses the aforementioned challenges. GEqO is a general framework for efficiently identifying semantically-equivalent subexpressions at scale. It applies a series of *equivalence filters* to sets of subexpressions, enabling accelerated detection. To ensure correctness, GEqO finally applies an expensive formal verifier—but only after filtering most nonequivalent subexpressions, which constitute the vast majority of the pairs. As a result, GEqO produces subexpression pairs that are, with perfect precision and near-perfect recall, semantically equivalent.

A desirable equivalence filter has two important properties: it should (i) admit virtually all of the equivalences (i.e., exhibit a high true positive rate; TPR) and (ii) reject most non-equivalences (i.e., have a high true negative rate; TNR). Table 1 illustrates this for GEqO's filters (detailed below), where the TPR is near-perfect, and the TNR steadily increases until all negatives have been eliminated.

To maximize performance, GEqO arranges filters to rapidly reject "easy" nonequivalent subexpression pairs, with faster filters applied

first, as shown in Table 1. Slower but increasingly complex filters are then applied to identify more difficult cases. This trade-off allows GEqO to achieve performance close to optimal, assuming an oracle that verifies only equivalent pairs, and is almost 200× faster than verifying all subexpression pairs.

While prior work has established quick-but-low-precision heuristic-based filters—i.e., matching common table and column sets [18], which we refer to as *schema filter (SF)* in Table 1—and expensive *automated verifiers* (*AV* in Table 1) that are slow with perfect precision, there currently exists no "middle ground": a way to filter non-equivalent subexpressions rapidly with high precision. GEqO fills this gap by introducing two such filters.

First, GEqO's *vector matching filter (VMF)* embeds subexpressions in a learned vector space and identifies likely equivalent pairs by applying an approximate nearest neighbor search (ANNS). ANNS is a popular, high-performance technique [16, 45] with moderate precision. GEqO leverages the *VMF* to efficiently prune moderately-difficult cases not handled by the SF, while at the same time ensuring that equivalence pairs are admitted with high recall.

Next, GEqO's *equivalence model filter* (*EMF*) employs a novel, high-precision, supervised ML model trained over a workload sample to predict semantic equivalence. As we detail below, the *EMF* is database- and schema-agnostic and can be easily transferred to other workloads. As far as we are aware, GEqO is the first work to present a machine-learning-accelerated framework for detecting semantic equivalence at scale.

A key challenge in training the *EMF* is the need for large amounts of labeled data. Although the cloud makes collecting query workloads much more accessible, labeling the equivalent subexpressions within the workload requires running expensive equivalence verifiers on all subexpression pairs (i.e., trillions of invocations). To reduce this cost, GEqO employs a semi-supervised feedback loop (*SSFL*) pipeline that iteratively improves the accuracy of the *EMF* until it matures. The *SSFL* employs inexpensive filters (i.e., the SF and *VMF*) to ensure approximately balanced classes in its generated training data. This approach enables GEqO to both avoid the cold start training problem and fine-tune its *EMF* model as new workload data becomes available for training.

A second challenge addressed by GEqO involves ensuring that its learned *EMF* model is not tied to a fixed database schema. For example, the *EMF* should be able to determine that the subexpressions shown in Figure 1 are equivalent even if table A's name was replaced with C. Unlike existing instance-based ML-for-DB solutions [25], GEqO uses a database and schema-agnostic approach that focuses on learning general semantic equivalence patterns. It accomplishes this during *EMF* featurization by replacing references to database schema with symbolic correspondences. This allows GEqO to pretrain on existing database workloads and apply the resulting model to new database workloads.

GEqO is a standalone framework that can be used alongside a query optimizer to complement its ability to detect equivalent computation. Unlike adding new rewrite rules, which requires changing the core database engine code, GEqO can learn any equivalence relationship in a workload, including those missed by the optimizer. We focus on subexpressions that contain selections, projections, and joins (i.e., SPJ subexpressions) with conjunctive predicates.

Table 1: Performance of GEqO and its filters (*SF, VMF, EMF*) on ~50k subexpression pairs and 50 equivalences generated using a TPC-DS schema; see §7.5.

Filter	Time (sec)	TPR	TNR	Complexity
Schema Filter (*SF*)	0.3	0.98	0.37	$O(n)$
Vector Matching Filter (*VMF*)	0.5	0.98	0.66	$O(n \log n)$
Equivalence Model Filter (*EMF*)	1.3	0.98	0.80	$O(n^3)$
Automated Verifier (*AV*)	898.5	1.00	1.00	$O(n \cdot 2^{\Omega(\gamma)})$
GEqO	**3.1**	**0.93**	**1.00**	$O(\epsilon \cdot 2^{\Omega(\gamma)}) + opt$
Oracle + AV	1.0	1.00	1.00	$opt = O(|E| \cdot 2^{\Omega(\gamma)})$

We report true positive rate (TPR) and true negative rate (TNR). The "Oracle+AV" row shows a hypothetical optimal case where an oracle correctly identifies all equivalent pairs, which are then verified. We assume a verifier with perfect recall. n is the number of subexpressions, γ is the number of symbols in the AV's SAT formulation, and E is the set of equivalent subexpression pairs. GEqO verifies ϵ more pairs than the oracle, which we empirically show to be ~5–10% (see Section 7.5).

Through detailed experiments, we demonstrate the efficiency and effectiveness of GEqO in detecting common computations. We systematically evaluate its filters and discuss their trade-offs between prediction accuracy and the overhead involved.

Contributions. The paper makes the following contributions:

- We introduce GEqO, a scalable ML-framework for detecting semantically-equivalent subexpressions. GEqO's novel *VMF* and *EMF* filters quickly prune non-equivalent subexpressions, reducing the overhead of running expensive equivalence verifiers by up to 200× (§2).
- We introduce a database agnostic featurization technique that generalizes instance-specific (non)equivalent subexpression pairs into (non)equivalent patterns, making the *EMF* transferable to new workloads and databases (§4).
- We address the challenge of requiring large volumes of labeled training data by introducing a semi-supervised feedback loop (*SSFL*) to iteratively improve GEqO's *EMF* filter (§6). This process is aided by drawing high-quality samples leveraging the cheaper *SF* and *VMF* filters.
- Our evaluation demonstrates the efficiency and effectiveness of GEqO (§7).

2 PRELIMINARIES AND OVERVIEW

This section defines the key concepts used in the paper and provides an overview of GEqO.

2.1 Problem definition

GEqO assumes that a SQL query can be transformed into a tree (i.e., a logical plan) Q consisting of operator nodes (we use $ops(Q)$ to denote the set of all operators in Q). We term each subtree rooted at node i to be a *subexpression* q_i of Q. Let $S(Q) = \{q_1, ..., q_n\}$ be the set of all subexpressions induced by Q. Note that $Q \in S(Q)$; the root of the logical plan is itself a (trivial) subexpression of Q.

GEqO assumes that as a subtree in a logical query plan, subexpressions are unambiguously executable. Let $q_i(d)$ denote the result of executing subexpression q_i on some database instance d. Let D be the set of all database instances. Given two subexpressions q_i and q_j, they are *semantically equivalent*, denoted as $q_i \equiv q_j$, if and only if $\forall d \in D, q_i(d) = q_j(d)$. Note that q_i and q_j need not be

drawn from the same query Q, and that this definition holds under both set and bag semantics [11].

An *equivalence verifier* applies an automated technique (e.g., a proof assistant [14] or formal solver [17]) to decide $q_i \equiv q_j$. We denote equivalence determined using an automated verifier AV as $q_i \overset{\text{AV}}{\equiv} q_j$. A verifier is correct but not complete (i.e., $(q_i \overset{\text{AV}}{\equiv} q_j) \Rightarrow (q_i \equiv q_j)$ but $(q_i \equiv q_j) \not\Rightarrow (q_i \overset{\text{AV}}{\equiv} q_j)$) and in general run in exponential time. Finally, given a pair of subexpressions, an *equivalence filter* applies a model, heuristic, or similar technique to approximately decide equivalence (i.e., *pseudo-equivalence*). In GEqO, filters trade off speed and precision to reduce the false positives that must be checked by an equivalence verifier. We denote pairwise pseudo-equivalence determined using a filter f as $q_i \overset{\text{f}}{\approx} q_j$.

Given the above, we now formally define the core problem addressed by GEqO:

PROBLEM (WORKLOAD EQUIVALENCE). *Given a workload $W = \{q_1, ..., q_n\}$ of subexpressions, GEqO approximates $E(W) = \{(q_i, q_j) \in W \times W \mid q_i \equiv q_j\}$, i.e., the equivalence set amongst all the pairwise combinations of subexpressions in W.*

There are two important special cases of the workload equivalence problem. In the first case, the workload just has a pair of subexpressions $W = \{q_i, q_j\}$. The task reduces to just detecting *pairwise equivalence* ($q_i \equiv q_j$). This version of the problem is common for applications such as query rewriting or view matching. The second special case is when the input is a set of queries $\{Q^1, ...Q^m\}$. Then the workload is the enumeration of all the subexpressions of the input queries, i.e., $W = \bigcup_k S(Q^k)$. This formulation is of critical importance to applications such as view recommendation, when the goal is to find common computation among a large set of queries. Although GEqO can handle pairwise equivalence detection very well, it is designed more as an efficient and scalable solution for supporting general workload equivalence when the workload set W is large (which includes the second special use case).

Figure 2: GEqO Overview

2.2 GEqO Overview

The overall architecture of GEqO is illustrated in Figure 2. GEqO approximates computing an equivalence set by applying the series of filters $F = \langle f_1, ..., f_n \rangle$ listed in Table 1 to a workload of subexpressions. Filters are applied in decreasing order of speed and increasing order of precision. Each filter is applied to every subexpression pair in the target workload W to approximate the equivalence set. To ensure correctness (e.g., for use in a view materialization algorithm), GEqO utilizes an automated verifier to eliminate false positives from the resulting equivalence set. It is important to note that if a pair is determined to be non-equivalent by a filter, it is not evaluated by subsequent filters and it is not verified (i.e., filters short-circuit).

We formalize the above process with the following two functions:

$$GEqO_{\text{SET}}(W, F) = \{(q_i, q_j) \in W \times W \mid GEqO_{\text{PAIR}}(q_i, q_j, F)\} \quad (1)$$

$$GEqO_{\text{PAIR}}(q_i, q_j, F) = \begin{cases} q_i \overset{AV}{\equiv} q_j & \text{if } F = \emptyset \\ \bot & \text{if } q_i \overset{f_1}{\not\approx} q_j \\ GEqO_{\text{PAIR}}(q_i, q_j, F \setminus f_1) & \text{otherwise} \end{cases} \quad (2)$$

2.2.1 *Detecting an Equivalence Set for a Large Workload*.
We now describe, given a large workload of subexpressions, how GEqO applies the filters in Table 1 to efficiently narrow down the candidate equivalent subexpression pairs, before calling the expensive automated verifier (AV).

The first filter applied is the widely-used *schema filter (SF)*. Subexpressions that access different sets of tables or return different numbers of columns are highly unlikely to be equivalent. Therefore, GEqO groups all subexpressions in the workload based on the tables used and the number of columns returned, resulting in *SF-groups*. From this point forward, only subexpression pairs from the same *SF*-group are considered by subsequent filters.

In the second step, for each *SF*-group, the *vector matching filter (VMF)* embeds the subexpressions in a learned vector space and identifies likely equivalent pairs by employing approximate nearest neighbor search (ANNS). It is formalized as follows:

DEFINITION 2.1 (VECTOR MATCHING FILTER (*VMF*)). *Let $e(q)$ be a function that embeds a subexpression q in a vector space $\mathcal{V}$. Let d be a distance metric on $\mathcal{V}$ and τ be a threshold distance. Given subexpressions q_1 and q_2, let $q_1 \overset{VMF}{\approx} q_2$ when $d(e(v_1), e(v_2)) < \tau$.*

To further improve efficiency, we construct a hierarchical navigable small world (HNSW) index [35], a common approach to applying ANNS at scale [27].

In the third step, GEqO applies the *equivalence model filter (EMF)*, which is a trained deep learning model, to predict whether each candidate subexpression pair from the *VMF* filter are equivalent.

Finally, GEqO utilizes an *automated verifier (AV)* (we leverage SPES [54]) to verify the correctness of the prediction from *EMF*.

Among the filters used in GEqO, both *VMF* and *EMF* are machine learning based. The *EMF* is a deep learning model comprising multiple tree convolutions and fully connected layers. On the other hand, the *VMF* utilizes the learned tree convolution from *EMF* to embed subexpressions into its metric space.

2.3 Equivalence Model Filter (*EMF*) Overview

The *EMF* is a deep learning model trained to classify equivalence. We now briefly describe its training process and the semi-supervised feedback loop (SSFL) to iteratively improve the model.

To train the *EMF*, GEqO first featurizes (§3) and labels a set of subexpression pairs as the training data. Labels are generated using the SPES automated verifier. During featurization, in addition to converting subexpressions to a fixed-length vector representation, the *EMF* applies its database-agnostic (db-agnostic) transformation (§4.2). This transformation replaces references to specific tables and column names with symbolic correspondences between subexpression pairs, generalizing the *EMF* learning from specific examples of (non)equivalent subexpressions to *patterns* of (non)equivalent subexpressions. It also ensures that the model learned on a particular workload and database is *transferable* to other workloads and databases, allowing for user-supplied or synthetically generated initial training workloads (§5).

GEqO employs the *SSFL* as a guardrail against regressions. It monitors the confidence levels of *EMF*'s predictions, and if confidence

falls below a threshold (e.g., due to new or evolving workloads), it iteratively fine-tunes the *EMF* model through the *SSFL* pipeline.

The key challenge in the *SSFL* pipeline is generating high-quality samples with balanced positive and negative examples for model fine-tuning in each iteration. Even a modest workload produces an intractably large training dataset that is quadratic in the number of subexpression pairs—1000 queries with 10 subexpressions each produces a training dataset of almost 100 million pairs! This dataset is also highly imbalanced, since most subexpression pairs are unlikely to be equivalent.

To address this challenge, GEqO employs the cheap *SF* and *VMF* filters to efficiently identify pseudo-equivalent subexpression pairs (i.e., it computes $q_i \overset{SF}{\approx} q_j \wedge q_i \overset{VMF}{\approx} q_j$ over a workload sample). This computation approximates Equation 2 without the verification step (§2.2.1). Together with another set of randomly-generated, likely non-equivalent pairs, they form an approximately-balanced new sample. As before, GEqO labels and applies its db-agnostic transformation to the new sample. It then augments its training dataset with the new data and fine-tunes the *EMF*.

As previously highlighted, GEqO identifies *general* semantic equivalence, agnostic to the underlying database. It therefore does not consider database constraints or other instance-specific metadata. Nonetheless, extending GEqO to incorporate database-specific constraints [15] remains an interesting direction for future work.

2.4 Complexity Analysis of GEqO Filters

This subsection provides a complexity analysis (summarized in Table 1) for applying each GEqO filter on a workload W containing n subexpressions.

Schema Filter (SF). Assuming a constant-sized schema, GEqO groups n subexpressions by the used tables and the number of returned columns in $O(n)$ time.

Vector Matching Filter (VMF). Given that the HNSW index used by the *VMF* has claimed search complexity logarithmic in the number of indexed objects [35], GEqO indexes the workload subexpressions in $O(n)$ time (we assume a constant embedding size; see §7). Next, for each vector, it performs a $O(\log n)$ radius search for neighbors within Euclidean distance τ, with total complexity in $O(n \log n)$.

Equivalence Model Filter (EMF). As shown in §5, GEqO's equivalence model contains two convolution layers followed by three fully connected layers. Its input is a pair of subexpressions, each with $ops(q_i)$ nodes. We assume that there are many more subexpressions in our workload than operators in the largest tree, i.e., $\max\{ops(q_i) \mid q \in W\} \ll n$. Total complexity is thus dominated by the matrix multiplication in the fully connected layers (i.e., $O(n^3)$).[2]

Automated Verification (AV). To ensure correctness, GEqO verifies pairs produced by its filters. GEqO's AV leverages SPES [54], which uses the Z3 SMT prover [13] to check equivalence. A SMT program can be transformed into an equivalent SAT formulation containing γ symbols, which is solvable in $O(2^{\Omega(\gamma)})$ time.

3 FEATURE ENGINEERING

In this section, we describe the features used by the *EMF* (§3.1) and how these features are mechanically featurized (§3.2).

3.1 Feature Selection

After conducting extensive feature analysis, we find that logical plans play the most important role in predicting equivalence, since the logical plan captures the semantics of a subexpression. As a result, in GEqO, we use the logical plans of the subexpression pairs as inputs to the *EMF* model.

We additionally considered leveraging cardinalities as an auxiliary feature. Intuitively, since $q_i \equiv q_j \Rightarrow |q_i| = |q_j|$, this would appear to be a strongly positive signal for equivalence. However, while our initial analysis indicates that cardinalities do improve *EMF* recall, actual subexpression cardinalities are not generally available for use as inputs to GEqO and executing candidate subexpressions to determine cardinality is infeasible at scale. Conversely, estimated cardinalities are quick to compute but yield only marginal benefit to the prediction task. Thus, we exclusively rely on logical plans as input features to *EMF*.

GEqO canonicalizes the conjunctive predicates in selection and join operators by splitting each n-clause predicate into a composite containing n single-clause predicates. For example, GEqO transforms a relational selection operator $\sigma_{x>25 \wedge y<10000}(R)$, into the composite $\sigma_{x>25}(\sigma_{y<10000}(R))$. As a result, each node in the logical plan has at most one selection or join predicate.

3.2 Logical Plan Featurization

Featurizing the tree structure of a logical plan is challenging since it is difficult to express an arbitrarily-shaped, variable-size tree as a fixed-size feature vector without losing the structure of the tree. To address this, we apply a *tree-vector* transformation that converts an arbitrary logical plan into a fixed-length vector [39]. Our transformation is inspired by [37]; however, we use a different encoding for each node in the tree.

Specifically, we first encode each node in the logical plan as a *node vector (NV)*. Each NV has the same size and format (to be described in §4), but the number of NVs (i.e., number of nodes in the logical plan) can vary widely. Given a tree of NVs, GEqO next performs a breath-first traversal of the tree and concatenates each visited NV into a $m \times l$ matrix M, where m is the number of nodes in a logical plan and l is the size of each NV. We finally apply the tree convolution layers of the *EMF*, to be described in §5, which transforms M into a vector of a fixed dimension h that summarizes a subexpression. In our prototype, $h = 128$ bytes. As demonstrated by prior work [36, 37, 40], tree convolution has been proven effective at representing tree-structured SQL query plans for various ML-for-DB tasks.

---

[2]The two convolution layers are each in $O(n^2)$ [49].

Figure 3: Instance-based node vector encoding of an SPJ subexpression. Each operator's metadata is converted to its "vector segment"; unrelated segments are set to zero.

4 LOGICAL PLAN ENCODING

We now detail how GEqO encodes each node in a logical plan as a node vector (NV). We begin by describing an *instance-based encoding*, i.e., one where the encoding is specific to a workload on a particular database instance. Though the specifics vary, this is a common transformation and most existing approaches are instance-based [25, 37, 42]. We then extend this approach to our novel *db-agnostic* encoding which is oblivious to the specific workload or database.

4.1 Instance-Based Encoding

Notation. Our instance-based encoding is inspired by the encoding technique described in [25]. Given a workload W on a database instance, let T_W, C_W, O_W, and J_W respectively represent the set of tables, columns, arithmetic operators (e.g., $\leq, =, \geq, \neq$), and join types ($\bowtie, \ltimes, \rtimes, \bowtie$) referenced in the workload. Let $onehot(e, U)$ produce a one-hot encoded vector of size $|U|$ with entry $e \in U$ set to one, $null(x)$ indicate whether x is null, and $norm(x)$ normalize x over all scalars in a workload.

Encoding method. Figure 3 illustrates the instance-based encoding process. Each NV consists of table, join and selection segments, denoted as V_{table}, V_{join}, and V_{select}, respectively. For a scan operator on table t, GEqO generates the segment $V_{table} = onehot(t, T_W)$. For a selection operator with a predicate referencing a column c, an arithmetic operator o, and up to one constant value v (we perform constant folding prior to encoding), GEqO generates the segment $V_{select} = onehot(c, C_W) \oplus onehot(o, O_W) \oplus norm(v) \oplus null(v)$, where $\oplus$ is the concatenation operation. Finally, a join operator has a join predicate referencing a left-side column c_l, an arithmetic operator o, right-side column c_r, and a join type j. GEqO generates the join segment $V_{join} = onehot(c_l, C_W) \oplus onehot(o, O_W) \oplus onehot(c_r, C_W) \oplus onehot(j, J_W)$.

As is common in ML featurization, we simply concatenate the table, join, and selection segments to form the final vector, i.e $NV = V_{table} \oplus V_{join} \oplus V_{select}$. For a segment that does not apply to a tree node, GEqO sets it to be zero, e.g. the join segment for a non-join operator is all zeros. Note that $|NV| = |T_W| + 3 \cdot |C_W| + 2 \cdot |O_W| + |J_W| + 2$ (in our prototype $|NV| = 210$; see §7).

```
q1: SELECT t1.c3, t2.c3 FROM t1,t2
    WHERE t1.c1 = t2.c1
        AND t1.c2 > t2.c2 + 10
        AND t2.c2 > 10
q2: SELECT t1.c3, t2.c3 FROM t2,t1
    WHERE t2.c1 = t1.c1
        AND t2.c2 + 10 < t1.c2
        AND t2.c2 + 10 > 20
        AND t1.c2 > 20
```

Figure 4: Symbolized versions of the subexpression pairs highlighted in Figure 1.

Reference	Symbol
A	t1
A.joinKey	t1.c1
A.val	t1.c2
A.x	t1.c3
B	t2
B.joinKey	t2.c1
B.val	t2.c2
B.y	t2.c3

Table 2: Symbols generated for queries in Figure 1 under db-agnostic encoding.

4.2 DB-Agnostic Encoding

Instance-based encoding makes sense for solving problems such as cardinality estimation and query optimization, where the solution targets a specific workload on a particular database instance. In contrast, the problem of learning equivalent subexpressions can be reformulated to be *database agnostic*. To motivate, consider the two subexpressions highlighted in Figure 1. If we were to change the table and column names to those shown in Figure 4, the two new subexpressions remain equivalent, even though they are now for a completely different database, workload, or dataset. This observation is the basis for our db-agnostic node vector encoding technique. Conceptually, for each labeled training data point, we generalize the pair of subexpressions into subexpression patterns, and feed these generalized patterns into our model. As a result, we are able to transfer the learning from one workload for a database instance to a different workload on a different database.

For equivalence detection, what really matters is the tables and columns referenced in the pair of subexpressions. Further, in terms of columns, only the columns actually referenced by the join conditions, selection predicates, and projections (instead of all columns from the referenced tables) need to be considered. Moreover, the

Figure 5: Example of converting instance-based to db-agnostic encoding (table segments only)

actual names of the tables and columns are unimportant. As a result, we can convert the tables and columns in a pair of subexpressions into a generic symbolic form to derive their underlying patterns.

GEqO does this by transforming referenced tables into a set of distinct, generic table symbols $\{t_1, ..., t_n\}$ based on an arbitrary lexicographical order (we sort alphanumerically in our implementation). It similarly symbolizes the referenced columns as $c_1,..., c_n$. Table 2 shows this for the example in Figure 4.

With db-agnostic encoding, we set $T'_W = \{t_1, ..., t_n\}$, where n is the maximum number of symbolized table correspondences expected in any workload, and $C'_W = \{t_1.c_1, ..., t_1.c_m, ..., t_n.c_1, ..., t_n.c_m\}$, where m is the maximum number of symbolized column correspondences per table expected in any workload. We can set n and m to be large enough numbers to cover arbitrarily complex subexpressions. However, in general n and $n \times m$ are much smaller than the total number of tables and columns in the workload, respectively.

Following this transformation, we now treat T'_W and C'_W as our new "workload tables" and "workload columns" to replace T_W and C_W described in §4.1. Then to encode, for each pair of subexpressions, we first symbolize each subexpression into symbolic pattern and apply the previously-introduced instance-based encoding to produce a final db-agnostic encoding NV_α of size $|NV_\alpha| = |T'_W| + 3 \cdot |C'_W| + 2 \cdot |O_W| + |J_W| + 2$.

4.2.1 Scaling to large workloads.
The db-agnostic encoding described above is for a pair of subexpressions. The encoding of one subexpression is different depending upon which other subexpression it is paired with during featurization. Given a large workload of n subexpressions, we have to re-compute the encoding for each pair—an $O(n^2)$ computation. In contrast, in the instance-based encoding, the encoding of one subexpression stays unchanged no matter what it is paired with. In other words, we only need to compute the encoding for each subexpression once (an $O(n)$ computation). For offline training, the performance of db-agnostic encoding is not so crucial, but for online inference any improvement in the encoding process is valuable. To speed up the db-agnostic encoding process, we develop an efficient method to quickly convert an instance-based encoding to a db-agnostic encoding. With this approach, we only incur $O(n)$ computation to produce the instance-based encoding, then apply a lightweight converter for each pair of subexpressions.

The converter takes as input the instance-based tree matrices for both subexpressions. For each subexpression, it first projects out

the submatrix $S_T = M[T]$ that corresponds to the table segment, and the submatrices $S_{C_s} = M[C_s]$, $S_{C_l} = M[C_l]$, and $S_{C_r} = M[C_r]$ that correspond to the column encoding from the selection segment, the left-side column encoding, and the right-side column encoding from the join segment. We first union the column submatrices by applying bit-wise *or* to compute the column submatrix $S_C = S_{C_s} \vee S_{C_l} \vee S_{C_r}$. For the table submatrix S_T, we compute a vector r that represents the column-wise union of the tables referenced in *each* subexpression, with $r_j = \bigvee_i S_T[i, j]$. Next, we generate a mask m_T by unioning the vectors from both subexpressions; this represents all tables referenced in either of subexpressions. We then apply this mask on S_T from both subexpressions to eliminate matrix columns corresponding to unreferenced tables. The resulting submatrix is S'_T. We apply the same process on the column matrices to compute the mask m_C, and use m_C to eliminate matrix columns corresponding to unreferenced table columns from S_{C_s}, S_{C_l}, and S_{C_r} for each subexpression, resulting in S'_{C_s}, S'_{C_l}, and S'_{C_r}. Finally, for each subexpression in the pair, we replace the submatricies S_T, S_{C_s}, S_{C_l}, and S_{C_r} with their transformed variants. The result is a pair of db-agnostic tree matrices M_α which eliminates references not found in either subexpression. Figure 5 illustrates the conversion process for table fragments in a pair of subexpressions.

Through experiments, we find that applying the converter described above is 1.8× faster than computing pairwise db-agnostic encodings from scratch.

4.2.2 Tensor-based extensions.
In the previous subsection we described the db-agnostic encoding process for a single pair of subexpressions. We now describe two tensor-based extensions.

Batch pairwise encoding. Db-agnostic encoding can be easily extended to support batch encoding n pairs. To do so, rather than performing n discrete operations on pairs of two-dimensional submatrices of size $|q_i| \times |X|$—where $|q_i|$ represents the number of tree nodes in a subexpression's logical plan and X is a table or column segments of the NV—GEqO represents the batch as a pair of three-dimensional tensors of size $\max(|q_i|) \times |X| \times n$. Subexpressions with fewer than $\max(q_i)$ operators are zero-padded, which does not affect correctness.

The resulting tensor is amenable to being operated on using tensor-oriented frameworks such as PyTorch [43]. As we describe in Section 5, the *EMF* batch-converts workloads using this approach.

Generalizing from pairs to *n*-subexpressions. The db-agnostic transformation we describe above is a binary operation

Figure 6: The *EMF* architecture. Instance-based encoding (NV) (§4.1); db-agnostic encoding (NV_α) (§4.2).

over two subexpressions. A second generalization involves extending it to be an *n*-ary operation over many subexpressions. This extension only impacts the computation of the mask (e.g., in Figure 5, rather than $r_1 \vee r_2$ we compute $r_1 \vee ... \vee r_n$); other operations are unchanged. This extension is also amenable to tensor execution.

In fact, this *n*-ary, tensor-based encoding is used in the *VMF* filter in GEqO. Recall that after applying the *SF* filter, GEqO groups input workload into *SF*-groups based on tables accessed and the number of columns returned (§2.2.1). In the *VMF*, GEqO then applies the *n*-ary db-agnostic transformation to all subexpressions in each *SF*-group. It then convolves the encoded node vectors to produce a fixed-size vector for each subexpression (§3.2). It finally conducts an approximate nearest-neighbor search (ANNS) on the resulting vectors to identify candidate subexpression pairs that are likely to be equivalent. Note that all subexpressions in an *SF*-group access the same set of tables. Therefore, the group-based db-agnostic encodings for subexpressions from the same *SF*-group approximate their pairwise db-agnostic counterparts.

5 EQUIVALENCE MODEL FILTER (*EMF*)

In this section, we discuss the architecture and training process of the equivalence model filter (*EMF*). Recall from §2.3 that the *EMF* is a schema-independent deep learning model trained to classify equivalence between a pair of subexpressions. In building the *EMF*, we evaluated many candidate architectures, including various supervised classifiers, logistic regression (LR) [41], random forests (RF) [26], and multi-layer perceptrons (MLP) [19]. While the LR and RF models are simple to train and exhibit moderate performance, they suffered from one fundamental limitation: they do not allow incremental training and fine-tuning. As we detail in §6, the ability to incrementally fine-tune a model is critical to adapting to changing workloads and maximizing transferability. On the other hand, MLPs are more expensive to train but support incremental training, thus we only need to feed newly-labeled samples to fine-tune the previous model.

As a result, we utilize the MLP model for classification in the *EMF*. The overall architecture of the *EMF* is illustrated in Figure 6. It comprises two tree convolution layers and three fully connected layers. As inputs, the *EMF* accepts a pair of instance-encoded logical plans, where each node is an instance-encoded vector of size $|NV|$ (see §4.1). These plans are then transformed into their db-agnostic

counterparts (i.e., vectors of size $|NV_\alpha|$) by applying the transformation described in §4.2. Next, the *EMF* applies two tree convolutions to the db-agnostic plans. Each convolution is followed by batch normalization and parametric rectified linear unit (PReLU) activation. The two resulting 128-byte summaries of each subexpression logical plan are then concatenated and passed through three fully connected layers for classification.

EMF training and testing data. To train the *EMF*, GEqO requires a large set of labeled training data (i.e., subexpression pairs). Because our db-agnostic encoding technique enables transferability between workloads and database instances, we can initially train the *EMF* model on a high-quality *synthetic* workload that contains a wide range of positively- and negatively-labeled subexpression pairs. To generate such data, we leverage two state-of-the-art query generation tools: AMOEBA [34] and WeTune [50].

AMOEBA employs a domain-specific fuzzing technique to generate a set of base queries B_Q. It then applies a set of semantic-preserving query rewrite rules R on a given query $q_i \in B_Q$ and generates a set of queries $Q'_i = \{q'^1_i, ..., q'^n_i\}$ equivalent to q_i.

GEqO leverages AMOEBA by applying it to produce a dataset of positively-labeled pairs $W_+ = \bigcup_i \{(q_\alpha, q_\beta) \mid q_\alpha \in Q'_i \cup \{q_i\} \wedge q_\beta \in Q'_i \cup \{q_i\} \setminus q_\alpha\}$.

To ensure a wide variety of training examples, we further leverage WeTune [50], which is an optimizer rule generator that automatically generates a set of non-reducible and interesting rewrite rules (including rules missed by prominent commercial query optimizers). We apply WeTune-generated rules to rewrite the set B_Q of base queries produced by AMOEBA. We then repeat the process described above to produce a WeTune-augmented training data set W'_+.

The above process yields a diverse set of equivalent subexpression pairs. To generate a corresponding set of non-equivalent pairs, we group all subexpressions in B_Q into schema-compatible groups ($\{B^1_Q, ..., B^n_Q\}$) by applying the *SF*. Each group contains subexpressions that reference the same base tables and return the same number of columns (i.e., they are non-degenerate and would not be subsequently filtered by the *SF*). Next, we generate a set of negative examples by randomly pairing the subexpressions in each group: $W_- = \{(q_\alpha, q_\beta) \mid q_\alpha, q_\beta \in B^i_Q \wedge (q_\alpha, q_\beta) \notin W_+\}$. While this process might (with low probability) yield a false negative (i.e., by

(a) Mean error by convolution layer size with two linear layers (128 and 64).

(b) Mean error for two convolution layers (sized 512 and 128) by linear layer size.

Figure 7: Mean error by linear and convolution layer sizes.

negatively labeling a pair that is actually equivalent), model training is resilient to small amounts of noise in training data and we did not observe a decrease in performance. Nonetheless, a perfect dataset could be produced by applying the automated verifier (AV) to confirm the label of each negative pair.

Using the above, GEqO finally draws a balanced set of labeled examples from W_+ and W_- to produce a synthetic training dataset that contains a variety of syntactically dissimilar and semantically equivalent logical plans. This, along with well-known machine learning techniques such as tree convolution and dynamic pooling [39] that provide resiliency to minor perturbations, enables GEqO to generalize to plans of varying shapes and sizes.

Hyperparameter tuning. To maximize *EMF* performance, we perform a search over model structure and hyperparameters. Our search considers various network architectures (i.e., between 1–5 linear and convolution layers and sizes 32–512), activation functions, dropout, and optimizer parameters such as learning rate and decay. We evaluate on the synthetic dataset based on TPC-H as described in §7.

As we show in Figure 7, we find that increasing the number of convolution and hidden layers beyond two did not improve accuracy. Layer sizes have a modest impact on accuracy. Optimizer choice and learning rate had a negligible impact on performance.

6 SEMI-SUPERVISED FEEDBACK LOOP (*SSFL*)

When applied to a new workload or as the distribution of (non)equivalent subexpressions in a workload drifts over time, the performance of the previously-trained *EMF* model may suffer. To mitigate this, GEqO employs a *semi-supervised learning bootstrapping feedback loop (SSFL)* inspired by Zhu et al. [55]. The *SSFL* continuously monitors *EMF* performance and retrains with newly-generated training data when needed.

To accomplish this, GEqO continuously measures the confidence level of classifications made by the *EMF*. If this confidence level falls below a threshold T_h, the *SSFL* dynamically samples a new, balanced set of labeled samples from the current workload. It uses this sample to fine-tune the *EMF*. The *SSFL* iterates this process until *EMF* performance reaches a desirable confidence level. We formalize the *SSFL* confidence level as follows:

DEFINITION 6.1 (*SSFL* CONFIDENCE LEVEL). *Let W be a set of queries we wish to compute $GEqO_{SET}(W, F)$ over (q.v. Equation 1). Let P_1^p be the probability estimate that the pair $p \in W \times W$ exhibits an equivalence relationship, and P_0^p the probability that p exhibits a*

Algorithm 1: The semi-supervised feedback loop (*SSFL*).

Input: A workload W.
Output: A *EMF* model, fine-tuned if confidence is low.

1 **function** *SSFL(W)*:
2   $P_0 \leftarrow \varnothing$
3   $P_1 \leftarrow \varnothing$
4   **foreach** $(q_i, q_j) \in W \times W$ **do**
5    $\rho \leftarrow Sigmoid\big(EMF(q_i, q_j)\big)$
6    $P_1 \leftarrow P_1 \cup \{\rho\}$
7    $P_0 \leftarrow P_0 \cup \{1 - \rho\}$
8   **if** *SSFL-CL*$(W, P_0, P_1) \leq T_h$ **then**
9    $S_+ \leftarrow AV\big(VMF(SF(W \times W))\big)$
10    $S_- \leftarrow sample\big((W \times W) \setminus S_+, |S_+|\big)$
11    $EMF \leftarrow train(EMF, S_+ \cup S_-)$
12   **return** *EMF*

---

non-equivalence relationship. We compute the SSFL confidence level SSFL-CL of W as:

$$SSFL\text{-}CL(W, P_0, P_1) = \frac{\sum_{p \in W \times W} \big[\max(P_0^p, P_1^p) \geq T_h\big]}{|W \times W|}$$

Ideally, we want the semi-supervised learning feedback to only trigger a few rounds of fine-tuning before it reaches a satisfactory confidence level. In order to achieve this, we need to choose good samples with good positive and negative examples in each iteration for retraining or fine-tuning the *EMF* model. A naive sampling approach is to random sample pairs of subexpressions from the workload. However, this simple approach is like shooting in the dark and is unlikely to provide sufficient positive examples for the model to learn, since positive examples (equivalent subexpressions) are generally rare events compared to negative examples in a typical workload. The key is to make sure we find some good positive examples for training.

As described in §2.2.1, by leveraging *SF* and *VMF*, we can quickly identify a set of likely equivalent subexpression pairs from a large search space, then label the pairs by actually running the equivalence verifier. We keep all the positive and negative examples. Moreover, if more negative examples are needed for a balanced sample, we can also get random sample pairs of subexpressions from the workload. As we show in §7, this *filter-balanced sampling* mechanism can significantly improve the model quality with fewer labeled sample data compared to random sampling.

We formalize the *SSFL* algorithm in Algorithm 1. It accepts a workload W and examines each pair in the cross product (line 4). For each pair, it applies the Sigmoid function to the *EMF* output to compute the probability ρ that the pair (q_i, q_j) exhibits an equivalence relationship (line 5-7). Note that probability estimates are trivial to compute when applying the *EMF* during prediction. Next, it computes a confidence level (line 8) and, if the model is insufficiently confident, generates a sample of likely-equivalent pairs by applying the *SF* and *VMF* (line 9). It then produces a complimentary sample of size $|S_+|$ sampled randomly from non-equivalent pairs to form S_-(line 10). It finally retrains or fine-tunes the *EMF* using the full sample (line 11).

7 EXPERIMENTAL EVALUATION

We now present an experimental evaluation of GEqO. The goals of our evaluation are as follows: to (i) compare various *EMF* models to determine their effectiveness in predicting equivalence relationships, as well as assessing their ability to transfer learning across different workloads and databases (§7.1); (ii) study the performance of the *VMF* filter in terms of its ability to filter out "easy" equivalence cases (§7.2); (iii) evaluate the *SSFL* pipeline with the filter-based sampling mechanism (§7.3); (iv) examine runtimes of *VMF* and *EMF* filters on CPU- and GPU-based implementations (§7.4); and (v) evaluate the impact of GEqO on scaling state-of-the-art equivalence solvers for a large workload (§7.5).

Implementation. We implement GEqO using Python 3.10.0 and Java 18.0.2. We manipulate subexpressions, parse and generate abstract syntax trees, and perform instance-based featurization using Calcite 1.27.0. The *EMF* is implemented using PyTorch 1.12 [43] and employs the Adam optimizer [33] with a learning rate of 10^{-3} and a weight decay of 5^{-4}. We train using a dropout of 50% applied to all layers. The *VMF* is implemented using FAISS 1.7.2, where we construct a quantizer using 128-bit locality-sensitive hashes (LSH), an 128-dimension inverted index, and limit neighbor searches to a radius of $d = 1$. Finally, we set the *SSFL* confidence level to $T_h = 0.9$.

Experimental Setup. We conducted our experiments using a single machine with two CPU sockets (Intel Xeon Platinum 8272CL) each with 16 physical cores (32 with hyper-threading), 264GB of main memory, and 512GB storage device. Our GPU-based experiments are executed on a single Nvidia Tesla T4 with 16GB memory.

Workloads. We generate a set of base subexpressions on the TPC-DS and TPC-H schema using AMOEBA augmented with rules from WeTune (§5) as our workload queries. The set of TPC-DS subexpressions comprises ~34k queries, while the TPC-H dataset contains ~19k queries. Section 5 describes how we obtained our balanced, labeled data to train our initial model.

7.1 *EMF* performance

We first evaluate the performance of *EMF* model in terms of model architecture, computational cost, and ability to transfer to unseen workloads and database schema.

7.1.1 **Model type.** This experiment compares the effectiveness of three candidate *EMF* classifiers: multi-layer perceptrons (MLP), random forests (RF), and logistic regression (LR). We train the three variants on the TPC-H workload and measure performance on the TPC-DS dataset. Table 3 summarizes the results. The MLP model provides superior accuracy versus the simpler models.

Figure 8 shows confusion matrices for each model type, drilling down into how the prediction aligns with the ground truth for each model. Since the *EMF* serves as a *filter*, it should be the one that strives to simultaneously minimize the false positives (i.e., α error in the top right quadrant) and false negatives (i.e., β error in the bottom left quadrant) of the prediction. Here β error is most important—since GEqO always invokes the equivalence verifier to verify the predicated equivalence, false positives from the *EMF* model do not affect the correctness of GEqO, but represent wasted computation (i.e., by invoking the expensive automated verifier). By

Table 3: Classifier performance (train TPC-H, test TPC-DS).

Model Type	Accuracy	F1
MLP	**0.970**	**0.964**
RF	0.592	0.030
LR	0.588	0.486

Table 4: Transfer learning performance on randomly-generated schema.

Dataset Size	Precision	Recall	F1
1.2k	0.94	0.99	0.97
5.0k	0.93	0.98	0.97
11.0k	0.90	0.96	0.94
19.9k	0.93	0.97	0.95
44.9k	0.88	0.96	0.94

contrast, false negatives represent the missed equivalent queries by the *EMF* model and thus should be minimized at all costs. Clearly shown in Figure 8, MLP is by far the clear winner in simultaneously minimizing the false positives and false negatives. In particular, the false negatives for MLP is kept around 0.1%, which is orders of magnitude smaller than the other two models. Due to the superiority of the MLP architecture to detect equivalence, all subsequent experiments in this section utilize this model.

7.1.2 **Computational Cost.** We next analyze the training, prediction, and space costs of the *EMF* trained using the architecture described in §5 averaged over five runs. We train the *EMF* using ~47k subexpression pairs drawn from the TPC-H dataset. On average, a training run with 20 epochs takes approximately 40 minutes. The size of the model when serialized to disk is approximately 2.3MB, including all the learned parameters. *EMF* prediction time is 0.00319s per pair of subexpressions averaged over ~70k random TPC-DS subexpression pairs.

7.1.3 **Transfer Learning**. We now discuss the ability of the *EMF* to transfer to unseen datasets. First, note that the results shown in Table 3 and Figure 8 already illustrate this ability, where the *EMF* is trained on the TPC-H workload and tested on TPC-DS workload.

Next, we generate five additional datasets ranging from approximately 1k to 50k on a random schema using the method described in §5. We then evaluate the *EMF* using the TPC-H-trained model and report model performance in Table 4. The high performance on additional unseen datasets reinforces *EMF*'s ability to easily adapt to new, unseen workloads.

7.2 *VMF* performance

In Table 5, we study the performance of the *VMF* filter, which filters out "easy" equivalence cases before GEqO applies the *EMF*. As in §7.1, we evaluate the *VMF* by applying it to the TPC-DS workload. We observe that the *VMF* is able to substantially reduce the search space and serves as an excellent filter prior to invoking the *EMF*.

(a) **MLP**   (b) **RF**   (c) **LR**

Figure 8: Confusion matrices of three candidate *EMF* models (trained on TPC-H and tested on TPC-DS).

Table 5: *VMF* performance (train TPC-H, test TPC-DS).

Accuracy	Precision	Recall	F1
0.74	0.42	0.98	0.60

(a) **Accuracy**   (b) **F1 Score**

Figure 9: *SSFL* accuracy and F1 for filter-based and random samples. Each sampling method was used to iteratively select and train over successive batches of 512 samples.

7.3 *SSFL* performance

In this experiment, we evaluate the semi-supervised feedback loop (*SSFL*) in GEqO. To do so, we iteratively train on additional labeled samples to fine-tune the *EMF* model. We compare our filter-based sampling method (§6) against random sampling.

For this experiment, we start with a scenario where the workload changes with new equivalent and non-equivalent patterns that the model has never seen before. We expect an initial model with low quality that improves with subsequent *SSFL* iterations. To model this, we first create a degenerate TPC-H dataset by omitting all queries that contain joins. We then train an initial model on the degenerate dataset and test on the TPC-DS workload.

Figure 9 shows the accuracy and F1 score of the variants as they are exposed to additional labeled samples. Since the initial model has only been exposed to limited forms of equivalent and non-equivalent patterns, it does not perform well on the new workload which contains lots of subexpressions with joins. In each iteration

of the feedback loop, we draw 512 labeled samples, using either filter-based or random sampling, from the new workload to help improve the model.

With random sampling, performance does not improve meaningfully. Due to the non-equivalence of most subexpressions in the workload, identifying positive examples through random sampling is nearly impossible. As a result, the accuracy and F1 score remain extremely low.

By contrast, the filter-based sampling is more intelligent in selecting balanced samples that contain both positive and negative examples. This leads to significant improvements in both accuracy and F1 score. It takes only ~4k samples to improve model accuracy and F1 score to 90%.

We next measure *SSFL* execution time at various batch sizes. Figure 10 shows the result. Each bar in the figure shows the end-to-end *SSFL* runtime, including both the time for sampling and training. Obviously, the filter-based sampling is more expensive than random sampling since it needs to do extra work to identify likely equivalent subexpression pairs (e.g., by executing the *SF* and *VMF* filters and verifying). As we see in the figure, as more samples are trained over, the difference between the two reduces from 6.9× to less than 2×. At the same time, it's worth recalling that the filter-based sampling requires many fewer iterations to achieve a satisfactory model accuracy and F1 score. Additionally, the *SSFL* process may be performed out-of-band with model prediction and the improved model may be substituted after training, mitigating performance impact on the prediction path. Once a model has stabilized, the *SSFL* will no longer be active, and no further overhead will be incurred.

Finally, Figure 11 shows the breakdown of the time for the feedback loop with filter-based sampling. As can be seen, the time spent in featurization, sampling, and verification is modest and does not substantially increase with batch size. On the other hand, the increase in training time is more dramatic and it quickly dominates *SSFL* runtime.

Figure 10: *SSFL* training time for filter-based sampling versus random sampling. Each iteration trains over a batch of 512 newly-labeled samples.

Figure 11: Runtime of the operations applied by the *SSFL*.

(a) *VMF* Runtime

(b) *EMF* Runtime

Figure 12: Total runtimes of the *VMF* and *EMF* filters on varying number of subexpression pairs (log-log scale).

7.4 *VMF & EMF* compute performance

In the previous sections we evaluated the performance of the *VMF* and *EMF* in terms of their ability to identify equivalences and eliminate non-equivalences. In this section, we further examine the runtimes of each filter. To do so, we execute each filter on increasingly large subsets of the TPC-DS dataset. We reduce confounds by disabling all other filters and compare performance using a CPU- and GPU-based implementation.

In Figure 12(a), we observe that the CPU-based *VMF* exhibits excellent performance for smaller numbers of subexpression pairs, whereas the GPU-based variant surpasses it for $\geq{\sim}1$ million pairs due to a decrease in the proportion of data transfer I/O overheads in the overall runtime. In contrast, in Figure 12(b), the *EMF* consistently shows superior performance with GPU-based execution, although the CPU variant performs well at lower numbers of pairs. These results highlight the flexibility of GEqO in targeting and adapting to heterogeneous hardware, providing a distinct advantage over other heuristic- and optimizer-based techniques

7.5 End-to-End GEqO performance

We now evaluate GEqO performance in detecting equivalent subexpression pairs in various workloads. To do so, we randomly create a series of forty ~50k pair datasets generated on the TPC-DS schema and unseen by the GEqO model. We verify that each dataset contains approximately 8, 16, 32, 64, or 128 equivalent pairs. We

ensure that we have at least five (or more) datasets at each equivalence count.[3] We then execute GEqO on each dataset by executing $GEqO_{\text{SET}}(W, \{SF, VMF, EMF\})$ defined in §2.2 along with the baselines described below. For this experiment, we assume that the equivalences admitted by the AV constitute ground truth and GEqO has executed its *SSFL* and reached a confidence level above its minimum threshold ($T_h \geq 0.9$).

We compare GEqO against three baselines: (i) the SPES query equivalence solver [54]; (ii) signature-based equivalence detection based on [32], which compares signatures computed on each subexpression's abstract syntax tree (AST); and (iii) an optimizer-based equivalence detection technique that leverages the Calcite optimizer to check whether two subexpressions are equivalent.

As shown in Figure 13(a), we observe that GEqO identifies nearly all the semantic equivalences in the dataset (with a true positive rate averaging 88% across all datasets and equivalence rates), whereas the Calcite and signature-based techniques average far fewer. Unsurprisingly, SPES correctly verifies all equivalences.

Next, Figure 13(b) shows the runtimes for each method. SPES's runtime is more than 200× more expensive than the other methods. Figure 13(c) omits SPES and illustrates that the Calcite and signature-based methods have approximately constant runtimes across all datasets, whereas GEqO exhibits a curve that is similar at low numbers of equivalences and gradually rises for datasets with more equivalences. These plots demonstrate that GEqO is able to

---

[3]A small number of datasets had additional (up to 6.25%) equivalences as a byproduct of the randomized selection process.

(a) **True positive rate**

(b) **Total runtime**

(c) **Total runtime (w/o SPES)**

(d) **Time / equivalence (w/o SPES)**

Figure 13: End-to-end GEqO performance vs. the Calcite query optimizer, signature-based detection, and SPES.

Figure 14: Total runtime under various filter combinations.

Figure 15: Result Caching Performance.

detect equivalences at an accuracy level near that of SPES but at a runtime similar to the heuristic-based techniques.

Finally, we observe that while Figure 13(c) suggests a gradually rising runtime for GEqO, this occurs because *it detects more equivalences*. Figure 13(d) plots the runtime per equivalence detected and demonstrates that GEqO spends approximately the same amount of time as Calcite and the signature-based method per equivalence. For scaling reasons, we do not show SPES values in this plot, which ranged from 13.8 to 118.2 seconds per identified equivalence.

7.6 GEqO Filters Ablation Study

Next we explore the relative contribution of each GEqO filter. We execute $GEqO_{\text{SET}}(W, F)$ (see §2.2), by varying F to be some combination of the filters available in GEqO (i.e., the nonempty power set of $\{VMF, EMF, SF\}$) and W to be each of the 32-equivalence

datasets described in §7.5. We report mean runtime over evaluated workloads, including verification time of the filtered pairs.

Figure 14 shows the result of this experiment. We observe that GEqO achieves best performance only when applying *all* filters; no other combination minimizes the total runtime. This implies that GEqO's filters are complimentary to each other, and not redundantly filtering the same sets of subexpression pairs.

7.7 A Case Study on Result Caching

In our final experiment, we evaluate how GEqO can be utilized in a result caching or materialization application, where results of queries are cached under a storage budget, to save computation for future semantically-equivalent queries. Using the workload from §7 on the 100GB TPC-DS dataset, we obtain approximately

~23k unique expressions after excluding those that produce empty results. Our experiment assumes no updates.

When executing with unlimited storage budget, the result cache using GEqO could materialize the first occurrence of each equivalent expression (there are 5,277 equivalence classes in the workload), resulting in a total of ~2GB storage space (in this workload, the expressions are computationally expensive but return small results), which we use as the *upper-bound* for our storage budget. We then vary the storage budget for the cache, and simulate a caching policy that materializes the most expensive queries (leveraging past runtime statistics). Figure 15 shows a reduction of up to 61.5% in the total workload execution time (running on a modern commercial database system), with 10% of storage budget. With 100% storage budget, a total of 96.2% computation reduction is achieved.

8 RELATED WORK

Materialized Views and Query Rewriting. As one of the most widely used approaches for computation reuse, materialized views are supported in many analytics engines. However, most systems—even including some modern cloud-based analytics engines like Snowflake [12], BigQuery [20], and NAPA [2]—still require manual identification of common computation and creation of views. To automate view materialization, many view selection algorithms have been proposed [3, 5, 32, 51] to choose views that maximize computation reuse for a workload.

Efficient and effective detection of overlapping computation is crucial in optimally selecting views to materialize. Some classical view selection methods heavily depend on the query optimizer to identify equivalences. They consider factors such as resource constraints when selecting which views to materialize. CloudViews [30] employs Merkle tree-like signatures to quickly detect equivalent subexpressions. The ML-based view selection algorithm in [51] utilizes a SQL equivalence verifier, called EQUITAS [53], to detect equivalent subexpressions (we discuss verifiers in detail below). In terms of efficiency and scalability, the signature-based approach is clearly the best; however, it is the least effective since it only admits syntactic equivalence. By contrast, equivalence verifiers are superior in detecting semantic equivalence, but are computationally expensive. The optimizer's ability to detect semantic equivalence is bound by its rewrite rules. Interestingly, the work in [50] found that even a mature optimizer like the one in SQL Server could still miss some rewrite rules. In addition, repeatedly invoking the optimizer to check equivalence at cloud scale could easily turn the optimizer into a bottleneck. Compared to all these existing approaches, GEqO is designed to be efficient and scalable, achieving effectiveness close to that of a verifier.

To utilize materialized views, view matching algorithms (e.g., [18]) match a query against previously-materialized views to determine whether the query can be rewritten into an improved variant by leveraging the views at runtime. In fact, query rewrite is generally an important query optimization step applied in many settings inside or outside the optimizer [21, 22]. Most optimizers continue to rely on rewrite rules to identify equivalence and transform the original query into a semantically equivalent alternative. GEqO can be used to learn equivalence relationships present in the given workload and complement these existing rewrite rules.

Query Equivalence Verification. Verification of SQL query equivalence has been a long-standing topic of research in database theory [1]. Several practical verifiers have been proposed [9, 10, 47, 53, 54]. Cosette [10] and its extended version UDP [9] transform SQL queries into algebraic expressions and then utilize the Coq proof assistant [44] to compare the two resultant algebraic expressions. However, these two approaches are computationally expensive due to the large number of normalized algebraic representations. Recently, EQUITAS [53] and its extension SPES [54] address this limitation by efficiently deriving symbolic representation of SQL queries and use satisfiability modulo theories (SMT) to determine their equivalence under set and bag semantics. Approaches such as Peggy [47] leverage equality saturation, where the optimizer enumerates equivalent expressions for a given input expression based on predefined rules and collects them in a compact graph representation. While saturating every subexpression in a workload is not scalable, this technique could be leveraged by GEqO as an alternative equivalence verifier.

Since GEqO is a general framework, it can plug in any of the above equivalence verifiers, or even new customized ones, to verify the predication from the *EMF* filter.

9 CONCLUSION AND FUTURE WORK

In this paper, we presented GEqO, a portable lightweight ML-based framework for efficiently identifying semantically equivalent subexpressions at scale. We introduced *VMF* and *EMF* filters to fill in the gap between a simple but very coarse schema-based filter and the accurate but very expensive equivalence verifier. We trained a deep-learning-based model to efficiently predict equivalence relationship between a pair of subexpressions. The db-agnostic featurization allows the learning from one workload and database to be transferrable to another. We also introduced an end-to-end semi-supervised learning feedback loop with clever sampling to circumvent the expensive data labeling process. Our experimental evaluation demonstrates that GEqO is up to 200× faster than verifiers on TPC-DS subexpressions.

9.1 Extension to Complex Subexpressions

Even though a query workload might contain non-SPJ queries, GEqO can still detect the equivalence of the SPJ subexpressions in the workload. Therefore, it represents a significant step forward in building an end-to-end framework for efficiently identifying semantically equivalent computations at scale. Nevertheless, We plan to extend GEqO to support complex subexpressions beyond SPJ (e.g., unions, aggregation, and complex predicates), using a similar approach to [25]. We next briefly sketch potential encoding extensions.

OR and IN operators. We convert the WHERE clause with the OR operator to DNF, considering each conjunctive as a separate query and introduce unions. Each conjunctive branch is encoded as described in Section 3. However, this approach encounters scalability issues due to the exponential growth in the number of clauses and the redundant encoding across union branches. The *IN* clause can be considered as a shorthand representation for multiple *OR* conditions. Unnesting the clause and introducing *OR* conditions still pose scalability issues, which we intend to investigate.

Union and except operators. We add a one-hot vector indicating union or except operators.

Group By and aggregation operators. We add a new group-by segment that contains a one-hot vector for each of the group-by columns and an aggregate segment [AGG,COL], which is a concatenation of AGG and COL one-hot vectors of supported aggregation functions and columns.

In addition to addressing encoding scalability issues with certain operators, we plan to assess the effectiveness of the current *EMF* model on complex queries and determine if any enhancement to the current architecture (e.g., augmenting the number of convolution layers) is necessary.

9.2   Extension to Query Containment

A second important future direction involves using GEqO to scalably detect semantic containment, which is crucial for some view selection algorithms [4, 8].

We think the GEqO framework should be applicable to semantic containment. The *EMF* model can be directly extended to classify containment. We conduct a preliminary experiment to demonstrate this by training a new containment model over TPC-H subexpressions with one-way joins and up to three predicates. This model achieved ∼98% accuracy on a test TPC-DS workload of similar complexity. As we increased the complexity of the workload (e.g., with additional joins), the accuracy dropped to ∼78%. We believe these results are promising since detecting containment is strictly harder than equivalence.

In the prediction pipeline, the *SF* filter is adaptable to support containment. For instance, for a given pair, the table set of one of the subexpressions should be a subset of the other subexpression's table set, same condition applies on the projected columns. However, the distance metric used in the *VMF* filter is not as easily adpatable, and we leave this as future work. In terms of automated verification of semantic containment, this problem is well-studied under set semantics [1], but far less understood under bag semantics (e.g., the class of unions of conjunctive queries is undecidable under bag semantics [28, 29]). We direct readers to the survey [24], which describes several practical containment checking algorithms in the context of rewriting queries using views, among which the algorithm in [18] is a popular one that has been adopted by SQL Server and Calcite [6] optimizers.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Vol. 8. Addison-Wesley Reading.

[2] Agiwal, Ankur and Lai, Kevin and Manoharan, Gokul Nath Babu and Roy, Indrajit and Sankaranarayanan, Jagan and Zhang, Hao and Zou, Tao and Chen, Min and Chen, Jim and Dai, Ming and others. 2021. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. 14, 12 (2021).

[3] 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.

[4] Rafi Ahmed, Randall Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated generation of materialized views in oracle. In *VLDB*, Vol. 13. 3046–3058.

[5] Amazon. 2023. Amazon Redshift: Automated materialized views. https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-auto-mv.html. Accessed: 2023.

[6] Apache Software Foundation. 2023. Apache Calcite: The foundation for your next high-performance database. https://calcite.apache.org.

[7] Apache Software Foundation. 2023. Apache Spark: Unified Engine for large-scale data analytics. https://spark.apache.org.

[8] Rada Chirkova and Jun Yang. 2012. Materialized views. *Foundations and Trends® in Databases* 4, 4 (2012), 295–405.

[9] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. In *VLDB*, Vol. 11. 1482–1495.

[10] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: Proving query rewrites with univalent SQL semantics. In *SIGPLAN*, Vol. 52. 510–524.

[11] Sara Cohen. 2006. Equivalence of queries combining set and bag-set semantics. In *PODS*. 70–79.

[12] Dageville, Benoit and Cruanes, Thierry and Zukowski, Marcin and Antonov, Vadim and Avanes, Artin and Bock, Jon and Claybaugh, Jonathan and Engovatov, Daniel and Hentschel, Martin and Huang, Jiansheng and others. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS*. 337–340.

[14] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *CADE-25*. 378–388.

[15] Alin Deutsch. 2018. FOL Modeling of Integrity Constraints (Dependencies). In *Encyclopedia of Database Systems, Second Edition*.

[16] Yuyang Dong, Chuan Xiao, Takuma Nozawa, Masafumi Enomoto, and Masafumi Oyamada. 2023. DeepJoin: Joinable Table Discovery with Pre-trained Language Models. In *VLDB*, Vol. 16. 2458–2470.

[17] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A metaprogramming framework for formal verification. In *ICFP*, Vol. 1. 1–29.

[18] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. In *SIGMOD*, Vol. 30. 331–342.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[20] Google. 2023. BigQuery. https://cloud.google.com/bigquery.

[21] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18 (1995), 19–29.

[22] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. 209–218.

[23] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD*. 1917–1923.

[24] Alon Y Halevy. 2001. Answering queries using views: A survey. *The VLDB Journal* 10, 4 (2001), 270–294.

[25] Rojeh Hayek and Oded Shmueli. 2020. Improved Cardinality Estimation by Learning Queries Containment Rates. In *EDBT*. 157–168.

[26] Tin Kam Ho. 1995. Random decision forests. In *ICDAR*, Vol. 1. 278–282.

[27] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. 2015. Query-aware locality-sensitive hashing for approximate nearest neighbor search. In *VLDB*, Vol. 9. 1–12.

[28] Yannis E Ioannidis and Raghu Ramakrishnan. 1995. Containment of conjunctive queries: Beyond relations as sets. *TODS* 20, 3 (1995), 288–324.

[29] TS Jayram, Phokion G Kolaitis, and Erik Vee. 2006. The containment problem for real conjunctive queries with inequalities. In *PODS*. 80–89.

[30] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. In *VLDB*, Vol. 11. 800–812.

[31] Alekh Jindal, Shi Qiao, Hiren Patel, Abhishek Roy, Jyoti Leeka, and Brandon Haynes. 2021. Production Experiences from Computation Reuse at Microsoft. In *EDBT*. 623–634.

[32] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation reuse in analytics job service at Microsoft. In *ICDM*. 191–203.

[33] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

[34] Xinyu Liu, Qi Zhou, Joy Arulraj, and Alessandro Orso. 2022. Automatic Detection of Performance Bugs in Database Systems using Equivalent Queries. In *ICSE*. 225–236.

[35] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. In *PAMI*, Vol. 42. 824–836.

[36] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *SIGMOD*. 1275–1288.

[37] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. In *VLDB*, Vol. 12. 1705–1718.

[38] Microsoft. 2023. Azure Synapse Analytics. https://azure.microsoft.com/en-us/services/synapse-analytics.

[39] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, Vol. 30.

[40] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *ICMD*. 2557–2569.

[41] John Neter, Michael H Kutner, Christopher J Nachtsheim, and William Wasserman. 1996. Applied linear statistical models. (1996).

[42] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. 2019. An Empirical Analysis of Deep Learning for Cardinality Estimation. *CoRR* abs/1905.06425 (2019).

[43] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS* 32 (2019).

[44] Christine Paulin-Mohring. 2011. Introduction to the Coq proof-assistant for practical software verification. In *LASER*. 45–95.

[45] Jianbin Qin, Wei Wang, Chuan Xiao, Ying Zhang, and Yaoshu Wang. 2021. High-Dimensional Similarity Query Processing for Data Science. In *SIGKDD*. 4062–4063.

[46] Timos K. Sellis. 1988. Multiple-Query Optimization. In *TODS*, Vol. 13. 23–52.

[47] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *POPL*. 264–276.

[48] Yicheng Tu, Mehrad Eslami, Zichen Xu, and Hadi Charkhgard. 2022. Multi-Query Optimization Revisited: A Full-Query Algebraic Method. In *Big Data*. 252–261.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*. 5998–6008.

[50] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *SIGMOD*. 94–107.

[51] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic view generation with deep learning and reinforcement learning. In *ICDE*. 1501–1512.

[52] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: Parallel Databases Meet MapReduce. In *VLDB*, Vol. 21. 611–636.

[53] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Dong Xu. 2019. Automated verification of query equivalence using satisfiability modulo theories. *VLDB* 12, 11, 1276–1288.

[54] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In *ICDE*. 2735–2748.

[55] Xiaojin Zhu and Andrew B Goldberg. 2009. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning* 3, 1 (2009), 1–130.

[56] Yiwen Zhu, Subru Krishnan, Konstantinos Karanasos, Isha Tarte, Conor Power, Abhishek Modi, Manoj Kumar, Deli Zhang, Kartheek Muthyala, Nick Jurgens, et al. 2021. KEA: Tuning an Exabyte-Scale Data Infrastructure. In *SIGMOD*. 2667–2680.