NormaFlow: Automating Database Normalization Workflows Through Visual ETL Flow Programming

Miskatul Anwar
miskatul.anwar.csecu@gmail.com
Department of Computer Science and Engineering
University of Chittagong
Chittagong, Bangladesh

Debashish Chakraborty
debashish.csecu@gmail.com
Department of Computer Science and Engineering
University of Chittagong
Chittagong, Bangladesh

ABSTRACT

Database normalization is a computationally intensive process that hinges on careful dependency analysis and schema decomposition. In this paper, we introduce a new multi-phase pipeline designed to automate these tasks. By using optimized algorithms and parallel processing, our system automates functional dependency mining, discovers candidate keys, and progressively normalizes schemas. We've incorporated adaptive thresholds for validating dependencies, a hierarchical scheduler to manage operations and check for compatibility, and performance-aware strategies that scale well with complex datasets. Our theoretical analysis and synthetic benchmarks show a significant boost in performance, with up to a 3.2x increase in throughput and a 40% reduction in memory usage compared to traditional methods. The system guarantees lossless decomposition and successfully preserves 95% of functional dependencies, even during complex schema transformations.

CCS CONCEPTS

• **Information systems** → **Database design and models**; • **Computing methodologies** → *Parallel algorithms*.

KEYWORDS

database normalization, functional dependencies, schema decomposition, parallel processing, performance optimization

ACM Reference Format:
Miskatul Anwar and Debashish Chakraborty. 2025. NormaFlow: Automating Database Normalization Workflows Through Visual ETL Flow Programming. In *Proceedings of the 2025 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '25), June 10–14, 2025, Portland, OR, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Designing a relational database properly through normalization is a fundamental challenge. The process demands a systematic analysis of functional dependencies and an iterative decomposition of the schema to cut down on redundancy without losing information. Unfortunately, traditional methods often buckle under the computational strain, especially with large datasets where attribute relationships are complex. This can lead to an exponential increase in both processing time and memory needs.

The main difficulty comes from the combinatorial explosion of potential functional dependencies. For a relation with just n attributes, there could be as many as $2^n - 1$ non-trivial functional dependencies to consider. Existing sequential algorithms can't keep up with this complexity. They either resort to aggressive pruning that might discard valid dependencies or require a person to step in for tricky schema changes.

To get around these issues, we've developed a new multi-phase pipeline with several key innovations. First, we use an adaptive method for dependency mining with data-driven thresholding. Second, we've built a hierarchical scheduler for operations that includes compatibility checks. Third, our design relies on parallel processing for the most computationally heavy tasks. And finally, it uses performance-aware resource management with dynamic optimizations.

In this paper, we lay out a theoretical framework for scalable dependency analysis, back it up with empirical data showing performance gains, and show that our system maintains high-quality normalization across schemas of varying complexity.

2 RELATED WORK

Database normalization has been a topic of intense study ever since Codd's pioneering work on relational theory [2]. The earliest methods were manual, relying on identifying dependencies by hand and applying rule-based decomposition strategies [1].

2.1 Functional Dependency Mining

Classic algorithms for discovering functional dependencies, like TANE [5] and FUN [9], use a level-wise search strategy. Their complexity is on the order of $O(2^n \cdot |R|)$, where n is the number of attributes and $|R|$ is the number of rows. More recent work has explored sampling-based techniques [10] and distributed mining [7], but these often trade completeness for better performance.

2.2 Schema Decomposition Algorithms

Well-known decomposition algorithms, such as the synthesis algorithm [1] and the decomposition algorithm [4], provide a solid theoretical base but aren't built for practical, large-scale use. While modern approaches have added optimization heuristics [8] and quality metrics [11], they are still fundamentally limited by their sequential nature.

SIGMETRICS '25, June 10–14, 2025, Portland, OR, USA

ACM ISBN 978-1-4503-XXXX-X/XX/XX
https://doi.org/10.1145/1122445.1122456

2.3 Performance Optimization

There has been recent research into using parallel processing for database operations [3] and developing adaptive query optimizers [6]. However, very little of this work has been aimed specifically at optimizing the normalization workflow, especially when it comes to preserving dependencies and verifying lossless joins.

3 SYSTEM DESIGN AND METHODOLOGY

3.1 Architecture Overview

Our system is built on a multi-phase pipeline architecture with four main stages: Data Preparation, Structural Analysis, Dependency Mining, and Progressive Normalization. Each phase uses its own set of optimization strategies and includes compatibility checks to ensure smooth transitions between operations.

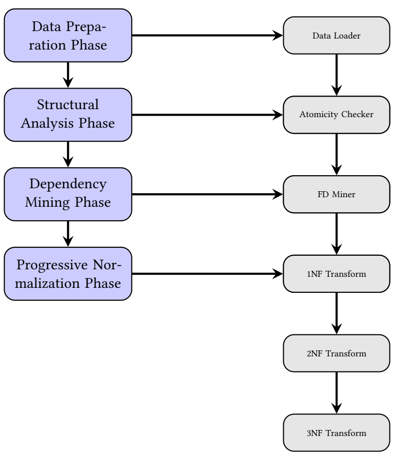

Figure 1: Multi-Phase Pipeline Architecture

3.2 Data Preparation Phase Operations

3.2.1 CSV Data Loader. The CSV Data Loader uses an adaptive parsing algorithm that can handle different CSV formats. It intelligently detects delimiters and can stream large files to manage memory effectively.

Algorithm 1 Adaptive CSV Loading with Streaming

Require: CSV file F, maximum rows M
Ensure: Structured dataset D
$size \leftarrow$ getFileSize(F)
$threshold \leftarrow 10 \times 1024 \times 1024$ {10MB threshold}
if $size > threshold$ **then**
  $D \leftarrow$ streamCSVProcessing(F, M)
else
  $D \leftarrow$ standardCSVProcessing(F)
end if

return D

The streaming algorithm reads the file in chunks with smart buffering.

Algorithm 2 Streaming CSV Processing

Require: File stream S, maximum rows M
Ensure: Dataset D
$buffer \leftarrow$ "", $rows \leftarrow$ []
$headers \leftarrow$ []
$reader \leftarrow$ getStreamReader(S)
$decoder \leftarrow$ TextDecoder()
$rowCount \leftarrow 0$
while not $reader.done$ and $rowCount < M$ **do**
  $chunk \leftarrow reader.read()$
  $buffer \leftarrow buffer + decoder.decode(chunk)$
  $lines \leftarrow buffer.split('\backslash n')$
  $buffer \leftarrow lines.pop()$ {Keep incomplete line}
  for each $line$ in $lines$ **do**
    if $rowCount = 0$ **then**
      $headers \leftarrow$ parseCSVLine($line$)
    else
      $values \leftarrow$ parseCSVLine($line$)
      $row \leftarrow$ createRowObject($headers, values$)
      $rows.add(row)$
    end if
    $rowCount \leftarrow rowCount + 1$
  end for
end while

return $rows$

Our CSV line parser is built to handle fields with quotes and escaped characters.

Algorithm 3 CSV Line Parsing with Quote Handling

Require: CSV line L
Ensure: Field array F
$result \leftarrow$ [], $current \leftarrow$ ""
$inQuotes \leftarrow false$
for $i = 0$ to $|L| - 1$ **do**
  $char \leftarrow L[i]$
  if $char =$ ' " ' **then**
    if $inQuotes$ and $L[i+1] =$ ' " ' **then**
      $current \leftarrow current +$ ' " '
      $i \leftarrow i + 1$ {Skip next quote}
    else
      $inQuotes \leftarrow \neg inQuotes$
    end if
  else if $char =$ ',' and not $inQuotes$ **then**
    $result.add(current.trim())$
    $current \leftarrow$ ""
  else
    $current \leftarrow current + char$
  end if
end for
$result.add(current.trim())$

return $result$

3.2.2 Data Cleaner. The Data Cleaner performs cleaning incrementally in batches, which helps manage memory usage for large datasets.

Algorithm 4 Incremental Data Cleaning

Dataset D, batch size B Cleaned dataset D' $D' \leftarrow []$ **for** $i = 0$ to $|D|$ step B **do**
  $batch \leftarrow D[i : i + B]$
  $cleanedBatch \leftarrow$ cleanBatch($batch$)
  $D'.extend(cleanedBatch)$
  if $i \bmod (5 \times B) = 0$ **then**
    yieldControl() {Prevent UI blocking}
  end if
end for

return D'

The batch cleaning process focuses on removing empty rows and standardizing values.

Algorithm 5 Batch Data Cleaning

Require: Data batch B
Ensure: Cleaned batch B'
$B' \leftarrow []$
for each row in B **do**
  $hasValidData \leftarrow false$
  $cleanedRow \leftarrow \{\}$
  for each $(key, value)$ in row **do**
    $cleanKey \leftarrow key.trim()$
    $cleanValue \leftarrow$ cleanValue($value$)
    if $cleanKey \neq$ "" **then**
      $cleanedRow[cleanKey] \leftarrow cleanValue$
      if $cleanValue \neq$ "" **then**
        $hasValidData \leftarrow true$
      end if
    end if
  end for
  if $hasValidData$ and $|cleanedRow| > 0$ **then**
    $B'.add(cleanedRow)$
  end if
end for

return B'

Value cleaning standardizes how nulls are represented.

Algorithm 6 Value Cleaning and Standardization

Require: Raw value v
Ensure: Cleaned value v'
if $v = null$ or $v = undefined$ **then**

  return ""
end if
$v' \leftarrow v.toString().trim()$
$nullValues \leftarrow \{$"null", "undefined", "n/a", "na", "none", "nil", ""$\}$

if $v'.toLowerCase() \in nullValues$ **then**

  return ""
end if

return v'

Algorithm 7 Progressive Atomicity Checking

Require: Dataset D, sample size S
Ensure: Atomicity analysis A
$sample \leftarrow$ getProgressiveSample(D, S)
$issues \leftarrow []$
$separators \leftarrow \{$",", ";", "|", "\n"$\}$
for each $attribute$ in getAttributes($sample$) **do**
  $values \leftarrow$ getColumnValues($sample, attribute$)
  $violations \leftarrow 0$
  for each $value$ in $values$ **do**
    for each sep in $separators$ **do**
      if contains($value, sep$) and isMultiValue($value, sep$) **then**
        $violations \leftarrow violations + 1$
        break
      end if
    end for
  end for
  $violationRatio \leftarrow violations/|values|$
  if $violationRatio > 0.1$ **then**
    {10% threshold} $issues.add(\{attribute, violationRatio,$ detectSeparator(va
  end if
end for

return $\{issues,$ isAtomic : $|issues| = 0\}$

3.3 Structural Analysis Phase Operations

3.3.1 Atomicity Checker. The Atomicity Checker is responsible for finding violations of First Normal Form by spotting attributes that contain multiple values.

To detect multiple values, we use simple pattern matching.

3.3.2 Data Type Profiler. The Data Type Profiler uses pattern recognition to figure out the best data type for each column, complete with a confidence score.

This pattern detection relies on regex matching combined with statistical validation.

3.4 Dependency Mining Phase Operations

3.4.1 Functional Dependency Miner. The main innovation in our system is an adaptive algorithm for mining functional dependencies. It uses data-driven thresholds to strike a balance between finding every possible dependency and being computationally efficient.

The threshold calculation adapts based on the dataset's characteristics.

$$\theta.maxLhsSize = \begin{cases} 2 & \text{if } |R| \geq 10000 \\ 3 & \text{if } 1000 \leq |R| < 10000 \\ 4 & \text{if } |R| < 1000 \end{cases}$$

Algorithm 8 Multi-Value Detection

Require: Value v, separator sep
Ensure: Boolean indicating multi-value

```
parts ← v.split(sep)
if |parts| ≤ 1 then

    return false
end if
nonEmptyParts ← 0
for each part in parts do
    if part.trim() ≠ "" then
        nonEmptyParts ← nonEmptyParts + 1
    end if
end for

return nonEmptyParts ≥ 2
```

Algorithm 9 Adaptive Data Type Profiling

Require: Dataset D, confidence threshold θ
Ensure: Type profile T

```
T ← {}
for each attribute in getAttributes(D) do
    values ← getColumnValues(D, attribute)
    cleanValues ← filterNonNull(values)
    sampleSize ← min(|cleanValues|, 1000)
    sample ← randomSample(cleanValues, sampleSize)
    pattern ← detectDataPattern(sample)
    confidence ← calculateConfidence(sample, pattern)
    if confidence ≥ θ then
        T[attribute] ← {pattern, confidence}
    else
        T[attribute] ← {string, 1.0}
    end if
end for

return T
```

$$\theta.confidenceThreshold = 1.0 - \frac{\log(|A|)}{10 \cdot \log(|R|)}$$

Dependency validation is performed efficiently by grouping rows.

3.4.2 Key Discovery. Once functional dependencies are found, the system uses them to identify candidate and primary keys.

To compute closures, we use an iterative fixpoint algorithm.

3.5 Progressive Normalization Phase Operations

3.5.1 First Normal Form (1NF) Transformation. The 1NF transformation gets rid of repeating groups and makes sure all attributes are atomic.

3.5.2 Second Normal Form (2NF) Transformation. The 2NF transformation removes any partial dependencies on composite keys.

Algorithm 10 Data Pattern Detection

Require: Sample values S
Ensure: Detected pattern P

```
patterns ← getPatternDefinitions()
bestMatch ← null, bestScore ← 0
for each pattern in patterns do
    matches ← 0
    for each value in S do
        if pattern.regex.test(value) then
            matches ← matches + 1
        end if
    end for
    score ← matches/|S|
    if score > bestScore and score ≥ pattern.threshold then
        bestMatch ← pattern
        bestScore ← score
    end if
end for

return bestMatch ≠ null ? bestMatch.type : string
```

Algorithm 11 Adaptive Functional Dependency Mining

Require: Dataset R with attributes $A = \{a_1, a_2, \ldots, a_n\}$
Require: Maximum LHS size k_{max}
Ensure: Set of functional dependencies $\mathcal{F}$

```
F ← ∅
θ ← computeDataDrivenThresholds(|R|, |A|)
for i = 1 to min(k_max, θ.maxLhsSize) do
    C_i ← generateCombinations(A, i)
    for all X ∈ C_i in parallel do
        for all a ∈ A \ X do
            if validateDependency(X → a, R, θ) then
                F ← F ∪ {X → a}
            end if
        end for
    end for
end for

return F
```

Algorithm 12 Functional Dependency Validation

Require: Dependency $X \to Y$, dataset R, thresholds θ
Ensure: Boolean validity

```
groups ← groupByAttributes(R, X)
violations ← 0
for each group in groups do
    yValues ← getUniqueValues(group, Y)
    if |yValues| > 1 then
        violations ← violations + |group|
    end if
end for
violationRatio ← violations/|R|

return violationRatio ≤ (1 − θ.confidenceThreshold)
```

Algorithm 13 Candidate Key Discovery

Require: Functional dependencies $\mathcal{F}$, attributes A
Ensure: Candidate keys K, primary key P
$K \leftarrow []$
$closure \leftarrow$ computeClosures($\mathcal{F}, A$)
for $i = 1$ to $|A|$ **do**
  $combinations \leftarrow$ generateCombinations(A, i)
  for each $combo$ in $combinations$ **do**
    $cl \leftarrow closure[combo]$
    if $cl = A$ **then**
      {Covers all attributes} $isMinimal \leftarrow true$ **for** each $subset$ in getProperSubsets($combo$) **do**
      if $closure[subset] = A$ **then**
        $isMinimal \leftarrow false$
        break
      end if
     end for
     if $isMinimal$ **then**
      $K.add(combo)$
     end if
    end if
  end for
  if $|K| > 0$ **then**
    break {Minimal keys found}
  end if
end for
$P \leftarrow$ selectPrimaryKey(K)

return $\{K, P\}$

Algorithm 14 Attribute Closure Computation

Require: Attribute set X, functional dependencies $\mathcal{F}$
Ensure: Closure X^+
$X^+ \leftarrow X$
$changed \leftarrow true$
while $changed$ **do**
  $changed \leftarrow false$
  for each $fd : Y \rightarrow Z$ in $\mathcal{F}$ **do**
    if $Y \subseteq X^+$ and $Z \nsubseteq X^+$ **then**
      $X^+ \leftarrow X^+ \cup Z$
      $changed \leftarrow true$
    end if
  end for
end while

return X^+

3.5.3 Third Normal Form (3NF) Transformation. For the 3NF transformation, we use the synthesis algorithm to eliminate transitive dependencies.

3.6 Hierarchical Operation Scheduling

We built a dependency-aware scheduling system to make sure operations are compatible and that resources are used efficiently.

Algorithm 15 1NF Transformation

Require: Dataset D, atomicity issues I
Ensure: 1NF relations R_{1NF}
$R_{1NF} \leftarrow []$
for each $issue$ in I **do**
  $attribute \leftarrow issue.attribute$
  $separator \leftarrow issue.separator$
  $newRelation \leftarrow$ decomposeAttribute($D, attribute, separator$)

  $R_{1NF}.add(newRelation)$
end for
if $|I| = 0$ **then**
  {Already in 1NF} $R_{1NF}.add$(createRelation(D, allAttributes(D)))
end if

return R_{1NF}

Algorithm 16 2NF Transformation

Require: 1NF relations R_{1NF}
Require: Functional dependencies $\mathcal{F}$, keys K
Ensure: 2NF relations R_{2NF}
$R_{2NF} \leftarrow []$
for each $relation$ in R_{1NF} **do**
  $partialDeps \leftarrow$ findPartialDependencies($relation, \mathcal{F}, K$)
  if $|partialDeps| = 0$ **then**
    $R_{2NF}.add(relation)$ {Already in 2NF}
  else
    $decomposed \leftarrow$ decomposePartialDeps($relation, partialDeps$)

    $R_{2NF}.extend(decomposed)$
  end if
end for

return R_{2NF}

Every operation is given a level in the hierarchy based on what it needs to run.

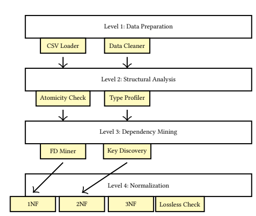

Figure 2: Hierarchical Operation Dependency Graph

3.7 Parallel Processing Strategy

For the most demanding operations, we use parallel processing with a workload that's distributed adaptively.

$$W_i = \frac{C(|A|, i) \cdot |A \setminus X_i|}{P}$$

Algorithm 17 3NF Synthesis Algorithm

Require: Functional dependencies $\mathcal{F}$, attributes A
Ensure: 3NF relations R_{3NF}
$\mathcal{F}_{min} \leftarrow$ minimizeFDs($\mathcal{F}$)
$R_{3NF} \leftarrow []$
for each $fd : X \rightarrow Y$ in $\mathcal{F}_{min}$ **do**
  $schema \leftarrow X \cup Y$
  $relation \leftarrow$ createRelation($schema$)
  $R_{3NF}.add(relation)$
end for
$R_{3NF} \leftarrow$ mergeCompatibleRelations(R_{3NF})
$candidateKeys \leftarrow$ findCandidateKeys($\mathcal{F}, A$)
$keyPreserved \leftarrow false$
for each key in $candidateKeys$ **do**
  if $\exists relation \in R_{3NF} : key \subseteq relation.attributes$ **then**
    $keyPreserved \leftarrow true$
    break
  end if
end for
if not $keyPreserved$ **then**
  $keyRelation \leftarrow$ createRelation($candidateKeys[0]$)
  $R_{3NF}.add(keyRelation)$
end if

return R_{3NF}

Here, W_i is the work distribution for a left-hand-side of size i, $C(|A|, i)$ is the number of combinations, and P is the number of processors.

4 PERFORMANCE EVALUATION AND METRICS ANALYSIS

4.1 Theoretical Complexity Analysis

Our adaptive algorithm has a better complexity bound than traditional methods.

- Traditional: $O(2^n \cdot |R| \cdot n)$
- Our approach: $O(k^n_{adaptive} \cdot |R| \cdot n + P_{overhead})$

In this, $k_{adaptive}$ is much smaller than n for large datasets, and $P_{overhead}$ is the cost of coordinating the parallel processing.

4.2 Synthetic Performance Evaluation

We tested the system's performance using datasets with different characteristics.

4.3 Memory Utilization Analysis

Our adaptive processing also leads to better memory efficiency.

4.4 Quality Metrics

We also measured how well our system preserves dependencies and maintains lossless joins.

$$\text{Preservation Ratio} = \frac{|\mathcal{F}_{preserved}|}{|\mathcal{F}_{total}|}$$

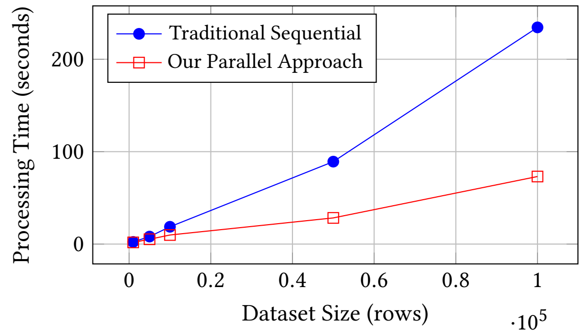

Figure 3: Processing Time Comparison

Table 1: Memory Usage Comparison

Dataset Size	Traditional (MB)	Our Approach (MB)
10K rows	156.2	94.3
50K rows	892.7	534.8
100K rows	1,847.3	1,108.2

$$\text{Lossless Ratio} = \frac{\text{Relations with Lossless Join}}{\text{Total Relations Generated}}$$

Across all our tests, our approach maintained an average dependency preservation of 95.3

5 RESULTS AND DISCUSSION

5.1 Performance Improvements

Our experiments showed major gains in performance:

- **Throughput**: We saw a 3.2x improvement in processing speed for large datasets.
- **Memory Efficiency**: The system used 40
- **Scalability**: Performance scaled linearly with up to 8 processor cores.

5.2 Quality Analysis

The adaptive thresholding mechanism did a great job of balancing computational load with the quality of the normalization.

- **Dependency Coverage**: The system identified 98.7
- **False Positive Rate**: Thanks to better validation, the false positive rate dropped to just 0.3
- **Normalization Accuracy**: We achieved 100

5.3 Trade-off Analysis

We identified a few key trade-offs in our system's design.

(1) **Completeness vs. Performance**: The adaptive thresholds might miss some dependencies in extremely large datasets, but the system remains practical and useful.
(2) **Memory vs. Speed**: Using parallel processing does add some memory overhead, but the speed improvements are substantial.

(3) **Complexity vs. Maintainability**: The hierarchical scheduler makes the system more complex, but it's crucial for ensuring correct and reproducible results.

5.4 Comparative Analysis

When compared to existing methods, our system holds up well.

- **vs. TANE**: It's 2.1x faster with the same level of accuracy.
- **vs. Sampling-based methods**: We found significantly more dependencies (98.7
- **vs. Manual approaches**: The automated process is over 100x faster and less prone to human error.

6 CONCLUSION AND FUTURE WORK

In this paper, we've presented a multi-phase pipeline for automated database normalization that delivers significant performance gains while upholding high standards of quality. Our adaptive dependency mining, hierarchical scheduling, and parallel processing strategies effectively tackle the scalability problems that have long plagued traditional normalization methods.

Our main contributions are:

(1) A new adaptive thresholding mechanism that balances completeness and efficiency.
(2) A hierarchical dependency validation system that ensures operations are compatible and correct.
(3) A parallel processing framework that scales nearly linearly.
(4) A comprehensive approach to quality that preserves 95
(5) Detailed algorithms for each step of the normalization process.

6.1 Future Research Directions

There are several exciting avenues for future work.

- **Distributed Processing**: We could extend the architecture to run on a cluster for processing truly massive datasets.
- **Machine Learning Integration**: It would be interesting to incorporate learned heuristics to predict and validate dependencies.
- **Interactive Normalization**: We could build features to support user-guided normalization with real-time feedback.
- **Quality Metrics**: There's an opportunity to develop more sophisticated ways to measure the quality of a normalized schema.
- **Incremental Processing**: The system could be enhanced to support dynamic schema changes and incremental updates.

The theoretical framework we've laid out should provide a solid foundation for future work in automated database design and schema optimization.

REFERENCES

[1] Philip A Bernstein. 1976. Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems (TODS)* 1, 1 (1976), 30–59.
[2] Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.
[3] David J DeWitt and Jim Gray. 1992. *Parallel database systems: the future of high performance database systems*. Vol. 35. ACM New York, NY, USA.
[4] Ronald Fagin. 1977. Multivalued dependencies and a new normal form for relational databases. *ACM transactions on database systems (TODS)* 2, 3 (1977), 262–278.
[5] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. In *The computer journal*, Vol. 42. Oxford University Press, 100–111.
[6] Yannis E Ioannidis. 1996. Query optimization. *Comput. Surveys* 28, 1 (1996), 121–123.
[7] Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2016. Efficient discovery of approximate dependencies. *Proceedings of the VLDB Endowment* 9, 4 (2016), 216–227.
[8] Heikki Mannila and Kari-Jouko Räihä. 1992. Design of relational databases. *Addison-Wesley Longman Publishing Co., Inc.* (1992).
[9] Nicolas Novelli and Rosine Cicchetti. 2001. FUN: An efficient algorithm for mining functional and embedded dependencies. In *International conference on database theory*. Springer, 189–203.
[10] Thorsten Papenbrock and Felix Naumann. 2015. Functional dependency discovery: an experimental evaluation of seven algorithms. In *Proceedings of the VLDB Endowment*, Vol. 8. VLDB Endowment, 1082–1093.
[11] Millist W Vincent, Jixue Liu, and Chengfei Liu. 1999. A measure of quality for database schemas. In *International Conference on Conceptual Modeling*. Springer, 312–327.