# HIERARCHICAL GRAPH LEARNERS FOR CARDINALITY ESTIMATION

Anonymous authors

Paper under double-blind review

### Abstract

Cardinality estimation – the task of estimating the number of records that a database query will return - is core to performance optimization in modern database systems. Traditional optimizers used in commercial systems use heuristics that can lead to large errors. Recently, neural network based models have been proposed that outperform the traditional optimizers. These neural network based estimators perform well if they are trained with large amounts of query samples. In this work, we observe that data warehouse workloads contain highly repetitive queries, and propose a hierarchy of localized on-line models to target these repetitive queries. At the core, these models use an extension of Merkle-Trees to hash query graphs which are directed acyclic graphs. The hash values can divisively partition a large set of graphs into many sets, each containing few (whole) graphs. We learn an online model for each partition of the hierarchy. No upfront training is needed; on-line models learn as the queries are executed. When a new query comes, we check the partitions it is hashed to and if no such local model was sufficiently confident along the hierarchy, we fall-back onto a default model at the root. Our experimental results show that not only our hierarchical on-line models perform better than the traditional optimizers, they also outperform neural models, with robust errors rates at the tail.

027 028 029

031

025

026

004

010 011

012

013

014

015

016

017

018

019

021

## 1 INTRODUCTION

Cardinality estimation plays a pivotal role in query optimization of relational databases, as the query optimizer uses these estimates to order the operators in the query graph and minimize data movement. The goal of cardinality estimation is to estimate the number of records returned by each query operator to answer a SQL query, without actually executing the query. Traditional cardinality estimation methods in databases like PostgreSQL rely on single column statistics (e.g., histogram and sketches), sampling, and sometimes "magic" constants. These methods, however, can lead to significant estimation errors when underlying data assumptions, such as independence between table column and uniform data distribution within columns, are violated (Leis et al., 2015).

Recently, several methods propose neural models for cardinality estimation (Kipf et al., 2019; Zhu et al., 2021; Negi et al., 2023), without making such simplifying assumptions. The core idea frames cardinality estimation as supervised learning and train machine learning models on representative (query, cardinality) observations. While learned methods show promising results, they require a large number of training data. Note that running lots of queries, especially over large collections of data, to collect training labels is very expensive, probably requiring hours-to-days of human and machine time.

We observe that database workloads in cloud databases for analytical workloads such as Google BigQuery or Amazon Redshift contain highly repetitive queries (van Renen et al., 2024) - 50% of the real world clusters have more than 90% queries repeated in templates (only changing the constant parameters). In this paper, we focus on these workloads and propose a hierarchy of localized on-line models to target these repetitive queries. Our method falls back to a default model for non-repetitive queries. These models use an extension of Merkle-Trees to hash query graphs which are directed acyclic graphs. The hash values can divisively partition a large set of graphs into many sets, each containing few (whole) graphs. We learn a separate model for each partition of the hierarchy. While

054 graph sizes can vary, graphs of identical structure (within a partition) must all have the same total 055 feature dimensionality. 056

Briefly, to enable this, our method employs *templatizers*. Each templatizer removes features X from 057 the input graph G emitting remaining graph structure ("template") T. We then compute a hash  $\#_T$ 058 of the template T. A canonical and permutation-invariant ordering of nodes, preserves their position 059 within the feature vector. For inference on test query, cardinality is estimated using all X's sharing 060 the same hash of the test graph. We start searching the hierarchy at the leaves; if the current template 061 has enough data points to make a prediction, then we use the model at that level, otherwise to move 062 to the next level, until we fall back to a default model at the root, which can be a traditional optimizer 063 or a learned cardinality estimator.

064 Our experimental studies show that our model can already learn to predict cardinality with a high 065 accuracy especially if repetitiveness is high. Our models outperform traditional and neural models, 066 and produce better accuracy even at the tail (P90 and P95). Moreover, by organizing the templates 067 in a hierarchy, we show that we can learn robust models since leaf templates are more specific and 068 thus can be trained with a few examples while templates in the higher levels need more examples 069 but are better in generalizing in case queries are different from what we have seen so far.

070 **Outline.** The rest of the paper is organized as follows: In  $\S2$ , we define hierarchical graph tem-071 plates and discuss the core method. §3 describes how these hierarchical graph learners are used 072 for cardinality estimation. §4 and §5 contain the detailed experimental study and the related work, 073 respectively. Finally, we conclude in  $\S6$ .

074 075 076

077

078

084

085

087

088

089

090 091 092

#### 2 HIERARCHICAL GRAPH TEMPLATES

### 2.1 **DEFINITIONS**

079 **Basic Notation.** Let [n] be the set of integers  $\{1, 2, ..., n\}$ . Let  $\pi \in \mathbb{Z}^n$  be a permutation of [n]. Let  $\{0,1\}^h$  be a bit-vector of length h and let  $\{0,1\}^*$  denote a bit-vector of arbitrary length. 081 We denote a (cryptographic) hashing function  $(0, 1)^* \to (0, 1)^h$ . Functional dom(.) accepts a 082 function as an argument and returns the domain of the argument. 083

Heterogeneous Directed Acyclic Graphs. Let  $\mathcal{G}$  denote the space of heterogeneous directed acyclic graphs (DAGs). An instance  $G \in \mathcal{G}$  has three parts:  $G = (\mathcal{V}, \mathcal{E}, f)$ , respectively, (nodes, 086 edges, features). Let  $|\mathcal{V}|$  denote the cardinality of  $\mathcal{V}$ . For simplicity, we assume nodes are integers, *i.e.*,  $\mathcal{V} \triangleq [|\mathcal{V}|]$ . We assume edge-set  $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$  encodes a DAG. This assumption is **necessary** for our DAG hashing function (§2.4). Finally, every node  $v \in \mathcal{V}$  has an associated "feature dictionary"  $f^{(v)}$ . We demonstrate two example  $f^{(v)}$ 's (pertaining to our application, §3):

$$f^{(v)} = \{n: \text{``movies'', c: 10000, i: 5days} \}, \qquad \text{for } v = \text{movies' table in Fig. 2a; (1)}$$
$$f^{(v)} = \{n: \text{``year'', t: int, u: 65, min: 1960, max: 2024} \}, \qquad \text{for } v = \text{year' column in Fig. 2a (2)}$$

093 094

101 102

 $f^{(v)}:\mathbb{Z}\to\Psi$  can be interpreted as a function that maps categories ( $\in\mathbb{Z}$ ) onto arbitrary objects 095  $(\in \Psi)$ . Our algorithm handles any object types, however, objects (1) must be representable as 096  $\{0,1\}^*$  (see §2.4) and (2) if it is used for learning, must be accompanied with **featurizer** function 097  $\psi: \Psi \to \mathbb{R}^{d_{\psi}}$ , where  $d_{\psi} \in \mathbb{Z}_+$  is dimensionality of extracted feature (see §3.2). We use subscript 098 notation to access feature values:  $f_{u}^{(v)}$  denotes the value at key u (in Eq. 2,  $f_{u}^{(v)} = 65$ ). Notation 099  $f_{\mathbf{S}}^{(v)}$  reads a set of features. Formally, 100

$$f_S^{(v)} = \{ \mathbf{s} \colon f_s^{(v)} \mid s \in \mathbf{S} \} \quad \text{for all} \quad \mathbf{S} \subseteq \operatorname{dom}(f^{(v)}).$$
(3)

103 For instance,  $f_{\mathbf{S}}^{(v)} = \{c: 10000, i: 5days\}$  when  $\mathbf{S} = \{c, i\}$  and  $f^{(v)}$  is defined per Eq. 1. 104 105

**Definition 1 (GRAPH ISOMORPHISM)**  $G_1 = (\mathcal{V}, \mathcal{E}_1, f)$  is isomorphic to  $G_2 = (\mathcal{V}, \mathcal{E}_2, z)$ , de-106 noted as  $G_1 \cong G_2$  (equivalently,  $G_2 \cong G_1$ ), if-and-only-if there exists a permutation  $\pi$  such that  $\mathcal{E}_1 = \{(\pi_u, \pi_v) \mid (u, v) \in \mathcal{E}_2\}$  and  $f^{(v)} = f^{(\pi_v)}$  for all  $u \in \mathcal{V}$ . 107



(b) **Template Hierarchy**. Columns correspond to template functions  $H_i \in \mathcal{H}$ , with Feature-Label  $(\mathbf{X}_i, \mathbf{y}_i)$  per template. Leaf templatizer  $H_1$  is the most-granular, grouping identical graphs with constant feature removed. Inference invokes models within each group, along one path from root to leaf (determined by **T**).

Figure 1: Stream of query graphs get indexed into the template hierarchy. Every graph will store its features on a **path from root-to-leaf**. Border-colors of stream queries correspond to  $(\mathbf{X}, \mathbf{y})$  pairs.

139 140 140 140 141 142 Definition 2 (TOPOLOGICAL ORDER) For any directed acyclic graph  $G = (\mathcal{V}, \mathcal{E}, f)$ , there exists one-or-more valid topological orderings. Let  $\pi \in \mathbb{Z}^{|\mathcal{V}|}$  denote one valid ordering.  $\pi$  is considered a valid ordering if  $\pi_v < \pi_{v'}$  for all  $(v, v') \in \mathcal{E}$ .

Definition 2 implies that v should be ordered before v' for all edges  $v \rightarrow v'$ . However, it is important to remember that topological order is not unique. DAGs can have many valid topological orderings.

2.2 TASK: ONLINE SUPERVISED LEARNING ON GRAPHS

148 Our task falls under *supervised learning on graphs* (not *within*<sup>1</sup> graphs). For each graph  $G \in \mathcal{G}$ , we 149 can obtain its (ground-truth) training label as  $y(G) \in \mathcal{Y}$ . We are interested in model  $\hat{y} : \mathcal{G} \to \mathcal{Y}$ 150 to approximate y(G) for every  $G \in \mathcal{G}$ . Graph Neural Networks (GNNs) (Chami et al., 2022), with 151 graph-pooling, are valid candidates for  $\hat{y}$ .

Further, we are interested in an **online setting**. Databases can receive query stream from users, during which, cardinality estimates can be obtained per incoming query (*e.g.*, to optimize joinorder). We wish to **incrementally improve our models**, as we collect observations from the stream.

155 156 157

143

144

145 146

147

2.3 GRAPH TEMPLATE EXTRACTIONS

We define "*templatizer*" function  $H : \mathcal{G} \to \mathcal{G} \times \mathbb{R}^d$ . Given graph  $G \in \mathcal{G}$ , The outputs of H(G) are (1) "*template*"  $T \in \mathcal{G}$ , *i.e.*, copy of the graph structure of G but many features are removed and (2)

<sup>160</sup> 161

<sup>&</sup>lt;sup>1</sup>While many recent GNN methods focus on node- or edge-level tasks, *e.g.*, node-classification or link-prediction, our method is designed for graph-level tasks, *e.g.*, graph classification or regression.



228

229 230 231

232 233

234

235

236

237

238

239 240

241

246

252

253

258 259

260

261

268

269

216

217

218

219

220

221

222

Algorithm 1 Procedures of Template History Learner

1: input hyperparameter:  $\mathcal{H} = \{H_1, H_2, \dots\}$  (defined in Eq. 5) 2: initialize:  $\mathcal{F} \leftarrow \{\}$ 3: initialize:  $\hat{y} \leftarrow MASTERMODEL()$  per Eq.10 4: function ADDEXAMPLE(G, y(G)) 5: for  $H_i \in \mathcal{H}$  do  $(T, \mathbf{x}) \leftarrow H_i(G)$ 6: 
$$\begin{split} \mathbf{X}_{i}^{[\#T]} &\leftarrow \mathbf{X}_{i}^{[\#T]} \cup \{\mathbf{x}\} \\ \mathbf{Y}_{i}^{[\#T]} &\leftarrow \mathbf{Y}_{i}^{[\#T]} \cup \{y(G)\} \end{split}$$
7: 8: 9: function INFER(G) $\mathbf{z} \leftarrow \left\{ \mathcal{F}_i^{[\#_T]}(G, \mathbf{x}) \mid H_i \in \mathcal{H} \text{ and } (T, \mathbf{x}) \leftarrow H_i(G) \right\}$ 10: return  $\widehat{y}(\mathbf{z})$ 11:

#### 2.5 ONLINE LEARNING OF TEMPLATE HIERARCHIES

**Outline.** Given a stream of graphs  $\mathcal{D} = (G_1, G_2, ...)$ , we design an algorithm that can make prediction on every  $G_j \in \mathcal{D}$  using all prior  $\{G_k \in \mathcal{D} \mid k < j\} \triangleq \mathcal{D}_{<j}$ . The main idea is to learn many (simple) models. All graphs whose templates are isomorphic share the same model. Each  $H_i \in \mathcal{H}$  processes every  $G \in \mathcal{D}$ . Templatizer  $H_i(G)$  extracts graph template T and features  $\mathbf{x}$ . For inference, model associated with  $\#_T$  is retrieved<sup>2</sup>, then invoked to predict y from  $\mathbf{x}$ . All  $|\mathcal{H}|$ predictions can be combined with high-level master model  $\hat{y} : \mathbb{R}^{|\mathcal{H}|} \to \mathcal{Y}$ . For training, once the ground-truth answer y(G) is retrieved, models update to learn from  $(\mathbf{x}, y(G))$ .

**Feature-Label Matrices per (templatizer, template)-Pair.** Given templatizer  $H_i \in \mathcal{H}$ , an arbitrary template T produced by  $H_i$ , and timestamp  $j \leq |\mathcal{D}|$ , then the set

$$\mathbf{X}_{i,(7)$$

Can be cast as matrix, since its rows of  $\mathbf{x}_j \in \mathbb{R}^{d_T}$  are of the same<sup>3</sup> dimensionality, whose graph templates are isomorphic. Hence,  $\mathbf{X}_{i,< j}^{[\#T]} \in \mathbb{R}^{\circ \times d_T}$  and label matrix

$$\mathbf{Y}_{i,(8)$$

**Model per (templatizer, template)-Pair.** Let  $\mathcal{F}_i^{[\#_T]} : \mathbb{R}^{d_T} \to \mathcal{Y}$  denote model specialized for template T of  $H_i$ . There are many possibilities for  $\mathcal{F}_i^{[\#_T]}$ , which we co-design with corresponding  $H_i$  (see §3.3). **Inference** on subsequent G can run  $|\mathcal{H}|$  (parallel) invocations:

$$\mathbf{z}_{G} = \left\{ \mathcal{F}_{i}^{[\#_{T}]}(\mathbf{x}) \mid H_{i} \in \mathcal{H} \text{ and } (T, \mathbf{x}) \leftarrow H_{i}(G) \right\} \in \mathbb{R}^{m}.$$
(9)

then invoke  $\hat{y}(\mathbf{z}_G)$ . For **training**, some models  $\mathcal{F}_i^{[\#_T]}$  update periodically using  $\left(\mathbf{X}_{i,< j}^{[\#T]}, \mathbf{Y}_{i,< j}^{[\#T]}\right)$ , while others incrementally absorb each incoming observation  $(G_j, y(G_j))$  – see, §3.3). Nonetheless, learning can happen in parallel for all  $\mathcal{F}_i^{[\#_T]}$ .

Algorithm 1 defines routines (ADDEXAMPLE, INFER), initializes master model  $\hat{y}$ , and initializes data structure  $\mathcal{F}$  to a Hashtable. At every  $G_j \in \mathcal{D}$ , routine INFER $(G_j)$  can return the estimated quantity of interest (*e.g.*, cardinality), by invoking  $\hat{y}$  on the output of  $|\mathcal{H}|$  invocations of  $\mathcal{F}$ . Once the caller retrieves the ground-truth value  $y(G_j)$  (*e.g.*, as the query results are assembled) then routine ADDEXAMPLE can incorporate the example  $(G_j, y(G_j))$  into the (simple) models within  $\mathcal{F}$ .

<sup>&</sup>lt;sup>2</sup>All models are small (kept in RAM). In practice, since probability of false collision is low albeit non-zero, the *actual* hashtable keys we use are  $(\#_T, d_T)$ , i.e., pairing with dimension of the x produced alongside T.

 $<sup>{}^{3}</sup>d_{T} = \sum_{v \in \mathcal{V}} \sum_{(j,\psi) \in \mathbf{S}_{T}^{(v)}} d_{\psi}.$ 

#### HIERARCHICAL GRAPH LEARNERS FOR CARDINALITY ESTIMATION

#### 3.1 TEMPLATIZATION

We studied three templatization strategies,  $H_1, H_2, H_3$ , ranging from fine-grained to coarse-grained templates. Table 1 shows the feature sets kept in the template T VS extracted to the dense vector x, for every  $H_i$ . For example, The fine-grained  $H_1$  removes just the {constant value} from the template. Hence query graphs found in the same  $H_1$  template differ only by the constant values.

Table 1: Templatization Strategies. Each  $H_i$  templatizes as  $(T_i, \mathbf{x}_i) \leftarrow H_i(G)$  where T and  $\mathbf{x}$ include features listed in, respectively,  $S_t$  and  $S_x$ . The choices  $S_t$  induce a divisive hierarchy as every  $S_t$  row includes the information of the next row (column name determines column type).

| Templatizer | Hash features $\mathbf{S}_t$            | Dense (model) features $\mathbf{S}_x$                  |
|-------------|---|--|
| $H_1$       | {Table name, column name, predicate op} | {constant value}                                       |
| $H_2$       | {Table name, column name}               | {constant value, predicate op}                         |
| $H_3$       | {Table name, column type}               | {constant value, predicate op,<br>column unique value} |

### 3.2 FEATURIZERS

The templatizer extracts features of many types into x, including numeric, string, date, time, boolean, respectively, we use featurizers  $\psi$  as, identity, ASCII of first-3 characters (in base 256), as numeric YYYYMMDD, as numeric hhmmss, as  $\{0, 1\}$ . Finally, we map each into the range [0, 1]. We explore two scaling techniques: normalizing (ie.  $\frac{v-\min}{\max-\min}$ ) and replacing with percentile. We map predicate operators (>, <, =, or, and, ...) to unique integers.

Further, we add one more feature that our models find useful: combining combine the constant with the predicate operator to produce range vector. For example, " $\leq 2000$ " is featurized as [0, 0.3], "= 2000" becomes [0.3, 0.3], and " $\geq$  2000" becomes [0.3, 1] (supposing constant 2000 scales to 0.3); all other predicate operators are currently featurized as [0, 1] for simplicity.

## 3.3 LEARNING

We use a rule-based  $\hat{y}$ . Its output  $\hat{y}(G)$  can be concisely described with a flow-chart:

$$\operatorname{start} \rightarrow \widehat{\mathfrak{y}}(G) = \underbrace{\operatorname{if}}_{f_1} \underbrace{\operatorname{else}}_{s(\#_{T_1}) > \tau_1} \rightarrow \underbrace{\operatorname{sl}}_{s(\#_{T_2}) > \tau_2} \rightarrow \underbrace{\operatorname{sl}}_{s(\#_{T_3}) > \tau_3} \rightarrow \underbrace{\operatorname{root}(G)}_{r_1(\#_{T_1})}$$

$$\underbrace{\operatorname{vhen}}_{\mathcal{F}_1^{[\#_{T_1}]}(\mathbf{x}_1)} \underbrace{\operatorname{vhen}}_{\mathcal{F}_2^{[\#_{T_2}]}(\mathbf{x}_2)} \underbrace{\operatorname{vhen}}_{\mathcal{F}_3^{[\#_{T_3}]}(\mathbf{x}_3)} , \quad (10)$$

where the history size  $s(\#_{T_i})$  equals the number of observations that hash to  $\#_{T_i}$ , *i.e.*, the height of matrices  $\mathbf{X}_{i}^{[\#T_{i}]}$  and  $\mathbf{Y}_{i}^{[\#T_{i}]}$ . Given graph  $G \in \mathcal{G}$ , let  $(T_{i}, \mathbf{x}_{i}) \leftarrow H_{i}(G)$  for  $i \in \{1, 2, 3\}$ . Further, let  $\tau_1 < \tau_2 < \tau_3$  denote "activation thresholds"<sup>4</sup>. If the size of  $\#_{T_i}$  meets the threshold  $\tau_i$ , we invoke the corresponding  $\mathcal{F}_i$ . If not, we move on to the next hierarchy level.

- root(G) will be invoked when incoming query G has an unfamiliar template (a.k.a, the cold-start problem). We propose to set root(G) to a default estimator, eg. Postgres.
- We try-out several choices for  $\mathcal{F}$ . All showing incremental and/or instant training, *e.g.*,

Linear Regression: 
$$\mathcal{F}_i^{[\#T]}(\mathbf{x}) = \left(\mathbf{X}_i^{[\#T]}\right)^{\dagger} \mathbf{Y}_i^{[\#T]} \mathbf{x}$$
 (11)

Gaussian Kernel: 
$$\mathcal{F}_{i}^{[\#T]}(\mathbf{x}) = \frac{\sum_{j} \mathbf{Y}_{i,j}^{[\#T]} e^{-d(\mathbf{x}, \mathbf{X}_{i,j}^{[\#T]})}}{\sum_{j} e^{-d(\mathbf{x}, \mathbf{X}_{i,j}^{[\#T]})}}$$
 (12)

**Gradient-Boosted Decision Trees:** implementation of (Chen & Guestrin, 2016) (13) <sup>4</sup>It will always be that  $s(\#_{T_1}) < s(\#_{T_2})$ , due to the divisive hierarchy

where  $(.)^{\dagger}$  denotes Moore-Penrose inverse and d(.,.) denotes distance function (see Appendix). For linear regression (Eq. 11), we add<sup>5</sup> column of 1 to x and to  $\mathbf{X}_{i}^{[\#T]}$ . As observations  $\left(\mathbf{X}_{i}^{[\#T]}, \mathbf{Y}_{i}^{[\#T]}\right)$ grow, it is unnecessary to re-compute (from scratch) the pseudo-inverse  $(.)^{\dagger}$ . It can be incrementally updated, *e.g.*, with rank-1 changes to the Singular Value Decomposition of  $\mathbf{X}_{i}^{[\#T]}$ , per Brand (2006).

#### 4 EXPERIMENTAL EVALUATION

324

325

326

327

328 329 330

331 332

333

334

339

340

341

342

343 344

345 346

347

348

349

350

351 352

353

354

355

356

357

358 359

360

377

**Metrics.** We quantify the error of cardinality estimate  $\hat{y}(G)$  and true (label) cardinality y(G) with:

 $\begin{aligned} Q_{\rm err} &= \max\left(\frac{y}{\widehat{y}}, \frac{\widehat{y}}{y}\right) \quad (14) \qquad A_{\rm err} = |\widehat{y} - y| \quad (15) \qquad R_{\rm err} = 1 - \frac{\min(\widehat{y}, y)}{\max(\widehat{y}, y)} \\ \text{respectively known as Q-error, absolute error, and relative error.} \end{aligned}$ (16)

Datasets. We run experiments on several database workloads, downloaded from benchmark (Cardbench, Chronis et al., 2024) (prefixed "binaryjoin-" within figures). Further, we extend their query generator to: 1) enable multi-way join queries (up-to 5 joins) to increase the query complexity; 2) incorporate the high repetiveness feature of data warehouse workloads as in Redshift (van Renen et al., 2024) (prefixed "multijoin-"). For all multijoin datasets, we fixed the sample constant size at 10 and varied the sample size (repetition rate) to evaluate its impact on accuracy in Fig 4.

**Models.** We use Cardinality Estimation models – Postgres, MSCN, ours  $\{H_i, \mathcal{F}_i\}_i$ .

- (1) Postgres: Traditional histogram-based estimator implemented in open-source PostgreSQL (PostgreSQL Group). This estimator can be invoked on any query (100% admit rate).
- (2) MSCN: Neural-based estimator (Kipf et al., 2019). We train two model copies, per database workload: "MSCN" and "MSCN+", respectively, on 1000 query graphs and on 25% of the graphs (3.3X-10X vs MSCN). Crucially, MSCN cannot admit queries containing "or" predicates<sup>6</sup>. On our workloads, MSCN admits 61% of the queries.
- (3) **Ours**: History-based estimator. We infer using  $(\mathcal{F}_i, H_i)$  per Eq.11–13, either for singular i = 1or multiple  $\{(\mathcal{F}_i, H_i)\}_{i \in \{1,2,3\}}$  that live on a hierarchy (§2.5). Singular  $(\mathcal{F}_i, H_i)$  can estimate only if there are enough observations of template  $\{T_i \leftarrow H_i(G)\}$ .

**Overview.** We conduct three kinds of experiments: §4.1 evaluates the **practical scenario** that inference is required for all queries. Here, a method can fall-back onto another. §4.2 conducts apples-to-apples comparison of our models against prior work; §4.3 Ablates our models;

4.1 HIERARCHICAL MODELS

361 In this set of experiments, methods **must always make a prediction**. Our method defaults to the Postgres estimator, in cases, where the graph structure is novel (has not appeared earlier in the online 362 setting). Our full hierarchy, depicted in Figure 1b and formalized in Equation 10, is abbreviated 363  $(H_1, H_2, H_3, \mathbb{P})$ , where  $\mathbb{P}$  denoting Postgres estimator. We set thresholds  $(\tau_1, \tau_2, \tau_3)$  in Eq.10 to (3, 364 10, 100) and employ Gradient-Boosted Decision Trees (GBDT) at each hierarchical level. 365

How effective are hierarchical learners? Table 2 compares hierarchical models with different 366 hierarchy combinations. Comparing  $(H_1, H_2, H_3, \mathbb{P}), (H_2, H_3, \mathbb{P}), (H_3, \mathbb{P})$ , and Postgres, we can 367 see the models keep improving when we add more levels of hierarchy and the full hierarchy of 368 models is always better than Postgres at all metrics. In addition, The full hierarchy leverages each 369 level effectively, as evidenced by the activation ratios (0.69, 0.04, 0.01, 0.26) for  $H_1$ ,  $H_2$ ,  $H_3$ , and 370 Postgres, respectively. These results demonstrate the effectiveness of our hierarchical models in 371 leveraging historical data to enhance the cardinality estimation capabilities of traditional optimizers. 372

The necessity of multiple hierarchy? Table 2 also shows the need of hierarchy. Comparing 373  $(H_1, \mathbb{P}), (H_1, H_2, \mathbb{P}), (H_1, H_2, H_3, \mathbb{P})$ , the latter two consistently outperform the first. This in-374 dicates that a simple hierarchy  $(H_1, P)$  is insufficient, highlighting the importance of multi-level 375 hierarchies. 376

<sup>5</sup>Equivalent to adding bias-term to one-layer model.

<sup>&</sup>lt;sup>6</sup>As-is, MSCN (Kipf et al., 2019) was developed for conjunctions only, its extension is beyond our scope.

|                               |               |                     |                   |                   |                         |               |               |                   | · ·               |                   |
|-------------------------------|---------------|---------------------|-------------------|-------------------|-------------------------|---------------|---------------|-------------------|-------------------|-------------------|
| hierarchy                     | $R_{\rm err}$ | $A_{\rm err}$       | $Q_{ m err}^{50}$ | $Q_{ m err}^{90}$ | $Q_{ m err}^{95}$       | $R_{\rm err}$ | $A_{\rm err}$ | $Q_{ m err}^{50}$ | $Q_{ m err}^{90}$ | $Q_{ m err}^{95}$ |
|                               |               | multijoin-cms       |                   |                   | multijoin-stackoverflow |               |               |                   |                   |                   |
| Postgres                      | 0.70          | $2.4e^{5}$          | 3.33              | 112               | $2.3e^{3}$              | 0.79          | $2.8e^{5}$    | 4.85              | 360               | $3.1e^{3}$        |
| $(H_3,\mathbb{P})$            | 0.69          | $2.2e^{5}$          | 3.21              | 110               | $2.2e^{3}$              | 0.77          | $1.8e^{5}$    | 4.30              | 367               | $3.8e^3$          |
| $(H_2, H_3, \mathbb{P})$      | 0.13          | $2.0e^{4}$          | 1.15              | 46.67             | 159                     | 0.14          | $1.7e^{3}$    | 1.16              | 44.33             | 464               |
| $(H_1, P)$                    | 0.06          | $9.1e^{3}$          | 1.07              | 22.22             | 97.00                   | 0.10          | 456           | 1.12              | 21.03             | 200               |
| $(H_1, H_2, \mathbb{P})$      | 0.06          | $8.5e^3$            | 1.06              | 20.10             | 94.48                   | 0.10          | <b>388</b>    | 1.11              | 18.01             | 182               |
| $(H_1,H_2,H_3,\mathbb{P})$    | 0.06          | $8.5e^3$            | 1.06              | 20.10             | 94.48                   | 0.10          | 388           | 1.11              | 18.01             | 182               |
|                               |               | mult                | ijoin-ac          | cidents           |                         |               | mu            | ltijoin-a         | irline            |                   |
| Postgres                      | 0.39          | $8.8e^{7}$          | 1.65              | 10.31             | 18.29                   | 0.39          | $2.6e^{4}$    | 1.63              | 97.30             | 216               |
| $(H_3,\mathbb{P})$            | 0.25          | $3.1e^{7}$          | 1.34              | 8.93              | 20.60                   | 0.37          | $2.4e^{4}$    | 1.59              | 97.00             | 216               |
| $(H_2, H_3, \mathbb{P})$      | 0.13          | $1.2e^{7}$          | 1.15              | 4.81              | 15.42                   | 0.17          | $6.0e^{3}$    | 1.20              | 13.88             | 91.00             |
| $(H_1, \mathbb{P})$           | 0.13          | $1.1e^{7}$          | 1.15              | 4.95              | 17.25                   | 0.12          | $3.2e^{3}$    | 1.13              | 4.50              | 29.20             |
| $(H_1, H_2, \mathbb{P})$      | 0.13          | $1.1e^7$            | 1.15              | 5.02              | 17.70                   | 0.12          | $3.1e^3$      | 1.13              | 4.29              | 25.00             |
| $(H_1, H_2, H_3, \mathbb{P})$ | 0.13          | $1.1e^7$            | 1.15              | 5.02              | 17.70                   | 0.12          | $3.1e^3$      | 1.13              | 4.29              | 25.00             |
|                               |               | mult                | ijoin-en          | ıployee           |                         |               | m             | ultijoin          | -geo              |                   |
| Postgres                      | 0.35          | $1.2e^{3}$          | 1.54              | 3.38              | 4.83                    | 1.00          | $9.2e^{6}$    | 224               | $2.1e^{5}$        | $1.2e^{6}$        |
| $(H_3,\mathbb{P})$            | 0.26          | 961                 | 1.35              | 3.14              | 4.42                    | 1.00          | $8.9e^6$      | 218               | $2.1e^{5}$        | $1.2e^{6}$        |
| $(H_2, H_3, \mathbb{P})$      | 0.04          | 481                 | 1.05              | 2.11              | 2.98                    | 0.09          | $1.6e^{4}$    | 1.10              | $5.8e^3$          | $7.3e^{4}$        |
| $(H_1,\mathbb{P})$            | 0.03          | 297                 | 1.03              | 2.09              | 3.07                    | 0.08          | $4.3e^{3}$    | 1.09              | 192               | $1.1e^{4}$        |
| $(H_1, H_2, P)$               | 0.03          | <b>269</b>          | 1.03              | 2.03              | 3.01                    | <b>0.07</b>   | $3.3e^3$      | 1.08              | 66.38             | $7.0e^3$          |
| $(H_1, H_2, H_3, \mathbb{P})$ | 0.03          | <b>269</b>          | 1.03              | 2.03              | 3.01                    | 0.07          | $3.3e^3$      | 1.08              | 66.38             | $7.0e^3$          |
|                               |               | binaryj             | oin-stac          | koverflov         | N                       |               | bina          | ryjoin-           | airline           |                   |
| Postgres                      | 0.69          | $1.5e^{7}$          | 3.28              | 160               | 470                     | 0.55          | $9.3e^{4}$    | 2.22              | 37.17             | 127               |
| $(H_3,\mathbb{P})$            | 0.66          | $9.0e^{6}$          | 2.93              | 149               | 382                     | 0.53          | $2.0e^{5}$    | 2.11              | 63.00             | 206               |
| $(H_2, H_3, P)$               | 0.42          | $1.8e^{6}$          | 1.74              | 60.48             | 183                     | 0.45          | $1.5e^{5}$    | 1.82              | 51.55             | 190               |
| $(H_1, P)$                    | 0.43          | $1.7e^{6}$          | 1.76              | 53.33             | 175                     | 0.44          | $5.3e^{4}$    | 1.80              | 28.15             | 112               |
| $(H_1, H_2, \mathbb{P})$      | 0.40          | $1.5e^{6}$          | 1.66              | <b>44.00</b>      | 174                     | 0.44          | $5.3e^4$      | 1.80              | 28.24             | 112               |
| $(H_1, H_2, H_3, P)$          | 0.39          | $1.5\mathrm{e}^{6}$ | 1.63              | 45.15             | 175                     | 0.44          | $1.4e^{5}$    | 1.80              | 43.00             | 179               |

#### Table 2: Hierarchical models. Median relative error, median absolute error and Q-Error percentiles.

410

378

#### 4.2 COMPARING INDIVIDUAL MODELS

411 In this section, we compare all methods on the intersection of queries they are able to admit – 412 about 25% of queries. While §4.1 shows practical hierarchies that are able to process any query, this provides a sound apples-to-apples comparison. 413

414 Table 3 summarizes the performance of four models: Postgres, MSCN, MSCN+, and only one 415 model-templatizer pair  $(\mathcal{F}_1, H_1)$ , specifically, GBDT (Eq. 13) with  $H_1$ . MSCN+ (trained on  $\approx 5X$ 416 more data) is much better than MSCN and is frequently better than Postgres. Overall, our method 417 is competitive and produces higher accuracy majority of the time. In particular,  $H_1$  is substantially 418 better (10X-50X+) than Postgres half-of-the-time. We also observe that our model is more robust at the tail of the error distribution (P90 and P95). 419

## 420

#### 4.3 ABLATION STUDIES 421

422 **Model Choice.** We compare across choices of models  $\mathcal{F}$  (Eq. 11–13) and  $H_i$ 's in Fig. 3. We find that Gradient-Boosted Decision Trees (GBDT) are consistently strong across different datasets and 423 level of hierarchy, so we choose GBDT for Table 3, and on every level of hierarchy in Table 2. 424

425 Repetition Rate. We modify the workload generator in Chronis et al. (2024) to enable more 426 constants for each predicate in the query. For example, instead of generating a query with predicates 427 "a > 5 AND b = 2", our modified generator will generate "a > 5 AND b = 2", "a > 5 AND b= 20", "a > 1 AND b = 2", "a > 1 AND b = 20" when the sample size is 2, meaning that each 428 429 predicate will have 2 constants to choose from (ie. a > [1, 10], b = [2, 20]). The constant sample sizes in the experiment we choose are [1, 3, 10], therefore it generates the repetition rate of 20%, 430 81% and 91% in query templates. As shown in Fig 4, all templatization strategies exhibit improved 431 performance with increasing workload repetition, while maintaining low q-error levels.



Table 3: Model Errors at various percentiles, per dataset. We **bold** strongest number per (database, q-error percentile).

Figure 4: Accuracy of our learners, as a function of repetition amount. Each chart shows one templatization strategy, containing 4 lines: {Gradient Boosted Decision Tree (Eq. 13), Postgres Estimator }  $\times \{50^{\text{th}}, 90^{\text{th}} \text{ Q-errors}\}$ . The Y-axis displays Q-errors.

482

483

484 485

History Size. We assess the performance of learners as a function of history size, in the Appendix.

# 486 5 RELATED WORK

Learned Cardinality Estimation. In the recent years, several lines of approach learned cardinally 488 estimation have been proposed (Han et al., 2021; Sun et al., 2021; Kim et al., 2022). The first line 489 is workload-driven learning (Kipf et al., 2019; Negi et al., 2023; Reiner & Grossniklaus, 2024), 490 which requires pre-collected workload queries and their executions against the database to collect 491 true cardinalities as the training data. To reduce cost of acquiring training data, the second direction 492 explores data-driven learning (Yang et al., 2019; 2021; Hilprecht et al., 2020; Wu et al., 2023; Kim 493 et al., 2024), which learns a model only on the data capturing its distributions without running any 494 queries. While these models do not have the overhead of running queries, for large databases it could 495 still take hours to train such models. Kim et al. (2024) develops auto-regressive model that samples 496 queries matching filters, crucially supporting string and disjunctive filters. Another line includes 497 localized-models which learn lightweight models that can to capture certain query patterns and can adapt online. Our own work falls into this category. Our method is most-similar to (Malik et al., 498 2007), since they also group queries by templates, and also do learning-and-inference on dense-499 vectors within each template. However, we differ in two ways: (1) The templates of (Malik et al., 500 2007) use a flat vector representation for queries, our are graphs and for grouping we use graph 501 hashes – as such, ours are invariant to node orderings (2) We learn hierarchies of models rather than 502 a flat grouping of models. Moreover, other approaches have explored also different directions to represent queries for localized models. For comparison, Dutt et al. (2019) creates conjunction trees 504 made of simple predicates while Woltmann et al. (2019) learn models on groups of related tables. 505 All these representations are less expressive than query graphs to provide a direct way to represent 506 queries in databases. In fact, our modeling approach to represent queries is very similar to methods 507 used to learn query cost prediction (e.g., execution time) (Hilprecht & Binnig, 2022; Wu et al., 2024) 508 which also uses a query graph representation while our approach uses them to represent groups of 509 similar queries for cardinality estimation.

510 Graph Hashing. Helbling (2020) compute hash values for directed graphs, also by extending 511 Merkle Trees (Merkle, 1988). There are also other methods that can operate on directed but also 512 undirected graphs, including (Portegys, 2008) and WL (Shervashidze et al., 2011). These methods 513 iteratively update node's hash using itself and its neighbors. Each update-round incorporates infor-514 mation from further neighbors. The number of iterations could be set to the graph diameter. Our 515 algorithm slightly differs as our graph nodes *could* be invariant neighbor orders *sometimes* (e.g., or junction), while being variant at *other times* (e.g., > operator). In addition, we only work with DAGs 516 and therefore iterating in topological order terminates the algorithm. 517

518 Decoupled Graph Neural Nets. Our method is also linked to methods that "*decouple*" the graph519 processing step from the learning. Specifically, methods that extract features using the graph and no
520 longer need the graph for learning. These methods include (Wu et al., 2019; Frasca et al., 2020). In
521 that regard, our method also uses the graph for pre-processing. We differ than those methods as they
522 use the structure to propagate information along edges whereas we hash the structure.

523 524

525

## 6 CONCLUSION

526 In this paper, we propose a localized on-line models for cardinality estimation. Queries with isomor-527 phic structures will be grouped-together, with different templatization strategies forming a hierarchy. 528 Within each group, a simple model, e.g., linear regression or gradient-boosted decision trees, can 529 be trained to estimate cardinality of a given query. A predictions is always made at the lowest-level node with sufficient observations, and falls back onto either neural or traditional methods at the root. 530 However, this new query already establishes an observation when the pattern is repeated. In the ex-531 periments, we show that our models outperform traditional and neural models, and produce robust 532 accuracy even at the tail (P90 and P95). Moreover,  $H_1$  is substantially better (10X-50X+) than Post-533 gres half-of-the-time. As future work, we plan to explore different grouping methods, increasing the 534 hierarchy with more templatization strategies, and explore different default models. 535

- 536
- 527
- 538
- 539

# 540 REFERENCES

547

553

566

567

568 569

570

571

576

577

578

582

583

584 585

586

588

- 542 Matthew Brand. Fast low-rank modifications of the thin singular value decomposition. In *Linear* 543 *Algebra and its Applications*, 2006.
- Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy. In *Journal on Machine Learning Research*, 2022.
- Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In SIGKDD Conference on Knowledge Discovery and Data Mining, 2016.
- Yannis Chronis, Yawen Wang, Yu Gan, Sami Abu-El-Haija, Chelsea Lin, Carsten Binnig, and Fatma
   Özcan. Cardbench: A benchmark for learned cardinality estimation in relational databases. In
   *arxiv:2408.16170*, 2024.
- Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment*, 2019.
- Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Benjamin Chamberlain, Michael Bronstein, and
   Federico Monti. Sign: Scalable inception graph neural networks. In *ICML 2020 Workshop on Graph Representation Learning and Beyond*, 2020.
- Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong,
   Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. Car dinality estimation in dbms: a comprehensive benchmark evaluation. *Proc. VLDB Endow.*, 2021.
- Caleb Helbling. Directed graph hashing. In International Conference on Combinatorics, Graph
   Theory & Computing, 2020. URL https://arxiv.org/abs/2002.06653.
  - Benjamin Hilprecht and Carsten Binnig. Zero-shot cost models for out-of-the-box learned cost prediction. *Proc. VLDB Endow.*, 2022.
  - Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. DeepDB: learn from data, not from queries! *Proceedings of the VLDB Endowment*, 2020.
- Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. Learned cardinality estimation: An in-depth study. In *Proceedings of the 2022 International Conference on Management of Data*, 2022.
  - Kyoungmin Kim, Sangoh Lee, Injung Kim, and Wook-Shin Han. Asm: Harmonizing autoregressive model, sampling, and multi-dimensional statistics merging for cardinality estimation. In *Proc.* ACM Manag. Data, 2024.
- Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper.
   Learned cardinalities: Estimating correlated joins with deep learning. In *Biennial Conference* on Innovative Data Systems Research, 2019.
  - Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 2015.
  - Tanu Malik, Randal Burns, and Nitesh Chawla. A black-box approach to query cardinality estimation. In *Biennial Conference on Innovative Data Systems Research (CIDR)*, 2007.
  - R. C Merkle. A digital signature based on a conventional encryption function. In Advances in Cryptology CRYPTO '87. Lecture Notes in Computer Science, 1988.
- Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. Robust query driven cardinality estimation under changing workloads.
   *Proc. VLDB Endow.*, 2023.
  - Tom Portegys. General graph identification by hashing. In arxiv:1512.07263, 2008.

| 594<br>595                      | PostgreSQL Group. Postgresql documentation 17.68.1: Row estimation examples.  |
|---------------------------------|---|
| 596                             | Silvan Reiner and Michael Grossniklaus. Sample-efficient cardinality estimation using geometric   |
| 597                             | deep learning. Proc. VLDB Endow., 2024.   |
| 598<br>500                      | R.L. Rivest. The md5 message-digest algorithm. In Internet Activities Board, 1992.  |
| 600<br>601                      | Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler lehman graph kernels. In <i>Journal of Machine Learning Research</i> , 2011.  |
| 602<br>603<br>604               | Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. Learned cardinality estimation: a design space exploration and a comparative evaluation. <i>Proceedings of the VLDB Endowment</i> , 2021.   |
| 606<br>607<br>608               | Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali<br>Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. Why tpc is not<br>enough: An analysis of the amazon redshift fleet. In VLDB 2024, 2024.  |
| 609<br>610<br>611               | Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. Cardinality estimation with local deep learning models. In <i>Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management</i> , 2019.                                  |
| 612<br>613<br>614               | Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simpli-<br>fying graph convolutional networks. In <i>International Conference on Machine Learning</i> , 2019.   |
| 615<br>616                      | Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. FactorJoin:<br>A New Cardinality Estimation Framework for Join Queries. 2023.  |
| 617<br>618<br>619<br>620<br>621 | Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. Stage: Query execution time prediction in amazon redshift. In <i>Companion of the 2024 International Conference on Management of Data</i> , 2024. |
| 622<br>623<br>624               | Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel,<br>Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation.<br>In <i>Proceedings of the VLDB Endowment</i> , 2019.  |
| 625<br>626<br>627               | Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica.<br>NeuroCard: One cardinality estimator for all tables. VLDB Endowment, 2021.  |
| 628<br>629<br>630<br>631        | Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. Flat: Fast, lightweight and accurate method for cardinality estimation, 2021. URL https://arxiv.org/abs/2011.09022.   |
| 632<br>633<br>634               |   |
| 635<br>636                      |   |
| 637<br>638<br>639               |   |
| 640<br>641                      |   |
| 642<br>643                      |   |
| 645<br>646                      |   |

# 648 A APPENDIX

#### A.1 DATASET STATS

This section presents the statistics of the datasets used in this paper. Importantly, Table 4 presents the repetition rates at different template levels, following the definition from van Renen et al. (2024). Our multi-join workloads, with  $H_1$  repetition rates between 83% and 96%, closely mimic the 90% template repetition rate reported in van Renen et al. (2024). Table 5 summarizes the diverse databases used in our experiments. The smallest databases (accidents and employee) have 3 and 6 tables, respectively, while the largest database (cms\_synthetic\_patient\_data\_omop) comprises 24 tables and 32 billion rows.

|                          |                                 |           | Repetition Rate (%) |       |       |  |
|--------------------------|---------------------------------|-----------|---------------------|-------|-------|--|
| Workload                 | Database                        | # Queries | $H_1$               | $H_2$ | $H_3$ |  |
| multijoin-stackoverflow  | -4                              | 16k       | 91                  | 94    | 95    |  |
| binaryjoin-stackoverflow | stackovernow                    | 13k       | 67                  | 85    | 96    |  |
| multijoin-airline        | -:-1:                           | 20k       | 93                  | 95    | 96    |  |
| binaryjoin-airline       | airine                          | 13k       | 34                  | 56    | 94    |  |
| multijoin-accidents      | accidents                       | 29k       | 95                  | 97    | 98    |  |
| multijoin-cms            | cms_synthetic_patient_data_omop | 14k       | 83                  | 87    | 88    |  |
| multijoin-geo            | geo_openstreetmap               | 13k       | 94                  | 96    | 96    |  |
| multijoin-employee       | employee                        | 62k       | 96                  | 98    | 98    |  |

#### Table 4: Workload Statistics.

#### Table 5: Database Statistics.

| # Tables | # Columns                       | # Rows                                     | # Join Paths  |
|----------|---------------------------------|--|---|
| 14       | 187                             | 3.0B                                       | 13  |
| 19       | 119                             | 944.2M                                     | 27  |
| 3        | 43                              | 27.4M                                      | 2   |
| 24       | 251                             | 32.6B                                      | 22  |
| 16       | 81                              | 8.3B                                       | 15  |
| 6        | 24                              | 48.8M                                      | 5   |
|          | <b># Tables</b> 14 19 3 24 16 6 | # Tables# Columns1418719119343242511681624 | # Tables# Columns# Rows141873.0B19119944.2M34327.4M2425132.6B16818.3B62448.8M |

#### A.2 HASHING FUNCTION EXTENDED

In this section, we includes the algorithm (Algorithm 2) and comparison table (Table 6) to further illustrate the hashing function in Section 2.4.

Table 6: Input data requirements. Merkle's method is designed for balanced search trees (BSTs), with features only on leaf nodes. Our generalization (Alg. 2) produces identical output to Merkle's when input is BST, additionally generalizing to DAG inputs.

| Comparison                    | Merkle Trees (Merkle, 1988) | DAG Hashing (Alg. 2)         |  |  |
|-------------------------------|-----------------------------|------------------------------|--|--|
| Hashable Structure is:        | Tree (w/ virtual edges)     | DAG (edges from query graph) |  |  |
| Input Data (features) are on: | only leaf nodes             | all nodes                    |  |  |
| Neighbors are:                | always ordered              | can be order-invariant       |  |  |

#### A.3 ABLATION STUDIES EXTENDED

We also conduct ablation experiments to show that, in general, our simple models improve as data accumulates in each template (Fig. 5). As  $H_1$  is the most-grained, it stabilizes earlier and has

702 Algorithm 2 Hashing function  $# : \mathcal{G} \to \{0, 1\}^h$  for Directed Acyclic Graphs (DAGs). 703 1: input: hashing function of bit-vectors ( : {0, 1}\*  $\rightarrow$  {0, 1}<sup>h</sup>), e.g., MD5 (Rivest, 1992). 704 2: **input:** Directed Acyclic Graph  $T = (\mathcal{V}, \mathcal{E}, f)$ . 705 3: 706 4: for  $v \in \mathcal{V}$  do  $\mu_v \gets \$(f^{(v)})$ 5: 708 6: for  $v \in \pi$  do // process in topological order 709 if operation v is invariant to order of predecessors then 7: 710 8:  $\mu_v \leftarrow \$(\mu_v || \mathsf{UNORDEREDCOMBINE}(\{\mu_u \mid (u, v) \in \mathcal{E}\})$ 711 9: else // Sometimes, order matters. E.g., A > B differs from B > A712 10:  $\mu_v \leftarrow \$(\mu_v || \mathsf{ORDEREDCOMBINE}(\{\mu_u \mid (u, v) \in \mathcal{E}\}))$ 713 11:  $\pi^* \leftarrow \text{DETERMINISTICTOPOLOGICALORDER}(T, \mu)$ 714 12: **return** (ORDEREDCOMBINE $(\{\mu_v \mid v \in \pi^*\})$ 715 13: 716 14: function OrderedCombine( $\{z \in \{0, 1\}^h\}$ ) 717 15: **return** CONCAT(z) 718 16: **function** UNORDEREDCOMBINE( $\{z \in \{0, 1\}^h\}$ ) 719 17: return CONCAT(sorted(z)) 720 18: **function** DETERMINISTICTOPOLOGICALORDER( $G, \mu$ ) 721  $\pi^* \leftarrow []$ 19: 722 UNPROCESSEDPREV<sub>v</sub>  $\leftarrow \{u \mid (u, v) \in \mathcal{E}\}, \text{ for all } v \in \mathcal{V}$ 20: 21: while  $\pi^*$ .size <  $\mathcal{V}$ .size do 723  $q \leftarrow q \cup \{(\mu_v, v) \mid v \in \mathcal{V} \text{ if } UNPROCESSEDPREV}_v = \emptyset \text{ and } v \notin \pi^*\}$ 22: 724  $(\_, u) \leftarrow \max(q)$ 23: 725  $\pi^*$ . APPEND(u)24: 726 for  $v \in \{v' \mid (u, v') \in \mathcal{E}\}$  do 25: 727 UNPROCESSEDPREV<sub>v</sub>  $\leftarrow$  UNPROCESSEDPREV<sub>v</sub> \{u} 26: 728

729 730

lower tail errors. Notably, the accuracy of coarser templatization, *e.g.*,  $H_3$ , combining records from multiple (columns, predicate operators), needs more training history data to converge. It also shows that GBDT always has better performance than Linear Regression (LR) and Gaussian Kernel(GK) models accross different datasets. This also matches our observation in Figure 3.

- 750 751
- 752
- 753
- 754



Figure 5: Each subplot shows Q-error percentiles as function of amount of history per workload & templatization strategy. In particular, each line color represents learner (Eq.11–13) and each line style represents percentile. History size is less than or equal to x-axis value.