Revisiting Statistical Techniques for Result Cardinality Estimation

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Technology (Research)

IN THE

Faculty of Engineering

BY

Dhrumilkumar Shah

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2021

Declaration of Originality

I, **Dhrumilkumar Shah**, with SR No. **04-04-00-10-22-18-1-15721** hereby declare that the material presented in the thesis titled

Revisiting Statistical Techniques for Result Cardinality Estimation

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2018-21**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: 11/06/2021 Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof. Jayant R. Haritsa Advisor Signature

DEDICATED TO

My Friends and Family

for their support and encouragement

Acknowledgements

First and foremost, I am highly grateful to my supervisor, Prof. Jayant Haritsa, for his invaluable advice, continuous support, and patience during my research study. His immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. The discussions with him have always uplifted my mindset along with my level of understanding regarding the subject or topic.

I am very thankful to Dr. Anshuman Dutt for providing me the unparalleled guidance for working on this widespread research area. The conversations with him have always brought me more clarity and understanding regarding my research problem. His ability to think deeply has always inspired me to do the same and taught me how to approach a research problem.

I also want to thank all my lab mates for their invaluable comments and constructive criticism, which made my thesis stronger and more precise. I would like to thank the Indian Institute of Science and the Department of Computer Science and Automation for providing an inspiring learning environment to improve my learning curve. Last but not least, I want to thank all my friends for supporting me throughout this journey.

Abstract

The Relational Database Management Systems (RDBMS) constitute the backbone of today's information-rich society, providing a congenial environment for handling enterprise data during its entire life cycle of generation, storage, maintenance, and processing. The Structured Query Language (SQL) is the standard interface to query the information present in RDBMS-based storage. Knowing the expected size of the SQL query result, measured in terms of the output row-cardinality, prior to execution can benefit both the RDBMS system and the user in several ways. The use-cases include assessing query feasibility, approximate query answering, query progress monitoring, and resource allocation strategies. In the context of our work, we define *cardinality estimation* as the estimation of the result size (number of rows in the output) of the given SQL query.

Unfortunately, the histogram and sampling-based techniques commonly used in industrial database engines for cardinality estimation are often woefully inaccurate in practice. This lacuna has motivated a recent slew of papers advocating the use of machine-learning/deep-learning techniques for cardinality estimation. However, these new approaches have their own limitations regarding significant training effort, inability to handle dynamic data-updates, and generalization to unseen queries.

In this work, we take a relook at the traditional random sampling and investigate whether they can be made to work satisfactorily when augmented with lightweight data structures. Specifically, we present *GridSam* - a Grid-based Dynamic Sampling technique, which essentially augments random sampling with histograms, incorporating both algorithmic and platform innovations.

From the algorithmic perspective, GridSam first creates a multi-dimensional grid overlay by partitioning the data-space on "critical" attributes, and then performs dynamic sampling from the confined query-specific region of the grid to capture correlations. A greedy methodology targeted towards reducing the Zero Sample Problem occurrence is used to determine the set of "critical" attributes as the grid dimensions. Further, insights from the Index-based Join Sampling (IBJS) technique are leveraged to direct the sampling in multi-table queries. Finally,

learned-indexes are incorporated to reduce the index-probing cost for join sampling during the estimation process.

From the platform perspective, GridSam leverages the massive parallelism offered by current GPU architectures to provide fast grid setup times. This parallelism is also extended to the run-time estimation process.

A detailed performance study on benchmark environments indicates that GridSam computes cardinality estimates with accuracies competitive to contemporary learning-based techniques. Moreover, it does so while achieving an orders-of-magnitude reduction in setup time. Further, the estimation time is in the same ballpark as both traditional and learning-based techniques. Finally, a collateral benefit of GridSam's simple and highly parallelizable design is that, unlike learned estimators, it is amenable to dynamic data environments with frequent data-updates.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Structured Query Language (SQL) is the standard interface to query information present in relational database management systems (RDBMS), which are the typical data processing platforms for enterprise applications. Knowing the expected *size* of the SQL query result, measured in terms of the output row-cardinality, prior to execution can benefit both the RDBMS system and the user in several ways. Thus, several applications of estimating the result size of the given SQL query have emerged as follows:

1. The result size of the SQL query can be useful in determining **resource allocation** strategies to store the output of the given query.

2. Estimation of the query result size can give several kinds of **feedback to system/user** as follows:

   - An expensive query can be detected based on the query result size. The system/user might want to skip the query evaluation depending upon the size of the query result [38].
   - Users may want to check the query for certain types of errors depending upon the query result's size (output number of rows).

3. Estimating the query result size can also be used in **statistical analysis** of data, where users are frequently interested only in the **approximate size** of the query result [34, 37].

For this work, we define *cardinality estimation* in RDBMS as estimating the SQL query result size.

Cardinality estimation is almost a 4-decade-year-old long-standing problem in the database systems world. Various types of models have been proposed in the literature for cardinality

estimation - ranging from statistical-based models (using histogram-based and sampling-based techniques) to recent learning-based models (using machine learning and deep learning techniques). The cardinality estimation module in current industrial database engines is typically implemented as the statistical-based models using histogram-based or sampling-based techniques. Despite decades of research and development, estimates of these models are generally recognized to be woefully inaccurate, with orders-of-magnitude errors being routinely reported in the literature [45]. However, these models are much cheaper in resource consumption with respect to memory usage, estimation time (time for estimating the cardinality), and maintenance.

In the remainder of this chapter, first, we briefly describe the traditional statistical-based models such as histogram-based and sampling-based techniques for cardinality estimation. Then, we give a high-level overview of the recent learning-based models, along with highlighting their limitations. Next, we introduce *GridSam* – the Grid-based Dynamic Sampling technique for result cardinality estimation. Further, we showcase the performance evaluation of GridSam at a high level using a widely used real-world single-table dataset named Power [4] and provide the comparisons of GridSam with state-of-the-art learning-based estimators in various aspects. We also showcase the performance evaluation of GridSam regarding accuracy on a widely used real-world multi-table dataset named IMDB [5] and compare it with various state-of-the-art learned-estimators which support multi-table queries.

1.1 Statistical-based Cardinality Estimators

1.1.1 Histogram-based Cardinality Estimators

The histogram-based cardinality estimators can be broadly divided into two types: (1) one-dimensional histograms and (2) n-dimensional histograms. The histogram-based techniques also maintain per-attribute statistical information such as the frequencies of most common values, number of unique values, and the fraction of null values.

For approximating the cardinalities of individual filter predicates in a query, the estimators with one-dimensional histograms assume uniform spread inside the respective histogram buckets. For approximating the combined cardinality of multiple filter predicates, these estimators (with one-dimensional histograms) usually apply erroneous assumptions such as *AVI* (Attribute Value Independence) – assuming no-correlation among the respective attributes, *EBO* (Exponential BackOff), and *MinSel* (Minimum Selectivity) – assuming full-correlation. Further, these estimators assumes uniform distribution across join-crossing correlation to estimate the cardinality for the multi-table queries. These assumptions are known to be highly

inaccurate in real-world datasets such as IMDB dataset [5] proposed in Join-Order Benchmark (JOB) [45].

The cardinality estimators with n-dimensional histograms (with $n \geq 2$) aim to approximate the joint data-distribution by partitioning the data-space using n-dimensional histogram buckets. Several techniques for multi-dimensional histograms have been proposed in the literature (e.g. [30, 55, 51, 20, 22]). There are also several surveys (e.g. [43, 36]) that have been presented in the literature, which also include more advanced techniques. They summarized several essential techniques and pointed out the following significant limitations of multi-dimensional histograms: (i) huge storage cost, and (ii) inability to capture correlations and distributions between attributes across tables. Thus, for multi-table queries, these techniques still have to rely on the highly erroneous assumptions.

1.1.2 Sampling-based Cardinality Estimators

The sampling-based techniques are promising alternatives to the histogram-based techniques. At a high level, sampling-based techniques suggest running a query on a small sampled database. Then, they extrapolate the query result count of the sampled evaluation with respect to the max cardinality possible for the given query. In contrast to heuristics-based assumptions such as *AVI* and *MinSel*, sampling-based techniques (e.g. [48, 54, 25, 26, 46, 63, 59]) can capture arbitrary correlations in the data well. Hence, they can produce more accurate estimates if samples can represent the underlying data-distribution well.

The sampling-based techniques face challenges for the selective predicates in a query. Samples may not contain adequate tuples, or in the worst case, they may not contain any qualifying tuples for the corresponding selective predicate from a query. Because of that, the sampling-based techniques often cause the Zero Sample Problem [46], which is simply getting the zero qualifying tuples in the result after the query's sampled evaluation. The Zero Sample Problem can also arise due to any one or many combinations of the following properties regarding attribute data: (i) large value domain, (ii) significantly skewed data distribution, and (iii) strong correlation among attributes from same or different tables. In the case of Zero Sample Problem, most sampling-based techniques often rely on erroneous assumptions such as *AVI* and *MinSel* similar to histogram-based techniques, which are known to be highly inaccurate, and produce large estimation errors.

1.2 Learning-based Cardinality Estimators

Spurred by the rapid growth of machine-learning technology over the past decade, there has been a recent slew of papers advocating the use of these techniques for cardinality estimations

(e.g. [23, 32, 33, 39, 40, 49, 52, 56, 62, 31]). Most of the proposed approaches use deep learning based-frameworks, modeling the result cardinality estimations as regression problems. Within this corpus, two broad classes have emerged – *query-based* and *data-based*. The former is an example of supervised learning methodology, with models constructed by training on a large set of queries and leveraging the actual cardinalities observed during execution as the labels. On the other hand, the data-based techniques fall under unsupervised learning methodology, and model the joint probability distribution functions of the underlying data to capture distributions and correlations.

These novel techniques certainly hold out the promise of substantively improving estimation accuracies. However, at the same time, they suffer from the common limitations of learning-based approaches as follows:

1. The learned-estimators require significant training effort (i) to acquire actual training labels in case of query-based approaches, and (ii) to perform training and hyper-parameter tuning for both kinds of approaches.

2. The query-based learned-estimators often suffer regarding accuracy because of their limited generalizability to unseen queries, especially for the queries with more complex structures.

3. The query-based learned-estimators require to perform training and hyper-parameter tuning again in case of query workload drifts which might take a significant amount of time.

4. The learned-estimators also suffer regarding accuracy because of their inability to cope with frequent data-updates. As the data-updates may potentially lead to a significant change in data distribution, the old models may not be viable to use. They require significant time and computation to acquire updated training labels and re-tune the model to incorporate updated data. In other words, they demand significant "shutdown" time to regain the estimation accuracy mainly because of the significant retraining time to incorporate new data into the model. This behavior of the learned-estimators has been established in the recent detailed study [60].

1.3 GridSam - Grid-based Dynamic Sampling

As sampling-based techniques can easily capture the correlations between attributes, they mainly suffer from the Zero Sample Problem (unable to find any qualifying tuples from samples). In this work, we take a relook at traditional random sampling and investigate whether its cardinality estimation performance could be appreciably improved in conjunction with *lightweight*

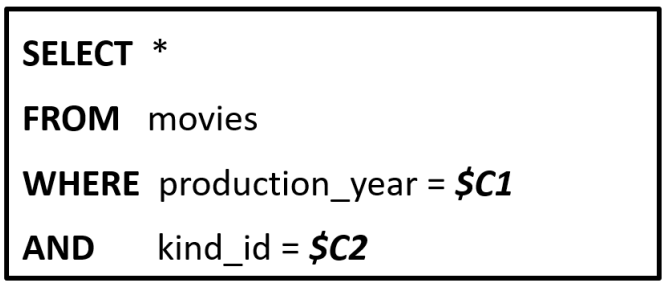


Figure 1.1: Parameterized IMDB Query Template (on *production_year* and *kind_id*)

supplementary data structures that reduce the Zero Sample Problem occurrence. Further, our investigation is in the same spirit as the 2013 study of [61], which showed that with basic calibration and tuning, *traditional cost models* could be surprisingly effective in comparison to the learning-based techniques previously advocated for the same purpose.

Specifically, GridSam augments random sampling with histograms. It partitions the data space on "critical" attributes as the dimensions by imposing *multi-dimensional grid overlay*. Subsequently, it performs random sampling from the *confined* query-specific region of the multi-dimensional grid overlay, leading to reduced occurrence of the Zero Sample Problem. In other words, by combining random sampling and histograms, GridSam tries to improve the quality of the estimates by rectifying the inherent limitations of each other - histogram provides the confined sample space to avoid the Zero Sample Problem, and random sampling provides the ability to capture arbitrary correlation to avoid erroneous heuristic-based assumptions.

1.3.1 Sample Evaluation of GridSam on Query Template

We try to exemplify the above arguments using the query template presented in Figure 1.1 based on IMDB database [5]. The query template contains two equality predicates on attributes *production_year* and *kind_id*. Figure 1.2 shows the result cardinality with respect to all possible values of *kind_id* and *production_year*, which also signifies that both attributes are highly correlated with each other. For example, the number of movies produced in a specific *year* (*production_year*) with a specific *genre* (*kind_id*) majorly depends on the values of both of the attributes (i.e., there can be a huge difference in the number of *animated* movies produced in the *year* 2010 compared to 1960). Thus, it is a pretty difficult estimation task for histogram-based techniques as heuristic-based assumptions such as *AVI* often perform poorly with correlated data. It is also difficult for sampling-based techniques regarding Zero Sample Problem as some combinations of values of both the attributes may not frequently occur in the data because of correlations between values and can often be missed out while sampling.

For a simple case of the concerned query template, GridSam partitions the table *movies* on

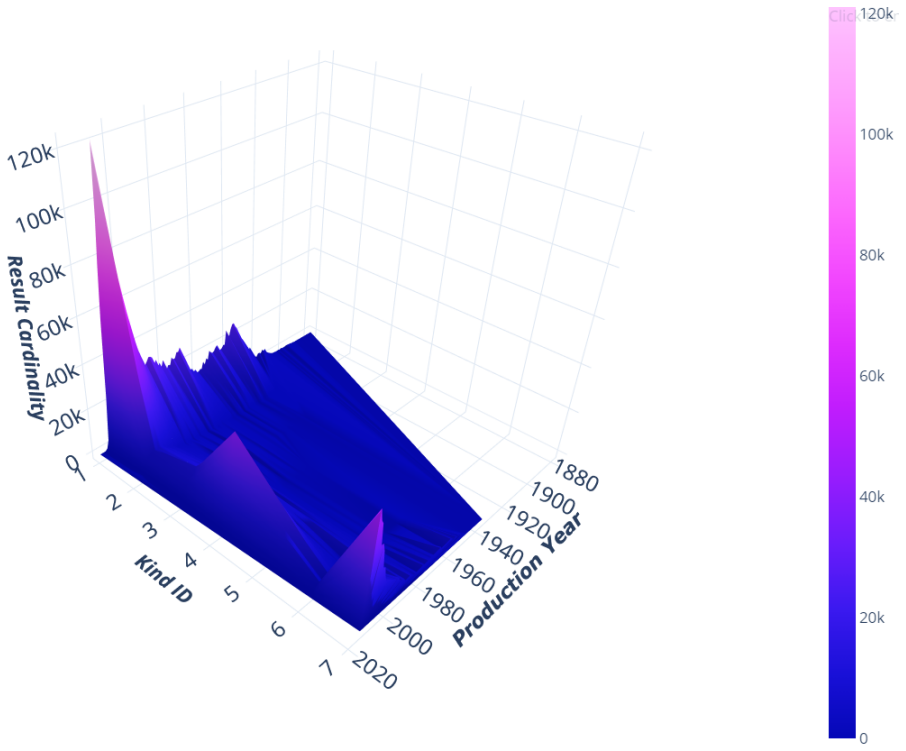


Figure 1.2: Correlation between Attributes *kind_id* and *production_year*

attribute *production_year* with X number of partitions, which is equivalent to having the one-dimensional grid overlay on the table *movies*. We address the design issues of the grid overlay with respect to selecting the attributes as the dimensions and determining the partition boundaries in Chapter 3. After partitioning the table *movies* on attribute *production_year*, GridSam draws dynamic random samples from the appropriate partitions based on the value of predicate constant ($\$C1$) in the filter predicate ($production\_year = \$C1$) from the corresponding query instance.

In Table 1.1, we have shown the accuracy of GridSam with X number of partitions, denoted as (GridSam(X)), over 500 query instances of the query template presented in the Figure 1.1. The query instances are generated by randomly choosing the values of $\$C1$ and $\$C2$ independently from their corresponding data-distributions. The accuracy is shown in the form of *q-error* [50] distribution of 500 query instances, where *q-error* is the widely used metric to denote the cardinality estimation error. Here, the large *q-errors* denotes the large estimation errors. We have formally introduced the *q-error* metric in Chapter 5.

Even with five partitions, GridSam improves accuracy drastically compared to both traditional Histograms and RMS (Random Materialized Samples). Note that, for Histograms in Table 1.1, we used the cardinality estimation module of the Postgres database engine [14]. For

Table 1.1: *Q-Error* Distribution for 500 Query Instances of Query Template Presented in Figure 1.1

Model	median	90th P.	95th P.	99th P.	max	mean	ZSP
Histograms	2.4	6.4	17	22	50	3.9	-
RMS	2.3	7.1	17.5	22	48	4	76%
GridSam(5)	1.2	2.9	3.7	8.8	21	1.8	29%
GridSam(10)	1.1	2.3	3.1	6.2	10	1.5	23%
GridSam(50)	1.0	1.3	1.5	2.0	3.1	1.1	10%
GridSam(100)	1.0	1.1	1.1	1.2	2	1.03	5%

the fair comparisons in the context of memory usage, we configured the Postgres cardinality estimator with the max number of histogram buckets (10,000) and the size of the most common values array (10,000) for both attributes present in the query, which are much higher than the value domain sizes of both attributes *production_year* and *kind_id*. The histogram-based technique still yields large estimation errors, as shown in Table 1.1, mainly because of the *AVI* assumption. Even though RMS (Random Materialized Samples) does not require any heuristic-based assumptions, it could not improve upon the accuracy mainly because of the high Zero Sample Problem rate. The ZSP metric in Table 1.1 denotes the percentage of queries resulted into the Zero Sample Problem for RMS and GridSam. Note that we used the same sample size of 1000 for both RMS and GridSam for evaluation. For a higher memory budget to RMS, we have also evaluated RMS with 10,000 samples. However, it could not achieve better accuracy than GridSam(5) with 1000 samples as the Zero Sample Problem rate remained still higher even with 10,000 samples. Moreover, increasing the sample size further can directly impact the estimation time. GridSam has improved the estimation accuracy mainly because of two following reasons:

1. The confined sample space using the grid-based partitioning for random sampling has helped in avoiding the Zero Sample Problem for the large number queries as depicted using the ZSP metric in Table 1.1.

2. The sampling itself has helped GridSam to capture the correlation between both attributes and not relying upon the heuristic-based assumption such as *AVI*.

By increasing the number of partitions such as 50, GridSam achieves superior accuracy and a much lower Zero Sample Problem rate compared to others by a huge margin. And with 100 partitions which is close to the data-domain (value domain) size of the attribute *production_year*,

GridSam estimated accurately for each query, because GridSam almost mimicked the index-based query evaluation in this case.

1.3.2 General Idea of GridSam

Specifically, we present **GridSam**, an estimation technique that augments random sampling with the following auxiliary data structures:

1. Simple but carefully chosen *persistent multi-dimensional grid overlays*, whose dimensions are constituted from "critical" table attributes, to focus the sampling on the query-relevant regions of the data-space. This restriction improves the estimation quality by reducing the impact of Zero Sample Problem, which is typically encountered when searching for values with low frequencies, the classical bane of sampling. We assume that a memory budget is provided by the user, which determines the permissible size of the grid overlays.

2. *Learned-indexes* [42] to provide cheap probes for capturing join crossing-correlations using the framework of Index-based Join Sampling (IBJS) [46]. These probes improve the time and space efficiency for multi-table queries.

Our interest in sampling stems from its inherent ability to capture correlations within attributes of a table, as well as join-crossing correlations across tables in conjunction with indexes [46]. Further, we use the dynamic variant of sampling, where the samples are taken on the fly, to ensure adequate coverage of the specific query region indicated by the grid overlay. Dynamic sampling had been historically considered expensive since each sample could, in principle, require a separate disk access – however, given that main-memory-resident databases have become commonplace, it appears reasonable to assume that the samples can now be sourced cheaply. Finally, thanks to Kolmogorov's statistics [29], we know that the accuracy of sampling is primarily a function of the *absolute* number of samples taken and not proportional to the database size, thereby seamlessly supporting scaling to large databases.

Building the grid overlay on the table includes the following tasks related to design and construction:

1. Determining a set of "critical" attributes as the grid overlay dimensions.

2. Determining the number of partitions for each dimensional attribute.

3. Determining the partition boundary values for each dimensional attribute.

4. Imposing the grid overlay on the data.

A greedy methodology targeted towards reducing Zero Sample Problem is used to determine the set of "critical" attributes, and the same methodology is used to also determine the number of partitions for each dimension. Specifically, this methodology preferentially chooses attributes having data characteristics such as: (i) large value domains, (ii) significantly skewed data distributions, and (iii) strong correlations with attributes in the same or different tables. We address all these design choices in Chapter 3.

As the overlay construction process is highly parallelizable, the grids can be designed and built in just a few seconds using a GPU. This allows GridSam to achieve much lower setup time in comparison to learned-estimators, thereby facilitating adaptation to dynamic data-updates. Moreover, GridSam also uses the GPU to perform data-parallel tasks on the grid overlay during estimation to decrease the overall estimation time. These data-parallel tasks typically include (i) calculating the size of the sample space of dynamic query-specific region; and (ii) performing random sampling from the non-empty grid cells in a query-specific region. The price of these improvements is substantially increased memory usage since tuple identifiers need to be stored in the corresponding grid cells to maintain their natural order. However, we already mentioned that the main-memory is sufficiently available in the system as it is cheap in the current scenarios. We hold this assumption also for learned-estimators by considering memory budget as the hyper-parameter while tuning the model.

The experimental evaluation shown in Chapter 5 and Chapter 6 suggests that, by trading off memory usage, GridSam achieves competitive estimation accuracy compared to state-of-the-art learned-estimators by incurring orders-of-magnitude less building time, which is significantly effective for data-warehouses with frequent data-updates and query workload drifts.

1.3.2.1 Grid Overlays

Our grid overlay is a static version of the celebrated Grid File indexing structure [53]. Specifically, for each table, we first identify the attributes that are likely to be difficult to estimate well with random sampling over the entire domain. These "critical" attributes, which we call as *e-attr* (set of error-prone attributes), are typically those with (i) large value domains, (ii) significantly skewed data distributions, (iii) strong correlations with sibling attributes in the same table, or (iv) strong correlations with attributes in external tables. Then, we create *linear scales* (the array of partition boundary values) for each attribute from *e-attr*, equivalent to the equi-depth attribute histograms that are already common in databases.

Finally, we create a multi-dimensional grid overlay with each attribute from *e-attr* forming a dimension and the linear scales determining the partitioning. In the last step, tuple identifiers (*TIds*) or pointers are included in each cell of the grid for all rows in the base table that fall

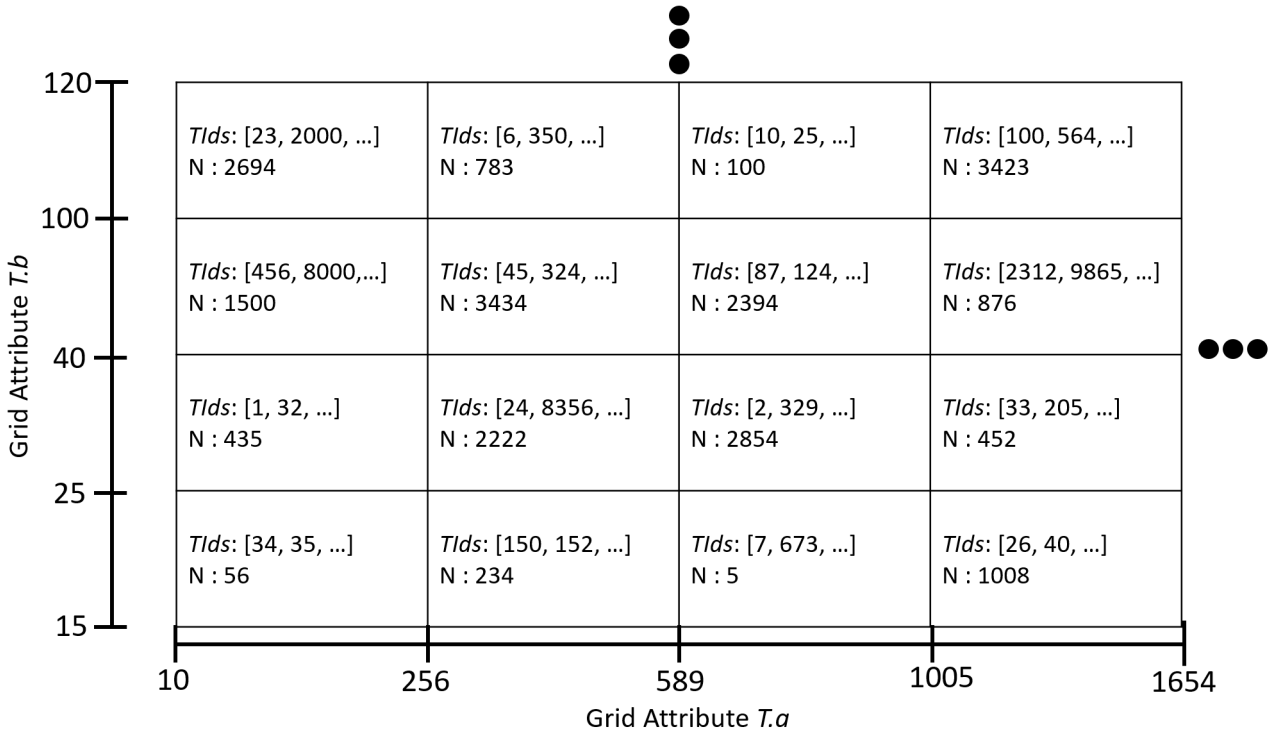


Figure 1.3: Grid Overlay Example

into the grid cell. A sample grid overlay is shown in Figure 1.3 on a hypothetical table T containing two attributes $T.a$ and $T.b$ which are used as the dimensions. It also includes the occupancy of each grid cell as a derived meta-data within the grid cell. For more than two dimensions, the grid overlay will look equivalent to a hyper-rectangle. The multi-dimensional array (contiguously stored in row-major order) is used to represent the grid overlay. Each element of this array stores the pointer to a list of *TIds* and the number of elements in the list (N). This array can be persisted onto the disk as well.

Subsequently, when we carry out sampling for a query with *e-attr* predicates, the sampling is restricted to the minimal bounding hyper-rectangle in the grid that completely contains the query. And the expectation is that this restricted sampling will provide more accurate results as compared to taking the same number of samples over the entire table space.

Obvious issues that arise with the above approach is the exponential space and time complexity of the grid overlay construction, which also impacts the practical number and choice of *e-attr* to include in the overlay, as well as the number of partitions in each dimension – we address these design and implementation issues in Chapter 3.

1.3.2.2 Type of Sampling in GridSam

In general, GridSam performs Simple Random Sampling With Replacement (SRSWR) from the query-specific region of the grid overlay. Specifically, GridSam performs weighted random sampling from all the grid cells within the query-specific region based on the volume of the respective grid cell (more samples from the grid cells with more volume). The sampled *TIds* from all the grid cells are checked for uniqueness for having distinct random samples in the final sample set. Internally, for all the grid cells within the query-specific region, GridSam randomly picks *TIds* from each grid cell and check for uniqueness using the hash-based data structure. For each query, GridSam can efficiently perform *SRSWR* from the query-specific region as big datasets nowadays are main-memory residents. As GridSam acquires dynamic samples for each query at run-time, it does not maintain any *reservoir* of random samples. GridSam applies the simple extrapolation estimator on the result size of the sampled evaluation with respect to the total volume of the query-specific region to estimate the cardinality for a given query. There is no induced bias in the sampling mechanism of GridSam as it just performs random sampling from the confined query sample space and uses the same sample space for extrapolation of the result count.

1.3.2.3 Incorporating Learned-Indexes

An elegant approach to implement index-based sampling for capturing join-crossing correlations was proposed in the IBJS scheme of [46]. However, they considered traditional B-tree and hash-indexes that require tree traversals for the sampling – with a large number of indexes and probes, and this could become expensive with regards to both space and time overheads. Therefore, we instead opt to leverage the recently proposed learned-indexes [42] which provide substantially improved efficiency due to their *constant* probe complexity. A hurdle, however, is that learned-indexes expect the indexed attribute to be physically sorted, and this is especially problematic when indexes have to be built on multiple attributes, as in our situation. We address this issue by the simple workaround of constructing auxiliary mappings that connect the logical sorted order of each attribute with the physical storage locations.

1.3.2.4 Performance Evaluation and Comparisons (Where GridSam Stands)

We have conducted a detailed experimental study of GridSam as shown in Chapter 5 and Chapter 6 on a Postgres platform [14] hosted on a vanilla workstation, using database evaluation environments similar to those considered in recent studies. We evaluated GridSam on different memory budgets and established that it can yield the estimation accuracies that are surprisingly competitive to the learning approaches. Our experiments indicate that GridSam can be built

Table 1.2: Performance Evaluation and Comparisons of Result Cardinality Estimation

Technique	Accuracy (Q-Error)		Est. Time (ms)	Storage (MB)	Building/Training Effort (secs)
	95th P	99th P			
GridSam-50	2.9	8.2	1.8	-	3
GridSam-100	2.75	8.17	3	-	3
GridSam-500	2.45	8.15	9	-	3
GridSam-1024	2.28	8.03	20	-	3
NARU-15	6	21	2.8	0.64	-
NARU-55	4.1	12	5.4	2.1	-
NARU-200	3.2	6.3	5.7	2.2	-
NARU-3100	2.0	4.0	5.7	2.2	-
LWM-360	8	21.6	0.06	0.25	-
MSCN-500	10.5	34	0.5	30	-
Postgres	14	65	0.01	0.13	1
RMS	13.5	61	0.01	0.02	0

efficiently in just a few seconds, enabling it to support frequent data-updates.

We have compared GridSam with state-of-the-art learning-based estimators such as MSCN [40], Light-Weight Models (LWM) [23], and NARU [62], and also compares with traditional estimators such as the cardinality estimation module of the Postgres [14] database engine (histogram-based technique) and RMS (Random Materialized Samples – sampling-based technique). The evaluation metrics were the following: (i) *Accuracy*, measured in terms of the 95th and 99th percentile of *q-error* [50], (ii) *Estimation Time*, (iii) *Storage Overheads*, and (iv) *Building/Training Effort*. For GridSam, storage budget M (in MB) is an input parameter, and it is therefore denoted as GridSam-M. Analogously, the training-time budget T (in secs) is an input parameter to the various learned-estimators, and they are denoted as MODEL-T, where MODEL is the specific learned-estimator. Note that the estimation time and storage overheads are considered as the *run-time efforts* and building/training effort is considered as an *off-line effort* (compile-time effort). Here, for storage metric, we are only considering the memory budget that is required for the estimation/prediction of the query's cardinality. Note that, for all learned-estimators, we have performed training and hyper-parameter tuning as suggested in the respective papers using GPU with various memory budgets (models sizes) and picked the best model for each learned-estimator for evaluation.

The results of a representative experiment featuring 3000 queries on the Power dataset [4] are shown in Table 1.2. In particular, GridSam-M was evaluated for a range of memory scenarios, including $M = 50$, 100, and 1024 MB, and a fixed sample size of 1000. As shown in Table 1.2, GridSam delivers competitive accuracy to the learned-estimators. Moreover, the off-line effort

Table 1.3: *Q-Error* Distribution for JOB-Light Workload

Model	median	90th	95th	99th	max	mean
Postgres [40]	6.1	161	818	2044	2097	127
IBJS	1.8	35.6	270	8167	9654	340
MSCN [40]	3.82	78.4	362	927	1110	57.9
End-to-End [56]	3.51	48.6	139	244	272	24.3
GridSam	**1.1**	**2.64**	**3.48**	**5.04**	**5.41**	**1.63**

for model building is orders-of-magnitude lower. For instance, NARU requires much higher training time than GridSam to achieve comparable accuracy. Moreover, the other learned-estimators could not perform better than GridSam even with large training budgets. Also, while GridSam may appear to have large estimation times in a relative sense, these times are quite practical from an absolute perspective since they are in milliseconds, minuscule in comparison to query execution times. On the flip side, GridSam requires significantly greater run-time memory to provide these benefits, thereby establishing a space-time tradeoff. Finally, turning our attention to the traditional techniques, we find they are highly inaccurate in comparison to GridSam, and therefore unviable for high-confidence estimations.

A sample instance of the performance evaluation for multi-table queries (join queries) is captured in Table 1.3 which compares GridSam with the contemporary learning techniques on the widely used JOB-light workload [40], which is based on IMDB dataset [5]. Table 1.3 shows the *q-error* distribution of GridSam against the reported numbers of MSCN [40] and End-to-End [56]. Here, IBJS (Index-based Join Sampling) [46] is considered as a baseline of the sampling-based techniques for multi-table queries. The performance of the cardinality estimation module of the Postgres database engine is also included as a baseline for histogram-based techniques. The learned-estimators such as NARU and LWM do not support multi-table queries, and therefore, they are excluded from this evaluation. We see in this table that over the entire distributional range, GridSam provides significantly lower estimation errors than the prior art.

For fair comparisons on memory budget, we have configured the cardinality estimation module of the Postgres database engine with max number of histogram buckets and the max size of the most common values array. We have also evaluated RMS and IBJS with a large sample size such as 10,000. Even with a large number of samples, they could not achieve better accuracy than GridSam, mainly because the Zero Sample Problem rate remained still higher even with 10,000 samples. Moreover, increasing the sample size further can directly impact the estimation time.

1.4 Organization

The thesis is organized as follows: The detailed description of related work is presented in Chapter 2 which covers both statistical-based and learning-based techniques. In Chapter 3, we describe the design and construction schemes for the grid overlay(s) for both single-table and multi-table datasets. The building of learned-indexes for the purpose of propagating sampling across joined tables is also outlined in Chapter 3. The use of these data structures to support cardinality estimation for the queries containing single table and multiple tables is presented in Chapter 4. The experimental framework, including grid construction selection parameters, grid construction building parameters, datasets, and comparative estimators, is presented in Chapter 5. A detailed evaluation over representative real-world database environments is presented in Chapter 6, highlighting the accuracy and efficiency of the new GridSam approach. Finally, we present our conclusions and outline future research avenues in Chapter 7.

Chapter 2

Related Work

2.1 Histogram-based Techniques

2.1.1 Cardinality Estimation using One-Dimensional Histograms

The cardinality estimators in most current industrial database engines maintain per-attribute statistics such as one-dimensional histogram, frequencies of the most common values, number of unique values, the fraction of null values, etc. Based on these statistics, the estimator determines the approximate cardinality of the individual single-attribute filter predicates of the query. Note that the cardinality of the filter predicate can also be expressed as the *selectivity* which is a normalized form of the cardinality with respect to the max cardinality possible. For determining the combined approximated cardinality of the multiple filter predicates in conjunction, the default method is based on a heuristic assumption such as Attribute Value Independence (AVI) [15, 17, 11]. For the AVI assumption, combined selectivity is being calculated by multiplying the selectivities of the filter predicates of a query, which essentially assumes no-correlation between attributes. Another heuristic-based assumption could be the Minimum Selectivity (MinSel) which essentially assumes full-correlation between attributes of a table. MinSel assumption determines the combined selectivity of the multiple filter predicates in conjunction by choosing the minimum selectivity from the selectivities of all filter predicates.

For the multi-table queries containing join predicates, the estimators often rely on the Join Predicate Independence assumption [57], also known as the Join Uniformity assumption. Here, the estimator assumes the uniform distribution in join crossing-correlation. In simple words, for PK-FK join predicates, the uniformity suggests that every PK value joins with the same number of FK values.

Usually, these assumptions which are used by the estimators do not hold in the real-world

datasets, such as the IMDB dataset [5] from Join-Order Benchmark (JOB) suggested in [45], and the estimators often produce significant estimation errors which is evidently highlighted in [45]. However, the histogram-based estimators are cheap in resource consumption in terms of storing, creating and maintaining per-attribute statistics.

2.1.2 Cardinality Estimation using n-Dimensional Histograms

A natural extension to the one-dimensional histograms is n-dimensional histograms for cardinality estimation. They aim to approximate the joint data-distribution to capture the intra-table correlations by partitioning the data-space using n-dimensional histogram buckets corresponding to the respective attributes. Several techniques based on various data-space partitioning schemes have been proposed for the multi-dimensional histograms in the literature (e.g. [30, 55, 51, 20, 22]). There are several detailed studies (e.g. [43, 36]) have been presented which summarize various recent strategies based on the following:

1. Partitioning methodologies [47, 24] to divide the data into multiple n-dimensional buckets in order to achieve better approximation quality

2. Workload-aware methodologies [20] for partitioning the data-space based on query feedback

3. Efficient and fast construction of the histograms based on various techniques like sampling

These studies also summarize the following major challenges for multi-dimensional histograms to be used for cardinality estimation:

1. There is a problem of huge space overheads with n-dimensional histograms. The storage cost increases drastically with an increase in the number of dimensional attributes (n).

2. In general, multi-dimensional histograms are unable to efficiently capture the correlations and distributions between attributes from different tables. They still have to rely on the assumption of Join Uniformity for join-crossing correlations, which are known to be highly erroneous [45].

Moreover, similar to the estimation methodologies of one-dimensional histograms, the multi-dimensional histograms still have to rely on the erroneous assumptions such as *AVI* (no-correlation) and *MinSel* (full-correlation) for the queries containing filter predicates on the attributes which are not the dimensions of the respective multi-dimensional histogram.

GridSam overcomes these challenges by combining multi-dimensional grid overlay and dynamic sampling. In a way, GridSam also creates the multi-dimensional histogram by imposing the grid overlay to partition the data-space and storing tuple identifiers (*TIds*) into grid cells. To achieve fast building time, GridSam employs a simple static partitioning scheme where partition boundaries are determined before partitioning the data-space. It is similar to the static version of the grid files [53]. The empirical evaluation shows that even a simple static design of GridSam along with query-specific sampling can achieve reasonably good estimation accuracy. In GridSam, the number of attributes as dimensions are predetermined based on the underlying data properties. To achieve better accuracy in the limited space budget, GridSam employs the greedy methodology targeted towards reducing the Zero Sample Problem rate to choose the "critical" attributes as dimensions for partitioning the data-space. Further, because of the integration of dynamic sampling with a multi-dimensional grid overlay, GridSam can also capture the correlations between attributes that are not part of the dimensional attributes. To capture join-crossing correlations, GridSam follows the index-based framework for correlated sampling suggested in [46] along with grid overlays to limit the Zero Sample Problem rate. GridSam also incorporates learned-indexes [42] in the IBJS framework to efficiently performing the index-probes for join sampling.

2.2 Sampling-based Techniques

Histogram-based techniques face many inherent challenges to capture the correlations and distributions for both intra-table and inter-table (join crossing-correlations) for cardinality estimation. Sampling-based techniques are promising alternatives to histogram-based techniques because of their inherent ability to capture the correlations. If sampled rows (tuples) represent the underlying data-distribution well, then the sampling-based techniques can accurately approximate the cardinalities of the queries. With this motivation, various types of sampling-based methodologies based on independent sampling to correlated sampling have been proposed in the literature (e.g. [48, 25, 26, 46, 63, 59]) to capture both intra-table and inter-table correlations.

However, the sampling-based techniques often cause the Zero Sample Problem [46], which is getting the zero qualifying tuples in the result after the query's sampled evaluation. Most of the sampling-based techniques fail when there are no qualifying samples to extrapolate the result count. In that case, they have to fall back to erroneous assumptions such as *AVI*, and *MinSel*, which are known for producing highly erroneous cardinality estimates.

In the case of multi-table queries (queries containing join predicates), the *naive* independent sampling approach is to randomly sample the tuples (rows) from the individual tables present in the query and then join those samples after applying the respective base-table filter

predicates present in the query. The major problem with this naive approach of independent sampling is that there is a huge number of samples required in expectation, even after assuming uniform distribution over join crossing-correlation, from each table to get the "enough" number of samples in the result [46] and to avoid Zero Sample Problem. The estimation time of the naive sampling approach can be significantly high because of the massive amount of necessary sampling. To solve the limitations of independent sampling for multi-table queries, many correlated sampling techniques have been proposed in the literature (e.g. [46, 63, 59]), which mainly use the pre-computed statistics with auxiliary data structures to obtain correlated samples. We now describe a couple of recent correlated sampling techniques which aim to avoid the empty query result on sampled evaluation with small sample size for multi-table queries (join queries). We also highlight their limitations with respect to the Zero Sample Problem and frequent data-updates.

The correlated sampling with the CS2 algorithm has been proposed in [63] for multi-table queries. This technique precomputes the correlated samples from the tables in the schema and materializes them to achieve run-time efficiency. Each query uses these samples to estimate the cardinality after applying the respective base table predicates. This technique needs a source table to obtain the initial samples, and then it chooses another table and the corresponding samples from it, which can be joined with the samples from the source table. The CS2 technique follows this procedure for the entire schema join-graph to obtain correlated samples. It aims to limit the chances of Zero Sample Problem, which often occurs with small sample sizes and many tables in the query, by precomputing and materializing correlated samples. As the CS2 technique can be performed efficiently for star database schema or snowflake database schema, it may not be efficiently applicable to any arbitrary schema graph. Moreover, the materialized correlated samples must be reacquired in case of updates in the data.

The recent correlated sampling approaches are summarized in the recently presented detailed study [43] which pointed out several major limitations of the recent sampling-based techniques as follows:

1. Materialized random samples (independent or correlated) need to be updated frequently in case of frequent data-updates, which can be very costly, especially in the case of correlated sampling for multi-table queries.

2. In case of multi-table queries, acquiring adequate correlated dynamic samples on the fly at run-time might be time-consuming.

However, the state-of-the-art sampling-based technique called Index-based Join Sampling (IBJS) [46] from the family of correlated sampling techniques has tried to rectify the above-

mentioned issues. We have given a detailed description of the IBJS technique in the following subsection.

2.2.1 Index-based Join Sampling (IBJS)

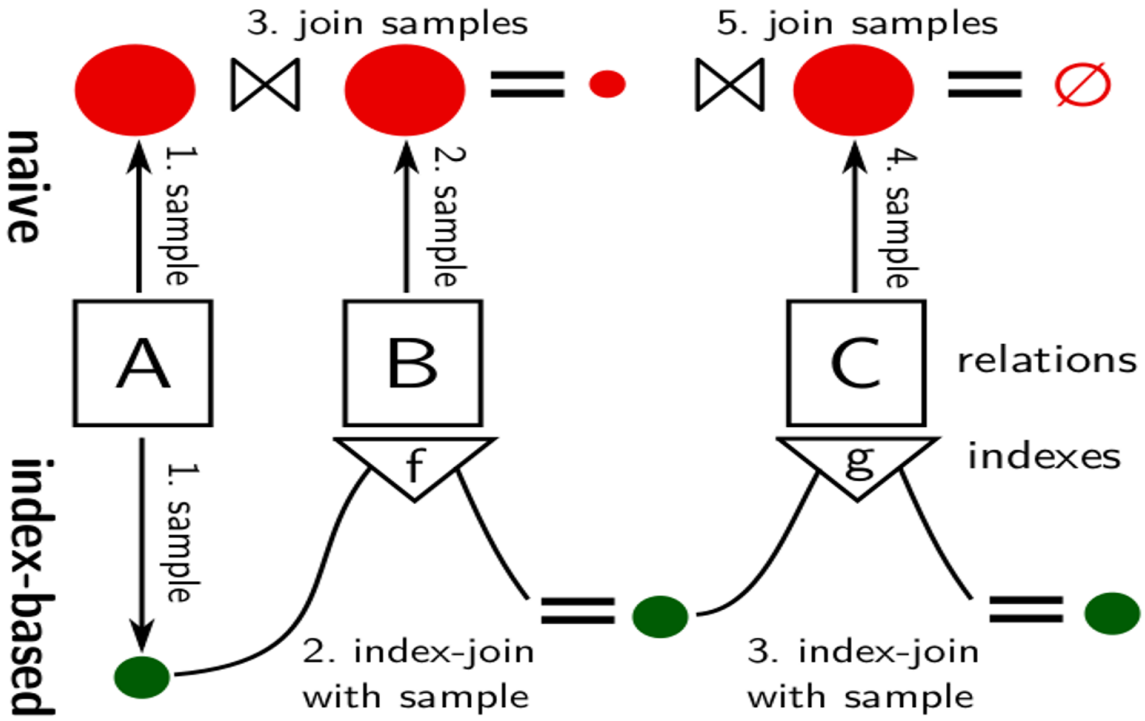


Figure 2.1: Index-based Join Sampling [46]

IBJS [46] addresses the limitations of naive independent sampling approach by probing the qualifying base table samples against the existing index structures to get the correlated samples. Further, IBJS efficiently supports dynamic samples on a per-query basis with the help of index structures built on join attributes, which makes IBJS more robust in the case of data-updates. Note that, the relevant indexes need to updated in case of data-updates. Unlike CS2 [63], IBJS efficiently supports arbitrary schema graphs for obtaining correlated join samples.

For example, as shown in Figure 2.1, consider a hypothetical query containing tables A, B and C with their corresponding filter predicates and join predicates. At the first step, after applying the filter predicates of a table A on initial samples from table A itself, this technique probes the index of the join-attribute of table B for each qualified sample from the initial table and get the sampled rows for the table B. Subsequently, the filter predicates of table B can be applied to these newly acquired join-samples of table B. After that, the same join-sampling

methodology can be repeated for table C. This technique assumes to have indexes built on all join-attributes for the significant improvement in the estimates. This technique requires a much less number of samples than the naive independent sampling approach as it directly gets adequate samples for each table using the corresponding index structures. Hence, it is much cheaper relative to the naive independent sampling approach for multi-table queries.

However, like other independent or correlated sampling-based techniques, IBJS fails when there are no qualifying samples from the initial table to start (Zero Sample Problem) after applying respective base table filter predicates. IBJS may also fail anywhere on the join path as the Zero Sample Problem can occur after applying the respective filter predicates on join-samples of any table on the join path.

GridSam tries to solve the earlier limitations of correlated sampling techniques by using the same index-based framework suggested in [46], to get the dynamic correlated samples on the fly for any multi-table query with equi-joins. Further, GridSam tries to limit the rate of Zero Sample Problem in IBJS by combining the grid overlays with the index-based framework of IBJS, for which we have provided a detailed description in Chapter 3. GridSam also incorporates learned-indexes for efficient join sampling in the IBJS framework to keep the estimation time lower. Moreover, the empirical evaluation presented for multi-table queries in Chapter 6 shows that GridSam can achieve much better accuracy compared to IBJS due to decreased Zero Sample Problem rate.

2.3 Learning-based Techniques

A lot of models and techniques have been proposed in the literature (e.g. [23, 32, 33, 39, 40, 49, 52, 56, 62, 31]), which apply state-of-the-art learning-based models using machine learning and deep learning techniques for cardinality estimation problem. These techniques generally employ advanced learning-based models such as deep autoregressive models, deep neural networks, convolutional neural networks, and gradient boosted trees for cardinality estimation purposes. All the models and techniques can be majorly classified into two categories: (i) supervised-learning models (*query-based*), (ii) unsupervised-learning models (*data-based*), which are described in the subsequent subsections with detailed examples of learned-estimators.

The supervised-learning models generally use regression-based techniques by forming the cardinality estimation as a regression problem. The models try to form a mapping/relationship between queries and their corresponding true cardinality labels in layman's terms. They try to learn the underlying data-distribution using a large number of query instances with respect to corresponding query templates while training and use their true cardinality labels to

learn. The unsupervised-learning models generally try to approximate the underlying joint data-distribution to estimate the cardinality, essentially by learning joint probability distribution function of the underlying data. Unsupervised-learning models only require data for the training without any assumptions on query workload. Thus, they are more robust to the query workload drifts compared to supervised-learning models. We have pointed out several common limitations of the learned-models for cardinality estimation in Chapter 1. Here, we provide the general overview of both supervised and unsupervised techniques for cardinality estimation with detailed descriptions of some of the essential learned-estimators.

2.3.1 Supervised-Learning Models

The generic workflow of all the supervised-learning based models follow a similar methodology to train the models and predict/estimate the cardinality of the query as follows:

1. A large number of queries are generated synthetically for training and testing by randomly choosing tables, filter predicates, and corresponding predicate constants.

2. Each query from the training and testing set is encoded and translated into the adequate input features for the respective learned-model. Along with query information, the additional enriching statistical-based information can also be added in the input features such as heuristic estimators [23] and bitmap of materialized samples [40] to improve the quality of the estimates.

3. The model is trained using the provided input features of all queries of the training set and their corresponding true cardinality labels.

4. For prediction, again, the query is encoded, and translated into the input features with additional information similar to training queries. Then, the estimate of the respective query can be determined as a result after applying the optional post-processing on the model output.

5. In case of frequent data-updates, newly updated true cardinality labels for all the queries of training set along with the additional statistical-based input features need to be reacquired. Further, the regression models need to be updated either using full retraining and re-tuning or incremental training.

Many supervised-learning models (e.g. [23, 56, 40, 52, 31]) have been proposed in the literature for cardinality estimation. We have reviewed a couple of essential techniques in detail that follow the same methodologies of supervised-learning models for cardinality estimation.

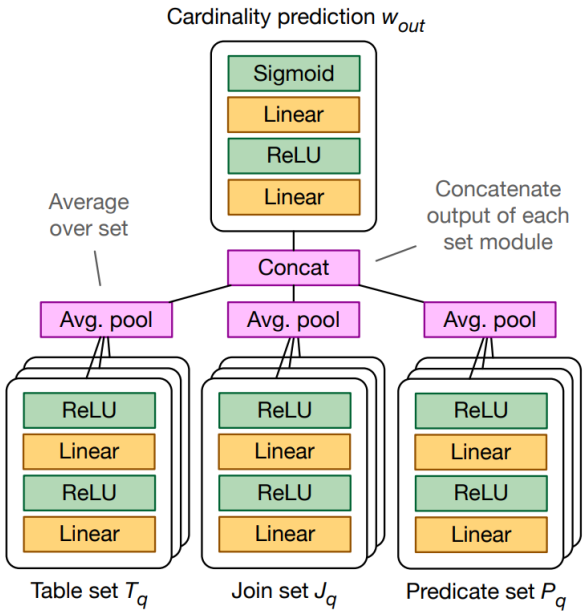


Figure 2.2: Architecture of the Multi-Set Convolutional Network (MSCN) [40]

2.3.1.1 Multi-Set Convolutional Network (MSCN)

This technique [40] provides an estimation model for multi-table queries using a set-based query representation scheme hosted on a multi-set convolutional neural network. The query is partitioned as a set of tables followed by a set of join predicates and finally followed by a set of filter predicates. These sets contain one-hot vectors to identify the respective tables, the attributes in join predicates and filter predicates, and the operators for the respective filter predicates. Along with one-hot vectors, these sets also contain the normalized values of the predicates constants $\in$ [0, 1], normalized using the minimum and maximum values of the respective attribute corresponding to the respective filter predicate. As shown in Figure 2.2, the architecture of the multi-set convolutional network contains tables, joins, and predicates which are represented as separate modules, comprised of one two-layer neural network per set element with shared parameters. Module outputs are averaged, concatenated, and fed into a final output network.

The model is trying to learn the potential underlying distribution of the data using the actual cardinalities of the query instances concerning randomly generated and equally distributed query templates. Thus, this paper [40] has proposed to train the model with a large number of synthetically generated queries (100,000 queries for 6-tables schema of the IMDB dataset [5]). Note that the training query set only contains the queries with the number of joins up to 2, and the testing query set may contain a higher number of joins in a query. The

model is expected to generalize for the higher number of joins. Moreover, they have enriched the training and testing query set with the bitmaps of materialized base-table samples. The corresponding filter predicates of all tables in each query from the training and testing query set are evaluated on 1000 randomly materialized samples, and their positions are being captured using the bitmaps. To capture the correlations and distributions of the underlying data in more detail, these bitmaps are used as the input features for all the tables along with other query-related information for all the queries while training and testing the model.

The MSCN model achieves significantly better accuracy compared to traditional statistical-based techniques for the queries containing up to 2 joins (queries seen in training). However, it fails to generalize for the queries with a higher number of joins and produces large estimation errors for the queries containing 3 to 5 number joins. Moreover, this technique suffers from huge training overheads as there are heavy-duty tasks to be performed, such as (i) acquiring training labels (true cardinalities) for a huge number of queries and (ii) hyper-parameter tuning.

This technique is the first one (best to our knowledge) to provide the singleton framework for summarizing the information from multi-table queries into one neural-network based model. Because of this property, the estimation time (inference/prediction time) of the MSCN model is competitive to the traditional statistical-based estimators. At a high level, MSCN yields much less run-time efforts (estimation time and storage) while incurring huge compile-time efforts (model training). Thus, this technique fails to achieve good accuracy in dynamic environments with frequent data-updates because the model needs a significant amount of time to incorporate the updated data. This behavior is also evident in the recent empirical study [60].

In contrast, GridSam is more like unsupervised-learning methodology as it does not employ any assumptions regarding query workloads to build the grid overlays. It designs and constructs the required data structures for the estimation from the data itself. GridSam tries to approximate the correlations between attributes and underlying data-distribution using dynamic sampling with simple extrapolation from the confined query-specific region of the grid overlay to limit the Zero Sample Problem. Because of a highly parallelizable design of GridSam, it achieves a competitive accuracy even with orders-of-magnitude less building time. We compare GridSam with MSCN in terms of both accuracy and estimation time in the experiments and evaluations shown in Chapter 6.

2.3.1.2 Lightweight NN-based Model (LWM)

To estimate the cardinalities of the queries containing multi-dimensional range predicates from a single table, this paper has [23] proposed the lightweight regression models such as neural networks and tree-based ensembles for faster prediction and lower memory usage. The exam-

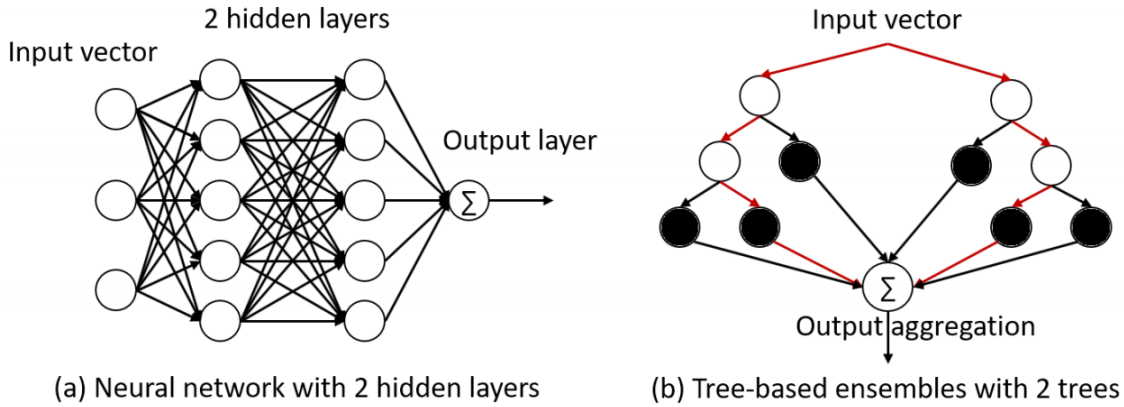


Figure 2.3: Example of the Architectures of the Lightweight Models [23]

ple lightweight models are shown in Figure 2.3. These models follow the supervised-learning methodology – learning from the properties of the queries in the training set and their corresponding true cardinality labels. They try to learn the "complex" non-linear cardinality functions of different query templates using their respective query instances and true cardinality labels that are seen while training. In the context of this paper, the term *lightweight* is with respect to run-time efforts (estimation time and storage).

To effectively handle the large differences in the cardinalities across different query templates, this technique has proposed to apply the log-transformation to the true cardinality labels during the training of the model. Further, this technique has also proposed to include the values of lightweight heuristic estimators (easily and efficiently computable) such as *AVI* (Attribute Value Independence), *EBO* (Exponential BackOff) and *MinSel* (Minimum Selectivity) into the input features of each query along with its identification properties (i.e., attributes and associated predicate constants). These heuristic estimators help the models differentiate between queries with significant cardinality difference but similar range predicates.

These models only support single-table range predicates in conjunction. They achieve significantly better accuracy compared to traditional statistical-based estimators. While achieving so, they incur very small run-time efforts (estimation time and storage) because of the small-sized models, even without GPU. However, they suffer from the common limitations of the supervised-learning techniques regarding huge training overheads similar to MSCN. Same as MSCN, they also suffer from the query workload drift as they need to fully update the model. The recent study [60] has shown that the lightweight models incur significant estimation errors in data environments with frequent data-updates. In this work, we have compared GridSam

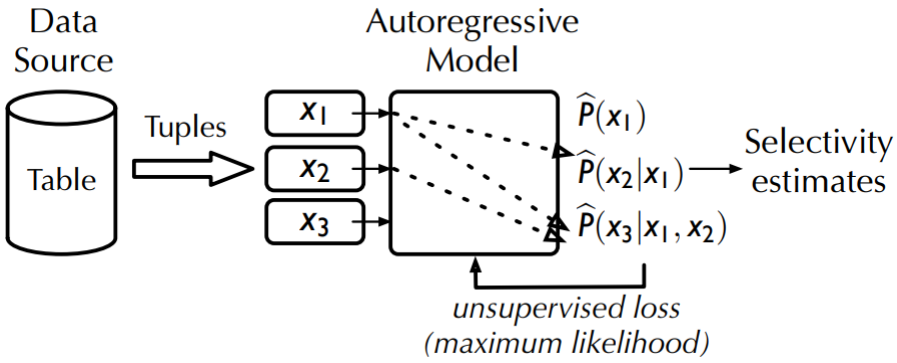


Figure 2.4: Overview of the NARU Estimator Framework [62]

with NN-based (neural-network based) lightweight model (LWM) for experimental evaluations and comparisons.

2.3.2 Unsupervised-Learning Models

Several unsupervised-learning models (e.g. [62, 31, 35]) have been proposed for cardinality estimation. In general, while training, they read the tuples from the table and try to construct the approximate joint data-distribution mainly using the deep learning models such as deep autoregressive model. Then, while predicting, the estimation for the query can be acquired by probing the corresponding model single or multiple times. The model needs to be retrained and re-tuned to capture the updated joint data-distribution in case of data-updates. We have reviewed the unsupervised technique called NARU [62] in detail in the subsequent subsection.

2.3.2.1 Neural Relation Understanding (NARU)

NARU [62] summarizes the data-distribution and the correlations of a table in an unsupervised manner by providing a neural cardinality estimator for the single-table scenarios. NARU approximates the joint data-distribution in its full form without any attribute value independence assumption. It leverages the conditional probability distribution function, learned using the deep autoregressive models (e.g. [58]) suggested in the literature. NARU is trained using the maximum likelihood principle as an optimization function to learn the full conditional probability distribution of the data. Figure 2.4 shows the general framework of NARU estimator, which reads the batch of random tuples in each epoch and factorize the joint distribution in the form of conditional distribution to train the autoregressive model.

NARU inherently provides the ability to estimate the cardinalities of equality predicates on numeric and categorical attributes in conjunction directly using the deep autoregressive models. To achieve good accuracy for high-dimensional range predicates efficiently, NARU introduced

the new Monte-Carlo integration technique called *progressive sampling*, which navigates the sampling to high probability density regions. An important point to note is, analogous to the progressive sampling of NARU, GridSam performs random sampling from the query-specific region (high probability density region) of the grid overlay. Because of NARU's unsupervised data-driven nature of learning, it can support a large set of queries compared to supervised-learning models, and it is inherently amenable to query workload drifts.

NARU yields a much superior accuracy than statistical-based estimators and other recent learned-estimators for the static single-table dataset scenarios. NARU incurs higher estimation time compared to industrial statistical-based techniques but practical in data-warehouse environments. However, even with GPU-based training, NARU incurs huge training overheads to achieve significantly better accuracy. It is evident from the recent study [60] that, in data environments with frequent updates, NARU incurs large estimation errors because of the significant training overheads to incorporate new data into the model. In our experimental evaluation, we compare GridSam with NARU in various aspects and show that GridSam can achieve competitive accuracy compared to NARU even with orders-of-magnitude less building time. We also showed in Chapter 1 that NARU with lower training efforts incurs large estimation errors compared to GridSam.

Chapter 3

Grid Overlay Design Parameters and Construction

This chapter discusses the design parameters selection and construction of grid overlays on *relational* tables, with the end objective of enabling focused sampling. The random sampling from the confined query-specific region can reduce Zero Sample Problem rate and, therefore, can improve the estimation quality. It is verified from the empirical evaluation shown in Chapter 6 that decreasing the Zero Sample Problem rate improves the estimation quality significantly.

While constructing the grid overlay, an exponentially large number of possible combinations concerning design parameters exist – (*Power Set* of *e-attr* $\times$ number of partitions per dimension from the respective set of attributes). We propose a greedy methodology targeted towards reducing the Zero Sample Problem to decrease the number of possible combinations to consider while designing the grid overlay. It also determines the number of partitions for each dimension and provides the "effective" choice of design parameters for constructing the grid overlay that decreases the Zero Sample Problem rate. We assume that there is a memory budget M within which the overlay should be accommodated. As of now, we are only considering the singe-table dataset scenario. Later in this chapter, we introduce the additional design parameters for the dataset with multiple tables. We propose the following design steps that need to be considered to construct the grid overlay:

1. Determine the ranking of the error-prone attributes with respect to each attribute's vulnerability to Zero Sample Problem. This gives the priority ordering for adding attributes in the grid overlay as dimensions to reduce the overall Zero Sample Problem rate.

2. Determine the number of attributes (top k attributes, where k ranges from 1 to total number error-prone attributes in the table) in the previously defined order which should

be used as the dimensions of the grid overlay.

3. Determine the number of partitions for each grid-attribute (attribute chosen as a dimension in the grid) within a given memory budget M.

4. Finally, create the linear scale (an array of boundary values or partitions boundaries) for each grid-attribute based on its corresponding number of partitions.

The first two from the design steps mentioned above are responsible for selecting the set of grid-attributes, which we refer to as *g-attr*. The last two steps are responsible for determining the array of partition boundary values for each grid-attribute with its size. In this chapter, we explore these design issues in detail. Moreover, the parallelism offered by the GPUs can be leveraged to efficiently perform above-mentioned steps. We used the GPU to efficiently perform the data-parallel tasks in each of the steps mentioned above. The experimental evaluation suggests that, with the GPU's support, the overall time taken on determining design parameters and constructing the grid overlay can be much lower (a few seconds) compared to the total training and tuning time of learned-estimators (several minutes to hours).

3.1 Ranking of the Error-Prone Attributes

Our first question is the determination of the ranking order of *e-attr* (set of error-prone attributes), the set of all table-attributes in the order of most vulnerability to the Zero Sample Problem. Here, we assume that *e-attr* is a set of all attributes from the concerned table. We introduce the metric ZSP-score for each attribute from *e-attr*, which quantifies the magnitude of the Zero Sample Problem with respect to that attribute. In simple words, ZSP-score of an attribute denotes the vulnerability to cause the Zero Sample Problem for a query by the respective attribute. We use the following methodology based on attribute data to determine the ZSP-score.

3.1.1 ZSP-score of an Attribute

The idea is to assign a higher ZSP-score to the attribute which has more likelihood of causing the Zero Sample Problem. We define a metric ZSP-score as the percentage of queries from a synthetically generated query workload that resulted in Zero Sample Problem on the sampled evaluation. To determine the ZSP-score for each attribute in *e-attr*, we synthetically generate n queries containing a single equality (or narrow-range) predicate on the respective attribute. The predicate constant will be randomly chosen from the attribute data (to represent distributional skew of the attribute) or data-domain (uniformly between a minimum and a maximum value of the attribute), equally. Then, we run this query set (*ZSP-testing query set*) on a single

random materialized uniform samples of size S from the respective table. Then, ZSP-score of an attribute is simply the percentage of queries that have an empty output.

We determine the ZSP-scores of all attributes considering that all attributes are equally likely to appear in the final query set for which the estimation needs to be performed. Note that, in the ZSP-testing query set, the predicate constants for all queries have followed the data distribution and the uniform distribution in an equal proportion. Thus, the ZSP-score of an attribute will represent the complexity in data distributional skew and data-domain size (i.e., this procedure probably assigns relatively higher ZSP-scores to more "complex" attributes with properties such as relatively large data-domain and significant distributional skew in the attribute data).

The decreasing order of the ZSP-scores determines the ranking order for the attributes. The ZSP-score of each attribute will also be used to determine the number of partitions required for the respective attribute, as described in Section 3.3.

3.1.2 Integrating Real-World Query Workload with ZSP-scores

Suppose a representative query workload is available from the users. In that case, we augment our ZSP-testing query set with all equality (or narrow-range) predicates on the respective attribute under consideration from the given real-world query workload. This helps capture the distribution and frequency skew of the query predicates (predicate occurrence patterns) in the real-world query workload. We have augmented the ZSP-testing query sets of all attributes with the synthetically generated testing query workloads for the experiments. These testing query workloads are the same as the workloads which we used for the evaluation in Chapter 6.

Integrating the knowledge of query predicates distribution of the representative query workload to assign the ZSP-score can be quite helpful in the example cases mentioned as follows:

1. Some attributes may rarely appear in the representative query workload. Thus, even if the attributes have "complex" data properties as mentioned earlier, they may get lower ZSP-scores compared to frequently occurring less "complex" attributes.

2. Some attributes rarely or may not be associated with equality (or narrow-range) predicate. Thus, they may not cause the Zero Sample Problem frequently even if they are frequently occurring in the real-world query workload.

3. The combination of some set of attributes may appear frequently or rarely.

3.1.3 Exploiting Parallelism to Calculate ZSP-scores

This task of determining ZSP-score of an attribute can be easily and efficiently performed on a GPU for each attribute from *e-attr*, as sampled values and ZSP-testing query sets of all attributes are independent of each other. For each attribute from *e-attr*, after copying the sampled values of the corresponding attribute to the GPU-memory, each query from ZSP-testing query set of a corresponding attribute has assigned to one core of a GPU. The corresponding GPU-core evaluates the assigned query's predicate on sampled values and determines whether the query resulted into the Zero Sample Problem or not. If a query resulted into the Zero Sample Problem, the corresponding GPU-thread increments the *atomic counter*, which denotes the final ZSP-score of the corresponding attribute. The number of queries in ZSP-testing query set can be very large as current state-of-the-art GPUs can easily have thousands of cores.

3.1.4 Capturing Correlations into ZSP-scores

The correlation between the combination of attributes may also cause the Zero Sample Problem. To determine the highly correlated attributes that could lead to Zero Sample Problem, we compute the ZSP-scores for all attribute pairs. The ZSP-testing query set of each attribute pair contains n queries with two individual equality (or narrow-range) predicates on each of the respective attributes. For simplicity, we restrict ourselves with only pairs of attributes for capturing the correlation into ZSP-score. Similar to the procedure described earlier, the predicate constants of both predicates in the ZSP-testing query set of each attribute pair have been randomly chosen from the data, the data-domain, and the query workload if available from the users, equally. Then, we run these queries on a single materialized random uniform samples of size S to determine the percentage of queries with empty output – this number constitutes the ZSP-score for the corresponding attribute pair. The ZSP-score of an attribute pair quantifies the vulnerability to the Zero Sample Problem because of the correlation between the respective attributes. This task of determining ZSP-score of an attribute pair can be easily and efficiently performed on a GPU in the same way as explained earlier for the single attribute case.

3.1.5 Example of ZSP-scores on Real-World Dataset

Following are the ZSP-scores of all seven individual attributes from the Power dataset [4] in the decreasing order: *attr1*: 84, *attr3*: 83, *attr5*: 36, *attr2*: 34, *attr6*: 33, *attr4*: 23, *attr7*: 12. Here, *attr1* and *attr3* have large ZSP-scores mainly because of their relatively large data-domain sizes (2837 and 4186, respectively). Despite *attr5* having a small data-domain size of 88, it has the third-highest ZSP-score mainly because of the skewed distribution in the values. Here, for

simplicity, we are not showing the ZSP-scores of all attribute pairs. Note that these ZSP-scores represent the relative vulnerability to the Zero Sample Problem between all attributes. For example, *attr5* may cause the Zero Sample Problem much less frequently compared to *attr1* and *attr3*. Both *attr3* and *attr1* are equally vulnerable to the Zero Sample Problem. Thus, more partitions should be assigned to *attr1* and *attr3* compared to *attr5* to decrease the overall Zero Sample Problem rate of the query workload as highlighted in Section 3.3.

We also calculated the variance of the ZSP-scores for all seven attributes for various sizes of ZSP-testing query set (n) ranging from 1000 to 50,000. The maximum variance among all ZSP-scores of all seven attributes is 0.73. So, the empirical evaluation shows that the ZSP-scores are accurate within the slight variance of 0.73, which is acceptable. For all seven attributes of the Power dataset, the overall time for determining the ZSP-scores of all attributes and all attribute pairs is only around tens-of-milliseconds using the GPU.

The ZSP-score of each attribute from *e-attr* determines the order of priority in which the attributes should be added to the grid overlay.

3.2 Selecting the Grid-Attributes

Given a memory budget M, two important parameters remain to decide as follows for constructing the grid overlay once the ranking of *e-attr* has been decided:

1. The number of attributes in *g-attr* (the subset of *e-attr* represented as dimensions of the grid overlay) – k.

2. Number of partitions for each attribute in *g-attr*– p_i (described in the Section 3.3 of this chapter).

A similar methodology, which is used to rank the *e-attr*, can also be applied here.

We construct total n_e (size of *e-attr*) grid overlays with k attributes as dimensions in the ranking order defined using the ZSP-score of each attribute in *e-attr*. Here, k (size of *g-attr*) ranges from 1 to n_e. This considers that the attribute with a higher ZSP-score must be present in the grid overlay if the attribute with a lower ZSP-score is already present. Note that we use the same memory budget M for all n_e grid overlays. To complete the bigger picture first, the details for determining the number of partitions for each attribute in *g-attr*, constructing the linear scale (partition boundary values) based on the number of partitions using GPU, constructing the grid overlay using GPU, and performing dynamic query-based sampling on the grid overlay using GPU have been elaborated in the subsequent sections and chapters.

Each of the n_e grid overlays is evaluated upon a query set of size n which is generated using the same philosophy of the ZSP-testing query set. Each query in this query set (*Grid-testing query set*) contains the random number of equality (or narrow-range) predicates on randomly chosen attributes from *e-attr* in conjunction, and predicate constant for each predicate of a query has randomly chosen from data or data-domain. The Grid-testing query set can also be augmented using the real-world query workload if it is available. This generation procedure of the Grid-testing query set enables it to equally contain all possible combinations of query conjuncts for equally representing predicates on dimensional and non-dimensional attributes.

For each grid overlay and for each query from the Grid-testing query set, we draw the S number of query-based dynamic samples from the grid overlay using the procedure described in Chapter 4 and determine the percentage of queries that resulted into the Zero Sample Problem. This denotes the ZSP-score – $G_{zsp}(k)$ for the respective grid overlay with k grid-attributes. Finally, for query estimation, we use the value of k for which $G_{zsp}(k)$ is minimum. This final grid overlay essentially represents the effective number of necessary grid-attributes with the memory budget M, and how the total number of cells needs to be divided among grid-attributes to bring down the overall Zero Sample Problem rate. The Grid-testing query set can be divided into multiple batches to execute them in parallel. Moreover, we show in Section 3.4 that constructing the grid overlays, which is a part of this solution, is extremely cheap using GPU (just takes a few hundred milliseconds with around 2 million rows), which makes this procedure much cheaper.

3.3 Creation of Linear Scales

Let $g_1, g_2, \ldots, g_k$ be the k grid-attributes in *g-attr* (the subset of *e-attr* represented as dimensions of the grid overlay) for the respective table. The elaboration for how k has been chosen is given in the previous section. We now create a linear scale for each of these grid-attributes (an array of partition boundary values) with p_i number of partitions for attribute i. First, we look at how we determine the boundary values given the number of partitions (p_i) for the respective grid-attribute. Then, we propose a methodology based on the ZSP-scores of the grid-attributes to determine the number of partition boundaries – p_i.

3.3.1 Determining Partition Boundary Values

Given p_i, the partition boundary values are chosen from data-distribution itself, equivalent to those produced from an equi-depth histogram with p_i buckets on the attribute. A naive way to generate an equi-depth histogram is by sorting the attribute, which is very costly with big-data and single-core implementation. Again, it is well-known that GPU-based sorting can be quite

efficient (i.e., even the array of size 1 million can be sorted in a few milliseconds). Here, we assume that the entire column of an attribute can be stored inside the GPU-memory, which is a safe assumption to make as state-of-the-art GPUs easily have 16/32 GB of main memory. As the table is stored in a row-order and the entire table may not be able to fit into GPU-memory, the table can be logically divided into multiple batches. For each batch of rows, the GPU-kernel can be launched, which assigns one row to one GPU-core, and the corresponding GPU-thread copies the attribute value on the desired location. Once we have all values of the attribute inside the GPU-memory, sorting can take place. After that, to choose the histogram boundaries, the GPU-thread for each value of the sorted attribute can be called, and appropriate GPU-threads corresponding to boundary values write the attribute value to GPU-memory. Note that, as this resultant array of partition boundary values may not be sorted, it needs to be sorted before copying back to CPU memory.

Although a naive way to generate an equi-depth histogram is cheap and feasible using GPU, the cheaper approximate techniques based on sampling (e.g. [21]) can also be used which do not require GPU.

3.3.2 Determining the Number of Partitions for all Dimensions

A simple solution to determine p_i would be to use the same granularity on all dimensions, i.e., $p_i = \sqrt[k]{E}$ where E is the number of cells that can be accommodated in the grid overlay, derived from the memory budget M. However, the specific data characteristics of each dimension could be leveraged using its ZSP-score to provide a customized partition cardinality while obeying the overall E grid cell constraint. The idea is to assign more number of partitions to the attributes that are more likely to cause the Zero Sample Problem. So, based on the ZSP_i-score corresponding to each g_i, the assignment of the number of partitions to g_i is based on the value of ZSP_i-score normalized to the sum of all ZSP_i-scores. Specifically, suppose we denote $Z = ZSP_1 + ZSP_2 + ... + ZSP_k$, the following equation determines p_i.

$$p_i = E^{(ZSP_i/Z)} \tag{3.1}$$

Here, ZSP_i-score and ZSP_i are used interchangeably. Moreover, the experiments in Chapter 6 show that these selection criteria of choosing the grid-attributes (Section 3.1 and Section 3.2) and determining the number of partition boundaries for each grid-attribute achieve significantly better accuracy compared to the grid overlay on randomly chosen grid-attributes with an equal number of partition boundaries.

3.4 Creation of Grid Overlay & Populating Grid Cells

Using the array of partition boundary values of the grid-attributes as described in Section 3.3, we create a memory-resident multidimensional array $\mathcal{G}$ containing $\Pi_{i=1}^{i=k} p_i$ number of grid cells which obeys the constraint on the total number of grid cells E. The grid cells are initially empty but are progressively populated without using GPU and with using GPU as described in subsequent subsections.

3.4.1 Constructing Grid Overlay Without using GPU

A sequential scan of table T is carried out, and for each row r in the table, we use its values for the *g-attr* attributes as a multi-dimensional index into $\mathcal{G}$. Then, we insert a tuple identifier (*TId*) of the row in the associated cell c of $\mathcal{G}$.

There are two ways to populate the grid overlay – the first would be to build the grid file as described in the paper [53], with splitting and merging of grid cells based on a maximum cell occupancy. The simpler alternative, which we use here, is to instead permit grid cells to have, in principle, arbitrary occupancy – this is achieved through the "vector" feature of the STL library in Cpp [2], which automatically reallocates the memory so as to retain the array structure. Specifically, the size of the array is increased by 50 percent each time there is a cell overflow, and this dynamic design provides constant amortized insertion cost. Total time for creating the grid overlay without using GPU takes a few seconds for 2 million rows of the Power dataset [4].

3.4.2 Constructing Grid Overlay using GPU

Instead of creating on a main-memory of the CPU, we can create a multi-dimensional array $\mathcal{G}$ on the GPU's main-memory. First, all the rows (or batch of rows) of the dataset need to be copied from CPU-memory to GPU-memory. Then, the grid cells are initially empty but are progressively populated as follows: each row r of a table will be assigned to a one GPU-core. The corresponding GPU-thread will atomically insert the *TId* of the row in the associated grid cell c of $\mathcal{G}$. For the GPU as well, we used a "vector" like implementation to support arbitrary grid cell size. The total time taken to create the grid overlay on GPU-memory, including allocating memory for the grid overlay and copying all the rows (or the batches of rows) of the table into GPU-memory, is extremely low. We evaluated this with the 2 million rows of Power dataset [4] by constructing a 5-dimensional grid overlay which only took around a few hundred milliseconds. Note that we do not need to copy a multi-dimensional array $\mathcal{G}$ back to CPU-memory from GPU-memory as grid-based dynamic sampling can also be performed on the GPU itself

as described in Chapter 4. Suppose the table cannot be fitted into GPU-memory. In that case, the table can be easily divided into multiple batches of rows, and the process mentioned above of populating the grid cell can be repeated for each batch of the table.

To make the above concrete, a sample two-dimensional grid overlay for a generic table T is shown in Figure 1.3 of Introduction. The attributes $T.a$ and $T.b$ form the dimensions, and the corresponding linear scales are shown on the axes. The "vector" tuple identifiers ($TIds$) and their total count (N) are shown within each grid cell.

3.5 Handling the Data-updates for GridSam

In the case of dynamic updates of the rows, only "vector" of the corresponding grid cell needs to be altered, which is a very cheap operation. Therefore, this grid overlay can easily support online updates to the dataset. Moreover, updates corresponding to different grid cells can be done in parallel. Here, we assume that the updates are going to be mostly "appends". In case of major updates to the dataset, although redesigning and reconstructing the grid overlay is cheap, there is no need to redesign the entire grid overlay by performing all the design steps explained earlier, unless the changes in data-distribution of attributes lead to the significant change in the ZSP-scores of the respective attributes relative to the ZSP-scores of other attributes. We can periodically compare the previous and current ZSP-scores of all the attributes in $e\text{-}attr$.

3.6 GridSam with Joins: Grid Overlays for Multi-Table Queries

As highlighted in [46], the Zero Sample Problem is particularly pronounced in the case of join-crossing correlations. So the solution that they proposed, called IBJS, was to leverage the presence of indexes as follows: A starting table from which initial samples are drawn is selected, and subsequently, the join order path is followed by drawing index-joined samples from the respective tables in the path and the filter predicates from the respective table in the join path are also being applied. There are certain limitations with the IBJS approach as follows:

1. The Zero Sample Problem can occur after applying filter predicates on join-samples (or initial-table samples) of any table from the join path.

2. The indexes like B-Tree and ART (Adaptive Radix Tree) [44] proposed to be used in [46] follows the hierarchical structure, which might be costly with query and database scale.

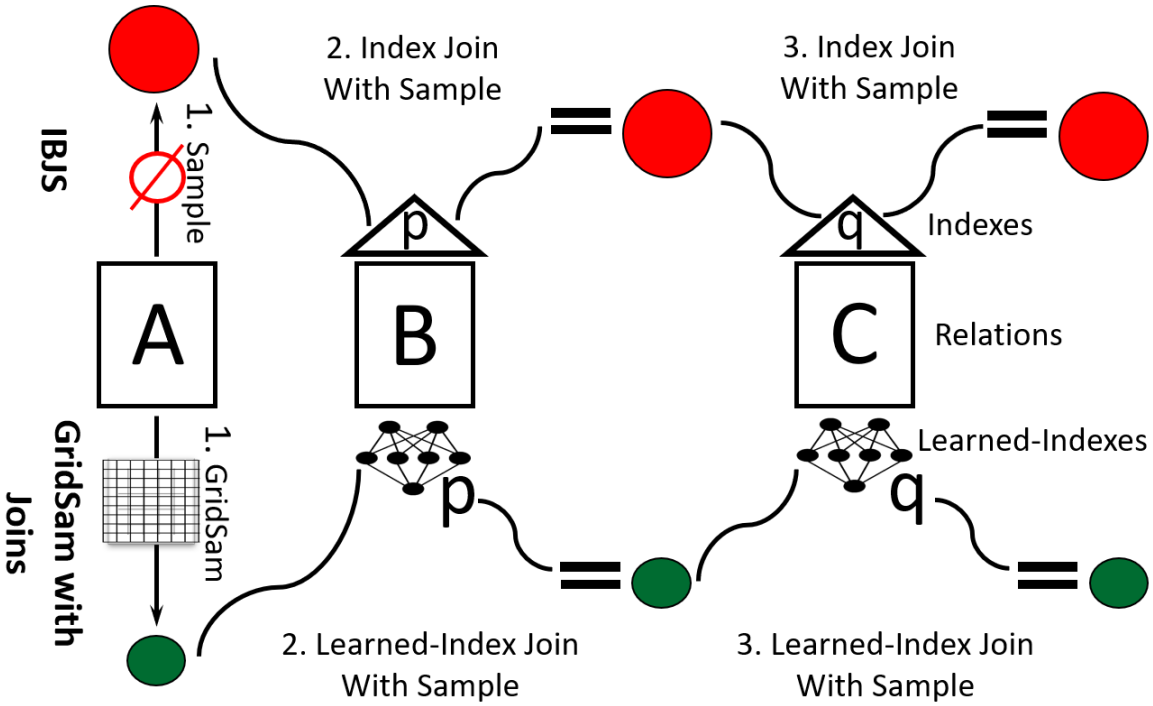


Figure 3.1: GridSam for Joins: GridSam with the Framework of IBJS

As shown in Figure 3.1, GridSam can be easily incorporated in the framework suggested for IBJS in [46] by constructing the grid overlays for multiple tables in the schema. However, a difficulty in the context of GridSam with the above approach is that while the initial samples can be taken from the grid overlay of the starting table, these samples are disconnected from the samples taken from the subsequent overlays, which again runs the risk of Zero Sample Problem. Therefore, we take recourse to construct the grid overlays with grid-attributes from the other tables, which can essentially be in the join path, along with the grid-attributes from the same table.

3.6.1 Grid Overlay with Foreign Attributes

The grid overlay for a table is constructed not only with the *native* attributes of the table but may also feature *foreign* attributes sourced from other tables. In particular, we consider *PrimaryKey-ForeignKey* (PK-FK) join attributes from the database schema graph and include those attributes that are likely to run into Zero Sample Problem issues. The process for determining these attributes is similar to that of the native attributes, described in previous sections, with the propensity for ZSP-score being used to determine the need to include a specific foreign attribute in the overlay – these ZSP-score statistics are determined by explicitly computing the joins on the table samples and evaluating their outcomes.

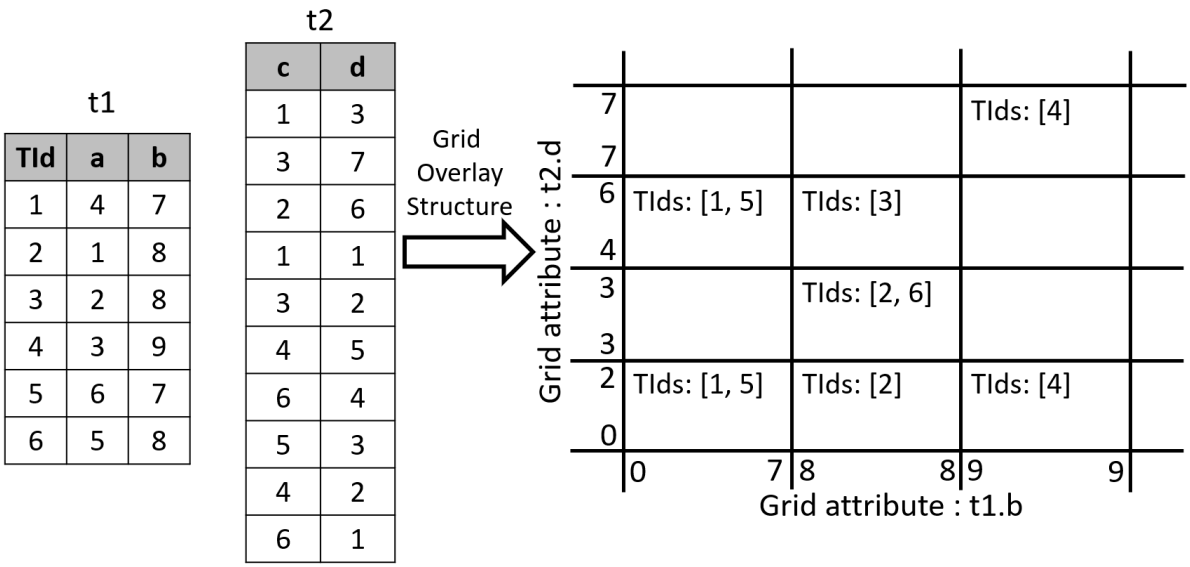


Figure 3.2: Grid Overlay Example for Join (t1.a=t2.c)

Moreover, we propose to use learned-indexes [42] in place of other traditional indexes proposed in [46] as it has much lower and constant probing cost compared to others [41]. The details for incorporating learned-indexes to GridSam are described in Section 3.9.

A related point to note is that while populating the grid cells when foreign attributes are included in the grid overlay, the tuple identifiers (*TIds*) of only the *source* table are maintained in the grid cells. To make the connections between the rows of the two tables, the learned-indexes, discussed in Section 3.9, are used. To make this construction clear, a sample grid overlay including foreign grid-attribute is shown in Figure 3.2 for the schema containing tables $t1$ and $t2$, where the grid-attributes are relevant to filter predicates on $t1.b$ and $t2.d$, and a learned-index is used to connect the join attributes $t1.a$ and $t2.c$.

In general, the grid overlay of a table can include foreign attributes from any other table with which it has either direct or transitive key-based relationships under the memory budget constraint. Note that, in order to create the grid overlay containing foreign grid attributes using GPU, all the foreign tables (tables that contain foreign grid-attributes) or the columns of the foreign attributes need to be fitted in GPU-memory. The local base-table can be divided into multiple batches if they cannot be fitted into GPU-memory. However, foreign tables cannot be divided as the order of rows in the respective batch of the foreign table may not be consistent with the storage order of the join attribute in the corresponding batch of the local table. So, the grid overlay containing foreign grid-attributes needs to be built without using GPU if foreign tables or columns cannot be fitted into GPU-memory. We plan to support the construction of

the grid overlay containing foreign grid-attributes using GPU as a part of future work.

3.7 Assigning Memory to Multiple Tables from the Total Memory Budget

This section discusses the issue of dividing memory to multiple tables in the database schema to build a grid overlay per table, given a fixed total memory budget of M. The simple solution is to divide the total memory budget M equally to all the tables in the schema. This may not be an "effective" choice to bring down the rate of Zero Sample Problem mainly because of the following reasons:

1. Some tables may not contain attributes with "complex" data properties, and thus, they may not need to build the grid overlay.

2. Different tables may contain different numbers and magnitudes of "complex" attributes, and thus, they need different sizes of memory to build a grid overlay.

We propose the following methodology based on ZSP-score, which takes care of both of the above issues.

Let $T_1, T_2, \ldots, T_t$ be the t tables in the concerned database schema. Each table T_i has its *e-attr* (set of error-prone attributes), and as described in Section 3.1, ZSP-score of each attribute from *e-attr* has been calculated. We define $T_{zsp}(i)$, the ZSP-score for the table T_i, which is an arithmetic mean of the ZSP-scores of all attributes from *e-attr* of the table T_i. Here, we used arithmetic mean as an aggregation function mainly because ZSP-score of each attribute is independent of each other, and each table in the schema can have a different number of attributes. $T_{zsp}(i)$ denotes the vulnerability to the Zero Sample Problem of the table T_i relative to other tables in the schema. $T_{zsp}(i)$ will have a relatively higher value if table T_i has more "complex" attributes relative to other tables in the schema. The idea is to assign more memory bytes to the tables that contain more "complex" attributes in magnitude and number.

So, based on the $T_{zsp}(i)$ score corresponding to each T_i, the assignment of the memory to T_i is based on the value of $T_{zsp}(i)$ normalized to the sum of all $T_{zsp}(i)$, same as described in Section 3.3. Specifically, suppose we denote $T_{zsp} = T_{zsp}(1) + T_{zsp}(2) + ... + T_{zsp}(t)$, the following equation determines M_i (memory budget for table T_i).

$$M_i = M \times (T_{zsp}(i)/T_{zsp}) \tag{3.2}$$

Computing $T_{zsp}(i)$ for each table is cheap as it just requires doing the arithmetic mean of the

ZSP-scores of all attributes.

3.8 Determining the Number of Tables to Build Grid Overlays

The $T_{zsp}(i)$ score to each table T_i also determines the order of priority in which memory should be assigned. Analogous to the methodology described in Section 3.2, given the fixed memory budget M, the number of tables for which the grid overlay needs to be built can be determined by the following steps:

1. Divide the total memory budget M to g_t number of tables (number of tables selected to build grid overlays) in the schema based on their respective $T_{zsp}(i)$ values, where g_t ranges from 1 to t.

2. Build the grid overlays on g_t tables in the decreasing order of their $T_{zsp}(i)$ values (i ranges from 1 to t) within assigned memory budget. The grid overlay of the corresponding table uses *g-attr* as the dimensions. The *g-attr* of each table is determined using the procedure described in Section 3.2.

3. Evaluate all these t configurations (grid overlays on g_t tables, Memory budget M) for *Multi-Grid-testing query set* on S query-based dynamic samples and determine the ZSP-score of each configuration. The procedure to build a Multi-Grid-testing query set is described later in this section.

4. For cardinality estimation, choose the configuration with minimum ZSP-score among all t configurations.

Within a fixed memory constraint, in order to bring down the rate of Zero Sample Problem, this final schema configuration (all the grid overlays in the schema together) represents the necessary tables to build the grid overlays and how much memory is required for each table.

Note that each query in the Multi-Grid-testing query set contains the random number of randomly selected tables from the schema. For each table, the query contains the random number of equality (or narrow-range) predicates in conjunction on random attributes from *e-attr* of the corresponding table. The predicate constant for every predicate of each query has randomly and equally chosen from data, data-domain or query workload if available.

In total, using assigned memory budget, we need to build and do sampled evaluation of $\sum_{i=1}^{i=t} n_e(i)$ number of grid overlays in order to identify the *g-attr* for all tables T_i in the schema using Grid-testing query set, where $n_e(i)$ denotes the size of *e-attr* for table T_i (Section 3.2).

After that, we need to build and do sampled evaluation of $\frac{t\times(t+1)}{2}$ grid overlays using the same *g-attr* to determine the ZSP-score of all t configurations using Multi-Grid-testing query set. As the construction of the grid overlay is extremely cheap on GPU or only needs a single pass on the corresponding table on CPU, and sampled evaluation of the corresponding query set is cheap, this procedure can be completed in a reasonable amount of time. As the complexity of this procedure mainly depends on t (number of tables in the schema), this procedure can be costly for a large value of t. We used the IMDB database [5] containing six tables, which is also suggested in the paper [40] for the experiments. We show that this entire procedure of determining the design parameters and constructing the required grid overlays for multi-table schema can be completed in a few hundred seconds.

3.9 Learned-Indexes for Join Sampling

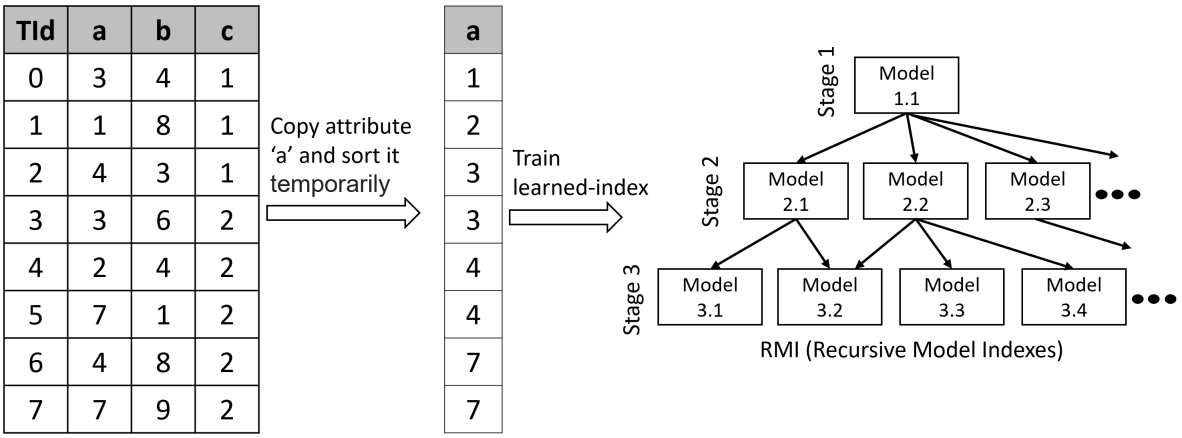
We now turn our attention to the choice of indexes for connecting the joining tables after sampling from the initial overlay table. In the IBJS technique [46], they considered both B-trees, as well as a customized hash-based alternative called ART [44], intended for faster probes. However, a common feature of both these techniques is the recursive traversal of hierarchical structures, which may become expensive with database and query scale. We have therefore chosen the alternative of the recently proposed learned-indexes [42] which incurs a constant probing cost that is substantially lower than traditional indexes and reduces space overheads – in fact, a recent detailed study [41] of a variety of indexes shows that learned-indexes are the preferred choice and incur sufficiently low training time. For the experiments and evaluations, we used the implementation of learned-index provided by the authors on GitHub [9].

A requirement, however, with learned-indexes is that they need to be physically sorted on the indexed attribute. This creates a problem if multiple learned-indexes need to be built on a table since the table cannot be simultaneously sorted on all these attributes. Therefore, we need to incorporate auxiliary data structures to adapt the learned-index to the IBJS framework. Specifically, we need to be able to compute the following.

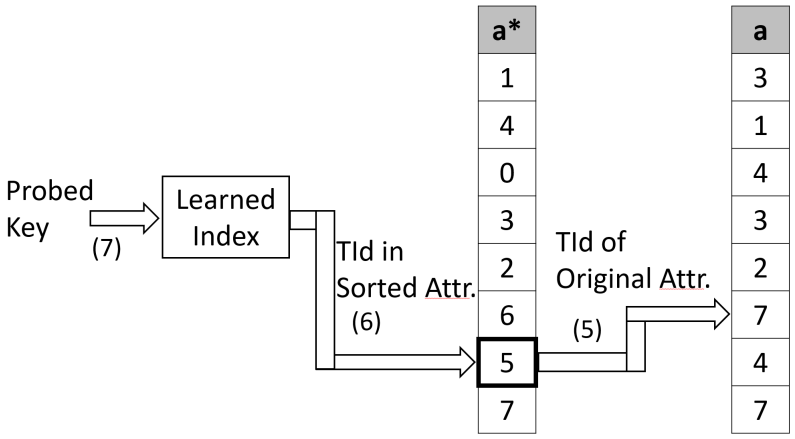
1. Given the position, what is the corresponding value?

2. Given the value, what is the corresponding position?

The first query is automatically supported as we do not change the physical data layout of the table. To support the second, a mapping is required from the sorted position of the key to its physical position. For this purpose, we temporarily copy each attribute into an array on which a learned-index has to be constructed, then sort the array, and finally build the learned-index on this array. Again, the power of GPU can be leveraged to sort the attribute and train



(a) Building Learned-Index on Unsorted Attribute T.a



(b) Using Learned-Index to get Physical Location

Figure 3.3: Learned-Index for Unsorted Attributes

the learned-index. An example is shown in Figure 3.3a where a learned-index for unsorted attribute $T.a$ is created using the RMI (Recursive Model Index) framework of [42].

Concurrently, we create a mapping from the sorted logical position to the physical position in the original table. This mapping is subsequently leveraged at index usage time to quickly locate the physical position. Figure 3.3b shows an example where the learned-index is probed with key 7, which gives the first *TId* of a key 7 in a sorted array which is 6 (starting from 0). Then, we lookup the *TId* 6 into the auxiliary mapping a^* of an attribute a, which will give us the first *TId* of a key 7 into the original unsorted table.

It is possible that given primary key maps to multiple foreign key entries in the join index. So, in order to know the join cardinality for primary key value v, we need to identify the number of such matching entries. This is done by first probing the learned-index with value v, and then with one of its adjacent values – either $v - 1$ or $v + 1$ – the difference of both the *TIds* gives the match cardinality assuming that *TIds* are continuously allocated.

Chapter 4

Cardinality Estimation

In the previous chapters, we described the lightweight auxiliary data structures that are constructed at system initialization time. We now move on to describe how these structures are used to efficiently estimate the result cardinalities of user queries that are submitted to the system. For ease of presentation, the estimation of filter predicates on a single table is described first, followed by filter predicates across joined tables.

4.1 Single-Table Query

Consider the query of the form shown in Figure 4.1 on the single table T with two attributes $T.a$ and $T.b$ based on the grid overlay shown in Figure 1.3 of Introduction. Assume for now that the grid overlay on table T includes both the attributes $T.a$ and $T.b$ as grid-attributes (dimensions of the grid overlay). Given this situation, the estimation procedure is as follows. Firstly, the *minimum bounding rectangle* (MBR) of the query on the grid overlay is identified using the linear scales on the $T.a$ and $T.b$ dimensions – this MBR is denoted as $\mathcal{H}$. Let the range of partition indices for $\mathcal{H}$ be $[p, q]$ and $[r, s]$ respectively on two dimensions. For the running example of query presented in Figure 4.1, $\mathcal{H}$ is highlighted in the grid overlay of table T shown in Figure 4.2. The range of partition indices for this query's $\mathcal{H}$ will be [1, 2] and [1,

```
SELECT  *
FROM   T
WHERE  T.a BETWEEN 270 AND 590
AND      T.b BETWEEN 30 AND 60
```

Figure 4.1: Query Example for Single-Table Estimation Procedure

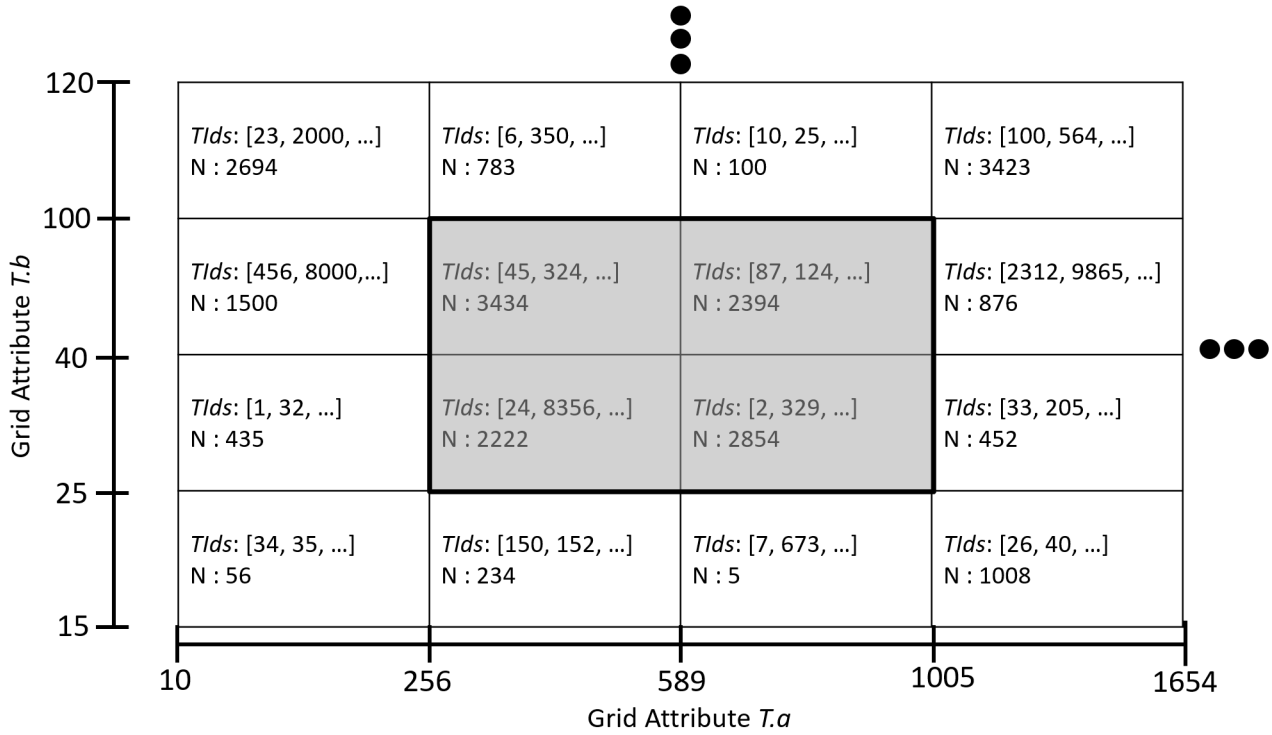


Figure 4.2: Grid Overlay with Selected Minimum Bounding Rectangle for the Query Presented in Figure 4.1

2], respectively, on both dimensions (linear scale index starts with zero for both dimensions). In this work, we are only considering the queries with AND operators (conjunction of filter predicates). However, the OR operators can be easily incorporated using the same methodology as AND operators by applying UNION to the partitions of the respective dimension(s) instead of INTERSECTION. The $\mathcal{H}$ can be identified in the grid overlay by performing the series of UNION and INTERSECTION operations for a given query containing AND/OR operators. We plan to conduct the detailed evaluation and performance comparison of GridSam on OR-queries as part of future work.

We next compute $N(\mathcal{H})$, the volume (total number of tuples) of $\mathcal{H}$. This could be achieved by accessing each grid cell $i \in \mathcal{H}$ and aggregating the individual N_i values. This can be very costly as the number of grid cells in $\mathcal{H}$ can be large. However, summing up the array elements is a heavily parallel task using the algorithm called *Parallel Reduction* [12]. So it is natural to use the GPU for this task. Current state-of-the-art GPUs contain thousands of cores, which can efficiently compute the sum of all elements of an array using the parallel reduction algorithm. The parallel reduction algorithm is described later in this section. The GPU can complete this task literally in microseconds, even for an array with a size in millions.

From the estimation perspective, $N(\mathcal{H})$ denotes the volume (total number of tuples) of the new sample space of the concerned query. For the running example of the query, $N(\mathcal{H})$ will be equal to 10,904. $N(\mathcal{H})$ will be further used for the extrapolation of the query's result count after the sampled evaluation.
Then, given a total sampling budget of S tuples, a weighted sampling budget, $S(cell_{i,j})$, is assigned to each non-empty grid cell (i, j) in $\mathcal{H}$ as follows.

$$\begin{aligned} S(cell_{i,j}) &= \lfloor \frac{N(cell_{i,j})}{N(\mathcal{H})} \times S \rfloor \;\; if \; N(\mathcal{H}) > S \\ S(cell_{i,j}) &= N(cell_{i,j}) \;\; if \; N(\mathcal{H}) \leq S \end{aligned} \tag{4.1}$$

Subsequently, the tuple contents of each grid cell are randomly sampled as per the assigned budget. Again, iteratively performing weighted random sampling for $\mathcal{H}$ containing a large number of grid cells can be very costly. Thus, we chose to perform this task by leveraging GPU's parallel resources. We describe the procedure for weighted random sampling on GPU later in this section. After getting the random samples, the user query Q is evaluated on these sampled tuples. Let the result cardinality from the samples be denoted as $res\_cnt(S)$. Finally, the result cardinality for Q over the data-space of $\mathcal{H}$ (confined sample space of the query Q) is estimated using the simple extrapolation estimator as:

$$ResCard_Q = \frac{res\_cnt(S)}{S} \times N(\mathcal{H}) \tag{4.2}$$

In general, if query Q does not contain filter predicate(s) on any of the grid-attributes from *g-attr*, the partition indices of the respective grid-attributes for $\mathcal{H}$ will be set between minimum and maximum. If filter predicates are not present on the grid-attributes from *g-attr*, it will be equivalent to performing random sampling over the entire table. Note that, if the filter predicates of a query Q are on the attribute set which is a subset of *g-attr*, the weighted random sampling would only be performed on the *partially covered* grid cells by $\mathcal{H}$. The partially covered grid cells can be easily identified among the grid cells on the boundaries of $\mathcal{H}$, if the upper and lower partition boundary values of the respective grid cell are not fully consumed by the boundary values of filter predicates of Q. The predicate boundary values of Q can be easily determined using the corresponding predicate operator and the predicate constant(s). The *fully covered* grid cells of $\mathcal{H}$ would directly be incorporated in $ResCard_Q$ without extrapolation. Apparently, for the running example query, all grid cells in query's $\mathcal{H}$ are on the boundaries of $\mathcal{H}$ and partially covered as well as shown in Figure 4.2. So, weighted

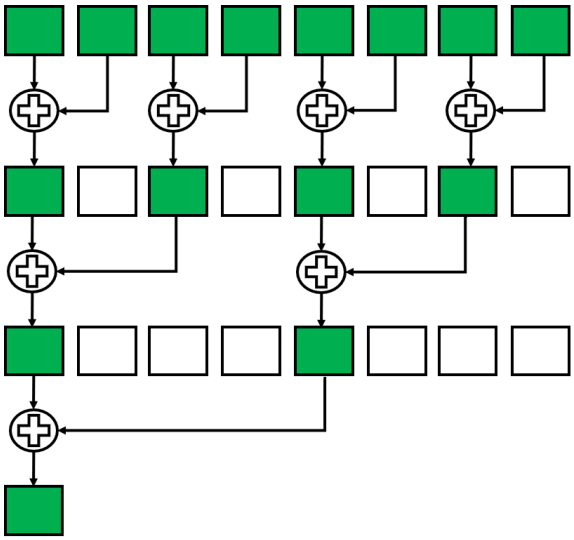


Figure 4.3: Parallel Reduction

random sampling needs to be performed on all grid cells of this query's $\mathcal{H}$.

4.1.1 Parallel Reduction

As shown in Figure 4.3, the parallel reduction algorithm works in the following way:

1. Assume that there are N elements in an array, $N/2$ GPU-threads will be initially invoked - one GPU-thread for every two elements (denoted by the addition operators of the first iteration in Figure 4.3).

2. Every GPU-thread is responsible for computing the sum of the corresponding two elements. Then, as shown in Figure 4.3, every GPU-thread stores the result of the sum at the position of the corresponding first element.

3. Iteratively, for each step:

   (a) The number of GPU-threads is going to be halved (e.g. if initially, 4 GPU-threads are invoked, then in the next iteration, it is going to be 2, and then 1).

   (b) The step size between the corresponding two elements is going to be doubled, respectively (denoted by the coloured boxes in Figure 4.3).

4. After $\log_2 N$ iterations, the result of parallel reduction (total sum of an array) will be stored in the first element of the array.

Note that we do not need to copy the entire multi-dimensional array $\mathcal{G}$ of the corresponding grid overlay (stored in row-major order) to the GPU-memory if it is already constructed in the GPU-memory itself. While summing up the total number of tuples of $\mathcal{H}$ using the parallel reduction algorithm, we use the entire grid overlay array $\mathcal{G}$. The reason is that the grid cells corresponding to $\mathcal{H}$ may not be contiguous. However, we consider only grid cells that belong to $\mathcal{H}$ and ignoring other grid cells.

4.1.2 Weighted Random Sampling within $\mathcal{H}$

There can be a large number of grid cells within $\mathcal{H}$, especially in the case of a large memory budget. The weighted random sampling needs to be performed for each non-empty grid cell that belongs to $\mathcal{H}$. The naive way to do this is by iteratively scanning each grid cell that belongs to $\mathcal{H}$, and perform weighted random sampling if it is non-empty. As the number of grid cells in $\mathcal{H}$ can be very large, this may be a very costly operation and can significantly hurts the overall estimation time. Again, identifying the non-empty grid cells and fetching random samples from the non-empty grid cells are heavily parallel tasks. Each grid cell can be checked independently whether it is non-empty and belongs to $\mathcal{H}$ or not.

The GPU can be easily leveraged for these tasks. As current GPUs usually handle hundreds of thousands of active threads, each grid cell can be assigned to one thread of a GPU to check whether it belongs to $\mathcal{H}$ of the query and non-empty or not. If the respective grid cell belongs to $\mathcal{H}$ of the query and non-empty, then the *TIds* of random samples from the respective grid cell can be written directly to GPU-memory. If the total number of grid cells is larger than the maximum number of active GPU-threads, the GPU-scheduler divides the array of grid cells ($\mathcal{G}$) in the batches of size equal to the maximum number of active threads possible in the GPU. These batches run serially on the GPU. This should not be an issue for even a very large array $\mathcal{G}$ with millions of grid cells, as the state-of-the-art GPUs can support several hundred thousands of active GPU-threads simultaneously.

In the experiments, we observed that these tasks could be completed within several milliseconds using the GPU, even with the half-a-billion grid cells in $\mathcal{G}$. After copying back the *TIds* of random samples from GPU-memory to CPU-memory, the respective query can be evaluated to get the estimates using CPU. The experimental evaluation shows that the above GPU-based algorithms keep the estimation time much lower (a few milliseconds).

4.2 Multi-Table Query

We now consider a sample multi-table query of the form shown in Figure 4.4 based on the grid overlay shown in Figure 3.2 of Chapter 3 where table $t1$ has primary-key attribute $t1.a$ and

```
SELECT  *
FROM   t1, t2
WHERE  t1.a = t2.c
AND      t1.b = 8 AND t2.d ≥ 6
```

Figure 4.4: Query Example for Multi-Table Estimation Procedure

table $t2$ has foreign-key attribute $t2.c$. Assume that the dataset schema only contains table $t1$ and $t2$. Further, assume for now that the two grid-overlays $\mathcal{G}$ ($t1$) and $\mathcal{G}$ ($t2$) are built for the table $t1$ and $t2$ respectively. $\mathcal{G}$ ($t1$) contains 2 dimensions: one native attribute ($t1.b$) and one foreign attribute ($t2.d$). $\mathcal{G}$ ($t2$) also contains 2 dimensions: one native attribute ($t2.d$) and one foreign attribute ($t1.b$).

4.2.1 Choosing Table for Initial Samples

There can be multiple tables whose grid overlays contain *g-attr* which are present in the filter predicates of the query. Ideally, the initial samples should be taken from the table with the smallest sample space. So, we first determine the $N(\mathcal{H}_i)$ for each table t_i which is the total volume of $\mathcal{H}_i$ (hyper-rectangle of the grid overlay for table t_i), based on the set of *g-attr* predicates present in the query. Then, we choose the table with the least $N(\mathcal{H}_i)$ to generate the initial samples in the context of IBJS technique.

If the grid overlay is not built on table t_i, then $N(\mathcal{H}_i)$ would be equal to total number of rows in table t_i and sample space would be an entire table t_i. If all the tables present in the query for which the grid overlay is not built, GridSam will behave equivalent to IBJS technique. In the above example, we compute $N(\mathcal{H}_1)$ and $N(\mathcal{H}_2)$ and choose the table with the lower value. Note that initial samples from the grid overlay of the respective table are chosen in the same way as described in Section 4.1.

4.2.2 Estimation Algorithm

After securing the samples from the chosen initial table, the following subsequent process for traversing the join graph and obtaining correlated join-samples is similar to that of IBJS technique:

1. In order to choose table from $t1$ and $t2$ to draw initial samples, we compare $N(\mathcal{H}_1)$ and $N(\mathcal{H}_2)$. For this example query, both $N(\mathcal{H}_1)$ and $N(\mathcal{H}_2)$ are same, as both *g-attr* present

in the query are common in both $\mathcal{G}$ ($t1$) and $\mathcal{G}$ ($t2$). So, we can pick any table to start with from $t1$ and $t2$. Let's assume that we start with table $t2$.

2. Select S samples using weighted random sampling policy among all qualifying grid cells of $\mathcal{G}$ ($t2$), as explained in Section 4.1 for the single-table scenario.

3. Apply the predicate of table t_2 (i.e. $t_2.d \geq 6$ to S samples), and assume that k_2 of them satisfy the predicate. Then, *est* is updated as:

$$\frac{N(\mathcal{H}_2) \times k_2}{S} \tag{4.3}$$

4. Select a table with which table $t2$ can be joined using a key (PK-FK or FK-PK) relationship. In this case, $t2$ can be joined only with table $t1$, otherwise select the target table randomly if t_2 joins with multiple tables.

5. Probe the index of $t1.a$ with the value of $t2.c$ for every tuple from k_2 samples. Also maintain the $total\_count(t_1)$ variable which stores the number of tuples from table $t1$ that join with the k_2 satisfied samples of table $t2$. Update the running estimate as:

$$est = \frac{N(\mathcal{H}_2) \times total\_count(t1)}{S} \tag{4.4}$$

6. If $total\_count(t1)$ is more than S, select S random tuples from this set. Then, apply the filter predicate(s) of table $t1$ (i.e. $t1.b = 8$) to S and assume that k_1 tuples satisfy the predicate. Update the running estimate as:

$$est = \frac{est \times k_1}{S} \tag{4.5}$$

After the above process terminates, the final *est* value provides the result cardinality estimation for the entire query. In general, these estimation steps can be followed for any query with any number of joins in the path.

Chapter 5

Experimental Framework

We have carried out a detailed evaluation on the performance of GridSam with regard to the following measures:

1. **Estimation Accuracy:** The estimation accuracy of the query result cardinality is measured using the defacto standard *q-error* metric [50], defined as

$$q\text{-}error = max(\frac{act}{est}, \frac{est}{act}) \tag{5.1}$$

   where *act* denotes the actual cardinality, and *est* denotes the estimated cardinality.

2. **Compile-time (Pre-processing) Overheads:** This measures the space and time overheads incurred to determine the grid overlays' design parameters and constructing the grid overlays along with learned-indexes.

3. **Run-time Overheads:** This measures the space and time taken for carrying out the cardinality estimation at run-time.

Note that we have used a NVIDIA Tesla V100 GPU [18] for the experiments along with CUDA Programming Interface [3] to implement the GPU-based algorithms in Chapter 3 and Chapter 4. This GPU has 80 multiprocessors and 64 cores per multiprocessor with warp size of 32, which makes the maximum number of active GPU-threads equal to 163,840, and that is enough to efficiently handle the array of size even in billions. Our experiments have been conducted on a variety of database environments, including those considered in the recent learning-based studies [23, 40, 60]. The details are given in the remainder of this chapter.

5.1 Datasets

The following three real-world datasets were considered in our evaluation:

1. **Forest [4]:** This dataset, previously evaluated in [23], has a single table containing over half-a-million rows with 54 attributes, several of which exhibit non-linear correlations. This dataset contains cartographic variables mostly for wilderness areas. Our attention is restricted to queries on the 10 numeric attributes since the other attributes are all binary.

2. **Power [4]:** This dataset, also used in [23], has a single table containing electric power consumption measurements in a household at a one-minute sampling rate over 4 years. It contains over 2 million rows, and our focus is on 7 numeric attributes in these rows.

3. **IMDB [5]:** This movie database, which featured in the popular Join-Order Benchmark (JOB) [6], has a set of 6 tables characterized by correlations that are challenging for cardinality estimators which is established in [45]. The dataset captures more than 2.5 million movie *titles* produced over 133 *years* by 234,997 different *companies* with over 4 million *actors*. Note that, for experiments and evaluation, we used the identical replica of this dataset which is used in the MSCN paper [40] availed by the authors.

5.2 Comparative Estimators

We compare GridSam with the following representative suite of estimation techniques, which run the gamut from the simple statistical estimators used in current database engines to deep-learning based techniques.

5.2.1 Statistical-based Estimators

1. **Postgres [14]:** This represents the traditional database estimators which are based on uni-dimensional histograms along with other statistics and assumptions such as Attribute-Value-Independence (AVI) for producing estimations. The configuration parameters of the installation were tuned using PGTune [13], except the number of buckets in all histograms and the size of the Most Common Values (MCV) array. We maxed out the MCV array size and the number of histogram buckets to 10,000 for each attribute for the fair comparisons on memory budget. Note that, 10,000 is a much higher number than the data-domain size of any attribute from the Power and Forest dataset.

2. **Random Materialized Samples (RMS):** This represents the traditional random sampling over the entire data space of a table and is only used in the evaluation of the

single-table datasets. We used the Postgres estimator if the query falls into the Zero Sample Problem in RMS.

3. **Index-based Join Sampling (IBJS) [46]:** This sampling-based technique uses the indexes on join-attributes to fetch the join-correlated samples to solve the issue of Zero Sample Problem due to join-crossing correlation for multi-table queries. We discussed IBJS technique in detail for multi-table queries in Chapter 2. For the evaluation, we have implemented the IBJS estimation technique using Cpp programming language [1] and used learned-indexes [42] on join-attributes for join-sampling. Here also, we used the Postgres estimation in the case of Zero Sample Problem. We consider the IBJS technique as a baseline for sampling-based techniques which support multi-table queries.

5.2.2 Learning-based Estimators

1. **Lightweight Model (LWM) [23]:** This represents a supervised lightweight NN-based regression technique for capturing intra-table correlations between attributes. We have implemented it using Python programming language [16] and Keras programming interface for deep learning [7], as described in [23]. This model is evaluated only for the single-table datasets as it does not support queries with multiple tables. As suggested in the paper [23], we added the heuristic estimators as features for each query during training and testing of the model. The GPU acceleration has been used for model hyper-parameter tuning and model evaluation while predicting the cardinalities. We have followed the same hyper-parameter tuning procedure suggested in the paper [23] to tune the model.

2. **MSCN [40]:** This technique provides an estimation model for multi-table queries using a set-based query representation scheme hosted on a multi-set convolutional neural network. For experiments and evaluation, we used the implementation, training data, and testing data provided by the authors on GitHub [8]. As suggested in the paper [40], the training is done with 100,000 synthetically generated queries going up to 2 joins. Further, as suggested in the paper [40], a bitmap of 1000 samples was additionally used as the input features for each table in all the queries in the workload during training and testing of the model. The GPU acceleration has been used for model evaluation while predicting the cardinalities. We also followed the same hyper-parameter tuning methodology on GPU as suggested in the paper [40].

3. **End-To-End (E2E) [56]:** This technique provides an end-to-end tree-structured NN-based model for both cost and cardinality estimation, including feature extraction from

the query execution plan. As per this technique, the model is trained using 100,000 synthetically generated queries going up to 3 joins with 1000 bitmap samples as input features. We used the reported results for both accuracy and efficiency from the paper [56] as we are using exactly the same query workloads for evaluation.

4. **NARU [62]:** This technique proposed to use the autoregressive model by factorizing the joint distribution into conditional distributions. They adopted state-of-the-art deep autoregressive models such as MADE [28] and Transformer [58] to approximate the joint distribution. For experiments and evaluation, we used an implementation provided by the authors on GitHub [10]. We used the same sample size (S) of 1000 that we used for GridSam and RMS for the evaluation of NARU to perform progressive sampling. Moreover, we used the same hyper-parameters suggested in paper [62] for our evaluation. Note that this technique only provides the model for single-table datasets. Thus, NARU has been excluded from our evaluation for the multi-table dataset.

5.3 Training Query Generation

Our generation of training queries for the single-table datasets (Power and Forest) is identical to the method suggested in [23]. Here, queries containing conjunctive range predicates are generated, having 2 to 7 predicates. Each range predicate is determined by identifying an attribute along with a center and range width. The attributes are picked randomly, while the query centers follow random-centric distributions in conjunction with uniformly-distributed widths over the attribute data-domain, or data-centric distributions in conjunction with exponentially-distributed widths. We generated nearly 25,000 such queries in each of our training exercises for both LWM and MSCN models. NARU does not need queries for training, as it directly learns from data itself, being an unsupervised learning estimator.

For multi-table dataset (IMDB), training query set provided by the authors of [40] on GitHub [8] is used.

5.4 Testing Query Generation

The test queries on the single-table datasets (Power and Forest) are divided into two classes –the first, *LowDim* having 2 to 4 predicates, and the second, *HiDim* having 5 to 7 predicates. We generated 3000 queries per class using a similar methodology as generating training queries.

We used the same three testing workloads for multi-table queries provided by the authors of MSCN [40] on GitHub [8]. These exact three workloads are also used by End-To-End paper [56]. The properties of these three workloads are as follows.

1. **Synthetic Workload** which contains 5000 randomly distributed queries with up to 2 joins

2. **Scale Workload** which contains 500 randomly distributed queries with up to 4 joins

3. **JOB-Light Workload**, a simplified version of the queries in JOB [6], containing 70 queries with 4 joins apiece.

Note that the filter predicates and join predicates on all the queries of all the workloads are on the disjoint set of attributes.

5.5 GridSam Design Parameters Selection and Construction

We now discuss the settings of the grid overlay design and construction parameters for GridSam. There are two essential parameters to be determined after the determination of the ZSP-based ranking order, as discussed in Chapter 3 – (1) number of grid-attributes (i.e. size of *g-attr*), and (2) number of partitions per grid-attribute (p_i) – both of which have to be set while obeying a memory budget M. The methodology to choose both of these parameters has been described in Chapter 3. Here, we exemplify this with the Power dataset [4].

First, we decide the ordering of all 7 attributes of the Power dataset using their respective ZSP-scores. As described in Section 3.1, the ZSP-testing query sets for each attribute containing 3000 queries are generated. Further, they are augmented by both *LowDim* and *HiDim* testing workloads and evaluated on single materialized random uniform samples of size (S) 1000. The ZSP-scores of all 7 attributes from the Power dataset have been shown in Section 3.1.

Now given a fixed memory budget M, the question is that how many attributes should be accommodated in the ZSP-based ranking order for the grid overlay such that the overall Zero Sample Problem rate is minimized. In other words, which configuration (number of grid-attributes, memory budget) of the grid overlay fits the best to decrease the overall Zero Sample Problem rate. As described in Section 3.2, we build every configuration of grid overlay for the Power dataset in which the number of grid-attributes ranges from 1 to n_e (size of *e-attr* for Power dataset) in the decreasing order of their ZSP-scores. We evaluate Grid-testing query set containing 3000 queries for all n_e possible grid overlays on query-specific grid-based dynamic random samples of size (S) 1000 and obtain $G_{zsp}(k)$ (percentage of queries resulted into the Zero Sample Problem for the respective grid overlay containing k grid-attributes) where k ranges from 1 to n_e. Here, n_e is equal to 7 for the Power dataset.

Table 5.1: Construction Parameters of Grid Overlay on Power Dataset

n_dim	Partitions	ZSP
1	2837	48%
2	974 × 966	28.6%
3	283 × 281 × 12	34.5%
4	128 × 127 × 8 × 7	32%
5	**71 × 70 × 6 × 6 × 5**	**23%**
6	50 × 50 × 6 × 5 × 5 × 3	28.2%
7	43 × 42 × 5 × 4 × 4 × 3 × 2	28%

Table 5.2: Total Time Spent on Design Parameters Selection for the Construction of Grid Overlay on All Datasets

Dataset	Memory Budget (MB)	Total Time (CPU/GPU)
Power	15	42 / 3 secs
Forest	15	40 / 2.8 secs
IMDB	200	250 / 25 secs

Table 5.1 shows the number of partitions for each of the n_e grid overlay configurations and their respective $G_{zsp}(k)$. As shown in this table, the grid overlay with 5 grid-attributes yields minimum queries with Zero Sample Problem. In principle, these design parameters should give a minimum Zero Sample Problem for any memory budget M. We verified this observation on the Power dataset for different memory budgets such as 50 MB, 100 MB, 500 MB, and 1 GB. So, to efficiently determining these design parameters, a small memory budget is enough, and after that, the configuration can be used to build the grid overlay with bigger memory budgets. For determining grid overlay design parameters, we used the memory budget of 15 MB, which is minimally required to build the grid overlay on the Power dataset after considering that all *TIds* need to be stored in the grid overlay.

For Forest and IMDB dataset, the same procedure as described in Chapter 3 for single-table and multi-table can be followed respectively to determine the configuration parameters and build an "effective" grid overlay(s). Table 5.2 shows the memory budget and total time taken to find all the design parameters on CPU/GPU as described in Chapter 3 for all three datasets. Note that the *Total Time* denoted in the Table 5.2 includes following operations:

1. Determination of ranking/ordering of attributes for all tables present in the schema (only one for single table dataset).

2. Determination of ranking/ordering of tables (for multi-table schema).

Table 5.3: Table Wise Design Parameters of Grid Overlays on IMDB Dataset

Table	n_dim	Partitions	Memory Budget (MB)	Total Time
movie_keyword	2	20,000 × 4	32	4.1 secs
movie_companies	2	20,000 × 4	18	3.6 secs
cast_info	2	30,000 × 4	150	4.5 secs

3. Determination of the number of grid-attributes and number of partitions per grid attribute for all tables present in the schema (only one for single table dataset).

4. Determination of the number of tables among which memory budget needs to be divided based on $T_{zsp}(i)$ (ZSP-score of the table t_i) of each table (for multi-table schema).

The memory budgets shown for all the datasets are the minimum possible memory budgets to build the grid overlay for the respective dataset considering that all the *TIds* need to be stored in the grid overlay. As shown in Table 5.2, the design and construction of grid overlay are extremely cheap, as most of the tasks of design parameters selection are data-parallel tasks, and GPU can be used to perform those.

Table 5.3 shows the final design parameters of the grid overlays, selected for the tables in IMDB dataset. The minimum possible memory budget (considering the number of rows of all tables) of 200 MB is assigned to the IMDB dataset (among all tables). Based on $T_{zsp}(i)$ (arithmetic mean of all ZSP-scores of all attributes of table T_i) of each table, the memory budget is divided, and the ranking/ordering of tables has been determined. Further, as described in Section 3.7 and 3.8, it is determined that the Zero Sample Problem rate has been decreased the most with 2-dimensional grid overlays for tables *movie_keyword*, *movie_companies* and $cast\_info$ for the Grid-testing query set. The grid overlays of all three tables contain one local grid-attribute and one foreign grid-attribute. For example, for the *movie_keyword* table, the local grid-attribute *keyword_id* has the data-domain of size 134,170, and the foreign grid-attribute *production_year* from *title* table has small data-domain size, but skewed distribution for some values. Thus, the more number of partitions are assigned to *keyword_id* compared to *production_year*. The table *title* joins to *movie_keyword* table with PK-FK relationship. Note that the *Total Time* shown in Table 5.3 denotes only the construction time of the grid overlay for the respective table after determining the design parameters of the grid overlay of the respective table. As it can be seen in the Table 5.3, the grid overlay construction time is quite low (a few seconds), majorly because the entire IMDB dataset, including all tables, can be fitted into GPU-memory, and GPU-based grid overlay construction is carried out.

Table 5.4: Grid Overlay Construction Time (seconds)

Dataset	15 MB	50 MB	100 MB	500 MB	1 GB
Power (5-dim)	4.3	4.7	5.1	8	12.2
Forest (5-dim)	4.1	4.6	5.0	8	12.1

Table 5.5: Grid Overlay Construction Time (milliseconds) on GPU

Dataset	15 MB	50 MB	100 MB	500 MB	1 GB
Power (5-dim)	5	23	36	161	371
Forest (5-dim)	4	17	33	162	380

We evaluate grid overlay construction for different memory budgets such as 15 MB, 50 MB, 100 MB, 500 MB, and 1 GB after determining design parameters. Table 5.4 and Table 5.5 show the construction time of the grid overlays after the design parameters selection for different memory budgets without using GPU and using GPU, respectively. These *times* include the operations such as (1) CPU/GPU memory allocation for the multi-dimensional array $\mathcal{G}$ of grid overlay, (2) Populating the grid cells with *TIds*, (3) Copying the multi-dimensional array $\mathcal{G}$ of grid overlay to GPU-memory. It can be seen that constructing a grid overlay even for higher memory budgets is much cheaper and completely viable.

While estimating, if any query falls into the Zero Sample Problem, GridSam uses estimation from the *normal* Postgres estimator for the respective query. Here, the *normal* Postgres estimator is configured with all the parameters suggested by the PGTune [13] tool including the number of histogram buckets and the size of the MCV array.

Chapter 6

Experimental Evaluation

Having described the framework and compile-time efforts, we now move on to present the quantitative results of our experiments regarding the estimation accuracy and run-time overheads.

6.1 Accuracy

The *q-error* distributions for GridSam with different memory budgets are shown in Table 6.1 for Power dataset [4], and in Table 6.2 for Forest dataset [4]. In both the tables, the two categories (Top-Half and Bottom-Half) show the *q-error* distribution for both of the respective single-table test workloads (*HiDim* and *LowDim*) containing queries with the number of range predicates {5,6,7} and {2,3,4} on Power and Forest dataset, respectively. Note that if a query does not contain any filter predicates on *g-attr*, then grid-based dynamic sampling performed by GridSam for a query will be equivalent to RMS. Combined from both GridSam and its fallback option of RMS, in Table 6.1 and Table 6.2, we also showed the percentage of queries from the total queries of the workloads as ZSP which resulted in Zero Sample Problem (not able to find any qualifying samples).

Note that, for the Power dataset, there were 99% of the queries from *HiDim* workload and 90% of the queries from *LowDim* workload, eligible for grid-based dynamic sampling, and the other queries, which do not contain any filter predicate on any grid-attribute, had to fall back to RMS. For the Forest dataset, there were 98% of the queries from *HiDim* workload and 84% of the queries from *LowDim* workload, eligible for grid-based dynamic sampling. In both tables, *Q-Dim* denotes the number of filter predicates present in each query of the respective test workload. Note that if any query falls into the Zero Sample Problem (for GridSam and RMS), we take the estimation from the well-tuned *normal* Postgres estimator for the respective query. Because of this reason, the tail distribution of *q-error* ($99thPerc.$ and Max) is still large

Table 6.1: *Q-Error* Distribution of *HiDim* (Top-Half) and *LowDim* (Bottom-Half) workloads on Power dataset

Technique	Median	95th Perc.	99th Perc.	Max	Mean	ZSP
GridSam (15 MB)	1.19	3.6	11.7	275	2.1	22%
GridSam (50 MB)	1.13	2.9	8.2	231	1.80	17%
GridSam (100 MB)	1.11	2.75	8.17	191	1.61	14%
GridSam (500 MB)	1.07	2.45	8.15	191	1.57	12%
GridSam (1 GB)	1.06	2.28	8.03	191	1.53	10%
RMS	1.31	13.5	61	802	4.7	61%
LWM	1.51	7.4	21.6	116	2.7	-
MSCN	1.6	10.5	34	723	4.04	-
NARU	1.08	2.39	4.15	133	1.4	-
Postgres	1.5	14	65	1798	5.5	-
GridSam (15 MB)	1.10	3.12	10.41	191	1.76	8.3%
GridSam (50 MB)	1.06	2.42	8.85	191	1.67	6.4%
GridSam (100 MB)	1.04	2.22	6.7	189	1.62	6.1%
GridSam (500 MB)	1.03	2.0	5.03	189	1.54	4.7%
GridSam (1 GB)	1.02	1.98	5.01	189	1.54	4.7%
RMS	1.12	6	34	998	3.7	30%
LWM	1.26	3.8	11.5	154	2	-
MSCN	1.19	7	32.3	1184	3.5	-
NARU	1.04	1.9	2.8	20	1.2	-
Postgres	1.14	6.8	40	1005	4.3	-

Table 6.2: *Q-Error* Distribution of *HiDim* (Top-Half) and *LowDim* (Bottom-Half) workloads on Forest dataset

Technique	Median	95th Perc.	99th Perc.	Max	Mean	ZSP
GridSam (15 MB)	1.09	3.8	9.67	46	1.53	33.9%
GridSam (50 MB)	1.05	3.6	9.5	46	1.48	32%
GridSam (500 MB)	1.04	3.1	9.27	46	1.26	27%
GridSam (1 GB)	1.03	2.9	8.46	29	1.53	26%
RMS	1.18	6.8	25	663	2.6	72%
LWM	1.2	4.3	11.5	53	1.86	-
MSCN	1.8	8	24.2	950	4	-
NARU	1.12	2.8	5.1	51	1.43	-
Postgres	1.18	6.2	19.9	663	2.5	-
GridSam (15 MB)	1.06	2.46	6.7	86	1.27	19.7%
GridSam (50 MB)	1.04	2.34	4.3	86	1.21	17%
GridSam (500 MB)	1.04	2.12	4	26	1.18	15%
GridSam (1 GB)	1.03	1.98	4	26	1.18	13%
RMS	1.08	3.5	10.5	2472	2.8	41%
LWM	1.2	2.7	5.6	64	1.52	-
MSCN	1.4	6.2	17.8	406	2.8	-
NARU	1.06	2.2	3.8	28	1.2	-
Postgres	1.06	3.2	9.6	2472	2.65	-

for GridSam and RMS for both datasets as shown in Table 6.1 and Table 6.2. Note that the sample size (S) is set to 1000 for the evaluation.

As shown in Table 6.1 and Table 6.2, GridSam achieves competitive accuracy compared to learned-estimators (LWM, MSCN, and NARU), and also achieves better accuracy compared to traditional estimators (RMS and Postgres). The reason for NARU outperforming GridSam is that NARU requires a much higher training time. Note that, for this evaluation, NARU is trained for nearly 3000 seconds, and GridSam only took 3 seconds even for higher memory budgets. As already shown in Table 1.1 of Introduction, GridSam can outperform NARU with regards to the accuracy if NARU is trained for lower training efforts (i.e., 15 secs, 55 secs, or 200 secs) which are still orders-of-magnitude higher than GridSam's building effort. With increased memory budgets, GridSam seems to be performing further better. The percentage of queries with Zero Sample Problem (denoted as ZSP) in the workload has also decreased sharply compared to RMS because of grid-based dynamic sampling.

Tables 6.3, 6.4 and 6.5 show the estimation errors using percentile distribution of the *q-error* for JOB-Light, Scale and Synthetic workloads, respectively. These three workloads are based on IMDB dataset [5]. Note that GridSam with the memory budget of 200 MB is used for this evaluation which is minimal for IMDB dataset. We have also evaluated GridSam on these three workloads with different memory budgets, and it is observed that there is no significant improvement over accuracy.

For JOB-Light, Scale, and Synthetic workloads, the tail distribution of GridSam has suffered mainly because of relying upon one-dimensional histogram-based cardinality estimator of Postgres [14] on Zero Sample Problem. As the Zero Sample Problem is not frequent in GridSam, the distributions till 95th Percentile is quite better than every other model for all three workloads. Note that the MSCN model is trained on the queries with up to 2 joins, and the End-To-End model is trained on the queries with up to 3 joins. The Synthetic workload has been generated in the same way as the training sets of MSCN and End-To-End, which is the best-case scenario for both the models when evaluating upon Synthetic workload. Even for the Synthetic workload, GridSam achieves better accuracy compared to MSCN and End-To-End till 99th percentile as shown in Table 6.5. IBJS could not achieve good accuracy mainly because of the higher Zero Sample Problem rate on all three workloads compared to GridSam. The per-table sample size (S) is 1000 for the evaluations of GridSam and IBJS, and the per-table bitmap sample (input features) size is also 1000 for both MSCN and End-To-End model. We have also evaluated IBJS with a per-table sample size of 10,000. Even with 10,000 samples per table, IBJS could not achieve better accuracy because the higher sample size could not bring down the Zero Sample Problem rate.

Table 6.3: *Q-Error* Distribution for JOB-Light Workload

Model	median	90th	95th	99th	max	mean
GridSam	**1.25**	**2.74**	**3.65**	**5.04**	**5.41**	**1.63**
IBJS	1.8	35.6	270	8167	9654	340
MSCN	3.82	78.4	362	927	1110	57.9
E2E	3.51	48.6	139	244	272	24.3

Table 6.4: *Q-Error* Distribution for Scale Workload

Model	median	90th	95th	99th	max	mean
GridSam	**1.1**	**2.18**	**3.57**	**23.64**	233863	486
IBJS	1.34	15.6	39	638	223232	480
MSCN	1.38	49	240	1177	5166	51
E2E	1.42	37.3	125	345	**1813**	**26.3**

Table 6.5: *Q-Error* Distribution for Synthetic Workload

Model	median	90th	95th	99th	max	mean
GridSam	**1.08**	**2.07**	**3.69**	**23.64**	2024	5.9
IBJS	1.21	5.28	13	100	2056	9
MSCN	1.18	3.32	6.84	30.51	1322	2.89
E2E	1.18	3.19	6.05	24.5	**323**	**2.81**

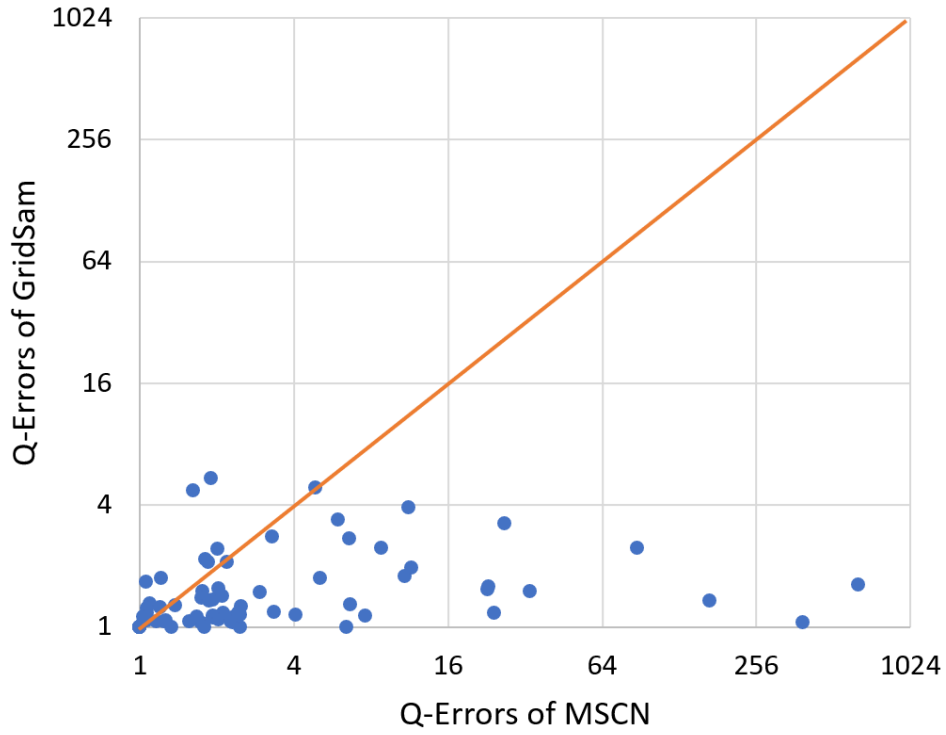


Figure 6.1: Query Wise Comparison between GridSam and MSCN for JOB-Light Workload

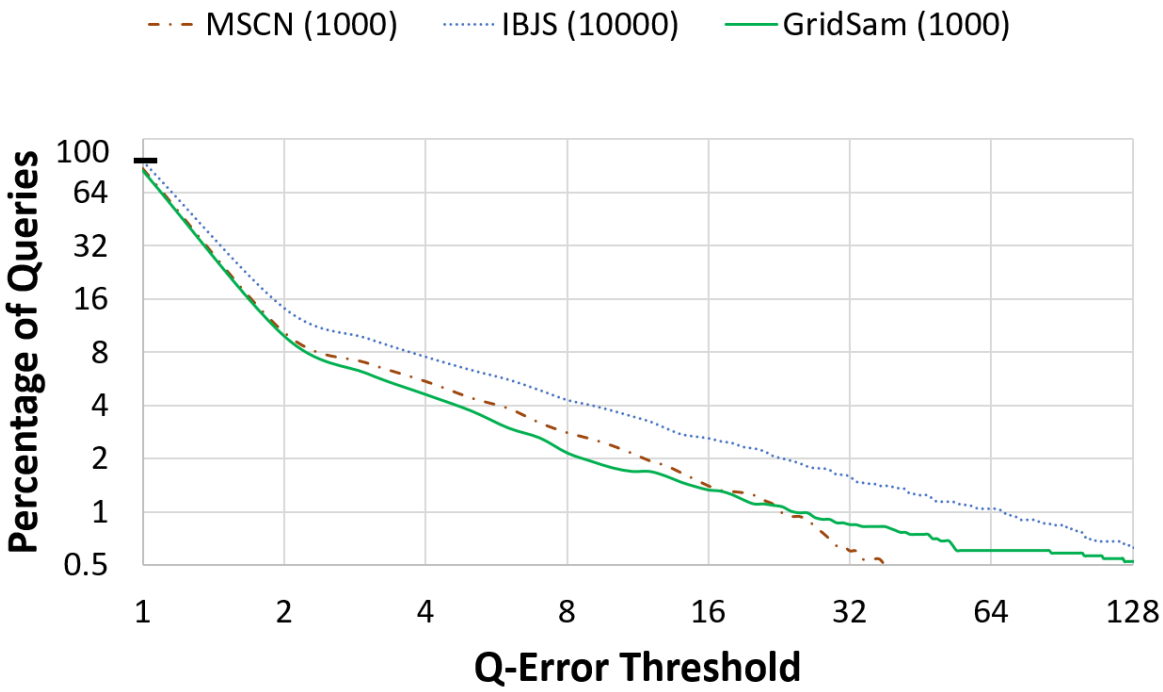


Figure 6.2: Percentage of Queries from the Synthetic Workload having *q-error* > THRESHOLD.

Figure 6.1 shows the query by query comparison of the JOB-Light workload between GridSam (with sample size 1000) and MSCN. It is clear from the figure that there are many points on the right side of the diagonal line, which indicates that MSCN has large estimation errors for the queries that have relatively low estimation errors for GridSam. Because of the Zero Sample Problem, there are few queries in the workload for which GridSam has high *q-error* relative to MSCN as shown in Figure 6.1. Figures 6.2 mainly depicts the percentage of queries from the Synthetic workload lies within the *q-error* threshold depicted on x-axis. The number inside the brackets in the legends of this figure denotes the sample size for IBJS and GridSam. For MSCN, it denotes the size of the bitmap samples. As shown in Figure 6.2, GridSam with 1000 samples performs competitive to MSCN for Synthetic workload which is the best-case scenario for MSCN. Also, IBJS could not achieve better accuracy than GridSam even with 10,000 samples. Thus, GridSam needs a much less number of samples than RMS to achieve good accuracy.

6.2 Estimation Time

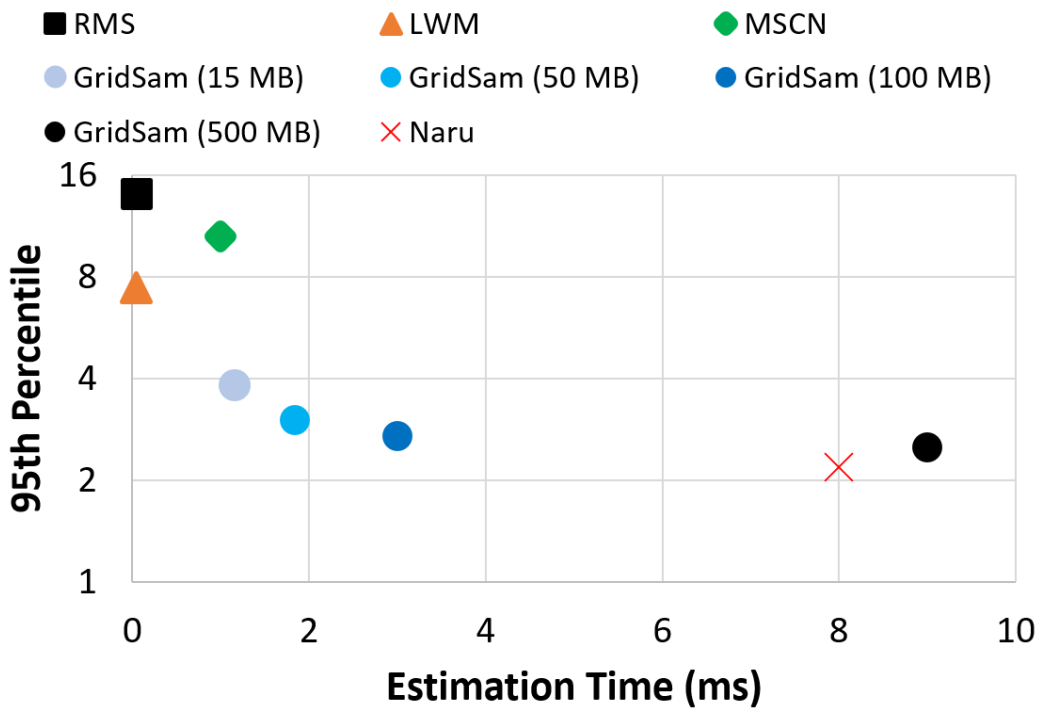


Figure 6.3: Estimation Time and Accuracy Comparison for Power Dataset

As shown in Figure 6.3, average estimation time (in milliseconds) of *HiDim* and *LowDim* query workloads and average 95^{th} Percentile of the *q-error* distribution of both the workloads

are represented in x-axis and y-axis, respectively. GridSam is presented with different memory configurations denoted in brackets. The estimation time for RMS is the smallest compared to GridSam and other estimators. While having much lesser estimation time and negligible space overhead, RMS incurs large estimation errors. On the other hand, GridSam with various memory budgets, which generates grid-based dynamic samples, performs better in terms of accuracy (or competitive to NARU) compared to all other techniques. GridSam may appear to have large estimation times in a relative sense, these times are quite practical from an absolute perspective since they are in milliseconds, minuscule in comparison to query execution times. After 100 MB of memory budget, GridSam does not improve much upon accuracy for memory budgets of 500 MB and 1 GB while incurring more estimation time (we did not show GridSam (1 GB) as it does not improve accuracy and incurs more estimation time).

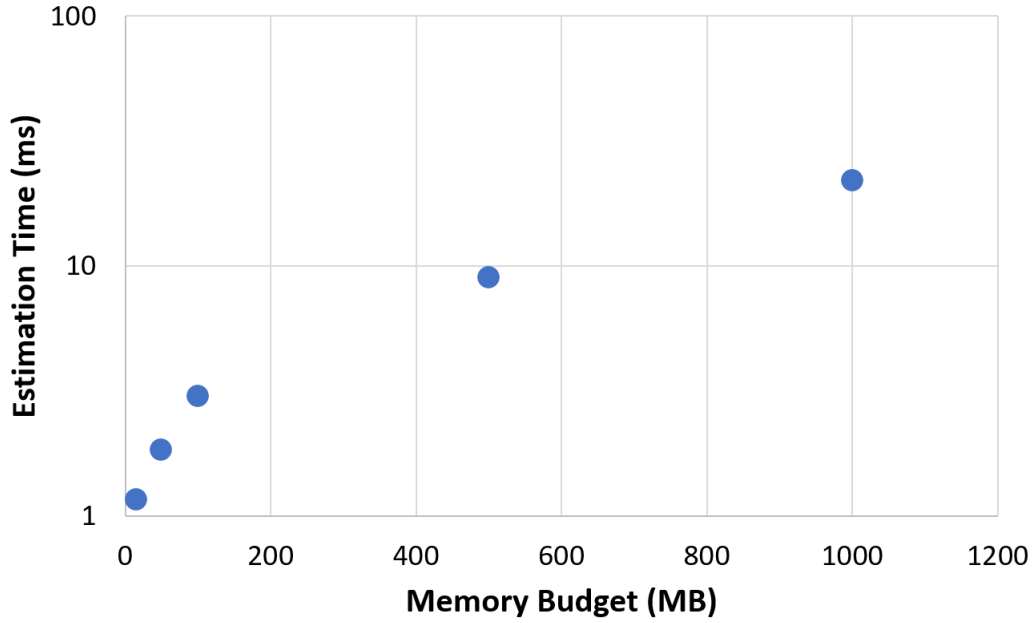


Figure 6.4: Estimation Time of GridSam with Different Memory Budgets for Power Dataset

Figure 6.4 shows the average estimation time of *HiDim* and *LowDim* query workloads for Power dataset on grid overlays with different memory budgets. The estimation time increases exponentially with the exponential increase in memory budget as the number of elements in the multi-dimensional array $\mathcal{G}$ of grid overlay will also increase exponentially. GPU can simultaneously compute the number of elements of the grid array equal to the maximum number of active threads in the GPU. So if the size of the array is larger than the maximum number of active threads possible in a GPU, the GPU scheduler will divide the array into multiple batches which run serially. That is why an increase in the estimation time with an increase in the array

size as the number of serially executed batches increases. Still, the estimation times for even larger memory budgets such as 500 MB and 1 GB are not more than tens of milliseconds. Note that the sample size (S) of 1000 is used for the comparisons.

However, estimation time can be brought down using multiple GPU-devices, almost by a factor of x, if we use x number of GPU devices, with the same implementation of GPU algorithms described in Section 4.1. The strategy to use multiple GPU devices to reduce estimation time is quite apparent in GridSam compared to learned-models. For example, in NN-based learned-models, the values of the neurons in a layer is dependent on the values of the neurons in a previous layer, and hence, there are dependencies. While in GridSam, the array of grid cells just needs to be divided equally among all GPU devices without any dependencies.

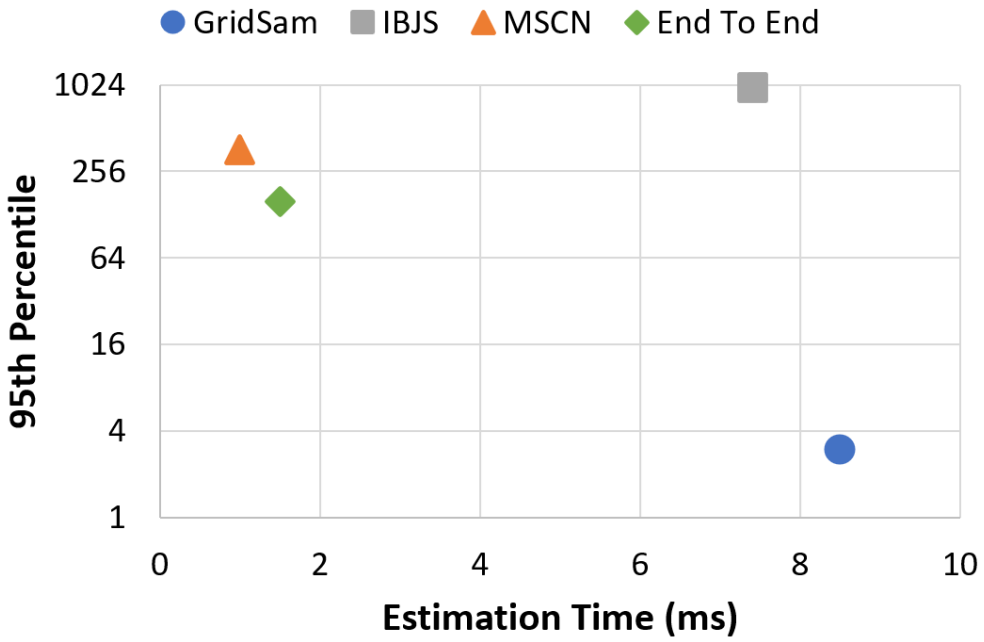


Figure 6.5: Estimation Time and Accuracy Comparison for IMDB Dataset on JOB-Light Workload

As shown in Figure 6.5, GridSam with 1000 samples (denoted in brackets of each technique) takes more estimation time compared to all other techniques mainly because of more number of tables (4-5) in each query of JOB-Light workload. However, it achieves significantly higher accuracy compared to all other techniques. Note that the estimation time for GridSam is practically small, minuscule in comparison to query execution times, while having a significant relative difference compared to MSCN and End-to-End models. IBJS with a sample size of 1000 incurs estimation time similar to GridSam but mostly suffers over accuracy because of the high Zero Sample Problem rate. The estimation times of the learning-based models do not

Table 6.6: Impact of Choosing "Right" Grid-Attributes

Technique	95th Percentile	ZSP
GridSam	3.1	17%
GridSam *	9.2	47%

depend on the number of tables in the query, and hence, they are quite low.

6.3 Impact of ZSP-based Heuristic

To select the "right" grid-attributes, ranking order produced based on ZSP-scores of attributes using the procedure described in Chapter 3 is essential as it can have a significant impact on accuracy. To validate this claim, for Power dataset [4], we chose 5 random attributes from total 7 attributes and divide the memory budget equally instead of following the described methodologies in Sections 3.1, 3.2 and 3.3. The *HiDim* query workload on the Power dataset is used for this experiment. We compared the accuracy of this new grid overlay (GridSam *) with the previous one, which is built using described design methodologies. As shown in Table 6.6, there is a significant increment in 95th Percentile of the *q-error* distribution and ZSP (percentage of queries with Zero Sample Problem) from GridSam to GridSam * for the sample size (S) of 1000. This suggests that choosing "right" grid-attributes can significantly impact the accuracy.

Chapter 7

Conclusions and Future Work

We revisited the 4-decade-year-old long-standing problem of cardinality estimation in RDBMS. The traditional estimators based on statistical models that are used in current industrial database engines, which mainly use histogram-based or sampling-based techniques, incur large estimation errors. On the other side, recent learned-estimators based on machine-learning/deep-learning models have shown promising accuracy in the literature. However, these learned-estimators have their own limitations regarding significant training effort, inability to handle dynamic data-updates, and generalization to unseen queries. Specifically, they can cause large estimation errors in the environments with frequent data-updates because of their significant training and tuning efforts. The learned-estimators need to be retrained and re-tuned to incorporate the updated data into model. In other words, they demand significant "shutdown" time to regain the accuracy.

In this work, we presented *GridSam* – a Grid-based Dynamic Sampling technique for result cardinality estimation. It augments random sampling with histograms to rectify the limitations of each other. GridSam partitions the data-space on "critical" attributes by imposing the grid overlay. Subsequently, it performs the query-specific sampling from the confined query sample space provided by the grid overlay, leading to a reduced Zero Sample Problem rate. Further, We proposed a greedy methodology targeted towards reducing the Zero Sample Problem occurrence to determine the grid overlay's design parameters. This methodology obeys the given memory budget while determining the design parameters. The design parameters of the grid overlay include (i) set of "critical" attributes used as the dimensions of the grid overlay to partition the data-space, and (ii) the number of partition boundaries for each dimensional attribute. Further, we showed a methodology to use grid overlays for multiple tables in the schema with the additional design parameters and learned-indexes for join sampling. Next, we presented the estimation procedures for efficiently performing grid-based dynamic sampling for both single-

table and multi-table queries using the parallel resources of GPU. At a high level, GridSam aims to achieve good accuracy by reducing the Zero Sample Problem rate by performing dynamic sampling from the confined query-specific region provided by the grid overlays.

With extensive empirical evaluation over multiple real-world datasets, we showed that GridSam could achieve superior accuracy than traditional estimators and achieve highly competitive accuracy compared to the state-of-the-art learned-estimators. The reason being a reduced Zero Sample Problem rate because of the sampling from the confined query-specific region provided by the grid overlays. The experimental evaluation showed that the overheads of the design parameters selection and the construction of GridSam are quite low (a few seconds) using GPU compared to GPU-based training and tuning of the learned-estimators (several minutes to hours). Specifically, we showed that GridSam could achieve competitive accuracy to NARU (an unsupervised learned-estimator) even with orders-of-magnitude less building time (off-line efforts) by trading-off memory usage. Thus, GridSam is more amenable to being used in environments with frequent data-updates than learned-estimators. In other words, GridSam demands orders-of-magnitude less rebuilding/retraining time to regain the accuracy compared to learned-estimators to incorporate major data-updates. However, GridSam incurs much higher memory usage, which we assumed to have sufficiently available in the system.

GridSam also uses the GPU to perform data-parallel tasks on the grid overlay during estimation to decrease the overall estimation time. GridSam incurred a little higher estimation time compared to other techniques but reasonable (within tens-of-milliseconds) from the absolute perspective.

7.1 Future Work

Following are the future research avenues regarding the construction and design of GridSam:

1. We plan to design more sophisticated GPU-based algorithms for the data-parallel tasks of the estimation procedure to reduce the estimation time even lower.

2. Current design of GridSam does not support the GPU-based construction of the grid overlays for multi-table schema if the foreign tables cannot fit into GPU-memory. We plan to provide this support to GridSam for efficiently determining the tables for which the grid overlays need to be built along with their design parameters.

3. Currently, we are storing tuple identifiers (*TIds*) of each tuple (row) of the table into the respective grid cell, making GridSam consume most of the memory from the provided memory budget. Thus, we are also planning to work on GridSam design to support much lower memory usage.

4. In this work, we have only focused on numeric-typed attributes. We plan to extend our work to string-typed attributes with respect to grid overlay design and construction. Moreover, we have only focused on queries with AND operators in this work; we plan to extend this technique to support other operators such as OR and Group-By and show performance evaluation.

Bibliography

[1] CPP Programming Language. https://en.cppreference.com/w/cpp/language. 52

[2] Containers in Standard Template Library. https://www.cplusplus.com/reference/stl/. 34

[3] Cuda Toolkit Documentation. https://docs.nvidia.com/cuda/index.html. 50

[4] UCI Machine Learning Repository. https://archive.ics.uci.edu/ml/index.php. 2, 12, 30, 34, 51, 54, 58, 67

[5] IMDB Dataset. https://www.imdb.com/interfaces/. 2, 3, 5, 13, 16, 22, 40, 51, 61

[6] JOB Benchmark Queries. https://github.com/gregrahn/join-order-benchmark/. 51, 54

[7] Keras: the Python deep learning API. https://keras.io/. 52

[8] Learned Cardinalities in PyTorch. https://github.com/andreaskipf/learnedcardinalities. 52, 53

[9] RMI (Recursive Model Indexes). https://github.com/learnedsystems/RMI. 40

[10] Neural Relation Understanding. https://github.com/naru-project/naru. 53

[11] Understanding Optimizer Statistics with Oracle Database 18c. https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-stats-concepts-0218-4403739.pdf. 15

[12] NVIDIA CUDA SDK - Data-Parallel Algorithms. https://www.nvidia.com/content/cudazone/cuda_sdk/Data-Parallel_Algorithms.html. 44

[13] PGTune: Tuning Parameters of your Postgres System. https://pgtune.leopard.in.ua/. 51, 57

[14] PostgreSQL: The world's most advanced open source database. https://www.postgresql.org. 6, 11, 12, 51, 61

[15] How the Planner Uses Statistics. https://www.postgresql.org/docs/current/row-estimation-examples.html. 15

[16] Python Programming Language. https://www.python.org/. 52

[17] Statistics in Microsoft SQL Server 2017. https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-2017. 15

[18] Nvidia Tesla V100 GPU. https://www.nvidia.com/en-gb/data-center/tesla-v100/. 50

[19] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. Learning-based Query Performance Modeling and Prediction. In *Proceedings of the 28th IEEE International Conference on Data Engineering, ICDE 2012, April 1 2012 - April 5 2012, Arlington, Virginia, USA*, pages 390–401, 2012.

[20] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano STHoles: A Multidimensional Workload-Aware Histogram. In *Proceedings of the 2001 ACM International Conference on Management of Data, SIGMOD 2001, May 21 2001 - May 24 2001, Santa Barbara, CA, USA*, pages 211–222, 2001. 3, 16

[21] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Random Sampling for Histogram Construction: How much is enough? In *Proceedings of the 1998 ACM International Conference on Management of Data, SIGMOD 1998, June 2 1998 - June 4 1998, Seattle, Washington, USA*, pages 436–447, 1998. 33

[22] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in databases*, 4(1-3):1–294, 2012. 3, 16

[23] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. Selectivity Estimation for Range Predicates using Lightweight Models. *PVLDB*, 12(9):1044–1057, 2019. vii, 4, 12, 20, 21, 23, 24, 50, 51, 52, 53

[24] Todd Eavis, and Alex Lopez. Rk-hist: an r-tree based histogram for multi-dimensional selectivity estimation. In *Proceedings of the 16th ACM Conference on Information and*

Knowledge Management, CIKM 2007, November 6 2007 - November 10 2007, Lisbon, Portugal, pages 475–484, 2007. 16

[25] Cristian Estan, and Jeffrey F. Naughton. End-biased Samples for Join Cardinality Estimation. In *Proceedings of the 22nd IEEE International Conference on Data Engineering, ICDE 2006, April 3 2006 - April 8 2006, Atlanta, GA, USA*, pages 20, 2006. 3, 17

[26] Sumit Ganguly, Phillip B. Gibbons, Yossi Matias, and Abraham Silberschatz. Bifocal Sampling for Skew-Resistant Join Size Estimation. In *Proceedings of the 1996 ACM International Conference on Management of Data, SIGMOD 1996, June 14 1996 - June 19 1996, Montreal, Quebec, Canada*, pages 271–281, 1996. 3, 17

[27] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proceedings of the 25th IEEE International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 592–603, 2009.

[28] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: Masked Autoencoder for Distribution Estimation. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, July 6 2015 - July 11 2015, Lille, France*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 881–889, 2015. 53

[29] Jean Dickinson Gibbons. *Nonparametric Methods for Quantitative Analysis (3rd Ed.)*. American Sciences Press, USA, 1996. ISBN 0935950370. 8

[30] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, 2005. 3, 16

[31] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *Proceedings of the 2020 ACM International Conference on Management of Data, SIGMOD 2020, June 14 2020 - June 19 2020, Portland, OR, USA*, pages 1035–1050, 2020. 4, 20, 21, 25

[32] Diego Havenstein, Peter Lysakovski, Norman May, Guido Moerkotte, and Gabriele Steidl. Fast Entropy Maximization for Selectivity Estimation of Conjunctive Predicates on CPUs and GPUs. In *Proceedings of the 23rd International Conference on Extending Database*

Technology, EDBT 2020, March 30 2020 - April 02 2020, Copenhagen, Denmark, pages 546–554, 2020. 4, 20

[33] Rojeh Hayek and Oded Shmueli. Improved Cardinality Estimation by Learning Queries Containment Rates. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, March 30 2020 - April 02 2020, Copenhagen, Denmark*, pages 157–168, 2020. 4, 20

[34] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *Proceedings of the 1997 ACM International Conference on Management of Data, SIGMOD 1997, May 13 1997 - May 15 1997, Tucson, Arizona, USA*, pages 171–182, 1997. 1

[35] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. DeepDB: Learn from Data, not from Queries! *PVLDB*, 13(7): 992–1005, 2020. 25

[36] Yannis E. Ioannidis. The History of Histograms (abridged). In *Proceedings of the 29th International Conference on Very Large Data Bases Technology, VLDB 2003, September 9 2003 - September 12 2003, Berlin, Germany*, pages 19–30, 2003. 3, 16

[37] Christopher M. Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable Approximate Query Processing with the DBO Engine. In *Proceedings of the 2007 ACM International Conference on Management of Data, SIGMOD 2007, June 12 2007 - June 14 2007, Beijing, China*, pages 725–736, 2007. 1

[38] Martin L. Kersten, Stratos Idreos, Stefan Manegold, and Erietta Liarou. The researcher's guide to the data deluge: querying a scientific database in just a few seconds. *PVLDB*, 4 (12):1474–1477, 2011. 1

[39] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models. *PVLDB*, 10(13):2085–2096, 2017. 4, 20

[40] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, January 13 2019 - January 16 2019, Asilomar, CA, USA*, 2019. vii, 4, 12, 13, 20, 21, 22, 40, 50, 51, 52, 53

[41] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. SOSD: A Benchmark for Learned Indexes. *CoRR*, abs/1911.13014, 2019. 37, 40

[42] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 ACM International Conference on Management of Data, SIGMOD 2018, June 10 2018 - June 15 2018, Houston, TX, USA*, pages 489–504, 2018. 8, 11, 17, 37, 40, 42, 52

[43] Hai Lan, Zhifeng Bao, and Yuwei Peng A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration. *Data Science and Engineering*, 6(1):86–101, 2021. 3, 16, 18

[44] Viktor Leis, Alfons Kemper, and Thomas Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, April 8 2013 - April 12 2013, Brisbane, Australia*, pages 38–49, 2013. 35, 40

[45] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015. 2, 3, 16, 51

[46] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality Estimation Done Right: Index-Based Join Sampling. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, January 8 2017 - January 11 2017, Chaminade, CA, USA*, 2017. vii, 3, 8, 11, 13, 17, 18, 19, 20, 35, 36, 37, 40, 52

[47] Xudong Lin, Xiaoning Zeng, Xiaowei Pu, and Yanyan Sun. A Cardinality Estimation Approach Based on Two Level Histograms. *Journal of Information Science and Engineering*, 31(1):1733–1756, 2015. 16

[48] Richard J. Lipton, Jeffrey F. Naughton, and Donovan A. Schneider. Practical Selectivity Estimation through Adaptive Sampling. In *Proceedings of the 1990 ACM International Conference on Management of Data, SIGMOD 1990, May 23 1990 - May 25 1990, Atlantic City, NJ, USA*, pages 1–11, 1990. 3, 17

[49] Ryan Marcus and Olga Papaemmanouil. Towards a Hands-Free Query Optimizer through Deep Learning. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems*

Research, CIDR 2019, January 13 2019 - January 16 2019, Asilomar, CA, USA, 2019. 4, 20

[50] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *PVLDB*, 2(1):982–993, 2009. 6, 12, 50

[51] M. Muralikrishna and David J. DeWitt. Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries. In *Proceedings of the 1988 ACM International Conference on Management of Data, SIGMOD 1988, June 1 1988 - June 3 1988, Chicago, Illinois, USA*, pages 28–36, 1988. 3, 16

[52] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. Cost-Guided Cardinality Estimation: Focus Where it Matters. In *Proceedings of the 36th IEEE International Conference on Data Engineering, ICDE Workshops 2020, April 20 2020 - April 24 2020, Dallas, TX, USA*, pages 154–157, 2020. 4, 20, 21

[53] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984. 9, 17, 34

[54] Frank Olken and Doron Rotem. Random sampling from database files: A survey. In *Proceedings of the 5th International Conference on Statistical and Scientific Database Management, SSDBM 1990, April 3 1990 - April 5 1990, Charlotte, NC, USA*, volume 420 of *Lecture Notes in Computer Science*, pages 92–111, 1990. 3

[55] Viswanath Poosala and Yannis E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proceedings of the 23rd International Conference on Very Large Data Bases Technology, VLDB 1997, August 25 1997 - August 29 1997, Athens, Greece*, pages 486–495, 1997. 3, 16

[56] Ji Sun and Guoliang Li. An End-to-End Learning-based Cost Estimator. *PVLDB*, 13(3): 307–319, 2019. 4, 13, 20, 21, 52, 53

[57] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. Lightweight Graphical Models for Selectivity Estimation Without Independence Assumptions. *PVLDB*, 4(11):852–863, 2011. 15

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Proceedings of the 31st Conference on Neural Information Processing Systems, NIPS 2017, December 4 2017 - December 9 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. 25, 53

[59] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. Join Size Estimation Subject to Filter Conditions. *PVLDB*, 8(12):1530–1541, 2015. 3, 17, 18

[60] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. Are We Ready For Learned Cardinality Estimation? *CoRR*, abs/2012.06743, 2020. 4, 23, 24, 26, 50

[61] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *Proceedings of the 29th IEEE International Conference on Data Engineering, ICDE 2013, April 8 2013 - April 12 2013, Brisbane, Australia*, pages 1081–1092, 2013. 5

[62] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep Unsupervised Cardinality Estimation. *PVLDB*, 13(3):279–292, 2019. vii, 4, 12, 20, 25, 53

[63] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, Mengxia Zhu CS2: a new database synopsis for query estimation. In *Proceedings of the 2013 ACM International Conference on Management of Data, SIGMOD 2013, June 22 2013 - June 27 2013, New York, NY, USA*, pages 469–480, 2013. 3, 17, 18, 19