Clustering-Initialized Adaptive Histograms and Probabilistic Cost Estimation for Query Optimization

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik

des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Andranik Khachatryan

aus Jerewan

Tag der mündlichen Prüfung:	30.04.2012
Erster Gutachter:	Prof. Dr.-Ing. Klemens Böhm
Zweiter Gutachter:	Prof. Dr. Bernhard Seeger

Acknowledgments

First of all, I would like to express my deepest gratitude to Prof. Klemens Böhm, who supervised this thesis, for the invaluable insight he brought into this work. His dedication and work ethics served as an example for me and everyone else in the group. Without his encouragement, willingness to discuss and patience I would not have made it.

I would like to thank also Prof. Bernhard Seeger, my second advisor, for his willingness to undertake the ungratifying task of reading and commenting on the thesis.

I am greatly obliged to Emmanuel Müller with whom we worked together for the last year and a half. He brought invaluable insight and experience into the topic. Working with him was both productive and enjoyable.

Peter J. Haas gave me a draft of his review on histograms. This review greatly influenced my view on the whole subject. The chapter on histograms in this thesis follows Peter's conventions on histogram categorization.

My family had all the patience and showed all the support during those years that I was doing this work. I would not have gotten here without them.

Last but not least, I want to thank my friend and office-mate Björn-Oliver Hartmann with whom we spent long hours in discussions, about our research topics and not only. We had quite different perspective on a variety of things, and our discussions were very interesting. He was and is always willing to help and support.

Contents

List of Figures

List of Tables

List of Tables

1 Zusammenfassung der Arbeit

Datenbanken ermöglichen es ihren Nutzern, deklarative Anfragen zu stellen. Der Nutzer beschreibt die Daten, die er von der Datenbank erhalten möchte, und ist davon befreit zu spezifizieren, wie die Daten gewonnen werden sollen.

Es existieren viele Alternativen eine Anfrage abzuarbeiten. Diese Alternativen werden auch als Ausführungspläne bezeichnet. Eine Komponente des Datenbank-Management Systems, der sogenannte Anfrageoptimierer, entscheidet, wie ein effizienter Ausführungsplan gewählt wird. Bisher nutzen Optimierer kostenbasierte Optimierungen. Näherungen für die Ausführungskosten werden für viele Ausführungspläne berechnet und ein Ausführungsplan mit geringen Kosten wird ausgewählt. Die Ausführungskosten sind eine gewichtete Funktion der Systemressourcen, die gebraucht werden, um die Anfrage auszuführen. Beispiele solcher Systemressourcen sind CPU-Zeit oder die Anzahl von Ein- und Ausgabeoperationen.

Um eine angebrachte Kostenschätzung zu erreichen, benötigt der Optimierer eine Schätzung der Größe von Teilanfragen. Dies ist zum Beispiel dann wichtig, wenn die Verbund-Reihenfolge (engl. join order) von Relationen bestimmt wird. Um die Größe von Teilanfragen zu schätzen, muss der Optimierer die Selektivität von Anfrage-Prädikaten kennen.

Die wesentliche Datenstruktur zur Bestimmung von Selektivitäten in Datenbanken sind Histogramme. Selbstoptimierende Histogramme sind eine Klasse von Histogrammen, die die Ergebnisse von bereits existierenden Anfragen nutzen, um sich selbst anzupassen. Dies ist eine Art des überwachten Lernens. Selbstoptimierende Histogramme sind in der Lage, Initialisierungskosten zu amortisieren, sich an die Anfragelast anzupassen und sind nach allgemeinem Verständnis eine flexible Alternative zu statischen Ansätzen, welche Histogramme anlegen und sie dann unverändert bestehen lassen. Histogramme haben eine Vielzahl von Anwendungen in Datenbanken und verwandten Anwendungen. Wir haben die Anfrageoptimierung erwähnt, weitere Beispiele sind die Top-k Anfrageverarbeitung, die näherungsweise Beantwortung von Anfragen, Skyline Anfragen, sowie geographische und spatio-temporale Datenbanken.

Diese Arbeit hat zwei Hauptbeträge. Der erste Beitrag sind signifikante Verbesserungen von selbstoptimierenden Histogrammen durch Initialisierung (Abschnitt 1.1).

Der zweite Beitrag ist die Verallgemeinerung von selbstoptimierenden Histogrammen zur Unterstützung von nicht-linearen Kostenmodellen (Abschnitt 1.2).

1.1 Initialisierung von Histogrammen

1.1.1 Probleme von Selbstoptimierungsansätzen

Obwohl sie viele attraktive Eigenschaften besitzen, haben Selbstoptimierungsansätze mehrere Nachteile. Die Hauptannahme hinter selbstoptimierenden Histogrammen ist, dass – gegeben, dass genug Anfragenergebnisse zum Lernen vorhanden sind – sie in der Lage sind, die zugrunde liegenden Daten akkurat zu erfassen. Wir zeigen, dass dies nicht der Fall ist. Die als erstes gelernten Anfragen haben für selbstoptimierende Histogramme eine größere Bedeutung als Anfragen, die später gelernt werden. (Dies ist ein übliches Verhalten auch bei anderen überwachten Lernverfahren.) Die ersten Anfragen definieren die obersten Ebenen der Histogramm-Strukturen. Wenn diese Strukturen schlecht sind, dann ist subsequentes Lernen normalerweise nicht in der Lage, dies zu beheben. Daher kann die Reihenfolge der gelernten Anfragen einen großen Einfluss auf die Strukturen und die Genauigkeit der Schätzungen von Histogrammen haben. Wir nennen dies: Sensibilität bezüglich Lernen. Ein assoziiertes Problem ist, dass selbstoptimierende Methoden Schwierigkeiten haben, komplexe Datenstrukturen in hochdimensionalen Räumen zu erfassen. Dies liegt in der Tatsache begründet, dass es schwierig ist, dichte Daten-Regionen in Projektionen von hochdimensionalen Räumen, insbesondere wenn wir die Daten nicht selbst erfassen können, zu finden, während lediglich Anfrageergebnisse begutachtet werden können.

Wir wollen diese Probleme lösen, ohne die Vorteile von selbstoptimierenden Methoden, namentlich ihre Fähigkeit, sich an die Arbeitslast anzupassen, und die Fähigkeit, Initialisierungskosten zu amortisieren, zu verlieren. Wir stellen das Konzept der Histogramm-Initialisierung vor. Die Idee ist, mit wenigen, aber vorsichtig gewählten, Regionen [engl. buckets] zu starten. Wir stellen im Folgenden dar, wie dies funktioniert.

1.1.2 Subraum-Clustering und Histogramme

Wir zeigen formal und experimentell, dass die Initialisierung von selbstoptimierenden Histogrammen die Genauigkeit von Schätzungen erhöht. Die hier beschriebenen Ergebnisse basieren auf [KMBK11]. Zur Initialisierung nutzen wir Subraum-Clustering-Algorithmen, die kompakte, dichte Cluster von Objekten in Projektionen von hochdimensionalen Räumen finden.

Initiale Regionen definieren wenige, aber vorsichtig gewählte Top-Level Regionen für Histogramme. Diese Regionen verhindern, dass schlecht zu lernende Anfragen die gesamte Datenstruktur ruinieren. Sie machen Histogramme weniger abhängig von der Qualität und der Reihenfolge der ersten gelernten Anfragen. Das Histogramm ist dann in der Lage, zu besseren Regions-Konfigurationen zu konvergieren, und bleibt nicht in lokalen Optima hängen. Dies berücksichtigt die Sensibilität bezüglich Lernen.

Jeder Subraum-Cluster ist mit einer Menge von relevanten Dimensionen verbunden. Genauer: Wenn einige Dimensionen irrelevant für einen bestimmten Cluster sind, dann werden sie verworfen. Dies erlaubt es, dass Histogramme schwer zu entdeckende Korrelationen speichern, und dass sie gleichzeitig speichereffizient sind.

Wir zeigen, dass der Berechnungsoverhead für die Initialisierung gering ist und dass er, gemessen an der Verbesserung der Schätzgenauigkeit, leicht zu akzeptieren ist.

Als nächstes vergleichen wir verschiedene Subraum-Clustering-Algorithmen in Bezug auf ihre Leistungsfähigkeit als Initialisierer. Einige Subraum-Clustering-Algorithmen können Cluster von beliebiger Form ausgeben, so dass wir sie in eine histogrammfreundliche Darstellung transformieren müssen.

Formale Ergebnisse. Wir definieren Sensibilität bezüglich Lernen formal. Wir zeigen, dass selbst für die einfachsten Datensätze eine angemessene Initialisierung die Sensibilität des Lernens reduziert. Das bedeutet, dass Initialisierung den negativen Effekt von "schlechtem" Lernen begrenzt.

Wir formalisieren den Begriff der Transformation von Clustering-Ergebnissen zu Histogramm-Regionen. Danach definieren wir Klassen von Transformationen, die nützliche Eigenschaften haben. Als nächstes zeigen wir, dass es zu teuer ist, strikt optimale Clustering-zu-Histogramm-Transformationen zu finden. Stattdessen schlagen wir eine Heuristik vor, die gute Transformationen findet.

Experimentelle Ergebnisse. Wir haben Experimente durchgeführt, deren Aufbau dem von in verwandten Arbeiten vorgenommenen Experimenten entspricht.

Wir haben sechs Subraum-Clustering-Algorithmen als Initialisierer verglichen. Einer von ihnen zeigte konsistente Verbesserungen (bei allen Experimenten) gegenüber uninitialisierten Histogrammen sowie gegenüber anderen Clustering-Initialisierungsverfahren. Um die gleichen Fehlerraten wie die uninitialisierten Verfahren zu erreichen, benötigt dieser Algorithmus achtmal so wenig Speicher für die Histogramme.

Wir zeigen, dass selbstoptimierende Histogramme sensibel bezüglich Lernen sind. Ohne Initialisierung ist das Histogramm nicht in der Lage, selbst simple Datenstrukturen zu lernen. Für komplexe Datensätze garantiert die Initialisierung, dass der Fehler des Histogramms um 50% reduziert wird. Als nächstes zeigen wir, dass die Effekte der Initialisierung nachhaltig sind. Selbst nach intensivem Lernen kann ein uninitialisiertes Histogramm nicht an die Leistung der initialisierten Version anknüpfen.

Insgesamt erlaubt es die Initialisierung von selbstoptimierenden Histogrammen, die Sensibilität bezüglich Lernen zu verlieren, und es verbessert die Genauigkeit der Schätzung signifikant. Gleichzeitig werden die positiven Eigenschaften von selbstoptimierenden Methoden bewahrt.

1.2 Nicht-lineare Kosten- und Kardinalitätsverteilungen

Relationale Kostenoptimierer nehmen an, dass die Kosten eine lineare Funktion der Selektivität sind. Aktuelle Forschungsergebnisse zeigen, dass diese Annahme zu ungenauen Kostenschätzungen führen kann. Genauer: Gewisse Anwendungen wie die Top-k Anfrageverarbeitung muss mit Kostenmodellen umgehen können, die nicht einmal nährungsweise linear sind.

Ein lineares Kostenmodell in ein nicht-lineares zu überführen, hat zur Konsequenz, dass auch das Selektivitätsabschätzungssubsystem angepasst werden muss. Statt einer einzelnen Selektivitätsschätzung benötigt der Optimierer nun eine Wahrscheinlichkeitsverteilung über mögliche Selektivitäten. Diese Verteilungen müssen präzise sein, müssen fundiert theoretisch abgeleitet sein und sollten wenig Overhead erzeugen. Die Unterstützung verteilungsbasierter Schätzungen ist eine große Herausforderung, beim Wechsel hin zu einem nicht-linearen Kostenmodell. Verteilungsbasierte Selektivitätsschätzungen in mehrdimensionalen Räumen sind der zweite Beitrag dieser Arbeit. (Die hier beschriebenen Ergebnisse basieren auf [KB10]).

Wir zeigen, wie der Übergang von einem Modell basierend auf einer einzigen Schätzung nahtlos durch die Nutzung von ausschließlich im Histogramm existierenden Informationen bewältigt werden kann. Die Wahrscheinlichkeitsverteilung wird durch die Annahme abgeleitet, dass ein Tupel mit gleicher Wahrscheinlichkeit an jedem Punkt der Region vorkommen kann.

Unsere Experimente zeigen, dass wahrscheinlichkeitsbasierte Kostenschätzungen genauer sind als konventionelle. Die Wahrscheinlichkeitsverteilungen haben darüber hinaus einige interessante theoretische Eigenschaften.

Formale Ergebnisse. Für jedes Histogramm existieren viele Datensätze, die zu diesem kompatibel sind. Wir zeigen das Folgende: Sind alle kompatiblen Datensätze gleichwahrscheinlich, so sind unsere verteilungsbasierten Schätzungen optimal. Die Annahme, dass alle kompatiblen Datensätze gleichwahrscheinlich sind, ist eine natürliche Annahme, wenn wir keine weiteren Informationen über die Verteilung der möglichen Datensätze besitzen.

Experimentelle Ergebnisse. Experimente zeigen, dass für nicht-lineare Kostenfunktionen aus der Literatur verteilungsbasierte Schätzungen (in allen Experimenten) besser als konventionelle Schätzungen sind. In einigen Versuchen ist der Fehler der verteilungsbasierten Schätzungen nur halb so groß wie bei den Vergleichsverfahren. Nicht-lineare Kostenmodelle sind essentiell für präzise Kostenschätzungen in Datenbanken und vergleichbaren Anwendungen. Wir zeigen, dass nicht-lineare Kostenmodelle durch existierende Datenstrukturen für Selektivitätsschätzungen unterstützt werden können, ohne dass zusätzlicher Overhead entsteht.

2 Thesis Abstract

Databases enable users to issue declarative queries. The user **describes the data** he wants to obtain from the database, and is relieved from specifying **how the data should be retrieved**.

There are numerous alternative ways to execute a query. These are so called *execution plans*. A component in the database management system called the *Query Optimizer* decides how to pick an efficient execution plan. To this end, the optimizer deploys cost-based optimization. Approximate execution costs are calculated for various plans, and one with low cost is chosen. The execution cost is a weighted function of the system resources needed to execute the query. Examples of such system resources are the CPU time or the number of I/O operations.

In order to come up with reasonable cost estimates, the optimizer needs to estimate the size of sub-queries. This is important, for instance, when choosing the join order of the relations. To estimate the sizes of sub-queries, the optimizer needs to know the *selectivity* of the query predicates.

The main data structures used for selectivity estimation in databases are histograms. *Self-tuning* histograms are a class of histograms which use the results of already executed queries to refine themselves. This is a sort of supervised learning. Self-tuning histograms are able to amortize the construction costs, adapt to the query workload and are generally considered to be a flexible alternative to static approaches which construct the histogram and leave it unchanged.

Histograms in general have multiple uses in databases and related applications. We mentioned Query Optimization, other examples are Top-k query processing, approximate query answering, Skyline queries, geographical and spatio-temporal databases.

There are two main contributions in this thesis. The first contribution is about significant improvement of self-tuning histograms by Initialization (Section 2.1).

The second contribution is about generalization of self-tuning histograms to support non-linear cost models (Section 2.2).

2.1 Histogram Initialization

2.1.1 Problems With Self-Tuning Approaches.

Despite their attractive features, self-tuning approaches have several disadvantages. The main assumption behind self-tuning histograms is that, given enough query results to learn, they will be able to accurately capture the underlying data distribution. We show that this is not the case. For self-tuning histograms, the first learning queries have a great importance compared to the queries that come later in the workload. (This is commonplace with other supervised learning algorithms as well). These first queries define the top-level structure of the histogram. If this structure is bad, the subsequent learning is usually unable to fix it. Thus, the order of the learning queries can have a big influence on the structure and the estimation precision of the histogram. We call this *sensitivity to learning*.

An associated problem is that self-tuning methods struggle to capture complex data correlations in high-dimensional spaces. This stems from the fact that it is hard to find dense data regions in projections of high-dimensional space, particularly if we do not access the data itself, but only look at query-execution results.

We want to solve these problems without sacrificing the advantages of the self-tuning methods, namely their ability to adapt to the workload and to amortize the construction costs.

We introduce the concept of *Histogram Initialization*. The idea is to start with few, but carefully chosen buckets. We now outline how this works.

2.1.2 Subspace Clustering and Histograms

We show formally and experimentally that the initialization of self-tuning histograms improves the estimation precision. This material is based on [KMBK11]. As initializers, we use subspace-clustering algorithms, which find compact, dense clusters of objects in projections of high-dimensional space.

Initial buckets define few, but carefully chosen top-level buckets for the histogram. These buckets prevent bad learning queries from spoiling the overall structure. They make the histogram less dependent on the quality and the order of the *first few* learning queries. The histogram then is able to converge to better bucket configurations and does not get stuck in local optima. This addresses *sensitivity to learning*.

Each subspace cluster comes with the set of relevant dimensions. That is, if certain dimensions are irrelevant for the particular cluster, they are skipped. This allows the histogram to store hard-to-detect local correlations and be memory-efficient at the same time.

We show that the computational overhead for initialization is small and is well acceptable given the gain in estimation precision.

We compare several subspace-clustering algorithms in terms of their performance as initializers. Certain subspace clustering algorithms can output clusters of arbitrary shape, so we have to transform these into histogram-friendly representation.

Formal Results. We formally define sensitivity to learning. We show that even for the simplest datasets, proper initialization reduces the sensitivity to learning. This means that initialization limits the negative impact of "bad" learning queries.

We formalize the notion of the transformation of the clustering results into histogram buckets. We then define classes of transformations which have useful properties. Next, we show that finding the strictly optimal cluster-to-bucket transformation is overly expensive. Instead, we propose a heuristic which finds good transformations.

Experimental Results. We have conducted experiments using settings which are commonly used in related work.

We compare six subspace clustering algorithms as initializers. One of them has shown consistent improvement (throughout all experiments) over uninitialized histograms as well as other clustering-initialized versions. To achieve the same error rates as the uninitialized version, it needs 8 times less memory for the histogram.

We show that self-tuning histograms are sensitive to learning. Without initialization, the histogram is unable to learn even very simple data distributions. For the complex datasets, initialization assures that the histogram error is reduced by around 50%. Next, we show that the effects of initialization are persistent. Even after extensive training the uninitialized histogram does not catch up with the uninitialized version.

Overall, initialization allows self-tuning histograms to avoid sensitivity to learning, and increases the estimation precision considerably. Meanwhile, the positive properties of self-tuning methods are retained.

2.2 Non-linear Costs and Cardinality Distributions.

Relational cost optimizers assume that the cost is a linear function of selectivity. Recent research has shown that this assumption can lead to inaccurate cost estimates. In particular, certain applications like Top-k query processing have to cope with cost models which are not even close to linear.

Changing the cost model from linear to non-linear requires changes in the selectivity estimation subsystem as well. Instead of a single selectivity estimate the optimizer now needs a probability distribution over possible selectivities. These distributions have to accurate, derived in a theoretically sound way and should not incur much overhead. Supporting such distribution-based estimates is one of the major difficulties if one wants to transition to a non-linear cost model.

Distribution-based selectivity estimation in multi-dimensional spaces is the second contribution of this thesis (material based on [KB10]).

We show how the transition from single estimate based model can be done seamlessly, using only the existing information contained in the histogram. The probability distribution is derived using the assumption that a tuple has equal chance of appearing anywhere within the bucket. Our experiments show that probability-based cost estimates are more precise than conventional ones. The probability distributions also have certain interesting theoretical properties.

Formal Results. Given a histogram, there are multiple datasets compatible with it. We show that if all the compatible datasets are equally likely, then our distribution-based estimates are optimal. The assumption about the equal likelihood of the compatible datasets is a natural one if we do not have additional information about the distribution of the possible datasets.

Experimental Results. Experiments show that for textbook non-linear cost functions, distribution-based estimates are better (in all experiments) compared to conventional estimates. In some settings, the error for distribution-based estimates is the halved compared to the baseline.

3 Introduction

Abstract. *Database management systems enable users to issue declarative queries, which means that the user writes* ***what*** *data he wants to get, and leaves the decision on* ***how*** *to get this data to the database management system. This makes it much easier for the user to query the data, but to make it happen, the system needs to figure out how to execute the queries. There are numerous ways to execute even a simple query, and picking a good execution strategy is tricky. A component of the database management system, the Query Optimizer, is responsible for this. The Query Optimizer estimates the costs of different plans and chooses one with low cost. How this estimation is done, and what are the difficulties, is the topic of this introduction.* □

3.1 Query Optimization

Declarative queries are one of the main reasons why database systems are so widespread. Declarative query languages such as SQL are higher level of abstraction than procedural languages such as Java. This makes things easy for the one who writes the queries. The queries however need to be executed and the physical data fetched, filtered, sorted before it is passed to the user. Modern database systems have to handle large amounts of data which is accessed and modified in parallel – thus, each query has to be executed as efficiently as possible.

Most database management systems employ a *cost-based* query optimizer to find the best query execution strategy (called an execution plan) among numerous alternatives. Schematically, this means the optimizer enumerates all execution plans, estimates the execution cost for each plan, and chooses a plan with the lowest estimated cost. In practice, there are two major difficulties to overcome, namely:

- There are too many possible execution plans, and enumerating all of them is impractical.
- It is difficult to accurately estimate the cost of a plan.

The first difficulty was tackled in the classical `System R` optimizer [SAC$^+$79]. Their query optimization algorithm is much celebrated and well known, and can be found on most textbooks on databases. For the purposes of this thesis, we will confine to mentioning that the dynamic programming algorithm proposed in [SAC+79] greatly reduces the search space of execution plans. For a more recent review on query optimization in relational systems, see [Cha98].

This thesis relates to the second problem – how to accurately estimate the execution cost of a plan. We now turn to the challenges in estimating the execution plan costs.

3.1.1 Query Plan Costs

Cost-based optimization is based on a *cost model*, which assigns costs to different execution plans. The cost of a plan is calculated based on the amount of system *resources* that are needed to carry out the execution. Such resources are the CPU time, number of I/O reads, number of I/O writes, the size of required main memory buffers and so on.

Definition 3.1 (Query Execution Costs)
The query execution cost is a function of the resources used for the execution. □

An actual cost model is defined by the resources considered and the cost function, i.e. how these resources are weighted in the final cost calculation.

Example 3.1: Let the resources considered be "disk reads (R)", "disk writes (W)", and "CPU (C)". The cost function can look like

$$cost(R, W, C) = R + 100W + 10^{-6}C$$

Here one disk write is roughly as costly as 100 reads, and the CPU is virtually free compared to these operations. ■

The cost function reflects how different resource costs relate to each other. The more "expensive" is a resource, the more is its weight in the cost function. In disk-based systems, the I/O costs usually dominate other factors. Among different cost optimizers, various cost models are used[HCLS97]. Generally, the more expensive plans tend to take longer to complete. So the execution time can be taken as a rough equivalent for the query cost.

3.2 Selectivity and Cardinality

A property that all practical cost models share is that they need to estimate the *selectivity* of *query predicates* in order to produce a cost estimate. We give definitions of the query predicate, the selectivity and cardinality, and then show on several examples why it is crucial for the optimizer to know the selectivity of a predicate in order to come up with a good execution plan.

Definition 3.2 (Query Predicate)
Given a relation R, a query predicate π on R is a boolean function:

$$\pi : R \rightarrow \{true, false\}$$

Thus, for each tuple in the relation the predicate is either true or false. We also denote the set of all tuples that satisfy the predicate as $\pi(R)$:

$$\pi(R) = \{t \in R | \pi(t) = true\}$$

□

Definition 3.3 (Predicate Cardinality and Selectivity)
Let R be a relation and π is a predicate on R. The cardinality of π is the number of tuples from R that satisfy π:

$$card(\pi) = |\pi(R)| = |\{t | t \in R \wedge \pi(t) = true\}| \tag{3.1}$$

The selectivity of π is its cardinality divided by the number of tuples in R:

$$sel(\pi) = \frac{card(\pi)}{|R|} \tag{3.2}$$

□

The cardinality is the number of tuples satisfying the condition. The selectivity is the portion of tuples satisfying the condition.

Example 3.2: Using the relation `Cars(ID, Model, Maker, Year)` in Table 3.1, we will compute the cardinality and the selectivity of some predicates.

1. `SELECT * FROM Cars WHERE Maker = 'Volkswagen'`
The predicate here is `Maker = 'Volkswagen'`, its cardinality is 1, and the selectivity is 0.2.

2. `SELECT * FROM Cars WHERE Maker = 'Peugeot'`
The predicate is `Maker = 'Peugeot'`, the cardinality = 2, selectivity = 0.4.

The query predicate can also be composite, i.e. refer to several attributes:

3. `SELECT * FROM Cars WHERE Maker = 'Peugeot'`
`AND Year < 1960`
The predicate is `Maker = 'Peugeot' AND Year < 1960`, cardinality = 1, selectivity = 0.2. ■

ID	Maker	Model	Year
1	Volkswagen	Golf Mk3	1992
2	Porsche	996	2001
3	Ford	Fiesta Mark VI	2008
4	Peugeot	204	1969
5	Peugeot	203	1949

Table 3.1: A sample relation Cars

Note. "High selectivity" and "highly selective" should not be confused. When we say "high selectivity" we mean the value of selectivity is high, e.g. when a predicate selectivity is 0.4 it is higher than 0.2. The term "highly selective" means that the predicate selects very few tuples, i.e. the selectivity is low. In this thesis, we use the term "high selectivity".

We demonstrate below that for different selectivities the optimal plan can change. We discuss the following cases

- **Index selection.** The optimizer has to choose between alternative plans which use different indexes. In addition, skipping indexes and scanning the whole data set can be an option too.

- **Two-table join.** When joining two tables, it is often best to read the smaller table into the main memory and leave the bigger table to the disk. The assumption is that the larger table does not fit into the main memory completely, and we want to read it from the disk only once.

- **Multi-table join.** When joining more than two tables, the order of the join affects the cost. In order to choose the optimal join order, we have to estimate the selectivity of predicates which filter the tables.

The relations we will be using for the examples are :

```
Customer(cID, Name, Age, Country)
Order (oID, cID, Total, Status)
LineItem (iID, oID, Category, Price)
```

3.2.0.1 Index selection

An index accelerates the access to the data. However, there is an extra cost which comes from the index access itself. If we are to retrieve almost all data in the relation, then consulting an index is wasteful. However, if we are to retrieve only very small percentage of the tuples, then using the index provides a benefit.

Let the query be

```
SELECT * FROM Customers WHERE Age < 25
```

Assume we have a non-clustered index on the attribute `Age`, call it `i_Age`. Figure 3.1 shows the costs for scanning the whole table vs using the index `i_Age`, for different selectivity values. The cost for scanning the table is constant; the cost for using the index increases linearly with selectivity. The two plans intersect when the selectivity = 0.1. In order to find out the best access method, we have to estimate the selectivity of the predicate `Age < 25`, or at least find out whether it exceeds 0.1.

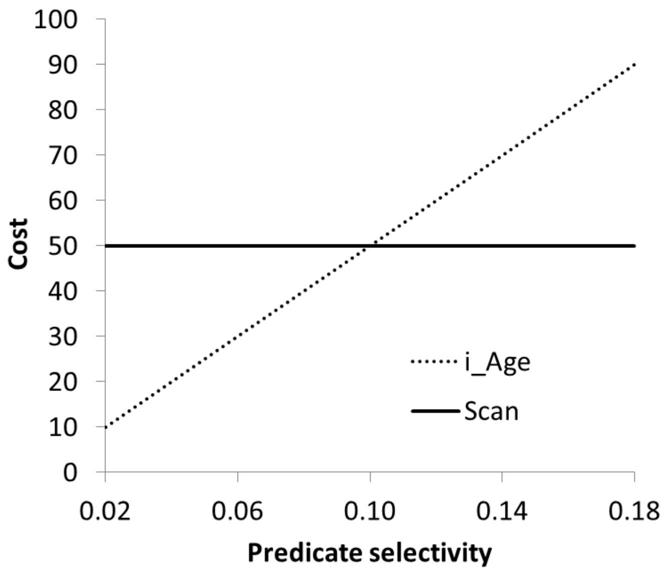


Figure 3.1: Cost graph for two plans, table scan and index seek

3.2.0.2 Two-table join

Consider the following join:

```
SELECT *
FROM Customer C JOIN Order O
ON C.cID = O.cID AND O.Total < 150
```

There are multiple ways to execute this join. For example, if we consider only the hash-joins, there are two plans possible, shown in Algorithms 1 and 2.

```
Algorithm 1: Hash join of Customers and Orders
    H = Filter(Customers, Predicate: $Total < 150$)
    HashJoin(H, Orders, JoinCondition: $H.cID = Orders.cID$)
```

```
Algorithm 2: Hash join of Orders and Customers
    HashJoin(Orders, Filter(Customers, Predicate: $Total < 150$), JoinCondition:
    $H.cID = Orders.cID$)
```

The difference between Algorithms 1 and 2 is the order of the join. In 1, we first filter the Customers relation and use it as the build input. Namely, the algorithm uses it to build a hash table. The relation `Orders` is then probed for matches. In contrast, Algorithm 2 uses the relation `Orders` as the build input and the filtered `Customers` relation as the probe input. Now assume for simplicity that the there are no indexes defined on both `Customers` and `Orders`. Then, the best strategy is to choose the *smaller* relation as the input. This means, the optimizer has to assess the cardinality of the expression `Filter(Customers, Predicate: "Total < 150").` Note that in this example we limit ourselves to only one join method, and we exclude indexes from consideration. This shows that even when the execution-plan search space is very limited, selectivity estimates are still needed to find out the better plan.

3.2.0.3 Multi-table join

Consider the join query:

```
SELECT * FROM Customer C join Order O
On C.cID=O.cID JOIN LineItem L ON O.oID=L.oID
WHERE L.Category = "Game"
```

The following join orders are possible:

$$\begin{aligned} P_1 &= (C \bowtie O) \bowtie \sigma(L) \\ P_2 &= (\sigma(L) \bowtie O) \bowtie C \end{aligned} \tag{3.3}$$

and $\sigma(L)$ is short for
`Filter(LineItems, Predicate: Category = 'Game').`

Each of the plans P_1 and P_2 are "logical" plans, in the sense that they map into multiple physical execution plans depending on which physical operators we choose. For instance, $\sigma(L) \bowtie O$ can be carried out using hash join, merge join, and the filter $\sigma(L)$ can be implemented by scanning L or by using an index. The costs of all these plans depend on the selectivity of the filter σ.

We will demonstrate this using a very simplified scenario. Assume our only optimization goal is to have a result size as small as possible after the first join. In this case, we want to compare the size of $C \bowtie O$ and $\sigma(L) \bowtie O$. First, note that

$$|C \bowtie O| = |O|$$

This is because each order has one customer. This constraint can be expressed using foreign keys and be made available to the optimizer.

Each lineitem belongs to one order, so

$$|\sigma(L) \bowtie O| = sel(\sigma) \cdot |L|$$

Now, in order to find out whether P_1 or P_2 is the best plan, we have to compare $|O|$ to $sel(\sigma) \cdot |L|$. Thus, optimal plan choice depends on accurate estamation of the selectivity of σ.

We demonstrated that in order to estimate the costs of the query plans accurately, the optimizer need to know the selectivity of the query predicates. The selectivities of predicates are crucial in estimating the cost access method of the a single relation, the join method for two relations or the join order of multiple relations. Thus, we have established that we need accurate selectivity estimates in variety of scenarios to ensure effective query optimization.

3.2.1 Properties of Selectivity Estimates

The selectivity of the predicate is the portion of the tuples from the base relation which satisfy the given predicate (see Definition 3). Now consider a predicate π and the random variable X_π, defined as follows:

$$X_\pi = \begin{cases} 1, & \pi(t) = true \\ 0, & otherwise \end{cases}$$

The probability $P(X_\pi = 1)$ is the probability that a randomly selected tuple satisfies the predicate π:

Observation 3.1: The probability that a randomly selected tuple satisfies a predicate π equals the selectivity of π.

$$P(X_\pi = 1) = sel(\pi)$$

□

This observation allows us to look at the notion of selectivity from the probability point of view. In the following, we will sometimes write $P(\pi)$ instead of $P(X_\pi)$ for improved readability.

Property 3.1: (Selectivity Range)
The selectivity of a predicate π is a value between 0 *and* 1.

Property 3.2: (Negation selectivity)
For and arbitrary predicate π, the selectivity of the negation $\neg\pi$ is given by

$$sel(\neg\pi) = 1 - sel(\pi)$$

Property 3.3: (Filtering property)
For arbitrary predicates π_1 and π_2,

$$sel(\pi_1) \geq sel(\pi_1 \wedge \pi_2)$$

The filtering property indicates that applying an additional condition (connected by "and") can only lower the selectivity.

Definition 3.4 (Predicate dimensionality)
The dimensionality of the predicate is the number of attributes it refers to. □

Example 3.3: Revisiting Example 2, let's compute the dimensionality of the predicates:
1. `SELECT * FROM Cars WHERE Maker = 'Volkswagen'`
The predicate refers to only one attribute – `Maker`, thus its dimensionality is 1.
2. `SELECT * FROM Cars WHERE Maker = 'Peugeot'`
Same as above.
3. `SELECT * FROM Cars WHERE Maker = 'Peugeot'`
`AND Year < 1960.`
In this case the predicate refers to two attributes, thus its dimensionality is 2. ■

The predicates that refer to only one attribute are called *unidimensional* or *single-dimensional*. The predicates that refer to more than one attribute are called *multi-dimensional*.

3.2.2 Query types

So far, we have not put any restrictions on what kind of boolean predicates the queries can have. Most database systems in fact support a fairly generic class of query predicates. However, the vast majority of the predicates that are used in actual queries come from a narrow class. As a consequence, most optimizers are most efficient when they encounter predicates from these classes.

Definition 3.5 (Range predicate and range query)
A query predicate on relation $R(A_1, \ldots, A_n)$ is a range predicate if it has the form

$$\pi = (c_1 \leq A_1 \leq C_1) \wedge (c_2 \leq A_2 \leq C_2) \wedge \ldots \wedge (c_n \leq A_n \leq C_n) \qquad (3.4)$$

where c_i and C_i are constants for all $i = 1, \ldots, n$. A query that has a range predicate is called a range query. □

The predicate of a range query spans a hyper-rectangle in the attribute-value space.

Example 3.4: Consider the relation `Employee(ID, Age, Income)` and the query

```
SELECT * FROM Employee
WHERE Income BETWEEN 25000 AND 45000
AND Age BETWEEN 20 AND 30
```

The dimensionality of the query predicate is 2.

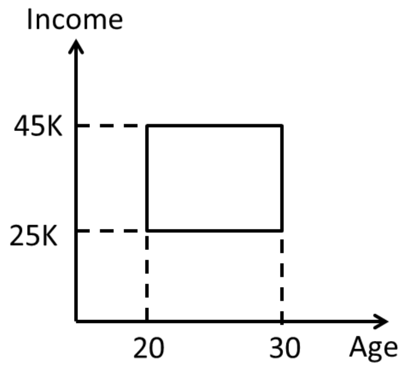


Figure 3.2: The range query as a rectangle in the two-dimensional space

The query predicate spans a rectangle in the two-dimensional attribute-value space, shown in Figure 3.2.

Note that Definition 5 indicates that the query predicate has to refer to all of the attributes of the relation, while in our example we have omitted `ID`. This is not principal because our query is equivalent to

```
SELECT * FROM Employee
WHERE Income BETWEEN 25000 AND 45000
```

```
AND Age BETWEEN 20 AND 30
AND ID BETWEEN minID AND maxID
```

Using this trick we can model any $n - k$ dimensional predicate via an n-dimensional predicate. ■

Definition 3.6 (Point query)
A point query is a special case of the range query, where $c_i = C_i$ for all $i = 1, \dots, n$. □

The distinguishing feature of the range queries is that they use constants to define the predicate range. In contrast, the following query:

```
SELECT * FROM A, B WHERE A.ID=B.ID
```

does not use constants but rather another attribute for the equality. Such queries are *join queries*.

Complex queries usually join several tables together and contain range predicates for filtering the some of the individual tables.

Example 3.5: The query

```
SELECT *
FROM Customer C JOIN Order O
ON C.cID = O.cID AND O.Total < 150
```

is a join query which has a range sub-query. The sub-query is equivalent to

```
SELECT * FROM Orders WHERE O.Total < 150
```

■

In this thesis, we focus on selectivity estimation of range predicates.

3.3 Requirements for Selectivity Estimation

Selectivity estimation needs meet several criteria in order to be effective. Here, we discuss such requirements.

If the estimates are not precise enough, the estimated and the real costs of the plans will differ significantly. Optimization will become pointless.

Requirement 3.1: (Precision)
Selectivity estimates need to be precise.

Recall that the number of possible execution plans can be rather large even for relatively simple queries [RH05]. This means the selectivity estimation subroutine need to be invoked multiple times for a single query. It is clear that selectivity estimation need to be very fast; otherwise the time spent for optimizing a query can become comparable to query execution time.

Requirement 3.2: (Speed)
Selectivity estimation need to be fast, as it it takes place in the inner loop of the optimization cycle.

Usually, selectivity estimation is based on some summary representation of the data. If this summary is large, it needs to be stored on a secondary storage. One should avoid this if one wants to fulfill the Requirement 2: after all, reading from the secondary storage is one of the most time-consuming operations. Thus, we want to keep the size of auxiliary data structures used for selectivity estimation small, such that we can keep the whole thing in main memory. Most systems allocate only several disk pages for these data structures, and pin these pages in the main memory. Pinning means the pages are marked to prevent the system from dumping those pages into the secondary storage. This is similar to what happens to the top level pages of an index. Note that these several disk pages are allocated for the whole database not for a single table or column.

Requirement 3.3: (Space)
The auxiliary data structures used for selectivity estimation need to be compact.

The data that the users choose to store in databases can be large and complex. The structure, the volume and the distribution of the data can change over time. In the majority of the cases, the database management systems is supposed to be able to cope with any data which fits the schemata. The system usually does not have a prior knowledge about the data.

Requirement 3.4: (Model-Free)
Selectivity estimation should work for a model-free world, where the data that is present is the only data of interest. The data structures used for selectivity estimation should support this world view.

3.3.1 Multi-dimensional Predicates

One of the major challenges for selectivity estimation techniques is the evaluation of multi-dimensional predicates. In this section, we will show that, in general case, multi-dimensional selectivity estimates cannot be derived directly from single-dimensional ones. Later in this thesis (in Chapter 4) we will see that the techniques which handle multi-dimensional predicates are much more complicated than those for the single-dimensional predicates.

Let us extend the relation `Cars` by adding another column, `Color`. Consider the query

```
SELECT * FROM Cars WHERE Maker= 'Peugeot'
AND Color='Red'.
```

The predicate here is 2-dimensional. We could assume that the attributes `Maker` and `Color` are *independent*. In terms of random variables, it is equivalent of saying that the variables $X(Maker =' Peugeot')$ and $X(Year < 1960)$ are statistically independent. Then, we can write

$$\begin{aligned} &P(Maker =' Peugeot' \wedge Color =' Red') = \\ &P(Maker =' Peugeot') \cdot P(Color =' Red') \end{aligned} \tag{3.5}$$

It is clear that the attributes `Maker` and `Color` won't satisfy the independence assumption in general. To see why this is the case, consider the predicate

```
  Maker='Ferrari' AND Color = 'Red'.
```

For some reason, most Ferraris are red, so applying the independence assumption would drastically underestimate the number of red Ferraris! An important observation here is that the kind of dependency of `Color` and `Maker` cannot be deduced from the schema.

Using the independence assumption to compute the selectivity of a multi-dimensional predicate as the product of single-dimensional predicates can lead to large estimation errors. We formulate the ability to handle the multi-dimensional predicates autonomously as a separate requirement.

Requirement 3.5: (Multi-Dimensional Predicates)
Selectivity estimation should be able to handle multi-dimensional query predicates without relying exclusively on the independence assumption.

3.4 Summary

Declarative queries rely on query optimization to cope with explosive number of possible execution plans. The optimizer in turn needs selectivity estimates of query predicates.

The selectivity estimates need to be *precise*, *fast*, *compact* and *model-free*. For *multi-dimensional predicates*, meeting these requirements becomes particularly challenging. In Chapter 4 we will introduce data structures and algorithms which attempt to solve this problem.

4 Histograms

Abstract. *In the previous chapter we talked about the importance of selectivity estimation in databases. We also outlined certain properties that the selectivity estimates should have.*

In this chapter we review histograms, which are the most commonly used data structure for selectivity estimation. As the literature on histograms is too large, we present only select techniques here.

*Parts of this chapter closely follow [**?**], which has a novel way of categorizing histograms. We consider this categorization to be very clear and effective and decided to stick to it (with minor changes).* □

4.1 First Histograms

Histograms are a summary representation of data. [Ioa03] points out that they were used as early as in 18th century.

Histograms have multiple uses in databases and related applications. In this chapter we will have in mind a selectivity estimation scenario. In Section 4.8, we will discuss several other applications of histograms.

For better understanding, we discuss the first histograms not on relational data but using the simplest statistical model where the data is a one-dimensional array. Later we return to the relational model. We will start with the simplest histogram, the Equi-Width [Koo80] histogram.

4.1.1 The Equi-Width Histogram

Let the data distribution be

$$D = \{0.8, 1.1, 1.2, 2.2, 3.3, 4.5, 4.6, 4.88, 5.9\} \tag{4.1}$$

The Equi-Width histogram partitions the data into ranges of equal width. The histogram stores the object count for each partition. A value-range together with some statistics is usually called a *bucket* (we give rigorous definitions later). Figure 4.1 shows a histogram with 6 buckets.

Assume we want to estimate the number of objects in the range $[1, 1.5]$. We know there are two objects in the range $[1, 2]$. The most natural way of approximating

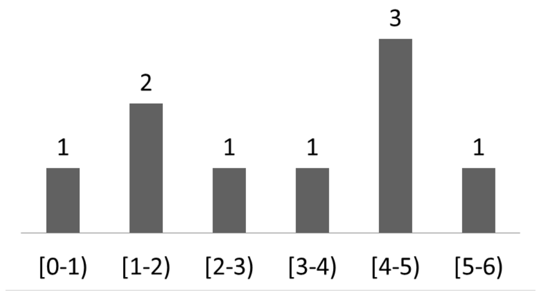


Figure 4.1: An Equi-Width histogram with 6 buckets

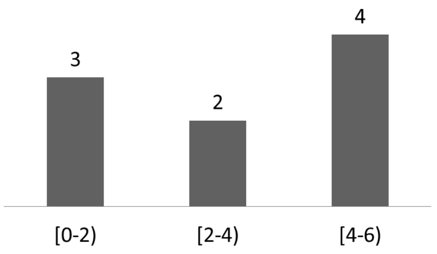


Figure 4.2: An Equi-Width histogram with 3 buckets

the number of tuples in $[1, 1.5]$ would be to assume that two values are distributed "uniformly" within the interval, which would mean one of them would fall in $[1, 1.5]$. Looking at the data, we can that the actual number of tuples is 2.

Figure 4.2 shows the case when we use three buckets in the histogram instead of six.

4.1.1.1 A disadvantage of the Equi-Width histogram.

The Equi-Width histogram partitions the data distribution into buckets of equal width. When the data is clustered in a small section of the domain, this partitioning can be very suboptimal. Consider the following data points:

$$C = \{5.01, 5.02, 5.03, \ldots, 5.49, 5.50\}$$

If we consider the distribution $D' = D \cup C$ and build a Equi-Width histogram with six buckets, we will see the problem. We added 50 data points to the bucket $[5 - 6)$. Now, if try to estimate the number of tuples in $[5.6, 5.9)$ we will get a the estimate $0.3 \cdot 50 = 15$ data points when in fact there is only one data point in that range (5.9). The reason for this large error is that we grouped the ranges [5.0-5.5] and (5.5-6) into one bucket. Those ranges have very different densities and "mixing" them will result in high estimation errors.

It is clear that for better estimations we would have to split the bucket $[5-6)$ into two, and merge some buckets elsewhere. It is also clear that by adding more points into a part of the bucket like we did with merging distributions D and C, we can make the estimation error within that bucket arbitrarily large.

4.1.2 The Equi-Depth histogram

An significant improvement over the Equi-Width histogram was the Equi-Depth histogram [PSC84]. The Equi-Depth histogram partitions the data into buckets so that each bucket contains approximately the same number of tuples. Thus, if in the Equi-Width histogram the *width* of the buckets is fixed, in the Equi-Depth histogram the *height* or the *depth* of the buckets is the same. Revisiting the distribution D (4.1), the Equi-Depth histogram with three buckets is shown in Figure 4.3.

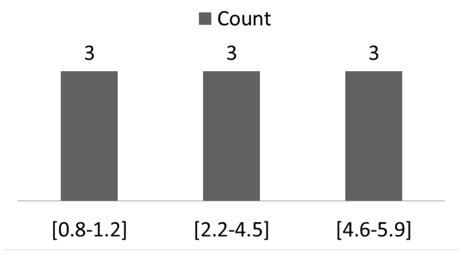


Figure 4.3: An Equi-Depth histogram with 3 buckets

The advantage of the Equi-Depth histogram over the Equi-Width histogram is that, for fixed amount of data points, it allocates fixed amount of memory to summarize those data points. If the data distribution is very dense for some small range, the histogram will divide this range and achieve a good approximation of the dense region, unlike the Equi-Width histogram .

4.1.3 Histograms On Relational Data

So far we have looked at examples of histograms where the data distribution is a one-dimensional array. This point of view is common in statistics, where the data is simply a sample from the (unknown) probability distribution. Contrary to this, relation model organizes data into tables.

There are several models on how to build histograms on relational data. Here we describe the model which is widely used because its aim is to assist cardinality estimation in databases. In this model, the data can be viewed a set of pairs

$$S = \{(v_i, f(v_i)) | 1 \leq i \leq N\} \tag{4.2}$$

where v_i is the attribute value and $f(v_i)$ is the frequency – the number of times it occurs in the relation. Without loss of generality, we can assume the values v_i to be drawn from $[0, M]$. Table 4.1 shows a table with two columns, order-id (oid) and amount. When building a histogram on the attribute "amount" our set of pairs will be

oid	amount
1	10
2	5
4	4
5	5
6	9
7	1

Table 4.1: Orders table with order-id and amount of order

$(10, 1), (5, 2), (4, 1), (9, 1), (1, 1)$

We discussed in Section 3.2.2 that there are two general types of selectivity estimation problems. Those are the *join* and the *range-query* selectivity estimation problems. We are going to focus on the range queries mostly. This means that we want the histogram to approximate *sums* of $f(v_i)$ values for some consequent values of i. Given a set of values S as in 4.2, the goal of the histogram is to approximate sums of form

$$\sigma(r) = \sum_{i \in r} f(i)$$

r is a range predicate, i.e. $r = \{j, j+1, \ldots, j+k\}$ for some j and k.

In order to estimate range-query cardinalities, histograms divide the data into *buckets*.

Definition 4.1 (Bucket)
A histogram bucket b is pair $b = (r(b), \Omega(b))$, where $r(b)$ is a subset of $[0, M]$, and $\Omega(b)$ is some aggregate information about tuple frequencies within that subset. □

The set $r(b)$ often represents a range. The statistic to store in the histogram bucket, $\Omega(b)$, is key for issuing selectivity estimates. In the simplest case, $\Omega(b)$ is simply the average frequency within the bucket. Clearly, it is possible to store more detailed information, which increases the memory footprint of a bucket. Thus, the choice of information to store in a bucket is a compromise: storing detailed information means we can approximate the in-bucket distribution better at the cost of having less buckets overall. The Equi-Width histogram divides the data into near-equal ranges. So the ranges $r(b_k) = r(b_{k+1})$. The statistics stored is

$$\Omega(b_k) = \sum_{i \in r(b_k)} f(i)$$

The Equi-Depth histogram divides the data into so that they have equal nearly amount of tuples in them, i.e. $\Omega(b_k) = \Omega(b_{k+1})$ for all buckets except maybe the last one.

Both histograms use the the Continuous Value Assumption to approximate the data distribution within a bucket.

Definition 4.2 (Continuous Value Assumption)
Under the Continuous Value Assumption , the number of tuples that lie within the query interval is assumed to be the fraction of bucket range that lies within the interval multiplied by the bucket count. Formally, if query range is q, then

$$count = n(b) \cdot \frac{|q \cap r(b)|}{|r(b)|}$$

□

Basically, the Continuous Value Assumption says that the density withing the bucket is uniform.

We already demonstrated the Continuous Value Assumption when estimating the cardinalities in Sections 4.1.1 and 4.1.2.

4.1.4 Histogram Categorization

So far we have seen that the Equi-Width and the Equi-Depth histograms arrange the buckets in the different manner, but they use the same method to approximate the distribution within a bucket.

It turns our that most histograms out there can be categorized regarding how they divide the data into buckets, what kind of statistic they store, how they approximate frequencies given a bucket information and so on. This kind of categorization helps understand how different histograms relate to each other, and what is their principal differences are.

The following aspects of histograms can be used to categorize them:

- **Bucketing scheme.** The Equi-Width and Equi-Depth histograms use disjoint, continuous ranges for buckets. As we know from Definition 1, buckets can be an arbitrary grouping of attribute values. Some histograms use overlapping or recursive bucketing schemes.

- **Statistics stored.** The Equi-Width histogram stores the number of tuples in the bucket, while the Equi-Depth histogram stores the bucket boundaries. Some histogram variants store number of distinct values or the variance of the frequencies of tuples in the bucket.

- **Approximation scheme.** So far we have mentioned only the Continuous Value Assumption as tuple-count estimation scheme. The approximation scheme depends strongly on the statistics stored in the buckets. Usually, the more aggregate information a single bucket contains, the more evolved the approximation scheme can be.

- **Class of queries answered.** Histograms are used to estimates the selectivity of *range*, *point*, and *join* predicates. Joins and range queries can use different data structures for selectivity estimation. Most of the histograms discussed here aim for range queries, however we mention several join-friendly histograms as well.

- **Incremental Maintenance.** Data in the database changes, and the histograms need either to be rebuilt periodically or allow incremental maintenance. Incremental maintenance can be important in the case when the data changes rapidly, or the building cost of the histogram is relatively high. This is especially relevant for multi-dimensional histograms, where the construction costs are typically high.

- **Misc.** Other features include error guarantees, size, build time etc. These are important issues but are somehow beyond the scope of this thesis, and we refer the reader interested in these issues to respective papers.

4.2 Estimation Schemes

The estimation scheme is the the method of estimation of the frequencies within a bucket.

4.2.1 Equi-Distant Schemes

The schemes described here are more commonly referred to as *Uniform* schemes in the literature, which is a rather inappropriate term. We will stick to the term "Equi-distant" here.

Equi-distant schemes store the number of tuples with non-zero frequency in the bucket, together with the total number of tuples.

Let $\{b_1, \dots, b_m\}$ be disjoint histogram buckets, each bucket representing an interval, so that the union of all bucket intervals covers the whole data range. An equi-distant scheme stores the number of tuples in the bucket $n(b_j)$ and the number of non-zero values in the bucket, $s(b_j)$. Figure 4.4 shows a distribution with where two out of four values have positive frequency.

Figure 4.5 shows a single-bucket Equi-Depth histogram built over this distribution.

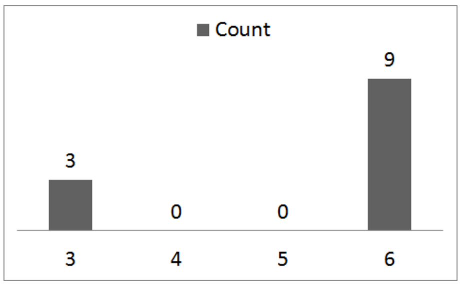


Figure 4.4: A sample distribution

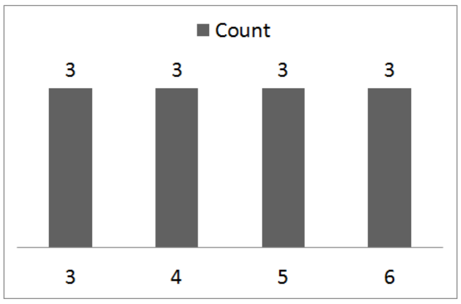


Figure 4.5: An Equi-Depth approximation of the distribution using a single bucket.

Figure 4.6 shows a histogram on the same data, this time using the Equi-distant approximation scheme instead of the Continuous Value Assumption . The histogram stores the number of positive values and the average frequency. It approximates frequencies assuming the positive values are distributed equi-distantly in the bucket.

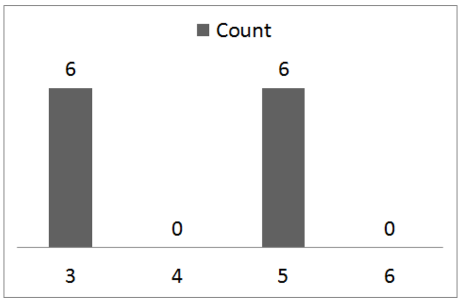


Figure 4.6: An Equi-distant approximation of the distribution, using 1 bucket with 2 non-zero values.

The Equi-distant estimation scheme is often used for JOIN and aggregation queries. The study in [WS08] shows that the Continuous Value Assumption usually outperforms the Equi-distant approximation scheme, both for single-dimensional and multi-dimensional data.

4.2.2 Other Approximation Schemes

Splines. The Continuous Value Assumption approximates the tuple frequencies within a bucket with a constant (the average). A natural generalization of this is a spline-based scheme where, for instance, the bucket stores two values instead of one and uses a linear function to approximate the densities. Indeed, it is possible to further enhance this by using polynomials. We refer to [KW99, ZL02, ZL96] for further reading.

Multi-Level Trees. 4-level trees (4LT) and n-level trees [BL04, BLS$^+$08] are hierarchical bucketing scheme for enhancing the precision of the histogram. We briefly explain the 4LT [BLS$^+$08] here, as it aims for a compact representation which allows to store the auxiliary information in one 32-bit or 64-bit integer. The idea is to store approximate partial sums of the elements that fall into the bucket. We can think of sub-buckets which can overlap and contain approximate rather than exact information. Assume we have 16 values, $v_1, \dots v_{16}$, that we want to store in a bucket. The bucket is divided into j segments of equal length, and the sum of the values of the i-th segment is denoted by $\sigma_{i/j}$. The sum of all the values in the bucket, $\sigma_{1/1}$ is stored exactly (that's the first level). The second level contains the σ-s for $j = 2$, the third level corresponds to $j = 4$ and the fourth level corresponds to $j = 8$. The number of bits allocated to each level is different as well. For j=2, 4LT allocates 6 bits for the $\sigma_{i/2}$, for j=4 its 5 bits, for j=8 only 4 bits. Such a storage of partial sums forms a tree. There is no need to store all $\sigma_{i/j}$ for all i, j. Notice that for instance $\sigma_{1/1} = \sigma_{1/2} + \sigma_{2/2}$, and we have the exact value for $\sigma_{1/1}$, so if we store say $\sigma_{1/2}$ we can compute $\sigma_{2/2}$ from those values. The same applies to the rest of the tree too. 4LT stores 1 value for j=1, 1 value for j=2, one 2 values for j=4 and 4 values for j=8. This makes 32 bits. For the exact approximation scheme and more detail, see [BLS$^+$08].

Combined Schemes. From all schemes mentioned above there is not one which is universally the best. [WS08] exploits the fact that different bucketing schemes can in fact be better for different datasets. In essence, it uses either 4LT, the Continuous Value Assumption or the Equi-distant schemes for different buckets. The overhead is that the histogram has to store descriptors about the scheme used. The experiments in [WS08] show this approach outperforming all "fixed" schemes, however, we would like to mention that the authors dedicate uncommonly large amount of memory for the histogram (5% of the data set).

4.2.3 Bucketing Schemes

So far, we have discussed the Equi-Width (Section 4.1.1) and the Equi-Depth (Section 4.1.2) bucketing schemes. Multi-level trees such as the 4LT and nLT deploy a hierarchical bucketing scheme. In the following, we discuss several other bucketing schemes which we haven't encountered so far.

4.2.3.1 Singleton Schemes

Singleton bucketing is used mostly for data where several of the most-frequent values fill almost the whole dataset. For instance, *End-biased* histograms store h elements with highest f-value and l elements with lowest f-value in singleton buckets. The rest of the data is assumed to be uniform and is allocated $s - (h + l)$ buckets, where s is the overall memory budget [IC93, Ioa93]. *High-biased* histograms have $l = 0$, i.e. they store only the high-f values. Similarly, *Low-biased* histograms have $h = 0$.

Compressed histograms [Ioa93] combine the High-biased and Equi-Depth histograms. They store the h elements with the highest f-values in singleton buckets. For the rest of the data range, they construct an Equi-Depth histogram.

4.2.3.2 MaxDiff Histograms

Maxdiff histograms draw the bucket boundaries where there is the highest frequency difference between f-values. Thus, if the histogram bucket budget is B, the data range is $[1, M]$, then there is a bucket boundary between i and $i+1$ of $|f(i+1)-f(i)|$ is among the $B - 1$ largest such differences. Maxdiff histograms require sorting the whole dataset in order to obtain the largest f-values. In practice, the cost for sorting is usually inacceptably high; for this reason Maxdiff histograms are usually constructed using a sample.

4.3 Static Multi-Dimensional Histograms

One of the requirements on selectivity estimation was that the multi-dimensional predicates need to be evaluated without relying on the independence assumption (Requirement 5 in Section 3.3.1).

In order to achieve this we need data summaries which can capture the joint data distribution of multiple attributes. *Multi-dimensional histograms* are a prominent representative of such summary structures.

Not surprisingly, first multi-dimensional histograms were generalizations of known one-dimensional approaches for multiple dimensions. Namely the method in [MD88] attempts to obtain a multi-dimensional Equi-Depth partitioning of the data. It starts with one bucket containing whole data. In each step, an existing bucket is split across a dimension, and this is repeated for all dimensions.

Figure 4.7 demonstrates the process for a 2D case. At first, the histogram contains a single bucket, which is the whole dataset. Then, a split dimension is chosen (in our example the X-axis) based on one-dimensional statistics, and the histogram is split

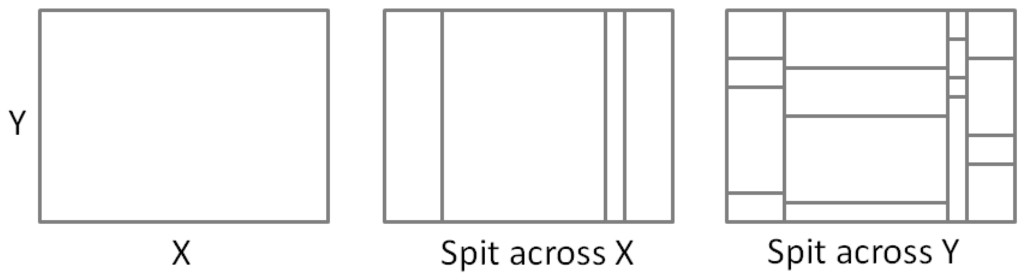


Figure 4.7: Creation of a multi-dimensional Equi-Depth histogram

across that dimension. The number of buckets created at each split step is fixed, and chosen so that in the end the number of buckets meets the space budget. As in the single-dimensional case, the buckets are created so that they contain approximately even number of tuples. Then, this step is repeated for each remaining dimension. So each bucket is split into four across dimension Y this time, again trying to preserve equal number of tuples inside the buckets.

The problem with the multi-dimensional Equi-Depth histograms is that the principle of making buckets with fixed number of tuples is inefficient, at least when coupled with the Continuous Value Assumption . If the Continuous Value Assumption is used, it seems clear that the buckets should resemble data regions with close to uniform distribution of tuples.

[PI97] attempts to solve this problem, pointing out that the underlying uni-dimensional histograms do not have to be Equi-Depth . They propose a generalization of the histograms introduced in [MD88]. Their approach is coined *MHist* and takes *MaxDiff(v, a)* (see [Ioa03]) uni-dimensional histogram instead of the Equi-Depth histogram. The *Maxdiff(v, a)* histogram attempts to minimize the variance of tuple frequency within the bucket.

GENHIST histograms [GKTD00] allow buckets to overlap. If a region lies within an intersection of two buckets, the density is the sum of densities of the buckets. GENHIST starts with a grid of cells over the data set, which is typically fine-grained. The algorithm then merges dense cells into buckets and removes the tuples that fall within that bucket from the dataset. This step is repeated with the new dataset (with tuples removed), this time with a coarser initial grid. The GENHIST algorithm results in good histograms, but the construction costs are restrictively high. The *RK-Hist* histogram [EL07] also uses intersecting buckets, resembling an R tree. First, a Hilbert curve is fitted into the data space. Then, the tuples are sorted according to their appearance along the curve. Buckets are formed so that close tuples are in the same bucket.

4.3.1 Disadvantages of Static Multi-Dimensional Histograms

Static histograms do not change after they are built. This means that the histograms need to be rebuilt regularly to reflect the changes in the data.

The construction costs of multi-dimensional histograms grow fast with the dimensionality of the dataset. This is particularly true with the approaches that have comparatively low error, e.g. GENHIST.

For high-dimensional datasets, the high construction costs and the necessity to rebuild the histograms regularly is a major obstacle for deploying multi-dimensional approaches.

Dimensionality reduction techniques (Section 4.4) attempt to avoid the problem by trying to detect "less relevant" attributes and do not consider them during histogram construction.

A radically different approach is to try to keep the histograms up to date all the time by utilizing query execution results. These results are used to update the histogram. These approaches are discussed in Section 4.5.

4.4 Dimensionality Reduction Techniques

The construction costs of histograms in high-dimensional spaces can be restrictively high. A possible workaround is to find a subsets of "highly correlated" attributes and build the histograms on these subsets. Then, these histograms can be combined to compute selectivities in full-dimensional space. [DGR01] capture the attribute correlations using a Markov network. If the dimensions i and j are correlated, then they are connected by an edge. In case there is a path $i \leftrightarrow j \leftrightarrow k$ but i and k are not directly connected by an edge, they are considered conditionally independent. The Markov network breaks down the attribute-value space into (possibly non-disjoint) set of cliques. The cliques represent a group of strongly-correlated attributes, and a histogram is maintained for each clique. When the query refers to attributes from different cliques, the distribution is computed using some heuristics. For instance, if the cliques are disjoint, the attribute value independence assumption can be used. Otherwise more complex rules have to appled, see [DGR01]. The construction of the graph is done incrementally. In the beginning, all dimensioned are assumed to be independent. Then, cliques are formed based on the improvement of the approximation (coming from forming the clique). This improvement is measured as Kullback-Liebler distance between the actual and the approximated distributions.

[GTK01] uses a Bayesian network instead of a Markov table. Similar to [DGR01], it uses a search through correlation models to find one which fits the data well.

4.5 Self-Tuning Histograms

The histograms discussed above are *static*, in the sense that they scan the data, construct a histogram which does not change later.

In contrast to this, *self-tuning* histograms use the query feedback to change themselves. (They are also called *dynamic, adaptive,* or *feedback-driven* histograms). The central idea behind self-tuning that after queries are executed, the results are known, and these results can be used to refine the histogram. This is essentially a form of supervised learning for histograms. Figure Figure 4.8 depicts the query-execution cycle of the database engine using a self-tuning (dynamic, feedback-driven) histogram. The

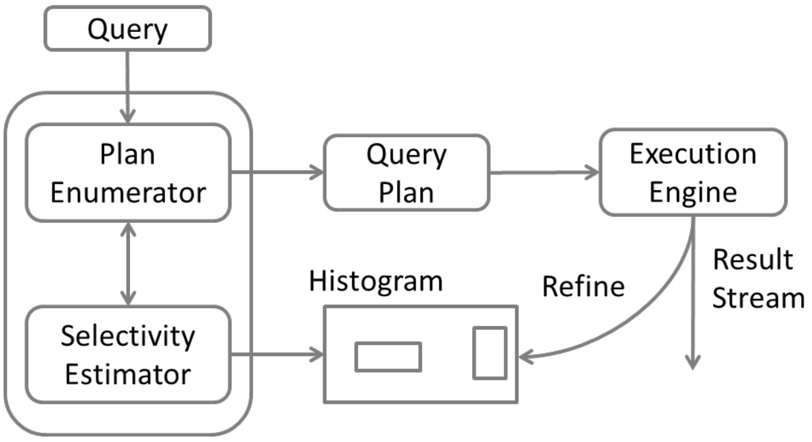


Figure 4.8: The query execution cycle and a self-tuning histogram

user issues a query, which is parsed by the DBMS and optimized. The optimizer accesses the histogram during the optimization. Then, the system executes the query. The user receives the query result stream; the same results are also available to the histogram. The histogram uses these results to refine its structure.

We now describe a self-tuning histogram, STHoles [BCG01], in detail. In this thesis we focus on self-tuning approaches, and STHoles is a prominent representative of such histograms. Several other multi-dimensional histograms, such as [RKC$^+$10, LZZ$^+$07, SHM$^+$06, FHL07] are based on STHoles (to a different degree).

4.5.1 The STHoles Histogram

The STHoles [BCG01] histogram is a self-tuning, multi-dimensional histogram. STHoles uses nested, non-overlapping buckets, partitioning the dataset into a tree, similar to R+ trees.

Figure 4.9 shows a histogram (on the left) and the corresponding bucket-tree on the right. STHoles usually has a "root" bucket which encloses the whole attribute-value space. Figure 4.9 it is the bucket r. Next to each bucket is the number of tuples

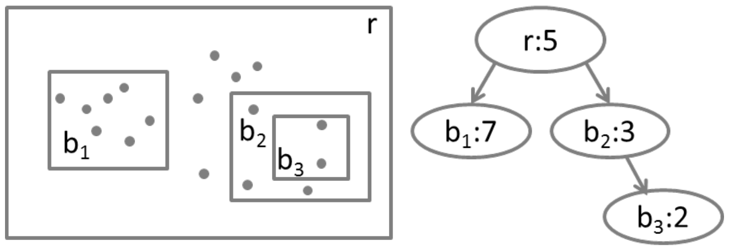


Figure 4.9: An STHoles histogram on the left and the bucket tree on the right

contained in the bucket. Note that this number does not include the tuples contained in child nodes. Indeed, we could include the child node tuple counts into a node tuple count. This would be an equivalent representation but some formulas which we introduce later would be more cluttered.

Definition 4.3 (Bucket Functions.)
We denote the number of tuples in a bucket b by $n(b)$. The bounding rectangle of a bucket is denoted by $box(b)$. The set of child buckets of b is $child(b)$. The volume of a bucket b is the volume of its bounding rectangle, minus the volume of child bucket bounding rectangles:

$$vol(b) = vol(box(b)) - \sum_{b_c \in child(b)} vol(box(b_c))$$

□

Example 4.1: Refer to Figure 4.9. $n(b_2) = 3$ and $vol(box(b_2)) = 5.83$. $n(b_3) = 2$ and $vol(box(b_3)) \approx 1.83$. We can calculate the $vol(b_2)$ from here: $vol(b_2) = 5.83 - 1.83 = 5$. ■

Thus, STHoles buckets are "responsible" for the area they cover excluding what is covered by child buckets. It is useful to think about child buckets as "holes" in the parent bucket.

The Algorithm 3 shows what happens to STHoles during a query execution cycle. Below we describe how each of the steps of the algorithm works in detail. We start with how STHoles estimates cardinalities (Section 4.5.1.1), then show how it adds new buckets using the query feedback (Section 4.5.1.2), and finally – how it compacts histogram, freeing up space to meet the space budget constraints (Section 4.5.1.3).

Algorithm 3: "Estimate, Refine, Compact" cycle for STHoles

```
Input: H: STHoles, q: Query
Output: H: Refined Histogram
  { Estimation}
  for all $b \in H$ do
    if $b \cap q \neq \emptyset$ then
      $estimate \leftarrow estimate + n(b) \cdot (vol(b \cap q)/vol(b))$
      $Intersections \leftarrow Intersections + (b \cap q)$
    end if
  end for
  $results \leftarrow q.Execute()$
  {Histogram refinement}
  for all intersection $(b \cap q) \in Intersections$ do
    Compute Tuples in $b \cap q$ using $results$
    Add new bucket(s) to $H$
  end for
  {Compacting the histogram}
  Remove buckets from $H$ to meet the space budget
```

4.5.1.1 Cardinality Estimation

STHoles uses the Continuous Value Assumption to approximate tuple cardinalities.

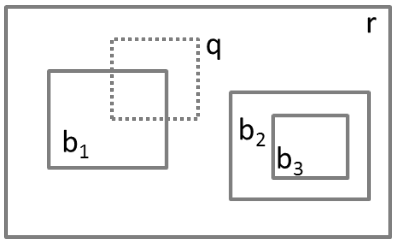


Figure 4.10: An STHoles histogram with query q

Refer to Figure 4.10. The query q is the dashed rectangle. It intersects with two histogram buckets – r and b_1. Applying the Continuous Value Assumption , STHoles estimates the number of tuples in intersection of a bucket and a query to be proportional to the volume of the intersection.

$$est(q \cap b) = n(b) \cdot \frac{vol(b \cap q)}{vol(q)} \tag{4.3}$$

Example 4.2: On Figure 4.10, the estimated number of tuples in $q \cap b_1$ according to

(4.3) would be

$$est(q \cap b_1) = 7 \cdot \frac{1}{4.45} \approx 1.57$$

■

To estimate the overall number of tuples in q, we sum up the estimated tuple counts for all intersections of q and histogram buckets:

$$est(q) = \sum_{b \in H} est(q \cap b) = \sum_{b \in H} n(b) \cdot \frac{vol(q \cap b)}{vol(b)} \tag{4.4}$$

Note that when the query does not intersect with a bucket ($q \cap b = \emptyset$), then $vol(q \cap b) = 0$.

4.5.1.2 Adding Buckets

Assume the query q intersects with histogram buckets $b_1, \ldots, b_k$. The cardinality estimate is computed using (4.4). After the query is executed, the *real* cardinalities are known for the intersections $q \cap b_j$ for $j = 1, \ldots, k$. Each of those intersections is a candidate for a new bucket. STHoles does not try to determine which of those buckets to drill and which not. It drills buckets for all intersections, and later removes the redundant ones.

In our example in Figure 4.10 the query intersects with the buckets r and b_1. After the query execution, the real cardinalities $r \cap q$ and $b_1 \cap q$ become known. Ideally, the histogram would add buckets in place of these two intersections. However, as we can see from Figure 4.10, this would be impossible because the intersection $r \cap q$ is not rectangular.

One workaround would be to partition $r \cap q$ into several rectangles. This is complicated, so STHoles takes another approach – it *shrinks* the intersection $r \cap q$ so that it has rectangular shape. Figure 4.11 shows the intersection $q \cap b$ and the two possible ways of shrinking it: across the X axis and across the Y axis.

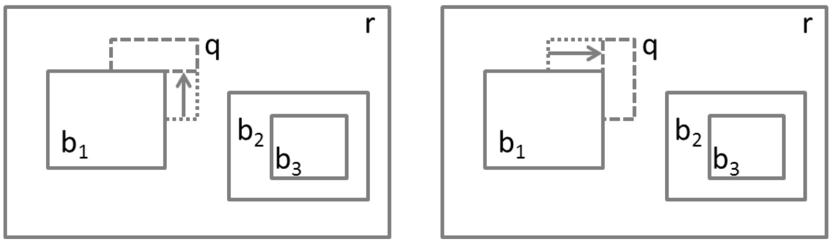


Figure 4.11: Two possible ways of shrinking $q \cap b$ into rectangular shape

Algorithm 4: Shrinking a query-bucket intersection

```
Shrink (Bucket $b$, Query $q$)
Candidate $c = q \cap b$
partiallyIntersected = $\{b_i \in children(b) : c \cap b_i \neq \emptyset \wedge b_i \nsubseteq c\}$
while partiallyIntersected $\neq \emptyset$ do
    Select $b_i \in$ partiallyIntersected and dimension $i$ such that shrinking $c$ along $i$ so
    that $b_i \cap c = \emptyset$ results in the smallest shrink
    Perform the shrink
    Update partiallyIntersected
end while
$c$.Frequency = $b$.Frequency $\cdot$ Volume($c$) / Volume($q \cap b$)
```

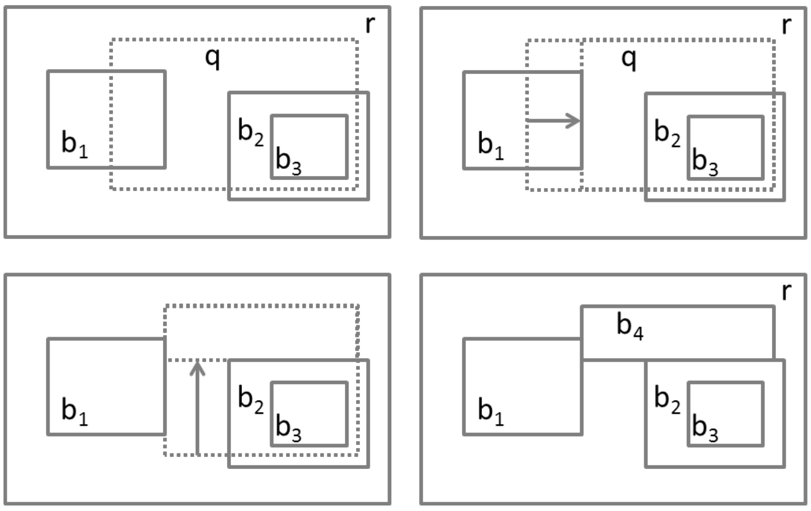


Figure 4.12: Progressive shrinking of $q \cap b$

Figure 4.12 shows a more complex case. Here, the query rectangle q partially intersects with both b_1 and b_2. We first shrink it horizontally, so that it excludes b_1 (top-right inner dashed rectangle on Figure 4.12). Then, we exclude b_2 by shrinking again, this time vertically (inner dashed rectangle on bottom-left). The final histogram with the added bucket is on the bottom-right. Note that when excluding bucket b_2, we could also have shrunk the outer dashed rectangle in bottom-left across the horizontal dimension. The vertical shrink is favored because the resulting rectangle has larger space.

4.5.1.3 Removing Buckets

STHoles adds buckets as it executes queries. One query can yield several buckets: this can happens when the query intersects with several buckets. The histogram rapidly grows in size. However, histograms usually are alloted restrictive space budget (the Requirement 2 on page 15). To meet the space requirements, the histogram has to deploy mechanisms to compact itself. STHoles does this by *merging* similar buckets.

At any point of time, the histogram represents a summary representation of the data. If we are going to compress it by removing buckets, we would expect our representation of the data to get worse. The idea behind the merging procedure of STHoles is to try to obtain a representation which is as close to the initial one as possible.

To achieve this, STHoles deploys penalty-based merging. Each potential merge is assigned a penalty, and the merge with lowest penalty is chosen. STHoles can perform parent-child or sibling-sibling merges. Parent-child merges are relatively simple. Let b_p be the parent bucket, b_c be the child bucket. The merge procedure simply adds the number of tuples of b_c to that of b_p, and removes b_c from the histogram. See Algorithm 5.

Algorithm 5: Parent-child merge

```
**ParentChildMerge** (Bucket $b_p$, Bucket $b_c$ )
$n(b_p) = n(b_p) + n(b_c)$
Transfer children of $b_c$ into children of $b_p$
$b_p$.Children.Remove($b_c$)
```

Merging sibling nodes is more complicated. The bucket which emerges as the result of the merge includes both initial buckets. Figure 4.13 demonstrates a merge

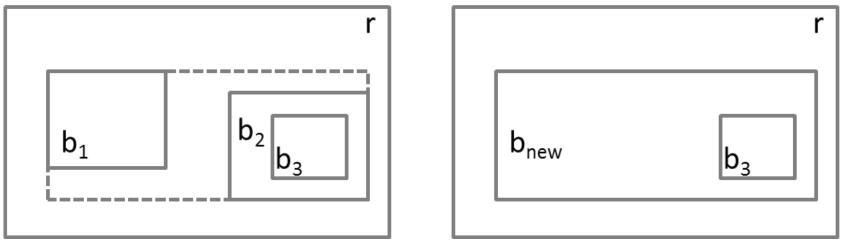


Figure 4.13: Sibling-sibling merge of b_1 and b_2

of two sibling buckets, b_1 and b_2. The minimum bounding rectangle which encloses both buckets is the dashed rectangle on left. That is the new bucket to be inserted into the histogram. The child nodes of b_1, b_2 are transferred into children of the new bucket. Notice that the new bucket includes part of the space of the parent region r. In order to estimate the number of tuples inside b_{new}, we simply take the number of tuples that the histogram would return as a cardinality estimate for a query $q = b_{new}$. A part of the volume and tuples from r falls into b_{new}. Before giving the

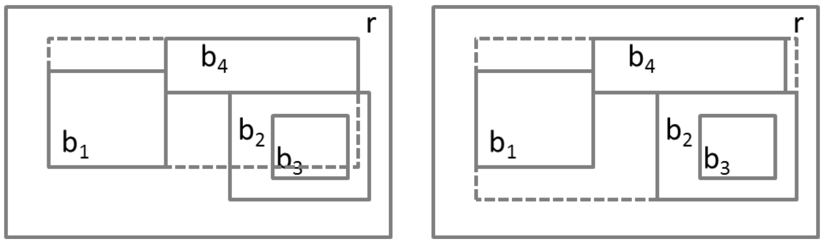


Figure 4.14: Sibling-sibling merge of b_1 and b_2

formulas for the updated bucket frequencies, we turn to another sibling-sibling merge example which is more complex. Refer to Figure 4.14. Here, we are merging the buckets b_1 and b_4. The minimum bounding rectangle of these buckets now intersects partially with other buckets, namely b_2 and b_3 (left on the picture). As b_3 is a child bucket of b_2, if we make sure that the new bucket does not intersect with b_2 it will automatically not intersect with b_3 as well. Thus, the problem is that the minimum bounding rectangle of two sibling nodes can have partial intersections with other siblings of the nodes that are merged. To avoid such partial intersections, STHoles *expands* the minimum bounding rectangle until it fully encloses partially intersecting buckets. In Figure 4.14, right, we can see the larger dashed rectangle which includes b_2. This will be the bounding box of the bucket resulting from the merge of b_1 and b_4.

A special case is when the bounding box of the extended merge rectangle is the same as the parent bucket rectangle. In this case, the sibling-sibling merge becomes equivalent to two parent-child merges.

We now go through the Algorithm 6 to explain some details of the sibling-sibling merge procedure. We start by computing the bounding rectangle of the new bucket, lines 2 to 5. The procedure `ExtendToExclude` in line 4 extends the rectangle br so that it includes b. This was demonstrated in Figure 4.14 (on the right side).

Line 6 checks whether the extended bounding rectangle br now equals the bounding rectangle of the parent bucket b_p. Figure 4.15 demonstrates this case. The bounding box of the buckets b_1 and b_2 equals to the bounding box of the parent bucket b_p. In this case, performing the parent-child merge is equivalent to merging b_p with b_1 and then the resulting bucket which would be a parent of b_2, with b_2.

In the `ELSE` branch of the algorithm we are performing the merge. First, we compute the volume of the parent region that is now covered by the merged bucket. The rectangle of the bucket to be created, br, includes b_1, b_2, and maybe some other child buckets of b_p. The volume not covered by the child buckets of b_p and covered by br is the volume v we are looking for:

$$v = vol(box(b_p)) - \sum_{\substack{b \in b_p.children \\ box(b) \subset br}} vol(box(b)) \tag{4.5}$$

Algorithm 6: Sibling-sibling merge

```
 1: SiblingSiblingMerge (Bucket $b_1$, Bucket $b_2$, Bucket $b_p$)
 2: $br$ = the minimum bounding rectangle of $b_1$ and $b_2$
 3: while ($\exists b \in b_p.children$) such that $br \cap b \neq \emptyset$ do
 4:    $br$ = ExtendToInclude($br$, $b$)
 5: end while
 6: if $br$ = $box(b_p)$ then
 7:    {The bounding $br$ equals the box of the parent bucket. This means a
       sibling-sibling merge becomes equivalent to performing two parent-child
       merges, of $b_1$ with $b_p$ and $b_2$ with $b_p$}
 8:    ParentChildMerge($b_p$, $b_1$)
 9:    ParentChildMerge($b_p$, $b_2$)
10: else
11:    $v$ = ParentVolume($br$, $b_p$)
12:    $n$ = $n(b_p) \cdot (v/vol(b_p))$
       {Estimate the frequency of the new bucket}
13:    $newFreq$ = $n(b_1) + n(b_2) + n$
14:    Create a new bucket with bounding rectangle $br$ and frequency $newFreq$
15:    $b_p = b_p - n$
       {Transfer child nodes}
16:    for all $b \in b_p.children$ such that $box(b) \subset br$ do
17:       $b_p.children$ = $b.p.children - b$
18:       $b_{new}.children = b_{new}.children + b$
19:    end for
20: end if
```

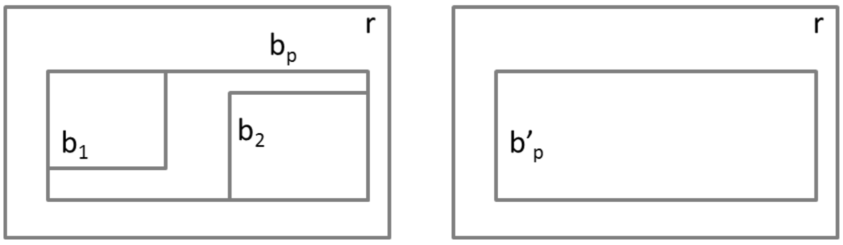


Figure 4.15: Sibling-sibling merge of b_1 and b_2

Next, we compute the number of tuples to be transferred from b_p to b_{new}, line 12. This is done using the Continuous Value Assumption .

We have described how STHoles performs merges. Given a bucket tree, every node can be merged with its parent bucket or with any of its siblings. As we mentioned, each merge is assigned a penalty and the merge with the smallest penalty is chosen by the histogram.

Merge Penalty. The idea of merge penalties is based on the estimation difference between the initial histogram and the one obtained after the merge. The assumption is that the current histogram is the best we can have, and we would like to pick the merge that results in a histogram which is most similar to the current one.

Definition 4.4 (Estimation Distance)
The estimation distance of H and H' is the cumulative difference in estimation of point queries:

$$\varepsilon(H, H') = \int_{u \in D} |est(H, u) - est(H', u)| du \tag{4.6}$$

where D is the attribute-value domain. □

$\varepsilon(H, H')$ is a measure about how different the histograms are. We would like to note that other distance measures, such as ones based on entropy or other metrics are conceivable too. The following definition of merge penalty will change respectively based on the distance measure.

Definition 4.5 (Merge Penalty)
Let $b_1, b_2 \in H$ such that they are either sibling nodes or parent-child nodes. Let $H' = Merge(H, b_1, b_2)$ denote the histogram which results from H if we merge b_1 and b_2. The Merge Penalty for merging b_1 and b_2 is

$$penalty = \varepsilon(H, H')$$

□

Next, we calculate the penalty for the parent-child and sibling node merges.

Parent-Child Merge Penalty. Let b_p be the parent node and b_c the child node. Looking at (4.6), notice that the estimation of the two histograms is the same everywhere except when $u \in box(b_p)$. Indeed, this is because the merge does not affect those regions. We denote the node obtained as the result of merging b_c and b_p as b_{new}. The penalty is

$$penalty(b_p, b_c) = \int_{u \in b_p} |est(H,u) - est(H',u)| + \int_{u \in b_c} |est(H,u) - est(H',u)| \tag{4.7}$$

According to the Continuous Value Assumption , the estimates $est(H, u)$ and $est(H', u)$ are constant for all $u \in b_p$. The same is true for all $u \in b_c$. Using this and Algorithm 5, we obtain

$$penalty(b_p, b_c) = |n(b_p) - n(b_{new}) \cdot \frac{vol(b_p)}{vol(b_{new})}| + |n(b_c) - n(b_c) \cdot \frac{vol(b_c)}{vol(b_n)}| \tag{4.8}$$

Sibling-Sibling Merge Penalty. The procedure of computing the sibling-sibling penalty is analogous to that of parent-child penalty. We compute the new bucket boundaries, and compute the error using the equation (4.6) and the Algorithm 6 for the new frequencies. For details, see [BCG01].

4.5.1.4 Discussion of STHoles

STHoles has several attractive features. It relies solely only on query feedback and amortizes the construction costs. The experimental comparison in [BCG01] shows that its estimation precision is comparable to the best static histograms. It adapts to the changes in the dataset smoothly.

Due to its attractive features, there has been significant follow-up work on STHoles.

STHoles+ [FHL07] focuses on memory-efficient bucket representation. The idea is to store bucket coordinates relative to a parent bucket, and also quantize the coordinates. This means the buckets cannot appear in arbitrary locations, but need to be aligned to a grid. This method allows significant memory savings.

[SHM$^+$06] uses a bucket structure similar to STHoles, and is discussed in Section 4.6.1.

Finally, [LZZ$^+$07] is about enhancing the convergence speed of the histogram for batch queries. We speak about this approach more in Chapter 6.

4.6 Consistent Selectivity Estimation

Histograms are a lossy compression of the data. A problem that often arises is that the available information allows several interpretations, which may be contradicting. This is the problem of *Inconsistency*. In this section we present two related approaches which deal with the problem of inconsistency. First is about constructing a consistent multi-dimensional histogram from available query feedback (Section 4.6.1). The other approach is about combining different pieces of incomplete statistics to obtain consistent global selectivity estimates (Section 4.6.2).

4.6.1 ISOMER: A Consistent Self-tuning Histogram

ISOMER [SHM+06] is a multi-dimensional, self-tuning histogram. It uses query feedback for consistent histogram construction.

The histogram treats available query feedback to construct the "most uniform" distribution which satisfies the constraints. In other words, it uses the maximum entropy principle to find the a probability distribution given the constraints.

We use the relation `Cars(ID, Model, Maker, Year, Color)`. Assume for simplicity there are only two car makers, 'Honda' and 'Volkswagen', and two colors, 'White' and 'Black'. There are four possible combinations of `Maker` and `Color`, and we will denote the respective probabilities $p_{HW}, p_{HB}, p_{VW}, p_{VB}$ (p_{HW} stands for the probability of a random tuple being a white Honda, and so on). The first constraint is:

$$p_{HW} + p_{HB} + p_{VW} + p_{VB} = 1 \tag{4.9}$$

which simply says that the probabilities sum up to 1. Let $I = \{HW, HB, VW, VB\}$. In order to find the individual probabilities we apply the maximum entropy principle:

$$-\sum_{i \in I} p_i \cdot \log(p_i) \to max \tag{4.10}$$

Given the constraints in place. At the moment our only constraint is given by (4.9). With only this constraint, the solution of the optimization problem (4.10) will be a distribution with all p_i equal to each other, so:

$$p_{HW} = p_{HB} = p_{VW} = p_{VB} = 0.25$$

Assume now that we execute the following query:

```
SELECT * FROM Cars
WHERE Maker = 'Honda'
```

and find out that the selectivity of the predicate `Maker = 'Honda'` is 0.8. This means that we now have the following constraint in addition to (4.9).

$$p_{HW} + p_{HB} = 0.8$$

Now, solving (4.10) yields

$$p_{HW} = p_{HB} = 0.4$$
$$p_{VW} = p_{VB} = 0.1$$

Further, if we execute another query:

```
SELECT * FROM Cars
WHERE Color = 'White'
```

Assume the selectivity of the predicate is 0.3. Figure 4.16 shows the available feedback.

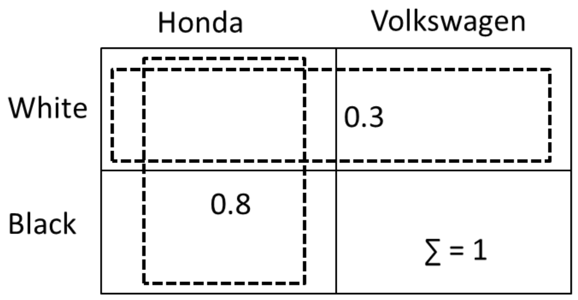


Figure 4.16: Available query feedback for the `Cars` relation.

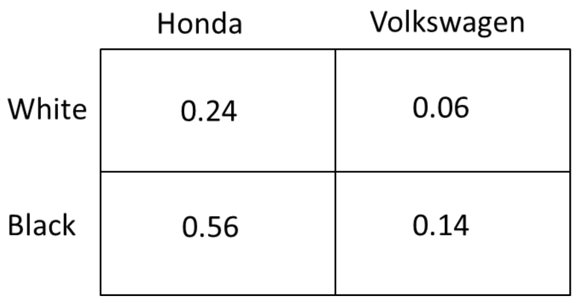
We have now the following constrained optimization problem:

$$\begin{aligned} &-\sum_{i \in I} p_i \cdot \log(p_i) \rightarrow max \\ &\sum_{i \in I} p_i = 1 \\ &p_{HW} + p_{HB} = 0.8 \\ &p_{HW} + p_{VW} = 0.3 \end{aligned} \tag{4.11}$$

This system can be solved using the Lagrange method. Usually, obtaining an analytical solution is impossible so ISOMER deploys an approximation method. The solution is depicted in Figure 4.17.

Figure 4.17 shows the available feedback.

Here we discussed how ISOMER works when there are two attributes and each takes only 2 distinct values. In a general case, ISOMER stores query feedback in a structure which is similar to STHoles. The difference is the bucket drilling procedure. STHoles aims at preserving existing buckets, so when new feedback arrives it shrinks



	Honda	Volkswagen
White	0.24	0.06
Black	0.56	0.14

Figure 4.17: Available query feedback for the `Cars` relation.

the new feedback to fit into the existing structure. ISOMER aims at storing feedback records "as is", without loss of information. Figure 4.18 shows a histogram with two buckets and a new query (the dashed rectangle). Figure 4.19 shows how STHoles

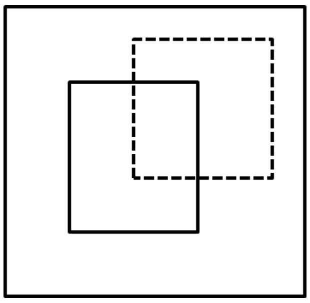


Figure 4.18: A new query intersecting with an existing bucket.

would add a bucket in after the query is executed. Figure 4.20 shows the bucket creation for ISOMER. Notice that in order to preserve existing bucket structure, the ISOMER approach had to make several small buckets out of the existing one.

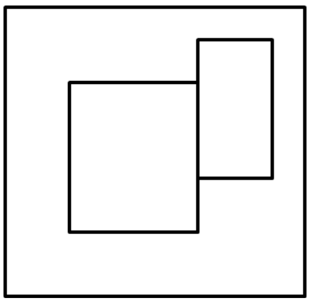


Figure 4.19: New bucket added according to the STHoles drilling procedure.

[SHM+06] shows how to rank the available query feedback in the order of importance, and remove the unimportant feedback as part of histogram compaction. It also

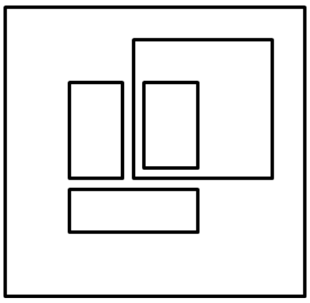


Figure 4.20: New bucket added according to the ISOMER drilling procedure.

provides a mechanism to resolve conflicts in different feedback records in a consistent manner.

4.6.1.1 Discussion of ISOMER

ISOMER is a multi-dimensional histogram based on the maximum entropy principle. It stores the feedback records without modification. Unlike STHoles, ISOMER does not count the tuples partial intersections of a bucket and a query. Instead, it deploys the maximum entropy principle to approximate the those tuple counts. Solving the constrained optimization problem and the usage of maximum entropy principle ensure the solution is consistent with existing feedback.

However, it is a very complicated technique, and that is probably one of the reasons there has been much follow-up work in this direction. Unfortunately, [SHM$^+$06] also does not provide experimental comparison with STHoles. The costs for computing the maximum entropy solution is super-linear in the experiments in [SHM$^+$06] and can be $O(n^2)$ in the worst case.

4.6.2 Consistent Estimates from Partial Statistics

Assume we have a relation with three attributes. However, a full three-dimensional statistics is not available, instead we have two dimensional statistics on attributes $\{1, 2\}$, $\{1, 3\}$ and $\{2, 3\}$, and we have all three single-dimensional statistics. The statistics can be in form of histograms or samples or any other form of statistics which supports selectivity estimation. Now assume we have a three-dimensional predicate and we want to compute its selectivity. We can use one of the two-dimensional statistics, say the one on attributes $\{1, 2\}$, and multiply it with the statistics on the third attribute using the independence assumption. We will write this as $s_{1,2} * s_3$. Other alternatives are $s_{1,3} * s_2$, $s_{1,2} * s_3$ or $s_1 * s_2 * s_3$. Clearly, we can arrive at situations where $s_{1,2} * s_3 \neq s_{1,3} * s_2$. So the choice of which statistics to use affects the estimated selectivity of a predicate.

This kind of scenario can occur frequently when the dimensionality of the relation is large and storing the full-dimensional histogram is excessively expensive.

[MHK$^+$07] shows that this can over-complicate the optimizer with needless heuristics while not fully solving the problem.

Instead, they propose a maximum-entropy solution for finding the selectivity of a predicate. Given a set of simple predicates $P = \{p_1, \dots, p_n\}$, the problem is to find the selectivity of an arbitrary disjunctive normal form (DNF) on alphabet P. First, the selectivities are interpreted as probability measure (see Section 3.2.1 in Chapter 3). In some cases, the selectivity is known from an existing statistic. These known selectivities (probabilities) form *constraints*. Then, we look for a distribution which satisfies the constraints and maximizes the entropy. The reason for this is that usually there are infinite number of distributions which satisfy the constraints. Picking the one with maximum entropy ensures that we do not make any "additional assumptions" about the dataset beyond what is known (these are the constraints).

This approach has the nice property that it is possible to find inconsistent information in the statistics and remove it.

4.7 Assessing Histogram Precision

In this chapter we have discussed numerous histograms. The subject of this section is the experimental evaluation and comparison of different histograms. The histogram evaluation framework is based on several components. These are the *error metric*, the *data set*, and the *query workload*. So a single unit of histogram-evaluation experiment is: "Compare these two histograms on the given dataset, using the given query pattern and the given error metric". We now briefly discuss why each of these components are important during evaluation.

Data Sets. Some histograms try to exploit certain statistical properties in the data which can be specific to an application. Such statistical properties allow certain techniques to achieve high estimation precision for select, domain-specific datasets. Clearly, the statistical properties of the data change from one application domain to another. Thus, we have to always include the data context on which we compare the histograms.

Queries. The same holds for the query pattern – querying patterns differ from one application domain to another. The queries are particularly important for self-tuning histograms where all the information the histogram can access is the query result-stream.

Metrics. The main criteria for assessing the usefulness of a histogram is the accuracy of the estimations issued by the histogram. Thus, the metrics which are used to compare histograms should be based on estimation accuracy (or, equivalently, the estimation error). Some application might use a threshold-error metric: when the error is below certain threshold, it does not influence the correct choice of the query plan. Otherwise, the optimizer makes a wrong decision. In this scenario, the only thing that matters is whether the error is below or above a threshold value. Another evaluation metric can be the average error, where the rationale is that the probability of choosing a suboptimal plan is proportional to the amount of estimation error.

4.7.1 Error Metrics

In order to give definitions of the error metrics we need to define formally what a workload is. We will discuss workload generation later in this section.

Definition 4.6 (Query Workload)
A query workload is an ordered sequence of queries:

$$W =< q_1, \ldots, q_n >$$

□

Definition 4.7 (Estimated and Real Cardinality)
Given a dataset D, histogram H and a query q, we write $card(q, D)$ to denote the cardinality of q and $est(q, D, H)$ to denote the estimated cardinality of q using the histogram H. □

Here, we discuss only the average-error based error metric. This is the most commonly used error metric in the literature.

Definition 4.8 (Average Absolute Error)
The average absolute error of the histogram H on dataset D using the workload W is

$$\epsilon(H, D, W) = \frac{1}{|W|} \sum_{k=1}^{|W|} |card(q_k, D) - est(q_k, D, H)| \tag{4.12}$$

□

In order to be able to compare the errors across datasets, [BCG01] employs the following trick: they normalize the average absolute error. As the normalizer, they take a 1-bucket histogram which uses the uniformity assumption, called H_0.

Definition 4.9 (Normalized Absolute Error)
The normalized absolute error is the average absolute error normalized against using

the single-bucket histogram H_0:

$$\varepsilon(H, D, W) = \frac{\epsilon(H, D, W)}{\epsilon(H_0, D, W)} \tag{4.13}$$

□

4.7.2 Data Sets.

Now we describe several the five data sets that we use in our experiments. Three of them are synthetic and two are real-world.

4.7.2.1 Synthetic Data Sets

The *Cross* dataset. The simplest dataset is the *Cross* dataset (Figure 4.21). It is a two-dimensional dataset which consists of two one-dimensional clusters and noise. The overall number of tuples in the dataset is 22,000. The clusters contain 10,000 tuples each, the remaining 2,000 tuples are random noise.

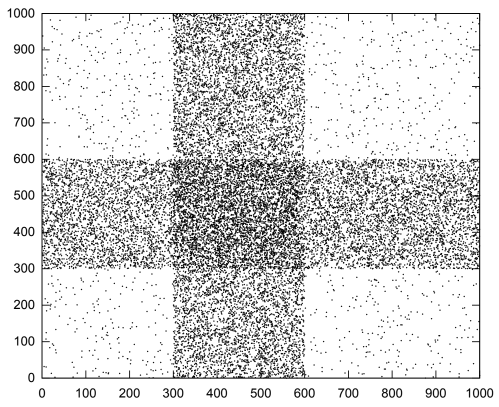


Figure 4.21: The *Cross* dataset

The *Array* dataset. This dataset models a discrete dataset where each tuple can appear multiple times. It is based on the Zipfian distribution [Zip49]. For a distribution with N total points, the probability of the k-th member is given as:

$$f(k, N, z) = \frac{1}{S} \cdot \frac{1}{k^z} \tag{4.14}$$

where z is the "Zipfian skew" parameter and S is a normalizing constant such that the probabilities sum up to 1:

$$S = \sum_{k=1}^{n} \frac{1}{k^z}$$

Algorithm 7: Generating the *Array* dataset

```
Generate $d^n$ points, store into $data$
$freqArray$ = ZipfianDistribution($d^n$, $z$)
Randomly shuffle the elements of $freqArray$
Assign the frequency of $data[i]$ to be $freqArray[i]$
```

Algorithm 7 shows the algorithm for generating an n-dimensional *Array* dataset with t tuples and skew parameter z. The number of distinct values per dimension, d, is another parameter. We generate d points for each dimension, thus, the overall number of points generated is d^n (this has to be less than t). We store the points in the array $data$. Now we generate a frequency array for these points. First, we generate a probability distribution $f(k, d^n, z)$ according to (4.14), let this be $freqArray$. Next, we randomly shuffle the elements of $freqArray$. Now we assign the frequency of the i-th data point, $data[i]$, to be $freqArray[i]$.

The *Gauss* data set. This dataset contains multi-dimensional Gaussian bells and models continuous data. The dimensionality of the dataset, the number of tuples and the number of the bells can vary. The number of tuples in the bells are distributed by a Zipfian distribution. Let the number of tuples in the dataset be t, the Zipfian skew is z and the number of bells is b. This means that b numbers are generated according to a Zipfian distribution with a skew z. These b numbers sum up to 1. We multiply all the numbers by t: these are our tuple counts for the individual bells. We then randomly assign each of those numbers to one of the bells.

Figure 4.22 shows a 2-dimenional *Gauss* dataset with 500,000 tuples, 30 bells and Zipfian skew = 1. Figure 4.23 shows the Gauss dataset with 200,000 tuples, 30 bells, and Zipfian skew = 0.5.

As we saw in the discussion above, many histograms use the Continuous Value Assumption to estimate the number of tuples in the dataset. The normal distribution used in to generate the bells is far from uniform. The purpose of the *Gauss* dataset is to see whether histograms which use the Continuous Value Assumption can handle such far-from-uniform distributions.

4.7.2.2 Real-World Data Sets

The *Census* Dataset. This dataset is a trimmed version of the U.S. Census bureau data. It contains a little over 210,000 tuples. It is two-dimensional, and contains

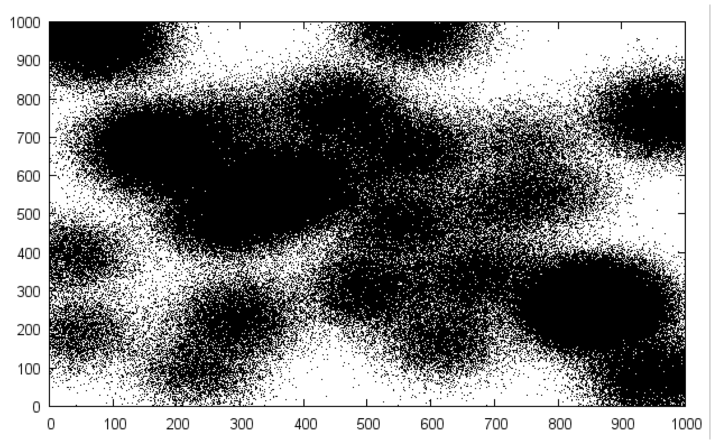


Figure 4.22: The *Gauss* dataset with 500,000 tuples

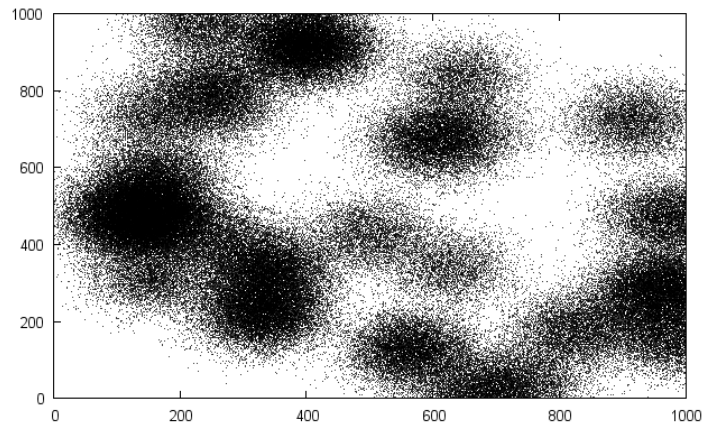


Figure 4.23: The *Gauss* dataset with 200,000 tuples

the attributes `Age` and `Income`. The `Age` varies between 14 to 90, the `Income` between -25897 and 347998. Figure 4.24 shows the Census dataset.

The *Sky* Dataset. This dataset is adapted from one of the datasets published by Sloan Digital Sky Survey [SDS11].

The dataset contains approximately 1,7 million tuples of astronomical observations. There is a categorical attribute called "class" which we removed from the relation. The remaining dataset is 7-dimensional. The first two dimensions are the coordinates of an object in the sky, the next five columns are brightness data – passed through different filters. Complex data correlations exist in the *Sky* dataset. There are several full-dimensional clusters, as well as subspace clusters in different projections of the data.

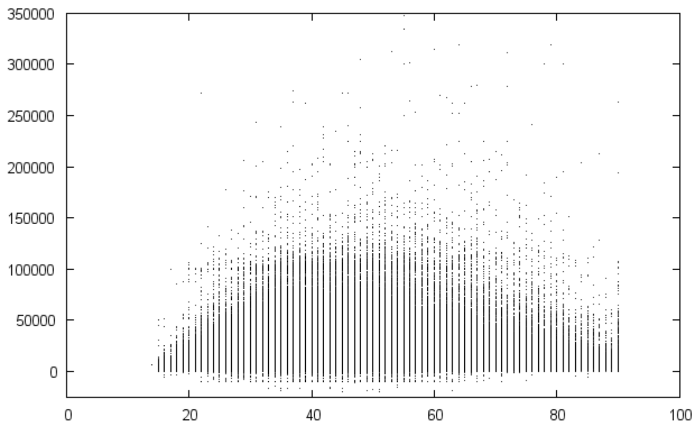


Figure 4.24: The *Census* dataset

4.7.3 Workloads

The patters according to which we generate the workload comes from [PSTW93].

The queries in the workload are range queries. The query pattern is a pair $(C, R[c])$. C stands for the query center distribution. $R[]$ is a constraint on the volume of the query, and c is a parameter to the constraint-function.

The query centers are generated according to one of the patterns:

- **Data**: The queries follow the data distribution.
- **Uniform**: The queries follow a uniform random pattern.

The volume constraint $R[]$ can be one of the following:

- $T[c_t]$: Each query contains $c_t\%$ of overall tuples.
- $V[c_v]$: Each query contains $c_v\%$ of overall data volume.

Assume we are using the *Array* dataset. Then, $Array(Uniform, T[1\%])$ means that the query centers are distributed uniformly, and each query rectangle includes 1% of the tuples.

For the self-tuning histograms, the query workload typically consists of two parts: the *training* part and the *evaluation* part. This means we use the training queries only for learning. The actual error computations start with the *evaluation* workload. In our experiments, we typically provide 1,000 training and 1,000 evaluation queries, and we typically use 1% volume or tuple queries. For instance, with a $(Uniform, V[1\%])$ workload consisting of 1,000 queries, a unit volume in the dataset is covered 10 times on average. Similarly, $(Uniform, T[1\%])$ covers each tuple 10 times on average. Such learning volumes should be sufficient for the self-tuning histograms to learn the dataset. Similarly, 1,000 evaluation queries are enough to accurately measure the histogram error.

4.8 Other Uses of Histograms

Histograms have multiple uses in databases beyond traditional selectivity estimation. Here we briefly talk about some database-related applications of histograms.

4.8.1 Skyline Queries

Skyline is a set of points in a multi-dimensional data space such that each point is not dominated by some other point in all the dimensions [PTFS03].

Example 4.3: Assume we are searching for a cheap hotel, which is as close to the beach as possible. Figure 4.25 shows hotels as circles. The line passes through the solid circles which are the hotels belonging to the skyline.

For instance, hotel 1 is dominated by hotel 2. They have the same distance from the beach while price of 2 is lower compared to 1. Similarly, hotels 3 and 4 have the same price but 3 is closer to the beach. ■

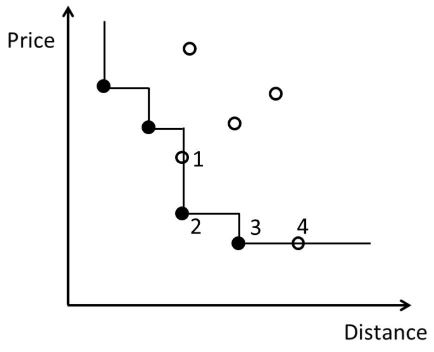


Figure 4.25: A hotel skyline

Histograms are used to compute *approximate skylines*. This can be useful, for instance, for giving the user an rough idea about how the skyline looks like before actually computing it. Assume we have a multi-dimensional histogram H. For each bucket, [PTFS03] computes a *hypothetical point*, which is essentially a point with the lowest expected coordinates along all dimensions. Figure 4.26 shows a rectangular bucket and a hypothetical point p. All the other points in the bucket are in the grey region; the point p dominates them. Note that because we are talking about the *expected* coordinates of the point p, we can readily conclude its on the diagonal of the bucket rectangle.

We now show how the coordinates of the hypothetical point are calculated. First, normalize the dimensions of the rectangle in Figure 4.26 so that they both have unit

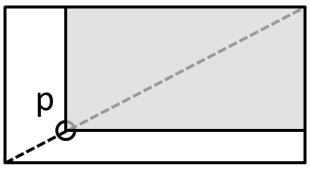


Figure 4.26: A hypothetical point and its dominated region (in grey)

length. Then, the coordinates of the p are (x, x). In order to compute the expected value of x, we assume the tuples inside the bucket are uniformly distributed. If $x > \xi$, this means all the tuples enclosed in the bucket fall into a square with sides $1 - \xi$. The probability of a tuple falling into a such a square is $(1 - \xi)^2$

If there are N tuples in the bucket, the probability that all of them fall into that square is $(1 - \xi)^{2N}$. The expected value of x is thus:

$$E[x] = \int_0^1 \xi \cdot dP(x \leq \xi) = \frac{1}{2 \cdot N + 1}$$

Similarly, for a d-dimensional space instead of a 2-dimensional we get.

$$E[x] = \frac{1}{d \cdot N + 1}$$

Given a histogram with B buckets, we obtain the approximate skyline by computing the hypothetical points of each bucket and then the skyline of these points. Figure 4.27 shows a multi-dimensional histogram on the left. On the right is the skyline of the hypothetical points obtained from each bucket.

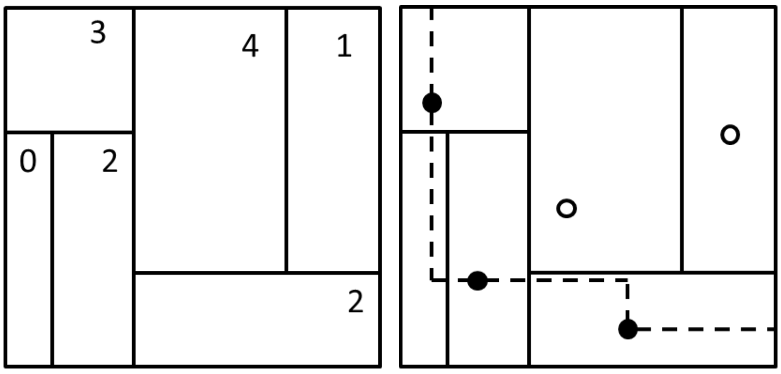


Figure 4.27: The histogram on the left and the approximate skyline of the hypothetical points on the right.

4.8.2 Top-k Queries

In many scenarios, the user specifies a query predicate but does not require *exact matches*. Typically the user expects a small number k of *closest matching* tuples to be returned. Such queries are called *Top-k* queries. Queries against a web search engine are the most prominent example of *Top-k* queries.

Here, we describe a solution from [BCG02] which focuses on supporting *Top-k* queries on top of a relational database system. The challenge here is to transform the *Top-k* query into a relational selection query. This would utilize the query optimizer and the execution engine of a RDBMS.

A *Top-k* query has the form

```
SELECT TOP k * FROM R
WHERE A1=v1 AND ...  AND An=vn
ORDER BY dist
```

Here, *dist* is a distance function which allows to rank approximately matched queries according to their distance from the point $(A_1 = v_1) \wedge \ldots \wedge (A_n = v_n)$.

The execution strategy of a *Top-k* query is as follows:

- **Search.** Estimate a distance d such that the region with a center in $(v_1, \ldots, v_n)$ and radius d contains at least k tuples. This is done using a multi-dimensional histogram.
- **Retrieve.** Execute the query and retrieve the tuples.
- **Verify/Restart.** Verify that the query contains k tuples. If not, restart the query with a larger d.

We will briefly describe the **Search** phase here. This method assumes there is a multi-dimensional histogram H present. The histogram should contain rectangular buckets. The search process consists of three-steps:

1. Create a synthetic relation R' which is consistent with the histogram H. R' has a tuple for each bucket in H, which is duplicated as many times as the count of the tuples inside the bucket.
2. Compute the distance of each tuple in R' from the query.
3. Pick the minimal d such that the radius d around q contains k tuples.

There are several ways to construct the relation R' from H. A pessimistic strategy is to place the tuples representing the bucket as far from q as possible. If we compute the distance d_{NR} using this strategy, we are guaranteed there will be no restarts. An optimistic strategy would be to construct the smallest possible d by putting all the tuples as close to q as possible. Denote this distance as d_R. A parametric method of choosing a distance would be:

$$d(\alpha) = d_R + \alpha \cdot (d_{NR} - d_R)$$

for some $0 \leq \alpha \leq 1$. [BCG02] discusses how to choose α looking at the query workload so that there are few restarts and, in the meantime, $d(\alpha)$ is not too large.

4.9 Summary

Histograms are the main data structure used for selectivity estimation in databases. They have numerous other applications in databases as well. Multi-dimensional histograms capture the joint attribute-value distribution for multiple attributes. They avoid the attribute value independence assumption which usually leads to largely inaccurate estimates. The flip side is that the construction and maintenance costs of multi-dimensional histograms are high.

Self-tuning histograms amortize the construction costs by looking only at query feedback. We discussed in detail the STHoles histogram in Section 4.5.1.

Self-tuning histograms in general and STHoles in particular assume that it is possible to "learn" the dataset from the scratch, i.e. starting with an empty bucket set. This has been a central assumption behind self-tuning methods, that an initial configuration, if provided, would only be of limited use.

Our focus on the next two chapters is this assumption. We show it does not hold. Initial configuration is important, and can have a permanent impact on histogram structure and precision. That is, if started with a carefully chosen initial configuration, the histogram benefits more from subsequent learning. It converges to state where the average error rate is considerably lower compared to the uninitialized version. This does not change even if the uninitialized version gets practically unlimited amount of training queries.

5 Histogram Initialization

Abstract. *Multi-dimensional histograms try to detect dense regions in the data space. Subspace clustering algorithms have a similar goal. In this chapter, we show that subspace clustering algorithms can be used to initialize a self-tuning, multi-dimensional histogram. We transform the clusters into histogram-native representation. The key is to preserve as much information as possible in this transformation. We derive a formal criterion for assessing the quality of transformations. This criterion allows to find the optimal transformation. In practice, however, this is expensive, so we propose a heuristic. Our experiments show that our initialization technique can significantly increase the estimation precision of resulting histograms.* □

5.1 Introduction

In Chapter 3 we discussed histograms as one of the main techniques used for selectivity estimation.

Self-tuning histograms are state-of-the-art methods for selectivity estimation. They use the query execution results (feedback) to refine themselves [BCG01, SHM$^+$06, FHL07, LZZ$^+$07]. They focus on the refinement steps during query processing to achieve high precision, arguing that even a good initial configuration provides only a short-term benefit. Thus, a central hypothesis with self-tuning histograms has been that their refinement techniques are enough to ensure high precision, while initialization is a minor tweak.

We show that this is only one side of the coin: Doing without initialization techniques has a serious drawback. First, histograms need many queries in order to adapt to the data set. Second, even given a large number of queries to train, the uninitialized histogram still cannot match the precision of our initialized version.

Another problem with multi-dimensional histograms is that they focus on capturing correlated data regions in full-dimensional space. This can be wasteful, because in different regions of the data space only a small set of attributes may be correlated, while additional attributes only add noise. Thus, traditional selectivity estimation methods spend too much memory and achieve only low estimation quality. Similar challenges have been observed for traditional clustering and solved by recent subspace clustering techniques.

In this chapter we focus on pre-processing steps to initialize a self-tuning histogram based on subspace clustering. Having detected high density regions (dense subspace clusters) with many objects we build memory-efficient histograms based on this. We make use of subspace clustering as a novel data mining paradigm. As highlighted by a recent study [MGAS09], subspace clustering can detect groups of objects in any projection of high-dimensional data. In particular, the resulting subspace clusters represent dense regions in projections of the data and capture the local correlation of attribute sets for each cluster individually. Thus, our hypothesis is that these dense subspace regions can be utilized to initialize the histogram and enable good selectivity estimations. Our experiments confirm this hypothesis. In order to initialize a histogram with dense subspace regions we have to transform subspace clusters into efficient histogram structures, in order to meet the memory constraints of the histogram. There are various ways to do this, but any transformation introduces some estimation error. We need to minimize this error. To this end, we formally derive a criterion which lets us compare different transformations and choose the better one. We also define special classes of transformations with interesting properties. For these classes, we are able to compute the transformation which is best according the aforementioned criterion. We show however that finding the optimal solution is too expensive in the general case. We propose an efficient heuristic with high quality estimation.

As mentioned above, a central hypothesis with self-tuning histograms has been that initialization yields only a short-term benefit [SHM$^+$06, BCG01]. We use six subspace clustering algorithms [MGAS09] to initialize a histogram, and use the uninitialized version as a baseline. We make the following important observations:

1. *Good* initialization makes a difference. One out of the six methods we tried has shown consistent improvement of estimation precision over the uninitialized version, even after a significant number of training queries.

2. Self-tuning histograms can achieve high precision using refinement. Namely, an uninitialized self-tuning histogram was able to catch up with the remaining five initialized histograms, each based on different subspace clustering paradigm.

A related observation is that initialized histograms need less memory to provide similar or even better estimation quality. Through different evaluation settings, they need about $1/4$ to $1/8$ of the memory required in uninitialized histograms to produce estimates of the same precision.

5.2 Subscpace Clustering

Clustering is an unsupervised data mining task for grouping of objects based on mutual similarity [HK01]. As an unsupervised task, it reveals the intrinsic structure of a data set without prior knowledge about the data. Clusters share a core property with histograms, as they represent dense regions covering many objects. However, the detection of meaningful clusters in high-dimensional data spaces is hindered by the "curse of dimensionality" as well [BGRS99]. Irrelevant attributes obscure the patterns in the data. Global dimensionality techniques such as Principle Components Analysis (PCA) try to reduce the number of attributes [Jol86]. However, the reduction may yield only a clustering in one reduced space. With locally varying attribute relevance, this means that clusters that do not show up in the reduced space will be missed.

Recent years have seen increasing research in subspace clustering, which aims at identifying locally relevant attribute sets for each cluster. Subspace clustering was introduced in the CLIQUE approach, which detects dense grid cells in subspace projections [AGGR98]. In the past decade, several approaches have extended this clustering paradigm [SZ04, PJAM02, YM03]. Furthermore, traditional clustering, such as the k-medoid and DBSCAN algorithm, have been extended to cope with subspace projections [AWY$^+$99, KKK04]. For example, density-based subspace clustering detects arbitrarily-shaped clusters in projections of the data space [KKK04, AKMS07, AKMS08, MAG$^+$09]. Overall, various cluster definitions focusing on different objective functions have been proposed.

In the following we will show how these clusterings can be transformed into a memory-efficient histogram and provide high quality initializations for selectivity estimation. As a common property, we use the ability of subspace clustering to detect high density regions in projections of high-dimensional data. We abstract from specific clustering models, focusing on compactness, density-connected and arbitrarily shaped clusterings. This makes our general transformation of subspace clusters into histograms applicable to a wide range of subspace clustering algorithms.

5.3 Cluster Transformation

The goal of histogram construction is to have a concise summary of data which enables precise selectivity estimates. Clustering algorithms in turn may report clusters in different ways, often by simply listing all elements. Furthermore, clusters can have different shapes, depending on the definition.

In order to make clusters usable for selectivity estimations, we need to transform clusters into memory-efficient histogram structures. The goal of such a transformation is to obtain a histogram which produces estimation error as small as possible. In this section, we address this issue first from a theoretical and then from a practical point of view. Namely, we

- formalize the transformation of a cluster to a bucket
- define classes of transformations with useful properties

- show that strictly optimal transformations are overly expensive
- introduce heuristics which find good representations

Histogram buckets usually have a strict form, e.g., are axis-aligned and rectangular. Our goal is to transform the output of the clustering algorithm to a set of rectangles. We call these rectangles *Representative Rectangles*, or RRs.

One seemingly straightforward idea is to use minimal bounding rectangles as RRs. However, as our theoretical analysis shows, such RRs can be far from optimal or even useless. This emphasizes the importance of choosing RRs carefully. We formalize the notion of quality of transformation and optimality in Section 5.3.1 and present several formal results regarding optimal RRs. We use these results in 5.3.2 when we calculate RRs.

5.3.1 The Optimal RR for Selectivity Estimation

Histogram buckets span (axis-aligned hyper-)rectangles in attribute-value space. A cluster is a set of points: our aim is to transform it to a rectangle while making sure that the transformation "falsifies" the cluster as little as possible.

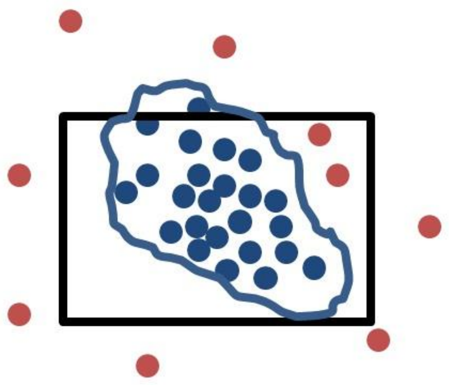


Figure 5.1: A cluster and a candidate RR

We denote the set of all rectangles in the data space as $\Re$. $\Re$ can be finite or infinite, depending on the data domain. The transformed rectangles serve as histogram buckets. In the histogram we essentially substitute the cluster C with RR. Figure 5.1 shows a cluster C and a candidate RR. We now look at clusters not as a discrete set of points but as regions with an extent in space and density, to bring rectangles and clusters into the same domain.

Definition 5.1 (Cluster Density)
Given a cluster C, we denote by $|C|$ the volume of its extent. The density of the cluster, $dens(C)$, is the number of objects in the cluster divided by $|C|$. □

Because $C \neq RR$ in general, substituting C with RR introduces an estimation error. Suppose that the density of the cluster is $dens(C)$, and outside of the cluster it is roughly 0, and the density of RR is $dens(RR)$. Then, as a result of substituting C with RR, the following density changes occur:

- $RR - C$ has density 0, but instead we estimate its density to be $dens(RR)$
- $C - RR$ has density $dens(C)$, instead we estimate its density to be 0.
- $C \cap RR$ has density $dens(C)$, instead we estimate its density to be $dens(RR)$.

The overall estimation error resulting from the substitution of a fixed C with RR is given by the function $\epsilon(RR, dens(RR))$. It is the sum of errors of the three regions mentioned above:

$$\begin{gathered}\epsilon(RR, dens(RR)) = \int_{RR\cup C} |est(u) - real(u)|\, du = \\ \int_{RR\cap C} |dens(RR) - dens(C)|\, du + \int_{RR-C} dens(RR) du + \int_{C-RR} dens(C) du = \\ (|dens(RR) - dens(C)|)\, |RR \cap C| + dens(RR)\, |RR - C| + dens(C)\, |C - RR|\end{gathered} \tag{5.1}$$

Definition 5.2 (Optimal RR)
A rectangle RR with density $dens(RR)$ is called optimal (w.r.t. $\Re$), denoted by $RR = opt(\Re)$ if

$$\epsilon(RR, dens(RR)) = \min_{r\in\Re} \epsilon(r, dens(r))$$

□

We first prove that the density of $opt(\Re)$ is upper-bounded by the density of the cluster:

Lemma 5.3.1. *For any cluster C with density $dens(C)$, if $RR = opt(\Re)$, then $dens(RR) \leq dens(C)$*

Proof. Let us assume that the opposite is true, for some $\alpha > 0$
$dens(RR) = dens(C) + \alpha$, RR is optimal, which means

$$\epsilon(RR, dens(R)) = \alpha\, |RR \cap C| + dens(C)\, |C - RR| + (dens(C) + \alpha)\, |RR - C|$$

is minimal. Take $dens'(RR) = dens(C) - \alpha$,

$$\epsilon'(RR, dens'(R)) = \alpha\, |RR \cap C| + dens(C)\, |C - RR| + (dens(C) - \alpha)\, |RR - C|$$

$\epsilon'(RR, dens'(R)) < \epsilon(RR, dens(R))$, which contradicts the assumption that ϵ is minimal. □ □

Figure 5.1 illustrates why $dens(RR)$ should not exceed $dens(C)$. RR possibly contains regions which are not in C, and does not necessarily cover all of C. So instead of some part of C with high density, RR contains a part which has density 0.

Using Lemma (5.3.1), we can simplify Equation (5.1)

$$\epsilon(RR, dens(RR)) = dens(C) \cdot |C| + dens(RR) \cdot (|RR - C| - |RR \cap C|) \quad (5.2)$$

We can now find the expression for the optimal value of $dens(RR)$.

Lemma 5.3.2. *For a fixed rectangle RR, the value of* $dens(RR)$ *which minimizes* $\epsilon(RR, dens(RR))$ *is given by:*

$$dens(RR) = \begin{cases} dens(C) & \textit{if}\ |RR \cap C| > |RR - C| \\ 0 & \textit{otherwise} \end{cases}$$

Proof. In Equation (5.2), the part depending on $dens(RR)$ is

$$dens(RR) \cdot (|RR - C| - |RR \cap C|)$$

In case $|RR - C| > |RR \cap C|$, it is positive. To minimize it, we put $dens(RR) = 0$. In case $|RR \cap C| > |RR - C|$, it is negative, and we put $dens(RR) = dens(C)$, which is the largest value for $dens(RR)$ according to Lemma (5.3.1). □

The first implication from this lemma is that if $|RR - C| > |RR \cap C|$ then the rectangle RR is not useful and can be omitted. RR is useless when the space contained in RR not belonging to C is larger than the common part of C and RR (Figure 5.1). However, when $|RR \cap C| > |RR - C|$, then the best strategy is to minimize the estimation for the region $|RR \cap C|$. This is achieved by putting $dens(RR) = dens(C)$. Below, we always consider RRs which satisfy $|RR \cap C| > |RR - C|$, and their density $= dens(C)$. Finding the optimal RR is not straightforward, however. Before turning to optimal RRs, we discuss some "obvious" RRs, such as minimal bounding rectangle.

Definition 5.3 (Enclosing Rectangles)
We denote the set of all rectangles which enclose C by $\Re_C^+$.

$$\Re_C^+ = \{R | R \in \Re, C \subseteq R\} \quad (5.3)$$

□

Obviously, the minimal bounding rectangle of C is in $\Re_C^+$.

Definition 5.4 (Enclosed Rectangles)
We denote the set of all rectangles enclosed in C by $\Re_C^-$

$$\Re_C^- = \{r | r \in \Re, r \subseteq C\} \quad (5.4)$$

□

The maximal inbound rectangle of a cluster is in $\Re_C^-$.

Lemma 5.3.3. *$\Re^+$ contains a unique optimal RR or is empty.*

Proof. We construct a rectangle R_0 such that $\forall R \in \Re_C^+,\ R_0 \subseteq R$. For dimension j, project all points on j, find the minimum and maximum – those would be the sides of the rectangle parallel to dimension j. Repeating this for all dimensions we will obtain the rectangle. Obviously, any rectangle in $\Re_C^+$ contains R_0. If R_0 satisfies the condition $|R_0 \cap C| > |R_0 - C|$ then $R_0 = opt(\Re_C^+)$, otherwise $\Re_C^+$ does not contain any RRs. □

Consider again Figure 5.1 for an example in 2-dimensional space. To find the minimal rectangle in $\Re^+$, take the up-most point of the cluster and draw a line parallel to the x-axis, do the same with the lowest point. Now, take the rightmost point and draw a line parallel to the y-axis, same with the leftmost point. The rectangle which is bounded by those 4 lines is R_0.

We now proceed as follows: We first present an algorithm which finds $opt(\Re_C^-)$. It constructs a convex hull of the cluster and fits the largest RR into it. In practice, this approach has limitations. In particular, it is too expensive for large clusters. As an alternative, we describe a heuristic which is both fast and effective.

5.3.2 Cluster-to-Bucket Transformation

Finding $opt(\Re_C^-)$.

As a first step, we compute the convex hull of the cluster. The complexity of this is $O(n \cdot log(n))$, where n is the number of data points [CLRS09]. Given the convex hull of the cluster, we fit the largest axis-aligned rectangle into it. The algorithm in [Ame94] transforms this problem to a convex optimization problem. The complexity is $O(2^d \cdot h)$, where h where h is the number of vertices of the polygon, and d is the dimensionality of the data space. The complexity of the overall procedure is $O(n \cdot log(n) + 2^d \cdot h)$. This does not scale well against the dimensionality or the number of data objects.

Heuristic.

We propose an alternative heuristic which is computationally affordable. It starts with a rectangle that fulfills the condition $|RR \cap C| > |R - C|$ and expands it iteratively. Figure 5.2 shows a rectangle RR which is expanded along the y-axis downwards. The expanded rectangle RR' is dashed. RR' is a better bucket than RR if $\epsilon(RR') < \epsilon(RR)$, i.e., it approximates the cluster with less error than RR. Given

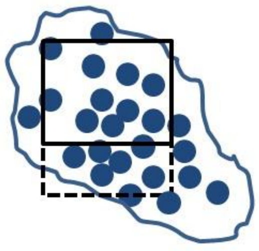


Figure 5.2: A cluster with rectangles RR (solid) and RR' (dashed).

Equation (5.2), this is equivalent to computing

$$\lambda(RR, RR') = |RR' \cap C| - |RR \cap C| + |RR - C| - |RR' - C| \tag{5.5}$$

and comparing it to 0. In order to compute λ, we have to compute $|R \cap C|$ and $|R - C|$ for a rectangle R. This is not straightforward because C has an arbitrary shape. We compute $R \cap C$ using the following idea: if we generate M data points uniformly distributed inside R, then the expected number of points m that will fall inside $R \cap C$ will be proportional to its area:

$$\frac{E[m]}{M} = \frac{|R \cap C|}{|R|}$$

From here we obtain

$$|R \cap C| = |R| \cdot \frac{E[m]}{M} \tag{5.6}$$

Now we can compute $\lambda(RR', RR)$, using Equation (5.6) and the fact that $|R - C| = |R| - |R \cap C|$.

Algorithm 8: Greedy algorithm for finding a RR

```
1:  $bestRR \leftarrow initial()$
2:  **repeat**
3:      $best \leftarrow \infty$
4:      $RR \leftarrow bestRR$
5:      **for all** possible expansions $e$ **do**
6:          $RR' \leftarrow expand(RR, e)$
7:          **if** $\lambda(RR, RR') > best$ **then**
8:              $best \leftarrow \lambda(RR, RR')$
9:              $bestRR \leftarrow RR'$
10:         **end if**
11:     **end for**
12: **until** $best = \infty$
```

In Line 1, we initialize $bestRR$ with some RR. In our implementation, we do this as follows: The center of the rectangle is the median. To compute the length of the projection of the rectangle on dimension i, we first project all points of the cluster to dimension i, let this be C_i. We define $diam(C_i) = max(C_i) - min(C_i)$. We took 1/10-th of $diam(C_i)$ as the length of dimension i.

5.4 Experiments

In the experiments we compare how different subspace clustering algorithms (Mineclus [YM03], PROCLUS [AWY$^+$99], CLIQUE [AGGR98], SCHISM[SZ04], INSCY[AKMS08] and DOC[PJAM02]) perform as histogram initializers. Implementations were used out of the *OpenSubspace* repository [MGAS09]. We first run each clustering algorithm against a data set, obtain the output, transform it into a bucket set and then measure the selectivity estimation error. We look at the following issues:

- *Precision.* We measure the estimation error of various clustering algorithms. We vary the number of tuples, the dimensionality of the dataset and the number of histogram buckets allowed. For the assessment of the quality of histograms, we use the Normalized Absolute Error metric ((4.13) in Section 4.7).
- *Memory consumption.* For a fixed estimation error threshold, we measure how much memory can be saved if we use initialized histograms vs. non-initialized STHoles.
- *Scalability.* The runtime cost of initialization depending on dimensionality.

5.4.1 Setup

For the experiments we used the *Gauss* dataset (Section 4.7.2.1 on page 44).

Multi-dimensional Gaussian bells are used both for selectivity estimation experiments [BCG01] and to assess the quality of clustering algorithms [HV01, LLX$^+$10]. We conducted two sets of experiments – for accuracy and for scalability. We generated 20 to 50 Gaussian bells with standard deviation = 50 in the data domain $[0, \ldots, 1000]^d$. The number of tuples t, the dimensionality d and the number of histogram buckets B are the main parameters to vary. This is because both clustering and selectivity estimation algorithms are sensitive to these parameters. In order to obtain clusters of different size, we generated tuple counts for each cluster according to a Zipfian distribution with $skew = 1$. Thus, we assign different numbers of tuples to each cluster. Table 5.1 gives an overview of parameter values for the accuracy and scalability experiments.

Experiment	Parameter	Value
Accuracy	d: dimensionality	2 to 10
	t: tuple count	10,000 to 50,000
	B: buckets	25 to 200
Scalability	d: dimensionality	10 to 20
	t: tuple count	500,000 to 1,000,000
	B: buckets	200

Table 5.1: Parameters values of experiments

We used 2,000 queries in the experiments. The query centers are uniformly distributed, and each query spans 1% of the volume of the data set. We used the first 1,000 queries for the algorithms to learn, this is the number from [BCG01] Thus, we start calculating the estimation error only after 1,000 queries have have run and all algorithms have learned. The error measure is the normalized absolute error, see Equation (4.13).

5.4.2 Estimation Precision

We first look at the selectivity estimation error for traditional STHoles compared to our initialized histograms. Figures 5.3 and 5.4 show the error for 2-dimensional space for 10,000 and 50,000 tuple-datasets respectively. Figures 5.5 and 5.6 show the 3-dimensional space, Figures 5.7 and 5.8 the 5-dimensional case, 5.11 and 5.12 the 10-dimensional case. Each figure plots the dependency of the normalized absolute error from the bucket number of the histogram.

Comparing the underlying clustering algorithms for our transformation, we observe one clear winner in all figures, namely the Mineclus algorithm. Mineclus clearly provides a high quality initialization particularly robust w.r.t. the dimensionality of the data space. While other clustering approaches show highly varying precision, Mineclus is the only algorithm with low estimation errors in all data sets.

More specifically, looking at the plots for $d \geq 4$ (Figures 5.7, 5.8, 5.11, 5.12), we can see Mineclus in the lower end of the graph, with other algorithms (STHoles included) in the upper part. PROCLUS is interesting: It is the best of this worse-performing pack for $d = 4$ and $d = 5, t = 10,000$. Starting with $d = 5, t = 50,000$ it becomes better and for $d = 10$ actually joins Mineclus . In all experiments Mineclus outperformed STHoles, while SCHISMand INSCYwhere consistently worse. Furthermore, a general observation from the figures is that adding more buckets increases the quality of histograms. However, the estimation quality highly depends on the type of initialization. Mineclus is the best method in all settings we have tried.

Looking at the methods which performed worse, we can see that STHoles is usually in the middle of the pack. This shows that:

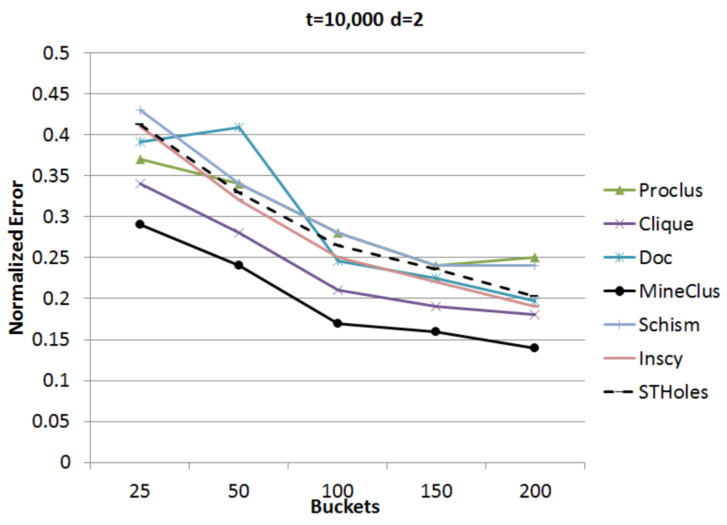


Figure 5.3: Error vs bucket count for 2-dimensional space, 10,000 queries

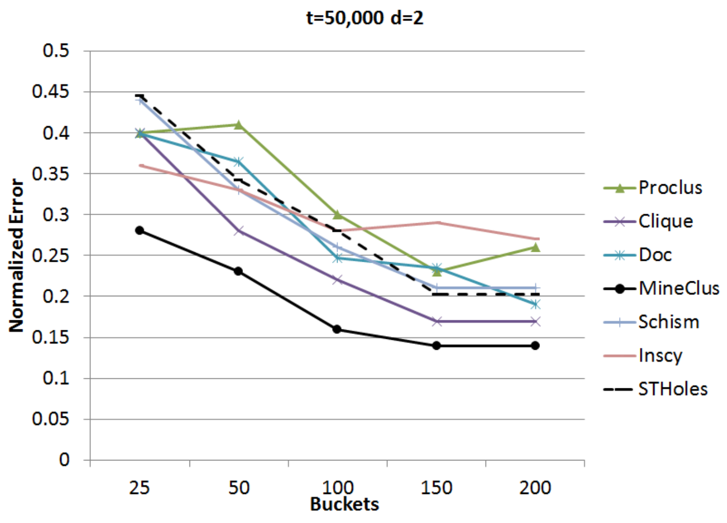


Figure 5.4: Error vs bucket count for 2-dimensional space, 50,000 queries

- Not every initialization is good or meaningful. After 1,000 queries for learning, STHoles performs about as good as most of the initialized methods. This confirms the statement in the original STHoles paper [BCG01].

- However, the underlying clustering methods make the difference between good and best. Mineclus is the winner. It provides a high quality initialization that yields better estimations than the original STHoles method.

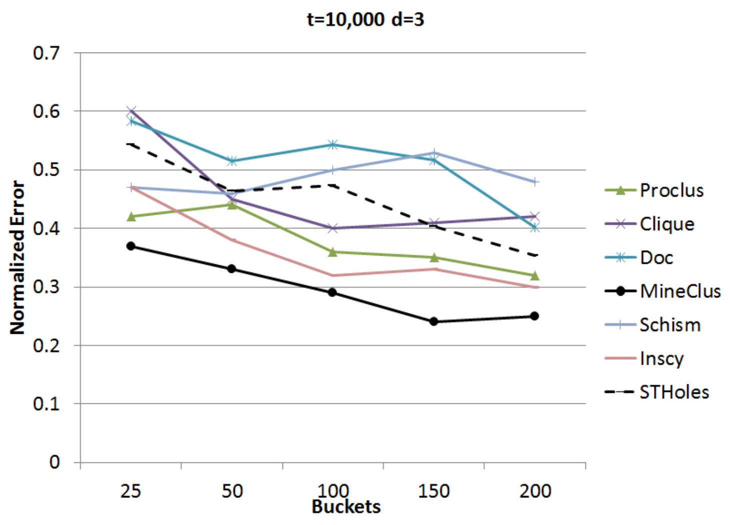


Figure 5.5: Error vs bucket count for 3-dimensional space, 10,000 queries

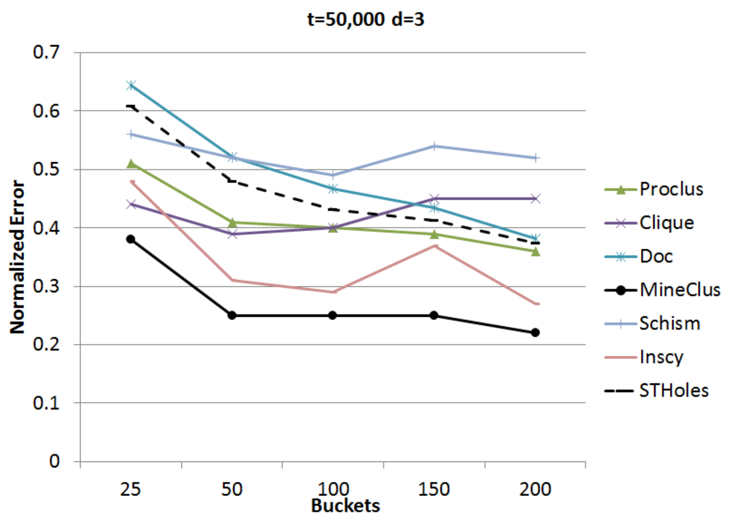


Figure 5.6: Error vs bucket count for 3-dimensional space, 50,000 queries

With increased dimensionality of the data space, Mineclus continues to perform better compared to STHoles, tackling the challenges of high-dimensional data better than the traditional selectivity estimation techniques. Further experiments with various high-dimensional data sets are required to find out how persistent this effect is throughout different application domains. Overall, we can see that initialization based on subspace clustering algorithms shows a clear benefit in terms of estimation precision.

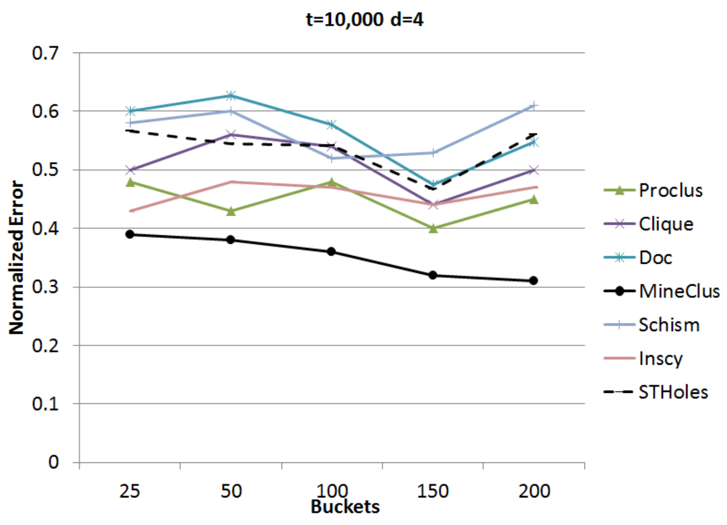


Figure 5.7: Error vs bucket count for 4-dimensional space, 10,000 queries

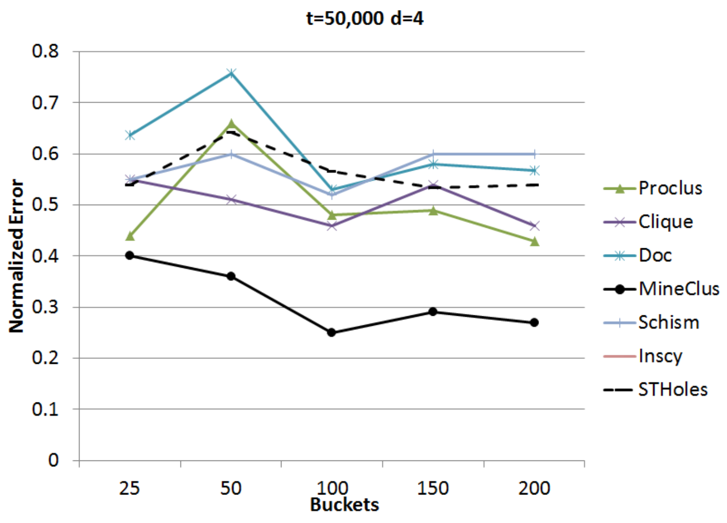


Figure 5.8: Error vs bucket count for 4-dimensional space, 50,000 queries

5.4.3 Memory-efficiency w.r.t. different initializations

Tables 5.2 and 5.3 shows a different perspective on the previous experiments. They highlight the memory-efficiency of our initialization compared to the relatively high memory consumption of STHoles. For each combination of parameter values, they shows how many buckets Mineclus needs to be at least as accurate as STHoles. For instance for Table 5.2 which shows the case when the dataset has 10,000 tuples. For

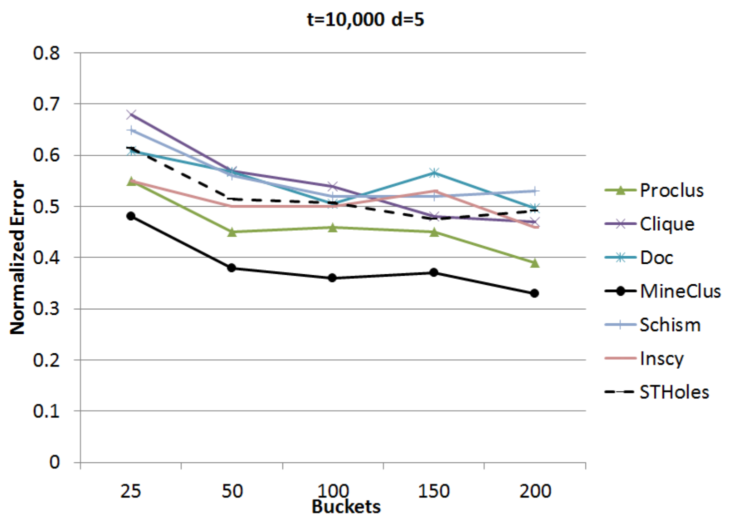


Figure 5.9: Error vs bucket count for 5-dimensional space, 10,000 queries

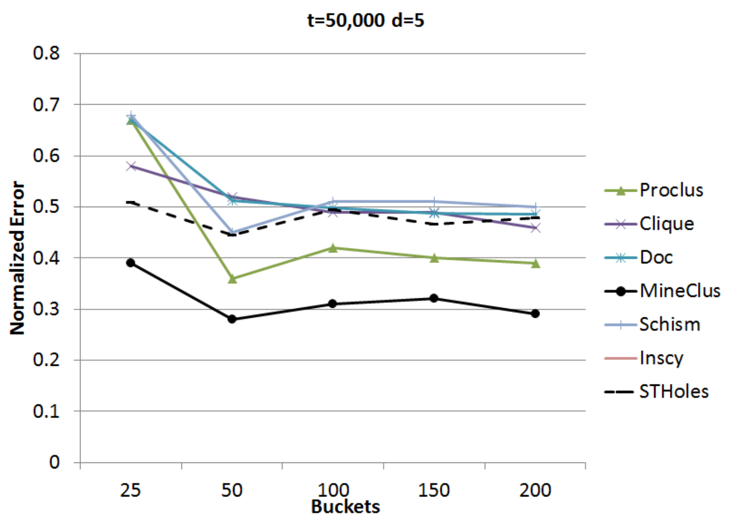


Figure 5.10: Error vs bucket count for 5-dimensional space, 50,000 queries

the 2-dimensional data set and 200 buckets allocated for STHoles, Mineclus needs only 100 buckets to produce estimates of equal or better quality.

We can obtain this from plot on Figure 5.3, by drawing a horizontal line at about 0.2, which is the error of STHoles for this setting and 200 buckets. This horizontal line intersects the Mineclus curve between 50 and 100 buckets. So 100 buckets is a conservative estimate of the buckets needed for Mineclus to match the precision of STHoles with 200 buckets. The tables shows that when the dimensionality of the

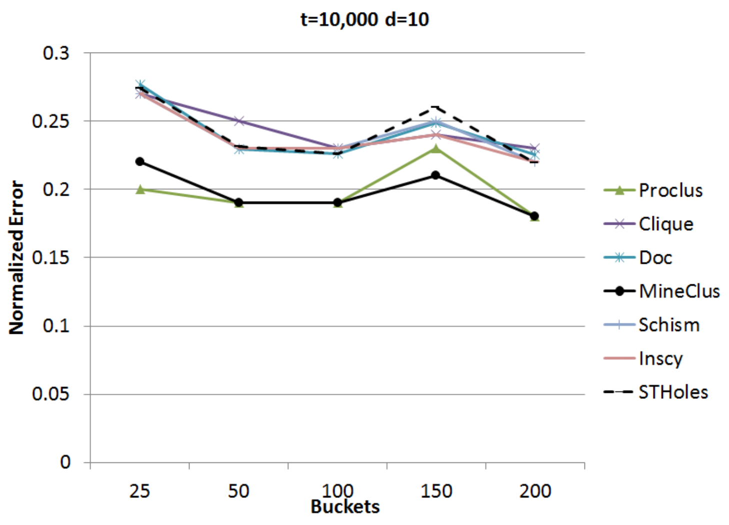


Figure 5.11: Error vs bucket count for 10-dimensional space, 10,000 queries

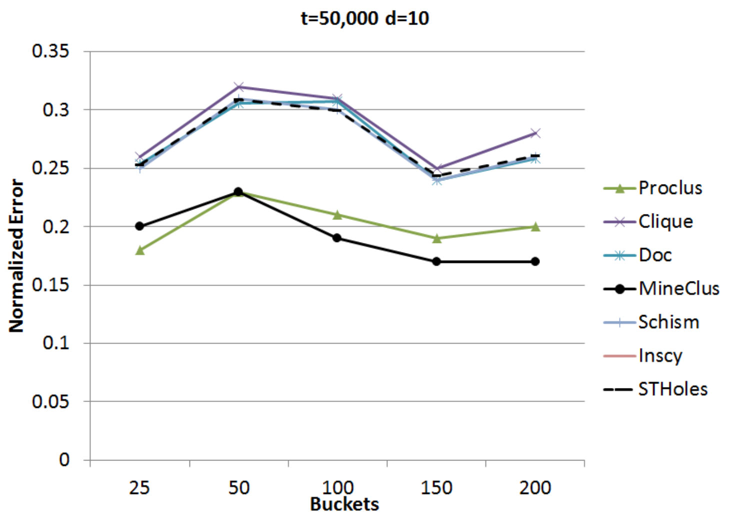


Figure 5.12: Error vs bucket count for 10-dimensional space, 50,000 queries

data set $d \geq 3$, 50 buckets for Mineclus are enough to match the precision of STHoles with 200 buckets. Even more surprisingly, out of 24 rows in both tables (rows corresponding to $d \geq 3$) only in two cases Mineclus needs 50 buckets, otherwise only 25 suffice to match the precision of STHoles with 100-200 buckets. This means that our initialization reduces the memory consumption by a factor of up to 8 in most cases.

Dim	STHoles Buckets	Mineclus Buckets
2	100	50
	150	50
	200	100
3	100	25
	150	25
	200	50
4	100	25
	150	25
	200	25
5	100	25
	150	25
	200	25
10	100	25
	150	25
	200	50

Table 5.2: Memory requirements for Mineclus compared to STHoles, to achieve same or better error rates as STHoles. Data set contains 10,000 tuples.

Dim	STHoles Buckets	Mineclus Buckets
2	100	25
	150	100
	200	100
3	100	25
	150	25
	200	25
4	100	25
	150	25
	200	25
5	100	25
	150	25
	200	25
10	100	25
	150	25
	200	25

Table 5.3: Memory requirements for Mineclus compared to STHoles, to achieve same or better error rates as STHoles. Data set contains 50,000 tuples.

5.4.4 Scalability w.r.t. data dimensionality

In general, the key parameter for subspace clustering is the dimensionality of the data space [MGAS09]. It affects the runtime performance of the clustering algorithm and thus is essential for our transformation as well: We fix the number of buckets to 200 and evaluate the influence of the dimensionality. Subspace clustering algorithms

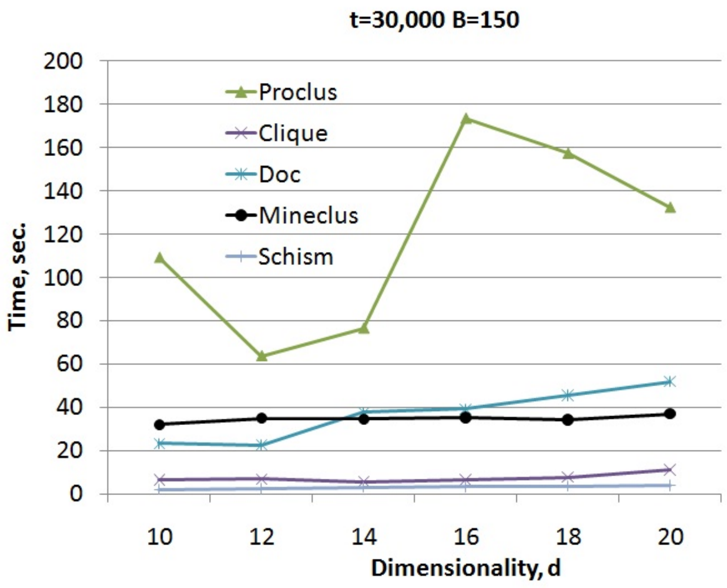


Figure 5.13: Execution time against the dimensionality

have been designed for efficient pruning in high-dimensional data. Thus, most of them show efficient and scalable runtime results (cf. Figure 5.13).

5.5 Conclusions and Future Work

In this chapter we studied initialization of self-tuning histograms using subspace clustering results. With our transformation of subspace clusters to memory-efficient histogram buckets, we could achieve significant improvement over traditional self-tuning selectivity estimators. In contrast to the traditional assumption that self-tuning can compensate the benefits of initialization, we show that our initialization is of clear benefit. Combining initialization with self-tuning results in a high quality histogram with low estimation errors.

6 Robust Self-Tuning Histograms

Abstract. *In the previous chapter, we introduced subspace-clustering as a preprocessing step to improve the accuracy of self-tuning approaches. We focused on the clustering part of the problem, namely how to transform the results of an arbitrary clustering algorithm into a histogram bucket-set. In this chapter, we draw the focus on the self-tuning histograms. Traditional self-tuning is sensitive to the order of queries, is unable to learn projections of high-dimensional data, and reaches only local optima with high estimation errors. We analyze these problems as well as the mechanisms which allow the initialized histogram to overcome them.* □

6.1 Introduction

Histograms are a fundamental data-summarization technique, and are used extensively in databases and related applications (see Chapter 4).

There are two different paradigms of histogram construction: *Static* and *Self-Tuning* histograms.

Static Histograms And Dimensionality Reduction. Static multi-dimensional histograms [PI97, GKTD00, BMB06, WS03, MPS99] are constructed by scanning the entire dataset. They need to be rebuilt regularly to reflect any changes in the dataset. For large relations, building a static multi-dimensional histogram in the full attribute space is expensive, both regarding construction time and the space occupied. *Dimensionality reduction techniques* try to solve the problem by removing less relevant attributes [GTK01, DGR01, Jol86]. These approaches leave aside that different combinations of attributes can be correlated in different subregions of the data set. Consider a database relation `Cars(Model, Manufacturer, Year, Color)`. The following correlations are possible: a) `Model` and `Manufacturer`, e.g., Golf implies Volkswagen. b) `Model` and `Year`, e.g., the Volkswagen Beetle was built until 2003. c) `Manufacturer` and `Color`, e.g., Ferraris are typically red. A static dimensionality-reduction technique would pick one of these correlations as the strongest one and build a histogram on that projection. All other correlations would be lost. However, in many applications we observed the phenomenon of locally relevant attributes. For example, the Sloan Digital Sky Survey (SDSS) dataset [SDS11] contains local correlations, i.e. specific regions of the sky showing high values for specific filters (cf. Section 6.5).

Traditional Self-Tuning Histograms. In contrast to static histograms, self-tuning histograms [BCG01, SHM$^+$06, LZZ$^+$07] use query feedback to learn the dataset. They amortize the construction costs, because the histogram is constructed on the fly as queries are executed. They are adaptive to the user querying patterns. As one representative, we consider the data structure of STHoles [BCG01], which is very flexible and has been used in several other histograms [SHM$^+$06, RKC$^+$10]. They try to learn bounding rectangles of uniformly distributed regions. However, similar to traditional index structures, such as R-Trees [Gut84], they fail in high dimensional data spaces due to the curse of dimensionality [BGRS99] and are affected by the order of tree construction [SRF87]. Similarly, self-tuning histograms are highly sensitive to the query order and fail on high dimensional data.

Recent methods have tried to address these issues. SASH [LWV03] is a higher-level framework which handles memory allocation, refinement and reorganization of histograms. It also decides which attributes to build the histogram on. Skipping of attributes is done for the whole data space, similar to the dimensionality reduction techniques mentioned above. Overall, the SASH is a framework above histograms, as it has to rely on some kind of histogram as underlying data structure (e.g. MHist from [PI97]). Another approach [LZZ$^+$07], re-schedules queries in order to achieve better histogram construction. The central assumption behind the approach is that it is permissible to delay the execution of queries or switch the query order.

Self-Tuning Histograms and Subspace Clustering. In contrast to all of these approaches, we do not make such assumptions and consider the common scenario where the queries are executed as they arrive. We focus on the general assumption with self-tuning methods, i.e. that they can learn the dataset from scratch – starting with no buckets and relying only on query feedback. We show that this is not the case and propose a novel method for initialization of histograms. In general, the first few queries define the top-level bucket structure of the histogram; if this structure is bad, then, regardless of the amount of further training, the histogram is unlikely to become good. In particular, we observe this for high-dimensional data, where self-tuning histograms have the same problem as the static histograms. They either pick some attributes statically and build the histogram on them, or ignore the issue by storing full-dimensional buckets. We show that in the later case the histogram is unable to learn important local correlations of data during future training. This, again, can be attributed to a bad initial top-level bucket structure.

To solve this top-level construction, we aim at initial structures that can be obtained by recent subspace clustering [KKZ09, PHL04, MGAS09]. In contrast to dimensionality reduction techniques, which aim at one projection, we aim at multiple local projections. In Chapter 5, we proposed a method to transfer arbitrary subspace clustering results into an initial histogram structure. We focused on the clustering aspect and evaluated the performance of several clustering algorithms. The take from there; one of our findings was that Mineclus performs best as an initializer. Thus, we take it as our basis for this chapter. However, Chapter 5 does not investigate the specific reasons *why* non-initialized self-tuning encounters problems, nor it investigates the mechanisms *how* subspace clustering improves the estimation accuracy. We investigate these reasons and mechanisms here, to highlight the hidden potential of self-tuning, which is not only revealed by our first solution, but is a general potential for future improvements.

Summing up, we show that the self-tuning has the following issues:

- **Sensitivity to learning.** This includes sensitivity to the type, shape, volume and order of the queries.
- **Stagnation.** Reaches only local optima due to missing initial configuration at the top levels of underlying data structures.
- **Dimensionality.** Inability to learn local correlations, which remain hidden in projections of high-dimensional databases.

We demonstrate that these problems exist and propose an initialization method with much lower error rates for large, high-dimensional datasets. We show formally that initial buckets can make self-tuning less sensitive to learning. In addition, we demonstrate that without initialization, self-tuning methods can struggle to find optimal bucket configurations and can stagnate with estimation high error. For the initialization we use subspace clustering in order to detect dense clusters in arbitrary projections of high-dimensional data [MGAS09]. In a nutshell, these subspace clusters provide essential information for top level buckets and their relevant dimensions. Thus, we address also the issue of finding relevant projections for self-tuning histogram in high-dimensional data spaces.

When conducting this work, we spent most of our effort to understand the *why* the three problems described above occur, and whether it is possible to address them all at once. For this reason, in this paper our focus is mainly the defining and understanding of the problems. From the technical point of view, our solution of using subspace clusters to initialize a self-tuning histogram is quite simple. However, we believe that this is an advantage of our approach.

6.2 Self-Tuning Histograms and Their Problems

We use STHoles [BCG01] as a representative for self-tuning histograms and describe its main properties and problems. We do this in four steps. First, we describe how the histogram partitions the data space and estimates query cardinalities. Then we describe how new buckets are inserted into the histogram. Third, we describe how the histogram compacts itself by removing buckets to free up space. Last, we derive the open challenges in this processing.

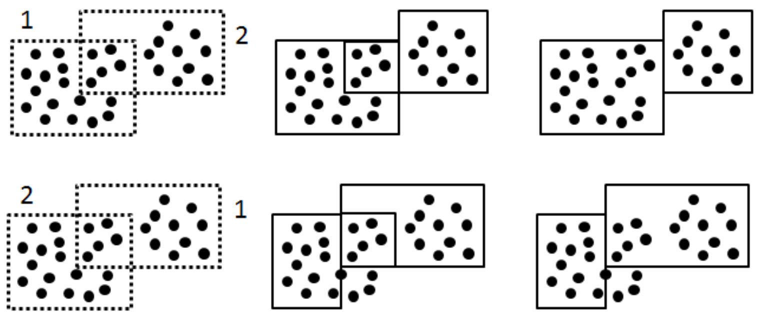


Figure 6.1: The queries and resulting histograms for two queries.

6.3 Problems with Self-Tuning

Let us now describe the problems associated with this self-tuning process. We focus on **Sensitivity to Learning** and *Stagnation*: we analyze how they occur and how Initialization helps to overcome them. We only briefly stop on **Dimensionality**, as this problem is relatively well known to the histogram construction and clustering communities.

6.3.1 Sensitivity to Learning

Informally, Sensitivity to Learning is when changing the order of the learning queries makes a significant impact on the histogram precision.

We will call the workloads W_1 and W_2 permutations of each other if they consist of the same queries, but in different order. We will write $W_2 = \pi(W_1)$, where π is some permutation. Given a histogram H and a workload W, we will write $H|W$ to indicate the histogram which results from H after it learns the query feedback from W.

At first sight, histograms resulting from two workloads where one is a permutation of the other one, $H|W$ and $H|\pi(W)$, should produce very close estimates. We first show on an example how permutation of queries can result in histograms which differ in structure considerably.

Example 6.1: Figure 6.1 demonstrates what happens when we change the order of the queries. The bucket limit for the histogram is two buckets. Each line shows a sequence: a) the left part of the figure shows the order in which the queries arrive (denoted by numbers), the middle one is after both queries are executed and buckets are drilled, and the right one is the final configuration after one bucket is removed to meet the 2-bucket budget. Clearly, the resulting histogram in the top-right is the better one. It captures the data distribution well, while the one in the bottom-right misses out some tuples and has one bucket with regions of different densities. Looking at the bucket-drilling procedure of STHoles, we can see why this happens. The histogram attempts to integrate the new information into the existing bucket structure, even if it means shrinking the new query rectangle. The rectangle corresponding to Query 1 in the bottom line is not good because contains regions with different densities. The second query (bottom line again) intersects with the first rectangle, and the resulting bucket is a shrunk version of the query. In other words, the second query which brings along useful information about the tuple distribution is deformed (bottom line, middle). ■

Let ε be some quality measure for the histogram. For instance, we can take

$$\varepsilon = \int_{u \in D} |real(u) - est(u)| du \tag{6.1}$$

where $real(u)$ is the estimated cardinalities of point-query u, and D is the attribute-value domain. For an error measure ε we define δ-sensitivity to learning.

Definition 6.1 (δ-sensitivity)
We call a histogram H δ-sensitive to learning w.r.t workload W if for some permutation π

$$|\varepsilon(H|W) - \varepsilon(H|\pi(W))| > \delta \tag{6.2}$$

□

δ-sensitivity means that changing the order of learning queries changes the histogram estimation quality by more than δ. Intuitively, if a workload W is informative (i.e. contains enough queries and does not miss out chunks of data), our expectation would be that a good histogram should not be very sensitive to workload permutations (i.e., the $delta$ should be small compared to $\varepsilon(H|W)$ and $\varepsilon(H|\pi(W))$).

6.3.2 Stagnation

Stagnation is the phenomenon of a histogram not being able to find a good bucket layout. There are several reasons why stagnation can occur. In this section we focus on one of the reasons, which is related to the "adaptiveness" of the histogram. We show that an adaptive histogram can get stuck in a locally optimal bucket layout, but is unable to improve it even with an unlimited number of learning queries. This problem of getting stuck in local optima is commonplace among learning algorithms.

First, we show that in the process of *detecting* a bucket configuration which guarantees a certain estimation precision, we typically need more memory than what is needed to *store* a configuration with the same guarantee. The simplest example is a large, uniform, rectangular cluster, which requires only one bucket to store. However, we cannot detect this cluster using small rectangular queries if our memory budget is only one bucket (we show this in detail later).

Stagnation occurs when the histogram does not have enough memory to detect certain clusters. It allocates the memory to other, unimportant data regions, due to the order of learning queries. As we will show, this can result in a situation when the histogram keeps the buckets in this unimportant regions and performs merges in important regions. Consequently, important clusters are not being detected because enough buckets are never created there.

We now turn to these issues in detail. First, we define the notions of *Cluster Detectability Threshold* and of *Storage Footprint*. These are, respectively, the memory needed to detect and to store the cluster so that the estimation error does not exceed a certain threshold. We compute these quantities for some simple data distributions. Next, we show by means of an example how a histogram can stagnate because of suboptimal allocation of memory buckets.

We give some auxiliary definitions.

Definition 6.2 (Histogram error on a cluster.)
The estimation error of the histogram H on cluster C is $\varepsilon_C(H)$:

$$\varepsilon_C(H) = \int_{u \in C} |est(u) - act(u)| \tag{6.3}$$

□

The difference between this and the error measure from Equation (6.1) is that instead of measuring the error over the whole data region D, we are confining it here to the cluster C.

Definition 6.3 (Bucket Count)
The number of buckets in histogram H is denoted by $b(H)$. □

We define the storage footprint for a region.

Definition 6.4 (Storage footprint.)
A cluster C has storage footprint $\sigma(C, \beta)$ for error threshold β if it is possible to construct a histogram H using $\sigma(C, \beta)$ buckets such that

$$\varepsilon_C(H) \leq \beta$$

□

The storage footprint is the minimal number of buckets which allows to capture the distribution with error $\leq \beta$.

In the following, we assume that the query workload W consists of uniformly distributed queries with unit volume. This is not crucial for the analysis, but it helps us to avoid cluttering the definitions.

Definition 6.5 (Cluster Detectability Threshold)
The detectability threshold of a cluster C for error threshold β, denoted by $\omega(C, \beta)$, is the minimal memory budget required to construct a histogram H such that

$$\varepsilon_C(H) \leq \beta$$

□

There is a caveat in the definition of Detectability Threshold which is that having $\omega(C, \beta)$ buckets only *makes possible* to detect C (in the sense that the error will be less than a given threshold). Having this much memory does not guarantee that a specific workload will detect the cluster, it only says that there is at least one workload out there which makes possible to detect the cluster.

The following result is easy to prove.

Lemma 6.3.1. *For any cluster C and error threshold β,*

$$\omega(C, \beta) \geq \sigma(C, \beta)$$

Proof. From the definition of storage footprint (Definition 4),

$$\sigma(C, \beta) = min\{b(H) | \varepsilon_C(H) \leq \beta\}$$

Now assume a histogram H detects the cluster C on some workload $W =< q_1, \ldots, q_n >$. Let the histogram obtained after executing the i-th learning query and performing merges be H_i.

$$\omega\{C, \beta\} = \max\{b(H_i) | 1 \leq i \leq n\}$$

From here we have that the amount of buckets in the final state, $b(H_n)$, is less or equal $\omega(H, \beta)$. But $H = H_n$ also is a histogram which produces $\leq \beta$ error on C, thus:

$$\omega(C, \beta) \geq b(H) \geq \sigma(C, \beta)$$

□

What makes this lemma more valuable are the examples that come next, where we show that in fact for several simple and common data distributions, $\omega(C, \beta) > \sigma(C, \beta)$ for certain (relevant) values of beta. That is, detecting a cluster is never cheaper memory-wise than storing it, and very often it is more expensive.

Before going on with the analysis we introduce a convention about resolving ties regarding possible merges which have the same penalty. If two merge candidates have the same penalty and one of them is a parent-child merge and the other one is a sibling-sibling merge, we perform the parent-child merge. We call this the *Parent-First Merge Convention*. Without this assumption the picture does not change: Storing a cluster typically requires less memory than detecting the cluster. However, the reasoning and the examples would become more complicated and we would have to consider many special cases.

Now we calculate $\sigma(C, 0)$ and $\omega(C, 0)$ for a uniform cluster.

Example 6.2: Assume that the clusters and the queries are aligned to a grid. The queries have unit volume. Let the dataset be $[1, \ldots, N] \times [1, \ldots, N]$, then the query rectangles have the form $[i, i+1] \times [j, j+1]$ where $i, j \in \{1, \ldots, N-1\}$. We will consider a square cluster with area s^2, where $s \geq 2$ (Figure 6.2 shows a cluster with $s = 5$).

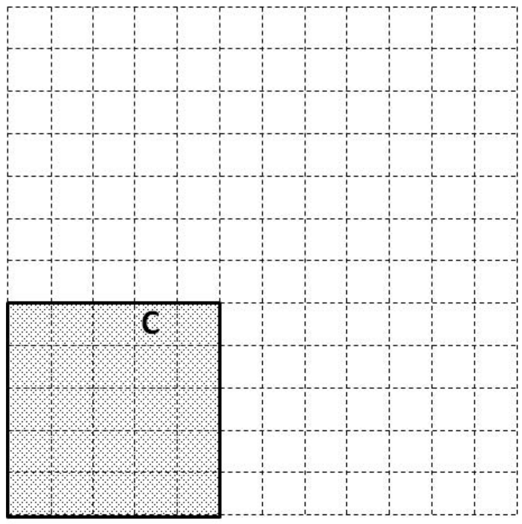


Figure 6.2: The data space and the cluster C

It is clear that $\sigma(C, 0) = 1$. We now show that $\omega(C, 0) = 2$. This means that

1. It is possible to detect the cluster using two buckets.

2. It is impossible to detect the cluster using only one bucket.

Detecting the cluster with 2 bucket budget. The cluster C has size 5×5, while the training queries are unit queries (1×1). In order to "assemble" the cluster using such queries, the histogram needs to repeatedly merge unit-volume queries into buckets and obtain larger buckets.

When the histogram memory budget is 2 buckets, the merge occurs when there are three buckets in the histogram. Figure 6.3 shows a possible histogram with three buckets. Note that in the figure the cluster boundary is not known, all the histogram "knows" about the dataset are the buckets 1, 2 and 3. In this situation buckets 1 and 2

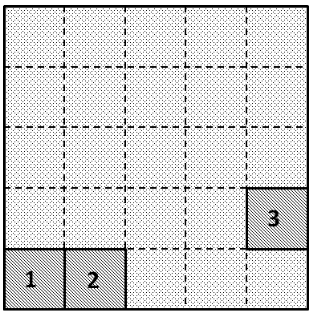


Figure 6.3: The cluster.

will be merged. It is easy to see that certain sequences of learning queries will result in finding the first "row" of the cluster. Then, it is possible to discover the second row, merge it with the first row and so on (Figure 6.4), until the whole cluster is detected.

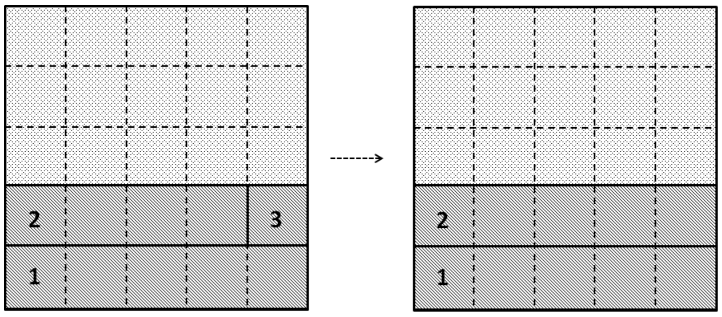


Figure 6.4: The cluster with one row detected (left) and the second row detected (right).

Now we will demonstrate that it is impossible to detect the cluster using only one bucket.

Note that under a one-bucket budget and the Parent-First Merge Convention, only adjacent cells can merge. The result of such merges will be a bucket which is similar to the one depicted on Figure 6.5, that is, it can be a single "row" or a "column" inside the cluster. Now, when a new bucket is added to the histogram, merging it back with the root bucket has less penalty than merging it with the root. Thus, after the histogram detects either a row or a column, it will stop expanding.

We have shown that it is possible to detect C using 2 buckets but it is impossible to detect it using 1 bucket. Therefore, $\omega(C, 0) = 2$.

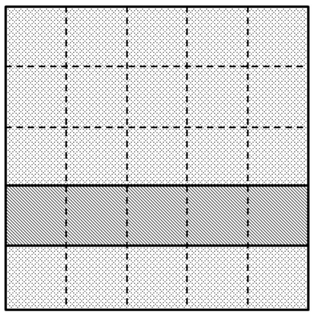


Figure 6.5: A bucket which is a result of several merges which occurred horizontally on the second row of the cluster.

■

We now define histogram stagnation formally and give several examples.

Definition 6.6 (Stagnation.)
We say that a histogram stagnates at error level β with reducible error Δ if the histogram error is $\varepsilon(H) = \beta$, the error of H does not change by more than $\epsilon << \Delta$ by subsequent learning, and there exists a histogram H' with $b(H)$ buckets such that $\varepsilon(H') = \beta - \Delta$. □

When the histogram stagnates but the reducible error is low, there is not much we can do. So when talking about stagnation, we mean the cases when Δ is comparable to β, say $\Delta \geq 0.3 \cdot \beta$.

Conjecture 6.1 (Stagnation.)
Without initialization, histograms are likely to stagnate with reducible error which is comparable to the error rate.

We have seen in Example 2 that the histogram cannot detect the cluster with one bucket. Given a one-bucket budget the uninitialized histogram will construct a bucket which will not cover the cluster, and the error level will be β. The initialized histogram has error equal to 0. So the uninitialized histogram stagnates at some error level β with reducible error $\Delta = \beta$. The next example demonstrates a more complex scenario involving two non-uniform clusters.

Example 6.3: In this example the dataset contains two square clusters, which are identical (Figure 6.6). The clusters have a core, a 1×1 rectangle which is 5 times more dense than the rest of the cluster, where the density is 1 tuple per unit volume.

Now assume the histogram budget is 2 buckets.

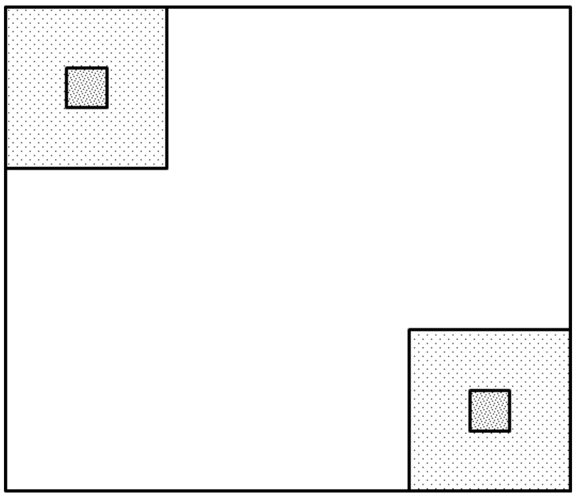


Figure 6.6: A histogram with two clusters, each of the clusters has a dense core.

It is easy to see that the optimal bucket layout assigns 1 bucket to each cluster, without separating the core of the cluster. Such a layout is impossible to construct using a 2 bucket budget and relying only on training. Assume one of the clusters is found and one bucket is assigned to it. Then, using one bucket budget, it is impossible to detect the second cluster. The argument here is very similar to the one we used in Example 2 where we showed that using one bucket only a row or column of the cluster are detectable.

In fact it is possible to show that a two-bucket budget is not enough to detect one of the clusters. We will end up partially detecting both clusters. We do not go through the details here, but for the training sequence that gives the best bucket layout, $\Delta \geq \beta/2$. ■

Examples 2 and 3 have something in common: They show cases where it is impossible to detect a globally optimal configuration. The difficulty in both cases to find the boundaries of the cluster using a limited number of small buckets.

6.3.3 Dimensionality

Histogram construction in high-dimensional spaces is a challenging problem [BCG01]. In particular, finding locally correlated groups of tuples becomes harder as we increase the dimensionality of the space [PI97, BCG01]. One reason is that in high-dimensional spaces for a given region not all attributes are relevant, i.e., some attributes are just random noise in certain regions, while they are relevant in others. In general, this observation is named the curse of dimensionality [BGRS99], and has been addressed by dimensionality reduction techniques. However, these techniques miss multiple local correlations. Recently, subspace clustering has tackled this problem [KKZ09, KKZ09, MGAS09].

Finding lower-dimensional buckets is similar to finding subspace clusters. Subspace clustering methods access and analyze the entire data set. In contrast, a self-tuning histogram looks only at the query feedback. Thus, self-tuning histograms have the disadvantage of having to deal with incomplete information. – Given this discussion, we suppose that the following holds, and we will validate it as part of our experiments.

Conjecture 6.2 (Dimensionality)
Subspace buckets, which represent local correlations of tuples in lower-dimensional projections, are hard to find using only full-space query-execution results.

6.4 Subspace Clustering and Histogram Initialization

In this section, we describe our solution of initializing the histogram with subspace buckets. Then, we show formally how this allows the histogram to become less sensitive to learning. In order to find a solution to the problems described above (Sensitivity to Learning, Stagnation, Dimensionality), we first discuss the shared *reasons* for those problems and a common *mechanism* in subspace initialization that solves these problems.

6.4.1 Initialization by Subspace Clusters

Self-tuning is able to *refine* the structure of the histogram. If started with no buckets at all, the histogram has to rely on the first few queries to determine the top-level partitioning of the attribute-value space. If this partitioning is bad, the later tuning is unlikely to "correct" it. The solution is to provide the histogram with a good initial configuration. This configuration should:

1. provide a top-level bucketing for the dataset, which can be later tuned using feedback;

2. capture the data regions in relevant dimensions, i.e. should exclude irrelevant attributes for each bucket.

We now describe how to initialize the histogram with subspace buckets. The subspace clustering algorithm finds dense clusters together with the set of relevant attributes. Then these clusters are transformed into histogram buckets.

Generally, clustering algorithms output clusters as a set of points. We need to transform this set of points into a rectangular representation. Cell-based clustering algorithms such as Mineclus [YM03] look for rectangular clusters. We could take these rectangles as the STHoles buckets. However, we have found out in preliminary experiments that this has a drawback, which is illustrated in Figure 6.7. Although the cluster found is one-dimensional (left), the MBR is two-dimensional (dashed rectangle on the right). The two-dimensional MBR would introduce additional intersections with incoming query rectangles without measurable difference in estimation quality. This is undesirable. We can bypass this problem using the information produced by

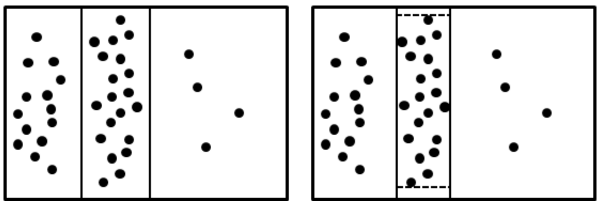


Figure 6.7: On the left, the cluster found. On the right, the dashed rectangle is the MBR of the cluster. The solid rectangle on the right is the extended BR.

Mineclus. Mineclus outputs clusters as sets of tuples together with the relevant dimensions. This means that the cluster spans $[min, max]$ on any unused dimension. To preserve subspace information, we introduce *extended BRs*.

Definition 6.7 (Extended BR.)
Let cluster C consist of tuples $\{t_1, \ldots, t_n\}$ and dimensions $d_1, \ldots, d_k$. The extended BR of C is the minimal rectangle that contains the points $\{t_1, \ldots, t_n\}$ and spans $[min, max]$ for every dimension not in $d_1, \ldots, d_k$. $\square$

Definition 6.8 (Initialization by Subspace Clusters)
If the dataset consists of disjoint clusters $C_1, \ldots C_m$, then the initialized histogram is a histogram with buckets $\{b_1, \ldots, b_m\}$, where the bounding box of b_i is the extended BR of C_i, and the number of tuples in b_i is the tuple count of the cluster C_i. $\square$

A useful characteristic of Mineclus is that is assigns importance to clusters. The algorithm has a score function which decides whether a set of points is a cluster or not. The clusters themselves are then sorted according to this score. We found out that, if we use the important clusters as first queries in the initialization, we have a better estimation quality.

6.4.2 Analysis on Subspace Solution

Here we analyze how initialization by subspace clusters helps the histogram obtain a better structure and have a lower error.

6.4.2.1 Initialization and Sensitivity to Learning

We first conduct our analysis on a simplified scenario, then explain how more general cases can be reduced to this simple case.

Assume the attribute-value domain contains only one rectangular cluster, C, which has close-to-uniform density. The tuple density outside the cluster considerably less than the cluster density. What matters in all the computations is the *density difference* between two regions, so without loss of generality we can set the outside density equal to 0.

Denote the attribute-value domain as D, and assume that $vol(D) >> vol(C)$.

First, we initialize the histogram with a bucket b_0 which has a bounding box that equals to the bounding box of C. Call this histogram H_0. The error of the histogram H_0, $\varepsilon(H_0) = 0$ (ε is defined as Equation (6.1)). H_0 is depicted in Figure 6.8.

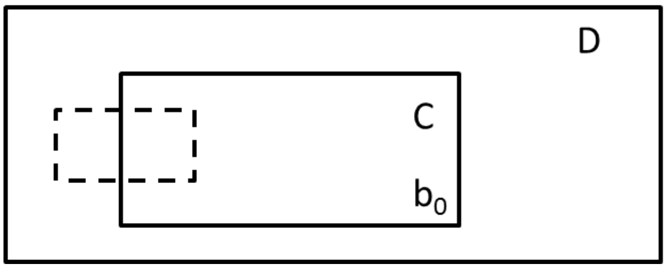


Figure 6.8: The histogram H_0, with cluster C as a bucket. The dashed rectangle is the incoming query q.

Lemma 6.4.1. *For any workload W and maximum bucket limit $m > 1$, the histogram $H_0|W$ will have no error: $\varepsilon(H_0|W) = 0$.*

Proof. Assume the contrary, that the histogram $H_0|W$ has a positive error. Then, it should have a bucket b which has positive error. b has to partially intersect with C and $D \setminus C$, otherwise it will have a constant density and will not contribute to the error. If the bucket b_0 exists in the histogram, the bucket b can never be created. If a query q arrives which partially intersects with b_0 (Figure 6.8), the intersections will be divided into two classes: $b_0 \cap q \subset C$ and $(D \setminus C) \cap q$. The intersections $b_0 \cap q$ will be drilled as children of b_0, the other buckets will be drilled into a subtree with some other root other than b_0. The bottom line is as long as b_0 exists in the histogram, it will "chop" incoming queries and make sure that there are no buckets which contain area both from C and $D \setminus C$.

Thus, we have concluded that in order for the bucket b to exist, the histogram has to eliminate the bucket b_0. The mechanism of elimination of b_0 is merging it with a sibling or a parent. We show that this cannot happen, and regardless of the number of maximal buckets allowed there always is a better merge. Assume that at some point of the time the histogram contains $m + k$ ($k \geq q$) buckets and has to eliminate k buckets through merges. There are m_1 buckets inside $D \setminus C$ and m_2 buckets inside b_0, such that $m_1 + m_2 = m + k - 2$ (the number of histogram buckets at the moment, minus the root and b_0). At least one of m_1, m_2 is ≥ 1. If $m_1 \geq 1$ then it will be merged with its parent bucket with 0 merge penalty. The reason is that the density in $D \setminus C$ is constant. The same argument applies to the case when $m_2 \geq 1$. The bucket b_0 always has a positive merge penalty with a sibling or its parent. Thus, we have shown that the bucket b_0 cannot disappear from the histogram because there are always merges with a lower penalty. This means that there cannot exist a bucket b which contains area both from C and $D \setminus C$, which is a contradiction. $\square$

This lemma shows that once the bucket b_0 has been drilled, any sequence of queries cannot "spoil" the histogram structure. The bucket b_0 itself is *stable*, i.e. it does not disappear because there will always be better merge candidates (with less merge penalty). Due to b_0, the histogram H_0 is insensitive to learning.

On the other hand, if we start with no buckets at all, the histogram bucket structure will depend on the query order (see Example 1). It is clear that there are numerous query orders which would result in histograms that have a bucket b_1 which only partially intersects with b_0. All such histograms will have error $\varepsilon > 0$.

We showed on a simple dataset with one dense cluster that capturing the cluster in a bucket makes the histogram insensitive to learning, for an arbitrary query workload and arbitrary number of maximum buckets allowed. In the absence of such a bucket, the histogram is sensitive to learning.

We can generalize the lemma and show the same for a number of disjoint clusters $C_1, \ldots, C_l$. Next, the in-cluster density does not have to constant, all we need is that the density drop from the cluster to outside the cluster is large enough to "discourage" the histogram from performing merges which include a bucket from within the cluster and one from outside.

In practice, of course, the data can be very complex. Having demonstrated here why initialization makes the histogram less sensitive to workloads, we now turn to the experimental evaluation.

6.4.2.2 Initialization and Stagnation

We showed in Section 6.3.2 that detecting a cluster is never cheaper compared to storing it (Lemma (6.3.1)), and Examples 2 and 3 demonstrate that even for very simple data distributions, the the detectability threshold can be higher than the storage threshold.

Notice that, when discussing the storage threshold in those examples, we took exactly the extended BR of the clusters as a bucket. Those examples in fact show how initialization helps to avoid stagnation by starting the histogram with hard-to-find buckets, which are in fact the cluster boundaries.

A general analysis for arbitrary clusters and data distributions is impossible. However, very often we are dealing with clusters which have smaller sub-clusters with significantly different density.

Now we informally discuss how initialization improves detectability for data distributions which are noticeably more complex than the ones we discussed in Section 6.3.2. Assume we have a rectangular cluster which has several dense subregions which we want to capture. Figure 6.9 shows a cluster with several dense subregions.

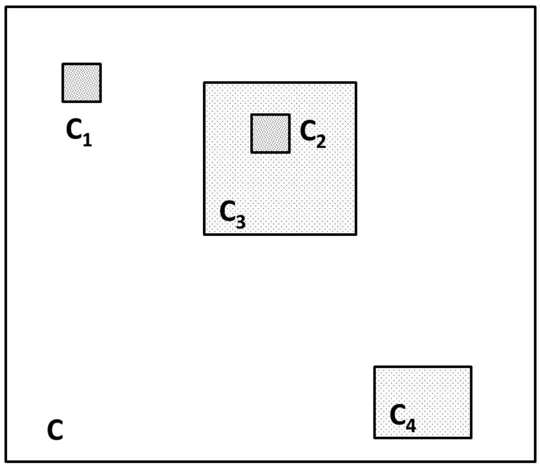


Figure 6.9: Cluster C with several dense subregions

We argue why initialization is helpful in this case. We will discuss two separate scenarios.

1. The density of C is relatively low, and we do not need it as a bucket in the histogram
2. The density of C is relatively high and we need to have it as a bucket in order to have low error.

1. The density of C is low. In this case it is more important to use a bucket for one of the sub-clusters. The Mineclus algorithm finds dense clusters. The minimum density threshold is controlled by a parameter, and the clustering algorithm will discard clusters which are below a certain density threshold. Thus, initialization will not be detecting the "useless", low-density buckets.

2. The density of C is high. In this case we want to have C as a bucket in the histogram. We may or may not want to have all $C_1, \dots C_4$ as separate buckets, depending on the density distribution elsewhere and the available memory budget. We have seen in Examples 2 and 3 that detecting the larger bounding box of the cluster is hard when the available memory budget is low. The histogram needs to have a significant coverage of the region of C in order to perform sibling-sibling merges and enlarge the boundaries of the dense region found. If the coverage of the area of C is small, the histogram merges one of the smaller buckets with the root because this has smaller merge penalty. Having several subregions with different densities fracture the data space, and even more buckets are needed to capture such a fractured distribution.

Initialization finds the boundaries of C. Further learning deals with the smaller clusters. As discussed here and seen in Examples 2 and 3, finding the boundaries of C is hard, and if the clustering algorithm finds it then it is likely to be useful in the histogram. Initialization needs only one bucket for the cluster boundaries, which is memory-efficient. Self-tuning in contrast needs a larger memory budget to be able to detect C. If this budget is not available, the histogram stagnates.

6.5 Experiments

So far, we have described the problems with self-tuning approaches and our solution based on initialization using subspace clustering. In the following, we will empirically show that initial subspace clusters make self-tuning more accurate and robust. First, we show that our solution based on initialization provides a clear accuracy improvement over the uninitialized version (Section 6.5.2). Then, we focus on the challenges from Section 6.3, namely – Sensitivity to Learning, Dimensionality, and Stagnation.

6.5.1 Experimental Setup

We have used two synthetic and one real-world datasets. Dataset parameters are summarized in Table 6.1. Dataset descriptions can be found on Section 4.7.2 on page 44.

Dataset	Type	Dimensionality	Tuples
Cross	Synthetic	2	22,000
Gauss	Synthetic	6	110,000
Sky	Real-World	7	$\approx$ 1.7 million

Table 6.1: Dimensionalities and tuple counts of our datasets

Queries, Buckets, Metrics.

We focus here on $V[\%]$ queries, as explained in Section 4.7.3 on page 47. These are queries which span a certain volume in the data space. We also have conducted experiments with different workload-generation patterns, and the trends have been the same. Hence, we stick to the pattern "random centers, fixed-volume queries" because this allows to compare results across experiments.

The precision of a histogram usually depends on the available space. We vary the number of histogram buckets from 50 to 250 like most other authors do [SHM+06, RKC+10, BCG01, WS08].

The quality of estimations is measured by the error the histogram produces over a series of queries. The error metric is described in Section 4.7 on page 42. Unless stated otherwise, the workload is the same for all histograms and contains 1,000 training and 1,000 simulation queries. The first 1,000 queries are only for training, and the error computation starts with the simulation queries.

6.5.2 Accuracy

In the first set of experiments we show that initialization improves estimation quality. Figures 6.10, 6.11 and 6.12 show the error comparison for the *Cross*, *Gauss* and *Sky* datasets. For all datasets, the initialized histogram outperforms the uninitialized version. As mentioned, the *Cross* dataset is simple and can easily be described with

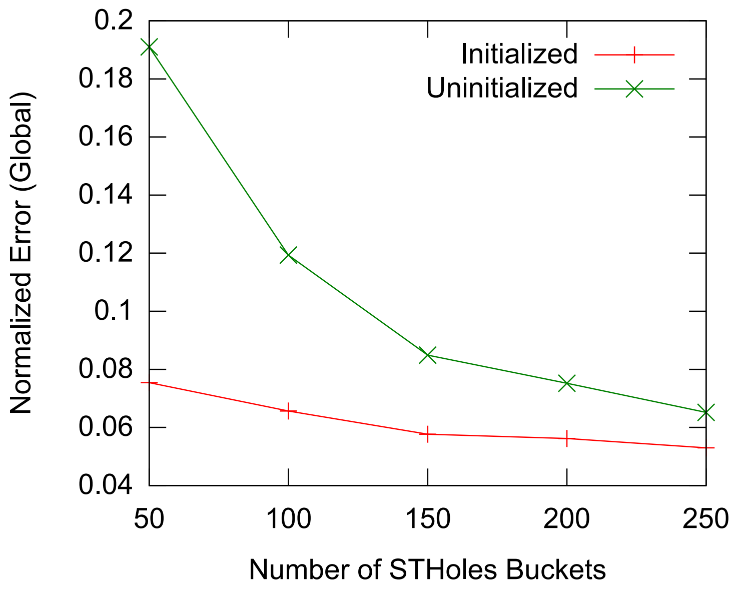


Figure 6.10: Error comparison for $Cross[1\%]$ setting

5 to 6 buckets. Nevertheless, Figure 6.10 shows that initialization has a significant effect in improving the estimation accuracy. This is an experimental confirmation of the analysis conducted in Section 6.4.2. Initialization finds the 5-6 buckets which are essential for the good histogram structure, while a random workload of even 1,000 training queries is not enough for the uninitialized histogram to find this simple bucket layout. Figure 6.11 shows the error on the *Gauss* dataset, which contains more complex structures in the database in the form of Gaussian bells hidden in different

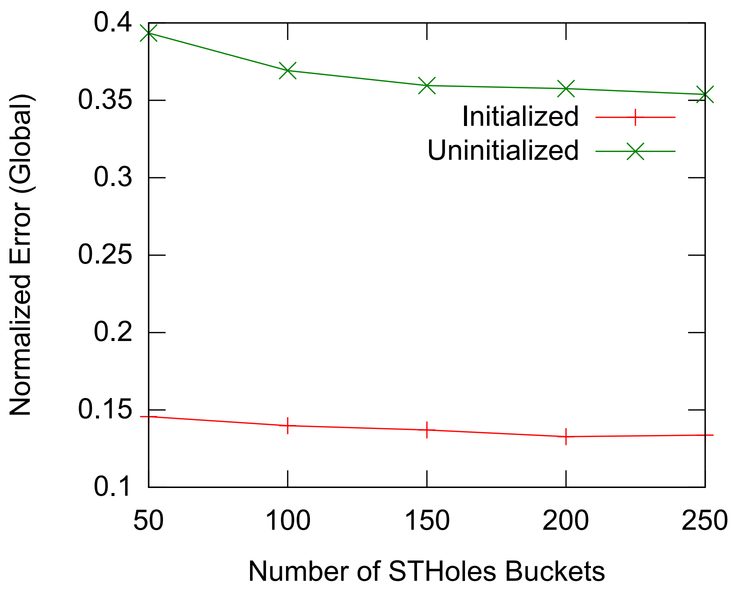


Figure 6.11: Error comparison for $Gauss[1\%]$ setting

projections of the data space. Comparing with the *Cross* dataset, we can see that the estimation error for both uninitialized and initialized is higher. This is expected and is due to the fact that *Cross* is a piecewise-uniform dataset and *Gauss* is not. On the *Gauss* dataset we can see the effect of the subspace clustering much better, as the initialization now provides a considerably bigger benefit compared to the *Cross* dataset. Figure 6.12 shows the comparison on the *Sky* dataset. Here, the errors are higher than both for *Cross* and *Gauss* datasets. The benefit of subspace clustering is again clear: The initialized version has about half the error rate compared to the uninitialized version.

In all cases, the initialized histogram outperforms the uninitialized version. Moreover, for the *Gauss* and *Sky* datasets, the initialized histogram with only 50 buckets is significantly better than the uninitialized histogram with 250 buckets. Only on the simple *Cross* dataset the uninitialized histogram with 250 buckets reaches the quality of the initialized histogram with 50 buckets.

6.5.3 Robustness

We now revisit the challenges mentioned in Section 6.3. The following experiments highlight the reasons why our initialized version outperforms traditional uninitialized histograms. Essentially we investigate the sensitivity to learning and the effects of dimensionality of the data space on self-tuning.

Sensitivity to learning. To show that STHoles is sensitive to learning, we con-

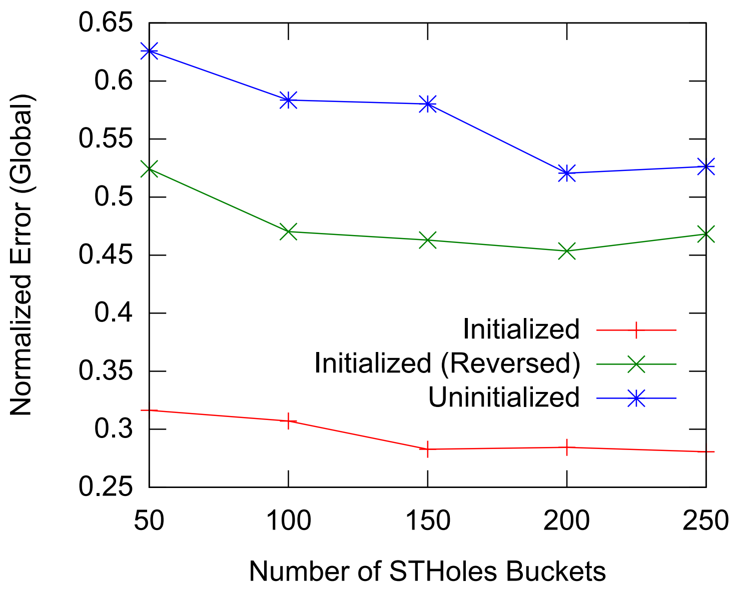


Figure 6.12: Error comparison for $Sky[1\%]$ setting. The meaning of the green line ”Initialized (Reversed)” is explained in Section 6.5.3.

ducted experiments using permuted workloads as defined in Section 6.3. To show the effect of changing the order of queries, recall how we initialize the histogram. We generate rectangles with frequencies from the clustering output and feed this to the histogram in the order of importance. This importance is an additional output of the clustering algorithm. In the experiment in Figure 6.12, we use the same set of clusters to initialize the histogram, but in a reverse order of importance. Clearly, there is a significant difference between the normal initialization and the reverse one. This shows two things. First, it is clear that permuting a workload changes the histogram error significantly (Sensitivity to Learning). Second, it shows the the importance of the order of initialization, as the "correct order" has a noticeably lower error compared to the reversed order. Finally, Figure 6.13 shows the *Sky* dataset, but with 2% volume queries instead of 1%. By comparing the results to the ones in Figure 6.12, we can see the effect of changed query volumes. Except for the case with 50 buckets, the error of the initialized version is essentially the same in both figures. Thus, the initialized version is considerably less sensitive to the change of query volume compared to the uninitialized histogram.

Dimensionality. Running Mineclus on the *Sky* dataset, we have found 20 clusters, referred to as $\{C_1, \ldots, C_{20}\}$ subsequently. Out of those, 11 were full-dimensional and 9 were subspace clusters. Table 6.2 sums up the the information of the clusters in the 7-dimensional *Sky* dataset. We list the irrelevant dimensions that the clusters **do not** use for bucket representations. Clearly, there are global structures detected in the full-space, but also very specific correlations between some of the dimensions w.r.t. a

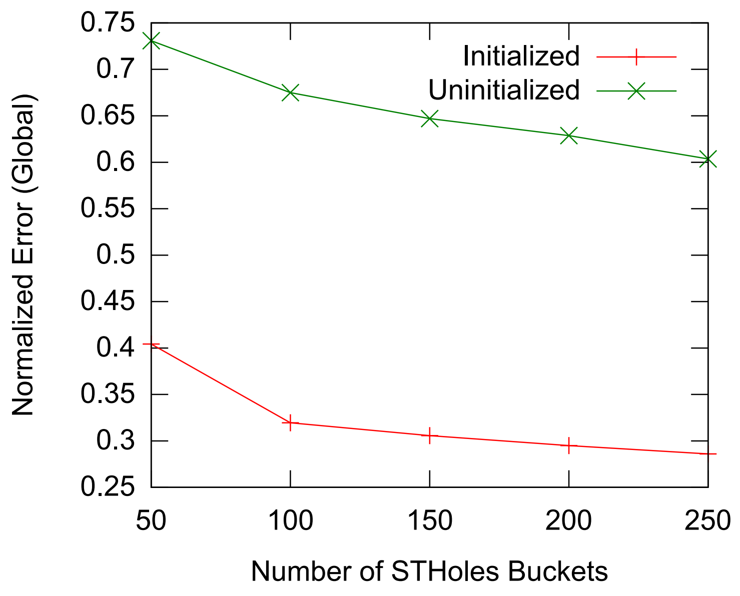


Figure 6.13: Error comparison for $Sky[2\%]$ setting

subset of tuples have been detected in the database. We have conducted experiments

Cluster	Dimensions not used	Cluster	Dimensions not used
$C_1, \ldots, C_{11}$	none	C_{18}	1, 2, 7
C_{12}, C_{13}	1	C_{19}	1, 2, 3, 7
C_{14}, C_{15}, C_{16}	$1, 2$	C_{20}	1, 2, 3, 5, 6
C_{17}	1		

Table 6.2: Clusters found in the *Sky* dataset and the dimensions they do not use

with varying bucket counts, from 50 to 250, as follows. After every 100 queries (out of 2,000 total), we dump the histogram structure and look for subspace buckets. For all bucket counts, the uninitialized histogram has not created a single subspace bucket. The initialized version starts with several subspace buckets, which eventually are merged as the simulation goes on. The only case when subspace buckets are preserved through 2,000 queries is the initialized histogram with 250 buckets. We find this quite interesting as the number of merges during 2,000 query-simulation with 250 buckets is very high, and 4 subspace clusters "survive" that many merges. With the initialized histogram , we observe that the higher the number of buckets, the longer the subspace buckets survive.

Our conclusion from these dimensionality experiments is the following:
STHoles is unable to find subspace buckets on its own. To do so, it needs our initial-

ization. Thus, Conjecture 2 is true – subspace clusters are hard to find using full-space query feedback.

Stagnation. In the figures above, use the same training workload both for initialized and uninitialized histograms. Looking at the *Sky* dataset in Figures 6.12 and 6.13, we can see that the uninitialized histogram has twice the error rate of the initialized version. One wonders whether additional training can help to overcome this difference. By extra training of the uninitialized version, we can find out whether the effects of initialization are temporary or persistent. The setup for this experiment is the following:

1. We start by training both initialized and uninitialized histograms with the same 1,000 queries.
2. We continue the training of the uninitialized version with an additional 18,000 queries.
3. We evaluate both histograms using the same 1,000 query workload.

Figure 6.14 plots the errors of the initialized and heavily-trained as well as of the uninitialized histograms. The initialized version consistently outperforms the heavily-

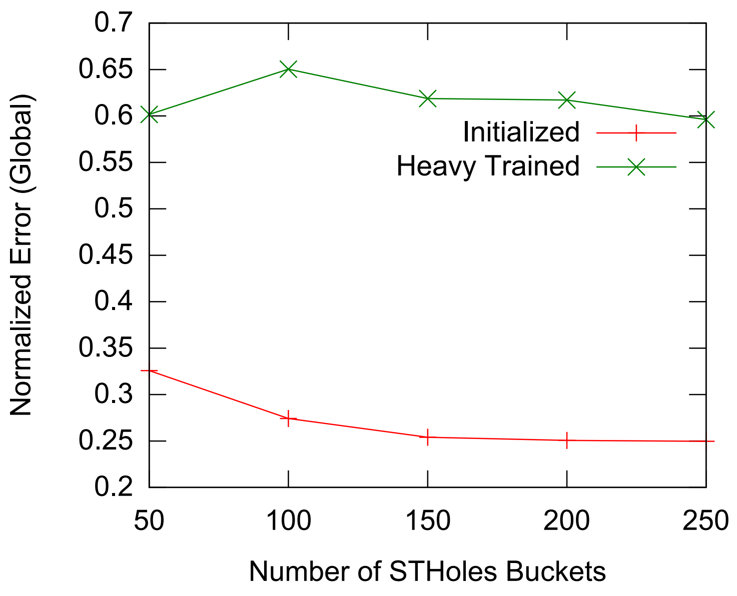


Figure 6.14: Error comparison of heavily-trained vs Initialized histograms, *Sky* $- 1\%$ setting.

trained histogram. Comparing Figures 6.12 and 6.14, we see that the error rate of the heavily-trained version is actually a bit higher than that of the normally trained histogram (both are uninitialized histograms). The reason is twofold. First, due to

Stagnation, extra training does not provide benefits after a certain number of queries. Second, there is variance due to different workloads – another manifestation of *Sensitivity to Learning*. This confirms Conjecture 1. Histogram learning stagnates after a number of learning queries. We have observed this effect throughout datasets and different workloads.

7 Probabilistic Cost Estimation

Abstract. *This chapter studies the problem of predicting cardinality distributions of query results, for multi-dimensional query predicates. Related work has not investigated this problem 'in full beauty', e.g., has studied predicting cardinality distributions, but for the uni-dimensional case. As a first contribution, we propose two methods which estimate cardinality distributions for the multi-dimensional case. The methods are computationally inexpensive and do not need to store any data in addition to the underlying histogram. We prove that one of the methods is optimal under non-restrictive assumptions. A problem which one must address when studying distribution-based estimation methods is the comparison of such methods. A straightforward comparison of distribution-based and point-based methods is rather costly. We derive formal criteria allow to bypass these costly comparisons, for a broad class of cost functions. An experimental comparison of our methods shows that they are a significant improvement over a baseline point-based cardinality estimation method.* □

7.1 Introduction

Problem Statement. Predicting the result size of queries based on histograms is an important problem. While much work has gone into estimating the expected result size/cardinality, considerably less work, e.g., [BC05], has studied how to estimate the cardinality distribution, i.e., how likely is it that the result is larger than a threshold value. We are not aware of any proposals for such estimations in the multi-dimensional case. In this case, predictions based on uni-dimensional histograms may be grossly off [PI97]. In a first step, we investigate exactly this problem – estimating the cardinality distribution based on multi-dimensional histograms, and we propose two such estimation methods. We assume that attributes are numeric.
A common assumption behind query optimization is that it suffices to find the best plan for the expected value of the cardinality. This results in good query plans when the cost of the plan is linear against the cardinality [CHS99, JHG02]. (A cost function says how expensive a query or an operator of the query is, given the size of its input.) However, linear cost functions do not necessarily prevail in practice.

Example 7.1: Given the relations Customers (C) and Orders (O), we have the following query:

```
SELECT * FROM C JOIN O ON
C.CID = O.CID AND
O.Quantity > 100 AND O.Total > 550
```

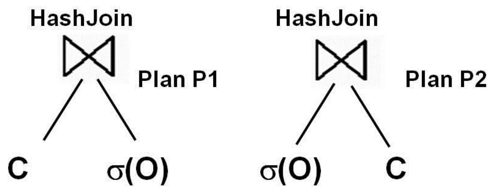


Figure 7.1: Hash join plans for the example query.

Figure 7.1 shows two alternative execution plans. The left relation of the plan tree is the input and the right relation is the probe. Whether P1 or P2 is cheaper depends on the selectivity of the filtering condition

```
O.Quantity > 100 AND O.Total > 550
```

We denote by $\sigma(O)$ the relation O filtered by that condition.

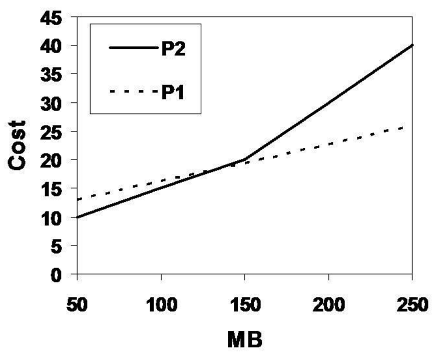


Figure 7.2: The dependency of the join cost from $\sigma(O)$

Figure 7.2 depicts the dependency of the plan costs on the size of $\sigma(O)$. The free main memory buffer of the DBMS is 150MB. The plan P1 has a cost linear against the size; this is because $\sigma(O)$ is the probe. The plan P2 is *piecewise linear*: If the size of $\sigma(O)$ is less than 150MB, it fits into main memory; then the hash algorithm needs only one pass. Otherwise, it needs at least two passes. The cost functions for plans P_1 and P_2 are:

$$P_1(x) = \frac{13}{200}x + \frac{39}{4}$$

and

$$P_2(x) = \begin{cases} \frac{1}{10}x + 5 & x \in [50, 150) \\ \frac{1}{5}x - 10 & x \in [150, 250) \end{cases}$$

If the optimizer estimates the size of $\sigma(O)$ to be 130MB, it chooses P2, as it is slightly cheaper than P1. Instead assume that the optimizer estimates that result sizes from 60MB to 200MB are equally likely. The expected result size is the same – 130MB, but now P1 is the cheaper plan. Simply taking the expected cardinality and computing the cost for it does not suffice. ■

This example illustrates two important points:

1. The plan costs are rarely linear. Almost all physical operators in a relational system are linear or almost linear. However, depending on the size of available memory buffers, they need one, two or several passes over the input. This results in piecewise linear functions like in Example 1.

2. When the cost function of a plan is piecewise linear, the conventional cost estimation procedure which takes the cost for the expected cardinality can produce bad cost estimates. They, in turn, will result in poor plan choices.

To demonstrate how different the estimated and the real costs of a query can be, we make use of the *Picasso* tool [RH05]. It allows to visualize the dependency of query-plan cost on the selectivity. It divides the selectivity space into steps and "asks" the database for the optimal plan and its cost for each point in the selectivity space. It then executes the queries and compares the real costs to the estimated ones. We have used the TPC-H query 9 and the PostgreSQL DBMS and depicted the results in Figure 7.3. Different colors correspond to different plans. Looking at the estimated

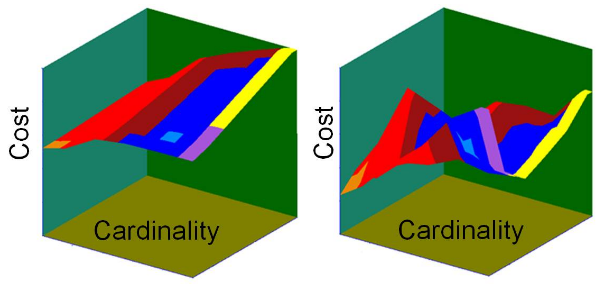


Figure 7.3: The estimated (left) and real plan costs for the TPC-H query 9

(left) and real (right) costs shows how far off the optimizer can be. The shape of the right plot (real costs) also shows that the estimated optimal plans are very unstable – a small change in parameter values can increase execution costs by much. Note that the estimated costs are monotonic increasing with cardinality, while for real costs this is not the case – another indicator that the cost estimates have been imprecise. [RH05] shows that this effect holds for different queries and popular commercial optimizers.

Parameterized queries are another example where optimizing for the expected cardinality can prove inefficient. The parameter values are not known beforehand, but the optimizer has to compile and store an execution plan. Taking the expected values for the parameters can lead to suboptimal plans [JHG02].

The methods we propose rely on an exiting histogram. In this chapter to we use STHoles as the underlying histogram. The methods, however, are applicable with slight modifications if the underlying histogram is different.

Difficulties. There are several challenges when designing a method which estimates probability distributions over cardinalities. One is that the approach for the uni-dimensional case in [DR99] does not generalize for the multi-dimensional case (more about this in Section 7.2). Another challenge is that the methods envisioned should be computationally inexpensive and memory-efficient. A further difficulty is that, since our methods issue distributions instead of point estimates, we cannot evaluate them using conventional metrics, e.g. normalized estimation error. In theory, we could evaluate our methods by building several versions of a least-expected cost query optimizer which estimate the cardinality distribution in different ways and then by comparing the resulting plans. However, this is not feasible, for two reasons. The first one is that this is overly expensive computationally. Next, this approach is bound to specific cost functions. This is an issue with related work as well, e.g., [DR99]. All this calls for an investigation of how to compare the estimation methods envisioned.

Contributions. As mentioned, we propose two methods to derive cardinality distributions. The first method, dubbed Sample-based method, treats past query executions as a sample and approximates the distribution from it. It is simple and efficient. The second method (the Uniformity method) is a generalization of a common cardinality-estimation method: It uses the uniform spread assumption and yields a distribution instead of a point estimate. The Uniformity method comes together with an important result: We prove that it is optimal under certain assumptions (non-restrictive, we argue): it minimizes the estimation error.

Regarding the evaluation problem, we have derived conditions sufficient to conclude that one estimation method is better than another one. We stress that these conditions do not rely on specific cost functions, but rather on general characteristics of cost functions, e.g., 'The cost function is convex'. The analysis addresses important and commonly used cost functions, as we will explain. To complement the analysis, we have carried out experiments that quantify the predictive power of our methods. We compared our methods with a point-estimation method from [BCG01]: Our methods offered better cost estimates.

7.2 Related Work

In this section, we brieflty discuss probabilistic (least expected cost) query optimization.
Related work [CHS99, JHG02, BC05, DR99] points out that optimizing the query for the expected cardinality value can be insufficient. [DR99] discusses optimization of relational Top-N queries: Here, one needs precise cardinality estimates for the intermediate result sizes. They associate a quality measure with the histogram, namely the maximum error of an open-ranged predicate over all possible attribute values. Formally, if D is the domain of attribute *Attr*, then the quality measure is:

$$\Delta = \max_{x \in D} |est(Attr \leq x) - real(Attr \leq x)| \tag{7.1}$$

$est(\cdot)$ is the estimated cardinality, and $real(\cdot)$ is the real cardinality. Thus Δ is the maximum difference between estimated and real cumulative distributions of attribute $Attr$ over the domain D. Using Δ and the specifics of the Equi-Depth histogram, it is possible to derive cardinality distributions for queries. For the multi-dimensional space, the query predicate has the form $(Attr_1 \leq x_1) \wedge \ldots \wedge (Attr_n \leq x_n)$. For this form, the cumulative distribution function is not defined. Another limitation of [DR99] is that it uses cost functions specific to their task, which is optimization of Top-N queries. In contrast, we study the problem for arbitrary cost functions.
[CHS99, JHG02] discuss the influence of various parameters on least-expected cost optimization. These parameters include available memory and other system resources, as well as selectivity estimates. The papers establish that conventional optimization is insufficient when the query cost is not linear against the cardinality. The papers also present optimization algorithms which rely on probability distributions over cardinality estimates, but they do not address how to derive such a distribution. [BC05] suggests a method to derive the cardinality distributions using samples: Here, each join one wants to optimize needs to have a precomputed sample. The cardinality distributions are then derived from this sample. This approach is applicable for star joins, to give an example. However, there is no generalization for arbitrary queries.
[Dob05] discusses various assumptions regarding tuple distribution inside the buckets and their impact on cardinality estimation for join queries. The paper shows that the uniform-spread assumption can be relaxed while obtaining the same formulas for cardinality estimates. [Dob05] considers uni-dimensional histograms; the relaxed assumptions do not readily generalize for the multi-dimensional case.
[BBD05] discusses a pro-active query optimizer. Such an optimizer may trigger a re-optimization of a query if it finds out during the execution that the cardinality estimates are wrong and the plan it has chosen is suboptimal. The optimizer uses bounding boxes around estimates to model uncertainty in estimates. These are, in fact, cardinality distributions. It uses values 0 (no uncertainty) to 6 (very high uncertainty) to model the level of uncertainty associated with an estimate. The values are

assigned based on heuristics.
[KB10] is a first stab at the problem investigated here. However, [KB10] only proposes the Sample-based method and does not feature the formal criteria for comparing distributions.

7.3 Definitions and Notation

We use histograms as our underlying data structure for cardinality estimation. Histograms store compressed information about the relation. Because of the compression, a histogram can be consistent with different relations: Each of the relations, when compressed, would yield the histogram. The set of relations *compatible* with the histogram H is denoted by $\mathcal{R}(H)$. Usually $|\mathcal{R}(H)| >> 1$. This gives way to ambiguity when issuing cardinality estimates, because for different relations in $\mathcal{R}$ the cardinality of a query can be different. To reflect this uncertainty in cardinality estimates, we model them as random variables. We write $card(q)$ to denote the cardinality of the query q. The cumulative distribution function $F(\cdot)$ of the random variable $card(q)$ is:

$$F(k) = Pr(card(q) \leq k). \tag{7.2}$$

Unless stated otherwise, all random variables are nonnegative. For each query, the cardinality cannot exceed a certain value, denoted by $max(q)$.
Given two discrete random variables X and Y with distribution functions P_X and P_Y respectively, their sum $Z = X + Y$ has the following distribution function:

$$P_Z(j) = \sum_{i=-\infty}^{+\infty} P_X(i) * P_Y(j - i) \tag{7.3}$$

Definition 7.1 (Convolution)
P_Z given in (7.3) *is the* convolution of P_X and P_Y *and is denoted as* $P_Z = P_X \circ P_Y$.
□

Each query plan π has an associated cost function, $v_\pi(\cdot)$, which maps input cardinalities to costs. Given the query q, the cost for plan π equals the expected value of $v_\pi(card(q))$:

$$cost(\pi) = E[v_\pi(card(q))] \tag{7.4}$$

Definition 7.2 (Heaviness)
Given two random variables X_1 and X_2 with probability-density functions f_1 and f_2, we say X_2 is heavier *than X_1 near point y, if for some $\delta > 0$*

$$\forall x \in (y - \delta, y + \delta) : \ f_2(x) > f_1(x) \tag{7.5}$$

If y is an endpoint of the interval, the interval becomes one-sided: $(0, \delta)$ *or* $(max(q) - \delta, max(q))$ *correspondingly.* □

7.4 Cardinality Distributions over Multi-Dimensional Histograms

In this section we propose two methods for deriving the probability distribution of cardinalities from a multi-dimensional histogram. First, we introduce the Sample-based method (Section 7.4.1) The second approach to approximate the probability distribution of cardinalities goes in Section 7.4.2. This subsection features an important result of ours: We prove that the Uniformity method is optimal when all relations compatible with the histogram are equally likely.

7.4.1 The Sample-Based Method

The Sample-based method is based on the assumption that we can treat past query execution results as a sample when approximating the random variable $card(q)$. Let X be a random variable and $\{x_1, \ldots, x_m\}$ be a sample. In the one-dimensional case, one can approximate the cumulative distribution function of X, $F(z) = Pr(X \leq z)$, from the sample:

$$F_m(z) = \frac{1}{m} \sum_{i=1}^{m} I(x_i \leq z) \tag{7.6}$$

where $I(P)$ is the "indicator" function, it equals 1 if the predicate P is true and 0 if it is false. As m grows, F_m converges to F [Kol41]. The constant $1/m$ is the normalizer.

Example 7.2: Let m = 5, and $x_1 = 0.4$, $x_2 = 3$, $x_3 = 1.2$, $x_4 = 1.3$ and $x_5 = 1.9$. We now want to estimate the probability that $X \leq 1.2$. According to Equation (7.6),

$$F_5(1.2) = \frac{1}{5}(I(0.4 \leq 1.2) + I(3 \leq 1.2) + \\ + I(1.2 \leq 1.2) + I(1.3 \leq 1.2) + I(1.9 \leq 1.2)) = 2/5$$

■

As in Section 7.4.2, let b be the bucket which encloses q.
$b_1, \ldots, b_m$ are the child buckets of b. To estimate the distribution of selectivities inside b, we use the selectivities we have already observed within this region. These are:

- The selectivities of the child buckets:
  $sel(b_1), \ldots, sel(b_m)$.
- The selectivity s which corresponds to the region covered by b, excluding its child buckets:

$$s = \frac{n(b) - \sum n(b_i)}{vol(b) - \sum(vol(b_i))}$$

If we deem $\{s, sel(b_1), \ldots, sel(b_m)\}$ a representative sample of the selectivities inside the bucket, we can approximate the cumulative distribution function of selectivities using Formula (7.6). – Note that this is not the only plausible way to select sample buckets. Depending on buckets we deem representative, we would obtain variations of the Sample-based method. E.g., we could treat only the buckets which intersect with the given query as a sample: But this may lead to small sample sizes and high variance. However, while selecting the sample differently is a tuning option, it does not change the essence of the method. To not clutter formulas without offering much additional insight, we limit the presentation to that relatively simple yet representative variant.

Because the buckets can have different volumes, we weight the "evidence" with its relative volume. For the child bucket b_i this is $vol(b_i)/vol(b)$, and for s it is $(1 - \sum vol(b_i)/vol(b))$. The formula for the cumulative distribution is

$$\begin{aligned} Pr(sel(q) \leq x) = (1 - \sum_{i=1}^{m} \frac{vol(b_i)}{vol(b)}) \cdot I(s \leq x) \\ + \sum_{i=1}^{m} \frac{vol(b_i)}{vol(b)} \cdot I(sel(b_i) \leq x) \end{aligned} \tag{7.7}$$

Example 7.3: Figure 7.4 shows a histogram with 7 buckets (the root bucket has the largest bounding box) and a query, which intersects with all buckets. Each child bucket spans $\approx 5\%$ of the parent-bucket area. Three buckets have selectivity 10, three

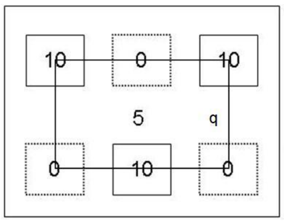


Figure 7.4: A histogram with query q

buckets have selectivity 0, and the root bucket has selectivity 5. According to (7.7), the probability that the selectivity is less than or equal to 7 is

$$Pr(sel(q) \leq 7) = 0.7 \cdot I(5 \leq 7) + 3 \cdot 0.05 \cdot I(0 \leq 7) + \\ + 3 \cdot 0.05 \cdot I(10 \leq 7) = 0.85$$

■

The Sample-based method issues probabilities from the range $[0, 1]$. In (7.7), if all indicator functions $I(sel(b_i) \leq x) = 0$ and $I(s \leq x) = 0$ for some x, then $Pr(sel(q) \leq x)$ is 0.

The Sample-based method does not incur storage costs additional to the ones of the underlying histogram.

Observation 1. *The computational complexity of calculating $Pr(sel(q) \leq x)$ is $O(m)$.*

Rationale. Calculating expressions $I(sel(b_i) \leq x)$ and $vol(b_i)/vol(b)$ in (7.7) require constant time, so the one of $Pr(sel(q) \leq x)$ requires $O(m)$ operations. □

7.4.2 The Uniformity Method

The Uniformity method is a generalization of the conventional cardinality-estimation procedure, which yields point estimates, to obtain distributions. It does not require any information beyond the one contained in the histogram. The method relies on the Continuous Value Assumption or the uniform-spread assumption as it is sometimes called in the context of multi-dimensional histograms [BCG01]. The essence is that any tuple has equal chance to appear anywhere in the bucket. Let $Q = \{b_1, \ldots, b_k\}$ be the histogram buckets which intersect with query q. Let D_i, $1 \leq i \leq k$, be a random variable which models the cardinality of intersection $b_i \cap q$. According to (4.4 on page 31), we compute the cardinality of q as follows:

$$card(q) = \sum_{i=1}^{k} D_i \tag{7.8}$$

We compute the probability distribution function of $card(q)$ (dubbed P_q) by computing the distribution functions for intersections D_i (dubbed P_i) and then their convolution [Ros09]:

$$P_q = P_1 \circ P_2 \circ \ldots \circ P_k \tag{7.9}$$

Next, we show how to compute P_i. Then we discuss how to compute the convolution efficiently.

Let $b \in Q$, and assume that b has a child bucket b_c which is fully enclosed in $q \cap b$ (Figure 7.5). Query q intersects with b_{root}, b and b_c. Computing the distribution for the intersection $b_{root} \cap q$ is analogous to $b \cap q$, so we show how to calculate the distributions for intersections $b \cap q$ and $b_c \cap q$.

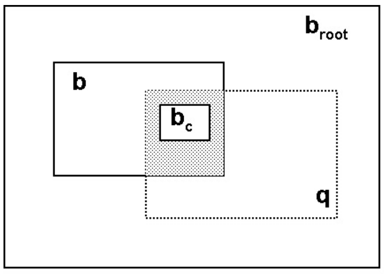


Figure 7.5: Query q, partially intersecting with bucket b.

Observation 2. *The cardinality of the intersection* $q \cap b_c$ *equals* $n(b_c)$ *with probability* 1.

Rationale. b_c is inside q, thus, with probability 1, all tuples belonging to b_c also belong to q. Thus, for the intersection $b_c \cap q$ the corresponding random variable takes one value, $n(b_c)$, with probability 1. □

Observation 3. *The cardinality distribution for the intersection* $q' = q \cap b$ *is given by:*

$$Pr(card(q') = m) = B(m; n(b), \frac{vol(q')}{vol(b)}) \tag{7.10}$$

$B(m; n, p)$ *is the Binomial distribution:*

$$B(m; n, p) = \binom{n}{m} p^m (1-p)^{n-m} \tag{7.11}$$

Rationale. q' is the dotted region in Figure 7.5. Each tuple t which belongs to bucket b can either be in the region q' or in $b \backslash q$. Due to the Uniformity assumption, the probability that t is inside q' is $vol(q')/vol(b)$, and the probability that it is inside $b \backslash q$ is $1 - vol(q')/vol(b)$. From here we obtain Binomial distribution (7.10). □

The expected value of the probability distribution obtained according to Equation (7.10) is:

$$E[B(m; n(b), \frac{vol(q')}{vol(b)})] = n(b) \cdot \frac{vol(q')}{vol(b)}$$

We now show that the distributions obtained using the Uniformity method are optimal if certain conditions hold. We assume that the data domain is discrete: For continuous data the proof mechanism is similar, but the proof is longer.

Theorem 7.4.1. *Suppose that each relation in* $\mathcal{R}(H)$ *is equally likely to occur. Then the distribution* P_q *according to* (7.9)*, where each* P_i *is a distribution given by* (7.10)*, is optimal, in the sense that it minimizes the error*

$$\sum_{r \in \mathcal{R}(H)} \sum_{q \in W} |E[v(card(q))] - v(c(q, r))| \tag{7.12}$$

where W is a set of queries, and $c(q,r)$ is the real cardinality of query q for relation r.

Proof. For any fixed query $q \in W$, the estimated distribution $E[v(card(q))]$ is the same for all $r \in \mathcal{R}$. The actual cardinality – $c(q,r)$ depends on r. It follows from here that in order to prove the theorem we need to show that the Uniformity method minimizes

$$\sum_{r \in \mathcal{R}(H)} |E[v(card(q))] - v(c(q,r))| \tag{7.13}$$

for a fixed query q. Let

$$\mathcal{R}(H) = \{r_1, \dots, r_t\}$$

(We know it is a finite set because the data domain is discrete). Consider the vector $V = (v_1, \dots, v_t)$, where $v_i = v(c(q,r_i))$. Then, minimizing (7.13) is equivalent to minimizing the distance between the vectors V and $E = (e, \dots, e)$ where $e = E[v(card(q))]$. To minimize the distance, e has to be equal to:

$$e = \frac{1}{t}\sum_{i=1}^{t} v_i$$

But e is the expected cost of $card(q)$, so:

$$e = \sum_{x=0}^{n(q)} p(x)v(x) = \frac{1}{t}\sum_{i=1}^{t} v_i \tag{7.14}$$

We distinguish two cases, when $t = |\mathcal{R}(H)| \geq n(q)$ and when $t = |\mathcal{R}(H)| < n(q)$.
Case 1. $|\mathcal{R}(H)| \geq n(q)$. On the right-hand side of (7.14) we have summands in the form $v_i = v(c(q,r_i))$. Because $c(q,r_i))$ accepts values from the range $[0, n(q)]$, and $t \geq n(q)$, we can rewrite the sum as follows:

$$\sum_{i=1}^{t} v_i = \frac{1}{t}\sum_{i=0}^{n(q)} N_i \cdot v_i$$

where N_i is the number of relations in $\mathcal{R}(H)$ for which $c(q,r) = i$. It suffices to show that

$$N_i = t \cdot p(i) \tag{7.15}$$

Recall that $p(i)$ is the probability given by (7.10). (7.15) is easiest to show if we notice that $N_i = p'(i) \cdot t$, where $p'(i)$ is the probability of query cardinality being i in $\mathcal{R}(H)$. Under the assumption that all relations in $\mathcal{R}(H)$ are equally likely we can state that each tuple with equal probability can "appear" anywhere inside the bucket space. This means we can use the same reasoning as in Observation 3, thus $p'(i) = p(i)$.

Case 2. $t = |\mathcal{R}(H)| < n(q)$ mean that we have duplicate tuples, which means all $p(x)$ in (7.14) are 0 for $x < n(q) - |\mathcal{R}(H)|$. After eliminating them from the sum, we can proceed as in Case 1. $\square$

Now it remains only to calculate the convolution according to Equation (7.9). If both distributions P_X and P_Y contain N points, then computing $P_{X+Y} = P_X \circ P_Y$ directly using Equation (7.3) requires $O(N^2)$ operations. A less expensive method relies on the fact that

$$\mathcal{F}(P_X \circ P_Y) = \mathcal{F}(P_X) \cdot \mathcal{F}(P_Y)$$

where $\mathcal{F}$ is the Fourier transform operator. From here, applying the inverse transform $\mathcal{F}^{-1}$ we obtain:

$$P_{X+Y} = \mathcal{F}^{-1}(\mathcal{F}(P_X) \cdot \mathcal{F}(P_Y))$$

In this case, the complexity is $O(N \log N)$ [Pre07].

7.5 Comparison Metrics

To evaluate estimation methods for cardinality distributions, we need metrics which deal with distributions. However, pure distance measures for distributions such as the ones in [RTG00] do not help in our case because they do not take the query-plan costs into account. This is a crucial difference between point and distribution estimates. With point estimates, there is the implicit assumption that the costs are (almost) linear against the cardinality, and in this case one can directly compare the estimated and the real cardinalities. This is because for a linear function $l(\cdot)$

$$E[l(X)] = l(E[X])$$

that is, to compute the expected cost according to a linear function, we need to know only the expected value of the distribution.

With non-linear cost functions, however, the distribution that yields a better approximation of the real query-plan depends on the cost function. In order to derive a suitable metric, we look at how an optimizer chooses the execution plan. This might allow us to say when a cardinality distribution is superior to another one for various cost functions.

7.5.1 Query-Plan Costs

The goal of a query optimizer is to choose a low-cost plan among various alternatives. This is usually done in several steps: First, the optimizer obtains the logical execution plan of the query. Then it chooses the physical execution plan. Usually, a logical plan translates into several physical plans. Those physical plans are equivalent in terms of the results they produce; however, they usually have different costs.

Cost functions for query plans have certain characteristics, which we write down next. We rely on them in our formal analysis in the next section.

The maximum possible cardinality of the query is 250.[1] We calculate the expected costs of both plans according to $f(\cdot)$ and $g(\cdot)$.

$$\begin{aligned}
&E_f[ILookup] = \int_{50}^{250} \frac{1}{50} z \cdot N(z; 110, 10) dz \\
&\approx \frac{1}{50} \cdot 110 = 2.2 \\
&E_f[Scan] = \int_{50}^{250} 2.5 \cdot N(z; 110, 10) dz \approx 2.5 \\
&E_g[ILookup] = \int_{50}^{250} \frac{1}{50} z \cdot N(z; 135, 15) dz \approx 2.7 \\
&E_g[Scan] = \int_{50}^{250} 2.5 \cdot N(z; 135, 15) dz \approx 2.5
\end{aligned} \tag{7.16}$$

According to f, the optimal plan is $ILookup$, according to g – it is $Scan$. The reason for this is that f assigns significantly higher probabilities to cardinality values less than 110, and g does vice versa.

7.5.2 Comparing Cardinality-Estimation Methods

The following steps allow to choose and use a suitable cardinality-estimation method.

1. Identify a candidate set of methods.
2. Compare the methods using a metric, possibly by means of experiments.
3. Embed the best method from Step 2 into the query optimizer and make it available to users.

The conditions on metrics we derive in Subsection 7.5.2 will serve to choose a method, i.e., optimize Step 2, and have nothing to do with query execution. At runtime, the estimation method will be fixed. Conventionally, cardinality-estimation methods are compared (Step 2) using metrics which quantify the difference of the estimated cardinality and the real one. $|c - e|$ indicates how good the estimation is. This quantity is usually normalized by dividing by c (relative error).
However, we cannot use this metric for distributions. In theory, we could embed several (different) estimators into a full-fledged query optimizer and look at the appropriateness of the query plans chosen. To assess the quality of an estimator which

[1]The probability $Pr(card(q) > 250) \neq 0$ according to both f and g, but it is so small that we can neglect it. The same for probabilities < 50.

issues cardinality distribution X, we compare the expected cost according to X to the real cost:

$$\epsilon_X = |1 - \frac{E[v_\pi(X)]}{v_\pi(c)}| \tag{7.17}$$

This is the normalized absolute error of the estimated cost. To compare two methods which yield distributions X and Y, it seems feasible to compare ϵ_X and ϵ_Y. The smaller number indicates that the corresponding distribution is better. But this is impractical: It requires executing the queries, several times, using different physical plans and comparing them to the real cost. In other words, point-estimation methods only require comparing real and estimated cardinalities in Step 2; distribution-based methods require comparing estimated and real plan costs – for potentially very large set of plans.

This calls for more light-weight ways of assessing our estimators. Our solution is to derive optimality conditions for *classes* of cost functions. If they hold, we can compare distribution-based estimators looking only at the real cardinality of the query and skip computation of the expected cost for numerous cost functions in Step 2. This essentially is what one does when comparing point-estimation methods. Note that we need to know the real query cardinality; however, this is not a problem since we are in Step 2.

7.5.3 Optimality Conditions

In this subsection we derive two optimality criteria which cover important cost functions.

We assume that there are two methods which issue different cardinality distributions. Formally, there are two continuous random variables X and Y with cumulative distribution functions F and G, respectively: $Pr(X \leq z) = F(z)$ and $Pr(Y \leq z) = G(z)$. The maximum possible cardinality for the query is N, which does not depend on the cardinality distribution. Thus $F(N) = G(N) = 1$. We denote the density function of X with f and the one of Y with g.

Lemma 7.5.1. (JENSEN'S INEQUALITY) *If $v(\cdot)$ is a convex function, then*

$$E[v(X)] \geq v(E[X])$$

If $v(\cdot)$ is a concave function, then

$$E[v(X)] \leq v(E[X])$$

See [Kra99] for a proof.

We continue by formulating a lemma which will help us to establish criteria for comparing two estimators for cardinality distributions.

Lemma 7.5.2. *If Y is heavier than X near N, and $E[X] \leq E[Y]$, and $v(\cdot)$ is a convex function, then:*

$$E[v(X)] \leq E[v(Y)]$$

Proof. The expected costs of X and Y are

$$\begin{aligned} E[v(Y)] &= \int_0^N v(y)g(y)dy \\ E[v(X)] &= \int_0^N v(x)f(x)dx \end{aligned} \tag{7.18}$$

The theorem assertion is therefore equivalent to the following:

$$\int_0^N [g(z) - f(z)]v(z)dz \geq 0 \tag{7.19}$$

We introduce a new function φ:

$$\varphi(z) = g(z) - f(z) \tag{7.20}$$

The function φ has the following properties:

- **Property 1.** The integral of φ in the interval $[0, N]$ equals 0:

$$\int_0^N \varphi(z)dz = \int_0^N g(z)dz - \int_0^N f(z)dz = 0 \tag{7.21}$$

- **Property 2.** φ is positive near N, i.e., exists $\delta > 0$ such that $\varphi(x) > 0$ for $x \in (N - \delta, N)$.

Let δ^* be the biggest number for which the Property 2 holds:

$$\delta^* = \sup_{\delta>0}\{\delta | \varphi(\delta) \leq 0\}. \tag{7.22}$$

We define

$$l(z) = \frac{v(\delta^*)}{\delta^*} z \tag{7.23}$$

$l(z)$ has the following property: for all $z \in (0, \delta^*)$ $l(z) \geq v(z)$ and for all $z \in (\delta^*, N)$ $l(z) \leq v(z)$. Let

$$\begin{aligned} \Phi^- &= \int_0^{\delta^*} \varphi(z)dz \\ \Phi^+ &= \int_{\delta^*}^N \varphi(z)dz \end{aligned} \tag{7.24}$$

According to Property 2,

$$\Phi^- + \Phi^+ = 0 \tag{7.25}$$

Further, according to the definition of δ^*, $\Phi^+ > 0$ and $\Phi^- < 0$.

$$\int_0^N \varphi(z)v(z)dz = \int_0^{\delta^*} \varphi(z)v(z)dz + \int_{\delta^*}^N \varphi(z)v(z)dz \tag{7.26}$$

Using the aforementioned property of $l(z)$ and the fact that $\Phi^- < 0$ we obtain:

$$\int_0^N \varphi(z)v(z)dz \geq \int_0^N \varphi(z)l(z)dz = l(E[Y] - E[X]) \geq 0. \tag{7.27}$$

Q.E.D. □

Example 7.4: We continue Example 1. We have described two distributions in Example 1:

$$f(x) = \begin{cases} 1 & x = 130 \\ 0 & \text{otherwise} \end{cases}$$

and

$$g(x) = \begin{cases} \frac{1}{140} & 60 \leq x < 200 \\ \\ 0 & \text{otherwise} \end{cases}$$

f and g have the same expected value, but g is heavier than f near 200. The lemma says that the expected cost corresponding to g is not less than that corresponding to f, for any convex cost function. In particular, this holds for both P_1 and P_2. ■

Now we can prove the following important corollary:

Corollary 1. (Criterion 1.) *If Y is heavier near N than X, $E[X] \leq E[Y]$, $v(\cdot)$ is a convex function, and the real query cardinality $c \leq E[X]$, then*

$$|v(c) - E[v(X)]| \leq |v(c) - E[v(Y)]|$$

Proof. Using Lemma 7.5.2 and Jensen's inequality (Formula 7.5.1) we obtain:

$$E[v(Y)] \geq E[v(X)] \geq v(E[X]) \geq v(c) \tag{7.28}$$

The last is due to monotonicity of v. □

This first optimality criterion states that the expected cost according to the random variable X is closer to the real plan cost than the expected cost according to Y provided that the cost function is convex. The convex case is important because many non-linear physical-plan costs are convex. Section 7.6 will feature a discussion.

Example 7.5: Continuing Example 1, suppose that the real cardinality of the query is 100. We established in Example 4 that the expected plan cost according to g is not less than that according to f for any convex cost function. As the real cardinality is 100, the real cost is less than both estimated costs; consequently – f is a better estimate than g. ■

We do not need to calculate the expected costs in Step 2 to decide which distribution yields an expected cost closer to the real query-plan cost. Note that in order to compare two methods which estimate distributions, we need to compare the distributions they issue, for a set of queries. Thus, if we are able to compare distributions without much effort, we can compare estimators the same way.

Example 7.6: This example features a case which the lemmas so far do not cover. Consider again the distributions from Example 1. Now, let the real cardinality be 140. In this case, the condition of Corollary 1 does not hold, because the real cardinality is bigger than the expected values – 130. ■

Corollary 1 lets us compare distributions even when we do not know the cost function; we only need to know it is convex. Checking the heaviness of a distribution compared to another one is not difficult as well. For discrete distributions or continuous distributions given with (7.7), this means comparing the probabilities in one or several points. If the continuous distributions come from a class (normal, Poisson etc.), one can perform the heaviness check analytically.

We next provide two upper bounds for the expected cost of a query plan. We use them later to formulate optimality criteria (Lemma 7.5.5).

Lemma 7.5.3. *The following inequalities hold:*

$$E[v(X)] \leq E[X] \cdot \int_0^N \frac{dv(x)}{x} \tag{7.29}$$

and

$$E[v(X)] \leq E[X]^2 \cdot \int_0^N \frac{dv(x)}{x^2} \tag{7.30}$$

Proof.

$$E[v(X)] = \int_0^N v(x)dF(x) \tag{7.31}$$

According to the formula of integration by parts,

$$\int_0^N v(x)dF(x) = v(x)F(x)|_0^N - \int_0^N F(x)dv(x) \tag{7.32}$$

$v(x)F(x)|_0^N = t(N)$, because $F(N) = 1$ and $F(0) = 0$.

$F(x) = P\{X \leq x\} = 1 - P\{X > x\}$, thus,

$$\begin{aligned} E[v(X)] = v(N) - \int_0^N F(x)dv(x) = \\ t(N) - \int_0^N (1 - P\{X > x\})dv(x) \end{aligned} \tag{7.33}$$

From here we obtain

$$E[v(X)] = v(N) - \int_0^N dv(x) + \int_0^N (P\{X > x\})dv(x) \tag{7.34}$$

$v(0) = 0$, so

$$E[v(X)] = \int_0^N P\{X > x\}dv(x) \tag{7.35}$$

From here we obtain (7.29) using the Markov's inequality:

$$P\{X \geq \delta\} \leq \frac{E[X]}{\delta}$$

Similarly, we obtain (7.30) using the Chebyshev's inequality:

$$P\{X \geq \delta\} \leq \frac{E[X]^2}{\delta^2}$$

□

It is worth mentioning that it is possible to obtain tighter bounds by putting restrictions on the distribution function of X. However, the distributions that appear in reality are not necessarily "nice". Instead, they are distributions one has to construct from the scarce information contained in the histogram. For this reason, we refrain from deriving further criteria by making stronger assumptions regarding the distributions. Instead, we deem it more natural to look at certain classes of cost functions and to simplify the upper bounds obtained in Lemma 7.5.3.

Lemma 7.5.4. *If the cost function $v(\cdot)$ is convex and differentiable, then*

$$E[v(X)] \leq E[X] \cdot N \cdot v'(N)$$

Proof. We have established in Lemma (7.5.3) that

$$E[v(X)] \leq E[X] \cdot \int_0^N \frac{dv(x)}{x} \tag{7.36}$$

The proof follows immediately from the fact that for a convex, monotonous differentiable function $v(\cdot)$

$$dv(x) \leq v'(N)dx \tag{7.37}$$

□

Using Lemma 7.5.4 we can formulate another criteron.

Lemma 7.5.5. (CRITERION 2.) *Let X and Y be random variables, the cost function $v(\cdot)$ is convex and differentiable, the real query cardinality $c \leq E[X]$, and*

$$v(E[Y]) \geq E[X] \cdot v'(N) \cdot N$$

Then

$$|v(c) - E[v(X)]| \leq |v(c) - E[v(Y)]|$$

Proof. Using the Jensen's inequality we obtain:

$$E[v(X)] \leq E[X] \cdot \int_0^N \frac{dv(x)}{x} \leq E[v(Y)] \tag{7.38}$$

And

$$v(c) \leq v(E[X]) \leq E[v(X)] \tag{7.39}$$

Combining these two we obtain that

$$v(c) \leq E[v(X)] \leq E[v(Y)] \tag{7.40}$$

□

Example 7.7: $v(x) = x^2$, $N = 100$, $E[Y] = 90$, $E[X] = 40$, and the real cardinality $c = 25$. Then the conditions of Lemma 7.5.5 hold:

$$v(E[Y]) \geq E[X] \cdot v'(N) \cdot N$$

and $E[v(X)]$ is closer to the real cost than $E[v(Y)]$. ■

Observe the difference between Lemma 7.5.5 and Corollary 1. In Lemma 7.5.5, we have further conditions on the cost function $v(\cdot)$, while in Corollary 1 we have more conditions on the distributions X and Y.

7.6 Experimental Evaluation

The lemmas from Section 7.5.2 enable us to compare cardinality distributions, under various assumptions. They allow to bypass calculating the expected costs. They are applicable to convex cost functions, or if the distributions fulfill certain conditions. This is very valuable if one needs to choose the better one of two distributions. The lemmas from Section 7.5.2 do not tell us *how much* one method is better compared to the other one. Also, we would like to compare our methods to a baseline, i.e., a

point-estimation method such as STHoles.
In this section, we describe such an evaluation of the Sample-based and the Uniformity methods. We first describe our experimental setup in general terms; in Section 7.6.2 we describe the technical details; the experiments themselves are in Section 7.6.3.

7.6.1 Experiments – Overview

In the following, the cost of an operation is the number of I/O accesses performed. To focus on the impact of the data distribution, we fix the cost function to $n \cdot logn$. This is the cost of a multi-pass hash-join when both relations have size $M >> B$, where B is the available memory [HCLS97].

$$cost(HJ) = O(M \cdot \log_B M) \tag{7.41}$$

Note that this cost function is convex. We use a set of queries Q (see Section 7.6.2) and calculate the normalized average error of the query-cost estimation with the two methods:

$$\epsilon = \frac{1}{|Q|} \sum_{q \in Q} \epsilon_q \tag{7.42}$$

ϵ_q for the Uniformity and Sample-based methods is given by:

$$\epsilon_q = |1 - \frac{E[\pi(X)]}{\pi(c)}|$$

For STHoles ϵ_q is

$$\epsilon_q = |1 - \frac{\pi(est)}{\pi(c)}|$$

where est is the estimated cardinality for query q using the STHoles histogram. The method which produces the smaller normalized average error is better. For the distributions produced by the Sample-based and the Uniformity methods, we can calculate the expected costs using the following formula:

$$E[v(X)] = \sum_x v(x) \cdot Pr(X = x) \tag{7.43}$$

This is because, for both the Sample-based and the Uniformity methods, the distributions are discrete.

7.6.2 Experiments – Data

We used three datasets in the experiments: *Array* , *Gauss* , and *Census* . All are described in Section 4.7.2 on page 44. Table 7.1 lists the parameters of the distributions behind the datasets.

Data Set	Attribute	Value
Common	d: dimensionality	2
	N: cardinality	500,000
	data domain	$[0, \ldots, 1000]^d$
Array	distinct attribute values	50
	z:skew	1
Gauss	number of peaks	20
	standard deviation	50

Table 7.1: Description of data sets

The queries in our workload are generated as described in Section 4.7.3 on page 47.

One run of our simulations consists of 1000 queries. We vary the maximal number of buckets in the histogram (100 to 300), in line with related work. We plot the ϵ-metric for both Sample-based and the Uniformity methods using the cost functions from Section 7.6. The X-axis is the maximal number of histogram buckets, the Y-axis is the value of the ϵ-metric.

7.6.3 Experiments

Figures 7.6, 7.7, 7.8 and 7.9 show the $epsilon$-measures for the following settings: *Gauss[Uniform, 1%]*, *Array[Uniform, 1%] [Data, 1%]* and *[Uniform, 0.5%]*. In all experiments, the Uniformity method has performed the best, the Sample-based method was second-best.

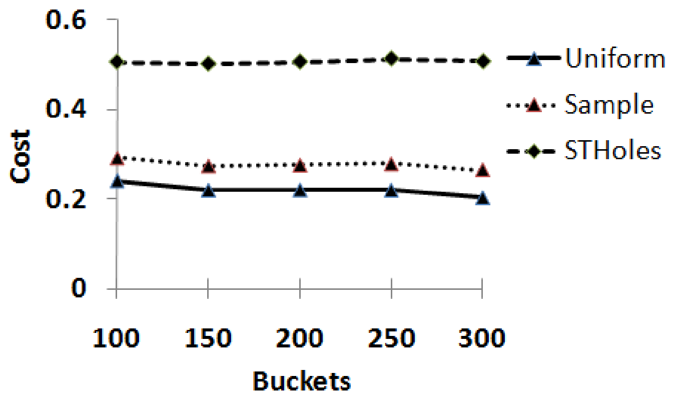


Figure 7.6: ϵ-measures for the $Gauss[Uniform, 1\%]$ setting

In Figures 7.6 and 7.7, our methods are clear winners against the baseline STHoles approach. Figure 7.9 is the only setting where the STHoles slightly outperforms one

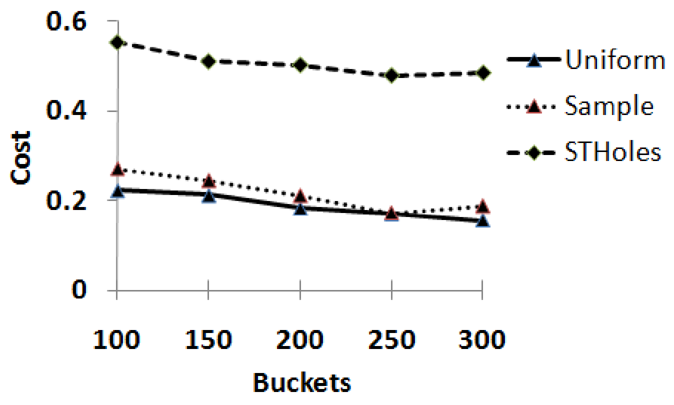


Figure 7.7: ϵ-measures for the $Array[Uniform, 1\%]$ setting

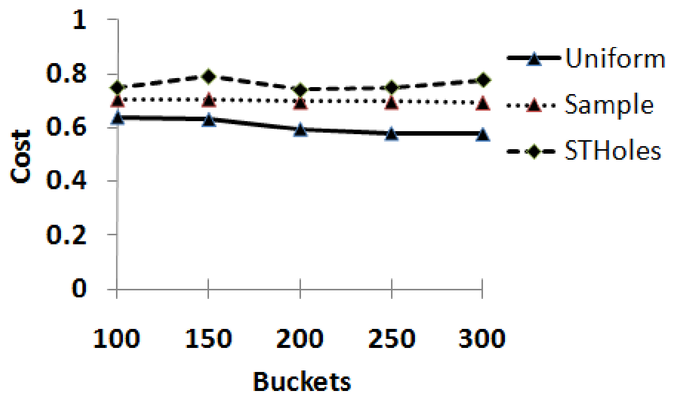


Figure 7.8: ϵ-measures for the $Census[Data, 1\%]$ setting

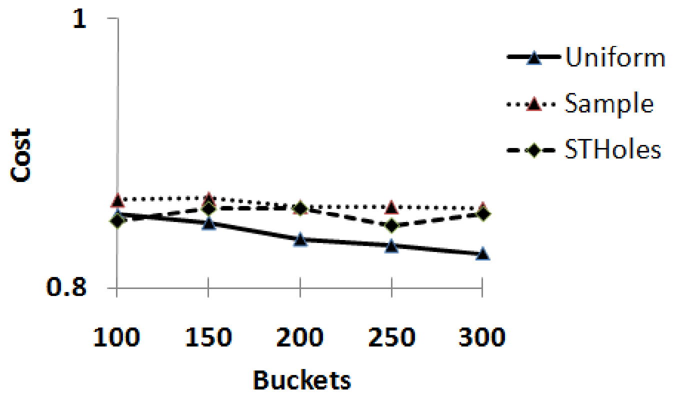


Figure 7.9: ϵ-measures for the $Census[Uniform, 0.5\%]$ setting

of our methods, the Sample-based method.

In the experiments, the values of ϵ decrease slightly – they start higher for 100 histogram buckets and decreases as the number of buckets increases. This effect is expected.

Summing up, the evaluation of the Uniformity and of the Sample-based methods shows that those methods outperform the point-based reference method. The Uniformity method is the most precise method, while the Sample-based method is a lightweight alternative with somewhat worse performance.

8 Conclusions

8.1 Self-Tuning Histograms And Their Problems

Self-tuning histograms rely on the query feedback for histogram construction. This allows them to stay up-to-date to changing data and amortize the high construction costs of static approaches. On the other hand, query feedback information can provide only a local view into the data.

We have shown that relying solely on the query feedback is insufficient. We investigated the idea of initializing histograms with subspace clusters. The reason for using subspace clustering is that they attempt to find dense regions of objects together with their relevant attributes. This is similar to what histogram construction algorithms try to achieve. The subspace clusters do not necessarily have the same format as histogram buckets, where the compactness of the representation is a central issue. We found out how to optimally transform a cluster into a histogram bucket, and also found out that it is too expensive computationally. Instead we have proposed a cheaper heuristic-based alternative. Initializing self-tuning histograms with subspace clustering results allows the histogram to converge to a better bucket structure with less error. However, clustering algorithms differ significantly on how good they perform as initializers. One of the algorithms we tried outperformed the others consistently in our experiments. Some other subspace clustering algorithms turned out to have no positive effect as histogram initializers.

Having found out that initialization works, we turned our focus to the histograms. Our goal was to find and categorize the problems which uninitialized histograms face in learning the dataset. We found out that the main problem is the sensitivity of the histograms to query order (we coin it sensitivity to learning). The first queries define the top-level bucket structure of the histogram, if this structure is not good, then subsequent learning is typically unable to "fix" it. This results in a suboptimal bucket configuration. In particular, such a bad configuration might be unable to capture the relevant subspace in which the bucket should be created; instead, lower or higher dimensional buckets will be created. Subspace clustering enables the histogram to start with a few buckets which capture the dense data clusters together with their relevant dimensions. Subsequent learning enables the histogram to refine itself and achieve much lower error rates compared to the uninitialized version. In this part, we fixed the subspace clustering algorithm. This algorithm outputs rectangular clusters, so the transformation of the clusters to histogram buckets became trivial. We conducted targeted experiments to find out how initialization helps with the aforementioned problem of sensitivity to learning.

Overall, initialization provides a clear improvement for a self-tuning histogram. The histogram becomes much less sensitive to workload and converges to considerably lower error rates. Uninitialized histograms do not achieve such error rates regardless of the amount of learning they are given.

8.2 Improved Cost Model

The second contribution of this thesis is how to support an improved cost model using existing histograms. A non-linear cost model is more accurate than assuming all plan costs are linear against the input cardinality. However, cost estimation using for a non-linear model is more complex. Instead of cardinality estimates we need a probability distribution of possible cardinalities to compute the expected cost of a plan. We show how to support such distribution-based estimates using only multi-dimensional histogram in place. We use the Continuous Value Assumption to derive the cardinality distribution, and show it is optimal under certain non-restrictive assumptions. Thus, we show that a better cost model can be supported using nothing more than existing histograms.

Bibliography

[AGGR98] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *SIGMOD Record*, 27(2), 1998.

[AKMS07] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. Dusc: Dimensionality unbiased subspace clustering. In *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, pages 409–414, Washington, DC, USA, 2007. IEEE Computer Society.

[AKMS08] Ira Assent, Ralph Krieger, Emmanuel Müller, and Thomas Seidl. Inscy: Indexing subspace clusters with in-process-removal of redundancy. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, pages 719–724, Washington, DC, USA, 2008. IEEE Computer Society.

[Ame94] Nina Amenta. Bounded boxes, hausdorff distance, and a new proof of an interesting helly-type theorem. In *Proceedings of the tenth annual symposium on Computational geometry*, SCG '94, pages 340–347, New York, NY, USA, 1994. ACM.

[AWY$^+$99] Charu C. Aggarwal, Joel L. Wolf, Philip S. Yu, Cecilia Procopiuc, and Jong Soo Park. Fast algorithms for projected clustering. *SIGMOD Rec.*, 28:61–72, June 1999.

[BBD05] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 107–118, New York, NY, USA, 2005. ACM.

[BC05] Brian Babcock and Surajit Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 119–130, New York, NY, USA, 2005. ACM.

[BCG01] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: a multidimensional workload-aware histogram. In *In SIGMOD*, pages 211–222, 2001.

[BCG02] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27:153–187, June 2002.

[BGRS99] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is ”nearest neighbor” meaningful? In *Proceedings of the 7th International Conference on Database Theory*, ICDT ’99, pages 217–235, London, UK, 1999. Springer-Verlag.

[BL04] Francesco Buccafurri and Gianluca Lax. Fast range query estimation by n-level tree histograms. *Data Knowl. Eng.*, 51:257–275, November 2004.

[BLS$^+$08] Francesco Buccafurri, Gianluca Lax, Domenico Saccà, Luigi Pontieri, and Domenico Rosaci. Enhancing histograms by tree-like bucket indices. *The VLDB Journal*, 17:1041–1061, August 2008.

[BMB06] Linas Baltrunas, Arturas Mazeika, and Michael Bohlen. Multidimensional histograms with tight bounds for the error. In *Proceedings of the 10th International Database Engineering and Applications Symposium*, pages 105–112, Washington, DC, USA, 2006. IEEE Computer Society.

[CGHJ12] 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.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *In PODS*, pages 34–43, 1998.

[CHS99] Francis Chu, Joseph Y. Halpern, and Praveen Seshadri. Least expected cost query optimization: An exercise in utility. In *In Proc. 18th ACM Symp. Principles of Database Systems*, pages 138–147, 1999.

[CLRS09] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

[DGR01] Amol Deshpande, Minos Garofalakis, and Rajeev Rastogi. Independence is good: dependency-based histogram synopses for high-dimensional data. *SIGMOD Rec.*, 30:199–210, May 2001.

[Dob05] Alin Dobra. Histograms revisited: when are histograms the best approximation method for aggregates over joins? In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS ’05, pages 228–237, New York, NY, USA, 2005. ACM.

[DR99] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top n queries. In *In VLDB*, pages 411–422, 1999.

[EL07] Todd Eavis and Alex Lopez. Rk-hist: an r-tree based histogram for multi-dimensional selectivity estimation. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, CIKM '07, pages 475–484, New York, NY, USA, 2007. ACM.

[FHL07] Dennis Fuchs, Zhen He, and Byung Suk Lee. Compressed histograms with arbitrary bucket layouts for selectivity estimation. *Inf. Sci.*, 177:680–702, February 2007.

[GKTD00] Dimitrios Gunopulos, George Kollios, Vassilis J. Tsotras, and Carlotta Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 463–474, New York, NY, USA, 2000. ACM.

[GTK01] Lise Getoor, Benjamin Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. *SIGMOD Rec.*, 30:461–472, May 2001.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In Beatrice Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[HCLS97] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. Seeking the truth about ad hoc join costs. *VLDB J.*, 6(3):241–256, 1997.

[HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[HV01] Maria Halkidi and Michalis Vazirgiannis. Clustering validity assessment: Finding the optimal partitioning of a data set. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, ICDM '01, pages 187–194, Washington, DC, USA, 2001. IEEE Computer Society.

[IC93] Yannis E. Ioannidis and Stavros Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of join results. *ACM Trans. Database Syst.*, 18:709–748, December 1993.

[Ioa93] Yannis E. Ioannidis. Universality of serial histograms. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 256–267, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[Ioa03] Yannis Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, pages 19–30. VLDB Endowment, 2003.

[JHG02] Francis Chu Joseph, Joseph Halpern, and Johannes Gehrke. Least expected cost query optimization: What can we expect? In *Proc. of the ACM Symp. on Principles of Database Systems*, pages 293–302, 2002.

[Jol86] Ian Jolliffe. *Principal Component Analysis*. Springer, New York, 1986.

[KB10] Andranik Khachatryan and Klemens Boehm. Quantifying uncertainty in multi-dimensional cardinality estimations. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pages 1317–1320, New York, NY, USA, 2010. ACM.

[KKK04] Karin Kailing, Hans-Peter Kriegel, and Peer KrÃűger. Density-connected subspace clustering for high-dimensional data. In *IN: PROC. SDM. (2004*, pages 246–257, 2004.

[KKZ09] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3:1:1–1:58, March 2009.

[KMBK11] Andranik Khachatryan, Emmanuel Müller, Klemens Böhm, and Jonida Kopper. Efficient selectivity estimation by histogram construction based on subspace clustering. In *Proceedings of the 23rd international conference on Scientific and statistical database management*, SSDBM'11, pages 351–368, Berlin, Heidelberg, 2011. Springer-Verlag.

[Kol41] A. Kolmogoroff. Confidence limits for an unknown distribution function. *The Annals of Mathematical Statistics*, 12(4):pp. 461–463, 1941.

[Koo80] Robert Philip Kooi. *The optimization of queries in relational databases*. PhD thesis, Cleveland, OH, USA, 1980. AAI8109596.

[Kra99] S. G. Krantz. *Handbook of Complex Variables*. Birkhäuser Boston, 1999.

[KW99] Arnd Christian König and Gerhard Weikum. Combining histograms and parametric curve fitting for feedback-driven query result-size estimation. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 423–434, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[LLX$^+$10] Yanchi Liu, Zhongmou Li, Hui Xiong, Xuedong Gao, and Junjie Wu. Understanding of internal clustering validation measures. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 911–916, Washington, DC, USA, 2010. IEEE Computer Society.

[LWV03] Lipyeow Lim, Min Wang, and Jeffrey Scott Vitter. Sash: a self-adaptive histogram set for dynamically changing workloads. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, pages 369–380. VLDB Endowment, 2003.

[LZZ$^+$07] Jizhou Luo, Xiaofang Zhou, Yu Zhang, Heng Tao Shen, and Jianzhong Li. Selectivity estimation by batch-query based histogram and parametric method. In *Proceedings of the eighteenth conference on Australasian database - Volume 63*, ADC '07, pages 93–102, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[MAG$^+$09] Emmanuel Müller, Ira Assent, Stephan Günnemann, Ralph Krieger, and Thomas Seidl. Relevant subspace clustering: Mining the most interesting non-redundant concepts in high dimensional data. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 377–386, Washington, DC, USA, 2009. IEEE Computer Society.

[MD88] M. Muralikrishna and David J. DeWitt. Equi-depth multidimensional histograms. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 28–36, New York, NY, USA, 1988. ACM.

[MGAS09] Emmanuel Müller, Stephan Günnemann, Ira Assent, and Thomas Seidl. Evaluating clustering in subspace projections of high dimensional data. *Proc. VLDB Endow.*, 2:1270–1281, August 2009.

[MHK$^+$07] V. Markl, P. J. Haas, M. Kutsch, N. Megiddo, U. Srivastava, and T. M. Tran. Consistent selectivity estimation via maximum entropy. *The VLDB Journal*, 16:55–76, January 2007.

[MPS99] S. Muthukrishnan, Viswanath Poosala, and Torsten Suel. On rectangular partitionings in two dimensions: Algorithms, complexity, and applications. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 236–256, London, UK, 1999. Springer-Verlag.

[PHL04] Lance Parsons, Ehtesham Haque, and Huan Liu. Subspace clustering for high dimensional data: a review. *SIGKDD Explor. Newsl.*, 6:90–105, June 2004.

[PI97] 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*, VLDB '97, pages 486–495, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[PJAM02] Cecilia M. Procopiuc, Michael Jones, Pankaj K. Agarwal, and T. M. Murali. A monte carlo algorithm for fast projective clustering. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 418–427, New York, NY, USA, 2002. ACM.

[Pre07] William H. Press. *Numerical recipes : the art of scientific computing*. Cambridge University Press, 3 edition, September 2007.

[PSC84] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 256–276, New York, NY, USA, 1984. ACM.

[PSTW93] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proceedings of the twelfth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '93, pages 214–221, New York, NY, USA, 1993. ACM.

[PTFS03] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 467–478, New York, NY, USA, 2003. ACM.

[RH05] Naveen Reddy and Jayant R. Haritsa. Analyzing plan diagrams of database query optimizers. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1228–1239. VLDB Endowment, 2005.

[RKC$^+$10] Yohan J. Roh, Jae Ho Kim, Yon Dohn Chung, Jin Hyun Son, and Myoung Ho Kim. Hierarchically organized skew-tolerant histograms for geographic data objects. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 627–638, New York, NY, USA, 2010. ACM.

[Ros09] Sheldon Ross. *First Course in Probability, A (8th Edition)*. Prentice Hall, 8 edition, January 2009.

[RTG00] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. The earth mover's distance as a metric for image retrieval. *Int. J. Comput. Vision*, 40:99–121, November 2000.

[SAC$^+$79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[SDS11] SDSS Collaboration. Sloan Digital Sky Survey, August 27 2011.

[SHM$^+$06] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. Isomer: Consistent histogram construction using query feedback. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 39–, Washington, DC, USA, 2006. IEEE Computer Society.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[SZ04] Karlton Sequeira and Mohammed Zaki. Schism: A new approach for interesting subspace mining. In *IN THE PROCEEDINGS OF THE FOURTH IEEE CONFERENCE ON DATA MINING*, pages 186–193. IEEE Computer Society, 2004.

[WS03] Hai Wang and Kenneth C. Sevcik. A multi-dimensional histogram for selectivity estimation and fast approximate query answering. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '03, pages 328–342. IBM Press, 2003.

[WS08] Hai Wang and Kenneth C. Sevcik. Histograms based on the minimum description length principle. *The VLDB Journal*, 17:419–442, May 2008.

[YM03] Man Lung Yiu and Nikos Mamoulis. Frequent-pattern based iterative projected clustering. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, pages 689–, Washington, DC, USA, 2003. IEEE Computer Society.

[Zip49] George K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley (Reading MA), 1949.

[ZL96] Qiang Zhu and Per-Ake Larson. Building regression cost models for multidatabase systems. In *Proceedings of the fourth international conference on on Parallel and distributed information systems*, DIS '96, pages 220–231, Washington, DC, USA, 1996. IEEE Computer Society.

[ZL02] Qing Zhang and Xuemin Lin. On linear-spline based histograms. In *Proceedings of the Third International Conference on Advances in Web-Age Information Management*, WAIM '02, pages 354–366, London, UK, UK, 2002. Springer-Verlag.