Approximating Aggregations in Probabilistic Databases

Lo Po Tsui

Kellogg College

University of Oxford

Supervisor: Dr. Dan Olteanu

SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

September 2013

"Based on the law of probability, everything is possible because the sheer existence of possibility confirms the existence of impossibility."

Dejan Stojanovic

Abstract

Aggregate queries in probabilistic databases can return probability distributions with exponentially many possible values, which is not only computationally expensive, but also overwhelming for the user. This dissertation proposes two approximation techniques for aggregate queries in probabilistic databases: *histogram approximation* and *top-k approximation*. While histogram approximation provides a broad overview of the distribution by grouping adjacent values into bins, top-k approximation retrieves the most probable k tuples, which are often the ones most relevant to the user. Efficient algorithms with low complexity have been developed for handling MIN, MAX, COUNT, and SUM aggregations for both approximation approaches. When measured against state-of-the-art algorithms designed to perform exact query evaluation, both approximation approaches demonstrate a clear advantage in scalability and allow for a performance speedup of multiple orders of magnitude.

Acknowledgements

I would like to express my sincere gratitude to Dr. Dan Olteanu for his support, guidance and advice throughout the last five months. Dr. Dan Olteanu not only introduced me to the field of probabilistic databases, but also spent countless hours contributing ideas to the dissertation. Most importantly, the encouragement during the hard time when things were all breaking apart are motivating and have kept me moving forward. It was indeed a great pleasure to collaborate with him.

Additionally, I am thankful to Jakub Zavodny for his input into the dissertation, especially on probability bounds and concentration inequalities.

I am also grateful for the guidance and advice provided by Professor Daniel Kroening in the Michaelmas and Hilary term, which was especially important to me as the masters course is my first proper exposure to Computer Science.

It's a real pleasure to have met my friends and everyone in the department in the previous year, they have made my last year at Oxford a fulfilling and enjoyable experience. I am in particular appreciative to the lecturers and class tutors, the time they dedicated in instilling knowledge into me not only forms the cornerstone of the dissertation, but also makes me a more humble person.

Last but not least, my special thanks go to my family for their love and support since the day I was born.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background

Probabilistic Databases

As mankind enters the age of Big Data, a rich profusion of data has suddenly become available for collection and use. Data uncertainty, however, is ubiquitous in real-world applications. Examples include imprecision in sensor measurements, incompleteness in user-generated forms, ambiguity in natural language processing, probabilistic data generated by information extraction techniques [18], and of course the uncertainty inherent in risk analysis. Indeed, uncertain data is a first class citizen in databases, and the ability to handle probabilistic data will therefore represent a significant step forward in the development of the next generation of database management systems (DBMSs). Probabilistic databases are promising candidates to extend the capabilities of traditional databases by storing, retrieving, and processing probabilistic data in the same way as deterministic data.

For example, consider a web data extraction tool that gathers movie information from the internet: different sources might provide contradictory information, in which case the tool could assign a confidence level to the information gathered based on the credentials of the source. Figure 1.1 provides an example of probabilistic regarding movie gross: Avatar, for instance, might have a total gross of $400M, $700M or $900M with probabilities of 0.1, 0.5, and 0.4, respectively. (We assume here that the tuples for the same title are mutually exclusive, and that tuples with different titles are independent.)

Movie

MID	Title	Gross	Probability
1	Avatar	400	0.1
1	Avatar	700	0.5
1	Avatar	900	0.4
2	Titanic	600	0.8
2	Titanic	800	0.2

Figure 1.1: An example of probabilistic database. (Gross in the unit of millions)

Probabilistic databases can be understood intuitively via possible worlds semantics: that is, a probabilistic database can be interpreted as a probability distribution over a finite set of possible worlds, where each world corresponds to a possible deterministic database instance. For example, the database in Figure 1.1 corresponds to a total of six possible worlds, as depicted in Figure 1.2. Note that the sum of the probability of all possible worlds must equal to one, as one of the six possible worlds must correspond to the true state of the probabilistic database.

W_1

MID	Title	Gross
1	Avatar	400
2	Titanic	600

Probability $= 0.1 \times 0.8 = 0.08$

W_2

MID	Title	Gross
1	Avatar	400
2	Titanic	800

Probability $= 0.1 \times 0.2 = 0.02$

W_3

MID	Title	Gross
1	Avatar	700
2	Titanic	600

Probability $= 0.5 \times 0.8 = 0.4$

W_4

MID	Title	Gross
1	Avatar	700
2	Titanic	800

Probability $= 0.5 \times 0.2 = 0.1$

W_5

MID	Title	Gross
1	Avatar	900
2	Titanic	600

Probability $= 0.4 \times 0.8 = 0.32$

W_6

MID	Title	Gross
1	Avatar	900
2	Titanic	800

Probability $= 0.4 \times 0.2 = 0.08$

Figure 1.2: Possible worlds of the probabilistic database in Figure 1.1

In contrast, in a traditional database, only one of the contradictory tuples could be stored (with the most reasonable choice being the entry with the highest probability), resulting in the deterministic database in Figure 1.3. Such incomplete representation leads not only to a loss of information, however, but also to inaccurate query results, as demonstrated in Example 1. The same is observed in real-world scenarios. For example, Gupta and Sarawagi [25] demonstrate how the inclusion of alternative extractions can significantly improve the overall recall of information extraction systems.

Movie

MID	Title	Gross
1	Avatar	700
2	Titanic	600

Figure 1.3: Incomplete representation of probabilistic data in a traditional database

Aggregations

In traditional databases, aggregations are useful in summarising multiple tuples – sometimes millions of them – into one single value. Common examples are MIN, MAX, COUNT, and SUM aggregations. Possible worlds semantics suggests that aggregations in probabilistic databases, on the other hand, can be understood as aggregations in each of the separate possible worlds, resulting in a probability distribution over the range of all possible values.

Example 1. *To find the total gross across all movies, one can use a* SUM *aggregate query:*

$$\texttt{SELECT SUM(}Gross\texttt{) AS } Result \texttt{ FROM } Movie$$

Applying the query to the traditional database in Figure 1.3 returns a single value

$$700 + 600 = 1300$$

On the other hand, applying the query to the probabilistic database in Figure 1.1 returns multiple possible values, each of which corresponds to the result from one or more possible worlds. The probability distribution is given by:

World	*Result*	*Probability*
W_1	*1000*	*0.08*
W_2	*1200*	*0.02*
W_3	*1300*	*0.4*
W_4 & W_5	*1500*	*0.1+0.32=0.42*
W_6	*1700*	*0.08*

The probabilistic result indicates that the most likely total gross is 1500*, though it is shown as almost equally likely to be* 1300*. The traditional database, in contrast, completely misses the most probable answer of* 1500*, which demonstrates how the retention of as many worlds as possible ultimately improves the accuracy of query result.*

Challenges

The challenges in probabilistic databases can be divided into three categories: semantics and representation, query evaluation, and user interfaces [10].

Semantics and Representation While possible worlds semantics provides a well-defined meaning to probabilistic databases, it is impractical to store each of the possible worlds separately, as the number of possible worlds is exponential to the number of tuples in the database; more succinct representation is therefore needed for probabilistic data. Simple representation systems include tuple-independent probabilistic databases, in which all tuples are assumed to be independent probabilistic events, and block-independent-disjoint databases (BID databases), in which mutual exclusiveness between tuples can be represented in addition to independence [44]. Another valuable tool is the probabilistic conditioned table (pc-table), which allows for the modelling of arbitrary correlations using proposition formulae over random variables to encode the correlations between individual tuples; such tables subsumes the simpler representations [23]. More recently, pc-tables have been extended into so-called probabilistic value-conditioned tables (pvc-tables), which were proposed as a means of succinctly representing the results of aggregate queries [19]. Further discussion on pc-tables and pvc-tables can be found in Chapter 2.

Query Evaluation Along with the ongoing work to define the query languages for probabilistic databases, efficient query evaluation is one of the most active areas of research in the field of probabilistic databases. Because probabilistic databases store multiple possibilities for individual tuple, they are often orders of magnitude larger their deterministic counterparts, making the efficiency of query evaluation one of the largest obstacles in managing probabilistic data. Additionally, while some queries $\mathcal{Q}$ have been proved to have polynomial data complexity, others can be #P-hard [44]. It has therefore recently become popular to tackle query evaluation for probabilistic databases through approximation techniques, such as the computation of probability bounds [38] or the use of Monte Carlo approximations [12].

User Interface Queries over probabilistic databases can return probability distributions with exponentially many possible answers. It is therefore often unnecessary, and sometimes even undesirable, to present the full distribution to users. Research on user interfaces has focused on methods to summarise and visualise the possible answers.

1.2 Motivation and Objectives

Motivation

The ubiquity of uncertain data has made probabilistic databases relevant in a wide range of real-world scenarios: information extraction [18], sensor data management [32], data cleaning [1], entity resolution [26], scientific data management [13], business intelligence and financial risk assessment [2], and even crime fighting [3]. While some of these applications will benefit from the inclusion of support for aggregations, which will allow users to analyse data from a whole new perspective, the support for aggregations is crucial in other situations such as decision support systems, as reflected in the TPC-H queries that involve aggregations in all 22 of them.

As shown in Example 1, an aggregate query over a probabilistic database returns a probability distribution instead of a single value. The size of the distribution is proportional to the number of tuples being aggregated for MIN, MAX, and COUNT aggregations, and it can be exponential for SUM aggregations. The fact that aggregations are often run on thousands, if not millions, of tuples, however, necessarily implies problems with both evaluation efficiency and user interface.

Evaluation Efficiency The fact that aggregations are often run on numerous tuples suggests that the speed of computation can have a far-reaching impact on the practicality of aggregations in probabilistic databases. It is therefore important for computations to be efficient and have low complexity to allow for better scalability.

User Interface Because only one of the possible values in a distribution corresponds to the ground truth, the sum of the probabilities for all possible values must add up to exactly one. This fact implies that there will be many possible values with very low probabilities. It is unlikely that the user would be interested in all of these individual values; thus, instead of bombarding the user with a plethora of low-quality answers, employing data summarisation can optimise the utility of query results.

Because of the above problems, earlier work on supporting aggregate queries in probabilistic databases has focused on the computation of the *expected value* instead of the full probability distribution. Expected value computations, however, can sometimes lead to an unintuitive understanding of query results, especially when the data is skewed. Furthermore, there are some situations when the computation of the expected value is insufficient: for example, an investor evaluating the potential profit of an investment will be concerned about the expected return as much as the tail probabilities, which are directly related to the risk of the investment. It was not until recently that Fink et al. [19] proposed a framework for exact evaluation (i.e. the computation of full probability distribution) on positive relational algebra queries with aggregate.

Objective

This dissertation aims to tackle the problem of evaluation efficiency and data summarisation of aggregate query evaluation over probabilistic databases by proposing two different types of approximations: histogram approximation and top-k approximation. Histogram approximation captures the overall impression of a probability distribution by grouping adjacent values into bins, thereby reducing the number of distinct answers and increasing the probability mass in each. Top-k approximation, on the other hand, retrieves only the most probable tuples, essentially filtering out low-quality answers and focusing only on the highest-quality results. In both cases, efficient algorithms exploiting the structure of the problem will be proposed to enhance the performance of aggregate query evaluation.

1.3 Contributions

Here we investigate the problem of approximating the results of aggregate queries in probabilistic databases. Approximation is essential for such queries: in addition to the hardness of probability computation already present for queries without aggregation [44], aggregate queries pose new challenges since their answer size can be exponential to the input database size (that is, each possible world of the input probabilistic database can lead to a different aggregate answer). We therefore consider here two effective complementary approaches to approximate the query results and thus keep their size within reasonable limits: histograms approximation and top-k approximation. Both approaches are natural and are used in standard relational databases and beyond to limit result size and computation time. Assuming a large probability distribution representing all query results, these techniques allow us to compute only a histogram representation of the distribution, or only the most probable values within the distribution. Arguably, these approaches present more intuitive results to users than very large, raw probability distributions. In both cases, the performance savings can be several orders of magnitude when compared to exact evaluation using the state-of-the-art algorithms, as demonstrated in the performance overview in Figure 1.4. The approximation approaches also demonstrate better scalability as all the algorithms in Figure 1.4 have a lower complexity than the state-of-the-art algorithms.

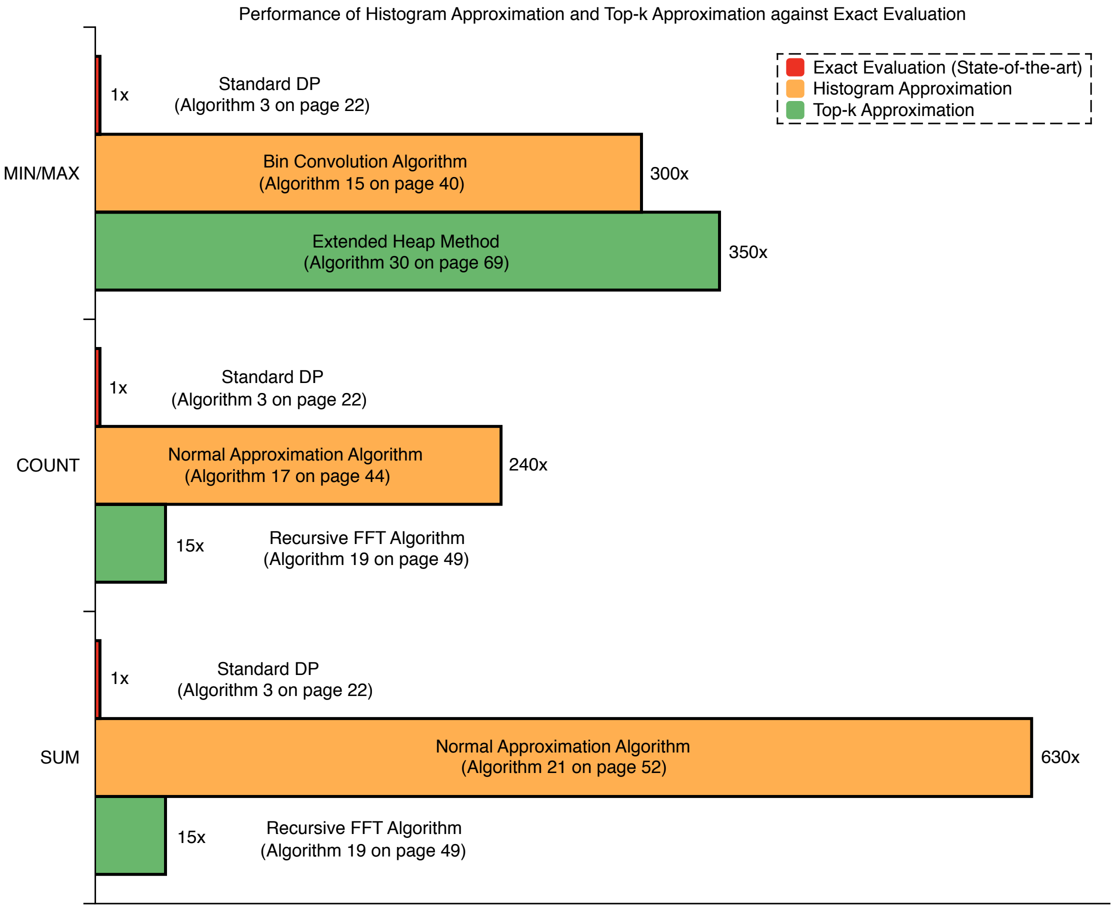


Figure 1.4: Performance of Histogram Approximation and Top-k Approximation. (Base on the average performance for aggregating 10,000 tuples)

In particular, our specific technical contributions are as follows:

- We leverage an existing knowledge compilation approach designed for exact evaluation on aggregate queries [19] and propose a framework for performing histogram approximation and top-k approximation on aggregate queries. Our approach to approximating complex aggregate queries essentially relies on evaluating a decomposition tree, which is a tree where the inner nodes are convolutions or Shannon expansions with respect to different aggregation monoids and the leaves are independent random variables with given finite probability distributions, under the two approximation settings.

- The proposed framework supports the computation of equi-width histograms, for which the user can specify either the number of bins or the width of the bins, as well as histograms with arbitrary bin intervals. Furthermore, the framework supports histogram zooming, in which the user can refine the resolution of any individual part of the histogram efficiently.

- The proposed framework supports ongoing retrieval of the next most probable tuple sequentially (i.e. the user does not have to decide k upfront).

- We devise a class of efficient algorithms for tackling the fundamental problem of MIN, MAX, COUNT, and SUM convolution of probability distributions under the two approximation settings. Convolution of probability distributions is fundamental in probability theory and appears in a diverse range of areas, such as physics [45], chemistry [45], biology [22], engineering [40], operational research [40], and finance [8]; the proposed algorithms are therefore applicable beyond the field of probabilistic databases.
- We devise a low complexity algorithm to handle Shannon expansion under the histogram approximation setting and observe that efficient computation of Shannon expansion for top-k approximation can be achieved by a slight modification of the instance-optimal Threshold Algorithm (TA) [16].
- By combining value-based approximations with probability-based approximations, we introduce the computation of histograms with approximate probabilities. The probability-based approximation is used for COUNT and SUM aggregations to get around their intrinsic NP-hard complexity for probability computation.
- A heap-based algorithm is proposed for exact evaluation on MIN and MAX aggregations. The algorithm is significantly faster and has lower complexity than the state-of-the-art algorithms.
- A Fast Fourier Transform (FFT)-based algorithm is proposed for exact evaluation on COUNT and SUM aggregations. The algorithm has lower complexity than the state-of-the-art algorithms.
- We develop the framework and implement the proposed algorithms. The algorithms are benchmarked thoroughly against the state-of-the-art algorithms for exact evaluation, and the behaviour of the algorithms under different situations is investigated. We also compare the different viable algorithms for the same aggregation and determine the preferred algorithm.

1.4 Related Work

While research into probabilistic databases dates back as far as the early 1980s [5, 21, 34], it has attracted an outbreak of interest in recent years, resulting in the development of well-known prototypes such as Trio [48], MystiQ [12] and MayBMS [28]. A recent book by Suciu, Olteanu, Ré, and Koch [44] provided a thorough treatment of cutting-edge research on probabilistic databases.

Among the plethora of researches into query evaluation for probabilistic databases [11, 12, 17, 17, 38], relatively few attempts have been made to support aggregations. Among those that did, it has been popular to tackle the problem in approximate settings due to its computational hardness. For example, Jampani et al. [29] proposed a Monte Carlo method: first the database is sampled, and then the query can be evaluated over the samples. This approach is very general and supports complicated queries; the result of such queries, however, is necessarily approximate and comes without strict error guarantees. Yang et al. [49] improved Monte Carlo sampling by considering aggregate constraints – that is, while individual records might be uncertain, there can still be global statistical constraints on the database instances. For example, while the gross of a movie contributed from each individual theatre is inherently uncertain, it might be known that the worldwide gross of the movie is between \$200M and \$300M, which serves to constrain the sampling.

It has also been popular to compute the expected value and other statistical measures of aggregate query results instead of the full probability distribution. Kennedy and Koch [31] proposed a Monte Carlo-based algorithm to estimate the expected values as well as other statistical measures for aggregations. Their approach includes the use of symbolic representations of probabilistic data computed alongside the query evaluation to defer Monte Carlo sampling, which allows goal-oriented sampling to increase the efficiency of the system. Jayram et al. [30] tackled a similar problem by estimating the expected value of aggregate queries, but with a focus on I/O efficiency by using data stream algorithms. Murthy et al. [36] described how aggregate queries were handled in the Trio system [48], which involves the computation of the exact expected value as well as the lowest and highest value of the resulting distribution.

Notably, the use of histogram approximation has gone relatively unexplored in the field of probabilistic databases. The only major work found in the literature is by Cormode and Garofalakis [9]: the authors grouped adjacent values in the data source into buckets, and optimal representative values for each bucket were then found using the proposed dynamic programming-based algorithm by minimising the distance between the actual values and the representative values. Query evaluation was then run over the reduced data representation, resulting in query results in approximate settings. Because errors in the data source are necessarily propagated during the evaluation stage, however, the accuracy of such query results is questionable at best. In contrast to their work, our evaluation technique acts on the complete distributions in the data source while building histograms during the evaluation stage. Thus, the resulting histograms are an exact representation of the probability distribution that would result if one were to perform an exact evaluation.

Ré et al. [39] proposed a top-k approximation approach for aggregate queries on probabilistic databases by running Monte Carlo simulations on each of the possible answers. In particular, by tightening the probability bounds on possible answers with more simulations, some answers can be excluded as soon as their upper bounds are lower than the lower bound of at least k possible answers. While results with higher accuracy can be achieved by running ever more simulations, they come at the cost of performance.

Finally, Soliman et al. [43] tackled a different top-k approximation problem by retrieving the most probable set of the k highest-scoring tuples based on a predefined scoring function. Their work stands in contrast to our work, in which we retrieve all possible tuples together with the k most probable aggregated values for each tuple. For example, a grouped aggregate (SQL query with the GROUP BY clause) will return one tuple for each group along with a probability distribution over the possible values in the aggregated column for each tuple. Their framework returns the most probable set of k tuples with the highest values in the aggregated column by considering the corresponding set in all possible worlds; in contrast, our framework retrieves all tuples with the k most probable values in the aggregated column for individual tuple.

1.5 Outline

- Chapter 2 introduces aggregate query evaluation in probabilistic databases using pvc-tables and defines the mathematical notions used in this dissertation, including monoids, semirings and semimodules.
- Chapter 3 introduces both approximation approaches and provides an overview of the underlying framework.
- Chapter 4 presents the details of the framework for histogram approximation as well as the class of algorithms for handling MIN, MAX, COUNT, and SUM aggregations efficiently.
- Chapter 5 is the top-k approximation counterpart of the previous chapter on histogram approximation.
- Chapter 6 discusses the details of our implementation.
- Chapter 7 benchmarks the proposed algorithms and discusses the experimental results.
- Chapter 8 concludes the dissertation and provides suggestions for future work.

Chapter 2

Preliminaries

2.1 Probabilistic Databases and Possible Worlds Semantics

A probabilistic database is one that can exist in any of several possible states such that each state corresponds to a traditional database conforming to the same schema. Out of the several possible states, only one of them matches the ground truth – in other words, the individual states are mutually exclusive events. The likelihood that any given state matches the ground truth is indicated by its associated probability, and the sum of all of these state probabilities must be equal to one. For example, the probabilistic relationship shown in Figure 1.1 can exist in any of the six possible states enumerated in Figure 1.2. Each state is also known as a *possible world*, an interpretation of probabilistic databases known as *possible worlds semantics*. Thus, a probabilistic database can also be viewed as a probability distribution over a series of deterministic database, and a traditional database as a special case of a probabilistic database: that case, of course, occurs when there is exactly one possible world with a state probability of one.

There are essentially two kinds of uncertainty in probabilistic databases: tuple-level uncertainty and attribute-level uncertainty. Tuple-level uncertainty occurs when it is uncertain whether an individual tuple belongs to a particular relation. For example, in a relation storing the winners of the Oscars since the establishment of the award in 1929, it might be uncertain whether a particular movie has won an Oscar when different sources provide contradictory information, in which case a probability could be attached to the tuple to represent that likelihood. On the other hand, attribute-level uncertainty occurs when there are multiple possible values for a particular attribute of a given tuple. The example we presented in Figure 1.1 is an attribute-level uncertain relation: the movie Avatar could have a total gross of \$400M, \$700M or \$900M, while the movie Titanic could have a gross of \$600M or \$800M. At any rate, because possible values can be treated as mutually exclusive tuples, any probabilistic database that can handle tuple-level uncertainty and represent mutual exclusiveness can also handle attribute-level uncertainty.

While evaluating a query $\mathcal{Q}$ on a deterministic database instance $\mathcal{D}$ returns a deterministic relation $\mathcal{Q}(\mathcal{D})$, possible worlds semantics suggests that evaluating the same query $\mathcal{Q}$ on a probabilistic database instance $\mathcal{D}'$ is equivalent to evaluating the query in each of the possible worlds $\mathcal{W}_i$, resulting in a probabilistic relation consisting of the possible world

$\mathcal{Q}(\mathcal{W}_i)$ with the original world probability $P(\mathcal{W}_i)$. This is known as *possible answer sets semantics*. Because it would be impractical, however, to present all possible worlds to the user, possible answer sets semantics is often used for views rather than queries. Query evaluation in probabilistic databases often employs *possible answers semantics*, where a set of pairs $(t, P(t))$ is returned, with t being a possible tuple in one or more of the possible worlds $\mathcal{Q}(\mathcal{W}_i)$, and $P(t)$ representing the *marginal probability* of the tuple t. Marginal probability (also known as *tuple confidence*) indicates the total probability of the tuple by considering all possible worlds. In other words, $P(t) = \sum_{t \in \mathcal{Q}(\mathcal{W}_i)} P(\mathcal{W}_i)$. The query results can then be presented in the form of a traditional relation, with an extra column indicating marginal probability.

2.2 Probabilistic Data Representation and PC-Tables

Probabilistic data can be stored by representing each possible world as a separate database instance, along with the associated probability for each. While this representation is complete, meaning that arbitrary correlations between tuples can be captured, it is not efficient: the number of possible worlds is exponential to the size of the database. For example, a tuple-level uncertain relation with 10,000 tuples will have a total of 2^{10000} possible worlds. It is obviously impractical to store all of those worlds separately; thus, having a more efficient method of representation is essential.

One succinct representation would simply store the possible tuples along with individual marginal probabilities. Such a representation would not be complete, however, as there would be no way to capture the correlations between tuples, effectively requiring that all tuples be independent. That is where probabilistic conditioned tables (pc-tables) become useful: they bridge the gap by providing a representation system that is complete and succinct at the same time [23]. A pc-table is similar to a traditional relation, but it has an extra column, Φ, storing the lineage of each tuple. Lineage is an annotation attached to the tuple in the form of a propositional formula composed of random variables with specified probability distributions. Each possible assignment of the random variables corresponds to a possible world, with the world probability equal to the probability of the assignment. The lineage can either evaluate to a Boolean, in which case the corresponding possible world consists of only the tuples having a *True* annotation (known as *set semantics*), or to an integer, in which case the annotation indicates the number of times the tuple exists in the possible world, thus supporting duplicates (known as *bag semantics*).

Example 2. *The table "Movie" presented in Figure 1.1 can be represented as a pc-table as follows:*

Movie

MID	*Title*	*Gross*	Φ
1	*Avatar*	*400*	$x = 1$
1	*Avatar*	*700*	$x = 2$
1	*Avatar*	*900*	$x = 3$
2	*Titanic*	*600*	$y = 1$
2	*Titanic*	*800*	$y = 2$

Variable

Variable	*Value*	Φ
x	*1*	*0.1*
x	*2*	*0.5*
x	*3*	*0.4*
y	*1*	*0.8*
y	*2*	*0.2*

We first note that because the lineages of the tuples evaluate to Booleans, this corresponds to set semantics. One possible assignment is $\{x \leftarrow 1, y \leftarrow 1\}$, *with probability*

$$P(x = 1) \times P(y = 1) = 0.1 \times 0.8 = 0.08$$

Keeping only tuples with a True *annotation for the assignment leads to the following possible world:*

$\{x \leftarrow 1, y \leftarrow 1\}$

MID	*Title*	*Gross*
1	*Avatar*	*400*
2	*Titanic*	*600*

Probability = 0.08

This corresponds to the possible world W_1 *in Figure 1.2, while the remaining five possible assignments to* x *and* y *result in the other five possible worlds.*

Another way of looking at the pc-table is to realise that the use of repeated random variables in multiple tuples introduces a correlation between them; thus, the individual tuples for Avatar *are correlated because of the variable* x*. In particular, tuples are mutually exclusive if the product of the annotations always evaluates to* False*, which is indeed the case for the tuples listed for* Avatar*:*

$$(x = 1) \wedge (x = 2) \wedge (x = 3) \implies False$$

On the other hand, because the tuples between Avatar *and* Titanic *do not share the same variables, they must be independent. This suggests that the table is attribute-level uncertain for the movies* Avatar *and* Titanic *over the attribute* Gross*.*

More complicated correlations between tuples can be represented by introducing more complicated lineages. In fact, pc-tables have been proved to be complete [23]; thus, any kind of correlation between the tuples can be represented by manipulating the lineages of the tuples.

Example 3. *Consider the same table "Movie" in Figure 1.1. More complicated correlations in the table can be introduced by manipulating the lineages for the tuples:*

Movie

MID	*Title*	*Gross*	Φ
1	*Avatar*	*400*	$x = 1$
1	*Avatar*	*700*	$x = 2$
1	*Avatar*	*900*	$x = 3$
2	*Titanic*	*600*	$(x = 1) \vee (x = 2)$
2	*Titanic*	*800*	$x = 3$

In this new table, there are positive correlations in addition to mutual exclusiveness. For example, the probability of Avatar *having a gross of $400M or $700M is positively correlated to the probability of* Titanic *having a gross of $600M. One possible scenario to explain this kind of correlation could be that the information for* Titanic *having a gross of $600M comes from the same source as that for* Avatar *having a gross of $400M or $700M.*

Query Evaluation

Because pc-tables are essentially traditional relations with an extra column for annotation, it is common to find probabilistic databases built on top of traditional databases in order to leverage the countless hours of research that has gone into traditional databases. Similarly, query evaluation for pc-tables can be achieved by rewriting a traditional query to take lineage into account. Fink et al. [19] devised a recursive algorithm for rewriting positive relational algebra queries without aggregates for pc-tables. The algorithm is depicted in Figure 2.1.

$$
\begin{aligned}
[\![R]\!] =\ & \text{select R.*, R.}\Phi\text{ from R} \\
[\![\delta_{B\leftarrow A}(Q)]\!] =\ & \text{select R.*, R.A as B, R.}\Phi\text{ as }\Phi\text{ from }\big([\![Q]\!]\big)\text{ R} \\
[\![\sigma_{A\theta B}(Q)]\!] =\ & \text{select R.*, R.}\Phi\cdot[A\theta B]\text{ as }\Phi\text{ from }\big([\![Q]\!]\big)\text{ R} \\
[\![\pi_{A_1,\ldots,A_n}(Q)]\!] =\ & \text{select R.A}_1,\ \ldots,\ \text{R.A}_n,\ \sum\big(R.\Phi\big)\text{ as }\Phi\text{ from} \\
& \big([\![Q]\!]\big)\text{ R group by R.A}_1,\ \ldots,\ \text{R.A}_n \\
[\![Q_1 \times Q_2]\!] =\ & \text{select R.*, S.*, R.}\Phi \wedge \text{S.}\Phi\text{ as }\Phi\text{ from }\big([\![Q_1]\!]\big)\text{ R, }\big([\![Q_2]\!]\big)\text{ S} \\
[\![Q_1 \bowtie_\psi Q_2]\!] =\ & \text{select R.*, S.*, R.}\Phi \wedge \text{S.}\Phi\text{ as }\Phi\text{ from }\big([\![Q_1]\!]\big)\text{ R, }\big([\![Q_2]\!]\big)\text{ S where }\psi \\
[\![Q_1 \cup Q_2]\!] =\ & \text{select R.*, }\sum\big(R.\Phi\big)\text{ as }\Phi\text{ from}\Big(\text{select * from} \\
& \big([\![Q_1]\!]\big)\text{ union all select * from }\big([\![Q_2]\!]\big)\Big)\text{ R group by R.*}
\end{aligned}
$$

Figure 2.1: Recursive Algorithm $[\![\cdot]\!]$ for rewriting a positive relational algebra query without aggregates $\mathcal{Q}$. We assume R.*, S.* do not select column Φ.

Example 4. *Let's consider the pc-tables "M" and "O", which store information about movies and Oscar winners respectively:*

M

mid	*title*	*country*	Φ
1	*Slumdog Millionaire*	*UK*	$x = 1$
1	*Slumdog Millionaire*	*India*	$x = 2$
2	*A Beautiful Mind*	*USA*	y
3	*Scary Movie*	*USA*	z

O

mid	*year*	Φ
2	*2005*	w
1	*2006*	u
1	*2007*	v

Now consider a query for retrieving the countries that have produced Oscar-winning films:

$$\pi_{country}(M \bowtie O)$$

By using the algorithm in Figure 2.1, the relational algebra query is rewritten to become:

```
SELECT
    $R.country$, $\sum(R.\Phi)$ AS $\Phi$
FROM
    (SELECT
        $M.*$, $O.*$, $M.\Phi \wedge O.\Phi$ AS $\Phi$
     FROM
        $M$, $S$
     WHERE
        $M.mid = O.mid$) $R$
GROUP BY
    $R.country$
```

The evaluation result is depicted in the following pc-table:

R

Country	Φ
UK	$(x = 1)(u + v)$
India	$(x = 2)(u + v)$
USA	yw

There is an intuitive meaning to the lineages in "R". For example, for the UK to have produced an Oscar-winning film, Slumdog Millionaire has to be a UK movie $(x = 1)$ and *the Oscar-winner in either 2006* or *2007* $(u + v)$.

Together with the table for the probability distributions of the variables, the lineages in the table R can be turned into tuple confidences. For example, given the following variable relation "V" [1]*:*

V

Variable	*Value*	*Probability*
x	*1*	*0.6*
x	*2*	*0.3*
y	*True*	*1.0*
z	*True*	*0.8*
u	*True*	*0.6*
v	*True*	*0.5*
w	*True*	*0.9*

Computing the probability of each of the lineages being True *using table "V" leads us to the final result with tuple confidences. For example, the tuple confidence of the answer* UK *is given by the probability that* $(x = 1)(u + v)$ *evaluates to be* True*, which has the probability*

[1] The sum of the probabilities for each variable must be equal to one. For cases where the sum is less than one, we assume the remaining mass attributes to the valuation of either *0* or *False*. For example, $P(u = False) = 1 - 0.6 = 0.4$

$$\begin{aligned} P((x=1)(u+v)=\top) &= P(x=1)\times P((u+v)=\top) \\ &= 0.6\times(1-P((u+v)=\bot)) \\ &= 0.6\times(1-P(u=\bot)\times P(v=\bot)) \\ &= 0.6\times(1-(1-0.6)\times(1-0.5)) \\ &= 0.48 \end{aligned}$$

Repeating the computation for all other tuples leads us to the final result of the query:

M

Country	Φ
UK	*0.48*
India	*0.24*
USA	*0.9*

2.3 Monoids, Semirings, and Semimodules

Because pc-tables allow probabilistic data to be stored with a size proportional to the number of possible tuples, they are efficient for handling queries without aggregates as the number of tuples returned is proportional to the product of the size of the database and the size of the query. Aggregate queries, however, can return an exponential number of tuples [33]; in such cases, pc-tables are no longer so efficient. In such cases, probabilistic value-conditioned tables (pvc-tables) come to the rescue, providing a way to represent the results of aggregate queries in polynomial space by employing the mathematical concepts of monoids, semirings and semimodules [19], which will be introduced in this section.

Definition 1. *A* monoid *is a set M with an operation $+_M : M \times M \to M$ and a neutral element $0_M \in M$ that satisfy the following axioms for all $m_1, m_2, m_3 \in M$:*

$$\begin{gathered} (m_1 +_M m_2) +_M m_3 = m_1 +_M (m_2 +_M m_3) \\ 0_M +_M m_1 = m_1 +_M 0_M = m_1 \end{gathered}$$

A monoid is commutative *if $m_1 +_M m_2 = m_2 +_M m_1$.*

Many aggregation operations can be described with monoids by choosing the appropriate set M, the operation $+_M$, and the neutral element 0_M. For example, MAX aggregation acts on integers $v_1, v_2, \ldots, v_n \in \mathbb{N}$ using the associative *max* operator

$$\max(v_1, v_2, \ldots, v_n) = v_1 +_{\max} v_2 + \ldots +_{\max} v_n$$

with the neutral element $-\infty$

$$\max(v, -\infty) = -\infty +_{\max} v = v +_{\max} -\infty = v$$

Together with the fact that the *max* operator is commutative, the MAX aggregation can be described by the commutative monoid $(\mathbb{N}^{\pm\infty}, \max, -\infty)$. Similarly, MIN aggregation can be described by the commutative monoid $(\mathbb{N}^{\pm\infty}, \min, \infty)$ and SUM aggregation can be described by the commutative monoid $(\mathbb{N}, +, 0)$. COUNT is a special case of SUM where the values being added are equal to 1.

Definition 2. *A* commutative semiring *is a set S together with operations $+_S, \cdot_S : S \times S \to S$ and neutral elements $0_S, 1_S \in S$ such that $(S, +_S, 0_S)$ and $(S, \cdot_S, 1_S)$ are commutative monoids and the following holds for all $s_1, s_2, s_3 \in S$:*

$$\begin{aligned} s_2 \cdot_S (s_2 +_S s_3) &= (s_1 \cdot_S s_2) +_S (s_1 \cdot_S s_3) \\ (s_1 + s_2) \cdot_S s_3 &= (s_1 \cdot_S s_3) +_S (s_2 \cdot_S s_3) \\ 0_S \cdot s_1 &= s_1 \cdot_S 0_S = 0_S \end{aligned}$$

Commutative semirings are the canonical algebraic structure for handling tuple annotations [24]. Tuple annotations for set semantics can be represented by commutative semirings over Booleans $(\mathbb{B}, \vee, False, \wedge, True)$. In this case, a semiring expression K would be a Boolean expression, such as x, $x \wedge (y \vee z)$ and $x \vee (u = 1)$, where x, y, and z are random variables over Booleans and u is a random variable over integers. On the other hand, the bag semantics can be represented by commutative semirings over integers $(\mathbb{N}_0, +, 0, \times, 1)$. In this case, a semiring expression K would be an algebraic expression, such as u, $u + 1$ or $u \times v$, where u and v are random variables over integers. Set semantics is used for aggregations in which duplicates have no impact on the result, such as MIN and MAX aggregations as $MAX(5, 5, 5, 1) = MAX(5, 1) = 5$, and bag semantics is used for SUM and COUNT aggregations, where duplicates can affect the aggregation results, such as $SUM(5, 5, 5, 1) = 16$ and $MAX(5, 1) = 5$.

Definition 3. *Let $(S, +_S, 0_S, \cdot_S, 1_S)$ be a commutative semiring. An S-*semimodule *M consists of a commutative monoid $(M, +_M, 0_M)$ and a binary operation $\otimes : S \times M \to M$ such that for all $s_1, s_2 \in S$ and $m_1, m_2 \in M$ we have*

$$\begin{aligned} s_1 \otimes (m_1 +_M m_2) &= s_1 \otimes m_1 +_M s_1 \otimes m_2 \\ (s_1 +_S s_2) \otimes m_1 &= s_1 \otimes m_1 +_M s_2 \otimes m_1 \\ (s_1 \cdot_S s_2) \otimes m_1 &= s_1 \otimes (s_2 \otimes m_1) \\ s_1 \otimes 0_M = 0_S \otimes m_1 &= 0_M \\ 1_S \otimes m_1 &= m_1. \end{aligned}$$

While monoids can be used to represent the values being aggregated and semirings can be used to represent their annotations, semimodules connect the two and provide a complete representation for aggregations over values conditioned on annotations. In particular, $\phi_1 \otimes v_1 +_M \phi_2 \otimes v_2 +_M \ldots +_M \phi_n \otimes v_n$ represents an aggregation with the operator $+_M$ over possible values $v_i, i = 1 \ldots n$, with ϕ_i conveying the probability of the existence of v_i (set semantics) or the number of occurrences of v_i (bag semantics).

2.4 Aggregations and PVC-Tables

Having introduced the notion of monoids, semirings and semimodules, we are now in a position to define pvc-tables for aggregate queries:

Definition 4. *Probabilistic value-conditioned tables (pvc-tables) are extension of pc-tables in which semimodule expressions are allowed to be stored in any attributes of a tuple other than values.*

Just as in pc-tables, pvc-tables can be built on top of traditional databases. Traditional aggregate queries can then be rewritten according to the algorithm presented in Figure 2.1, together with the two additional rules defined in Figure 2.2 [19].

$$
\begin{aligned}
[\![\varpi_{A_1,\ldots,A_n;\alpha_1\leftarrow \mathrm{AGG}_1(B_1),\ldots,\alpha_l\leftarrow \mathrm{AGG}_l(B_l)}(Q)]\!] = {} & \text{select R.A}_1,\ \ldots,\ \text{R.A}_n,\ \Gamma_1 \text{ as } \alpha_1,\ldots,\Gamma_l \text{ as } \alpha_l,\\
& \left[\left(\textstyle\sum_K \text{R.}\Phi\right) \neq 0_K\right] \text{ as } \Phi \text{ from } \left([\![Q]\!]\right) \text{ R group by R.A}_1,\ \ldots,\ \text{R.A}_n\\
[\![\varpi_{\emptyset;\alpha_1\leftarrow \mathrm{AGG}_1(B_1),\ldots,\alpha_l\leftarrow \mathrm{AGG}_l(B_l)}(Q)]\!] = {} & \text{select } \Gamma_1 \text{ as } \alpha_1,\ldots,\ \Gamma_l \text{ as } \alpha_l,\ 1_K \text{ as } \Phi \text{ from } \left([\![Q]\!]\right) \text{ R}\\
\text{where } \Gamma_i = {} & \begin{cases} \sum_{\mathrm{AGG}_i}\left(\text{R.}\Phi \otimes \text{R.B}_i\right) & \text{if } \mathrm{AGG}_i = \text{MIN,MAX,SUM,PROD}\\ \sum_{\mathrm{SUM}}\left(\text{R.}\Phi \otimes 1\right) & \text{if } \mathrm{AGG}_i = \text{COUNT} \end{cases}
\end{aligned}
$$

Figure 2.2: Recursive algorithm $[\![\cdot]\!]$ for rewriting a positive relational algebra query with aggregates $\mathcal{Q}$ to account for computation of semiring (K) and semimodule expressions.

To demonstrate how semimodule expressions and pvc-tables can be used for aggregate query evaluations, let's consider the following pc-tables "O" and "V" in Figure 2.3, which store information about Oscar winners and the probability distribution for the random variables respectively.

O

mid	title	country	viewers	Φ
1	Avatar	USA	50	x
2	Forrest Gump	USA	40	$y+z$
3	Gandhi	India	30	z
4	Harry Potter	UK	50	y
5	Slumdog Millionaire	UK	45	$x+z$
6	Titanic	USA	60	yz

V

variable	value	probability
x	*True*	1.0
y	*True*	0.4
z	*True*	0.6

Figure 2.3: PC-Table for the Oscars winners. (viewers in the unit of millions)

Full-Table Aggregates

We first consider an aggregate query for finding the maximum number of viewers of the Oscar winners:

$$\varpi_{\emptyset;result\leftarrow \text{MAX}(viewers)}(O)$$

This kind of query is known as a full-table aggregate since it involves aggregating values from individual tuples across the entire table. This kind of query corresponds to SQL aggregate queries without the GROUP BY clause:

$$\texttt{SELECT MAX(}viewers\texttt{) AS } result \texttt{ FROM } O$$

According to the query rewriting rules in Figure 2.2, the query is rewritten to handle the lineages as follows:

$$\texttt{SELECT } \textstyle\sum_{MAX}(O.\Phi \otimes O.viewers) \texttt{ AS } result\texttt{, } True \texttt{ AS } \Phi \texttt{ FROM } O$$

Running the rewritten query returns the following pvc-table:

R

result	Φ
$x \otimes 50 +_{\max} (y+z) \otimes 40 +_{\max} z \otimes 30$ $+_{\max} y \otimes 50 +_{\max} (x+z) \otimes 45 +_{\max} yz \otimes 60$	$True$

The semimodule expression encapsulates all of the information regarding the probability distribution over the possible values in the result of the aggregation. The final step would be to use the variable table V to turn the semimodule expression into a probability distribution, which will be explained in Section 2.5.

Grouped Aggregates

Now consider another aggregate query for finding the maximum number of viewers of the Oscar winners *grouped by country.* This query takes the form

$$\varpi_{country;result\leftarrow \text{MAX}(viewers)}(O)$$

Unlike full-table aggregates, grouped aggregates do not perform aggregations over all tuples in the table. Instead, they begin by grouping the tuples based on one or more attributes; aggregations are then performed separately over the tuples in each of the groups. Group aggregates correspond to SQL queries with the GROUP BY clause:

$$\texttt{SELECT } country\texttt{, MAX(}viewers\texttt{) AS } result \texttt{ FROM } O \texttt{ GROUP BY } country$$

The query can be rewritten according to Figure 2.2 to handle the lineage:

$$\texttt{SELECT } country\texttt{, } \textstyle\sum_{MAX}(\Phi \otimes viewers) \texttt{ AS } result\texttt{, } (\sum_{K} \Phi) \neq False \\ \texttt{FROM } O \texttt{ GROUP BY } country$$

Evaluating the rewritten query returns the following pvc-table:

O

country	result	Φ
USA	$x \otimes 50 +_{\max} (y+z) \otimes 40 +_{\max} yz \otimes 60$	$x + (y+z) + yz$
UK	$y \otimes 50 +_{\max} (x+z) \otimes 45$	$y + (x+y)$
India	$z \otimes 30$	z

While full-table aggregates return exactly one tuple with one semimodule expression per attribute, grouped aggregates return multiple tuples, with each tuple corresponding to a group; grouped aggregates therefore also return multiple semimodule expressions. In both cases, evaluating aggregate queries in probabilistic databases boils down to turning semimodule expressions into probability distribution over the possible values.

2.5 Decomposition Trees and Convolutions

While pvc-tables contain the query results in the form of semimodule expressions, these expressions have to be converted into probability distributions over the possible values before presenting them to users. One straight-forward approach is to substitute all possible assignments of the variables into the semimodule expressions. For example, consider the semimodule expression representing the maximum number of viewers for Oscar winners from the USA for the relation given in Figure 2.3:

$$x \otimes 50 +_{\max} (y+z) \otimes 40 +_{\max} yz \otimes 60$$

One possible assignment is $\{x \leftarrow \top, y \leftarrow \bot, z \leftarrow \top\}$, with probability

$$P(x = \top) \times P(y = \bot) \times P(z = \top) = 1.0 \times (1 - 0.4) \times 0.6 = 0.36$$

according to table "V" in Figure 2.3. Substitution into the semimodule expression gives

$$\begin{aligned}
& \top \otimes 50 +_{\max} (\bot \vee \top) \otimes 40 +_{\max} (\bot \wedge \top) \otimes 60 \\
&= \top \otimes 50 +_{\max} \top \otimes 40 +_{\max} \bot \otimes 60 \\
&= 50 +_{\max} 40 +_{\max} 0_M \\
&= (50 +_{\max} 40) +_{\max} 0_M && \text{(associativity)} \\
&= 50 +_{\max} -\infty && \text{(neutral element of MAX = -}\infty\text{)} \\
&= 50
\end{aligned}$$

Therefore one of the possible value for the aggregation is 50, with probability ≥ 0.36.[2] This brute-force approach, however, takes time exponential to the number of variables in the semimodule expression. For example, for a semimodule expression with 100 Boolean variables, there will be a total of 2^{100} possible enumerations.

[2] The probability can be larger than 0.36 as there could be other assignments that lead to the same value 50, in which case the probability adds up.

Fink et al. [19] proposed a more efficient method by recursively dividing the semimodule expression into two independent semimodule expressions, resulting in a tree-like structure known as the decomposition tree (d-tree). When it is impossible to further divide the semimodule expression into two independent parts – for example, when there is a repeating variable in every part of the expression – Shannon expansion can be employed to resolve the dependency by enumerating the possible substitutions for the repeating variable. Once the d-tree is formed, the probability distribution of the aggregation result can be obtained by recursively combining the random variables in the d-tree in a bottom-up manner. The full algorithm can be found in Algorithm 1.

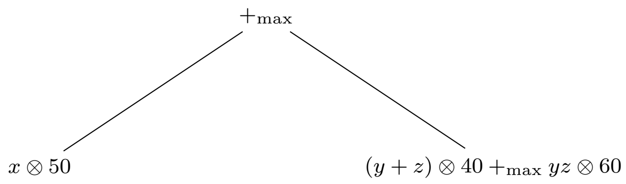
ALGORITHM 1: Compilation of a semimodule expression into a decomposition tree

```
Input: Semimodule expression $\Phi$
Output: Decomposition Tree of $\Phi$

COMPILE ($\Phi$)
begin
    if $\Phi$ has no variables then
        return $\Phi$
    end
    if $\exists$ independent $\Phi_1, \Phi_2$ s.t. $\Phi_1 + \Phi_2 = \Phi$ then
        return COMPILE($\Phi_1$) $\oplus$ COMPILE($\Phi_2$)
    end
    if $\exists$ independent $\Phi_1, \Phi_2$ s.t. $\Phi_1 \otimes \Phi_2 = \Phi$ then
        return COMPILE($\Phi_1$) $\otimes$ COMPILE($\Phi_2$)
    end
    if $\exists$ independent $\Phi_1, \Phi_2$ s.t. $\Phi_1 \cdot \Phi_2 = \Phi$ then
        return COMPILE($\Phi_1$) $\odot$ COMPILE($\Phi_2$)
    end

    Choose variable $x \in \mathbf{X}$ occurring in $\Phi$
    return $\bigsqcup_x$ ($\forall s \in S, P_x[s] \neq 0$: COMPILE($\Phi_s$))
end
```

To illustrate how this works, we continue our demonstration using the same semimodule expression:

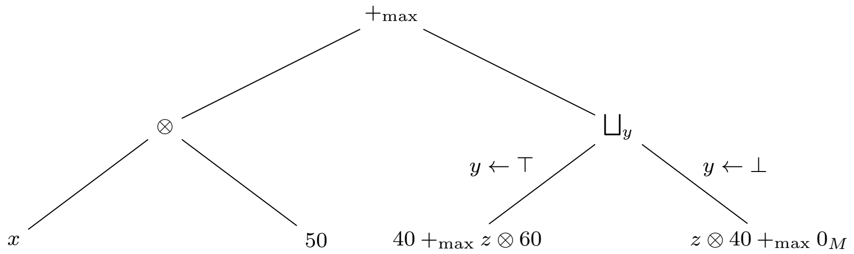
$$x \otimes 50 +_{\max} (y + z) \otimes 40 +_{\max} yz \otimes 60$$

Because the part $x \otimes 50$ is independent from the part $(y+z) \otimes 40 +_{\max} yz \otimes 60$, the first step involves dividing the expression into the two parts connected by the operator $+_{\max}$:

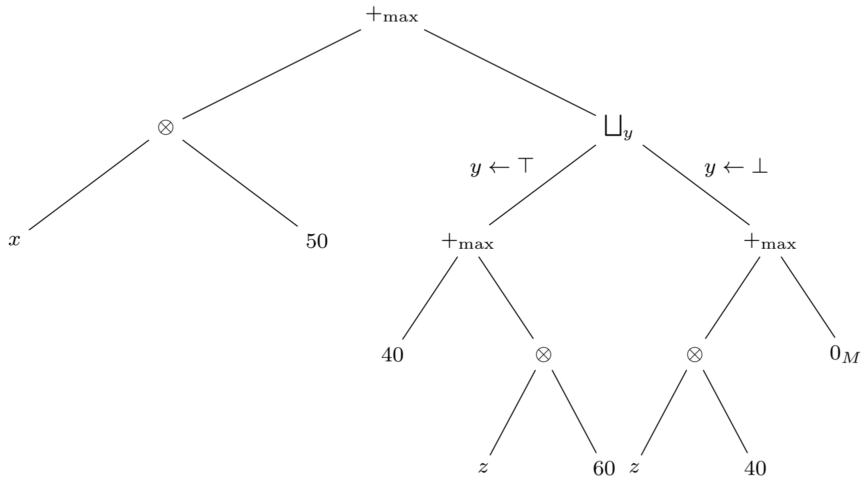


The left branch can be further divided into the semiring part and the monoid part, connected by the operator $\otimes$. At the same time, the right branch contains repeating variables y and

z, and there is no way of dividing the expression into two independent parts; thus, we will have to apply Shannon expansion on either y or z. On this occasion, we will apply Shannon expansion on the variable y:



At this point, all of the expressions in the leaves are free of repeating variables. We will expand them recursively until the d-tree is binary:



At this point, the decomposition tree is complete, and it is equivalent to the semimodule expression $x \otimes 50 +_{\max} (y + z) \otimes 40 +_{\max} yz \otimes 60$.

Exact Evaluation on a D-Tree

Once the d-tree is formed, the remaining step is to reduce the d-tree into a probability distribution. First of all, we note that all of the leaves in the d-tree are random variables.[3] Thus, a subtree at the bottom of the d-tree can be reduced into a probability distribution by combining the two random variables using the operator connecting them. The process can then be repeated until the entire d-tree is reduced to a probability distribution.

It is clear that the efficiency of d-tree evaluation hangs on the efficiency of combining probability distributions for different types of nodes. In fact, besides VARIABLE nodes, there are two types of nodes: UNION Nodes and CONVOLUTION Nodes. While UNION Nodes correspond to Shannon expansions, CONVOLUTION Nodes correspond to all other types of operators, including $+_{\max}, +_{\min}, +_{\text{sum}}, +_K, \times_K$, and $\otimes$.

[3] Constants can be treated as random variables with exactly one possible outcome.

For UNION nodes, all children – which are the probability distributions computed from the expanded expressions – can be combined using the algorithm presented in Algorithm 2 [19]. The algorithm has a complexity $\mathcal{O}(NM)$, where N is the number of children and M is the support size (the number of values with non-zero probability in the probability distribution) of the children.

ALGORITHM 2: Shannon expansion for a semimodule expression Φ on variable α

Input: Semimodule expression Φ, Expanding variable α
Output: Probability distribution of Φ

```
ShannonExpansion($\Phi$, $\alpha$)
begin
    $z \leftarrow \{\}$
    foreach $(v_\alpha, p_\alpha) \in \alpha$ do
        $x \leftarrow ExactEvaluation(\Phi|_{\alpha \leftarrow v})$
        foreach $(v_x, p_x) \in x$ do
            $z[v_x] \leftarrow z[v_x] + p_x \times p_\alpha$
        end
    end
    return $z$
end
```

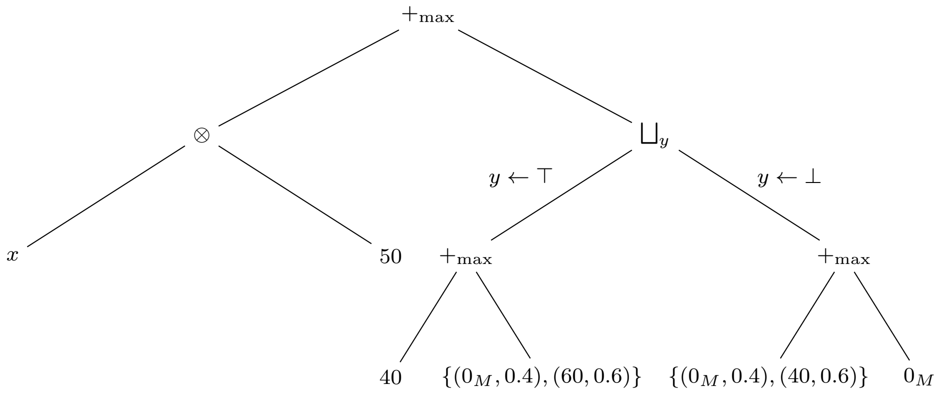
For CONVOLUTION nodes, Fink et al. [19] proposed an efficient dynamic programming-based algorithm for combining the probability distributions. This action of combining probability distributions over an operator is also known as the convolution of probability distributions, and it is fundamental in probability theory. The algorithm is presented in Algorithm 3, and will be referred as the Standard DP throughout the dissertation. The algorithm has a complexity $\mathcal{O}(M^2)$ for convolving two random variables with a support size M.

ALGORITHM 3: Standard DP for evaluating $x \oplus y$

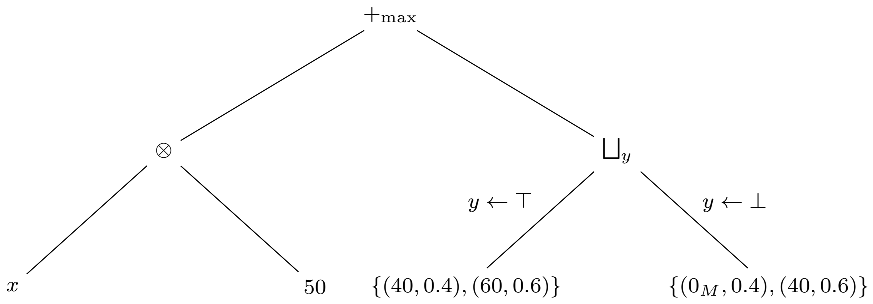
Input: Operator $\oplus$, random variables x and y
Output: Probability distribution of the convolution result $x \oplus y$

```
StandardDP($\oplus$, $x$, $y$)
begin
    $z \leftarrow \{\}$
    foreach $(v_x, p_x) \in x$ do
        foreach $(v_y, p_y) \in y$ do
            $z[v_x \oplus v_y] \leftarrow z[v_x \oplus v_y] + p_x \times p_y$
        end
    end
    return $z$
end
```

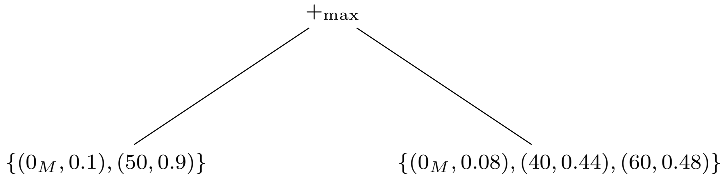
We continue our demonstration based on the d-tree we derived at the end of Section 2.5 to show how evaluation on a d-tree proceeds. We start with convolution of the random variables in the fourth level, resulting in a reduced d-tree:



Repeating the process by convolving the random variables in the third level leads us to a further reduced d-tree:



Continuing the process even further results in a d-tree with only two leaves:


$+_{\max}$
$\{(0_M, 0.1), (50, 0.9)\}$
$\{(0_M, 0.08), (40, 0.44), (60, 0.48)\}$

Lastly, convolution of the two probability distributions over the MAX operator leads to the final probability distribution:

$$\{(0_M, 0.008), (40, 0.044), (50, 0.468), (60, 0.48))\}$$

The result suggests that the most probable maximum number of viewers for Oscar winners from the USA is $60M$ and $50M$, with probabilities of 0.48 and 0.468, respectively. Note that the neutral element 0_M has a non-zero probability in the result, suggesting there is a chance that none of the movies from the USA exists in the original table O and therefore do not form a group in the result of the aggregate query at all. For this result, the neutral element 0_M will also be referred as *null*.

Chapter 3

Two Approximation Flavours: Histograms and Top-k

3.1 Overview

3.1.1 Workflow

The proposed framework for aggregate query evaluation consists of three steps:

Step 1. Traditional SQL aggregate queries are rewritten according to the rules in Figure 2.1 and Figure 2.2 to take into account the lineage of the tuples. The rewritten queries can then be evaluated over the probabilistic databases using a query engine similar to the ones used for traditional databases, resulting in a pvc-table that consists of a set of tuples whose values may be semimodule expressions. The exact form of the semimodule expressions depend on the aggregate operation and type of random variables in the input pvc-table.

Step 2. The semimodule expressions in the resulting pvc-table are compiled into decomposition trees (d-trees) using Algorithm 1 on page 20. As shown in prior work [19], d-trees allow computation of semimodule expressions in one pass over such d-trees.

Step 3. Our novelty comes in the computation of the probability distribution of a given d-tree (and hence of its equivalent semimodule expression). While we also provide an efficient algorithm for exact computation, our focus is on proposing two distinct and natural approximation approaches for this computation step. As shown in Chapter 7, these approaches can outperform the state-of-the-art best exact method by orders of magnitude.

3.1.2 Assumptions

We have made a few assumptions in the design of the framework:

Queries The framework supports positive relational algebra queries with aggregates.

Aggregate Values The values to be aggregated are assumed to be positive integers. Nonetheless, negative integers can be supported by shifting the domain of the values such that all of them are positive, and real numbers with finite decimal places can be handled by scaling the values to integers.

Null Value Null, also known as the neutral element, is represented by the integer 0 in the framework.

3.2 Value-based Approximations

Instead of retrieving the entire probability distribution, which can have very many distinct possible values, value-based approximation reduced the number of possible values by summarising and finding representatives for the possible values of the distribution.

3.2.1 Histogram Approximation

Histogram approximation summarises a probability distribution by grouping adjacent values into bins. This action not only reduces the number of distinct answers but also increases the probability, and hence the quality, of each answer. Histogram approximations are most suitable to situations when an overall impression of the distribution is needed, but not the exact probability of each possible answer. Figure 3.1 provides a comparison of the result of exact evaluation and histogram approximation for a MAX aggregate query. For most practical purposes, the histogram conveys as much information as the full distribution, but in a much more concise and intuitive way.

Additionally, the proposed framework supports histogram zooming, which allows the user to refine the resolution (i.e. increase the number of bins) of part of the histogram efficiently. This has far-reaching implications as it allows users to explore and interact with the distributions in a whole new level. In particular, the user can first instruct the framework to evaluate an aggregate query using equi-width histograms by specifying either the number of bins or the width of each bin for the histograms. The user can then discover interesting patterns or anomalies in the histograms returned, and zoom into the interesting parts to find out more about the pattern. In the extreme case, the user can even zoom into part of the histogram with a resolution of bin width = 1 unit. Most importantly, the efficiency of the algorithms for histogram approximation demonstrates that the computation of the histogram, together with multiple zooming, can still be orders of magnitude more efficient than the computation of the entire distribution.

The proposed algorithms are designed to be flexible and support the computation of the histogram with arbitrary number of bins having arbitrary intervals. In fact, the flexibility lays the foundation for the computation of equi-width histogram, histogram zooming, and range queries, which corresponds to feeding the algorithms with different bin intervals:

Equi-Width Histogram With the input for the number of bins B or the width of each bin W from the user, the framework can derive the bin intervals for the equi-width histogram by first computing the least value and the largest value of the probability distribution. For example, given a probability distribution with values in the range

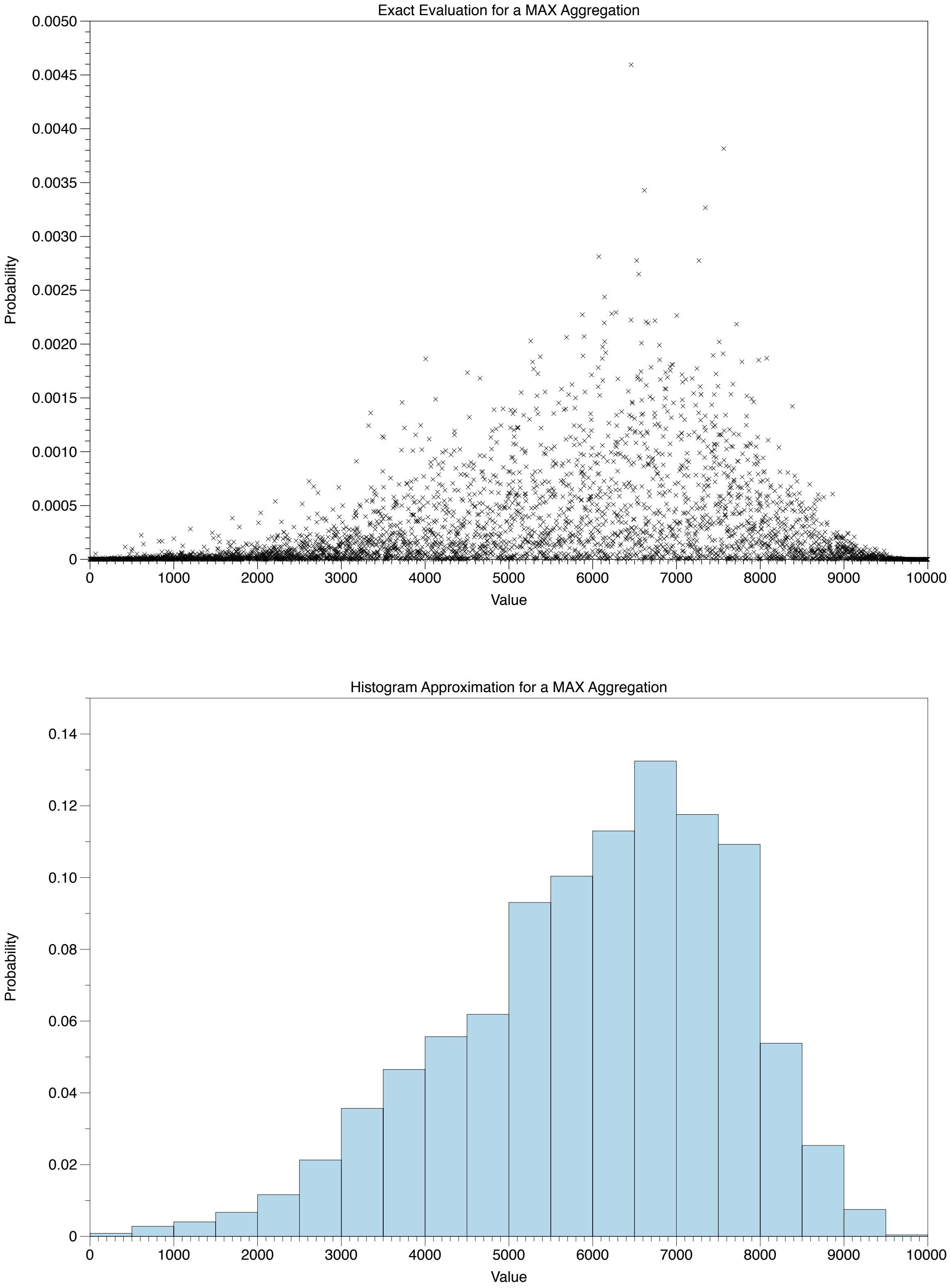


Figure 3.1: Comparison of exact evaluation and histogram approximation

$[1, 500]$, an equi-width histogram with $B = 5$ or $W = 100$ can be obtained by running the histogram approximation algorithms with bin intervals

$$[1, 100], [101, 200], [201, 300], [301, 400], [401, 500]$$

Histogram Zooming On top of the number of bins B or the width of each bin W, histogram zooming also requires the lower limit and the upper limit of the region to be zoomed from the user. For example, zooming into the region $[101, 150]$ with $B = 5$ or $W = 10$ for a histogram with values in the range $[1, 500]$ can be achieved by running the histogram approximation algorithms with bin intervals

$$[1, 100], [101, 110], [111, 120], [121, 130], [131, 140], [141, 150], [151, 500]$$

Note that the inclusion of the bins $[1, 100]$ and $[151, 500]$ ensures the bin intervals cover the entire distribution, which is needed for the algorithms to run properly. The two auxiliary bins can then be discarded from results before presenting to the user.

Answering Range Queries The framework for the evaluation of histogram approximations can also be used for answering range queries. For example, a movie investor might be interested in finding the probability that the total gross of horror movies in 2013 is over \$500M, in which case the query can be answered by computing the histogram with bin intervals

$$[L, 499], [500, U]$$

where L and U are the least and the largest value in the distribution respectively. The required probability will be equal to the bin probability for $[500, U]$.

3.2.2 Top-k Approximation

While histogram approximation provides a broad overview of the complete probability distribution, there are situations when the user is more concerned with the most probable answers.[1] Top-k approximation summarises the probability distribution by retrieving only the most probable answers, which are the ones with the highest quality.

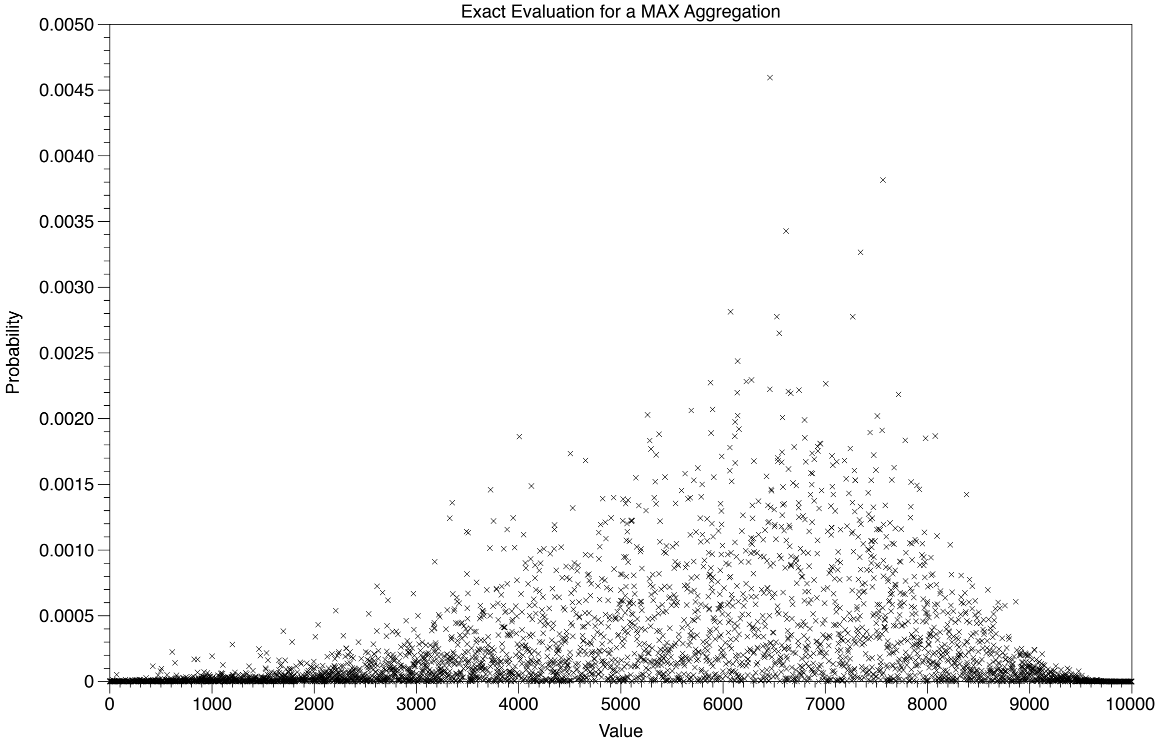
In particular, the top-k approximation framework we propose retrieves the answers sequentially in sorted order. The sequential nature of the algorithm implies the user does not have to decide k upfront, and can retrieve further tuples efficiently until enough information is found in the retrieved tuples.

Figure 3.2 compares the result of exact evaluation and top-k approximation with $k = 3$ for a MAX aggregation. It is clear that, while there are many more possible answers in the full distribution, most of them are of low quality and do not convey much information.

3.3 Probability-based Approximations

While histogram approximation summarises a probability distribution by grouping adjacent values into bins, the bin probability is exactly equal to the sum of probabilities for the

[1] *"It is a truth very certain that, when it is not in our power to determine what is true, we ought to follow what is most probable."* - René Descartes



Rank	Value	Probability
1	6459	0.00459487
2	7564	0.00381569
3	6617	0.00342772

Figure 3.2: Comparison of exact evaluation and top-k approximation

values between the lower and upper intervals of the corresponding bin. Similarly, while top-k approximation retrieves only the most probable values from a probability distribution, their probabilities are exactly equal to the corresponding probability in the probability distribution. Therefore, histogram approximation and top-k approximation are both value-based approximations.

However, it is not uncommon to find situations when exact probabilities are not crucial and a rough indication for the magnitudes of the probabilities is equally acceptable. Indeed, it has been suggested that relative probability between possible answers is much more important than absolute probabilities [44]. This opens up a whole new dimension of approximation: the computation of approximate probabilities with bounds. In particular, the approximate probability provides an estimate of the actual probability, and the lower and upper bounds provide a strict guarantee on the possible range for the probability. To ensure the framework can take advantage of both value-based approximations and probability-based approximations, we propose algorithms for the computation of histograms with approximate probabilities and top-k most probable values with approximate probabilities, which allows for an efficiency beyond the already promising performance of value-base approximation.

At any rate, we ensure the algorithms can fall back to pure value-based approximations when exact probabilities are crucial.

Figure 3.3 depicts an example of histogram with approximate probabilities from the evaluation of a COUNT aggregation: the height of each bar indicates the approximate probability, and the red error indicator marks the bound of the approximation. It is clear that as long as the bounds are tight in comparison of the bin probability, the introduction of probability-based approximation has little effect on the result.

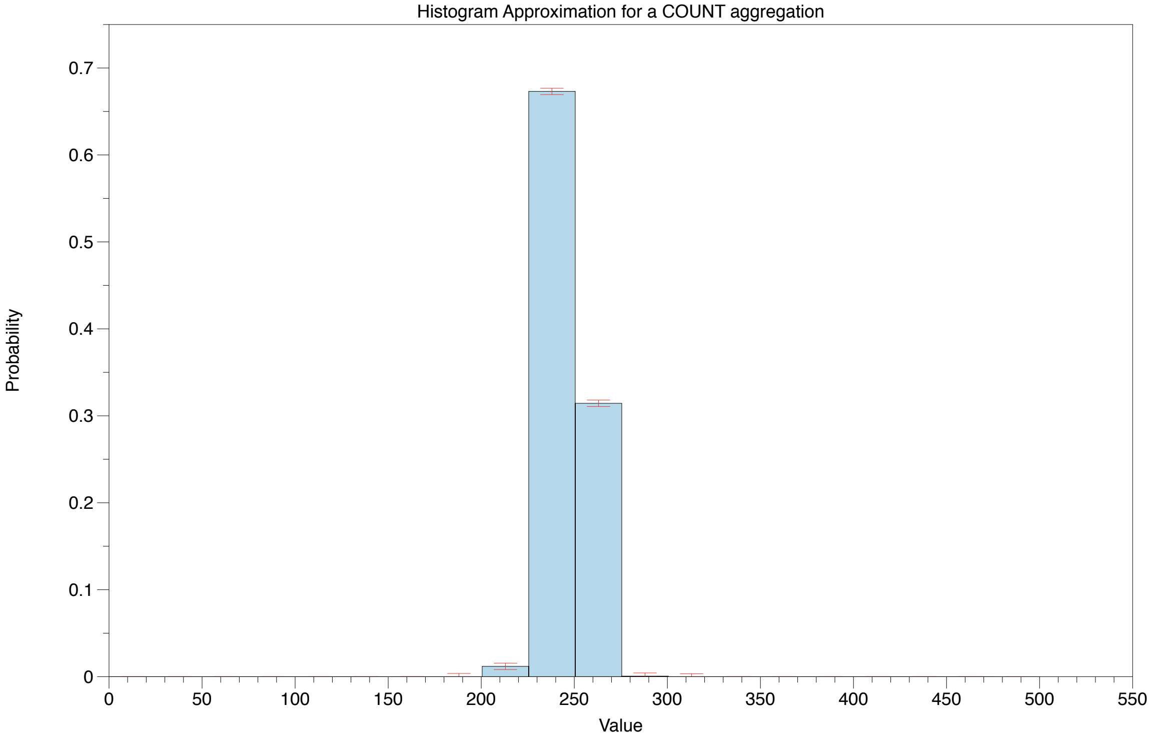


Figure 3.3: Example of a histogram with approximate probabilities

3.3.1 Working with Probability Bounds

The introduction of approximate probabilities calls for the need to perform arithmetic operations over probability bounds while preserving the strict guarantee of the bounds. For instance, the evaluation of a sub-query might return a histogram with bounded probabilities, and the main query must be able to process the bounded probabilities to return the final histogram. In particular, the algorithms proposed in the dissertation performs addition, subtract and multiplication over probabilities, therefore the support for the corresponding arithmetic operations over probability bounds is necessary.

Consider the arithmetic operator $\diamond$ and the probability bounds $[l_1, u_1]$ and $[l_2, u_2]$, the correctness for the probability bound $[l_1, u_1] \diamond [l_2, u_2]$ can be ensured by computing the lowest and highest values reachable for $p_1 \diamond p_2$, where $p_1 \in [l_1, u_1]$ and $p_2 \in [l_2, u_2]$. For example, adding the probability bound $[0.4, 0.6]$ to another probability bound $[0.1, 0.2]$ leads to the bound $[0.5, 0.8]$. Algorithms 4, 5 and 6 provide the detailed algorithms for addition, subtraction and multiplication over probability bounds respectively.

ALGORITHM 4: Add Probability Bounds

```
Input: Probability Bounds $left$ and $right$
Output: Probability Bound $left + right$

AddBoundedProb($left$, $right$)
begin
    $result.lower \leftarrow left.lower + right.lower$
    $result.upper \leftarrow left.upper + right.upper$
    return $result$
end
```

ALGORITHM 5: Subtract Probability Bounds

```
Input: Probability Bounds $left$ and $right$
Output: Probability Bound $left - right$

SubBoundedProb($left$, $right$)
begin
    $result.lower \leftarrow left.lower - right.upper$
    $result.upper \leftarrow left.upper - right.lower$
    return $result$
end
```

ALGORITHM 6: Multiply Probability Bounds

```
Input: Probability Bounds $left$ and $right$
Output: Probability Bound $left \times right$

MulBoundedProb($left$, $right$)
begin
    $result.lower \leftarrow left.lower \times right.lower$
    $result.upper \leftarrow left.upper \times right.upper$
    return $result$
end
```

Algorithms 4, 5 and 6 can easily be extended to add, subtract or multiply a probability bound with an exact probability: one simply treats the exact probability p as a probability bound $[p, p]$.

3.4 Optimisations

In this section, we present optimisations for the framework that will improve the performance of both histogram approximation and top-k approximation.

3.4.1 Tree Flattening

The associativity and commutativity of MIN, MAX, COUNT, and SUM suggests that if $+$ is one of the aggregate operators $+_{\text{MIN}}, +_{\text{MAX}}, +_{\text{COUNT}}, +_{\text{SUM}}$, then $((\alpha + \beta) + \gamma)$ is equivalent to $(\alpha + \beta + \gamma)$, where the order in which the operator acts on α, β and γ will not affect the final outcome. This suggests that two nodes of the same convolution type can be combined if they have a parent-child relationship in the d-tree, as depicted in Figure 3.4.

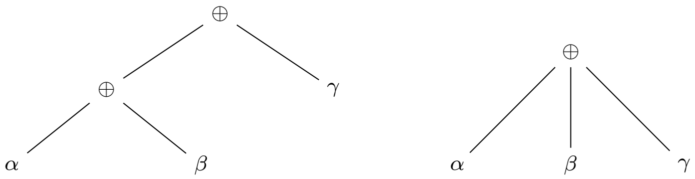


Figure 3.4: An example of tree flattening, where the two d-trees are equivalent.

Similar operation can also be performed on UNION nodes. However, one also has to take into account the variable assignment attached to each branch in this case. This is done by simply combining the variable assignments of the branches. An example is demonstrated in Figure 3.5.

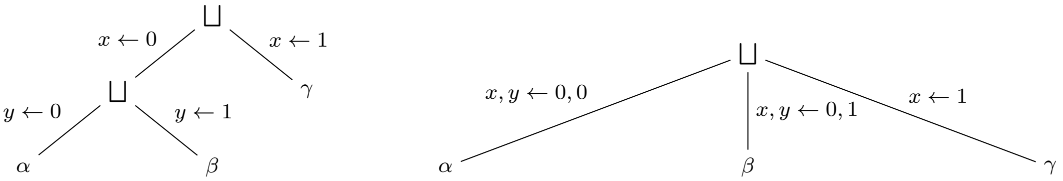


Figure 3.5: Tree flattening on UNION nodes

The algorithm for flattening a d-tree is presented in Algorithm 7. However, with careful bookkeeping, it is even possible to incorporate tree flattening into the tree compilation stage directly, thereby eliminating the need for the extra flattening step.

ALGORITHM 7: Flatten a decomposition tree

```
Input: Root Node node of Unflattened D-Tree dtree
Result: dtree is flattened in-place

FlattenTree(node)
begin
    foreach child ∈ children of node do
        FlattenTree(child)
        if type(node) = type(child) then
            foreach grandchild ∈ children of child do
                Add grandchild to the children of node
                if type(node) = UNION then
                    /* node.probs[child] stores the probability of the assignment from
                       node to child                                                    */
                    node.probs[grandchild] ← node.probs[child] × child.probs[grandchild]
                end
            end
            Remove child from the children of node
        end
    end
end
```

Tree flattening provides two benefits to the performance of the framework. Firstly, by minimising the height of decomposition trees, the number of recursive function calls needed during the evaluation stage are reduced, which in turn reduces the size of the recursive stack. More importantly, tree flattening exposes more information to a node by giving it direct access to more children, which allows the algorithms to optimise the evaluation by making use of the extra information.

Chapter 4

Histogram Approximation

4.1 Overview

In contrast to exact evaluation, where a complete probability distribution over the possible values of the aggregate query result is returned, histogram approximation evaluates the histogram representation of the probability distribution by grouping the possible values into bins. In other words, histogram approximation provides a synopsis of the probability distribution. In this chapter, we will present efficient algorithms for evaluating of MIN, MAX, COUNT, and SUM aggregate queries in this approximation setting.

4.1.1 Strategy

A straight forward approach for histogram approximation is to first perform an exact evaluation on the query to obtain the complete probability distribution, and the values in the distribution can then be put into the corresponding bin one by one. While this approach results in a histogram representation of the result, which is arguably more concise and intuitive than the complete distribution, the approach does not take advantage of the approximation setting to speed up the query evaluation.

For this reason, we will look into the special structure of histogram approximation, and devise algorithms that will fully take advantage of the approximation setting. More specifically, given a convolution node (or UNION node) $\oplus$ in a d-tree, and assume we have reduced the child subtrees of $\oplus$ into random variables $X_1, X_2, \ldots, X_N$, then the computation of histogram approximation with B bins $[a_1, b_1], [a_2, b_2], \ldots, [a_B, b_B]$ for the node $\oplus$ boils down to the computation of

$$\text{Probability of bin } [a_i, b_i] = P(a_i \leq X_1 \oplus X_2 \oplus \ldots \oplus X_N \leq b_i) \tag{4.1}$$

The ultimate goal is to compute Equation 4.1 without the full probability distributions for the random variables X_i, but with only their histogram representation, i.e. the probabilities $P(a_j \leq X_i \leq b_j)$, $j = 1 \ldots B$. This ensures the child subtree of $\oplus$ can be computed as efficiently as the node itself using the algorithms presented in this chapter, which maximises the performance of the approximation by allowing the efficient algorithms to act on all levels of the d-tree recursively.

4.1.2 Framework

The interface for histogram evaluation is presented in Algorithm 8, where each node type in d-trees must implement the interface. It supports the computation of histogram with arbitrary bin intervals, as long as bin intervals cover the entire distribution. When given a d-tree, the evaluation for histogram approximation can be initiated by calling the interface on the root node of the tree, and the evaluation request will then be propagated recursively through the interface to other nodes in the d-tree.

ALGORITHM 8: Histogram Approximation Interface

```
Input: A Decomposition Tree Node node, Bin Intervals bins
Output: Histogram Evaluation for Decomposition Tree node with bins
HistogramEvaluation(node, bins)
```

While the interface presented in Algorithm 8 provides the flexibility to evaluate histograms with arbitrary bins, it is often the case that the user might be unsure of what intervals to use. Algorithm 9 bridges the gap by computing equi-width histogram when the target bin width w or target number of bins B is given. (B can be turned into w via $w = \lfloor \frac{R-L}{B} \rfloor$, where L and R are the least and largest values in the distribution respectively).

ALGORITHM 9: Equi-Width Histogram Evaluation for a Decomposition Tree

```
Input: Root Node of Decomposition Tree root, Bin Width w
Output: Histogram Evaluation for Decomposition Tree root with bin width w
EquiWidthHistogramEvaluation(root, w)
begin
    $[L, R] \leftarrow GetDistributionRange(root)$
    $B \leftarrow \lfloor (R - L)/w \rfloor$
    $bins \leftarrow [L, L + w - 1], [L + w, L + 2w - 1], ..., [L + B \times w, R]$
    return $HistogramEvaluation(root, bins)$
end
```

Similarly, histogram zooming can be achieved by supplying the interface with the correct bin intervals, as depicted in Algorithm 10.

ALGORITHM 10: Histogram Zooming for a Decomposition Tree

```
Input: Root Node of Decomposition Tree root, Bin Width w, Zooming Boundaries l and r
Output: Zoomed Histogram with bin width w
ZoomHistogram(root, w, l, r)
begin
    $[L, R] \leftarrow GetDistributionRange(root)$
    $B \leftarrow \lfloor (r - l)/w \rfloor$
    $bins \leftarrow [L, l - 1], [l, l + w - 1], ..., [l + B \times w, r], [r + 1, R]$
    $histogram \leftarrow HistogramEvaluation(root, bins)$
    Remove bins $[L, l - 1]$ and $[r + 1, R]$ from histogram
    return histogram
end
```

4.2 Algorithms

Based on the discussion in the previous section, the evaluation of histogram approximation for d-trees requires two algorithms for each type of nodes:

GetDistributionRange(*node*) This returns the least and the largest values of the distribution obtained from the convolution of the *children of node* according to the *operator of node.*

HistogramEvaluation(*node*, *bins*) This returns the result of histogram approximation for the distribution obtained from the convolution the *children of node* according to the *operator of node.*

4.2.1 VARIABLE

VARIABLE nodes correspond to the random variables in semimodule expressions, where each random variable carries a complete probability distribution. Because the leaves of d-trees are always VARIABLE nodes, the algorithms for VARIABLE nodes form the base cases for the recursive computation of histogram approximation.

Distribution Range

Getting the distribution range of a VARIABLE node is as simple as finding the minimum and maximum of the probability distribution, as presented in Algorithm 11. The algorithm has a complexity of $\mathcal{O}(M)$, where M is the support size of the distribution. Additionally, because it is natural to store the probability distributions in sorted order by value in the database, the complexity would be $\mathcal{O}(1)$ in such cases.

ALGORITHM 11: Get Distribution Range for a VARIABLE Node

```
Input: VARIABLE Node node
Output: Distribution Range of node
DISTRIBUTION RANGE(node)
begin
    $least \leftarrow \infty$
    $largest \leftarrow -\infty$
    foreach $(v, p) \in$ probability distribution of node do
        $least \leftarrow \min(v, least)$
        $largest \leftarrow \max(v, largest)$
    end
    return $[least, largest]$
end
```

Histogram with Exact Probabilities

Because the entire probability distribution is already present in a VARIABLE node, histogram evaluation for a VARIABLE node simply involves going through the values in the distribution one by one and placing the values into the correct bins, as depicted in Algorithm 12. The algorithm has a complexity $\mathcal{O}(M \log B)$, where M is the size of the support of the probability distribution and B is the target number of bins to be computed.

ALGORITHM 12: Histogram Evaluation for a VARIABLE node

```
Input: VARIABLE Node node, Bin Intervals bins
Output: Histogram with bin intervals bins

HistogramEvaluation(node, bins)
begin
    histogram ← New Histogram with bin intervals bins
    Set probability of each bin in histogram to 0
    foreach $(v, p) \in$ probability distribution of node do
        $bin_{target}$ ← the bin in bins that $v$ belongs to        /* via Binary Search */
        Add $p$ to the bin probability of $bin_{target}$ in histogram
    end
end
return histogram
```

4.2.2 UNION

UNION node is created when Shannon expansion is performed on one or more variables for a semimodule expression, and each of the branches of a UNION node corresponds to one of the possible assignments for the expanding variables. Consider a UNION node Y with children $X_1, X_2, \ldots, X_N$, where the probability of the assignment for the branch connecting the UNION node Y and the child X_i is w_i, then the result of the computation on the UNION node is given by the probability distribution of Y, where

$$P(Y = v) = \sum_{i=1}^{N} w_i \times P(X_i = v) \tag{4.2}$$

This suggests exact evaluation on a UNION node (using Algorithm 2 on page 22) has a complexity $\mathcal{O}(NM)$, where M is the support size of the probability distribution for individual child X_i. Technically, if the support sizes of the probability distributions for $X_i, i = 1 \ldots N$ are different, M refers to the maximum one.

Distribution Range

According to Equation 4.2, the possible values for a UNION node Y directly come from the possible values in its children $X_i, i = 1 \ldots N$, hence

$$supp(Y) = \bigcup_{i=1}^{N} supp(X_i)$$

This indicates that the least and largest values for a UNION node must inherit from its children, resulting in Algorithm 13. The algorithm has a complexity $\mathcal{O}(N)$.

ALGORITHM 13: Get Distribution Range for a Union Node

Input: UNION Node *node*
Output: Distribution Range of *node*

```
GetDistributionRange(node)
begin
    $least \leftarrow \infty$
    $largest \leftarrow -\infty$
    foreach child ∈ children of node do
        $[L, R] \leftarrow GetDistributionRange(child)$
        $least \leftarrow \min(L, least)$
        $largest \leftarrow \max(R, largest)$
    end
    return $[least, largest]$
end
```

Histogram with Exact/Approximate Probabilities - Bin Union Algorithm (BUA)

Equation 4.2 can easily be extended to give

$$P(l \leq Y \leq r) = \sum_{i}^{N} w_i \times P(l \leq X_i \leq r) \tag{4.3}$$

Equation 4.3 suggests that histograms with the same bins can be combined in a similar way as complete distributions for a UNION node to obtain a histogram representation for Y. The idea is used in Algorithm 14, which has a complexity $\mathcal{O}(NB)$, where B is the target number of bins.

ALGORITHM 14: Histogram Evaluation for a UNION Node - Bin Union Algorithm (BUA)

Input: UNION Node *node*, Bin Intervals *bins*
Output: Histogram with bin intervals *bins*

```
HistogramEvaluation(node, bins)
begin
    histogram ← New Histogram with bin intervals bins
    Set probability of each bin in histogram to 0
    foreach child ∈ children of node do
        w ← Probability of the branch from node to child
        $histogram_{child} \leftarrow HistogramEvaluation(child, bins)$
        foreach bin in bins do
            /* histogram[bin] refers to the bin probability of bin in histogram */
            $histogram[bin] \leftarrow histogram[bin] + w \times histogram_{child}[bin]$
        end
    end
    return histogram
end
```

4.2.3 MIN/MAX

MIN and MAX nodes correspond to the operators $+_{\min}$ and $+_{\max}$ in semimodule expressions respectively. Because MIN and MAX are symmetric to each other, the algorithms developed for one of them can readily be adapted to work on the other. For this reason, we will only focus on MAX nodes in this section. Exact evaluation for a MIN/MAX node can be done via multiple applications of Standard DP (Algorithm 3 on page 22), with a complexity $\mathcal{O}((NM)^2)$, where N is the number of children for the MIN/MAX node and M is the support size of the probability distributions for its children.

Distribution Range

The algorithm we propose for getting distribution range for MAX nodes replies on the following proposition:

Proposition 1. *Given the random variables $X_1, X_2, \ldots X_N$ over the integers,*

$$supp(\mathrm{MAX}(X_1, X_2, \ldots, X_N)) = \bigcup_{i=1}^{N} supp(X_i) - \{x \in \mathbb{N} : x < \max_{i=1\ldots N}(\min(supp(X_i)))\}$$

Proof. Consider the situation when the random variable $X_i, i = 1 \ldots N$ takes the outcome v_i, in which case $\mathrm{MAX}(X_1, X_2, \ldots, X_N)$ becomes $\mathrm{MAX}(v_1, v_2, \ldots, v_N)$, returning exactly one value (the one with the maximum value) among the set $\{v_1, v_2, \ldots, v_N\}$. This demonstrates that the possible values in the result of $\mathrm{MAX}(X_1, X_2, \ldots, X_N)$ must come from the supports of the input variables, hence

$$supp(MAX(X_1, X_2, \ldots, X_N)) \subseteq \bigcup_{i=1}^{N} supp(X_i)$$

Next, consider the value u and the index m, where

$$u = \max_{i=1\ldots N}(\min(supp(X_i))), \quad m = \operatorname*{argmax}_{i=1\ldots N}(\min(supp(X_i)))$$

Now, for the convolution to evaluate to $v \in \bigcup_{i=1}^{N} supp(X_i)$, the outcomes taken by all the random variables $X_i, i = 1 \ldots N$ must be $\leq v$, which is always possible if $v \geq u$ as the minimum possible outcomes of all the random variables are $\leq u$. However, if $v < u$, then it is impossible for the random variable X_m to take an outcome $\leq v$ as its minimum possible outcome is u, suggesting

$$P(\mathrm{MAX}(X_1, X_2, \ldots, X_m, \ldots, X_N) < u) = 0$$

Therefore $supp(\mathrm{MAX}(X_1, X_2, \ldots, X_N))$ contains all the elements in the set $\bigcup_{i=1}^{N} supp(X_i)$, except for the elements from the set $\{x \in \mathbb{N} : x < \max_{i=1\ldots N}(\min(supp(X_i)))\}$. □

However, we also note that while the values in the set

$$\left(\bigcup_{i=1}^{N} supp(X_i)\right) \bigcap \left\{x \in \mathbb{N} : x < \max_{i=1\dots N}(\min(supp(X_i)))\right\}$$

have probability $= 0$ in the final distribution, those values are part of the active domain of the query as they are the possible values of the input random variables. For this reason, we will define the distribution of a MAX node as $\bigcup\limits_{i=1}^{N} supp(X_i)$, which has the same form as the distribution for a UNION node. Consequently the algorithm for getting the least and largest values in the distribution for a MAX node is identical to that for a UNION node (Algorithm 13 on page 37).

Histogram Approximation with Exact Probability - Bin Convolution Algorithm (BCA)

Proposition 2. *Given the independent random variables $X_1, X_2, \dots, X_N$ over the integers and an integer v,*

$$P(\text{MAX}(X_1, X_2, \dots, X_N) \leq v) = \prod_{i=1}^{N} P(X_i \leq v) \tag{4.4}$$

Proof. While the proposition can be proved algebraically, we will present a more intuitive proof by contradiction.

Firstly, consider each of the random variables X_i taking the outcome $u_i \leq v$, in which case $\text{MAX}(X_1, X_2, \dots, X_N) = \text{MAX}(u_1, u_2, \dots, u_N)$ returns exactly one value (the maximum one) from the set $\{u_1, u_2, \dots, u_N\}$, proving that the result must also be $\leq v$. This suggests that if each of the random variables X_i takes the outcome $\leq v$, the outcome of the MAX convolution must also be $\leq v$. In other words, the set of events that leads to $\text{MAX}(X_1, X_2, \dots, X_N) \leq v$ must consist of all the events where all the random variables X_i have outcomes $\leq v$. Hence,

$$P(\text{MAX}(X_1, X_2, \dots, X_N) \leq v) \geq \prod_{i=1}^{N} P(X_i \leq v) \tag{4.5}$$

To turn the inequality in Equation 4.5 into equality, we have to prove that it is impossible for $\text{MAX}(X_1, X_2, \dots, X_N) \leq v$ if one or more random variables have outcomes $> v$, therefore the events where all the random variables X_i have outcomes $\leq v$ form the complete set of events for $\text{MAX}(X_1, X_2, \dots, X_N) \leq v$. This can be proved by contradiction.

Let's assume that there exists an event, where $k > 0$ random variables have outcomes $\alpha_i > v$ for $i = 1 \dots k$, and the other $N-k$ random variables have outcome $\beta_j \leq v$ for $j = 1 \dots N-k$, which will lead to $\text{MAX}(X_1, X_2, \dots, X_N) \leq v$. However, because the MAX convolution will always return the maximum value among its input, $\text{MAX}(\alpha_1, \alpha_2, \dots, \alpha_k, \beta_1, \beta_2, \dots, \beta_{N-k}) = \text{MAX}(\alpha_1, \alpha_2, \dots, \alpha_k) > v$ will be returned, which contradicts to our assumption that $\text{MAX}(X_1, X_2, \dots, X_N) \leq v$. This proves that such event cannot exist, hence the events where $X_i \leq v$ for all $i = 1 \dots N$ form a complete set of events for $\text{MAX}(X_1, X_2, \dots, X_N) \leq v$. □

Based on Proposition 2, an algorithm for histogram evaluation for a MAX node is presented in Algorithm 15. The algorithm proceeds by computing the cumulative histogram for its children, and combining the histograms according to Equation 4.4. Lastly, the algorithm reverts the cumulative histogram back to a regular histogram. The algorithm has a complexity of $\mathcal{O}(NB)$, where B is the target number of bins.

ALGORITHM 15: Histogram Evaluation for a MAX Node - Bin Convolution Algorithm (BCA)

```
Input: MAX Node node, Bin Intervals bins
Output: Histogram with bin intervals bins

HistogramEvaluation(bins)
begin
    histogram ← New Histogram with bin interval bins
    Set bin probability of each bin in histogram to 1
    foreach child ∈ children of node do
        $histogram_{child} \leftarrow HistogramEvaluation(child, bins)$
        $p_{cumulative} \leftarrow 0$
        foreach bin ∈ bins do
            /* histogram[bin] is the probability of bin in histogram */
            $p_{cumulative} \leftarrow p_{cumulative} + histogram_{child}[bin]$
            $histogram[bin] \leftarrow histogram[bin] \times p_{cumulative}$    /* Equation 4.4 */
        end
    end

    /* Revert the cumulative histogram back to a regular histogram */
    B ← Number of bins in bins
    for $i = B$ to 2 do
        $histogram[bins[i]]$ -= $histogram[bins[i-1]]$
    end
    return histogram
end
```

As we have mentioned, the symmetric nature of MIN and MAX nodes suggests the propositions and algorithms we presented for MAX in this section can be adapted for MIN straightforwardly. In particular, the equivalent of Proposition 2 for MIN nodes is stated in Proposition 3.

Proposition 3. *Given the independent random variables $X_1, X_2, \ldots, X_N$ over the integers and an integer v,*

$$P(MIN(X_1, X_2, \ldots, X_N) > v) = \prod_{i=1}^{N} P(X_i > v)$$

4.2.4 COUNT

A COUNT node can be treated as a special case of a SUM node, where the input random variables have support $\{0, 1\}$. However, due to the popularity of COUNT aggregate queries, we will develop algorithms optimised for COUNT nodes. Exact evaluation for a COUNT node can be done via multiple applications of Standard DP (Algorithm 3 on page 22) with a complexity $\mathcal{O}(N^2)$, where N is the number of children for the COUNT node.

In particular, the computation for the distribution of a COUNT node Y can be defined by

$$Y = \sum_{i=1}^{N} X_i \tag{4.6}$$

where $X_1, X_2, \ldots, X_N$, representing the children of the COUNT node, are independent random variables with probability distributions

$$\begin{aligned} P(X_i = 1) &= p_i \\ P(X_i = 0) &= \bar{p_i} = 1 - p_i \end{aligned}$$

The distribution for Y is also known as Poisson Binomial distribution.

Distribution Range

According to Equation 4.6, the distribution Y ranges from 0 to N, therefore the algorithm presented in Algorithm 16 is surprising simple.

ALGORITHM 16: Get Distribution Range for a COUNT Node

```
Input: COUNT Node node
Output: Distribution Range of node

GetDistributionRange(node)
begin
    $N \leftarrow$ Number of children of node
    return $[0, N]$
end
```

Histogram Approximation with Approximate Probability - Normal Approximation Algorithm (NA)

Central Limit Theorem (CTL) states that the distribution obtained from the sum of N independent random variables can be approximated by normal distribution, provided that N is large enough. As it is often the case that one is trying to aggregate a huge number of tuples, this is highly relevant to the situation. However, central limit theorem does not provide a guarantee on the error for the probability approximated. Since the confidence level of the approximation would be a valuable information to the user, we look for a way to bound the error on the approximate probability, which lead us to the error bound derived via asymptotic expansions using Taylors formula by Neammanee [37]. The work is designed specifically for Poisson-Binomial distribution, therefore the bounds provided are tighter than other more general approaches.

Firstly, Neammanee [37] improves the central limit theorem by including two terms in the asymptotic expansions, as described in Proposition 4.

Proposition 4. *Define*

$$G(x) = \Phi(x) + \frac{1}{6\sqrt{2\pi}\sigma^3}(\sum_{i=1}^{N} p_i\bar{p_i}(p_i - \bar{p_i}))(1 - x^2)e^{-\frac{x^2}{2}} \tag{4.7}$$

where $\Phi(x)$ *is the standard normal distribution*

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int\limits_{-\infty}^{x} e^{-\frac{t^2}{2}} dt$$

and σ *is the standard deviation of the distribution* Y

$$\sigma^2 = \sum_{i=1}^{N} p_i(1 - p_i)$$

Then $P(a \leq Y \leq b)$ *can be approximated by*

$$G(\frac{b - \mu + 0.5}{\sigma}) - G(\frac{a - \mu - 0.5}{\sigma}) \tag{4.8}$$

Neammanee [37] then derived an error bound for the approximation in Equation 4.8. While the error bound can be derived for distributions with arbitrary value of σ^2, the bound gets worse as σ decreases, which is consistent to the central limit theorem that the distribution is only well approximated when the number of random variables N is large. For this reason, we will only compute histogram with approximate probabilities when $\sigma^2 \geq 25$, and fall back to the computation of histogram with exact probabilities when $\sigma^2 < 25$, using the efficient algorithms propose in the next section.

Proposition 5. *We define the absolute error between the exact probability and the approximate probability as* Δ*, where*

$$\Delta = |P(a \leq Y \leq b) - (G(\frac{b - \mu + 0.5}{\sigma}) - G(\frac{a - \mu - 0.5}{\sigma}))|$$

Neammanee bound states that

$$\Delta \leq \begin{cases} \frac{0.3056}{\sigma^2} & \text{if } 25 \leq \sigma^2 < 100 \\ \frac{0.1618}{\sigma^2} & \text{if } \sigma^2 \geq 100 \end{cases} \tag{4.9}$$

Neammanee bound is a uniform bound, which means the bound is independent of the values a and b. For this reason, the error is equally large in the central portion as the tails. However, because central portion contains most of the probability mass, and the probabilities at the tails are usually close to 0, the error in the tails is therefore relatively larger. In sight of this, we will introduce Chernoff bound [6], which provides an exponentially tighter bound as one moves away from the mean of the distribution. The Chernoff bound for Poisson Binomial distribution is presented in Proposition 6.

Proposition 6. *Chernoff bound dictates that*

$$\begin{aligned} P(Y \le \mu - \lambda) &\le e^{-\frac{\lambda^2}{2\mu}} \\ P(Y \ge \mu + \lambda) &\le e^{-\frac{\lambda^2}{2(\mu+\lambda/3)}} \end{aligned} \tag{4.10}$$

where μ is the expected value of the distribution Y.

Clearly, the Chernoff bound can be much tighter than Neammanee bound, but only in the tails. By using both the Neammanee bound the Chernoff bound, we can achieve a relatively tight uniform bound in the central portion of the distribution, and exponentially tighter bounds in the tails. The Normal Approximation Algorithm (NA) is presented in Algorithm 17. The Normal Approximation Algorithm involves the computation of the expected value and the variance of its children in the preprocessing step, which takes $\mathcal{O}(N)$. Computing the approximate probability takes $\mathcal{O}(1)$ per bin, therefore the computation of all the bin probability takes $\mathcal{O}(B)$. In general, $B \ll N$, therefore the Normal Approximation Algorithm has an overall complexity of $\mathcal{O}(N)$.

Histogram with Exact Probabilities - Deferred Binning Algorithm (DBA)

Sometimes the computation of histogram with approximate probabilities is not applicable, either because σ^2 is too small for accurate approximation, or when exact probability is crucial to the situation. For this reason, we will introduce an efficient algorithm for the evaluation of histogram with exact probabilities for COUNT nodes.

Firstly, we demonstrate that any algorithm for the computation of SUM convolution cannot possibly involve the computation of histograms as intermediate results. This is because combining histograms leads to a rapid increase in the bin width under SUM convolution, as demonstrated in Example 5.

Example 5. *Consider random variables X_1 and X_2, where the probability distributions are approximated by histograms H_1 and H_2 with bins*

H_1

Lower	*Upper*	*Probability*
1	*5*	*0.6*
6	*10*	*0.4*

H_2

Lower	*Upper*	*Probability*
1	*2*	*0.1*
3	*4*	*0.9*

To find the histogram representation for the distribution $Y = X_1 + X_2$, we combine the histograms according to the rule that summing two bin intervals can be done via $[L_1, R_1] + [L_2, R_2] = [L_1 + L_2, R_1 + R_2]$, therefore the histogram $H_1 + H_2$ is given by:

$H_1 + H_2$

Lower	*Upper*	*Probability*
2	*7*	*0.06*
7	*12*	*0.04*
4	*9*	*0.54*
9	*14*	*0.36*

ALGORITHM 17: Histogram Evaluation for a COUNT Node - Normal Approximation Algorithm (NA)

Input: COUNT Node *node*, Bin Intervals *bins*
Output: Histogram with approximate probabilities and bin intervals *bins*

```
HistogramEvaluation($node$, $bins$)
begin
    $histogram \leftarrow$ New Histogram with bin intervals $bins$

    for $i = 1$ to $N$ do
        $child \leftarrow i^{th}$ child of children of $node$
        $p_i \leftarrow 1.0 - GetNullProb(child)$
    end
    $\mu \leftarrow \sum_{i=1}^{N} p_i$
    $\sigma^2 \leftarrow \sum_{i=1}^{N} p_i \times (1 - p_i)$

    /* Neammanee Bound in Equation 4.9                                  */
    if $\sigma^2 < 25$ then
        Revert to evaluation of histogram with exact probabilities using other algorithms
    else if $\sigma^2 < 100$ then
        $\Delta \leftarrow 0.3056/\sigma^2$
    else
        $\Delta \leftarrow 0.1618/\sigma^2$
    end

    foreach $bin \in bins$ do
        $[a, b] \leftarrow bin$                         /* [a,b] is the boundaries of $bin$ */

        /* Computation of Approximate Probability                        */
        $prob \leftarrow G(\frac{1}{\sigma}(b - \mu + 0.5)) - G(\frac{1}{\sigma}(a - \mu - 0.5))$    /* G is defined in Equation 4.7 */

        /* Chernoff Bound in Equation 4.10                               */
        if $b < \mu$ then
            $\lambda \leftarrow \mu - b$
            $chernoff \leftarrow \exp(-\lambda^2/(2 \times \mu))$
        else if $a > \mu$ then
            $\lambda \leftarrow a - \mu$
            $chernoff \leftarrow \exp(-\lambda^2/(2 \times (\mu + \lambda/3)))$
        else
            $chernoff \leftarrow 1$
        end

        /* Computation of Lower and Upper Probability Bounds             */
        $lower \leftarrow max(prob - \Delta, 0)$
        $upper \leftarrow min(prob + \Delta, chernoff)$

        /* $histogram[bin]$ is the probability of $bin$ in $histogram$        */
        $histogram[bin] \leftarrow$ the approximate probability $prob$ with bound $[lower, upper]$
    end
    return $histogram$
end
```

However, bins with overlapping intervals is not valid in a histogram as there will be no way to tell the probability for the overlapping region, therefore bins with overlapping intervals have to be merged into one bin. For example, the bin $[2, 7]$ *and the bin* $[7, 12]$ *overlap in the region* $[7, 7]$*, therefore they have to be merged to give the bin* $[2, 12]$ *with probability* $0.06 + 0.04 = 0.1$*. However, the merged bin* $[2, 12]$ *also overlaps with the bins* $[4, 9]$ *and* $[9, 14]$*, suggesting that they all have to be merged, resulting in exactly one bin* $[2, 14]$ *with probability* 1*.*

Based on the observation in Example 5, we will derive an algorithm that does not involve the computation of histogram as intermediate result, but can still take advantage of the histogram approximation setting. In particular, the algorithm involves dividing the children of a COUNT node into two groups with equal sizes, and performs an exact evaluation using Standard DP (Algorithm 3 on page 22) over the two groups separately, resulting in two exact probability distributions. The distributions can then be combined to give histogram efficiently using Algorithm 18. Because Standard DP has a complexity of $\mathcal{O}(N^2)$ of COUNT convolution, splitting them into two groups and evaluating each of them exactly will have a complexity of $\mathcal{O}(2 \times (\frac{N}{2})^2) = \mathcal{O}(\frac{N^2}{2})$. Lastly, combining the two distributions according to Algorithm 18 will have a complexity $\mathcal{O}(NB)$, therefore the total complexity is given by $\mathcal{O}(N(\frac{N}{2} + B))$. In general, we have $B \ll N$, therefore $\mathcal{O}(N(\frac{N}{2} + B)) \to \mathcal{O}((\frac{N^2}{2}))$, suggesting the algorithm can be up to twice as fast as Standard DP.

ALGORITHM 18: Histogram Evaluation for a COUNT Node - Deferred Binning Algorithm (DBA)

```
Input: Count Node $node$, Bin Intervals $bins$
Output: Histogram with bin intervals $bins$

HistogramEvaluation($node$, $bins$)
begin
    $histogram \leftarrow$ New Histogram with bin intervals by $bins$
    Set the bin probability of all bins in $histogram$ to 0
    for $i = 1$ to $N$ do
        $child_i \leftarrow i^{th}$ child of children of $node$
    end
    $pdf_{left} \leftarrow$ Perform Exact Convolution for $\{child_i, i = 1 \ldots \frac{N}{2}\}$ using Standard DP
    $pdf_{right} \leftarrow$ Perform Exact Convolution for $\{child_i, i = \frac{N}{2} + 1 \ldots N\}$ using Standard DP
    foreach $(v, p)$ in $pdf_{left}$ do
        foreach $bin$ in $bins$ do
            $[a, b] \leftarrow$ the boundaries of $bin$
            Add $P(a - v \leq pdf_{right} \leq b - v)$ to $bin$ in $histogram$
                    /* via Binary Search on the cumulative distribution of $pdf_{right}$ */
        end
    end
    return $histogram$
end
```

Histogram with Exact Probabilities - Recursive Fast Fourier Transform (FFT)

Despite the Deferred Binning Algorithm (Algorithm 18) has exploited the approximation setting and provides up to two times speedup over exact evaluation using Standard DP, it has the same quadratic complexity as Standard DP, suggesting that Deferred Binning Algorithm might not be efficient enough when the number of values to be aggregated N gets larger. For this reason, we will present a Fast Fourier Transform (FFT) based algorithm that has a lower complexity.

We first note that if we represent the probability distribution of a random variable X_i by the polynomial (x is an auxiliary variable):

$$\sum_{v=0}^{R} P(X_i = v)x^v = P(X_i = 0) + P(X_i = 1)x + P(X_i = 2)x^2 + \ldots + P(X_i = R)x^R$$

Then the convolution result $X_1 + X_2$ will be equal to the multiplication of the polynomial representation of X_1 and X_2. Intuitively, this is because the convolution result

$$P(X_1 + X_2 = V) = \sum_{u=0}^{V} P(X_1 = u) \times P(X_2 = V - u)$$

matches the coefficient of the V^{th} order term in polynomial multiplication.

Example 6. *Consider two random variables X_1 and X_2, with probability distributions*

X_1	
Value	*Probability*
0	*0.5*
1	*0.3*
2	*0.2*

X_2	
Value	*Probability*
0	*0.1*
1	*0.5*
2	*0.4*

The polynomial representations of the two probability distributions are

$$X_1 : 0.5x^0 + 0.3x^1 + 0.2x^2$$
$$X_2 : 0.1x^0 + 0.5x^1 + 0.4x^2$$

The multiplication of the two probability distributions is given by

$$\begin{aligned}&(0.5x^0 + 0.3x^1 + 0.2x^2) \times (0.1x^0 + 0.5x^1 + 0.4x^2)\\ &= 0.05x^0 + 0.28x^1 + 0.37x^2 + 0.22x^3 + 0.08x^4 \qquad (4.11)\end{aligned}$$

The polynomial in Equation 4.11 can then be interpreted as the convolution result of X_1+X_2:

$X_1 + X_2$	
Value	*Probability*
0	*0.05*
1	*0.28*
2	*0.37*
3	*0.22*
4	*0.08*

It is well known that the multiplication of two polynomials with size R can be computed in $\mathcal{O}(R \log R)$ by using Fast Fourier Transform (FFT) [7], which implies convolution for two random variables $X_1 + X_2$ can also happen in $\mathcal{O}(R \log R)$, where R is the distribution range (the difference between the least and largest values of the distribution) of X_1 and X_2.

In our case, we need to convolve not only two distributions, but N of them:

$$Y = \sum_{i=1}^{N} X_i = X_1 + X_2 + \ldots + X_N$$

Because FFT can only be applied to convolve two random variables at one time, we have split the convolutions into two halves:

$$Z_1 = X_1 + X_2 + \ldots + X_{N/2}$$
$$Z_2 = X_{N/2+1} + X_{N/2+2} + \ldots + X_N$$

Therefore $Y = Z_1 + Z_2$, where FFT can be applied to convolve Z_1 and Z_2 efficiently. Clearly, before we can convolve Z_1 and Z_2, we have to evaluate the convolution for Z_1 and Z_2. However, as the convolution of Z_1 and Z_2 is an identical problem as the convolution of Y, but with smaller size, it can be evaluated in a similar fashion by divide (splitting the convolution into two halves) and conquer (convolving the two halves via FFT), resulting in a recursive algorithm. For example, consider the convolution for $X_1 + X_2 + \ldots + X_8$, the algorithm we propose will perform the computation according to the recursion tree in Figure 4.1, where each inner node corresponds to an FFT convolution.

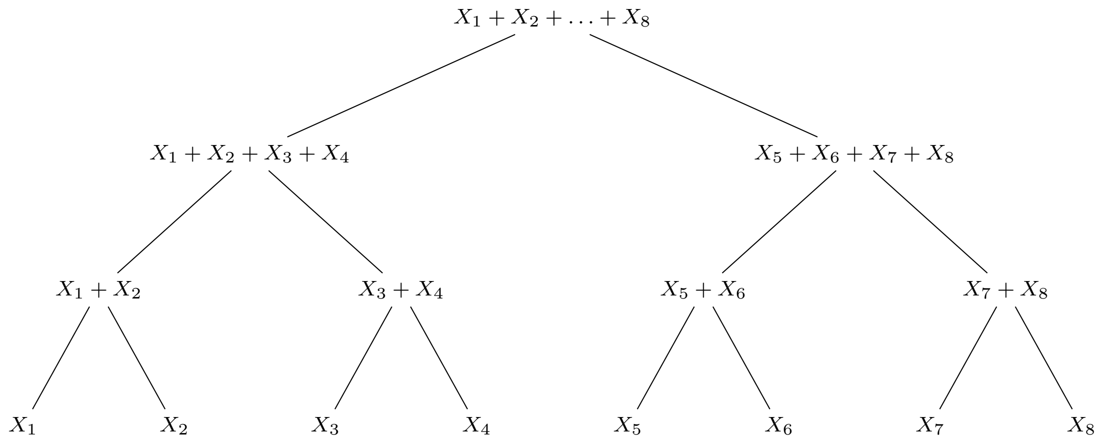


Figure 4.1: Recursion tree for the Recursive FFT Algorithm to evaluate $X_1 + X_2 + \ldots + X_8$

Additionally, the recursion tree in Figure 4.1 suggests the recursive algorithm will apply FFT convolution multiple times (7 times Figure 4.1), where each time the convolution is over random variables with distribution of different ranges. For example, imagine all the random variables in Figure 4.1 have a range R, then $X_1 + X_2$ involves a convolution of variables with range R, while $(X_1 + X_2 + X_3 + X_4) + (X_5 + X_6 + X_7 + X_8)$ involves a convolution of variables with range $4R$. While FFT convolution has a lower complexity than Standard

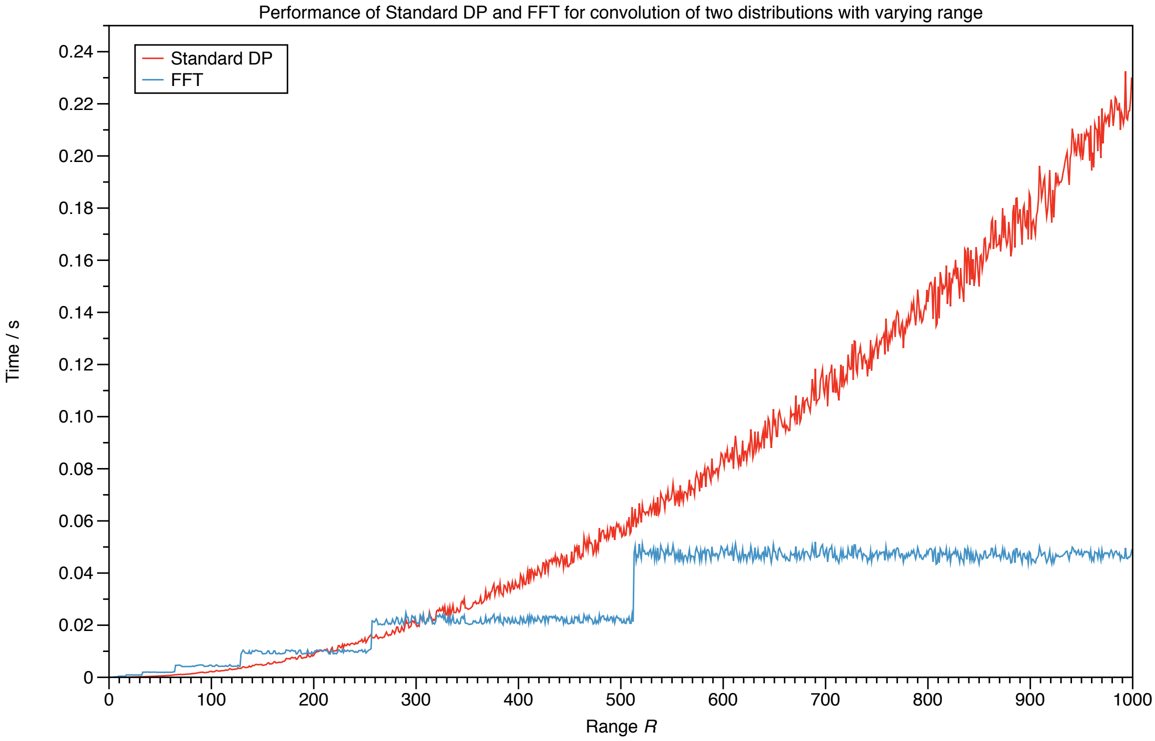


Figure 4.2: Performance of Standard DP and FFT for convolution of two random variables

DP (Algorithm 3 on page 22), FFT convolution involves more overhead, therefore it might be less efficient than Standard DP when the range for the two distributions to be convolved is small. It is clear in Figure 4.1 that the recursive algorithm performs many convolutions of random variables of small range (those at the bottom of the recursion tree), therefore it is important for the recursive algorithm to revert to Standard DP when convolving random variables at the bottom of the tree. In particular, Figure 4.2 compares the runtime of DP and FFT convolutions of two distributions with different ranges R, which suggests that Standard DP performs better when $N < 300$.[1] The recursive algorithm that combines FFT and Standard DP for exact evaluation is presented in Algorithm 19. The algorithm has a complexity $\mathcal{O}(N(\log N)^2)$, where N is the number of children of the COUNT node.

Despite its lower complexity in comparison to the Deferred Binning Algorithm (Algorithm 18 on page 45), the Recursive FFT Algorithm can only evaluates the full distribution for the COUNT node but not take advantage of the histogram approximation setting, therefore it is unclear which of the two algorithms will be better under different circumstances.[2] Their performances will be benchmarked and investigated in Chapter 7.

[1] There exists different variants of implementation for FFT convolution, and the one we used is the Cooley-Tukey algorithm [7]. The algorithm requires the range of the two input distributions to be a power of 2, therefore the algorithm needs to pad the distributions with 0 so as to make the size of the distributions a power of 2, making the runtime a constant in region between adjacent powers of 2.

[2] We note that, however, after obtaining the histogram representation of the distribution using Algorithm 12 on page 36, time saving due to the approximation setting can happen in other nodes in the d-tree.

ALGORITHM 19: Exact Evaluation for COUNT/SUM Node - Recursive FFT Algorithm (FFT)

```
Input: COUNT/SUM Node node
Output: The Probability Distribution of the Evaluation Result

ExactEvaluation(node)
begin
    /* Getting the random variables for the child subtrees of node */
    for $i = 1$ to $N$ do
        $child \leftarrow i^{th}$ child of children of node
        $X_i \leftarrow ExactEvaluation(child)$
    end
    return $FFTRecursive(\{X_1, X_2, \dots, X_N\})$
end

FFTRecursive($\boldsymbol{A}$)
begin
    if Size of $\boldsymbol{A}$ = 1 then
        return the only element in $\boldsymbol{A}$
    end

    $Y_{left} \leftarrow FFTRecursive$(First half of $\boldsymbol{A}$)
    $Y_{right} \leftarrow FFTRecursive$(Second half of $\boldsymbol{A}$)
    if Size of $Y_{left} < 300$ then
        return $ConvolveByStandardDP(Y_{left}, Y_{right})$    /* See Algorithm 3 on page 22 */
    else
        return $ConvolveByFFT(Y_{left}, Y_{right})$    /* See Cooley and Tukey [7] */
    end
end
```

4.2.5 SUM

For a SUM node Y with N children, which can be represented by the independent random variables $X_1, X_2, \dots, X_N$, the computation for the SUM node can be defined as the SUM convolution of the random variables:

$$Y = \sum_{i=1}^{N} X_i \tag{4.12}$$

While the random variables $X_1, X_2, \dots, X_N$ are independent, they can be non-identically and arbitrarily distributed as long as the distribution is finite. SUM nodes have the most expensive computation compared to MIN, MAX, and COUNT nodes as up to an exponential number of possible values can be returned. Exact Evaluation for a SUM node can be done via multiple applications of Standard DP (Algorithm 3 on page 22), with a complexity $\mathcal{O}((NR)^2)$, where R is the range of the distributions.

Distribution Range

According to Equation 4.12, the least and largest possible values in Y come from the least and largest possible values in the random variables $X_i, i = 1 \dots N$, therefore the distribution range for a SUM node can be retrieved by Algorithm 20.

ALGORITHM 20: Get Distribution Range for a SUM Node

Input: COUNT Node *node*
Output: Distribution Range of *node*

```
GetDistributionRange($node$)
begin
    $least \leftarrow \infty$
    $largest \leftarrow -\infty$
    foreach $child \in children\ of\ node$ do
        $[L, R] \leftarrow GetDistributionRange(child)$
        $least \leftarrow least + L$
        $largest \leftarrow largest + R$
    end
    return $[least, largest]$
end
```

Histogram with Approximate Probabilities - Normal Approximation Algorithm (NA)

Since a SUM node is the general version of a COUNT node, the Central Limit Theorem (CTL) is also applicable here. However, the Neammanee bound (Equation 4.9) is only applicable when the random variables have the support $\{0, 1\}$, therefore we need a more general version of it for SUM nodes. This leads us to the Berry-Esseen theorem [4, 14], where it provides an error bound for using normal approximation on Y, as long as the random variables X_i are independent to each other. Berry-Essent theorem is presented in Proposition 7, where the lower bound for C is derived by Esseen [15]. The upper bound for C has been subsequently lowered since the original estimate at 7.59 [14], and the best estimate in the literature is 0.5600 due to Shevtsova [42].

Proposition 7. *For Equation 4.12, Berry-Esseen theorem states that*

$$|P(Y < x) - \Phi(\frac{x-\mu}{\sigma})| \leq C \cdot \frac{\sum_{i=1}^{N} \rho_i}{(\sum_{i=1}^{N} \sigma_i^2)^{3/2}} \tag{4.13}$$

where ρ_i and σ are the absolute third moment and standard deviation of X_i respectively, and

$$C \geq \frac{\sqrt{10}+3}{6\sqrt{2\pi}} = 0.40973218\ldots$$

Similarly, while Chernoff bound (Equation 4.10) serves well for COUNT nodes to provide an exponentially tighter bounds in the tails of the distribution, it is only applicable when the input variables have support $\{0, 1\}$. Hoeffding's inequality [27] is the general version of Chernoff bound, where it lifts the constraint that the input variables need to have the support $\{0, 1\}$, and provides a bound even when the input variables have arbitrary distribution, which makes it the suitable for a SUM node. Hoeffding's inequality is presented in Proposition 8.

Proposition 8. *With Equation 4.12 and*

$$\mu = \sum_{i=1}^{N} p_i$$

Hoeffding's inequality states that

$$\begin{aligned} P(Y \leq \mu - \lambda) &\leq 2e^{-\frac{2\lambda^2}{\sum_{i=1}^{N}(b_i - a_i)^2}} \\ P(Y \geq \mu + \lambda) &\leq e^{-\frac{2\lambda^2}{\sum_{i=1}^{N}(b_i - a_i)^2}} \end{aligned} \tag{4.14}$$

By combining Berry-Esseen bound and Hoeffding bound, we arrive at the Normal Approximation Algorithm (NA) for the evaluation of histogram with approximate probabilities for SUM node in Algorithm 21. Similar to its counterpart for COUNT node (Algorithm 17), it has a complexity of $\mathcal{O}(N)$. We also note that the algorithm calls for the computation of mean, variance and absolute third moment for the child subtrees, where efficient algorithms for this purpose will be derived in the Appendix A.

Histogram with Exact Probabilities - Early Binning Algorithm (EBA)

While the Normal Approximation algorithm (Algorithm 21) provides an efficient approach for the computation of histogram with approximate probabilities for SUM nodes, there are situations when it is crucial for the bin probability to be exact. In this section, we will discuss some efficient approaches for the computation of histogram with exact probabilities for a SUM node.

As we have demonstrated in Example 5, summing random variables with histogram representations will lead to a rapid increase in the bin width, making the resulting histogram completely useless. This implies the computation cannot be proceeded when only the histogram representations of the SUM node's children are given, therefore the complete probability distributions of its children have to be retrieved. This suggests that to compute histogram with exact probabilities for a SUM node, the subtrees below the SUM node will have to be evaluated exactly, making SUM node a blocking operator. Example 5 also suggests that any algorithm for the computation cannot possibly involves histograms as intermediate result, which provides a guideline when devising efficient algorithms for the computation.

Firstly, we note that the Deferred Binning Algorithm for COUNT node (Algorithm 18 on page 45), where we divide the children into two groups and perform exact evaluation on them separately before combining the probability distributions in the form of a histogram efficiently, will also be applicable for SUM nodes with minor changes to the algorithm. The similarity between SUM and COUNT node suggests that there will be around 2 times performance speedup using this approach.

Additionally, we will propose an algorithm that is optimised for histogram zooming for SUM aggregate queries, based on the idea that when using Standard DP repeatedly to convolve random variables one by one, it is sometimes possible to move values in the intermediate distribution to the correct bins earlier, therefore reducing the size of intermediate distribution and saving sequential computation time.

ALGORITHM 21: Histogram Evaluation for a SUM Node - Normal Approximation Algorithm (NA)

Input: Sum Node *node*, Bin Intervals *bins*
Output: Histogram with bounded probabilities and bin intervals *bins*

```
HistogramEvaluation($node$, $bins$)
begin
    $histogram \leftarrow$ New Histogram with $bins$

    $\mu, \sigma^2, \rho, \kappa \leftarrow 0$
    for $i = 1$ to $N$ do
        $child \leftarrow i^{th}$ child of $node$

        $\mu \leftarrow \mu + GetMean(child)$
        $\sigma^2 \leftarrow \sigma^2 + GetVariance(child)$
        $\rho \leftarrow \rho + GetThirdMoment(child)$

        /* $\kappa$ is used in the computation of Hoeffding Bound */
        $[L, R] \leftarrow GetDistributionRange(child)$
        $\kappa \leftarrow \kappa + (R - L)^2$
    end

    /* Berry-Esseen Bound in Equation 4.13 */
    $\Delta = 0.5600 \times \rho/\sigma^3$

    foreach $bin \in bins$ do
        $[a, b] \leftarrow$ the boundaries of $bin$

        /* Computation of Approximate Probability */
        $prob \leftarrow \Phi(\frac{b-\mu}{\sigma}) - \Phi(\frac{a-\mu-1}{\sigma})$

        /* Hoeffding Bound in Equation 4.14 */
        if $b < \mu$ then
            $\lambda \leftarrow \mu - b$
            $hoeffding = 2exp(-2\lambda^2/\kappa)$
        else if $a > \mu$ then
            $\lambda \leftarrow a - \mu$
            $hoeffding = exp(-2\lambda^2/\kappa)$
        else
            $hoeffding = 1$
        end

        /* Computation of Lower and Upper Probability Bounds */
        $lower \leftarrow max(prob - 2\Delta, 0)$
        $upper \leftarrow min(prob + 2\Delta, hoeffding)$

        /* $histogram[bin]$ is the probability of $bin$ in $histogram$ */
        $histogram[bin] \leftarrow$ the approximate probability $prob$ with bound $[lower, upper]$
    end
    return $histogram$
end
```

To understand how the algorithm works, let's consider the convolution of N random variables $X_1, X_2, \ldots, X_N$

$$Y_N = \sum_{i=1}^{N} X_i$$

We define Y_k to be the intermediate result after the convolution of the first k random variables

$$Y_k = \sum_{i=1}^{k} X_i$$

Let V_{min}^k and V_{max}^k be the least and largest values in the distribution $\sum_{i=k+1}^{N} X_i$:

$$V_{min}^k = \sum_{i=k+1}^{N} \min(X_i)$$

$$V_{max}^k = \sum_{i=k+1}^{N} \max(X_i)$$

For $(v, p) \in Y_k$, if $v + V_{min}$ and $v + V_{max}$ belong to the same bin $[a, b]$, i.e.

$$a \leq v + V_{min} \leq v + V_{max} \leq n$$

By definition, we have $V_{min} \leq \sum_{i=k+1}^{N} X_i \leq V_{max}$, hence

$$a \leq v + \sum_{i=k+1}^{N} X_i \leq b \tag{4.15}$$

Equation 4.15 suggests that while the value v in the intermediate distribution might be convolved into other values by the remaining variables $X_{k+1}, X_{k+1}, \ldots, X_N$, the convolved values will always belong to the bin $[a, b]$. Therefore the result will not be changed if one simply remove the v from the intermediate distribution and add p to the bin $[a, b]$ directly.

Based on this idea, we develop the Early Binning Algorithm (EBA), which extends Standard DP (Algorithm 3 on page 22) by moving values in the intermediate distribution to the appropriate bin as early as possible. This results in a smaller intermediate distribution, potentially leading to an exponential saving. Intuitively, the algorithm works the best when some of the bins are wide enough, making the algorithm suitable for histogram zooming. The Early Binning Algorithm is presented in Algorithm 22. Despite the algorithm's slightly higher complexity $\mathcal{O}(N^2 R(R + \log B))$, where R is the range of the distributions for the node's children and B is the target number of bins, the algorithm is expected to have a better average runtime due to the potentially shrinking intermediate distribution.

Together with the fact that the Recursive FFT Algorithm (Algorithm 19 on page 49) is also applicable to a SUM node, we now have three different algorithms for the computation of histogram with exact probabilities for SUM nodes, namely the Deferred Binning Algorithm (Algorithm 18), the Early Binning Algorithm (Algorithm 22) and the Recursive FFT Algorithm. While the Recursive FFT Algorithm has lower complexity $\mathcal{O}(NR \log(NR) \log(N))$, where N is the number of children of a SUM node and R is the range of the distribution of its children, the Deferred Binning Algorithm and the Early Binning Algorithm are heavily optimised for the histogram approximation setting, therefore it is unclear which algorithms will have the best performance at this point. For this reason, their performance will be examined and compared in Chapter 7.

ALGORITHM 22: Histogram Evaluation for a SUM Node - Early Binning Algorithm (EBA)

Input: Sum node *node*, Bin Intervals *bins*
Output: Histogram with bin intervals *bins*

```
HistogramEvaluation(node, bins)
begin
    histogram ← New Histogram with bins

    $X \leftarrow \{\}$                                  /* X holds the intermediate distribution */
    $X[0] \leftarrow 1$
    for $i = 1$ to $N$ do
        /* Convolution of random variables via Standard DP                          */
        $Y \leftarrow$ ExactEvaluation($i^{th}$ child of node)
        $Z \leftarrow \{\}$                                                 /* Z = X + Y */
        for $(v_x, p_x) \in X$, $(v_y, p_y) \in Y$ do
            $Z[v_x + v_y] \leftarrow Z[v_x + v_y] + p_x p_y$
        end

        /* Calculating minimum and maximum shift                                    */
        $V_{min}, V_{max} \leftarrow 0, 0$
        for $j = i + 1$ to $N$ do
            $[L, R] \leftarrow$ GetDistributionRange($j^{th}$ child of node)
            $V_{min} \leftarrow V_{min} + L$
            $V_{max} \leftarrow V_{max} + R$
        end

        /* Early Binning Extension - Run only when there are values to be binned
           early                                                                    */
        if $V_{max} - V_{min} \leq$ width of at least one of the bins then
            foreach $(v, p) \in Z$ do
                $b_{min} \leftarrow$ the bin containing $v + V_{min}$       /* via Binary Search */
                $b_{max} \leftarrow$ the bin containing $v + V_{max}$       /* via Binary Search */
                if $b_{min} = b_{max}$ then
                    Remove $v$ from $Z$
                    $histogram[b_{min}] \leftarrow histogram[b_{min}] + p$
                end
            end
        end

        $X \leftarrow Z$
    end

    return histogram
end
```

Chapter 5

Top-k Approximation

5.1 Overview

Top-k approximation retrieves only the most probable values and the corresponding probabilities in the probability distribution of aggregate query result. While histogram approximation provides a synopsis, top-k approximation provides the highest quality results in the distribution. In this chapter, we will present efficient algorithms for evaluation of MIN, MAX, COUNT, and SUM aggregate queries in this approximation setting.

5.1.1 Strategy

Similar to histogram approximation, the most straight-forward approach to support top-k approximation is to first perform an exact evaluation on the aggregate query to obtain the full probability distribution over the possible answers, and the most probable k answers can then be extracted straightforwardly after the distribution is sorted by probability. Arguably, by returning only the k most probable answers, this approach provides more concise and intuitive result than exact evaluation. However, the approach does not take advantage of the approximation setting fully to improve the performance of query evaluation.

For this reason, we will develop algorithms specifically designed for top-k approximations. The recursive nature of a decomposition tree (d-tree) suggests that the evaluation of top-k approximation can be tackled by solving the identical problem for each node in a d-tree. More specifically, given a convolution node (or UNION node) $\oplus$ in a d-tree, with N children represented by the random variables $X_1, X_2, \ldots, X_N$, we need to develop algorithms to compute

$$P_{v_i} = P(X_1 \oplus X_2 \oplus \ldots \oplus X_N = v_i) \tag{5.1}$$

in increasing order of $i \in \mathbb{N}_1$ sequentially, where $P_{v_i} \geq P_{v_i+1}$.

The ultimate goal is to perform the computation for Equation 5.1 without having the full probability distributions for its children X_i, but only with the most probable values of the distributions. This ensures the algorithms can be applied to the child node of $\oplus$ in the same way as the node $\oplus$ itself in a recursive manner, allowing speedup to happen in all levels of the d-tree.

5.1.2 Framework

The framework for top-k approximation relies on one core method and two auxiliary methods for each type of nodes in a decomposition tree:

Next Most Probable Value

This is the core method for the evaluation of top-k approximation for each node, where the tuple containing the next most probable value and the corresponding probability for the result of the current node is returned. The interface is presented in Algorithm 23.

```
ALGORITHM 23: Next Top Interface
Input: A Node in Decomposition Tree node
Output: The Next Most Probable Tuple (Value, Probability)
NextTop(node)
```

After compiling an aggregate query into a d-tree, top-k approximation can then be achieved by the calling the interface NextTop (Algorithm 23) k times on the root node of the d-tree, as depicted in Algorithm 24.

```
ALGORITHM 24: Top-k Evaluation for a Decomposition Tree
Input: Root Node of a Decomposition Tree root, Number of Most Probable Values k
Output: Top-k Most Probable Values with the corresponding Probability
TopkEvaluation(node)
begin
    topk ← Empty List
    for $i = 1$ to $k$ do
        Add NextTop(node) to topk
    end
    return topk
end
```

Next Least Value & Next Largest Value

The proposed algorithm for the computation of the next most probable values for a MIN/-MAX node (Algorithm 30 on page 69) consumes the probability distributions of its children sequentially in sorted order of values. Owing to the fact that the children of a MIN/MAX node can be any type of nodes, it is necessary for all types of nodes to support efficient retrieval of the least and largest values (and the corresponding probabilities) sequentially. For this reason, the framework requires two auxiliary methods for the nodes, and their interfaces are presented in Algorithm 25 and Algorithm 26 respectively.

While retrieving the next least and largest values are auxiliary methods for the computation of the most probable values, there are situations when they can be applicable in their own rights. For example, a risk-taking investor might be interested in finding out the highest possible profit of an investment and the corresponding probability, where the probability distribution for the profit is derived from an aggregate query. This can be done efficiently by calling NextLargest() on the root node of the d-tree to retrieve the largest value in the final distribution.

```
ALGORITHM 25: Next Least Tuple Interface
Input: A Node in Decomposition Tree $node$
Output: The Next Least Tuple $(Value, Probability)$
NextLeast($node$)
```

```
ALGORITHM 26: Next Largest Tuple Interface
Input: A Node in Decomposition Tree $node$
Output: The Next Largest Tuple $(Value, Probability)$
NextLargest($node$)
```

Because NextLeast() and NextLargest() are symmetric to each other, we will focus on the algorithms for NextLargest() in the rest of the chapter. It should then be straight forward to adapt the algorithms to work for NextLeast().

5.2 Algorithms

5.2.1 VARIABLE

VARIABLE nodes correspond to the random variables in semimodule expressions, where each random variable carries a complete probability distribution. Because the leaves of d-trees are always VARIABLE nodes, the algorithms for VARIABLE node also form the base cases for the recursive computation of top-k approximation.

Next Largest

With the complete probability distribution carried by a VARIABLE node, the next largest value in the distribution can easily be retrieved by first sorting the distribution by value, and the tuple $(value, probability)$ can then be returned one by one in a similar manner as an iterator. Because it is natural to store the probability distributions of the random variables by value in the database, the sorting step can often be skipped. For a VARIABLE node with a (unsorted) distribution of support size M, the preprocessing step takes $\mathcal{O}(M \log M)$ and the subsequent retrieval takes $\mathcal{O}(1)$ per tuple.

Next Top

Retrieving the next most probable tuple $(value, probability)$ for a VARIABLE node is similar to the process for retrieving the next largest tuple, except the distributions are sorted by probability instead of value. Once again, for a VARIABLE node with a distribution of support size M, the preprocessing step takes $\mathcal{O}(M \log M)$ and the subsequent retrieval takes $\mathcal{O}(1)$ per tuple.

5.2.2 UNION

UNION node is created when Shannon expansion is performed on one or more variables for a semimodule expression, and each of the branches of a UNION node corresponds to one of the possible assignments for the expanding variables. Consider a UNION node Y with children $X_1, X_2, \ldots, X_N$, where the probability of the assignment for the branch connecting the UNION node Y and the child X_i is w_i, then the result of the computation on the UNION node is given by the probability distribution of Y, where

$$P(Y = v) = \sum_{i=1}^{N} w_i \times P(X_i = v) \tag{5.2}$$

This suggests exact evaluation on a UNION node (using Algorithm 2 on page 22) has a complexity $\mathcal{O}(NM)$, where M is the support size of the probability distribution for individual child X_i. Technically, if the support sizes of the probability distributions for $X_i, i = 1 \ldots N$ are different, M refers to the maximum one.

Next Largest

According to Equation 5.2, the possible values in Y come directly from the possible values in X_i, hence

$$supp(Y) = \bigcup_i supp(X_i)$$

The result indicates that the next largest value in Y must come from the next largest value in its children. Base on this idea, we propose the algorithm to retrieve the next largest value for a UNION node in Algorithm 27, where a maximum heap is used to retrieve the largest value from its children efficiently.

The number of tuples to be popped from the heap for the retrieval of the next largest tuple (v, p) depends on the number of the children having the value v in their distributions. When the value v is only in the distribution of one child, then only one tuple needs to be popped. On the other hand, if the value v is in the distribution of all N children, then N tuples need to be popped from the heap for the retrieval. Therefore the complexity of the algorithm ranges from $\mathcal{O}(\log N)$ to $\mathcal{O}(N \log N)$ per retrieval.

Example 7. *To demonstrate how Algorithm 27 proceeds in retrieving the next largest value and the corresponding probability, consider a* UNION *node with three children represented by the random variables* X_1, X_2*, and* X_3*, where the branch probabilities from the* UNION *node to* X_1, X_2*, and* X_3 *are 0.5, 0.3, and 0.2 respectively.*

We will represent the probability distributions of the random variables in the form of tuples (value, probability) in the following table:

$w_1 = 0.5$	$w_2 = 0.3$	$w_3 = 0.2$
X_1	X_2	X_3
(100, 0.8) *(20, 0.2)*	 *(20, 0.6)* *(10, 0.4)*	 *(20, 1.0)*

ALGORITHM 27: Next Largest Tuple for a UNION Node

```
Input: UNION Node $node$
Output: The Tuple $(Value, Probability)$ with Next Largest Value

NextLargest($node$)
begin
    /* Execute only the first time the method is called */
    if $not\ initialised$ then
        $pool \leftarrow$ Empty Max Heap
        for $i = 1$ to $N$ do
            $child_i \leftarrow i^{th}$ children of $node$
            $w_i \leftarrow$ Branch Probability from $node$ to $child_i$
            $(v, p) \leftarrow NextLargest(child_i)$
            Add $(v, p, i)$ to $pool$ with $Key = v$
        end
        $initialised \leftarrow True$
    end

    $p_{total} \leftarrow 0$
    $v \leftarrow$ Key of Largest Element in $pool$
    while $Key\ of\ the\ Largest\ Element\ in\ pool = v$ do
        $v, p, i \leftarrow$ Pop Largest Element in $pool$
        $p_{total} \leftarrow p_{total} + p \times w_i$                /* Equation 5.2 */
        $(v_{next}, p_{next}) \leftarrow NextLargest(child_i)$
        Add $(v_{next}, p_{next}, i)$ to $pool$ with $Key = v_{next}$
    end
    return $(v, p_{total})$
end
```

During initialisation stage, the next largest value for each of the children will be retrieved and placed into a maximum heap. The values currently in the heap are labelled in red:

$w_1 = 0.5$	$w_2 = 0.3$	$w_3 = 0.2$
X_1	X_2	X_3
(100, 0.8)		
(20, 0.2)	*(20, 0.6)*	*(20, 1.0)*
	(10, 0.4)	

Firstly, the largest value in the heap is popped, which corresponds to the tuple $(100, 0.8)$ *from* X_1*. After the tuple probability* 0.8 *is multiplied by* $w_1 = 0.5$*, we have computed the first largest tuple* $(100, 0.4)$ *for the* UNION *node. At the same time, because we have popped the value that belongs to* X_1 *in the heap, we will retrieve the next largest value from* X_1 *and add it to the heap:*

$w_1 = 0.5$	$w_2 = 0.3$	$w_3 = 0.2$
X_1	X_2	X_3
~~*(100, 0.8)*~~		
(20, 0.2)	*(20, 0.6)*	*(20, 1.0)*
	(10, 0.4)	

The next largest value in the heap is 20*. However, because the value is in the distribution of* $X_1, X_2,$ *and* X_3*, three tuples* $(20, 0.2), (20, 0.6), (20, 1.0)$ *will be popped from the heap. Combining the tuples according to Equation 5.2 leads us to the next largest tuple* $(20, 0.48)$ *for the* UNION *node. Because the value we popped belongs to all three variables, therefore we should retrieve the next largest tuples for all three children and add them to the heap. However, the values in* X_1 *and* X_3 *have been exhausted, therefore we can only add the next largest value from* X_2 *to the heap:*

$w_1 = 0.5$	$w_2 = 0.3$	$w_3 = 0.2$
X_1	X_2	X_3
~~*(100, 0.8)*~~		
~~*(20, 0.2)*~~	~~*(20, 0.6)*~~	~~*(20, 1.0)*~~
	(10, 0.4)	

Lastly, the final value in the heap is popped, corresponding to the tuple $(10, 0.4)$*. Together with the weight* $w_2 = 0.3$*, the next largest tuple for the* UNION *node* $(10, 0.12)$ *is computed. At this point, the computation is done as the heap is empty.*

$w_1 = 0.5$	$w_2 = 0.3$	$w_3 = 0.2$
X_1	X_2	X_3
~~*(100, 0.8)*~~		
~~*(20, 0.2)*~~	~~*(20, 0.6)*~~	~~*(20, 1.0)*~~
	~~*(10, 0.4)*~~	

Next Most Probable

Equation 5.2 indicates that for a UNION node with N children represented by the random variables $X_1, X_2, \ldots, X_N$, $P(\text{UNION}(X_1, X_2, \ldots, X_N) = v)$ is a monotonic function of the corresponding probabilities from its children $P(X_i = v), i = 1 \ldots N$. This suggests that if we retrieve the next most probable tuples from the children sequentially, it is possible to deduce the next most probable tuple for the UNION node early by using Threshold Algorithm (TA) [16].

We will explain TA via an example. Consider a UNION node with two children X_1, X_2, where the probabilities of the branches from the node to X_1 and X_2 are $w_1 = 0.6$ and $w_2 = 0.4$ respectively. Part of the probability distributions for X_1 and X_2 are shown in the following table:

$w_1 = 0.6$	$w_2 = 0.4$
X_1	X_2
(100, 0.3)	(50, 0.6)
(80, 0.1)	(100, 0.1)
(45, 0.08)	(65, 0.05)
⋮	⋮

In each step, we retrieve the next most probable tuples from X_1 and X_2, and derive a lower bound and upper bound for $P(\text{UNION}(X_1, X_2) = v)$, where $v \in$ the set of values retrieved from either X_1 or X_2. Therefore the first step involves retrieving the tuples $(100, 0.3)$ and $(50, 0.6)$ from X_1 and X_2 respectively.

Because the tuple $(100, 0.3)$ we just retrieved from X_1 is the tuple with the highest probability in X_1, it is guaranteed that all the other tuples we retrieve from X_1 subsequently will have a lower probability than 0.3. For this reason, we will call this value the threshold of X_1, labelled as $T_1 = 0.3$. While the tuple we retrieved from X_2 indicates that $P(X_2 = 50) = 0.6$, we do not have enough information to compute $P(\text{UNION}(X_1, X_2) = 50)$ exactly as we haven't retrieved the tuple with value 50 from X_1 yet. In fact, it is possible that 50 is not even in the distribution of X_1. However, it is guaranteed that

$$0 \leq P(X_1 = 50) \leq T_1$$

The inequality and the information that $P(X_2 = 50) = 0.6$ can then be used in Equation 5.2 to derive the following bound on $P(\text{UNION}(X_1, X_2) = 50)$:

$$0 \times w_1 + P(X_2 = 50) \times w_2 \leq P(\text{UNION}(X_1, X_2) = 50) \leq T_1 \times w_1 + P(X_2 = 50) \times w_2$$
$$0.24 \leq P(\text{UNION}(X_1, X_2) = 50) \leq 0.42$$

Similarly, we can derive a threshold on X_2 based on the information that the tuple $(50, 0.6)$ we retrieved from X_2 has a higher probability than any other subsequent tuples, hence $T_2 = 0.6$. However, we also note that the sum of probabilities for all the tuples in X_2 is 1, therefore we can place a tighter bound on the threshold: $T_2 = 1 - 0.6 = 0.4$. The threshold can then be used with the information $P(X_1 = 100) = 0.3$ in Equation 5.2 to get the following bound:

$$P(X_1 = 100) \times w_1 + 0 \times w_2 \leq P(\text{UNION}(X_1, X_2) = 50) \leq P(X_1 = 100) \times w_1 + T_2 \times w_2$$
$$0.18 \leq P(\text{UNION}(X_1, X_2) = 50) \leq 0.34$$

The current state of the computation is depicted in the following tables (the red labels indicate the tuples that have been retrieved for processing in the current step):

$w_1 = 0.6$	$w_2 = 0.4$
X_1	X_2
(100, 0.3)	(50, 0.6)
(80, 0.1)	(100, 0.1)
(45, 0.08)	(65, 0.05)
⋮	⋮
$T_1 = 0.3$	$T_2 = 0.4$

Retrieved Tuples

Value	Lower Prob	Upper Prob
50	0.24	0.42
100	0.18	0.34

Because the bounds for the values 50 and 100 overlap, we are not in a position to deduce whether 50 or 100 is the most probable tuple yet, therefore we have to continue the computation to tighten the bounds. Retrieving the next most probable tuples from X_1 and X_2 and updating the bounds of the retrieved values using a similar procedure as the previous step lead us to the following state:

$w_1 = 0.6$	$w_2 = 0.4$
X_1	X_2
~~(100, 0.3)~~	~~(50, 0.6)~~
(80, 0.1)	(100, 0.1)
(45, 0.08)	(65, 0.05)
⋮	⋮
$T_1 = 0.1$	$T_2 = 0.1$

Retrieved Tuples

Value	Lower Prob	Upper Prob
50	0.24	0.30
100	0.22	0.22
80	0.06	0.10

At this point, we have confirmed that the value 50 has a lower probability bound higher than the upper probability bounds of all *seen* tuples (i.e. value 100 and 80), making it a candidate of being the most probable tuple for UNION(X_1, X_2). To confirm this, we have to ensure that the lower probability bound of the value 50 is also higher than the upper probability bounds for all the *unseen* tuples (such as values 45 and 65). Base on T_1 and T_2 with Equation 5.2, we can derive an overall threshold T that provides an upper bound on the probabilities of the unseen tuples:

$$\begin{aligned} T &= T_1 \times w_1 + T_2 \times w_2 \\ &= 0.1 \times 0.6 + 0.1 \times 0.4 \\ &= 0.1 \end{aligned}$$

Because $T < 0.24$, the result confirms that the value 50, with a probability bound $[0.24, 0.30]$ is the most probable tuple.

In summary, TA consists of the following four steps in each stage:

1. Retrieve the next most probable tuples from X_1 and X_2. These tuples will be referred to as the seen tuples.
2. Deduce an upper bound on the probability of the unseen tuples for X_1 and X_2, labelled T_1 and T_2. Specifically, assuming the last tuple retrieved from X_i is (v_i, p_i), then $T_i = min(p_i, 1 - p_i)$.
3. For the values we have seen (either in this step or in previous steps), compute a lower bound and an upper bound for the probability using Equation 5.2 with T_1 and T_2.
4. Compute the upper probability bound for the unseen tuples $T = T_1 \times w_1 + T_2 \times w_2$. If there exists a seen tuple with a lower probability bound higher than the upper probability bound of all other seen tuples (i.e. the values we have computed in step 3) and unseen tuples (i.e. the value T), return the tuple. Otherwise repeat from step 1.

The TA algorithm we adapted for the purpose of computing the next most probable tuple on a UNION node is presented in Algorithm 28. TA has been proved to be instance optimal [16], which means it has the optimal complexity for all input instances.

Note that while the algorithm returns probability bounds (i.e. top-k approximation with approximate probability), the bounds can be tightened by allowing the algorithm to scan more tuples from the children. Nonetheless, the bounds will generally be very tight, as they would usually overlap with the probability bounds of another tuple otherwise.

ALGORITHM 28: Next Most Probable Tuple for a UNION Node - Threshold Algorithm (TA)

Input: UNION Node *node*
Output: The Next Most Probable Tuple ($Value, Probability$)

```
NextLargest(node)
begin
    /* Execute only the first time the method is called */
    if not initialised then
        $pool \leftarrow \{\}$
        $retrieved \leftarrow \{\}$
        $guide \leftarrow MaxHeap$
        for $i = 1$ to $N$ do
            $child_i \leftarrow i^{th}$ children of node
            $w_i \leftarrow$ Branch Probability from node to $child_i$
            $(v, p) \leftarrow NextTop(child_i)$
            Add $(v, p, i)$ to guide with Key $(p \times w_i)$
            $cumulative_i \leftarrow 1$
        end
        $initialised \leftarrow True$
    end

    while True do
        $(v, p, i) \leftarrow$ Pop the largest item from guide    /* Optimisation: Retrieve from the
        children that will lower the threshold the most */
        if $v \notin retrieved$ then
            if $v \in pool$ then
                $v, l, unseen \leftarrow$ Pop $v$ from pool
                Remove $i$ from unseen
                $l \leftarrow l + p \times w_i$    /* Update the lower probability bound for $v$ */
            else
                $unseen = \{1, 2, \ldots, N\}$
                $l \leftarrow p \times w_i$    /* Compute the lower probability bound for $v$ */
            end
            Add $(v, l, unseen)$ to pool
        end

        $cumulative_i \leftarrow cumulative_i - p$
        $T_i \leftarrow \min(p, cumulative_i)$
        $threshold \leftarrow \sum_{i=1}^{N} T_i \times w_i$    /* Update threshold $T$ */

        $(v, p) \leftarrow NextTop(child_i)$
        Add $(v, p, i)$ to guide with Key $(p \times w_i)$

        $v_{top}, l_{top}, unseen_{top} \leftarrow$ tuple in pool with largest $l$
        if $l_{top} \geq threshold$ then
            if $l_{top} \geq$ all elements in $\{l + \sum_{i \in unseen} T_i \times w_i,\ (v, l, unseen) \in pool \wedge v \neq v_{top}\}$ then
                $u_{top} \leftarrow l + \sum_{i \in unseen_{top}} T_i \times w_i$    /* Upper probability bound for $v_{top}$ */
                Move the tuple for $v_{top}$ from pool to retrieved
                return $(v_{top}, l_{top}, u_{top})$
            end
        end
    end
end
```

5.2.3 MIN/MAX

MIN and MAX nodes correspond to the operators $+_{\min}$ and $+_{\max}$ in semimodule expressions respectively. Because MIN and MAX are symmetric to each other, the algorithms developed for one of them can readily be adapted to work on the other. For this reason, we will only focus on MAX nodes in this section. Exact evaluation for a MIN/MAX node can be done via multiple applications of Standard DP (Algorithm 3 on page 22), with a complexity $\mathcal{O}((NM)^2)$, where N is the number of children for the MIN/MAX node and M is the support size of the probability distributions for its children.

Next Largest

Proposition 1 on page 38 states that

$$supp(\mathrm{MAX}(X_1, X_2, \ldots, X_N)) = \bigcup_{i=1}^{N} supp(X_i) - \{x \in \mathbb{N} : x < \max_{i=1\ldots N}(\min(supp(X_i)))\} \quad (5.3)$$

Equation 5.3 suggests the next largest value for a MAX node comes directly from the next largest value of its children.

Proposition 2 on page 39 states that

$$P(MAX(X_1, X_2, \ldots, X_N) \leq v) = \prod_{i=1}^{N} P(X_i \leq v)$$

Therefore

$$P(MAX(X_1, X_2, \ldots, X_N) = v) = \prod_{i=1}^{N} P(X_i \leq v) - \prod_{i=1}^{N} P(X_i < v) \quad (5.4)$$

Equation 5.3 and Equation 5.4 can be combined so that one can retrieve the next largest tuple for a MAX node by retrieving the next largest tuples from its children sequentially and keeping track of the total remaining probabilities for the children. Base on this idea, we propose the Heap Method in Algorithm 29, which is similar to the algorithm for retrieving the next largest tuple for a UNION node (Algorithm 27 on page 59), except for the computation of probabilities for the tuples. Similar to Algorithm 27, the algorithm has a complexity ranging from $\mathcal{O}(\log N)$ to $\mathcal{O}(N \log N)$ per retrieval.

Example 8. *We demonstrate the Heap Method by considering a MAX node with three children represented by the random variables X_1, X_2 and X_3. We represent the probability distributions of the random variables in the form of tuples (value, probability) in the following table:*

X_1	X_2	X_3
(100, 0.8)		
(20, 0.2)	*(20, 0.6)*	*(20, 1.0)*
	(10, 0.4)	

ALGORITHM 29: Get Next Largest Tuple for a MAX Node - Heap Method (HM)

Input: MAX Node *node*
Output: The Tuple (*value*, *probability*) with Next Largest Value

```
NextLargest(node)
begin
    /* Execute only the first time the method is called */
    if not initialised then
        $heap \leftarrow$ Empty Max Heap
        for $i = 1$ to $N$ do
            $child_i \leftarrow i^{th}$ child of $node$ of children
            $(v, p) \leftarrow NextLargest(child_i)$
            Add $(v, p, i)$ to $heap$ with $key = v$
            Set $R_i \leftarrow 1$
        end
        $R_{total} \leftarrow 1$
        $initialised \leftarrow True$
    end

    $R'_{total} \leftarrow R_{total}$
    $v \leftarrow$ Key of Largest Element in $heap$
    while Key of Largest Element in heap $= v$ do
        $(v, p, i) \leftarrow$ Pop Largest Element in $heap$
        $R'_i \leftarrow R_i$                                /* $R'_i = P(children_i \leq v)$ */
        $R_i \leftarrow R_i - p$                             /* $R_i = P(children_i < v)$ */
        $R_{total} \leftarrow R_{total} \times R_i / R'_i$  /* $R_{total} = \prod_{i=1}^{N} R_i$ */
        $(v_{next}, p_{next}) \leftarrow NextLargest(child_i)$
        Add $(v_{next}, p_{next}, i)$ to $heap$ with $key = v_{next}$
    end
    $p_{total} \leftarrow R'_{total} - R_{total}$           /* Equation 5.4 */
    return $(v, p_{total})$
end
```

During initialisation stage, the next largest value for each of the children will be retrieved and placed into a maximum heap. Additional, the remaining probability for the variable X_i will be labelled as R_i, which has the value 1 initially. We arrive at the following state after the initialisation stage, where the values currently in the heap are labelled in red:

X_1	X_2	X_3
(100, 0.8)		
(20, 0.2)	*(20, 0.6)*	*(20, 1.0)*
	(10, 0.4)	
$R_1 = 1$	$R_2 = 1$	$R_3 = 1$

Firstly, the tuple with the largest value $(100, 0.8)$ *from* X_1 *is popped from the heap. According to Equation 5.4, we have*

$$\begin{aligned} P(\text{MAX}(X_2, X_2, X_3) = 100) &= R_1 \times R_2 \times R_3 - (R_1 - 0.8) \times R_2 \times R_3 \\ &= 0.8 \end{aligned}$$

Therefore the largest tuple $(100, 0.8)$ *is returned. We then decrease* R_1 *by* 0.8*, and put the next largest value retrieved from* X_1 *to the heap, leading us to the following state:*

X_1	X_2	X_3
~~*(100, 0.8)*~~		
(20, 0.2)	*(20, 0.6)*	*(20, 1.0)*
	(10, 0.4)	
$R_1 = 0.2$	$R_2 = 1$	$R_3 = 1$

At this point, the largest value in the heap is 20*, shared by three tuples from each of the random variables:* $(20, 0.2), (20, 0.6), (20, 1.0)$*. We pop all three of them from the heap, and compute the probability according to Equation 5.4 to give:*

$$\begin{aligned} P(MAX(X_2, X_2, X_3) = 20) &= R_1 \times R_2 \times R_3 - (R_1 - 0.2) \times (R_2 - 0.6) \times (R_3 - 1.0) \\ &= 0.2 \end{aligned}$$

Therefore the next largest tuple $(20, 0.2)$ *is returned. Because we have consumed all the values in the distribution of* X_1 *and* X_3*, we will only retrieve the next largest value from* X_2 *and add it to the heap. Updating the values for* R_1, R_2, R_3 *leads us to*

X_1	X_2	X_3
~~*(100, 0.8)*~~		
~~*(20, 0.2)*~~	~~*(20, 0.6)*~~	~~*(20, 1.0)*~~
	(10, 0.4)	
$R_1 = 0$	$R_2 = 0.4$	$R_3 = 0$

Lastly, the final tuple $(10, 0.4)$ *is popped from the heap. The probability is given by*

$$\begin{aligned} P(MAX(X_2, X_2, X_3) = 100) &= R_1 \times R_2 \times R_3 - R_1 \times (R_2 - 0.4) \times R_3 \\ &= 0 \end{aligned}$$

At this point, the computation is completed as all the values in the distributions have been consumed.

X_1	X_2	X_3
~~*(100, 0.8)*~~		
~~*(20, 0.2)*~~	~~*(20, 0.6)*~~	~~*(20, 1.0)*~~
	~~*(10, 0.4)*~~	

Exact Evaluation using the Heap Method

We observe that using the Heap Method, which is designed for top-k approximation, to retrieve all the tuples in the distribution can still acquire a lower complexity than using Standard DP, suggesting the Heap Method as a replacement for Standard DP for exact evaluation over MIN/MAX nodes.

Firstly, consider exact evaluation on a MAX node with N children, represented by the random variables $X_1, X_2, \dots, X_N$. We further assume that the support size of the distributions of the children is M. While Standard DP does not require the distributions of the children to be sorted, the Heap Method acts on distributions that are sorted by value. Therefore the preprocessing step for using the Heap Method for exact evaluation involves sorting N distributions of size M, which has a complexity of $\mathcal{O}(NM \log M)$.

With the distributions sorted, the Heap Method can then be used to compute the exact evaluation result on the MAX node one by one. In exact evaluation, we are interested in the entire probability distribution, therefore we will compute not just the largest k tuples, but all of them instead. Because the Heap Method proceeds by adding and popping each value retrieved from the children to a heap of size N once, it has a complexity $\mathcal{O}(NM \log N)$ in the computational step.

Combining the complexity of the preprocessing step and the computational step leads us to

$$\mathcal{O}(NM \log M) + \mathcal{O}(NM \log N) = \mathcal{O}(NM \log(NM))$$

The result suggests that using the Heap Method for exact computation has a lower complexity than using Standard DP, which has a complexity $\mathcal{O}((NM)^2)$.

Next Most Probable

Before we propose the algorithm for computing the next most probable tuple for a MAX node, we present a key observation regarding the probability distribution of the evaluation result for a MAX node:

Base on Proposition 2 on page 39, we have

$$P(\text{MAX}(X_1, X_2, \dots, X_N) \leq v) = \prod_{i=1}^{N} P(X_i \leq v)$$

Therefore

$$P(\text{MAX}(X_1, X_2, \dots, X_N) > v) = 1 - \prod_{i=1}^{N} P(X_i \leq v) \tag{5.5}$$

According to Equation 5.5, the probability $P(\text{MAX}(X_1, X_2, \dots, X_N) > v)$ converges to 1 quickly as N increases, even when v is relatively large (i.e. $P(X_i \leq v)$ is also large). For example, if $P(X_1 \leq v) = p$ for $i = 1 \dots N$, then the probability for

$$P(\text{MAX}(X_1, X_2, \dots, X_N) > v) = 1 - \prod_{i=1}^{N} p = 1 - p^N$$

is captured in the following table:

$1 - p^N$

	$p = 0.5$	$p = 0.9$	$p = 0.99$	$p = 0.999$
$N = 100$	1.000000	0.999973	0.633968	0.095208
$N = 1000$	1.000000	1.000000	0.999957	0.632304
$N = 10000$	1.000000	1.000000	1.000000	0.999955
$N = 100000$	1.000000	1.000000	1.000000	1.000000

The result demonstrates that most of the probability mass in $\text{MAX}(X_1, X_2, \ldots, X_N)$ lies on the tuples with the largest values, even if the distributions of the input variables are heavily skewed. In other words, the most probable tuples are likely to be the ones with the largest values. Therefore if we compute the tuples starting from the ones with the largest values, it would not take long before we encounter the most probable tuple (v, p).

At the same time, by tracking a threshold T on the upper probability bound for the unseen tuples (i.e. tuples with smaller values from the children that haven't been retrieved yet), the most probable tuple can be confirmed once $p > T$. More specifically, if we keep track of the remaining probability R_i for each of the random variables X_i (i.e. if the last tuple retrieved from the children has the value u, then $R_i = P(X_i < u)$), then the total probability of the unseen tuples is given by:

$$\begin{aligned} P(\text{MAX}(X_1, X_2, \ldots, X_N) < u) &= \prod_{i=1}^{N} P(X_i < u) \\ &= \prod_{i=1}^{N} R_i \end{aligned}$$

Because any unseen tuples cannot possibly have a probability larger than the total probability of all the unseen tuples, the threshold T is defined by:

$$T = \prod_{i=1}^{N} R_i$$

Since the Heap Method (Algorithm 29 on page 65) already computes the largest tuples sequentially, we simply have to adapt the algorithm so that it also takes into account of the threshold T before returning a tuple. The algorithm, known as the Extended Heap Method (EHM), is presented in Algorithm 30.

5.2.4 COUNT

A COUNT node can be treated as a special case of a SUM node, where the input random variables have support $\{0, 1\}$. However, due to the popularity of COUNT aggregate queries, we will develop algorithms optimised for COUNT nodes. Exact evaluation for a COUNT node can be done via multiple applications of Standard DP (Algorithm 3 on page 22) with a complexity $\mathcal{O}(N^2)$, where N is the number of children for the COUNT node.

In particular, the computation for the distribution of a COUNT node Y can be defined by

$$Y = \sum_{i=1}^{N} X_i \tag{5.6}$$

where $X_1, X_2, \ldots, X_N$, representing the children of the COUNT node, are independent random variables with probability distributions

$$\begin{aligned} P(X_i = 1) &= p_i \\ P(X_i = 0) &= \bar{p_i} = 1 - p_i \end{aligned}$$

ALGORITHM 30: Get Next Most Probable Tuple for a MAX Node - Extended Heap Method (EHM)

Input: MAX Node *node*
Output: The Next Most Probable Tuple (*value*, *probability*)

```
NextTop(node)
begin
    /* Execute only the first time the method is called */
    if not initialised then
        heap ← Empty Max Heap
        pool ← Empty Max Heap        /* Holds tuples computed but not yet returned */
        for $i = 1$ to $N$ do
            $child_i \leftarrow i^{th}$ child of node of children
            $(v, p) \leftarrow NextLargest(child_i)$
            Add $(v, p, i)$ to heap with $key = v$
            Set $R_i \leftarrow 1$
        end
        $R_{total} \leftarrow 1$
        $initialised \leftarrow True$
    end

    while pool is empty or probability of the most probable tuple in pool $< R_{total}$ do
        /* Computing the next largest tuple */
        $R'_{total} \leftarrow R_{total}$
        $v \leftarrow$ Key of Largest Element in heap
        while Key of Largest Element in heap $= v$ do
            $v, p, i \leftarrow$ Pop Largest Element in heap
            $R'_i \leftarrow R_i$                          /* $R'_i = P(children_i \leq v)$ */
            $R_i \leftarrow R_i - p$                       /* $R_i = P(children_i < v)$ */
            $R_{total} \leftarrow R_{total} \times R_i / R'_i$       /* $R_{total} = \prod_{i=1}^{N} R_i$ */
            $(v_{next}, p_{next}) \leftarrow NextLargest(child_i)$
            Add $(v_{next}, p_{next}, i)$ to heap with $key = v_{next}$
        end
        $p_{total} \leftarrow R'_{total} - R_{total}$               /* Equation 5.4 */
        Add $(v, p_{total})$ to pool with key $p_{total}$
    end
end

$v, p_{total} \leftarrow$ Pop most probable tuple from pool
return $(v, p_{total})$
```

The distribution for Y is also known as Poisson Binomial distribution.

For top-k approximation, we are only interested in evaluating some of the tuples (v, p) in the distribution for Y, it is therefore tempting to express a closed formula for the distribution of Y as a function of v. The closed formula has the following form [47]:

$$P(Y = v) = \sum_{A \in F_v} \prod_{i \in A} p_i \prod_{j \in A^c} \bar{p_j}$$

where F_v is the set of all subsets of $\{1..N\}$ with size v. For example, with $N = 3$ and $v = 2$, we have $F_v = \{\{1,2\},\{1,3\},\{2,3\}\}$. In general,

$$|F_v| = \binom{N}{v}$$

However, $|F_v|$ can be astronomical even for N with moderate size. For example, for $N = 100$, $|F_v|$ can be as large as 100891344545564193334812497256, making it impractical for real-world usage. Fortunately, the computation can be performed more efficiently via dynamic programming.

In fact, Standard DP (Algorithm 3 on page 22) is one of the dynamic programming approaches to tackle the computation efficiently. To demonstrate how Standard DP proceeds, we consider a COUNT node Y with 3 children represented by the random variables X_1, X_2, and X_3:

$$Y = \sum_{i=1}^{3} X_i$$

where $supp(X_i) = \{0, 1\}$ for $i = 1 \ldots 3$, and

$$P(X_i = 0) = 1 - p_i, \quad P(X_i = 1) = p_i$$

$$p_1 = 0.2, \quad p_2 = 0.4, \quad p_3 = 0.6$$

The computation can be visualised via a grid, where the cell (u, v) stores the probability

$$P(\sum_{i=1}^{u} X_i = v)$$

Therefore the cells with $u = 3$ correspond to the distribution for $Y = \sum_{i=1}^{3} X_i$, and the goal is to fill some or all of those cells efficiently.

Standard DP proceeds by initialising the cell $(0, 0)$ to 1. Additionally, we know that $P(\sum_{i=1}^{u} X_i = v) = 0$ for $v > u$ as it is impossible to sum u random variables with a maximum value 1 to a number larger than u. Therefore the cells for $v > u$ are set to 0, resulting in the following grid:

$u = 3$	?	?	?	?
$u = 2$	?	?	?	0
$u = 1$	?	?	0	0
$u = 0$	1	0	0	0
	$v = 0$	$v = 1$	$v = 2$	$v = 3$

In the u^{th} step, Standard DP considers the u^{th} variable X_u, and computes the probability distribution for $\sum_{i=1}^{u} X_i$ by using the probability for $\sum_{i=1}^{u-1} X_i$ computed in the previous step. More specifically, $P(\sum_{i=1}^{u} X_i = v)$ will be computed using the following formula:

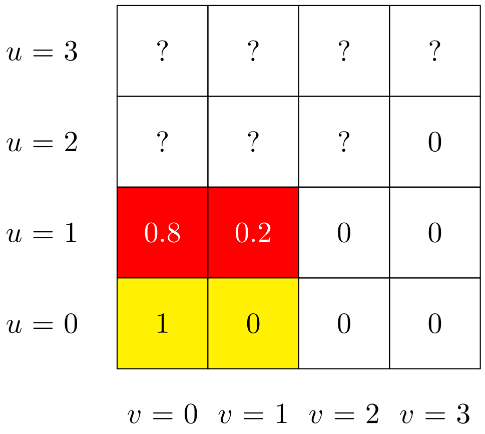
$$P\left(\sum_{i=1}^{u} X_i = v\right) = \begin{cases} P\left(\sum_{i=1}^{u-1} X_i = v\right) \times (1 - p_u) & \text{if } v = 0 \\ P\left(\sum_{i=1}^{u-1} X_i = v - 1\right) \times p_u + P\left(\sum_{i=1}^{u-1} X_i = v\right) \times (1 - p_u) & \text{if } v > 0 \end{cases} \tag{5.7}$$

Equation 5.7 has an intuitive meaning: For $\sum_{i=1}^{u-1} X_i + X_u$ to take the outcome v, then either $\sum_{i=1}^{u-1} X_i$ takes the outcome $v - 1$ and X_u takes the outcome 1, or $\sum_{i=1}^{u-1} X_i$ takes the outcome v and X_u takes the outcome 0.

Therefore the first step involves considering the first random variable X_1 ($p_1 = 0.2$), and computes the values for the cells $(1, 0)$ and $(1, 1)$ using Equation 5.7:

$$\begin{aligned} (1, 0): &\quad 1 \times 0.8 = 0.8 \\ (1, 1): &\quad 1 \times 0.2 + 0 \times 0.8 = 0.2 \end{aligned}$$

The current state of the grid is as follows, where cells that have just been filled in are labelled in red, and cells that have been used for the computation are filled in yellow:



$u = 3$	?	?	?	?
$u = 2$	?	?	?	0
$u = 1$	0.8	0.2	0	0
$u = 0$	1	0	0	0
	$v = 0$	$v = 1$	$v = 2$	$v = 3$

The next step considers the random variable X_2 ($p_2 = 0.4$), and computes the values for the cells $(2, 0)$, $(2, 1)$ and $(2, 2)$ using Equation 5.7:

$$\begin{aligned} (2, 0): &\quad 0.8 \times 0.6 = 0.48 \\ (2, 1): &\quad 0.8 \times 0.4 + 0.2 \times 0.6 = 0.44 \\ (2, 2): &\quad 0.2 \times 0.4 + 0 \times 0.6 = 0.08 \end{aligned}$$

Filling in the newly computed cells leads to the following grid:

$u = 3$	?	?	?	?
$u = 2$	0.48	0.44	0.08	0
$u = 1$	0.8	0.2	0	0
$u = 0$	1	0	0	0
	$v = 0$	$v = 1$	$v = 2$	$v = 3$

Lastly, the random variable X_3 ($p_3 = 0.6$) will be considered in a similar way as in the previous steps, leading to a completely filled grid:

$u = 3$	0.192	0.464	0.296	0.048
$u = 2$	0.48	0.44	0.08	0
$u = 1$	0.8	0.2	0	0
$u = 0$	1	0	0	0
	$v = 0$	$v = 1$	$v = 2$	$v = 3$

At this point, the computation is complete and the entire distribution for $Y = \sum_{i=1}^{3} X_i$ can be read off from the cells in the row $u = 3$. However, the demonstration also reveals that despite we are only interested in some of the tuples in the final distribution, none of the tuples is computed until the very last step using Standard DP.

For this reason, we will adapt Standard DP to compute $P(\sum_{i=1}^{3} X_i = v)$ for arbitrary v by filling in the minimal number of cells in the grid. In particular, based on the initial conditions used by Standard DP and Equations 5.8, we reach at the following recurrence relation for the computation of the cells in the grid:

$$\alpha_{u,v} = \begin{cases} 1 & \text{if } u = 0, v = 0 \\ 0 & \text{if } v > u \\ \alpha_{u-1,v} \times (1 - p_u) & \text{if } u > 0, v = 0 \\ \alpha_{u-1,v-1} \times p_u + \alpha_{u-1,v} \times (1 - p_u) & \text{if } u > 0, 0 < v \leq u \end{cases} \tag{5.8}$$

where $\alpha_{u,v} = P(\sum_{i=1}^{u} X_i = v)$.

Equation 5.8 allows the probability $P(\sum_{i=1}^{N} X_i = V)$ to be computed simply by evaluating $\alpha_{N,V}$, where all the necessary computation for filling the other cells will be invoked by the recurrence. With the computed results from the previous steps stored, the recurrence approach will be at least as efficient as Standard DP, therefore this approach will be used in both computing the next largest tuple and computing the next most probable tuple.

Next Largest Tuple

The largest value in the result of a COUNT node is N. Therefore the algorithm for computing the next largest tuple for a COUNT node will make use of the recurrence formula in Equation 5.8, starting from the cell $\alpha_{N,N}$. Subsequent largest tuples can be retrieved by evaluating $\alpha_{N,N-1}$, $\alpha_{N,N-2}$, and so on using the recurrence relation. The algorithm is presented in Algorithm 31, and has a time complexity $\mathcal{O}(N)$ and space complexity $\mathcal{O}(N)$.

ALGORITHM 31: Get Next Largest Tuple for a COUNT Node

```
Input: COUNT Node node
Output: The Next Largest Tuple ($value, probability$)

NextLargest($node$)
begin
    /* Execute only the first time the method is called */
    if not initialised then
        for $i = 1$ to $N$ do
            $child \leftarrow i^{th}$ child of children of $node$
            $p_i \leftarrow 1.0 - GetNullProb(child)$
        end
        $v \leftarrow N$                    /* The largest tuple has value $N$ */
        $grid_{curr}[0] \leftarrow 1$
        for $i = 1$ to $N$ do
            $grid_{curr}[i] \leftarrow grid_{curr}[i-1] \times p_i$
        end
        $grid_{prev} \leftarrow grid_{curr}$
        $initialised \leftarrow True$
        return $(v, grid_{curr}[N])$
    end

    $v \leftarrow v - 1$                    /* Next Largest Value */
    $grid_{curr} \leftarrow$ Array with Size $N$
    Set every element in $grid_{curr}$ to 0
    for $i = N - v$ to $N$ do
        $grid_{curr}[i] \leftarrow grid_{prev}[i-1] \times (1 - p_i) + grid_{curr}[i-1] \times p_i$
    end
    return $(v, grid_{curr}[N])$
end
```

We will demonstrate Algorithm 31 by computing the 2 largest tuples on the same example we used to demonstrate Standard DP. Specifically, we consider a COUNT node Y with 3 children represented by the random variables X_1, X_2, and X_3:

$$Y = \sum_{i=1}^{3} X_i$$

where $supp(X_i) = \{0, 1\}$ for $i = 1 \ldots 3$, and

$$P(X_i = 0) = 1 - p_i, \quad P(X_i = 1) = p_i$$

$$p_1 = 0.2, \quad p_2 = 0.4, \quad p_3 = 0.6$$

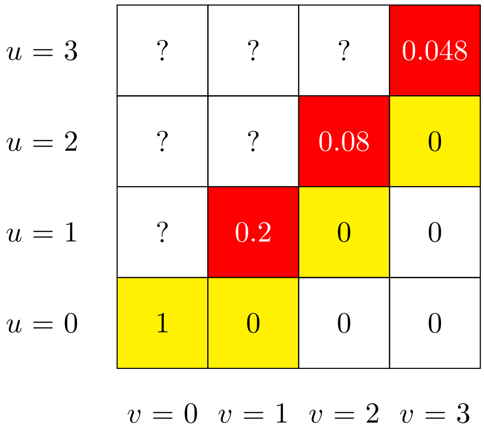
Firstly, the grid is initialised using the first two formulae in Equation 5.8, leading to the following grid:

$u = 3$	?	?	?	?
$u = 2$	?	?	?	0
$u = 1$	?	?	0	0
$u = 0$	1	0	0	0
	$v = 0$	$v = 1$	$v = 2$	$v = 3$

To retrieve the largest tuple, we use the last formula in Equation 5.8 to compute $\alpha_{3,3}$, which involves the computation of the following cells:

$$\begin{aligned}
(1,1): &\quad \alpha_{0,0} \times 0.2 + \alpha_{0,1} \times 0.8 = 1 \times 0.2 + 0 \times 0.8 = 0.2 \\
(2,2): &\quad \alpha_{1,1} \times 0.4 + \alpha_{1,2} \times 0.6 = 0.2 \times 0.4 + 0 \times 0.6 = 0.08 \\
(3,3): &\quad \alpha_{2,2} \times 0.6 + \alpha_{2,3} \times 0.4 = 0.08 \times 0.6 + 0 \times 0.4 = 0.048
\end{aligned}$$

The largest value 3 with probability 0.048 is returned, and the computation in the current step is visualised in the following grid, where the cells that have just been filled in are labelled in red, and the cells that have been used for the computation are labelled in yellow:

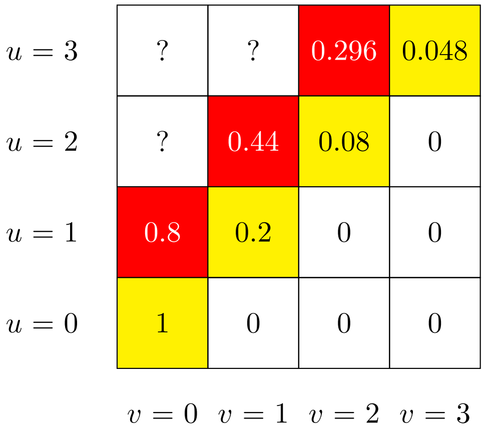


$u = 3$	?	?	?	0.048
$u = 2$	?	?	0.08	0
$u = 1$	?	0.2	0	0
$u = 0$	1	0	0	0
	$v = 0$	$v = 1$	$v = 2$	$v = 3$

To compute the next largest tuple, $\alpha_{3,2}$ needs be evaluated using the recurrence relation in Equation 5.8. Assuming the cells computed in the previous step are stored, the computation of the following extra cells are needed:

$$\begin{aligned}
(1,0): &\quad \alpha_{0,0} \times 0.2 = 1 \times 0.8 = 0.8 \\
(2,1): &\quad \alpha_{1,0} \times 0.4 + \alpha_{1,1} \times 0.6 = 0.8 \times 0.4 + 0.2 \times 0.6 = 0.44 \\
(3,2): &\quad \alpha_{2,1} \times 0.6 + \alpha_{2,2} \times 0.4 = 0.44 \times 0.6 + 0.08 \times 0.4 = 0.296
\end{aligned}$$

At this point, the next largest tuple with value 2 and probability 0.296 is returned, and the computation for this step can be visualised in the following grid:



$u=3$	?	?	0.296	0.048
$u=2$	?	0.44	0.08	0
$u=1$	0.8	0.2	0	0
$u=0$	1	0	0	0
	$v=0$	$v=1$	$v=2$	$v=3$

Next Most Probable Tuples

While the recurrence relation in Equation 5.8 enables us to compute $P(\sum_{i=1}^{N} X_i = v)$ efficiently for arbitrary value v, we first need to find out the value with the next highest probability in the distribution $\sum_{i=1}^{N} X_i$. This can be done by using Proposition 9, where the proof can be found in Samuels [41].

Proposition 9. *The distribution for $\sum_{i=1}^{N} X_i$, where $supp(X_i) = \{0,1\}$ for $i = 1 \ldots N$, is bell-shaped, and the most probable value in the distribution is in the pair of integers nearest to $\mu = \sum_{i=1}^{N} p_i$.*

With Proposition 9, the most probable value in the distribution is restricted to $\lfloor \mu \rfloor$ and $\lceil \mu \rceil$. The recurrence relation in Equation 5.8 can then be used to evaluate $\alpha_{N,\lfloor \mu \rfloor}$ and $\alpha_{N,\lceil \mu \rceil}$, which correspond to the probabilities for $P(\sum_{i=1}^{N} X_i = \lfloor \mu \rfloor)$ and $P(\sum_{i=1}^{N} X_i = \lceil \mu \rceil)$ respectively. The one with a higher probability can then be returned as the most probable value. Because Proposition 9 also states that the distribution is bell-shaped, therefore the next most probable value will be restricted to the pair of integers that are closest to the most probable value but haven't been returned, and their probabilities can once again be computed using the recurrence relation. The method is referred as the Grid Method, and is presented in Algorithm 32.

Using Standard DP on a COUNT node with N children requires $\frac{(N+1)(N+2)}{2}$ cells in the grid to be filled before any tuple in the final distribution can be extracted, but the number of cells

ALGORITHM 32: Get Next Most Probable Tuple for a COUNT Node - Grid Method (GM)

Input: COUNT Node *node*

Output: The Next Top Tuple (*value*, *probability*)

```
NextTop(node)
begin
    /* Execute only the first time the method is called */
    if not initialised then
        for $i = 1$ to $N$ do
            $child \leftarrow i^{th}$ child of children of $node$
            $p_i \leftarrow 1.0 - GetNullProb(child)$
        end
        $\mu \leftarrow \sum_{i=1}^{N} p_i$
        $v_{left}, v_{right} \leftarrow floor(\mu), ceiling(\mu)$
        $grid \leftarrow$ Two dimensional array of size $(N+1) \times (N+1)$ filled with $-1$
        $initialised \leftarrow True$
    end

    $p_{left}, p_{right} \leftarrow GetGrid(grid, N, v_{left}), GetGrid(grid, N, v_{right})$
    if $p_{left} \leq p_{right}$ then
        $tuple \leftarrow (v_{right}, p_{right})$
        $v_{right} \leftarrow v_{right} + 1$
        return $tuple$
    else
        $tuple \leftarrow (v_{left}, p_{left})$
        $v_{left} \leftarrow v_{left} - 1$
        return $tuple$
    end
end

GetGrid(grid, u, v)
begin
    /* Dynamic Programming - Return the computed results directly */
    if $grid[u][v] \geq 0$ then
        return $grid[u][v]$
    end

    /* Recurrence Relation in Equation 5.8 */
    if $u = 0$ and $v = 0$ then
        $grid[u][v] \leftarrow 1$
    else if $v > u$ then
        $grid[u][v] \leftarrow 0$
    else if $v = 0$ then
        $grid[u][v] \leftarrow GetGrid(grid, u-1, v) \times (1 - p_u)$
    else
        $grid[u][v] \leftarrow GetGrid(grid, u-1, v-1) \times p_u + GetGrid(grid, u-1, v) \times (1 - p_u)$
    end

    return $grid[u][v]$
end
```

to be filled to extract the most probable tuple (i.e. top-1) using the Grid Method depends on value for μ. The percentage of cells to be filled to extract the most probable tuple using the Grid Method compared to Standard DP is presented in Figure 5.1, where the result suggests up to 2× speedup can be achieved using the Grid Method for $\mu = N/2$, and even better performance can be attained as μ moves away from $N/2$. The result suggests the Grid Method has a time and space complexity of $O\left(N^2\right)$, but provides multiple performance speedup compared to Standard DP for the computation of the most probable tuple. Once the most probable tuple is computed, computation for subsequent most probable tuples can be done in time $O\left(N\right)$.

On the other hand, the Recursive FFT Algorithm (Algorithm 19 on page 49) is also applicable in the current situation, with a lower time complexity $\mathcal{O}(N(\log N)^2)$ and space complexity $\mathcal{O}(N)$, making it more scalable than the Grid Method. However, the Recursive FFT Algorithm computes the entire probability distribution instead taking advantage of top-k approximation, where only some of the tuples in the distribution are required, therefore it is unclear whether the Grid Method or Recursive FFT Algorithm will provide a better performance for typical range of N. The performance of the Recursive FFT Algorithm and the Grid Method will be compared in Chapter 7.

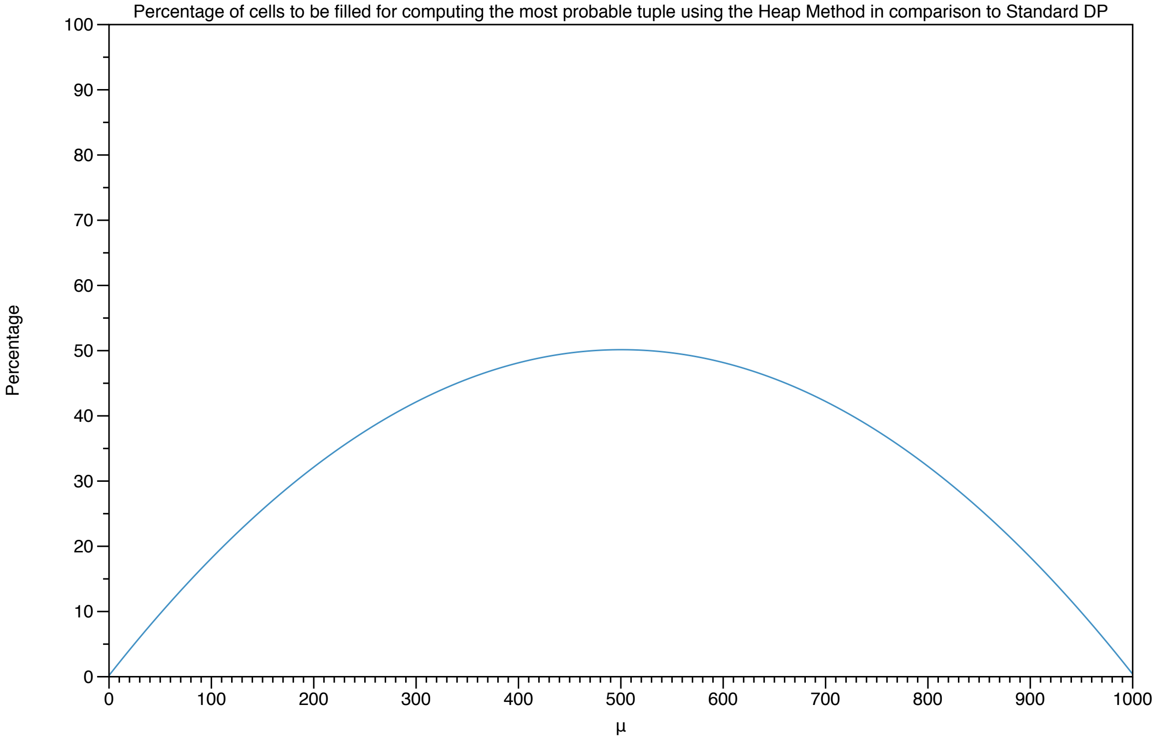


Figure 5.1: Percentage of the grid to be filled to compute the probability of the most probable value as μ varies ($N = 1000$)

We continue to use the same example to demonstrate the computation for top-2 most probable tuples with the Grid Method. Specifically, we consider a COUNT node Y with 3 children represented by the random variables X_1, X_2, and X_3:

$$Y = \sum_{i=1}^{3} X_i$$

where $supp(X_i) = \{0, 1\}$ for $i = 1 \ldots 3$, and

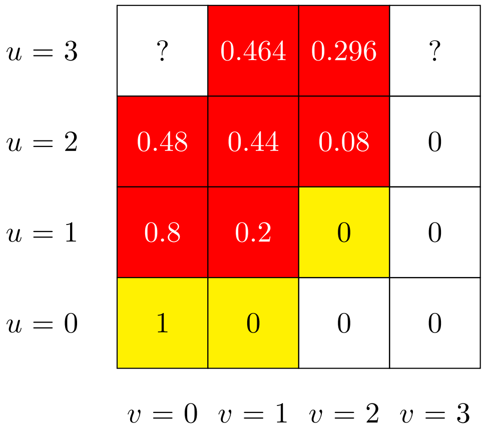
$$P(X_i = 0) = 1 - p_i, \quad P(X_i = 1) = p_i$$

$$p_1 = 0.2, \quad p_2 = 0.4, \quad p_3 = 0.6$$

Firstly, the grid is initialised using the same initialisation condition as in Standard DP, where the first two formulae for the recurrence relation (Equation 5.8) are used, resulting in the following grid:

	$v = 0$	$v = 1$	$v = 2$	$v = 3$
$u = 3$	?	?	?	?
$u = 2$	?	?	?	0
$u = 1$	?	?	0	0
$u = 0$	1	0	0	0

Because $\mu = 0.2 + 0.4 + 0.6 = 1.2$, therefore the cell $(3, 1)$ and cell $(3, 2)$ will be computed using the recurrence relation, resulting in values 0.464 and 0.296 respectively. The result shows that the tuple $(1, 0.464)$ is the most probable tuple in the distribution and should be returned. The computed result is visualised in the following grid, where the cells that have just been computed are labelled in red, and the cells that have been used for the computation are labelled in yellow:



	$v = 0$	$v = 1$	$v = 2$	$v = 3$
$u = 3$	?	0.464	0.296	?
$u = 2$	0.48	0.44	0.08	0
$u = 1$	0.8	0.2	0	0
$u = 0$	1	0	0	0

Proposition 9 states that the distribution is bell-shaped. As the most probable value we just computed is 1, the next most probable value must either be 0 or 2. Because we have already computed cell $(3, 2)$ in the previous step, the only computation that needs to be done is for the cell $(3, 0)$, which has a value 0.192. The result suggests that the next most probable tuple is $(2, 0.296)$, and the computation is done at this point. The grid for the current step is as follows:

$u = 3$	0.192	~~0.464~~	0.296	?
$u = 2$	0.48	0.44	0.08	0
$u = 1$	0.8	0.2	0	0
$u = 0$	1	0	0	0
	$v = 0$	$v = 1$	$v = 2$	$v = 3$

5.2.5 SUM

For a SUM node Y with N children, which can be represented by the independent and non-identically distributed random variables $X_1, X_2, \ldots, X_N$, the computation for the SUM node can be defined as the SUM convolution of the random variables:

$$Y = \sum_{i=1}^{N} X_i \tag{5.9}$$

Top-k approximation for a SUM node Y involves computing $P(Y = v)$ for some values v. However, the computation for $P(Y = v)$ can be reduced into the sub-set sum problem, which is a well-known NP-hard problem, suggesting any attempt to compute the probability of a particular tuple will already take exponential time. We will therefore tackle the problem using dynamic programming to compute the probabilities of all the tuples at once with a pseudo-polynomial time, where the most probable tuples can then be extracted and returned.

Clearly, Standard DP (Algorithm 3 on page 22) is one of those approaches with a complexity $\mathcal{O}((NR)^2)$, where R is the range (the difference between the least and largest values of the distribution) of the distributions. However, the quadratic complexity makes Standard DP inadmissible when N becomes larger. Fortunately, the Recursive FFT Algorithm (Algorithm 19 on page 49) we proposed earlier is also applicable for the top-k approximation on SUM nodes, with a lower complexity $\mathcal{O}(NR\log(NR)\log(N))$.

Granted, the Recursive FFT Algorithm can only be used to compute of the full distribution, which implies there will be no time saving due to the top-k approximation setting for SUM nodes. Nonetheless, because SUM can produce a support that is exponential to N, there could be a significant time saving for other nodes in the d-tree as those nodes will only need to consume the most probable tuples from the SUM nodes instead of the entire exponentially sized distributions.

Chapter 6

Implementation

6.1 System Overview

The proposed framework has been built using Python, supporting exact evaluation, histogram approximation and top-k approximation for MIN, MAX, COUNT, and SUM aggregations. Python has the advantage of being concise and clear, allowing readers to capture the essence of the implementation without getting into the language specifics. Additionally, the versatile nature of Python allows new approximation algorithms to be implemented and benchmark against the state-of-the-art algorithms readily. The system has been designed to be very modular and flexible to support future development, even to the extent of adding support for completely new types of approximations.

Granted, Python is known to be slower than some other languages, such as C, but the language speed does not affect the relative performance between the proposed algorithms and the state-of-the-art algorithms, which is the essence of the dissertation. However, it is important for all the implementations to be written in native Python for fair comparison. For example, while the python package *scipy* provides an implementation of FFT in C, we have reimplemented the algorithm in native Python to ensure the comparison is language agnostic. While the system uses external packages with a C backend such as *numpy* and *scipy*, they are only used in places where they will not affect the benchmark result, such as random data generation. This ensures similar performance speedup will be achieved when the algorithms are ported into any other probabilistic database management system, possibly implemented using other languages.

6.1.1 Code Organisation

The system consists of multiple packages serving different purposes. We will brief the details of each of the packages in this section.

- **debug**
  - The debugging package ensures correctness of the implementation by providing facilities to compare the result of exact evaluation with histogram approximation and top-k approximation. Specifically, the probability distribution obtained from

exact evaluation will be converted into its histogram representation to compare with the result of histogram approximation, and the top-k most probable tuples will be extracted from the distribution for comparing with the result of top-k approximation.

  - The system has been tested with this package using thousands of randomly generated decomposition trees to ensure its correctness.

- **dtree**
  - The dtree package contains the data structure to represent decomposition trees, which contains the pointer to the root node of the decomposition tree as well as methods for loading and exporting the decomposition tree from and to disk.
  - The data structure also serves as the starting point of the evaluation, where it performs all the necessary preprocessing, such as deriving the bin intervals, before delegating the evaluation to the nodes in the tree recursively.
- **generators**
  - The generators package provides facilities for generating decomposition trees using randomly generated random variables, according to the specification provided by the user.
  - Details of random decomposition tree generation can be found in Section 6.2.3.
- **nodes**
  - The nodes package provides the algorithms for exact evaluation, histogram approximation, and top-k approximation as well as all the auxiliary methods for different node types.
  - The node interface is also defined in the package, where the implementation of different node types must conform to. Details of the node interface can be found in Section 6.2.4.
  - Implemented node types include MIN, MAX, COUNT, SUM, UNION, VARIABLE, and PROD, which are the set of node types that could exist in the d-tree for MIN, MAX, COUNT, or SUM aggregations.
  - The proposed algorithms are then implemented as a method of the corresponding node type.
- **shared**
  - Containing data structures that are used across the entire framework.
  - Including the data structure for representing probability distributions, histograms and probability bounds.
- **utilities**
  - The package provides supporting facilities that are used across the entire framework.
  - Examples include an implementation of minimum/maximum heap, timing facilities for benchmarking, and facilities for outputting result in a beautiful manner.

6.2 Implementation Details

While the details of the algorithms have been presented in the previous chapters, we will discuss some of the implementation decisions in this section.

6.2.1 Handling Null

Despite we have made the design decision of representing null using a seemingly ordinary integer 0, it is important to handle null carefully as its true meaning changes with the neutral element 0_M of the node type under consideration, as depicted in the following table:

Node Type	0_M
MIN	∞
MAX	$-\infty$
COUNT	0
SUM	0

For example, 0 needs to be interpreted as ∞ in a MIN node, which can have a significant affect on the correctness of the result as 0 and ∞ are on the opposite ends of the number line.

While it is possible to work around this issue by careful book keeping in exact evaluation and top-k approximation, such as casting the value 0 retrieved from the children to ∞ when the current node type is MIN, things could get more complicated in histogram approximation. For example, there is no trivial way to extract the null probability from the bin $[0, 5]$, which is essential for casting the value 0 to ∞. In sight of this, null probability will be treated specially in histogram approximation. In particular, it is important that 0 does not share the same bin with other values in the histograms. This can be achieved by ensuring that 0 is excluded from the histograms completely, and null probability can then be computed exactly alongside histogram approximation. Luckily, because the computation of the null probabilities for MIN, MAX, COUNT, and SUM nodes is as simple as computing the product of the null probabilities of their children, this can be done efficiently according to Algorithm 33.

ALGORITHM 33: Compute Null Probability for a MIN/MAX/COUNT/SUM Node

Input: MIN/MAX/COUNT/SUM Node *node*
Output: Null Probability *prob*

```
GetNullProb($node$)
begin
    $prob \leftarrow 1.0$
    foreach $child \in$ children of node do
        $prob \leftarrow prob \times GetNullProb(child)$
    end
    return $prob$
end
```

6.2.2 Data Structures

Probability Distributions

Because the proposed algorithms involve only sequential access of probability distributions, the most efficient way to represent a probability distribution is to store the tuples (*value*, *probability*) directly in a list. Additionally, null probability will also be stored separately to enable efficient null probability computation. We will assume the probability distributions stored in the database are sorted by value, which is the most natural order.

Histogram

A histogram is defined by the intervals of the bins and the corresponding probabilities. Because there should be no gap between adjacent bins, one only needs to store the left boundary of the bin intervals in a list. The corresponding probabilities can then be stored in a separate list. As discussed in the Section 6.2.1, the null value 0 will be excluded from the histogram, therefore the null probability will be stored separately.

On top of its succinctness, the representation also has the advantage of supporting binary search for identifying the bin a particular value belongs to readily in $\mathcal{O}(\log B)$ time, where B is the number of bins in the histogram.

Example 9. *Consider a histogram with the following bin intervals and probabilities:*

Lower Boundary	*Upper Boundary*	*Probability*
1	*5*	*0.3*
6	*10*	*0.2*
11	*20*	*0.4*
21	*100*	*0.1*

It can be represented succinctly using two lists:

```
intervals = [1, 6, 11, 21, 101]
probabilities = [0.3, 0.2, 0.4, 0.1]
```

Probability Bounds

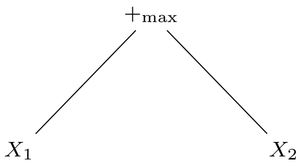
Support for probability bounds in the system can be accomplished without changing the underlying implementation of the algorithms by creating a class for probability bounds that overrides the $+$, $-$ and $*$ operator according to the algorithm proposed in Section 3.3.1 (page 29). By ensuring the probability bounds have the same interface as exact probabilities (floating points), the implementations can be agnostic to the nature of the probability being worked on, and let the overridden methods take care of arithmetic of probability bounds.

6.2.3 Random Data Generation

To benchmark the algorithms for different types of aggregations, it is important for the system to be capable of generating decomposition trees with arbitrary structures. For this reason, the system has a built-in tree generator, where the user can specify the type of nodes in each level of the tree along with the specifics of the probability distributions for the random variables, such as the distribution skewness or the range of values in the distribution. The system will then produce a decomposition tree conforming to the instructions using randomly generated random variables. The generated decomposition tree will be exported to disk in the form of JSON, which allows the user to visualise the decomposition tree using any existing JSON parser, or even fine tune the tree by changing the values in the JSON. The JSON can lastly be loaded back to the system for benchmarking.[1]

The JSON represents the tree recursively by storing each node in the form the triplet `[NodeType, Children, Others]`. *Children* is a list of triplets, each of which is a child of the node; *Others* is a dictionary, representing the specific details of the node.

Example 10. *Consider a decomposition tree with the following form*



where X_1 and X_2 are random variables with probability distributions

X_1	
Value	*Probability*
0	*0.8*
1	*0.2*

X_2	
Value	*Probability*
0	*0.4*
1	*0.6*

The decomposition tree will have the following JSON representation when exported to disk:

```
[
    "MAX",
    [
        [
            "VARIABLE", [], {"pdf": [[0, 0.8], [1, 0.2]]}
        ],
        [
            "VARIABLE", [], {"pdf": [[0, 0.4], [1, 0.6]]}
        ]
    ],
    {}
]
```

[1] The exported tree is also important for rerunning an experiment if any anomaly is observed.

6.2.4 Node Interface

Performing evaluation for a decomposition tree boils down to performing the same evaluation in each of its nodes (and hence the corresponding subtrees) in a recursive manner, therefore a common interface across all types of nodes is necessary. Base on all the discussions in the dissertation, the node interface is summarised in the following table:

Node Interface

Method	Description
Exact Evaluation	
ExactEvaluation	Return the result of exact evaluation for the subtree
Histogram Approximation	
HistogramEvaluation	Return the result of histogram approximation for the subtree
GetLeastVal	Return the least value in the distribution for the subtree
GetLargestVal	Return the largest value in the distribution for the subtree
Top-k Approximation	
NextTop	Return the next most probable tuple for the subtree
NextLeast	Return the next least tuple for the subtree
NextLargest	Return the next largest tuple for the subtree
Others	
GetNullProb	Return the null probability for the subtree
GetMean	Return the mean of the distribution for the subtree
GetVariance	Return the variance of the distribution for the subtree
GetThirdMoment	Return the third moment of the distribution for the subtree

Chapter 7

Experiments

In preparing this dissertation, we benchmarked the performance of each of the proposed algorithms using semimodule expressions corresponding to the different types of aggregate queries. Our experimental results provide solid evidence of the advantages of employing histogram approximation and top-k approximation in lieu of exact evaluation; namely, an observed performance speedup of over two orders of magnitude beyond the state-of-the-art algorithms for exact evaluation. The experimental results will be presented in this chapter.

7.1 Experimental Setup

7.1.1 Methodology

In the experiments described in this chapter, we investigate the following aspects of the proposed algorithms:

Scalability, or the performance of the algorithms as the number of tuples to be aggregated increases.

Dependency, or the effect on algorithm performance as correlations are introduced between the tuples.

Skewness, or the effect on algorithm performance as the tuples are annotated by skewed variables.

Accuracy of the algorithms in producing probability-based approximations.

These investigations were carried out by designing a set of similar semimodule expressions, identical in all but one aspect. For example, in the experiment investigating the scalability of the algorithms, we used a set of semimodule expressions of varying size but identical in all other ways. Each semimodule expression was then compiled into the corresponding decomposition tree (d-tree) such that the random variables in the d-tree is represented by randomly generated probability distributions.

Different algorithms were then used to evaluate the d-tree into a full probability distribution (exact evaluation), a histogram representation of the distribution (histogram approximation), or a representation of the most probable tuples in the distribution (top-k approximation). The wall clock time for running each algorithm was recorded.

To minimise the effect of runtime fluctuations on the experimental results, each experiment was repeated for at least 10 times (up to 50 times if the runtime of each trial is relatively short), and the average wall clock time over all trials was reported. Because this dissertation focuses on improving Standard DP (Algorithm 3 on page 22; the state-of-the-art algorithm proposed by Fink et al. [19] for exact evaluation), it was sometimes more natural to present the results in the form of performance speedup relative to Standard DP, a metric that will be referred to hereafter as the *benchmark score*:

$$\text{Benchmark Score} = \frac{\text{Wall Clock time for Standard DP}}{\text{Wall Clock Time of Target Algorithm}}$$

7.1.2 Environment

All experiments were conducted in the following environment:

CPU: 2.6 GHz Intel Quad Core i7
RAM: 16 GB
OS: Mac OS 10.8.4 Mountain Lion 64bit
Version: Python 2.7.5 using cPython with GCC 4.2.1
Packages: Numpy 1.7.1, Scipy 0.12.0

7.1.3 Algorithms Benchmarked

The following table provides a summary of the algorithms proposed in the dissertation for benchmarking:

Name	Aggregation Type	Reference
Histogram Approximation		
BCA: Bin Convolution Algorithm	MIN/MAX	Algorithm 15 (Page 40)
NA: Normal Approximation Algorithm	COUNT SUM	Algorithm 17 (Page 44) Algorithm 21 (Page 52)
DBA: Deferred Binning Algorithm	COUNT/SUM	Algorithm 18 (Page 45)
EBA: Early Binning Algorithm	SUM	Algorithm 22 (Page 54)
FFT: Recursive FFT Algorithm	COUNT/SUM	Algorithm 19 (Page 49)
Top-k Approximation		
EHM: Extended Heap Method	MIN/MAX	Algorithm 30 (Page 69)
GM: Grid Method	COUNT	Algorithm 32 (Page 76)
FFT: Recursive FFT Algorithm	COUNT/SUM	Algorithm 19 (Page 49)

The other algorithms proposed were all auxiliary algorithms; thus, they were not benchmarked directly. Because of the importance of Shannon expansion in the proposed framework, it is worth noting that the Bin Union Algorithm (BUA; Algorithm 14 on page 37) for histogram approximation and the Threshold Algorithm (TA; Algorithm 28 on page 63) for top-k approximation were benchmarked indirectly in experiments involving UNION nodes.

7.1.4 Semimodule Expressions for the Experiments

We will now define two types of queries (and thus their equivalent semimodule expressions) that were used across multiple experiments. Other types of semimodule expressions that were used in only one experiment will be introduced along with that experiment.

Type I

Type I queries correspond to aggregate queries over N independent tuples. The semimodule expression has the following form:

$$(s_1 \otimes m_1) + (s_2 \otimes m_2) + \ldots + (s_N \otimes m_N) \tag{7.1}$$

where

N corresponds to the size of the semimodule expression.

Semiring expression s_i is a random variable with support $\{0, 1\}$, where $P(s_i = 1)$ is randomly generated float in the range $[0, 1]$.

Monoid expression m_i is a randomly generated integer ranging from 1 to R.

$+$ can either be $+_{\text{MIN}}$, $+_{\text{MAX}}$, $+_{\text{COUNT}}$ or $+_{\text{SUM}}$

Compiling the semimodule expression in Equation 7.1 yields the (flattened) d-tree in Figure 7.1.

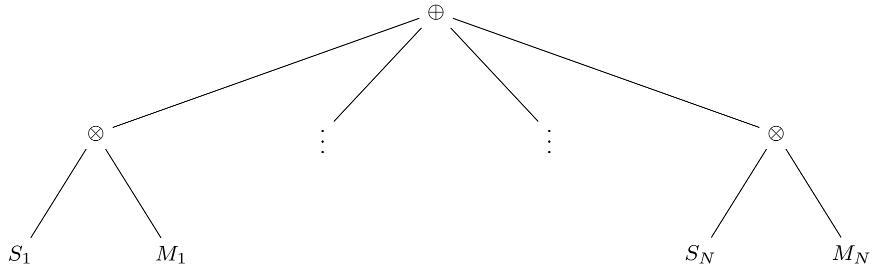


Figure 7.1: Decomposition tree for a type I semimodule expression

Type II

Type II queries correspond to aggregate queries over N correlated tuples; The semimodule expression has the following form:

$$(s_1 \otimes m_1) + (s_2 \otimes m_2) + \ldots + (s_N \otimes m_N) \tag{7.2}$$

where

N corresponds to the size of the semimodule expression.

Semiring expression s_i has the form

$$s_i = \sum_{j=0}^{D} (x = j) y_{i,j}$$

x is a random variable with support $\{0, \ldots, D\}$, and $y_{i,j}$ are independent random variables with support $\{0, 1\}$. All tuples are therefore dependent because they are all correlated with x, and one can control the depth of dependency between the tuples by adjusting D, where $D = 0$ implies no dependency (as s_i becomes $(x = 0)y_{i,0} = (0 = 0)y_{i,0} = y_{i,0}$), and larger D implies more dependency.

Monoid expression m_i is a randomly generated integer ranging from 1 to R.

$+$ can either be $+_{\text{MIN}}$, $+_{\text{MAX}}$, $+_{\text{COUNT}}$ or $+_{\text{SUM}}$

Compiling the semimodule expression returns the d-tree depicted in Figure 7.2.

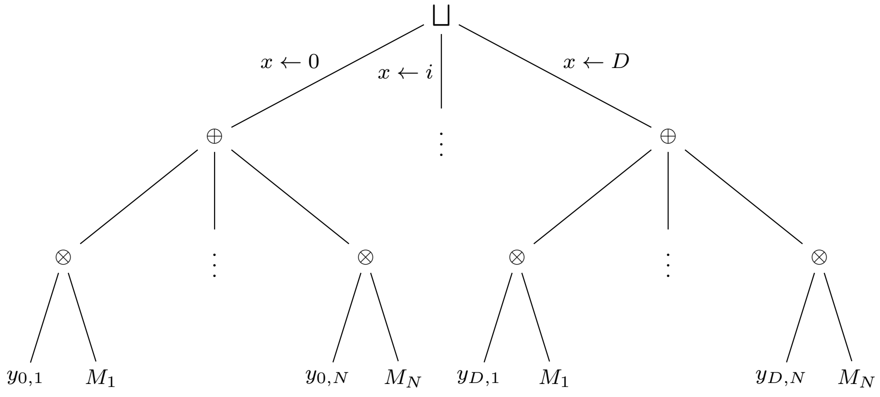


Figure 7.2: Decomposition tree for a type II semimodule expression

7.2 Summary of Experimental Findings

Figure 7.3 provides an overview of the performance compared to Standard DP (Algorithm 3 on page 22) when the number of tuples to be aggregated is on the order of 10,000. The algorithms recommended for deployment in real-world systems are highlighted in red. While some of the other algorithms might provide better performance under special circumstances (such as when the probability distribution of the input variables is highly skewed), the experimental results suggest that the highlighted algorithms provide the best performance in most settings.

For histogram approximation, the results are clearly promising, with a speedup of over two orders of magnitude measured across all types of aggregations. More importantly, all of the highlighted algorithms demonstrate better scalability than exact evaluation. While the recommended algorithms for COUNT and SUM aggregations compute histograms with

Histogram Approximation

Name	Performance	Note
MIN/MAX		
BCA: Bin Convolution Algorithm	300×	
COUNT		
NA: Normal Approximation Algorithm	240×	Accuracy over 99%
FFT: Recursive FFT Algorithm	15×	
DBA: Deferred Binning Algorithm	4×	
SUM		
NA: Normal Approximation Algorithm	630×	Accuracy over 95%
FFT: Recursive FFT Algorithm	15×	
DBA: Deferred Binning Algorithm	2×	
EBA: Early Binning Algorithm	2×	Up to 10× for histogram zooming

Top-k Approximation

Name	Performance	Note
MIN/MAX		
EHM: Extended Heap Method	350×	
COUNT		
FFT: Recursive FFT Algorithm	15×	
GM: Grid Method	5×	Up to 15× for skewed variables
SUM		
FFT: Recursive FFT Algorithm	15×	

Figure 7.3: Performance for the proposed algorithms compared to exact evaluation using Standard DP

approximate probabilities, the accuracy of those approximate probabilities is generally very high (over 95%). Thus, those probabilities, when taken together with the lower and upper probability bounds, generally provide a confidence level high enough for real-world applications.

For top-k approximation, the performance across different algorithms varies. With a speedup of over two orders of magnitudes speedup measured for MIN/MAX aggregations, the algorithms for top-k approximation provide a speedup only 10 times faster than exact evaluation using Standard DP for COUNT and SUM aggregations. While all of the highlighted algorithms demonstrate better scalability than Standard DP, there is still a clear advantage to using histogram approximation over top-k approximation for COUNT and SUM aggregations.

Additionally, the experimental results confirms that the introduction of dependency between tuples has little effect on the performance speedup of the proposed algorithms over exact computation. The introduction of dependency does naturally increase the runtime of the algorithms, however, as it introduces more nodes into the d-tree to resolve the dependency.

7.3 Histogram Approximation

7.3.1 Scalability

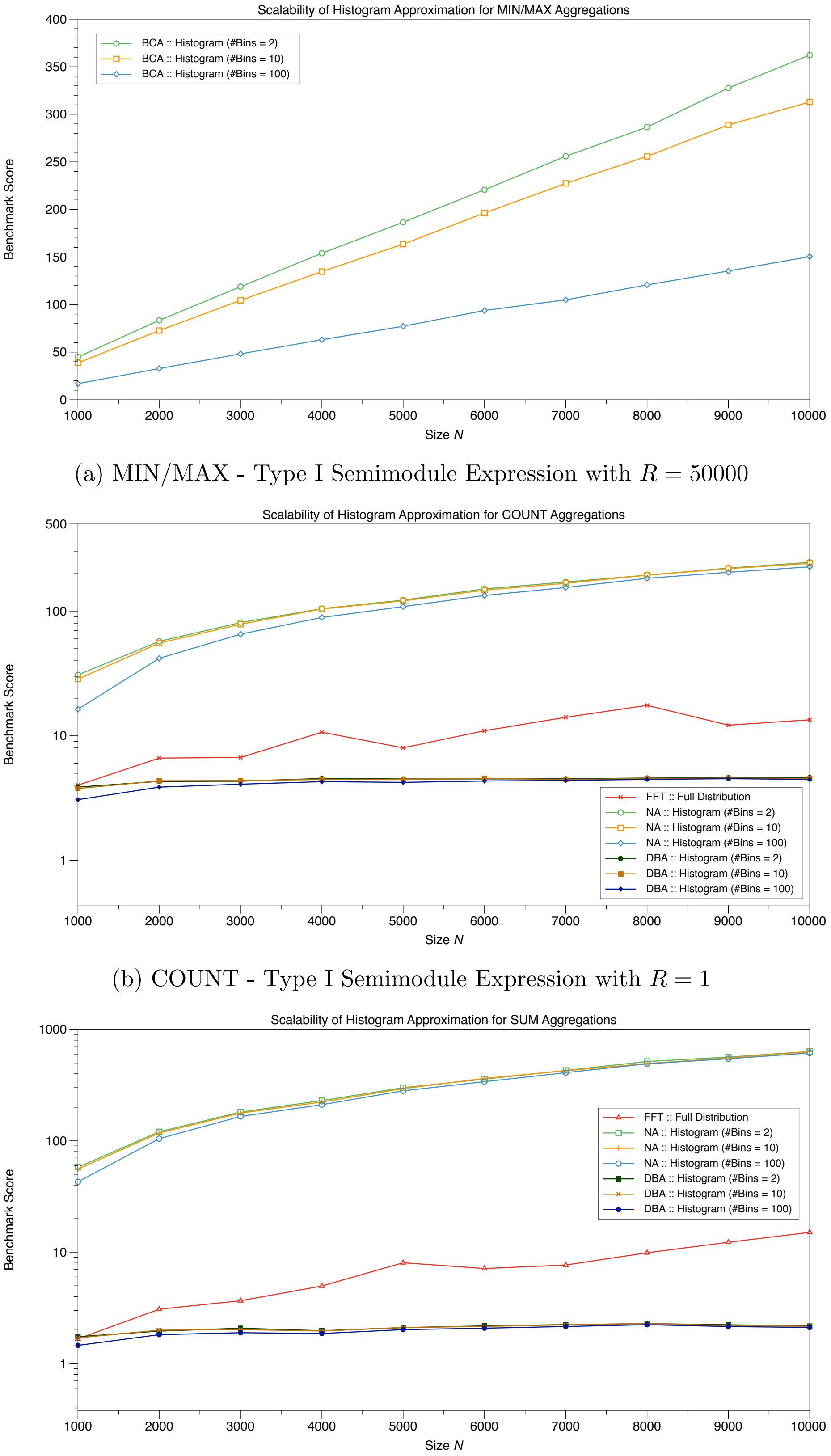
The first experiment uses a Type I semimodule expression (Equation 7.1) and investigates the performance of the algorithms as the size of the expression N increases. The result is presented in Figure 7.4.

Figure 7.4a demonstrates the clear advantage of using histogram approximation for MIN/-MAX aggregations; the benchmark scores suggests that over two orders of magnitude in performance speedup can be achieved. Most importantly, the benchmark scores increase with N, which is an indication that the algorithm scales better than Standard DP. The result reflects the lower complexity $\mathcal{O}(NB)$ of the Bin Convolution Algorithm (BCA) relative to Standard DP $\mathcal{O}((NM)^2)$, where B is the number of bins for the target histogram and M is the support size of the random variables (in this experiment, $M = 1$).

Both the COUNT and SUM aggregations demonstrate similar behaviours due to the similarity of the algorithms used. In both cases, the Normal Approximation Algorithm (NA) demonstrates the best performance and scalability, which reflects its low complexity: $\mathcal{O}(N)$. Additionally, a higher benchmark score is observed for SUM aggregations than for COUNT aggregations, which is due to the higher complexity of using Standard DP on SUM aggregations than COUNT aggregations: $\mathcal{O}((NR)^2)$ vs $\mathcal{O}(N^2)$, where R is the range of values to be aggregated. It should be noted that Normal Approximation Algorithm produces histograms with approximate probabilities, the accuracy of which will be investigated in Section 7.3.4.

On the other hand, both the Deferred Binning Algorithm (DBA) and the Recursive FFT Algorithm (FFT) can be used to compute histograms with exact probabilities (the Recursive FFT Algorithm returns a full probability distribution, which can then be turned into a histogram without a significant impact on the benchmark score). Although the Deferred Binning Algorithm exploits the approximation settings for the computation, it does not provide significantly better performance than the Recursive FFT Algorithm even when N is relatively small. Most importantly, the constant benchmark score suggests that the Deferred Binning Algorithm is as scalable as Standard DP, while the Recursive FFT Algorithm shows better scalability.

At any rate, the Recursive FFT Algorithm is still one order of magnitude slower than the Normal Approximation Algorithm. Thus, the Normal Approximation Algorithm is the



(a) MIN/MAX - Type I Semimodule Expression with $R = 50000$

(b) COUNT - Type I Semimodule Expression with $R = 1$

(c) SUM - Type I Semimodule Expression with $R = 10$

Figure 7.4: Scalability of Histogram Approximation

preferred algorithm for histogram approximation on COUNT and SUM aggregations unless it is crucial to return a histogram with exact probabilities, in which case the Recursive FFT Algorithm should be used instead.

7.3.2 Dependency

One of the most powerful features of pc-tables and pvc-tables is their ability to represent arbitrary correlations between tuples; it is therefore important for histogram approximation techniques to be efficient in handling not only independent tuples, but also correlated tuples. Technically, the introduction of dependency between tuples introduces Shannon expansion in the form of a UNION node in the d-tree.

In this experiment, we used a Type II semimodule expression (Equation 7.2) with varying D, which corresponds to an aggregate query over correlated tuples with varying dependency. The result is presented in Figure 7.5.

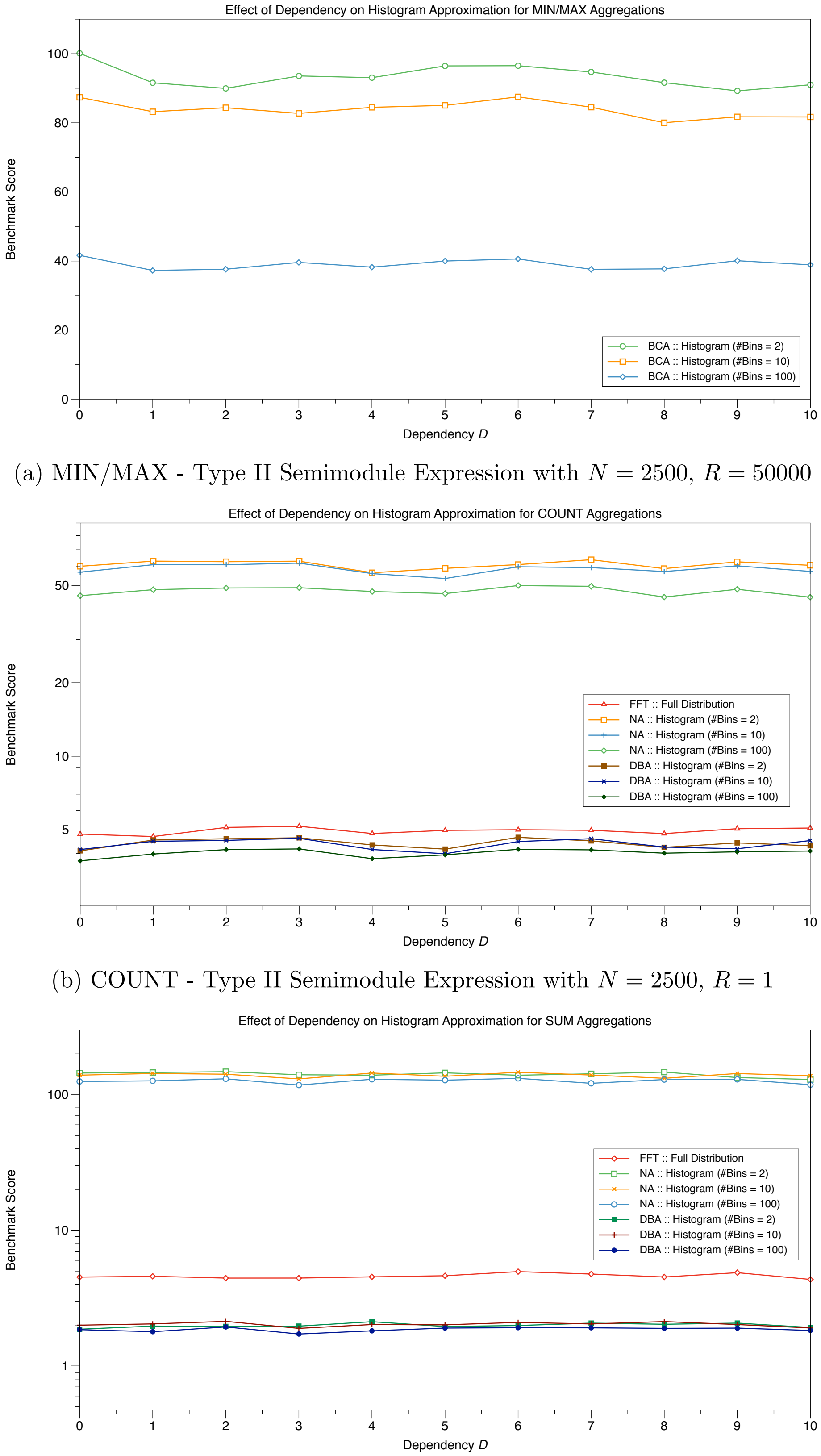
The fluctuation in the benchmark scores as D, the depth of dependency between tuples, varies is relatively small across all results for MIN, MAX, COUNT, and SUM aggregations. This indicates that the introduction of dependency between the tuples has little effect on the performance speedup of the proposed algorithms. Intuitively, this is because the computation for a UNION node is relatively inexpensive. The bottleneck of the evaluation still lies on the convolution nodes; thus, the introduction of the UNION nodes has little effect on the benchmark scores.

It should be noted, however, that when greater dependency is introduced between the tuples, more nodes must be added to the d-tree in order to resolve the dependency. For this reason, the actual runtime of the algorithm increases with D. The runtime for the algorithms increases at the same rate as that for Standard DP, however, thereby keeping the benchmark scores constant.

7.3.3 Performance of Histogram Zooming

Behind the scenes, histogram zooming is really just regular histogram computation with specific bin intervals. Because the performance of most of the algorithms proposed is dependent on only the number of bins and not the intervals of the bins themselves, the performance of histogram zooming is very similar to that of regular histogram computation. The only exception is the Early Binning Algorithm (EBA) for SUM aggregations, which is designed to exploit the irregularity of the bin intervals to put values into the correct bin as early as possible; thus, that algorithm will perform differently depending on the details of zooming.

To benchmark the performance of the Early Binning Algorithm when zooming into different regions of a histogram, we conducted an experiment based on the histogram in Figure 7.6. That histogram is the evaluation result of a Type I semimodule expression (Equation 7.1) with $N = 200, R = 1000$ over SUM aggregations. The experiment proceeded by zooming into different parts of the histogram, and we report the benchmark score of the algorithm in Figure 7.7.



(a) MIN/MAX - Type II Semimodule Expression with $N = 2500,\ R = 50000$

(b) COUNT - Type II Semimodule Expression with $N = 2500,\ R = 1$

(c) SUM - Type II Semimodule Expression with $N = 2500,\ R = 10$

Figure 7.5: Effect of Dependency on Histogram Approximation

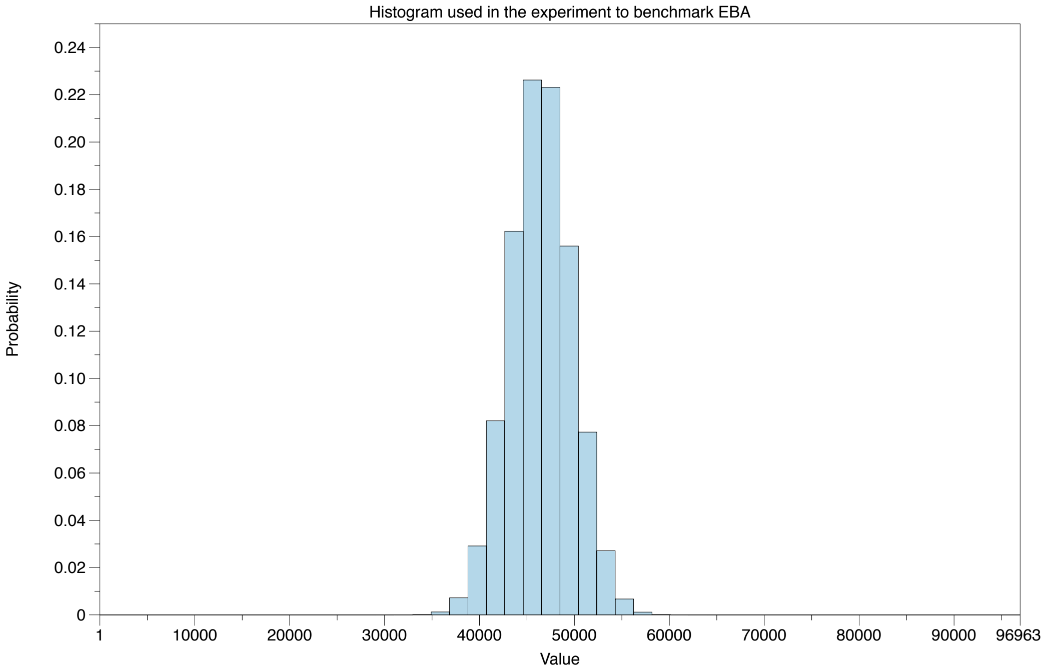


Figure 7.6: Histogram used in the experiment to benchmark the Early Binning Algorithm. The histogram has a non-zero probability in the entire range 1 to 96963. However, the probability in the tails are too low to be observable.

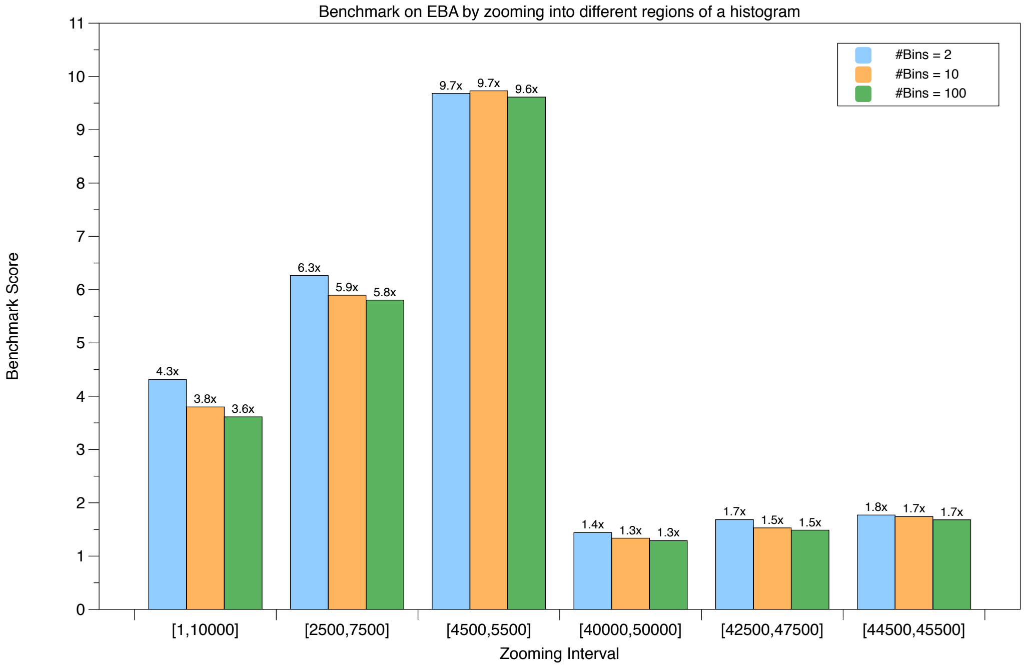


Figure 7.7: Benchmark Score for using the Early Binning Algorithm to zoom into different regions of the histogram in Figure 7.6

The result demonstrates that the Early Binning Algorithm provides better performance when used to zoom into a region where the probability is low. For example, the algorithm provides a speedup of up to 10 tens when used to zoom into the region $[4500, 5500]$, which has a total probability close to 0 as indicated in Figure 7.6. This effect is because a region where the probability is high implies there are many ways for the random variables to sum to a value belonging to that region; there are comparatively few ways to do the same in regions with low probability. By zooming into a region with lower probability, we allow the Early Binning Algorithm to put more intermediate values outside of the region into the correct bin early on, thus saving later computations. Because a user will generally be more interested in zooming into a region with a higher probability, however, the result demonstrates that the performance of the algorithm is limited to a speedup of around 2 times in most common scenarios. This makes the performance of the Early Binning Algorithm similar to the Deferred Binning Algorithm, which was shown to be suboptimal to the Recursive FFT Algorithm in Section 7.3.1.

7.3.4 Accuracy of Histograms with Approximate Probability

In Section 7.3.1, we demonstrated that the Normal Approximation Algorithm (NA) provides much better performance than competitor algorithms for histogram approximation on COUNT and SUM aggregations. Unlike its competitors, however, the Normal Approximation Algorithm produces histograms with approximate probabilities rather than exact probabilities. Thus, it is essential to ensure that the accuracy of the approximate probabilities is high enough for real-world usage.

For each bin, the Normal Approximation Algorithm computes an approximate probability as well as the lower and upper bounds for the probability. We will therefore introduce two metrics, one for measuring the tightness of the bounds and the other for the accuracy of the approximate probability.

The first metric is referred as the percentage bound Δ_{bound}, defined as follows (where B is the number of bins in the histogram):

$$\begin{aligned}\Delta_{bound} &= \frac{\left(\sum_{i=1}^{B} \frac{1}{2}\left(\text{Upper Bound for } bin_i - \text{Lower Bound for } bin_i\right)\right)/B}{1/B} \times 100\% \\ &= \frac{1}{2} \times \sum_{i=1}^{B}(\text{Upper Bound for } bin_i - \text{Lower Bound for } bin_i) \times 100\% \qquad (7.3)\end{aligned}$$

Intuitively, Δ_{bound} indicates the average percentage error per bin between the approximate probability and the lower/upper probability bound. For example, if $\Delta_{bound} = 10\%$, then for a bin with approximate probability 0.5, the probability bounds are expected to be 0.5 ± 0.05.

The second metric is referred to as the percentage error Δ_{error}, defined as follows (where B still is the number of bins in the histogram):

$$\begin{aligned}\Delta_{error} &= \frac{\left(\sum_{i=1}^{B} |\text{Approximate Probability} - \text{Actual Probability}|\right)/B}{1/B} \times 100\% \\ &= \sum_{i=1}^{B} (|\text{Approximate Probability} - \text{Actual Probability}|) \times 100\% \qquad (7.4)\end{aligned}$$

Intuitively, Δ_{error} indicates the average percentage error per bin between the approximate probability and the actual probability. For example, if $\Delta_{error} = 10\%$, then for a bin with approximate probability 0.5, the actual probability is expected to be 0.5 ± 0.05.

The experimental results measuring Δ_{bound} and Δ_{error} are presented in Figure 7.8. First, both Δ_{bound} and Δ_{error} notably decrease with increasing N, an observation consistent with the Central Limit Theorem, which states that the accuracy of normal approximation increases with the number of random variables being convolved. Additionally, Δ_{error} is one or two orders of magnitude less than Δ_{bound} across the two experiments. This observation indicates the possibility to further tighten the theoretical bounds even though the Normal Approximation Algorithm we propose already leverages the best theoretical bounds that currently exist in the literature.

In particular, the Normal Approximation Algorithm on COUNT aggregations shows very promising results: the accuracy is over 99.9% and the bounds are less than 1% of the bin probability. The accuracy is so high and the probability bounds are so tight that there is almost no difference when compared to histograms with exact probabilities. An example of a histogram with approximate probabilities evaluated using the Normal Approximation Algorithm on COUNT aggregations is presented in Figure 7.9a: the bounds are so tight that a magnifier is needed to show where they were marked by a red indicator.

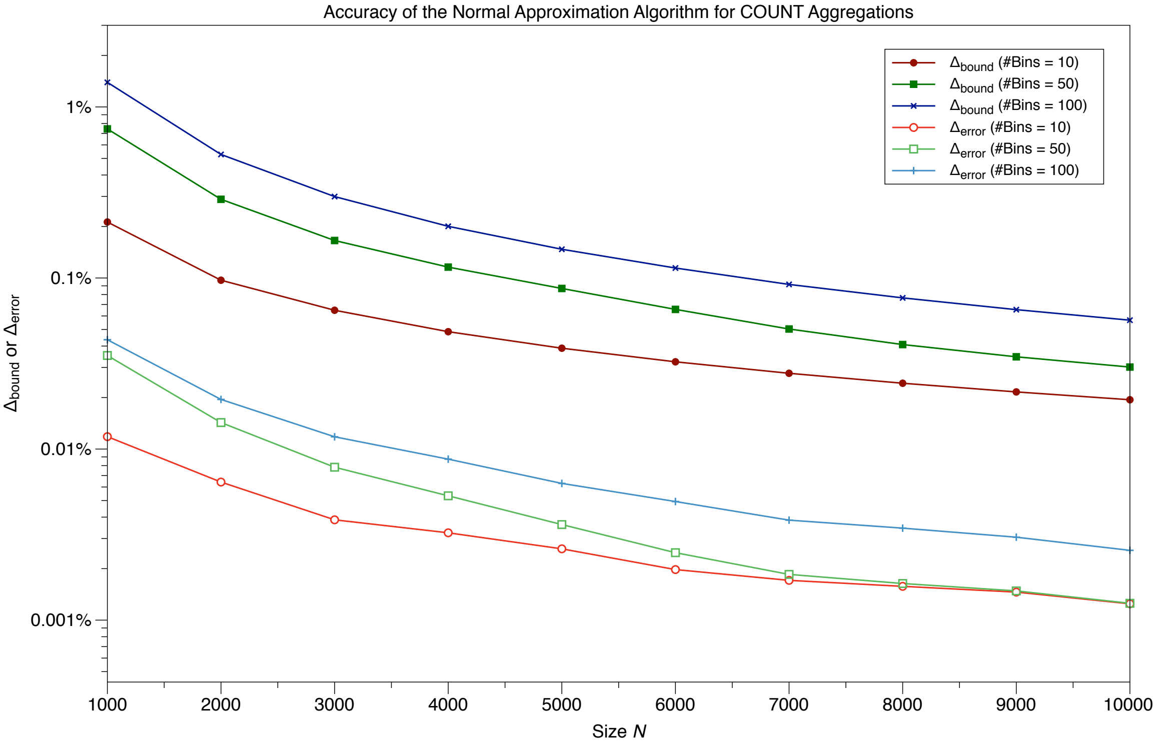
On the other hand, while Δ_{bound} and Δ_{error} are slightly larger for SUM aggregations, the result still suggests that accuracy over 99% can be achieved with sufficiently tight bounds ($< 10\%$ of bin probability for $N > 5000$). The accuracy of the approximate probability and the tightness of the bounds are therefore still sufficient for most real-world applications. An example of a histogram with approximate probabilities evaluated using the Normal Approximation Algorithm on SUM aggregations is presented in Figure 7.9b.

Given the accuracy measured in this section and the high performance recorded in Section 7.3.1, the Normal Approximation Algorithm is the most promising candidate for histogram approximation on COUNT and SUM aggregate queries, especially when the number of tuples being aggregated is large.

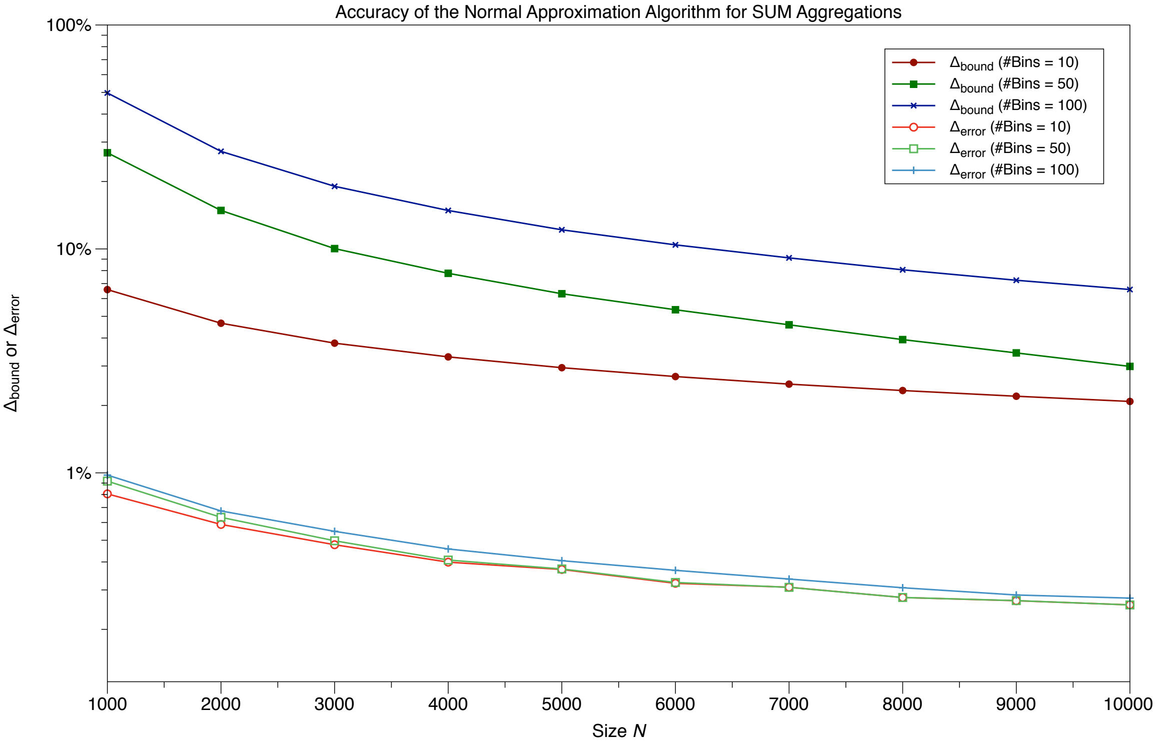
7.4 Top-k Approximation

7.4.1 Scalability

This experiment was based on settings identical to those in the scalability experiment for histogram approximation in Section 7.3.1; the two experiments are therefore directly comparable. More specifically, the experiment uses a Type I semimodule expression (Equation 7.1) with varying N; the experimental result is presented in Figure 7.10.

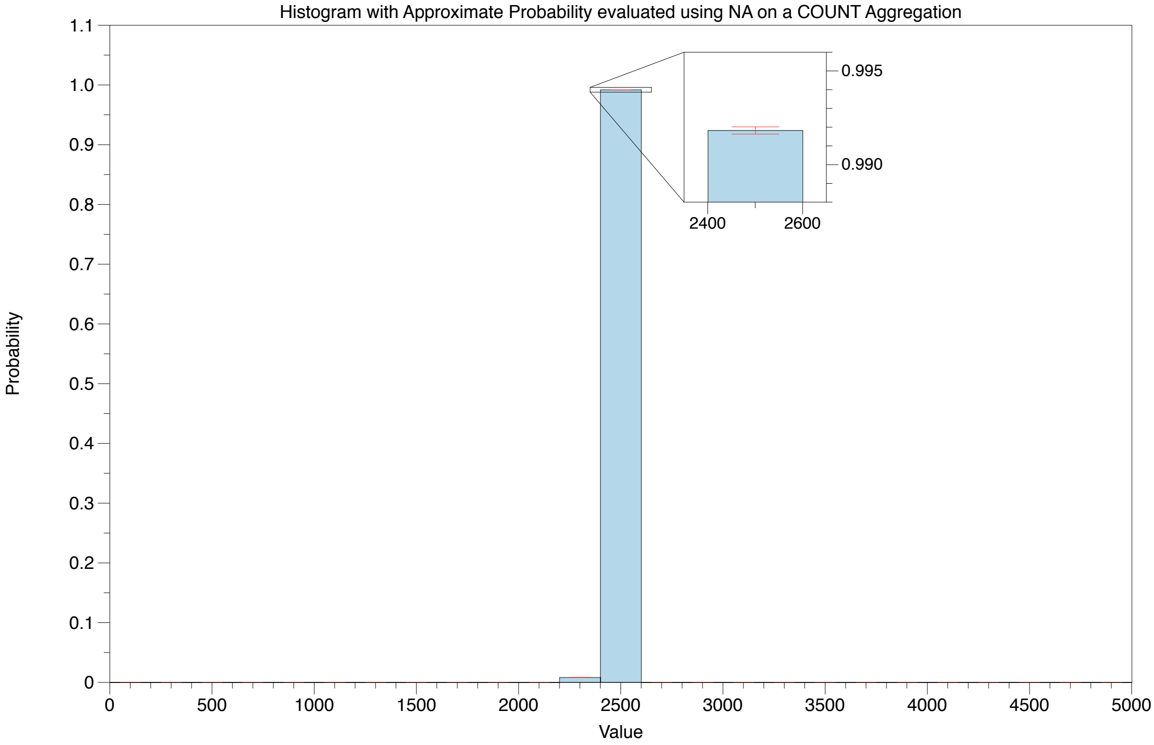


(a) COUNT - Type I Semimodule Expression with $R = 1$

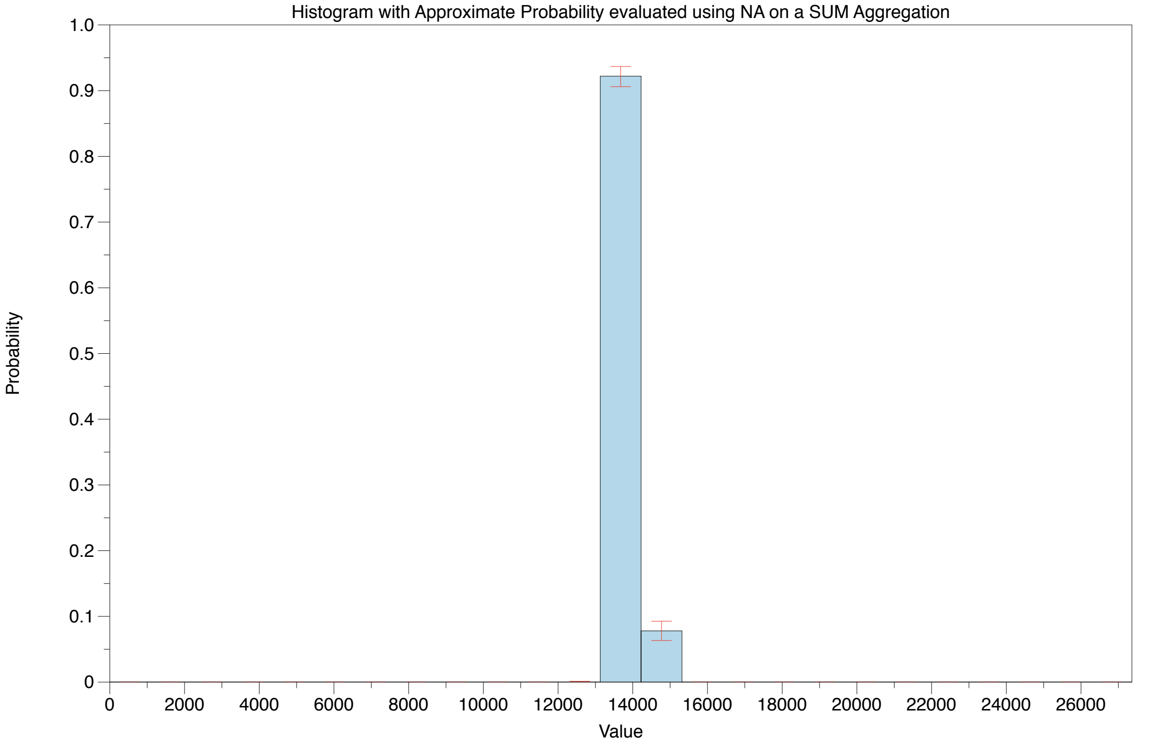


(b) SUM - Type I Semimodule Expression with $R = 10$

Figure 7.8: Accuracy of NA for COUNT and SUM aggregations

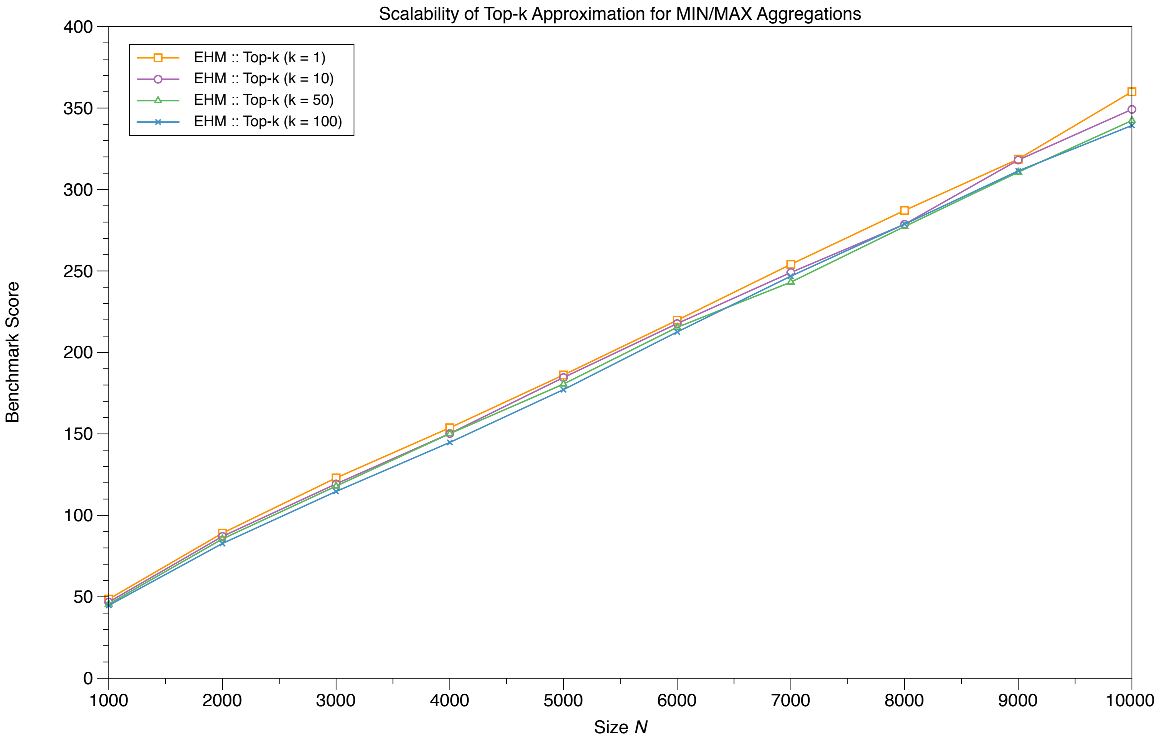


(a) COUNT - Type I Semimodule Expression with $N = 5000$, $R = 1$

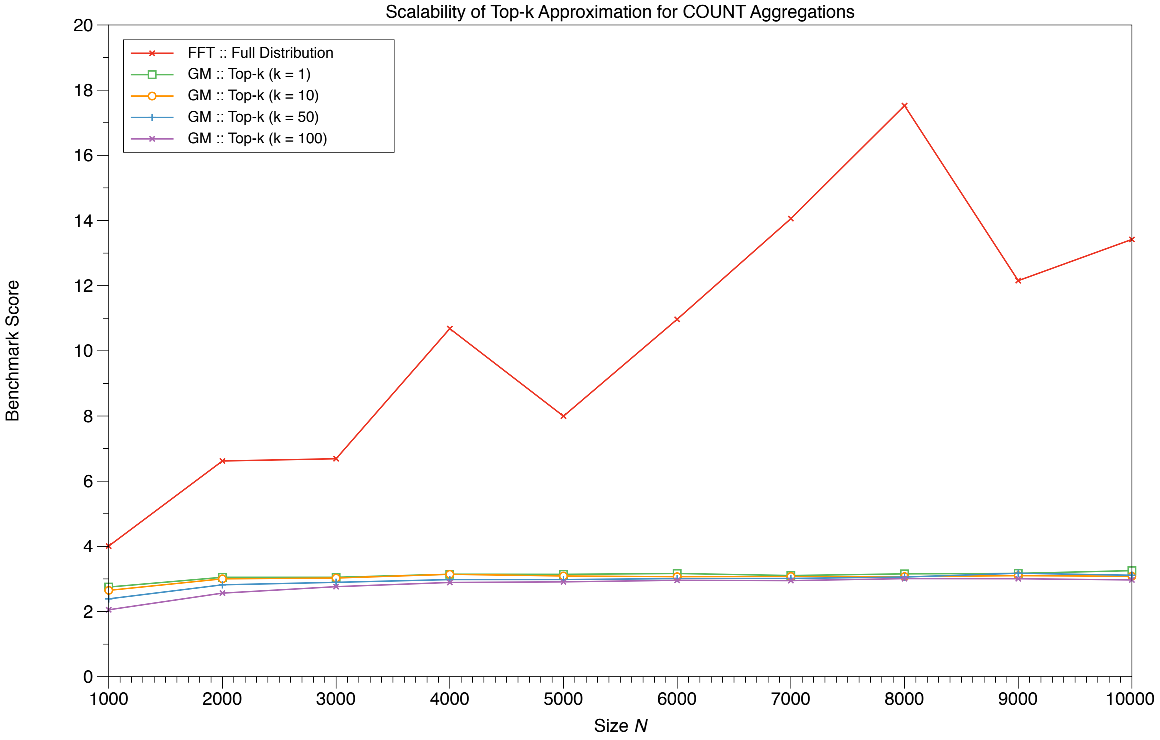


(b) SUM - Type I Semimodule Expression with $N = 5000$, $R = 10$

Figure 7.9: Examples of Histogram with Approximate Probabilities evaluated using NA for COUNT and SUM aggregations (#Bins = 25)



(a) MIN/MAX - Type I Semimodule Expression with $R = 50000$



(b) COUNT - Type I Semimodule Expression with $R = 1$

Figure 7.10: Scalability of Top-k Approximation

For MIN/MAX aggregations, the Extended Heap Method (EHM) demonstrates very promising results, outperforming Standard DP by two orders of magnitude. Most importantly, the benchmark score increases with N, which is an indication of the better scalability of the Extended Heap Method compared to Standard DP. We also note here that the benchmark scores for different values of k are very close to each other, which suggests that including extra tuples in the top-k approximation is inexpensive, conforming to our expectation that the complexity per tuple retrieval for the Extended Heap Method can be as low as $\mathcal{O}(\log N)$.

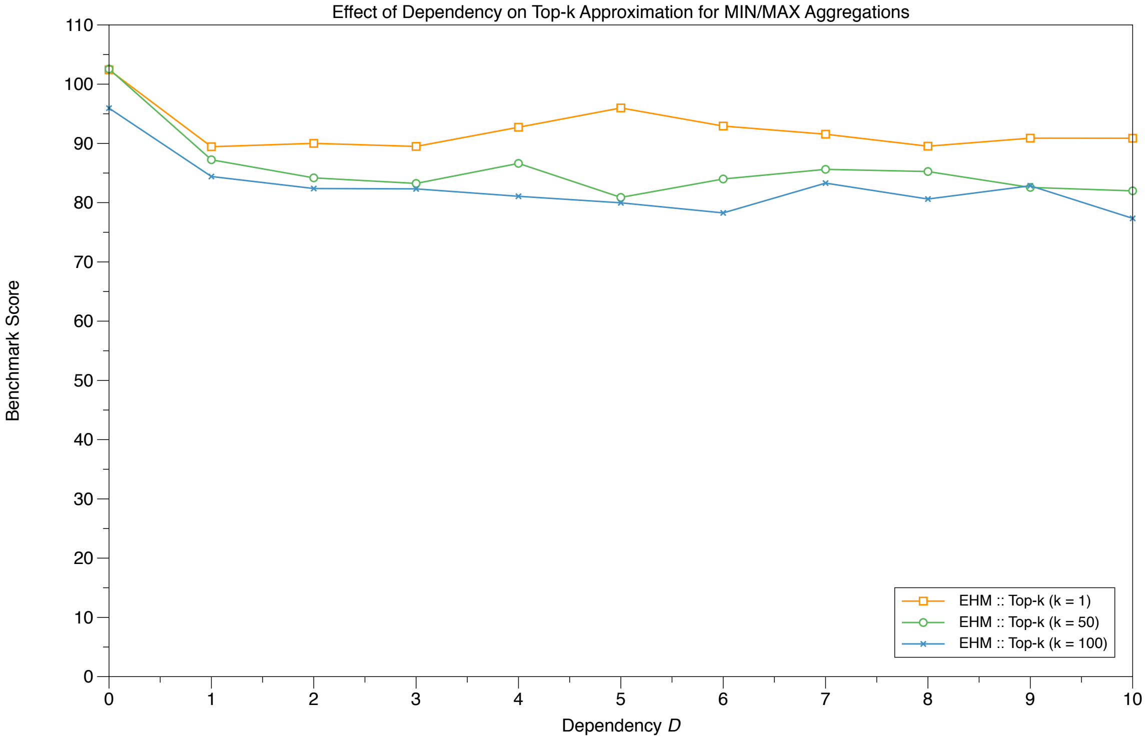
On the other hand, the story is quite different for COUNT aggregations, where the Grid Method (GM) provides a speedup of only roughly 3 times. Additionally, the benchmark is unchanging as N increases, suggesting the Grid Method is not more scalable than Standard DP. We should emphasise, however, that we are observing the worst performance of the Grid Method; this experiment used random variables with a uniform distribution, meaning that the expected value μ of the final distribution is close to $N/2$; that value was demonstrated in Figure 5.1 to be the worst case for the Grid Method. We will investigate the performance of the Grid Method with skewed random variables in Section 7.4.3. Importantly, though, the Recursive FFT Algorithm (FFT) demonstrates better results and better scalability than the Grid Method. Although the Recursive FFT Algorithm does not take advantage of the approximation setting and simply computes the entire distribution (from which top-k tuples can then be extracted), the result confirms that its lower complexity – $\mathcal{O}(N \log N)$ – can beat the Grid Method, which has been optimised for the top-k approximation setting, even for N with moderate size.

Top-k approximation for SUM aggregations is based on the Recursive FFT Algorithm, which was benchmarked in Section 7.3.1. (The Recursive FFT Algorithm computes the full distribution, but the most probable tuples can be extracted from the distribution without significantly affecting the benchmark scores.) The result is similar to applying the Recursive FFT Algorithm on COUNT aggregations, for which a speedup of up to 15 times has been measured.

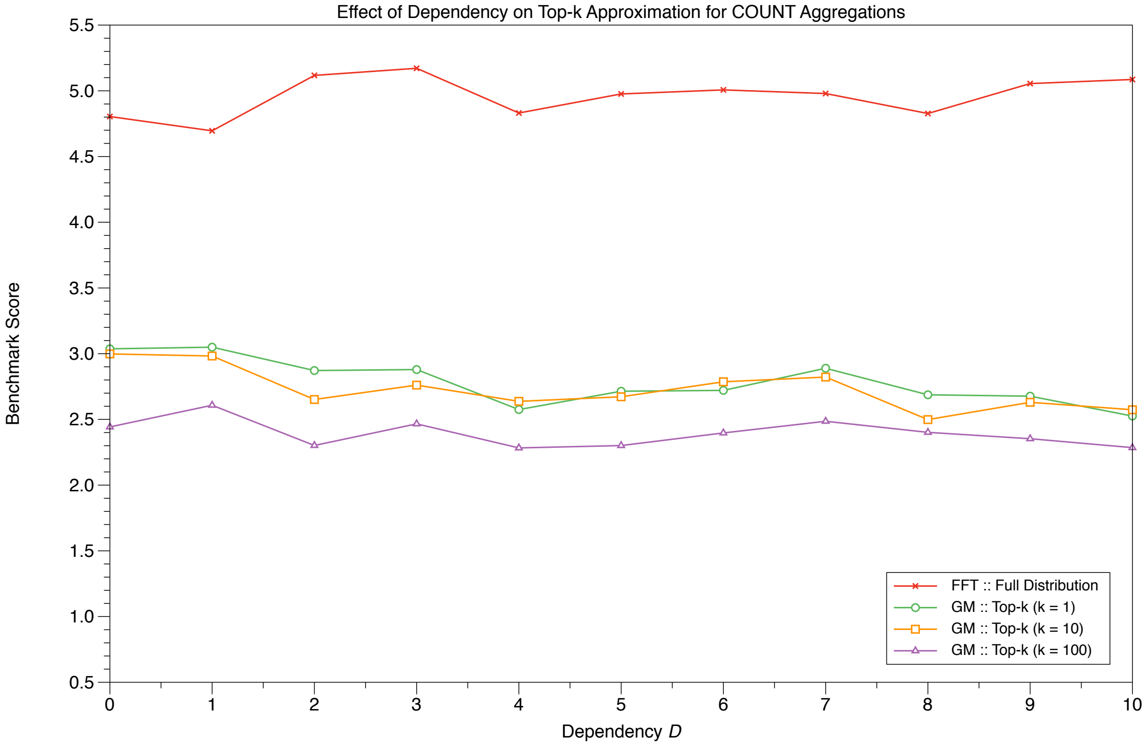
7.4.2 Dependency

Once again, this experiment was based on the same experimental settings as the dependency experiment for histogram approximation in Section 7.3.2 in order to make the two directly comparable. More specifically, the experiment uses a Type II semimodule expression (Equation 7.2) with varying D. The experimental result is presented in Figure 7.11.

Unsurprisingly, the result conveys the same message discovered earlier in the dependency experiment for histogram approximation: the introduction of correlations between the tuples has little effect on the benchmark scores, but does increase the runtime of the algorithms (since more nodes must be included in the d-tree as dependency is introduced). The result provides evidence that the algorithms proposed for top-k approximation can continue to outperform Standard DP even when dependency between tuples is introduced.



(a) MIN/MAX - Type II Semimodule Expression with $N = 2500, R = 50000$



(b) COUNT - Type II Semimodule Expression with $N = 2500, R = 1$

Figure 7.11: Effect of Dependency on Top-k Approximation

7.4.3 Skewness

Unlike histogram approximation or exact evaluation, top-k approximation involves the computation of only the most probable tuples in the probability distribution, therefore the performance of the algorithms could depend on where the most probable tuples locate in the distribution, which is in turn determined by the skewness of the input random variables. In this section, we will measure how skewness could affect the performance of the algorithms for top-k approximation.

MIN/MAX Aggregations

For MAX aggregations, the Heap Method (HM) is expected to have better performance when the random variables are negatively skewed (i.e. more likely to have larger values), as this allows the Heap Method to scan fewer tuples before encountering and retrieving the most probable tuple. (Remember that the Heap Method scans the tuples with larger values first; see Algorithm 30 on page 69 for more details.) For MIN aggregations, the exact opposite is true.

In this experiment, we will consider the following MAX convolution:

$$MAX(x_1, x_2, \ldots, x_{250})$$

where x_i is a random variable with $supp(x_i)$ being a set of 25 randomly generated integers in the range $[1, 5000]$. Skewness is introduced into x_i by setting the probability distribution to be exponential. More specifically, given a skewness factor s, and assume $supp(x_i) = \{v_{i,1}, v_{i,2}, \ldots, v_{i,25}\}$, where $v_{i,j} < v_{i,j+1}$, each value $v_{i,j}$ is assigned a weight according to

$$W(v_{i,j}) = rand(0,1) \times \begin{cases} e^{\frac{ln(10^{|s|})(v_{i,j}-v_{i,1})}{v_{i,25}-v_{i,1}}} & \text{if } s < 0 \\ 1 & \text{if } s = 0 \\ e^{-\frac{ln(10^{s})(v_{i,j}-v_{i,1})}{v_{i,25}-v_{i,1}}} & \text{if } s > 0 \end{cases}$$

where $rand(0,1)$ generates a random float in the range $[0,1]$.

The probability distribution for the outcome $v_{i,j}$ is then given by the normalised weight:

$$P(x_i = v_{i,j}) = \frac{W(v_{i,j})}{\sum_{k=1}^{25} W(v_{i,k})}$$

Intuitively, if $s < 0$, then $P(x_i = v_{i,25})$ is on average $10^{|s|}$ times larger than $P(x_i = v_{i,1})$, and the probability for the values in between increases exponentially. If $s = 0$, then the distribution is uniform. If $s > 0$, then $P(x_i = v_{i,25})$ is on average 10^{s} times less than $P(x_i = v_{i,1})$, and the probability for the values in between decreases exponentially. The compiled d-tree used for the experiment is depicted in Figure 7.12.

The experimental result is presented in Figure 7.13. The result confirms that the performance of Standard DP is unaffected by the skewness of the random variables since it

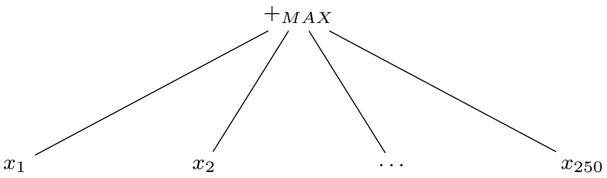


Figure 7.12: D-Tree used for investigating the effect of skewness in top-k approximation for MAX aggregations.

necessarily involves the computation of all of the tuples in the distribution. Additionally, the result conforms to our expectation that the Extended Heap Method (EHM) has better performance for negatively skewed input variables. The impact on the performance of the Extended Heap Method due to skewness, however, is relatively small. For example, introducing a positively skewed distribution in which lower values are 10^6 times more probable slows down the algorithm by only a factor of 10. This confirms the argument we made for Equation 5.5 (page 67) that the most probable tuples are within the largest tuples even when the input variables are heavily skewed. At any rate, the result demonstrates that using the Extended Heap Method for top-k approximation can still outperform Standard DP by a factor of 100 even for heavily skewed random variables.

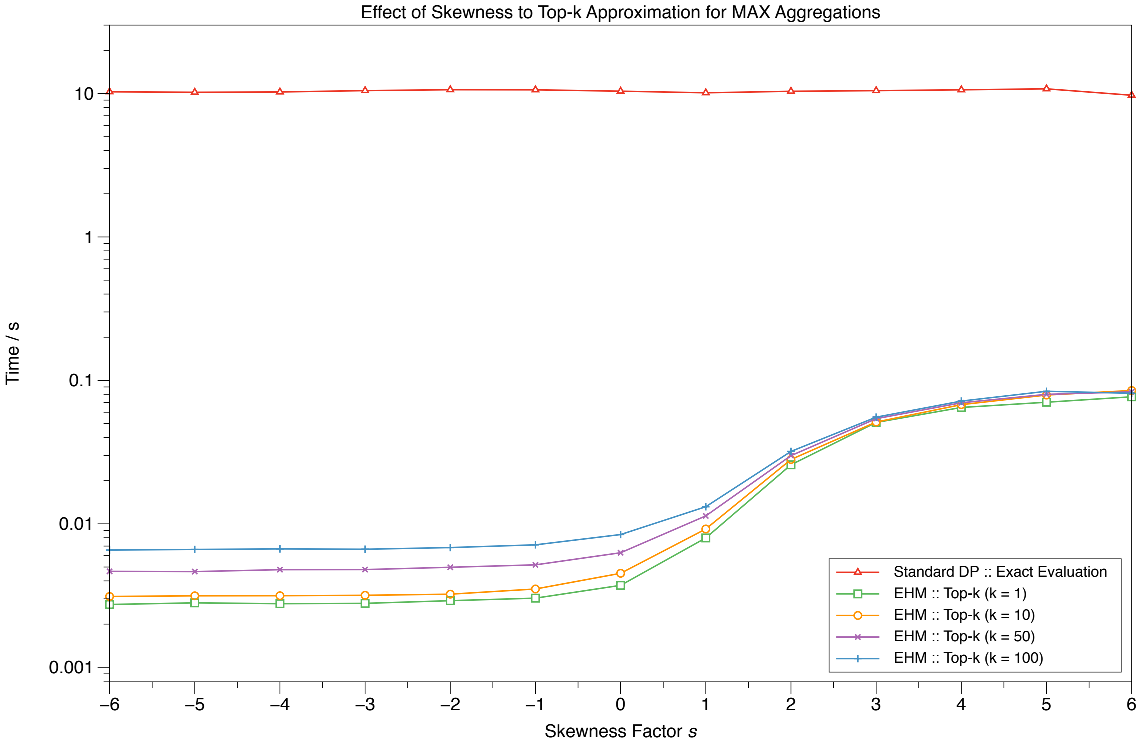


Figure 7.13: Effect of skewed variables on Top-k Approximation for MAX aggregations

COUNT Aggregations

The Grid Method (GM) can be used for the computation of top-k approximations for COUNT aggregations: we have demonstrated in Figure 5.1 (page 77) that the number of cells to be filled to compute the probability of the most probable value is dependent on the sum of the expected value μ for the input variables. Importantly, the Grid Method is expected to perform worst when μ is close to $N/2$, where N is the number of input variables for the COUNT aggregation, and perform best when μ is close to 0 or N.

We conducted this experiment using a Type I semimodule expression (Equation 7.1), with $N = 5000$ and $R = 1$ over the SUM operator. However, we override the default uniform distribution for the random variables (semirings) so that $P(s_i = 0) = 1 - \mu/N$ and $P(s_i = 1) = \mu/N$ for all i, where μ is varied in the experiment. This ensures that the expected value of the sum of the input variables is given by:

$$\sum_{i=1}^{N} E(s_i) = \sum_{i=1}^{N} 0 \times (1 - \mu/N) + 1 \times (\mu/N) = \mu$$

The experimental result is presented Figure 7.14. First, the result confirms that exact evaluation using Standard DP is unaffected by the skewness. Additionally, the runtime for the Grid Method fits the theoretical expectation in Figure 5.1 perfectly: worst performance was observed when $\mu = N/2$, in which case the algorithm outperformed Standard DP by a factor of about 3 times; best performance was observed when $\mu = 0$ or $\mu = N$, in which cases the algorithm outperformed Standard DP by up to 12 times. We must note here, however, that it is not uncommon to find skewness in null probability of the input variables in real-world scenarios. For example, a poorly written web extraction tool might produce many tuples with high null probability since most of the data extracted is uncertain, while a well-written web extraction tool might produce many tuples with lower null probability.

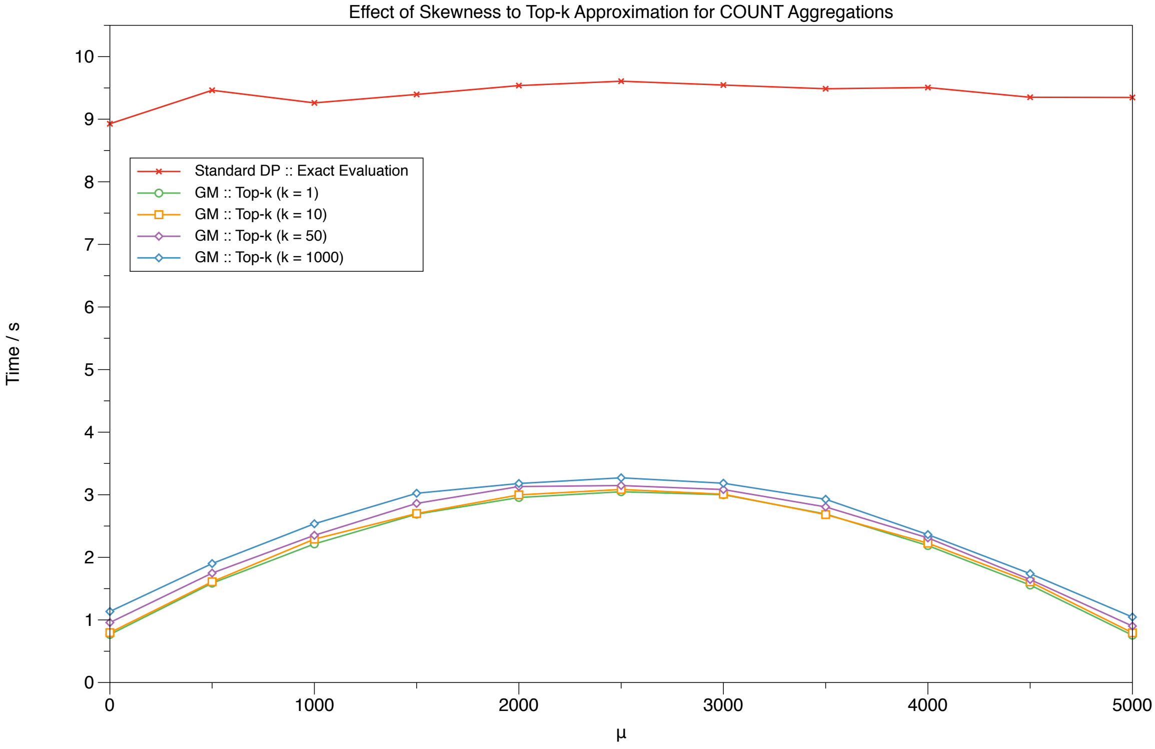


Figure 7.14: Effect of skewed variables on Top-k Approximation for COUNT aggregations

Chapter 8

Conclusion

8.1 Summary

This dissertation proposes a framework for evaluating aggregate queries over probabilistic databases in two completely different approximate settings, *Histogram approximation* and *Top-k Approximation*, by leveraging an existing knowledge compilation technique [19]. The proposed approximation approaches stand in contrast to *all* existing approximation approaches currently in the literature, for which the computation of expected values for query answers is the most common practice. In fact, there is no known work on the use of histogram approximation techniques for even non-aggregate queries in probabilistic databases.

Histogram approximation groups the possible query answers into bins, providing a synopsis of an overall probability distribution. The framework we developed supports the computation of histograms with arbitrary bin intervals, thereby allowing histograms of arbitrary resolution. Manipulation of the bin intervals leads to further exciting applications, such as zooming into part of the histogram for finer resolution and answering range queries efficiently. Top-k approximation, in contrast, is orthogonal to histogram approximation since it retrieves only the most probable tuples in the distribution, the ones that are often the most relevant to the user. We have demonstrated how the two approximation approaches provide a more intuitive meaning to the query results by capturing the essence of the distribution instead of bombarding the user with millions of low-quality possible answers.

We then develop algorithms conforming to the framework for MIN, MAX, COUNT, and SUM aggregations. More specifically, we develop efficient algorithms for computing the histogram representation or the top-k most probable tuples for the distribution result from the convolution of random variables over MIN, MAX, COUNT, and SUM aggregations. Because the convolution of probability distributions is fundamental to probability theory and has been used in a wide range of areas, it is expected that the algorithms can be adapted to situations outside of the field of probabilistic databases. Most of the proposed algorithms have a lower complexity than the state-of-the-art algorithms for exact evaluation, while the others provide a guaranteed performance speedup of at least several times over. To investigate the behaviours of the algorithms under different circumstances, the proposed framework has been implemented and used to benchmark the algorithms thoroughly.

An overview of the performance of the proposed algorithms is presented in Figure 1.4 (page 6): histogram approximation techniques demonstrate very promising results, with a speedup of over two orders of magnitude measured across all aggregations. While the histogram approximation algorithm for COUNT and SUM aggregations (the Normal Approximation Algorithm) produces histograms with approximate probabilities, we note that these approximate probabilities are highly accurate (over 99%) with tight lower and upper bounds (less than a few percent of the respective bin probabilities).

Top-k approximation, on the other hand, demonstrates a performance speedup of over 350× for MIN/MAX aggregations, which is equally promising. While the performance for top-k approximations on COUNT and SUM aggregations is limited by their NP-hard complexity, it remains possible to achieve performance improvements of over 15×. Most importantly, all the algorithms in Figure 1.4 provide better scalability than the state-of-the-art algorithms for exact evaluation.

Finally, we have demonstrated how expected values and other statistical measures of the query answers can also be computed efficiently under the proposed framework, making the framework capable of performing exact evaluation, histogram approximation, top-k approximation, and expected value computation for aggregate queries over probabilistic data. Taken together with the promising performance of the proposed algorithms, now confirmed by experimentation, we believe that the framework lays down a strong foundation for efficient aggregate query evaluation applicable in a wide range of real-world situations.

8.2 Future Work

In addition to the aggregation operators that have been incorporated into the framework (MIN, MAX, COUNT, SUM), AVERAGE is also a popular aggregation operation; support for AVERAGE aggregation could therefore further bolster the utility of the framework. While AVERAGE is conceptually a combination of SUM and COUNT aggregations, the fact that both SUM and COUNT aggregations return a probability distribution makes the problem more complicated than simple divisions.

Additionally, while the accuracy of the Normal Approximation Algorithm for COUNT and SUM aggregations is generally high with tight lower and upper bounds, there are situations in which the user might request even tighter bounds. In the current framework, this can be only achieved by resorting to the evaluation of histograms with exact probabilities, which is substantially slower. An algorithm in which the bounds can be incrementally tightened would bridge the gap between the two methods. For example, recent work by Li and Shi [35] tackles this problem, but their work is mainly for theoretical purposes, and experimental results show a performance slower than exact evaluation using the state-of-the-art algorithms.

Lastly, while the algorithms proposed in this dissertation are based on computation over complete decomposition trees, Fink et al. [20] proposed a technique to compute the lower and upper probability bounds of a Boolean propositional formula via a partially compiled decomposition tree. Extending their work to semimodule expressions would open up a new dimension of approximation via partially compiled decomposition trees, an exciting direction for future work indeed.

Bibliography

[1] Lyublena Antova, Christoph Koch, and Dan Olteanu. 10^{10^6} worlds and beyond: efficient representation and processing of incomplete information. *The VLDB Journal*, 18 (5):1021–1040, October 2009. ISSN 1066-8888. doi: 10.1007/s00778-009-0149-y. URL `http://dx.doi.org/10.1007/s00778-009-0149-y`.

[2] Subi Arumugam, Ravi Jampani, Luis Leopoldo Perez, Fei Xu, Christopher M. Jermaine, and Peter J. Haas. Mcdb-r: Risk analysis in the database. *PVLDB*, 3 (1):782–793, 2010. URL `http://dblp.uni-trier.de/db/journals/pvldb/pvldb3.html#HaasJAXPJ10`.

[3] Omar Benjelloun, Anish Das Sarma, Alon Halevy, and Jennifer Widom. Uldbs: Databases with uncertainty and lineage. In *IN VLDB*, pages 953–964, 2006.

[4] Andrew C. Berry. The accuracy of the gaussian approximation to the sum of independent variates. *Transactions of the American Mathematical Society*, 49(1):pp. 122–136, 1941. ISSN 00029947. URL `http://www.jstor.org/stable/1990053`.

[5] Roger Cavallo and Michael Pittarelli. The theory of probabilistic databases. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 71–81, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc. ISBN 0-934613-46-X. URL `http://dl.acm.org/citation.cfm?id=645914.671645`.

[6] Herman Chernoff. A Note on an Inequality Involving the Normal Distribution. *The Annals of Probability*, 9(3):533–535, June 1981. ISSN 0091-1798. doi: 10.1214/aop/1176994428. URL `http://dx.doi.org/10.1214/aop/1176994428`.

[7] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):pp. 297–301, 1965. ISSN 00255718. URL `http://www.jstor.org/stable/2003354`.

[8] P.H. Cootner. *The random character of stock market prices*. M.I.T. Press, 1964. URL `http://books.google.co.uk/books?id=jW9gT8U6dqQC`.

[9] Graham Cormode and Minos Garofalakis. Histograms and wavelets on probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 22(8):1142–1157, 2010. ISSN 1041-4347. doi: http://doi.ieeecomputersociety.org/10.1109/TKDE.2010.66.

[10] Nilesh Dalvi, Christopher Ré, and Dan Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, July 2009. ISSN 0001-0782. doi: 10.1145/1538788.1538810. URL `http://doi.acm.org/10.1145/1538788.1538810`.

[11] Nilesh Dalvi, Karl Schnaitter, and Dan Suciu. Computing query probability with incidence algebras. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '10, pages 203–214, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0033-9. doi: 10.1145/1807085.1807113. URL `http://doi.acm.org/10.1145/1807085.1807113`.

[12] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 864–875. Morgan Kaufmann, 2004. ISBN 0-12-088469-0.

[13] Landon Detwiler, Wolfgang Gatterbauer, Brent Louie, Dan Suciu, and Peter Tarczy-Hornoch. Integrating and ranking uncertain scientific data. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 1235–1238, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3545-6. doi: 10.1109/ICDE.2009.209. URL `http://dx.doi.org/10.1109/ICDE.2009.209`.

[14] C. G. Esseen. On the liapunoff limit of error in the theory of probability. *Arkiv fr matematik, astronomi och fysik*, A28(2):1–19, 1942. ISSN 0365-4133.

[15] C. G. Esseen. A moment inequality with an application to the central limit theorem. *Scandinavian Actuarial Journal*, 1956(2):160–170, 1956. doi: 10.1080/03461238.1956.10414946. URL `http://www.tandfonline.com/doi/abs/10.1080/03461238.1956.10414946`.

[16] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 102–113, New York, NY, USA, 2001. ACM. ISBN 1-58113-361-8. doi: 10.1145/375551.375567. URL `http://doi.acm.org/10.1145/375551.375567`.

[17] R. Fink, D. Olteanu, and S. Rath. Providing support for full relational algebra in probabilistic databases, 2011. ISSN 1063-6382.

[18] Robert Fink, Andrew Hogue, Dan Olteanu, and Swaroop Rath. Sprout^2: a squared query engine for uncertain web data. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *SIGMOD Conference*, pages 1299–1302. ACM, 2011. ISBN 978-1-4503-0661-4. URL `http://dblp.uni-trier.de/db/conf/sigmod/sigmod2011.html#FinkHOR11`.

[19] Robert Fink, Larisa Han, and Dan Olteanu. Aggregation in probabilistic databases via knowledge compilation. *CoRR*, abs/1201.6569, 2012.

[20] Robert Fink, Jiewen Huang, and Dan Olteanu. Anytime approximation in probabilistic databases. *The VLDB Journal*, pages 1–26, 2013. ISSN 1066-8888. doi: 10.1007/s00778-013-0310-5. URL `http://dx.doi.org/10.1007/s00778-013-0310-5`.

[21] Erol Gelenbe and Georges Hébrail. A probability model of uncertainty in data bases. In *ICDE*, pages 328–333. IEEE Computer Society, 1986. ISBN 0-8186-0655-X. URL `http://dblp.uni-trier.de/db/conf/icde/icde86.html#GelenbeH86`.

[22] N.S. Goel and N. Dyn. *Stochastic Models in Biology*. Academic Press, 1974. ISBN 9780122874604. URL `http://books.google.co.uk/books?id=xK4fC6UYQ68C`.

[23] Todd J. Green and Val Tannen. Models for incomplete and probabilistic information. In Torsten Grust, Hagen Hpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Mller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou, and Jef Wijsen, editors, *EDBT Workshops*, volume 4254 of *Lecture Notes in Computer Science*, pages 278–296. Springer, 2006. ISBN 3-540-46788-2. URL `http://dblp.uni-trier.de/db/conf/edbtw/edbtw2006.html#GreenT06`.

[24] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '07, pages 31–40, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-685-1. doi: 10.1145/1265530.1265535. URL `http://doi.acm.org/10.1145/1265530.1265535`.

[25] Rahul Gupta and Sunita Sarawagi. Creating probabilistic databases from information extraction models. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 965–976. VLDB Endowment, 2006. URL `http://dl.acm.org/citation.cfm?id=1182635.1164210`.

[26] Oktie Hassanzadeh and Renée J. Miller. Creating probabilistic databases from duplicated data. *The VLDB Journal*, 18(5):1141–1166, October 2009. ISSN 1066-8888. doi: 10.1007/s00778-009-0161-2. URL `http://dx.doi.org/10.1007/s00778-009-0161-2`.

[27] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):pp. 13–30, 1963. ISSN 01621459. URL `http://www.jstor.org/stable/2282952`.

[28] Jiewen Huang, Lyublena Antova, Christoph Koch, and Dan Olteanu. Maybms: a probabilistic database management system. In Ugur etintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 1071–1074. ACM, 2009. ISBN 978-1-60558-551-2. URL `http://dblp.uni-trier.de/db/conf/sigmod/sigmod2009.html#HuangAKO09`.

[29] Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 687–700, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376686. URL `http://doi.acm.org/10.1145/1376616.1376686`.

[30] T. S. Jayram, Satyen Kale, and Erik Vee. Efficient aggregation algorithms for probabilistic data. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '07, pages 346–355, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics. ISBN 978-0-898716-24-5. URL `http://dl.acm.org/citation.cfm?id=1283383.1283420`.

[31] O. Kennedy and C. Koch. Pip: A database system for great and small expectations. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 157–168, 2010. doi: 10.1109/ICDE.2010.5447879.

[32] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. Probabilistic event extraction from rfid data. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen,

editors, *ICDE*, pages 1480–1482. IEEE, 2008. URL http://dblp.uni-trier.de/db/conf/icde/icde2008.html#KhoussainovaBS08.

[33] Jens Lechtenbörger, Hua Shu, and Gottfried Vossen. Aggregate queries over conditional tables. *J. Intell. Inf. Syst.*, 19(3):343–362, November 2002. ISSN 0925-9902. doi: 10.1023/A:1020197923385. URL http://dx.doi.org/10.1023/A:1020197923385.

[34] Ezio Lefons and Albert Silvestri. An analytic approach to statistical databases. In *In 9th Int. Conf. Very Large Data Bases*, pages 260–274. Morgan Kaufmann, Oct-Nov, 1983.

[35] Jian Li and Tianlin Shi. An fptas for approximating a sum of random variables. *CoRR*, abs/1303.6071, 2013.

[36] R. Murthy, R. Ikeda, and J. Widom. Making aggregation work in uncertain and probabilistic databases. *Knowledge and Data Engineering, IEEE Transactions on*, 23 (8):1261–1273, 2011. ISSN 1041-4347. doi: 10.1109/TKDE.2010.166.

[37] K. Neammanee. A refinement of normal approximation to poisson binomial. *International Journal of Mathematics and Mathematical Sciences*, 2005(5):717–728, 2005. doi: http://dx.doi.org/10.1155/IJMMS.2005.717.

[38] Dan Olteanu, Jiewen Huang, and Christoph Koch. Approximate Confidence Computation in Probabilistic Databases. In *Proceedings of the 26th International Conference on Data Engineering*, 2010. Long paper.

[39] Christopher Ré, Nilesh Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In *in ICDE*, pages 886–895, 2007.

[40] S. Redner. *A Guide to First-Passage Processes*. A Guide to First-passage Processes. Cambridge University Press, 2001. ISBN 9780521652483. URL http://books.google.co.uk/books?id=xtsqMh3VC98C.

[41] S. M. Samuels. On the Number of Successes in Independent Trials. *The Annals of Mathematical Statistics*, 36(4), 1965. ISSN 00034851. doi: 10.2307/2238127. URL http://dx.doi.org/10.2307/2238127.

[42] I.G. Shevtsova. An improvement of convergence rate estimates in the lyapunov theorem. *Doklady Mathematics*, 82(3):862–864, 2010. ISSN 1064-5624. doi: 10.1134/S1064562410060062. URL http://dx.doi.org/10.1134/S1064562410060062.

[43] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. Probabilistic top-k and ranking-aggregate queries. *ACM Trans. Database Syst.*, 33(3):13:1–13:54, September 2008. ISSN 0362-5915. doi: 10.1145/1386118.1386119. URL http://doi.acm.org/10.1145/1386118.1386119.

[44] Dan Suciu, Dan Olteanu, Chris Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[45] N.G. Van Kampen. *Stochastic Processes in Physics and Chemistry*. North-Holland Personal Library. Elsevier Science, 2011. ISBN 9780080475363. URL http://books.google.co.uk/books?id=N6II-6HlPxEC.

[46] Bengt von Bahr and Carl-Gustav Esseen. Inequalities for the rth absolute moment of

a sum of random variables, $1 \leq \mathrm{r} \leq 2$. *The Annals of Mathematical Statistics*, 36(1): pp. 299–303, 1965. ISSN 00034851. URL `http://www.jstor.org/stable/2238095`.

[47] Y. H. Wang. On the number of successes in independent trials. *Statistica Sinica*, 3: 295–312, 1993.

[48] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical Report 2004-40, Stanford InfoLab, August 2004. URL `http://ilpubs.stanford.edu:8090/658/`.

[49] Mohan Yang, Haixun Wang, Haiquan Chen, and Wei-Shinn Ku. Querying uncertain data with aggregate constraints. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 817–828, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0661-4. doi: 10.1145/1989323.1989409. URL `http://doi.acm.org/10.1145/1989323.1989409`.

Appendix A

Computation of Mean, Variance, and Third Moment For D-Trees

The computation of expected value (mean) for the answers of aggregate queries over probabilistic databases has been well studied in the literature [30, 31, 36]. While the focus of the dissertation is on the computation of histogram approximation and top-k approximation, the Normal Approximation Algorithm (NA; Algorithm 21 on page 52) for computing histogram approximation for SUM aggregations calls for the need to compute mean(μ), variance(σ^2) and absolute third moment(ρ) efficiently for a given decomposition tree (d-tree). Additionally, by bringing the computation of expected values (and other statistical measures) to the proposed framework, we now have a consistent framework that supports exact computation, histogram approximation, top-k approximation and expected value computation for aggregate queries.

A.1 Overview

The definition of mean, variance and absolute third moment is presented in Definition 5.

Definition 5. *Given a probability distribution*

$$\{(v_1, p_1), (v_2, p_2) \ldots, (v_M, p_M)\}$$

The mean, variance and absolute third moment of the distribution is defined by

$$\text{Mean} = \mu = \sum_{i=1}^{M} v_i \times p_i \tag{A.1}$$

$$\text{Variance} = \sigma^2 = \sum_{i=1}^{M} (v_i - \mu)^2 \times p_i \tag{A.2}$$

$$\text{Absolute Third moment} = \rho = \sum_{i=1}^{M} |v_i - \mu|^3 \times p_i \tag{A.3}$$

It is clear that the computation of mean, variance and absolute third moment for a d-tree (or part of a d-tree) can be done by first obtaining the full probability distribution via exact computation, and the definitions of the statistical measures can then be applied over the distribution. However, similar to the strategy we used for the computation of histogram approximation and top-k approximation, our ultimate goal is to devise recursive algorithms that can skip the exact evaluation phrase in all levels of the tree, which requires algorithms that can compute mean, variance and third-moment for a node when given the mean, variance and third-moment of its children.

A.2 VARIABLE

Because a VARIABLE node carries the complete probability distribution, the computation of μ, σ^2 and ρ can be done by simply applying Equations A.1, A.2 and A.3 to the distribution.

A.3 UNION

For a UNION node Y with N children, represented by the random variables $X_1, X_2, \ldots, X_N$, and the probability of the assignment(s) on the i^{th} branch is w_i, the probability distribution Y is given by

$$P(Y = v) = \sum_{i=1}^{N} w_i \times P(X_i = v) \tag{A.4}$$

Now consider $S = \bigcup_i supp(X_i)$ and a function $f(v)$,

$$\begin{aligned}
\sum_{v \in S} f(v) \times P(Y = v) &= \sum_{v \in S} f(v) \sum_{i=1}^{N} w_i \times P(X_i = v) && \text{(Substitution of Equation A.4)} \\
&= \sum_{v \in S} \sum_{i=1}^{N} f(v) \times w_i \times P(X_i = v) \\
&= \sum_{i=1}^{N} \sum_{v \in S} f(v) \times w_i \times P(X_i = v) \\
&= \sum_{i=1}^{N} w_i \sum_{v \in S} f(v) \times P(X_i = v) && \text{(A.5)}
\end{aligned}$$

Now, if $f(v) = v$, and the mean of $X_i = \mu_i$,

$$\begin{aligned}
\mu &= \sum_{v \in S} v \times P(Y = v) \\
&= \sum_{i=1}^{N} w_i \sum_{v \in S} v \times P(X_i = v) && \text{(Substitution of Equation A.5)} \\
&= \sum_{i=1}^{N} w_i \times \mu_i && \text{(A.6)}
\end{aligned}$$

Similarly, if $f(v) = (v - \mu)^2$, and the variance of $X_i = \sigma_i^2$,

$$\begin{aligned}
\sigma^2 &= \sum_{v \in S} (v - \mu)^2 \times P(Y = v) \\
&= \sum_{i=1}^{N} w_i \sum_{v \in S} (v - \mu)^2 \times P(X_i = v) && \text{(Substitution of Equation A.5)} \\
&= \sum_{i=1}^{N} w_i \times \sigma_i^2 && \text{(A.7)}
\end{aligned}$$

And if $f(v) = |v - \mu|^3$, and the third-moment of $X_i = \rho$,

$$\begin{aligned}
\rho &= \sum_{v \in S} |v - \mu|^3 \times P(Y = v) \\
&= \sum_{i=1}^{N} w_i \sum_{v \in S} |v - \mu|^3 \times P(X_i = v) && \text{(Substitution of Equation A.5)} \\
&= \sum_{i=1}^{N} w_i \times \rho_i && \text{(A.8)}
\end{aligned}$$

The result demonstrates that the mean, variance and third moment of a UNION node is the weighted sum of the mean, variance and third moment of its children respectively. Therefore the computation will simply involve getting the mean, variance and third moment of the node's children using the algorithms presented in this chapter, and combine them according to Equations A.6, A.7 and A.8.

A.4 MIN/MAX

We start by making the observation that there is not enough information for the computation of the mean of a MIN/MAX node when only the means of its children is given. For example, consider the following two random variables X_1 and X_2 with probability distributions:

X_1

Value	Probability
20	0.2
60	0.8

$\mu = 52$

X_2

Value	Probability
10	0.4
100	0.6

$\mu = 64$

The convolution result of MAX(X_1, X_2) has the probability distribution

MAX(X_1, X_2)

Value	Probability
100	0.6
60	0.32
20	0.08
10	0

$\mu = 80.8$

However, consider the random variables X_3 and X_4 with probability distributions:

X_3

Value	Probability
52	1

$\mu = 52$

X_4

Value	Probability
64	1

$\mu = 64$

The convolution result of MAX(X_3, X_4) has the probability distribution

MAX(X_3, X_4)

Value	Probability
64	1
52	0

$\mu = 64$

The result demonstrates while X_1 and X_2 have the same means as X_3 and X_4, the mean of the convolution result can be different, suggesting having the means of a MIN/MAX node's children are insufficient for the computation of the mean for the node. For this reason, the computation of mean, variance and third moment for MIN/MAX node will simply involve an exact evaluation to obtain the complete distribution, Equations A.1, A.2 and A.3 can then be applied to turn the distribution into mean, variance and third moment respectively. This suggests that MIN and MAX are blocking operators under the computation of statistical measures. Fortunately, the Heap Method (Algorithm 29 on page 65) supports efficient exact evaluation for MIN/MAX node.

A.5 COUNT/SUM

For a COUNT/SUM node Y, with N children by the random variables $X_1, X_2, \ldots, X_N$, the convolution result is given by

$$Y = \sum_{i=1}^{N} X_i \tag{A.9}$$

Because the children of a COUNT/SUM node must be independent, as otherwise Shannon expansions will take place until this is the case, the variables $X_1, X_2, \ldots, X_N$ are independent.

It is well-known that if the random variables X_i are independent, and have mean and variance μ_i and σ_i^2 respectively, then

$$\mu = \sum_{i=1}^{N} \mu_i \tag{A.10}$$

$$\sigma^2 = \sum_{i=1}^{N} \sigma_i^2 \tag{A.11}$$

While there is no such simple equality for absolute third moment, it is possible to provide an upper bound for it [46]:

$$\rho \leq 4 \times \sum_{i=1}^{N} \rho_i \tag{A.12}$$

We note that the upper bound for ρ can then be used in Equation 4.13 to provide an overestimated bound for the normal approximation error in Algorithm 21 (page 52).

The result demonstrates that the computation of mean, variance and absolute third moment for a COUNT/SUM node is as simple as getting the mean, variance and absolute third moment of its children using the algorithms presented in this chapter, and combining them according to Equations A.10, A.11 and A.12.

Appendix B

Source Code

With over 8000 lines of code, it would be impractical to include all of them in the dissertation. Thus, the source code will be available to be downloaded at

`https://dl.dropboxusercontent.com/u/88890773/ProBASE.zip`

The source code is protected with the password

`Unc3rta1nty`