UNIVERSITY OF CALIFORNIA,
IRVINE

Supporting Query-Driven Cleaning in Probabilistic Databases

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Abdulrahman Abdulhamid Alsaudi

Dissertation Committee:
Professor Sharad Mehrotra, Chair
Professor Michael J. Carey
Professor Yaming Yu

2021

DEDICATION

To my late father, Abdulhamid, who was an example to follow with his incredible accomplishments. To my beloved mother, Jawza, who filled me with unconditional love from birth. To my loving wife, Nourah, and my beautiful daughter, Hadeel, for their ever-lasting support and love. To my sisters and brother, Noura, Hind and Abdullah.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

First and foremost, I would like to extend my sincerest gratitude towards my advisor Professor Sharad Mehrotra for his continuous guidance, mentorship and support throughout my Ph.D. years. Professor Sharad taught me how to identify exciting and challenging research problems and tackle them with confidence and intelligence. Due to his immense knowledge of the area, I witnessed firsthand the skill of abstracting complex ideas and finding a general solution by viewing the larger picture of the problem. I had the most pleasure working under the supervision of professor Sharad, and for that, I am grateful.

I am thankful to professor Yaming Yu for his invaluable knowledge in the statistical field. His guidance and support were crucial for the works in this thesis to flourish. I am also thankful to him for being part of my dissertation committee and his helpful feedback and comments.

I am also grateful to Professor Michael J. Carey for joining my dissertation committee. His technical and editorial feedback was valuable and appreciated. I am also grateful for the unforgettable time I spent as a teaching assistant for his classes and the exciting conversations regarding the different modern database systems.

I would like to extend my appreciation to Dr. Yasser Altowim for his tremendous help and support, which significantly impacted the work presented in chapter 3. I would like to also thank Dr. Hotham Altwaijry for being a great help at the start of my Ph.D. journey.

I am also grateful to King Saud University in Saudi Arabia for providing me with a generous scholarship to obtain my Master's and Ph.D. degrees. The work in this thesis was also supported in part by NSF Grants 1527536, 1545071, 2032525, 1952247, 1528995, 2008993, 2044107, 2139103 and DARPA under Agreement No. FA8750-16-2- 0021.

I would like to thank my colleagues in the ISG group, Professor Shantanu Sharma, Professor Roberto Yus, Wail Alkowaileet, Vishal Chakraborty, Glenn Galvizo, Sameera Ghayyur, Dr. Dhrubajyoti Ghosh, Peeyush Gupta, Shanshan Han, Shiva Jahangiri, Subhamoy Karmakar, Dr. Taewoo Kim, Nada Lahjouji, Yiming Lin, Dr. Chen Luo, Malik Luti, Dr. Primal Pappachan, Sriram Rao, Dr. Mehdi Sadri, Dr. EunJeong Shin, Guoxi Wang and Guangxue Zhang for the amazing times we spent together and the fruitful discussions we had.

I am also thankful for my incredible friends, Dr. Abdulmajeed Alameer, Dr. Abdulrahman Alamer, Dr. Hashem Alayed, Tawfiq Alhathloul, Maan Alnasser, Bassam Alnemer, Malek Alowain, Nasser Alrayes, Ahmed Alrumaih, Abdulaziz Alshayban for the fun times we had, which helped ease the stress and for the unforgettable memories.

Words cannot begin to explain my absolute gratefulness towards my mother, Jawza, who showed me nothing but support, love and care. I will forever be in debt for her sacrifices, and I am utmost grateful to call her my mother.

I would like to give my thanks to my brother, Abdullah, and my sisters Noura and Hind for

their kind and encouraging messages and for being the best siblings a person can have.

Last but not least, I will always be grateful for my loving wife, Nourah, who lived my Ph.D. journey with me through thick and thin and has always supported and motivated me to where I am today. Without her help and sacrifice, this work would not have been possible. I am also thankful to my in-laws for their continuous support.

And to my beautiful daughter, Hadeel, your smile is what keeps me going forward and seeing your happy face after a hard day at work never fails to fill my heart with joy. Thank you for being the sunshine of my life.

VITA

Abdulrahman Abdulhamid Alsaudi

EDUCATION

Doctor of Philosophy in Computer Science **2021**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2013**
University of Southern California *Los Angeles, California*

Bachelor of Science in Computer Science **2010**
King Saud University *Riyadh, Saudi Arabia*

RESEARCH EXPERIENCE

Graduate Research Assistant **2015–2021**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2016–2021**
University of California, Irvine *Irvine, California*

SELECTED PUBLICATIONS

TQEL: framework for query-driven linking of top-k entities in social media blogs **2021**
International Conference on Very Large Databases (VLDB)

Adaptive topic follow-up on twitter **2017**
International Conference on Data Engineering (ICDE)

ABSTRACT OF THE DISSERTATION

Supporting Query-Driven Cleaning in Probabilistic Databases

By

Abdulrahman Abdulhamid Alsaudi

Doctor of Philosophy in Computer Science

University of California, Irvine, 2021

Professor Sharad Mehrotra, Chair

Organizations collect a substantial amount of user' data from multiple sources to explore such data analytically and derive meaningful insights. One of the obstacles that prevent organizations from reaping the benefits of the analysis task is the low quality of the previously collected data. Hence, most of the data preparation time is dedicated to cleaning the data from fixing type errors to removing the uncertainty or ambiguity of some data using data cleaning techniques. A new paradigm for handling such issues is integrating the cleaning process within the query execution workflow to clean the needed tuples rather than performing the cleaning step prior to query execution on the entire dataset. In this thesis, we tackle the challenge of applying the query-driven cleaning approach in the case of probabilistic queries.

First, we present TQEL, a framework that integrates the entity linking task with query processing to answer top-k entities' queries on top of a collection of social media blogs. The entity linking process removes the ambiguity of certain words in any textual snippet by linking such words to real-world entities. The TQEL framework offers two variants: TQEL-exact and TQEL-approximate, that retrieve the exact/approximate top-k results. TQEL-approximate, using a weaker stopping condition, achieves significantly improved performance (with the fraction of the cost of TQEL-exact) while providing strong probabilistic guarantees (over two orders of magnitude lower EL calls with a 95% confidence threshold compared to

TQEL-exact).

Subsequently, we propose TQELX, a framework that generalizes the previous approach to support multiple aggregation functions and other group-based aggregation queries. TQELX is an analysis-aware cleaning for probabilistic queries that use the approximate confidence computation technique. TQELX tightly incorporates the cleaning step in multiple stages of the Monte-Carlo simulation execution to return the results as quickly as possible. We compare our approach against multiple probabilistic query answering baselines and show that TQELX outperformed them in total execution times.

Lastly, we discuss the incremental view maintenance problem in probabilistic databases and provide a solution to speed up the execution process in the case of database updates. Naively, if a tuple is inserted, deleted or updated, the previously computed results become obsolete, requiring the query's re-execution. In cases where the query uses approximate confidence computation techniques, the overhead of such process incurs overheads and unacceptably delays the overall execution experience. We implement PIVM, a solution built on top of PostgreSQL to support delta computation techniques efficiently. Our experiments demonstrate the effectiveness of using such an approach on multiple select-project-join queries and report that PIVM offers a massive execution speed-up in the case of updates.

Chapter 1

Introduction

The last two decades have witnessed a rapid growth in our ability to generate and collect data from diverse sources such as social media, sensors and web data. Organizations collect such data for various purposes ranging from real-time monitoring, supply chain optimizations to short-term and long-term analytic tasks such as forecasting product demand, understanding efficacy of advertisement and determining product demand. As another example, news organizations may collect diverse information for political analysis, and think tanks may collect and analyze data to gain insight on international relations and the impact of policy changes on society. The digitization of our everyday activities coupled with unprecedented growth in technologies to capture our daily life has created an opportunity to acquire, store, analyze and maintain large volumes of data. Today, for example, 1.7 MB of data is generated per user per minute around the world [2] consisting of social media posts, commercial transactions and business interactions.

Such data is often defined as big data that is characterized by four *V*'s that correspond to *volume, velocity, variety* and *veracity* of data. It has become imperative to build efficient data management and analytic tools that address the four V's – that is, they scale to the

massive volumes and velocity of big data, and can handle heterogeneity in the types of data, as well as continue to provide valuable insights when data might contain errors or may be missing. While each of the 4Vs is important and interdependent and need to be addressed by modern information systems, this thesis focuses on challenges related to *veracity* (or quality) of data which plays a major role in the analytical process and may negatively affect the outcome of the data analysis process.

Different factors can affect the quality of the collected data, such as data entry errors (e.g., a receptionist enters the age of a patient as 119 rather than 19). Another factor is manifested by the uncertainty that is present in the collected data. One form of data uncertainty is the ambiguity of words or sequence of words in any textual information (e.g., a tweet that reads "I just arrived in Cambridge" where Cambridge can either be a city in England or a city in the state of Boston). Imprecise machine-generated data such as readings of a sensor are also a different form of uncertainty that affect the data quality by reporting different values for each reading. For example, a speed radar sensor could return two values for a sensor reading at 12:30 pm where the value of the car's speed is 100 mph with a probability of 0.7 or 95 mph with a probability of 0.3.

Data quality problems hinder data scientists' and data analysts' ability to examine and analyze the data thoroughly. Moreover, most of the captured data come from an automatic data generation source that captures data in its raw form, making it harder to derive valuable insights from such data. Hence, removing such difficulties by cleaning/enriching the data helps improve the data quality, which improves the outcome of the analysis. For instance, given a dataset that has the devices' connectivity information for a WIFI router such that the scope of the WIFI router's coverage spans multiple rooms, one might want to find the room with the most connected devices in a given data. The answer would be imprecise if the analysis relied on the raw data alone since it does not provide the values at a fine-grained level. In order to accurately locate each device's connection instance, one might use

a cleaning function such as LOCATER /citelin2020locater which deterministically provides room-level localization using semi-supervised learning techniques. By running the cleaning function on the dataset, the uncertainty of the data would be reduced, which allows for the proper execution of the analysis task.

Probabilistic databases systems that have been introduced in the literature can be used to manage data with uncertainty where the value of an attribute or the values of a tuple is associated with an existence probability in the database. Systems such as TRIO [88], MCDB [52] and MayBMS [50] are examples of probabilistic database management systems that enable storing, maintaining and querying uncertain datasets. The naive method for answering probabilistic queries is by enumerating all the possible worlds (instances) of the database and then summing up the probabilities of the worlds that a tuple t appear in as an answer for that world. For example, table 1.1 is an uncertain table that holds car owners' information along with the details of their owned cars. The different possible worlds, along with their probabilities, are represented in tables 1.2, 1.3, 1.4 and 1.5.

However, this approach is not feasible since the number of possible worlds is exponential. Instead, probabilistic database systems usually follow two different queries answering semantics that return a probabilistic answer with a confidence score with each answer. These query answering semantics are the exact confidence computation semantics and the approximate confidence computation semantics. The exact confidence computation semantics refers to the method of returning the exact confidence score of a tuple or group of tuples satisfying a given query (e.g., by transforming a query to a DNF form statement and calculating the

CarOwners				
Tuple_id	Owner	Price	Year	P
1	John	60,000	2019	0.2
2	Amy	65,000	2020	0.7
3	Emily	50,000	2018	1

Table 1.1: Dataset for Car Owners Information.

CarOwners			
Tuple_id	Owner	Price	Year
1	John	60,000	2019
2	Amy	65,000	2020
3	Emily	50,000	2018

Table 1.2: Instance 1 with P = 0.14

CarOwners			
Tuple_id	Owner	Price	Year
3	Emily	50,000	2018

Table 1.3: Instance 2 with P = 0.24

CarOwners			
Tuple_id	Owner	Price	Year
1	John	60,000	2019
3	Emily	50,000	2018

Table 1.4: Instance 3 with P = 0.06

CarOwners			
Tuple_id	Owner	Price	Year
2	Amy	65,000	2020
3	Emily	50,000	2018

Table 1.5: Instance 4 with P = 0.56

answer's confidence from a tuple's existence probability) [30, 59]. On the other hand, the approximate confidence computation semantics is deployed by generating, randomly, enough samples from the possible world's space and then running the query on each randomly generated world and returning the confidence of the given answer based on the answers of all generated worlds (e.g., by using Monte-Carlo simulation techniques in order to answer DNF styled queries) [56, 57, 58]. The exact confidence computation method can only be used for queries that have safe-query plans [30] (e.g., the confidence score of the answer can be computed exactly in polynomial time without generating the entirety of the possible worlds for the database). In contrast, other queries can only be answered using the approximate confidence computation method in polynomial time.

While probabilistic database systems enable applications to store and query probabilistic data, they by themselves cannot address the restrictions applications face due to uncertainty. As an example, an analyst wants to find the most popular iPhone model on social media. The popularity of a model is measured by the number of times it has been mentioned in each social media post. Given the text "I just bought the new iPhone," the word "iPhone" mentioned in the text could possibly refer to multiple models. If one were to run the query as a probabilistic query, the returned answer would be probabilistic also where each possible answer is associated with a confidence score. Hence, a returned answer set: {"iPhone 13":

0.3, "iPhone 12": 0.2, ...} will not suffice for analysis purposes.

The technique that is mainly followed when addressing data uncertainty in literature is cleaning. Data cleaning is an essential task for deriving meaningful insights and observations about the given data. This process usually takes a considerable portion of the overall analysis time due to multiple complexities. It is reported that data scientists spend 80% of the allocated time for cleaning and preparing the data, whereas the rest 20% is spent for insights derivation [65]. The process of data cleaning can take place before loading the data, where the cleaning process is an offline step and is performed once and is performed on the data as a whole. Another possibility is to perform cleaning periodically where a continuous stream of data would undergo the cleaning process in batches. Alternatively, at query-time, where the cleaning step is integrated within the query processing pipeline. The key concept is to focus on a subset of the dataset rather than cleaning the entire dataset [10, 11, 22, 43, 68].

There are several advantages to adopting a query-driven approach compared to other proposed approaches, especially when dealing with large volumes of data. One of the advantages is avoiding the nuisance of cleaning the entire dataset since it is time-consuming and considered wasteful when the query rate is not high. The query-driven approach can help filter out some of the tuples in the dataset using the query's predicate by exploiting the semantics of the query's operator, such as in top-k queries. Another form of exploitation uses functional dependencies to identify the tuples that need cleaning in a query context. Another advantage is the option of analyzing the data as it arrives and reporting the results for recent data rather than running the cleaning step offline and then reloading a cleaned version of the data or waiting for the periodic cleaning schedule to clean the newly arrived data.

The basic flow of a query-driven approach is depicted in the cleaning cycle in figure 1.1. When a user asks a query on top of a collection of dirty data (i.e., data with uncertainty), the query is sent to the query processing engine for evaluation. After the query result is returned, it is examined to see if it satisfies the query answer's semantics (e.g., the returned

answer is equivalent to the answer after cleaning the entire collection of data) or if the user feels content with the returned answer. However, if the result's quality does not suffice, some tuples are chosen from the database to be cleaned. After the data undergo the cleaning phase, the data is updated accordingly. The flow of the query-driven approach goes iteratively until the results are accepted.

In this thesis, we explore novel ways to support integrating the cleaning step in probabilistic query processing while reducing the overhead associated with it (e.g., reducing the number of tuples to be cleaned). We first focus on solving a more specific query: a top-k query over counts of entities in Twitter. In particular, we present TQEL, a query-driven framework for efficiently finding top-k entities in a collection of social media blogs by calling the entity linking function. Given the noisy nature of the textual information, entity linking refers to the process of linking an entity mentioned in the text of tweets to a real-world entity in a knowledge base such as Wikipedia. In order to find the top entities mentioned in a collection of tweets, we first need to link those mentions and then count the number of occurrences of each real-world entity to return the k entities with the most counts. TQEL framework is capable of handling top-k queries and returning either an exact answer or a probabilistic one with confidence guarantees. Our contributions focus on the probabilistic solution where we apply the approximate confidence computation approach by adopting the Monte-Carlo simulation technique to evaluate the top-k query. In addition, we introduce a

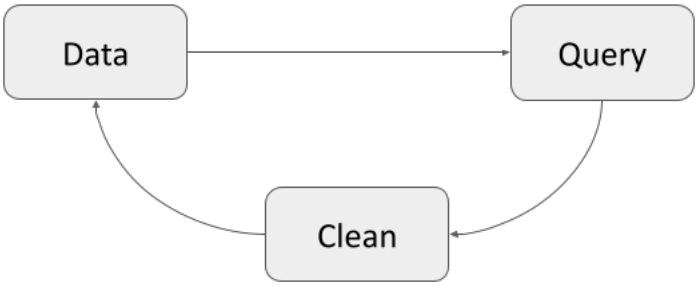

Figure 1.1: Uncertainty Cleaning Cycle

filter based on normal approximation calculations, which acts as a low-cost test to check if an answer for the top-k query is potentially found rather than repeatedly using the more expensive query evaluation based on Monte-Carlo simulation. Our experiments show that the proposed approximate approach is over two orders of magnitude lower calls to the entity linking function with a 95% confidence threshold compared to the exact approach. Moreover, the exact approach is orders of magnitude better than a naive approach that calls the entity linking function on the entire collection of tweets.

We then generalize the TQEL framework in order to serve more group-based aggregation queries defined using SQL. We illustrate the enhancements to the uncertainty cleaning cycle to answer queries with probabilistic guarantees using the Monte-Carlo simulation technique. We expand the types of supported queries to include different top-k aggregation queries such as top-k SUM and top-k AVERAGE queries. We also include group-based aggregation queries with a having clause (e.g., a query that finds entities that appeared more than 50 times in a collection of tweets). We describe how the normal approximation filter calculations are affected by each query and introduce different optimizations to the Monte-Carlo simulation technique in order to evaluate the queries. The experiments show massive savings in the number of cleanings performed when answering the query using the probabilistic guarantees approach compared to the exact and naive approaches.

After that, we focus on tackling the issue of the repeated and expensive execution of the query evaluation using the Monte-Carlo simulation in the context of the uncertainty cleaning cycle. We implement an incremental view maintenance solution tailored for probabilistic queries to expedite the delta computation process. We implement our solution on top of PostgreSQL to support a wide range of select-project-join style queries. Our primary intuition is to leverage the Monte-Carlo samples generated in previous query executions for tuples that have not been updated. Our experiments show that using our proposed implementation of IVM, delta computations techniques are more efficient than rerunning the query entirely

after each update.

The rest of this thesis is organized as follows. Chapter 2 covers the related work. In Chapter 3, we describe the TQEL framework for answering top-k entities on top of social media blogs and then discuss the extension of the TQEL framework to include different aggregation queries in chapter 4. In chapter 5, we present our implementation of IVM for probabilistic query processing. Finally, we conclude this thesis and discuss future work in chapter 6.

Chapter 2

Related Work

In this chapter we discuss the different works introduced in the literature that are related to my thesis in order to set the stage for chapters 3, 4 & 5.

2.1 Probabilistic Databases

The literature has extensively studied the idea of representing uncertainty in data using a probabilistic model in a relational database. It was first presented in works such as [15, 40] to handle NULL values and attribute-level uncertainty, where the entire tuple's values are deterministic except for one uncertain value, by providing data models and simple query evaluation techniques. Other works focused on a different direction and looked at the model of uncertain data in the direction of possible world semantics [38]. ProbView [61] explored the problem of different possible dependencies among records where the previous probabilistic modeling assumed independence between different tuples. ProbView studies the cases where some degree of dependency is present between tuples, especially in image classification. ProbView assumes some input from the user to define the relation between tuples in

terms of dependency and explains if the query evaluation complexity would be the same as the deterministic evaluation of the method that the user chooses, has polynomial time complexity.

Other works have proposed different probabilistic query evaluation approaches for different queries that have polynomial time complexity. For instance, Dalvi et al. [30] categorize query plans into either safe plans or unsafe plans. Safe plans are plans that could be extended by associating the existing probability with each intermediate relation in the traditional relational query plan. Their study concludes that if the query's intermediate relations probabilities can be computed in polynomial time, the query plan is considered safe, and the query execution has polynomial time complexity. In [23], authors have provided a different approach for query processing of tuples with attribute-level uncertainty where an uncertain attribute value is given as a probabilistic range instead of a probabilistic value. Once a query is given, either a value or a set of values are returned with a probability density function corresponding to the confidence that it/they satisfies the query.

Some works are more tailored to probabilistic query evaluation for aggregates, such as [18, 54] which returns, as an answer, the estimate of the aggregate value where other works focus on the expected value for the aggregation queries [12, 50, 52, 76]. In [75], authors present a solution for solving an arbitrary SQL query by returning the top-k answers with the highest probability that satisfy the query's predicate. The authors propose an approximate confidence calculation technique and limit the sample generation to interesting tuples (i.e., tuples with a high chance of being in the top-k answer).

Different proposed probabilistic databases use different probabilistic data models and query semantics. Trio [7, 88] introduced the x-tuples model and a method for expressing lineage. Trio data model considers a database or a relation composed of different probabilistically independent x-tuples. Each x-tuple has multiple sub-tuples associated with it such that $\sum$ of probabilities of all sub-tuples ≤ 1. Trio supports queries on the confidence scores and

lineage from the query's result to the base tables by considering the combinations of all possible answers for the given query. The system supports single-block select-project-join style queries and uses exact confidence computation techniques for confidence computation. Another system is the Monte-Carlo Database system (MCDB) [52] which uses the approximate confidence computation technique for answering all types of probabilistic queries. Probabilities for tuples are presented as either probability values or parameters for a distribution function (e.g., normal distribution function with μ & σ as parameters). MCDB delays the Monte-Carlo simulation for query evaluation until other query operators are applied to filter out tuples that do not satisfy the query's predicate. In the coming chapters, we dive into the details of implementing the Monte-Carlo simulation technique in our query-driven frameworks.

MayBMS [12, 50] is another probabilistic database that is a modified version of PostgreSQL where each tuple has a probability of existence associated with it and independence between tuples is assumed. MayBMS provides two approaches for query processing where it follows the exact confidence if the query plan is safe and uses the approximate confidence computation otherwise. The returned answer is usually a tuple or set of tuples and the confidence is associated with each one of them. Another example is MYSTIQ [17] which uses PostgreSQL in order to define BID (block independent disjoint) relations. Their query processing approach is similar to MayBMS where they utilize both confidence computation semantics.

The quality of the query's answer has also been researched for different queries and domains. In [24, 32, 49], the query's result has to be above a predefined threshold confidence value in order to be considered satisfactory to the query semantics. In [22, 68] the answer quality is considered to be the entropy [79] of the probabilities given by the possible world semantics where if the entropy is zero, that means the answer is a deterministic answer for the query. Other quality metrics have been proposed for specific queries, such as in [23] where the

SpeedReadings				
xid	Reading_id	License_plate	Speed	P
x_1	r_1	ABC	100	0.6
x_1	r_2	XYZ	80	0.3
x_1	r_3	MNO	30	0.1
x_2	r_4	MNO	120	0.3
x_2	r_5	XYZ	70	0.7
x_3	r_6	XYZ	90	1
x_4	r_7	ABC	110	0.4

Table 2.1: Speed readings table

Possible_world	Probability	Top-2
r_1, r_4, r_6, r_7	0.072	r_4, r_7
r_1, r_5, r_6, r_7	0.168	r_7, r_1
r_1, r_4, r_6	0.108	r_4, r_1
r_1, r_5, r_6	0.252	r_1, r_6
r_2, r_4, r_6, r_7	0.036	r_4, r_7
r_2, r_5, r_6, r_7	0.084	r_7, r_6
r_2, r_4, r_6	0.054	r_4, r_6
r_2, r_5, r_6	0.126	r_6, r_2
r_3, r_4, r_6, r_7	0.012	r_4, r_7
r_3, r_5, r_6, r_7	0.028	r_7, r_6
r_3, r_4, r_6	0.018	r_4, r_6
r_3, r_5, r_6	0.042	r_6, r_5

Table 2.2: Possible worlds results.

quality metric is tailored for queries on tuples where uncertainty is captured by probability intervals values rather than exact probabilistic values.

For a deeper look at other probabilistic databases and probabilistic query evaluation techniques, surveys such as [6, 85] summarize most of the related work in that area.

2.1.1 Probabilistic top-k Query

Let us briefly review the concept of top-k queries in probabilistic databases that have been extensively studied in different contexts and variations of the problem [28, 49, 83, 84, 90]. Based on the survey by Ilyas et al. [51], the top-k queries studied in the literature can be characterized into three categories: top-k selection queries, top-k join queries and top-k aggregation queries. Furthermore, top-k queries are also characterized by their semantics that we explain below using a running example.

Consider Table 2.1 that follows the semantics of x-tuple [7, 88] where the relation has 4 mutually exclusive rules x_1, x_2, x_3 and x_4. The rules indicate that at most, one of the

readings associated with such a rule will be chosen in each instance of the relation. For example, an instance of this relation could be: r_1, r_4, r_6. Table 2.2 represents the entire possible worlds for this relation along with the probability and the top-2 answer for that instantiated world. The table records the information of different cars that are passing a sensing radar that captures each car's speed. Due to the fuzzy nature of such sensors, the readings are not certain and hence are reported with a probability of existence. We illustrate different top-k query semantics that has been proposed by using an example of the following top-k selection query to retrieve top-2 readings for the highest speed:

```
SELECT      Reading_id
FROM        SpeedReadings
ORDER BY    Speed DESC
LIMIT       2;
```

U-Top$k$ [83, 84] semantics determines the probability of each possible top-k answer and returns the most probable one. For instance, probability of the answer r_4, r_6 is the sum of probabilities corresponding to the possible worlds r_2, r_4, r_6 which is 0.054 and r_3, r_4, r_6 which is 0.018. Thus, probability of r_4, r_6 to be the top-2 answers is 0.072. In this example, the actual answer would be r_1, r_6 with p = 0.252 since r_1, r_6 since it is the most probable answer amongst all possible top-2 answers. As should be clear, U-Top$k$ can be easily answered if we have the entire set of possible worlds along with each world probability.

U-kRanks [83, 84] semantics is different then the previous one as it finds for each rank 1, 2 ... k, the tuple that has the highest probability of being ranked i where $1 \leq i \leq k$. If every possible world result is reported, one can simply calculate the probability of each tuple r_j being ranked i by the summation of possible worlds probabilities where r_j is ranked i. For example the probability that tuple r_6 being tanked top-1 is : 0.126 + 0.042 = 0.168. U-2Ranks answer is = r_4, r_6. P(r_4 is rank 1) = 0.3 and P(r_6 is rank 2) = 0.436.

Another semantics for top-k is Global-Top$k$ [90] which is based on the top-k probability of

each tuple being in the top-k. Top-k probability for each tuple is calculated, then the first k tuples with the highest top-k probability are reported. In order to calculate the top-k probability for tuple r_j, the probability of all possible worlds w_i where r_j appears in the answer set of the top-k query are summed. For example, P(r_1 in top-k) = 0.108 + 0.168 + 0.252 = 0.528. Global-Top2 answer for the query would be: r_6, r_1. P(r_6 in top-k) = 0.604 and P(r_1 in top-k) = 0.528.

PT-k [49] follows the same semantics as Global-Top$k$ but with two differences. The first difference is that there is a user-specified threshold τ associated with the top-k query such that the query will only return tuples with a top-k probability $> \tau$. The second difference is that unlike Global-Top$k$ where the top-k query will exactly return k tuples, PT-k might return less or more than k tuples depending on the threshold value τ. For example, if the threshold value $\tau = 0.55$, then the answer for PT-$k = r_6$. On the other hand, if $\tau = 0.5$ then the answer for PT-$k = r_6$, r_1.

Another top-k semantics is the expected score [28] where for each tuple, the expected score is calculated by multiplying the score $\times$ existing probability. The tuples are then sorted based on their expected score and then a top-k query is directly applied. In our running example, top-2 would be: r_6, r_1, where the expected score for r_6: 90 and the expected score for r_1: 60. Expected rank [28] is another proposed top-k semantics which captures the expectation of the rank of each tuple by multiplying rank score of each tuple in each possible world w_i $\times$ p(w_i), where the rank score for r_j corresponds to the number of tuples that are ranked higher than r_j in possible world w_i. The expected rank for the top-2 running example is: r_6, r_1 where the expected rank for r_6: 1.3 and the expected rank for r_1: 1.78.

Most top-k algorithms that have explored different top-k semantics, including PT-k semantics, have explored top-k selection queries. As discussed clearly in [84], such approaches to support top-k selection queries cannot be applied to top-k aggregation queries. Top-k aggregation queries have also been studied in [84], where possible worlds are incrementally

materialized for groups with tuples that have the highest function score. Each group G_x is given a range of scoring function [0, UB * $|G_x|$] where UB is the score of the tuple with the highest score in that group. However, their algorithm is only limited to aggregate functions such as AVERAGE and SUM. Although one can treat COUNT as SUM(1), such an adoption would not benefit from their incremental materialization of the possible worlds as it would end up materializing most of the possible worlds in order to get the answer for the top-k query.

2.2 Data Cleaning

Data cleaning is an essential part of the ETL (extract, transform and load) process and has been extensively studied in the context of data warehouses. Given massive volumes of collected data, data warehouses load them for online analytical analysis and business intelligence applications using OLAP queries. However, before the loading happens, data has to be in a state that is ready for processing to provide accurate results. Hence, data cleaning techniques are performed within the transformation stage [73].

A well-known data cleaning problem is the entity resolution (ER) or tuple deduplication problem. The entity resolution problem refers to the process of the identification and merging of duplicate objects or tuples that refer to the same entity [16, 29, 35, 48, 60]. Given a database and records within it, the abstract idea is to run the entity resolution between each pair of records to find the records or tuples that are the same. Different approaches have been proposed to efficiently tackle the issue by utilizing blocking techniques in order to cluster records into groups that have a high chance of pairing and then calling the entity resolution function for pairs within the same cluster or group [9, 48, 53, 66]. Other data cleaning problems study the inconsistency in the data due to integrity constraints (IC). Such data inconsistencies occur for several reasons, including manual typing errors during data

entry or errors introduced when merging data from various sources. Such errors can often be detected and repaired by exploiting integrity constraints over defined over data [25, 26].

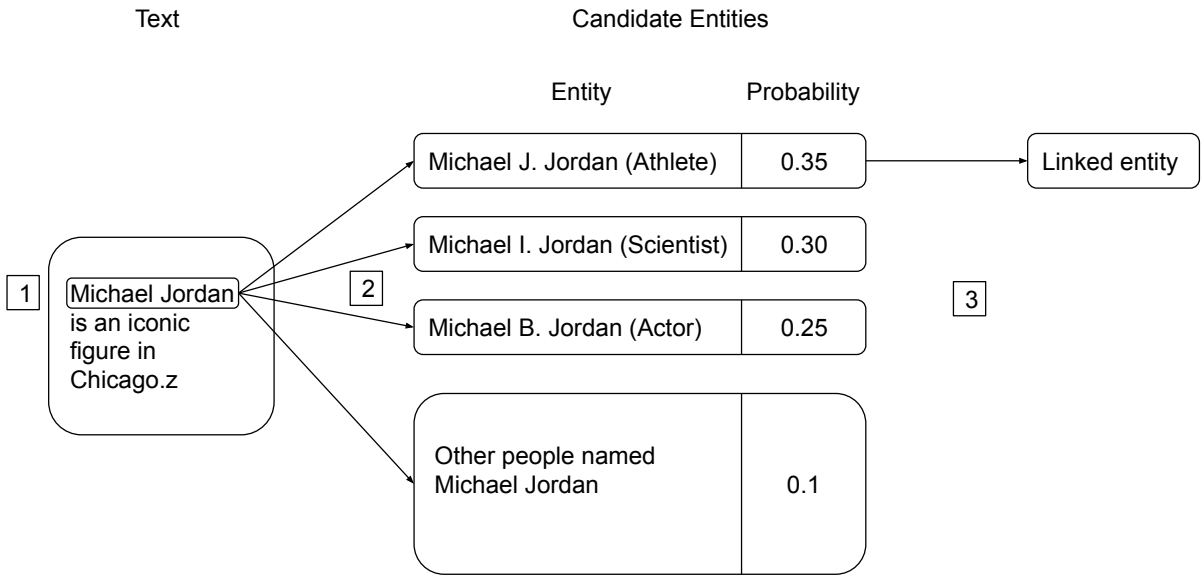

Figure 2.1: An illustration of the entity linking task.

In chapters 3 & 4, we use a specific cleaning challenge that is the linking of mentions (a word or a sequence of words) in a textual body to real-world entities in a knowledge base which is referred to as the entity linking task. The entity linking task usually goes through different stages. The entity linking task is usually performed through different stages. Given a snippet of text, we first identify the words or sequence of words that could potentially refer to a real-world entity, which is usually referred to as the entity extraction process. This step can be performed by following a named entity technique such as the Standford NER tool [37] or by finding the most extended sequence of words that could potentially refer to a real-world

entity using a dictionary-based lookup approach [39]. The second step involves generating different real-world entities candidates for the extracted mention from the text. The main common approach for generating the candidates is using a dictionary-based approach (e.g., disambiguation page of Wikipedia) [36, 39, 67, 81]. Moreover, each candidate is accompanied by a linking probability that measures its probability of linking to the extracted mention. Another common approach for candidate generation is by using search engines to retrieve the top-k documents by using the mention as the query term [47]. Figure 2.1 illustrates the stages of the entity linking task.

The last stage of the entity linking task is choosing the real-world entity that links to the mention. Techniques to disambiguate entity mentions have been extensively studied over the past decade. Different proposed approaches in the literature mainly use a knowledge base such as Wikipedia to facilitate the entity disambiguation process [36, 39, 67, 72, 74, 80, 81, 82]. Other approaches resort to machine-learning techniques and supervised learning algorithms [45, 89] for the entity linking process. Other works incorporate the feedback of the user in the system in order to correctly identify the actual real entity for ambiguous mentions using a crowdsourcing approach [31].

2.3 Query-Driven Cleaning

There has been a growing body of work when it comes to query-driven or query-aware data cleaning [10, 11, 19, 21, 22, 27, 42, 63, 64, 68, 87] where the data cleaning task is integrated within the query execution plan allowing for cleaning a subset of the database rather than the entire database. In [10, 11] authors provide a framework for integrating entity resolution with query processing efficiently to answer complex select-project-join style queries by smartly selecting tuples that need to be cleaned in order to return an answer for the query. The framework takes advantage of the predicate's selectivity that allows for

efficient and cost-aware execution. They use blocking techniques to guide the cleaning task and divide the data into smaller chunks to be efficiently handled. Daisy [43] looks at the problem of fixing denial constraint violations by relaxing the query results while involving the cleaning process inside the query execution plan. Their goal is to pinpoint the cleaning process's placement in the query execution plan to avoid cleaning unnecessary tuples.

Other works integrated the cleaning step in the case of uncertain query processing in order to reduce uncertainty [22, 64, 68]. The work in [22] focuses on cleaning uncertain data for selection queries with a range predicate and queries that ask to find a min or max value for an uncertain database. Given a budget, they aim to reduce the uncertainty of the answer by identifying the set of tuples, when cleaned, that will have the highest impact on the quality of the result.

Exploiting top-k query semantics driven by lowering data cleaning cost has been considered in prior work as well. The most related such work is that of Verroios and Garcia-Molina [87] where authors considered the problem of top-k over entity resolution. They proposed an interactive algorithm for Locality Sensitive Hashing that essentially exploits the top-k semantics to cut down the cost of blocking only to clusters that are "dense" so that the blocking cost per tuple on "sparse" areas of data is very small. Another work exploring top-k selection query in the context of cleaning is [68] where they work within a cleaning budget to reduce the uncertainty of the top-k query answer on an x-tuple style database. The cleaning process is associated with a cost and a probability of success. The approach is based on choosing an optimal set of tuples to clean that will improve the answer quality.

Crowdsourcing has been used to answer top-k selection queries such as in [27, 63, 64] where an expert or a crowd of experts are given questions regarding the order of the records in the list and based on feedback systems update the database and answer the top-k query efficiently. In addition, a notion of budget in terms of the number of questions or inquires is enforced due to the high cost of the crowdsourcing process.

Chapter 3

TQEL: Framework for Query-Driven Linking of Top-K Entities in Social Media Blogs

Social media blogs usually contain ambiguous *mentions* that could potentially refer to real-world *entities*. In this chapter, we study how top-k queries in the context of social media blogs can efficiently be evaluated. Given a collection of social media blogs $\mathcal{T}$ that contain a number of mentions, the goal is to identify the top-k real-world entities that are mentioned the most in $\mathcal{T}$.

Consider, for example, a user creates a collection of tweets (by sampling the public Twitter API and/or by running keyword/phrase queries using Twitter's query interface [62, 78]) and would like to characterize $\mathcal{T}$ based on the top-k entities of a certain category – e.g., top-k "movies", top-k "athletes", or top-k "locations". If the text/metadata in the tweets explicitly identified real-world entities, we could look up the associated categories in knowledge bases such as DBpedia [13] or Wikipedia [4] to appropriately tag the tweets with the corresponding

categories. Then, finding the top-k entities in $\mathcal{T}$ in the context of the category of interest (e.g., movies, politicians, athletes) would be straightforward; we would simply count the number of times an entity corresponding to the category of interest is mentioned in $\mathcal{T}$, and choose the k entities with the highest counts.

However, entities are not explicitly associated with the tweets. Instead, entity extraction and lookup functions [80] are used to determine them. Such functions take as input the set of words, as well as, metadata associated with the tweet, and return a set of a sequence of words (referred to as a mention) that could correspond to real-world entities [80, 39]. For instance, a lookup function applied to a tweet *"Black Panther won an Oscar!"* may identify two mentions "Black Panther" and "Oscar". Such mentions rarely correspond to a unique real-world entity. The study in [55] shows that each mention, on average, could correspond to 13.1 real-world entities, each of which is associated with different categories. For instance, the "black panther" mention could either refer to a movie or an animal corresponding to the "movies" or "animals" categories respectively. Before the top-k entities within a given category can be identified, we must first disambiguate such mentions. Once mentions have been linked to the correct entities, the top-k entities in the tweet collection within the category of interest can be easily determined.

The challenge in implementing top-k queries arises since entity linking functions are expensive. As such, applying it to the entire collection requires significant computation, leading to long latency in the results. Moreover, such computation is also wasteful since it requires linking mentions that are simply not of interest, i.e., those that are clearly not part of the top-k results. One strategy to overcome the challenge is to simply associate the mention with every possible entity rather than running the entity linking function. For example, "black panther" in the above tweet will be associated with the "movie" entity and the "animal" entity as well. Then we can simply return the top-k set after aggregating the number of occurrences for each entity. For instance, for the tweets shown in Table 3.1 (represented

in the figure 3.1), a query for top-2 movies will return as a result: "Black Panther (2018 movie)" & "Black Panther (1977 movie)" since each of them has 4 occurrences.

Clearly, the above strategy results in erroneous answers since the right response in this example should have been "Black Panther (2018 movie)" & "Beautiful Creatures" had we fully disambiguated all the mentions. When adopting the same strategy for top-10 movies, top-10 politicians, and top-10 locations on our dataset, the average precision of the results was 0.16 while the rank distance was 377.9! We explain how to measure precision and rank distance in section 3.5.

In this chapter, we propose TQEL (Top-k Query processing using Entity Linking), a framework that exploits the query semantics for adaptive application of entity linking to only a subset of the mentions that are required to answer the query. The TQEL framework can be invoked to answer the top-k query exactly. The resulting implementation, referred to as TQEL-exact, improves upon the naive strategy of fully linking all mentions (prior to query execution), to linking only a small subset that could determine the top-k results. TQEL-exact returns exactly the same answer as would be returned by the naive strategy, though at a fraction of the expense.

While TQEL-exact improves upon the naive mechanism, it, nonetheless, incurs overheads concisely when there are a large number of entities with possible frequency counts that are large and also close to each other in value competing to be in the top-k spots. As such, TQEL-exact is not suitable for queries that require faster responses due to delays it incurs to prove that each element is truly in the top-k. The main contribution of this chapter is an approximate approach which we refer to as TQEL-approximate.

TQEL-approximate, instead of continuing to link entities until it guarantees that it has found a top-k result, stops much earlier as soon as it can establish that the entities in top-k result, it has found so far, have a probability of being in the answer above a user-specified threshold

τ. To achieve this, TQEL-approximate uses two statistical models to efficiently deliver the answer.

First, it estimates the number of occurrences of each entity by applying normal approximation statistics on the distribution of mentions associated with a given entity where such an estimation can be computed efficiently. Based on the normal approximation step, TQEL-approximate then decides whether to link mentions further, or whether it expects that a *potential* answer to the top-k query has been found. It then invokes the validation step which uses the Monte-Carlo simulation technique that generates N samples of the possible worlds and then calculates the probability of each entity being in the top-k using the N samples. If verification fails, TQEL-approximate performs more entity linkings until the verification step's stopping condition is met.

Note that normal approximation estimation provides a fast mechanism to predict that an answer has been found, however, it is not used to verify answers but rather the Monte-Carlo simulation is used for that purpose.

Additionally, TQEL doesn't require the distribution of mentions' linking probabilities to be normally distributed although the filter works best when they are. In effect, TQEL-approximate allows users to trade quality with latency. Our results over different data sets show that TQEL-approximate achieves an order of magnitude improvement over TQEL-exact – it finds top-k answers with confidence as high as 95% within 5-10 seconds, while an exact approach takes 100-300 seconds for the same query on the same machine.

In summary the main **contributions** of this chapter are:

- We develop a framework that uses entity linking to evaluate top-k queries efficiently (section 3.1 & 3.2).
- We propose two heuristic approaches that return an exact answer for the top-k queries

(section 3.3) (TQEL-exact).

- We propose an approximate approach that relaxes the quality of the result by returning a top-k answer with probabilistic guarantees (section 3.4). (TQEL-approximate)
- We experimentally evaluate our framework extensively using three datasets. We illustrate the results of the two approaches and provide a detailed evaluation of the approximate approach (section 3.5).

3.1 Preliminaries

In this section, we will present the needed preliminaries that form the basis for the TQEL framework.

3.1.1 Dataset and Required Functions

Social Media Datasets

Let $\mathcal{T}$ be a collection of tweets $t_1, t_2, ..., t_n$, M be the set of mentions in tweet t_i's text $m_1^i, m_2^i, ..., m_{|M|}^i$., E be the set of entities $e_1, e_2, ..., e_{|E|}$. and each entity e_x is associated with one or more category $c_1, c_2, ..., c_{|C|}$. Tweets, in general, contain the text along with metadata about the tweeter and the tweet itself such as username, timestamp, and location of the tweet. We identify a word or sequence of words in the body of the text as a mention m_j^i if it could *potentially* be linked to real-world entity e_x. These entities could be of any category such as movies, people, or locations. At the ingestion time, tweets will undergo a prepossessing step in order to extract and normalize the text in the tweet. This is executed by removing the unnecessary words in the body of the tweet such as stop words and Twitter's special notation (e.g., "rt").

TweetID	Tweeter	Text	Time	Location
t_1	u_1	**Black panther** ~~has~~ finally grossed $700 million domestically!	ts_1	l_1
t_2	u_2	**Emmy Rossum** ~~in~~ **Beautiful Creatures** ~~is~~ stunning	ts_2	l_2
t_3	u_3	**Pixar** makes the best animated movies	ts_3	l_3
t_4	u_4	**La La Land** best movie ~~of all~~ time!	ts_4	l_4
t_5	u_5	Another live reading ~~of a~~ random book ~~in my tbr~~ pile, **Beautiful Creatures**	ts_5	l_5
t_6	u_1	**Black Panther** won an **oscar!**	ts_6	l_1
t_7	u_6	~~A lot of~~ athlete will ~~be in~~ **space jam 2**	ts_7	l_6
t_8	u_7	Gonna watch **stardust** ~~and~~ **beautiful creatures**	ts_8	l_7
t_9	u_8	~~a~~ **black panther** ~~was~~ photographed ~~in~~ **Africa** last week	ts_9	l_8
t_{10}	u_3	**Emma Stone** ~~was~~ amazing ~~in~~ **la la land**	ts_{10}	l_9
t_{11}	u_9	**Marvel** made **michael jordan** famous ~~in~~ **black panther**	ts_1	l_{10}

Table 3.1: Sample of prepossessed tweets. The crossed sequence of words is dropped in the preprocessing step. The bolded sequence of words represents a mention that links to an entity. Underlined mentions refer to an entity from the movies category.

Query Model

We model our top-k query $\mathcal{Q}^k$ where it is executed on top of the tweets collection $\mathcal{T}$. Each query $\mathcal{Q}^k$ contains a category filter c_g such as people or locations. The main concept is to find k entities, that are associated with category c_g, mentioned the most in the tweets collection $\mathcal{T}$. As an example, an editor who is working for a magazine that rates and reviews movies is asked to provide a list of the top-2 movies mentioned in a given tweet collection $\mathcal{T}$. The query $\mathcal{Q}^2$ is to find 2 entities mentioned in $\mathcal{T}$ where entities are associated with the category movies and the number of times they were mentioned in $\mathcal{T}$ is larger than the rest of entities. This query can be evaluated using our query model.

Entity Extraction and Lookup

An entity extraction and lookup function LU(t_i) takes tweet t_i and returns a set of 3-tuples {<m_j^i, e_x, $p(m_j^i, e_x)$>} where m_j^i is the j_{th} mention in the tweet t_i, e_x is a possible entity for the mention and $p(m_j^i, e_x)$ is the probability m_j^i links to e_j. The sum of all probabilities for a specific mention m_1^i is 1. For example, let t_1 be "Black Panther won an oscar!", LU(t_1) returns <m_1^1, Black Panther (2018)>, 0.74), <m_1^1, Black Panther (Animal)>, 0.12>, <m_1^1, Black Panther (1977)>, 0.10>, <m_1^1, OTHER>, 0.02>. where OTHER corresponds to linking m_1^1 to no entity.

This process is known as the candidate list generation for a specific mention. Most of the approaches rely on name dictionary based techniques [33, 39, 45, 81] which generates a map where keys are a list of all possible mentions and values are a set of entities that could potentially refer to the key. This limits the generation of mentions to only the the ones that are stored in the map. Another approach uses search engines [47, 69] as the vehicle for finding candidate entities where they return the top hits for each mention.

Entity Linking

Given a tweet t_i, mention m_j^i and a linking function (EL), the entity linking function links

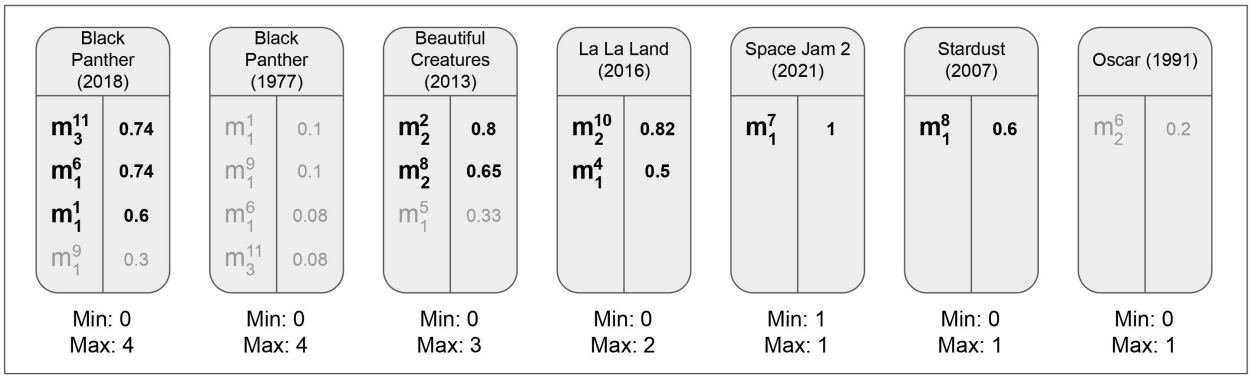

Figure 3.1: Entity list representation of Table 1. Bolded mentions represent mentions that links to associated entity. Faded mentions do not link to the associated entity.

mention m_j^i in t_i with a real-world entity e_x. For example, Given the tweet "Black Panther won an oscar!", and the mention "Black Panther", EL will link the mention "Black Panther" to Black Panther (2018) entity.

The entity linking function is considered a cost-heavy function that require complex computations to achieve the sought precision when linking a mention in the text of a tweet with an entity. Some entity linking functions might also require analyzing and linking entities of the previous tweets from the same user along with any tweets from the same location and time in order to accurately pinpoint the mentioned entity [39].

Most of the techniques takes advantage of linking the mentions to their target entities in a known knowledge base (e.g. Wikipedia) [36, 39, 81, 82] by leveraging the Wikipedia topology and the Wikipedia articles along with the text of the tweet. Other techniques use a crowd sourcing approach in order to link the identified mentions [31] to an entity. Other approaches adopt a machine learning approach where supervised and semi-supervised learning algorithms are run on top of annotated datasets to find the best candidate entity for each mention [89].

Since the entity linking functions require complex computations to get the best candidate, it is considered a bottleneck in any query processing system that relies on entity linking to answer the asked query. In our setup, we consider the functions LU(t_i) and EL(t_i, m_j^i) black box functions that could be replaced with any entity extraction, lookup and linking functions.

3.1.2 Exact Top-k Definitions

Answer Semantics

Let E_{c_g} be all the possible entities that are associated with category c_g and let $count(e_x)$ be

the number of occurrences of e_x in $\mathcal{T}$ as identified by the entity linking function. Let $A(\mathcal{Q}^k)$ be the answer of applying $\mathcal{Q}^k$ over $\mathcal{T}$. We call $A(\mathcal{Q}^k)$ a valid answer iff:

- $A(\mathcal{Q}^k) \subset E_{c_g}$.
- $\forall\ e_x \in A(\mathcal{Q}^k)$, e_x appears only once in $A(\mathcal{Q}^k)$.
- $\forall\ e_x \in A(\mathcal{Q}^k) : \nexists\ e_y \in (E_{c_g}$ - $A(\mathcal{Q}^k))$, s.t. $count(e_y) > count(e_x)$

Given a query $\mathcal{Q}^2$ looking for the top-2 movies in tweets in Table 3.1 and represented in figure 3.1, answer $A(\mathcal{Q}^k) = \{$Black Panther (2018), Beautiful Creatures (2013)$\}$ and answer $A(\mathcal{Q}^k) = \{$Black Panther (2018), La La Land (2016)$\}$ are both valid.

Standard Solution

To generate a valid answer for $\mathcal{Q}^k$ over $\mathcal{T}$ for a specified category c, the standard solution will link each mention in $\mathcal{T}$ by calling the EL function. Entities that are associated with category c_g will be filtered then the k entities with largest number of mentions are returned.

This approach requires the query executor to call the linking function on every mention in $\mathcal{T}$. Such an approach will be inefficient and ineffective when dealing with datasets that are ingested at high volume and velocity such as social media posts. We also don't need to disambiguate every mention in the data set in order to reach a correct answer due to the semantics of the top-k operator. In section 3.3, we will describe more efficient algorithms to compute the top-k answers that exploit query semantics to significantly reduce the number of entity linking functions invoked.

Problem Definition

Given a top-k query $\mathcal{Q}^k$ on top of a collection of tweets $\mathcal{T}$, an entity lookup function LU, an entity linking function EL and a category c_g, efficiently generate an answer $A(\mathcal{Q}^k)$ that

satisfies the answer semantics of the exact approach.

3.1.3 Approximate Top-k Definitions

Possible Worlds

A possible world w_a in our context consists of an assignment of each ambiguous mention to one of the possible entities based on its linking probability. Thus, the probability of the possible world p(w_a) can be computed as a product of such assignments. For example, world w_1 can consist of the following assignments based on figure 3.1, {(m_3^{11}: Black Panther (2018)), (m_1^6: Black Panther (2018)), (m_1^1: Black Panther (1977)), (m_1^9: OTHER), (m_2^2: Beautiful Creatures (2013)), (m_2^8: Beautiful Creatures (2013)), (m_1^5: OTHER), (m_2^{10}: La La Land (2016)), (m_1^4: OTHER), (m_1^7: Space Jam 2 (2021)), (m_1^8: Stardust (2007)), (m_2^6: OTHER)}. p(w_1) ≈ 0.0022.

Probabilistic Top-k Evaluation

Given a possible world $w_a \in \mathcal{W}$, we evaluate the top-k query on the world w_a. Let $ans(w_a)$ be the top-k answer set for w_a, then $p(e_x \in$ top-k answers of $(\mathcal{W}))$, abbreviated $tkp(e_x)$, can be calculated using the following equation:

$$tkp(e_x) = \sum_{a=1}^{|\mathcal{W}|} p(w_a) \cdot I(e_x, w_a) \qquad \text{s.t. } I(e_x, w_a) = \begin{cases} 1 & \text{when} e_x \in ans(w_a) \\ 0 & \text{when} e_x \notin ans(w_a) \end{cases} \tag{3.1}$$

Answer Semantics

Let $A(\mathcal{Q}^k)$ be the answer of applying $\mathcal{Q}^k$ over $\mathcal{T}$. We call $A(\mathcal{Q}^k)$ a valid answer iff:

- $A(\mathcal{Q}^k) \subset E_{c_g}$.
- $\forall\ e_x \in A(\mathcal{Q}^k)$, e_x appears only once in $A(\mathcal{Q}^k)$.
- $\forall\ e_x \in A(\mathcal{Q}^k), \quad tkp(e_x) > \tau$.

where τ is a user-defined confidence.

Our top-k query evaluation semantic is similar to PT-k [49] where only entities that have a top-k probability higher than a specific threshold τ will be included in the answer set. The order among the returned top-k answer set is ignored in our settings. Given the same query $\mathcal{Q}^2$ that looks for the top-2 movies in tweets in Table 3.1, A possible answer for $A(\mathcal{Q}^2)$ could be {Black Panther (2018), Beautiful Creatures (2013)} after the EL function has been called on some mentions. Note that there could be more than k entities that satisfy the answer semantics, however choosing any k entities from the answer set, if all mentions were linking, is sufficient and correct.

Problem Definition

Given a top-k query $\mathcal{Q}^k$ on top of tweets collection $\mathcal{T}$, and entity lookup function LU, an entity linking function EL, a category c and a threshold value τ, we aim to provide an approximate solution for the Query $\mathcal{Q}^k$ that generates $A(\mathcal{Q}^k)$ such that $A(\mathcal{Q}^k)$ follows the predefined answer semantics for the approximate approach and minimizes the number of calls to EL.

3.1.4 Top-k Example Solution

Following the exact top-k definition, $\mathcal{Q}^2$ for finding the exact top-2 movies in Table 3.1, the entity linking function is called on the following mentions set $\{m_3^{11}, m_1^6, m_2^2, m_2^8\}$. We can

safely return the set {Black Panther (2018), Beautiful Creatures} as the answer without linking any more mentions as the answer satisfies the answer semantics. To solve the same query using the approximate approach where $\tau = 0.9$, the calculation of $tkp(e_x)$ for all e_x is required. In our example the top-k probabilities are {(Black Panther (2018): 0.93), Beautiful Creatures (2013): 0.84), La La Land (2016), Space Jam 2 (2021): 0.27), (Stardust (2007): 0.16), (Black Panther (1977): 0.13), (Oscar (1991): 0.05)}. After calling the EL function on $\{m_2^2\}$, the top-k probabilities become {(Black Panther (2018): 0.93), Beautiful Creatures (2013): 0.91), ..}. Hence, we stop the execution and report the answer for $\mathcal{Q}^2$ as {Black Panther (2018), Beautiful Creatures} since it satisfies the approximate solution's answer semantics.

3.2 TQEL overview

This section will talk about the general approach of TQEL.

In traditional query processing and optimization, once a query plan is chosen based on the workload and different heuristic factors, the chosen query plan is fixed and will execute until it reaches its life cycle. However, in this framework we present an algorithm that is dynamic where it is adjusted based on the change of status of the objects that are being processed. The algorithm consists of multiple phases where the first phase acts as a preparatory step for the mentions to be linked to entities. It also consists of a thinking phase that generates the execution plan for the current iteration and an act phase that executes the plan. In Algorithm 1 and figure 3.2, We describe the steps that are followed by our framework in general regardless of the approach used to answer $\mathcal{Q}^k$.

Algorithm 1 TQEL Approach

procedure GETTOPK($\mathcal{Q}^k, \mathcal{T}, LU, EL$)
    $E\_LIST \leftarrow \{\}$
    for each $t \in T$ **do**
        $\{(m_j^i, e_x, p)\} \leftarrow \text{LU}(t_i)$
        $\{(m_j^i, e_x, p)\} \leftarrow \text{filterCategories}(\{(m_j^i, e_x, p)\})$
        $\text{addToLists}(E\_LIST, \{(m_j^i, e_x, p)\})$
    while !$\text{stoppingCondition}(\mathcal{Q}^k, E\_LIST)$ **do**
        $m_j^i \leftarrow \text{selectMention}(E\_LIST)$
        $e_x \leftarrow \text{EL}(m_j^i, t_i)$
        $\text{updateLists}(e_x, m_j^i, E\_LIST)$
    $\text{produceAnswer}(\mathcal{Q}^k, E\_LIST)$

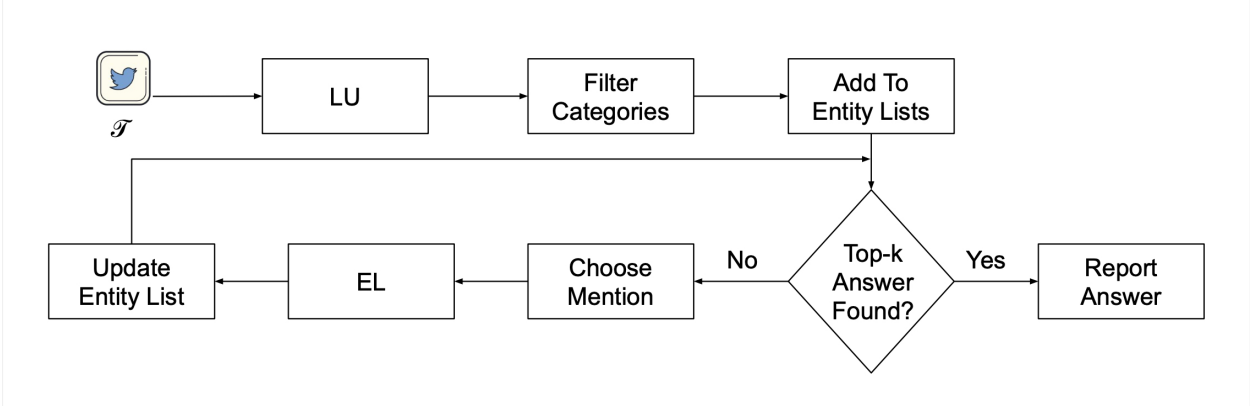

Figure 3.2: TQEL flow diagram

3.2.1 Preparatory Phase

In order to prepare the mentions for the thinking and execution phase, The algorithm iteratevely loops over every tweet t_i in $\mathcal{T}$ and calls the LU function on t_i to return a list of mentions with their possible entities and the linking probabilities. After that, the entities are filtered based on the category of the query $\mathcal{Q}^k$. Entities that are not associated with category c_g are removed and their linking probability are added to the OTHER probability for that mention.

The algorithm creates a list for every entity e_x associated with category c_g. The entity list (l_{e_x}) has an unresolved mention list which holds pointers to the mentions that could refer to entity e_x along with the linking probabilities. Mentions are sorted descendingly based on their linking probability allowing for instant access to mentions with highest & lowest probabilities if needed when selecting a mention to link. Every entity list l_{e_x} stores min & max counters that correspond to the number of linked mentions, after running the EL function, and the maximum number of mentions that could link to e_x respectively. Entity lists are held in another list and are sorted based on the max counter in a descending order as shown in figure 3.3. Additionally, every mention is represented as an object where it has a list of pointers to all the possible entities it could refer to. The probability that mention m_j^i does not link to any possible entity is (1 - $\sum_{x=1}^{y} p(m_j^i, e_x)$, where e_x is an entity associated with category c) and this is considered the OTHER probability.

3.2.2 Thinking & Execution Phase

Mention Selection

In order to save time in the top-k query execution, we need to intelligently select the mentions to link that will help in reaching an answer the top-k query efficiently. The mention to be

selected in the next iteration for dismabiguation is influenced by different factors (e.g. the current spot of the possible entities of m_j^i and the probability of linking mention m_j^i to entity e_x that is currently in the top-k). This process is different when answering the query using the exact approach or using the approximate approach and will be discussed in the coming sections.

Entity Linking and Updating Entity Lists

If mention m_j^i is linked to entity e_x, l_{e_x} is updated accordingly. This might cause the entity lists order to be changed which could help in reaching an answer for $\mathcal{Q}^k$. If mention m_j^i is shared among different possible entities, this will result in updating all the entity lists that have mention m_j^i in their unresolved mentions list as well. For example, in figure 3.1, m_1^1 could refer to Black Panther (2018) with probability 0.6, Black Panther (1977) with probability 0.1 or OTHER with probability 0.3. After calling the EL function on m_1^1, we find that it refers to Black Panther (2018). When updating the lists, it is added to the list of Black Panther (2018) and deleted from the list of Black Panther (1977). After linking some of the mentions, the min & max of the affected entity lists will change causing the order of

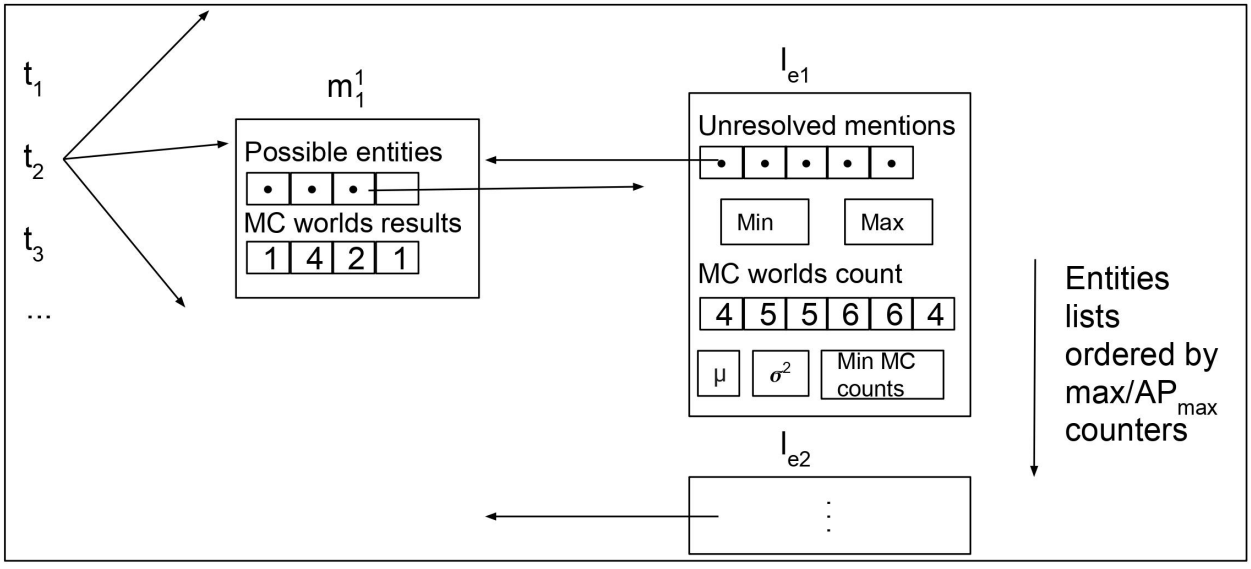

Figure 3.3: TQEL data structures.

entity lists to be adjusted accordingly.

Stopping Condition

In this step, the algorithm checks if a solution for $\mathcal{Q}^k$ is found and can be validated. This test will be different for the exact approach as well as the approximate approach and will be discussed in detail in sections 3.3 & 3.4.

3.3 TQEL-exact Approach

In this section we will present the exact approach of the framework in order to answer the top-k queries. The exact approach of the top-k query guarantees that the returned answer follows the answer semantics discussed in section 3.1.

Finding the minimal set of mentions to link in order to find the exact answer set for $\mathcal{Q}^k$ is difficult it depends on the outcome of the entity linking function which is unknown prior to execution. Furthermore, minimizing the number of entity linkings in the expected sense requires enumerating an exponential search space and hence not practical. Moreover, even if such approach is found, it may still require large number of entity linkings to find the exact answer resulting in unacceptable response time. Nonetheless, specifying TQEL-exact will help us explain TQEL-approximate in the next section. We therefore, introduce two distinct heuristics that can be used to implement TQEL-exact. Although the proposed heuristics achieves better results compared to the naive approach (linking all mentions), we present such heuristics briefly since it is not the main contribution of our work. To specify TQEL-exact, we specify how to implement *stopping condition* and *mention selection*.

3.3.1 Stopping Condition

Checking of the stopping condition, in TQEL-exact is straightforward. The framework maintains for each entity list the minimum and maximum possible counts of mentions associated with the list. The stopping condition simply checks to see if there exists k entity lists that have min counters $\geq$ max counters of the rest of the entity lists. Since the entity lists are sorted in a decreasing order according to their max counters, we can check for the stopping condition by checking if the min values of the first k lists are greater than or equal to that of $k+1$ entity list.

3.3.2 Mention Selection

The goal of mention selection is to identify the next mention to link that will enable the stopping condition to be reached as quickly as possible. Our mention selection algorithms are based on the intuition that the entity lists with the k highest max values are likely amongst the real top-k results. To be able to prove these entities to satisfy the stopping condition, we will need to increase the min values of these entities to become larger than the max values of other entities. This can be achieved by linking the mentions that refer to such entities. Thus, our heuristic algorithms choose from the mentions that refer to entities with k highest max values.

An alternate approach could have been to use entity linking to reduce the max values of entities that are not in the top-k, such that the max values go below the min values of the top-k answer set. Such an approach, however, will require us to link mentions in the entity lists that are not from the top-k lists which might require significantly larger number of calls to the entity linking function if the k is much smaller than the number of entities E which is usually the case. Other heuristic functions use a hybrid approach that adapts to different situations and uses the approach that has a higher chance of reaching an answer faster than

the other approach.

We, thus, briefly present two heuristic strategies referred to as *greedy*, and *benefit-based*, where *greedy* follows the first approach and *benefit-based* uses a hybrid approach.

Greedy Approach

In this approach, TQEL-exact greedily choose the mention that has the highest linking probability from the entity list with the highest max counter. For example, in figure 3.1, mention m_3^{11} will be chosen. After the mention is linked, the max and min counters of each list will be updated along with the list order according to the entity linking function result. It will stop linking mentions from entity list e_i when $min(l_{e_i}) \geq max(l_{e_{k+1}})$ given that all the entity lists are sorted descendingly according to their max value. The algorithm stops after the stopping condition is met for the exact top-k solution.

Benefit-based Approach

In this approach, we use a function that helps in identifying a mention, if linked, that helps in reaching an answer to $\mathcal{Q}^k$ faster. Such function uses an optimistic estimation of the number of calls to entity linking needed (ENL) to prove that the k entities with the highest max counter is the top-k result set. It may either choose to link mentions from the first k entities with the highest max scores to prove they link to such entities or choose entities from the other entity lists to prove that they do not link to their associated entity. we calculate ENL as follows:

$$ENL = \sum_{i=1}^{k} maxof(max(l_{e_{k+1}}) - min(l_{e_i}),\ 0) \tag{3.2}$$

where maxof is a function that returns the max of two numbers.

Since k is generally way larger than the number of entities, our intuition is that cleaning the entities with the highest max counters in order to prove they truly are in the top-k answer set will require less unneeded calls to the EL functions. For example, let us consider a top-2 query that looks for the 2 entities with the highest count and the total number of entities is 100. And furthermore, consider that entities e_1, e_2 ... e_{100} are ordered based on their max counter descendingly. Moreover, since TQEL works in an iterative fashion, we want to choose a mention that brings us closer to the answer in each iteration based on the mention's linking probabilities.

If we choose a mention that is associated with e_{50} for example, this will not have any impact on the direction of cleaning for the next immediate iterations. Remember that our stopping condition requires finding 2 entities that have min counters > max counters of the other 98 entities. Therefore, if e_{50} is not in the answer set this will be considered a wasteful linking. However, if e_{50} is in the answer set for the query, we still would have to clean enough mentions in the first 49 entities to prove that they do not refer to their corresponding entities causing their max counter to be lower than the min counter of e_{50}. So based on this intuition, we believe that ENL would be an estimation of the least number of needed linkings but nonetheless, the actual number of linkings could be more than that.

Let $ENL(S)$ be the the expected number of calls to entity linking function for the given current entity lists and $ENL(S^{m_j^i,e_x})$ be expected number of calls to entity linking function when m_j^i links to e_x after calling the entity function on m_j^i. Then we choose the mention with the highest benefit function score (shown in equation 3.3). If an entity list is guaranteed to be in the top-k answer set (i.e. it has a min counter $\geq max(l_{e_{k+1}})$), then mentions that are associated with such entity will not be considered for linking.

$$\begin{aligned} Benefit(m_j^i) = ENL(S) - (\sum_{m_j^i \in l_{e_x}} (p(m_j^i, e_x) \\ \times ENL(S^{m_j^i,e_x})) + p(m_j^i, OTHER) \times ENL(S^{m_j^i,OTHER})) \end{aligned} \tag{3.3}$$

The benefit-based approach can be optimized by limiting the calculation of the benefit function to be on mentions that indeed will reduce the value of ENL if linked. Such optimization can be achieved by calculating mentions from the first $2 \times k$ entities as they are the only mentions that will reduce ENL in the next iteration.

For example, using the benefit function for mentions in figure 3.1, ENL(S) = 6 and mention m_3^{11} has a benefit score of: 6 - (0.74 * 5 + 0.08 * 5 + 0.18 * 6) = 0.82. On the other hand, mention m_2^6 has a benefit score of: 6 - (0.2 * 6 + 0.8 * 6) = 0. Therefore, choosing m_3^{11} is helpful, at this stage, while choosing m_2^6 is wasteful.

3.4 TQEL-approximate Approach

In this section, we describe our approach to linking mentions to entities in order to evaluate top-k queries with probabilistic semantics as defined in section 3.1. In particular, we specify how we implement the three main functions in Algorithm 1, viz., functions to check the *stopping condition*, *select mention*, and *update lists*.The latter two are straightforward once we have developed our strategy for checking the stopping condition. We, thus, focus the discussion to checking the *stopping condition* and will briefly describe the *select mention* and *update list* functions at the end.

In TQEL-approximate, the stopping condition is weaker compared to that for the exact approach. In particular, TQEL-approximate stops early when it has identified k entities whose *tkp* is above a user specified threshold τ. One can determine the probability of an entity being in the top-k from the linking probabilities of each mention to correspond to given entity. If the stopping condition is not met, TQEL-approximate will iteratively link additional mentions by calling the entity linking function. Linking mentions reduces uncertainty by resolving the entity that the mention refers to. This, in return, causes uncertainty

in the counts associated with the entities to be reduced which could result in the stopping condition to be met.

Checking the stopping condition in TQEL-approximate, however, is quite complex. It requires the enumeration of all possible worlds based on the linking probability of each mention and computing the probability of a particular entity to be in top-k. Since the number of such worlds is exponential, enumeration is not feasible and we are not aware of any efficient algorithm for processing probabilistic top-k entities over count, that runs in polynomial time, that can applied in TQEL.

To address the above, we use an estimation technique based on sampling the entire possible worlds space that allows us to check for the stopping condition (i.e., the probability of the result set to be in the top-k is above a user specified threshold τ) with high confidence, discussed below.

3.4.1 Checking for the Stopping Condition

Stopping condition in TQEL-approximate uses Monte-Carlo (MC) simulation similar to prior work on probabilistic query processing [52, 75]. MC simulation relies on randomness to generate a sample world w_a from the set of possible worlds $\mathcal{W}$. To implement MC simulation, we iteratively select a mention m_j^i and assign an entity to that mention based on the linking probability distribution of all possible entities e_x that could be referred to by m_j^i (including the probability of assigning mention m_j^i to OTHER). After assigning all the mentions, a sample world w_a is generated and the result of the top-k query is computed. Given N sample worlds, we can compute the top-k answers to $\mathcal{Q}^k$ in those worlds, the results of which can be used to estimate the probability of an entity being in the top-k by using normal approximation to the binomial distribution. We compute $\hat{p_i}$ as in equation 3.4 and use it to

find the confidence interval for the entity's probability of being in the top-k.

$$\hat{p_i} = \frac{\text{number of times } e_x \text{ appears in top-}k}{N} \tag{3.4}$$

$$lcb = \hat{p_i} - z \times \sqrt{\frac{\hat{p_i} \times (1 - \hat{p_i})}{N}} \tag{3.5}$$

The z-score in equation 3.5 is a critical value for defining the confidence interval and is decided based on the user-defined τ using a z-score table and the lower confidence bound (lcb), based on a normal approximation to the binomial distribution, accounts for the uncertainty due to the finite number of MC samples.

TQEL-approximate checks the stopping condition by checking lcb against τ. If lcb is above τ, the stopping condition has been reached and the top-k answer can be returned. On the other hand, if lcb is below τ, TQEL-approximate continues with linking additional mentions which changes the uncertainty of entities associated with the mention. With enough number of iterations, the stopping condition is guaranteed to be met (e.g., for instance, when all the mentions have been linked). The time complexity of the above implementation of the stopping condition is is O($M \times N$), where M is the number of mentions and N is the number of MC samples.

The implementation of MC simulation to check the stopping condition described above suffers from two limitations. First, as specified, the approach has to pay the overhead of running a MC simulation (complexity order of $O(MN)$) after each linking operation. One could batch the number of mentions to be linked into batches of size b to reduce overhead of the running the simulation repeatedly. The size of the batch b (i.e., number of entity linking to be performed prior to checking the stopping condition again) has to chosen carefully. Sub-optimal choices could result in overhead. For instance, if b it is too large, we will pay

the overhead of executing additional entity linking operations (which are also expensive) unnecessarily since calling the stopping function without linking all the entities in the batch might also have met the required condition. On the other hand, if we select b to be too small, then we end up repeatedly paying overhead of the MC simulation. A more efficient algorithm would minimize the number of entity linking tasks performed prior to calling the stopping function while simultaneously guaranteeing that the stopping condition is met when it is called. Such an approach would minimize both the linking cost, as well as, the cost of MC simulation.

Another limitation is that each time the stopping condition is called, a new set of N samples are generated using the simulation. Such an overhead can be partially mitigated by observing that between any two iterations (i.e., calls to the stopping condition) the probability values of only a small fraction of mentions (i.e., the mentions selected for linking in the previous batch) have changed. We can speed up the simulation by leveraging the work done during previous iterations.

We next describe ways TQEL-approximate uses to overcome the above two limitations.

3.4.2 Exploiting Filters

To reduce the number of times we have to execute the expensive MC simulations, we employ a cheaper/less expensive filter to estimate if the stopping condition will be met by the MC simulations. Our motivation of invoking such a filter is analogous to the way blocking functions is used to reduce calls to the more expensive entity resolution functions in the data cleaning literature [11, 87], or cheaper predicates (e.g., simple selections) are executed first prior to calling more expensive tests in query processing literature [14].

In our setting, if a filter fails (i.e., determines that the stopping condition will not be met

by the MC simulation), we continue linking mentions to reduce uncertainly. We continue doing so until the filter condition is satisfied and the filter determines that it is likely that the result will meet the stopping criteria during the MC simulations. If the filter succeeds (i.e., determines the stopping condition would be met), the suggested results are fed into the MC simulation to check/validate if indeed the stopping condition has been reached. If not, then additional cleaning is performed and the filter is appropriately refined (i.e., made more conservative/tighter) and the process continues iteratively. The filter based algorithm is depicted in figure 3.4.

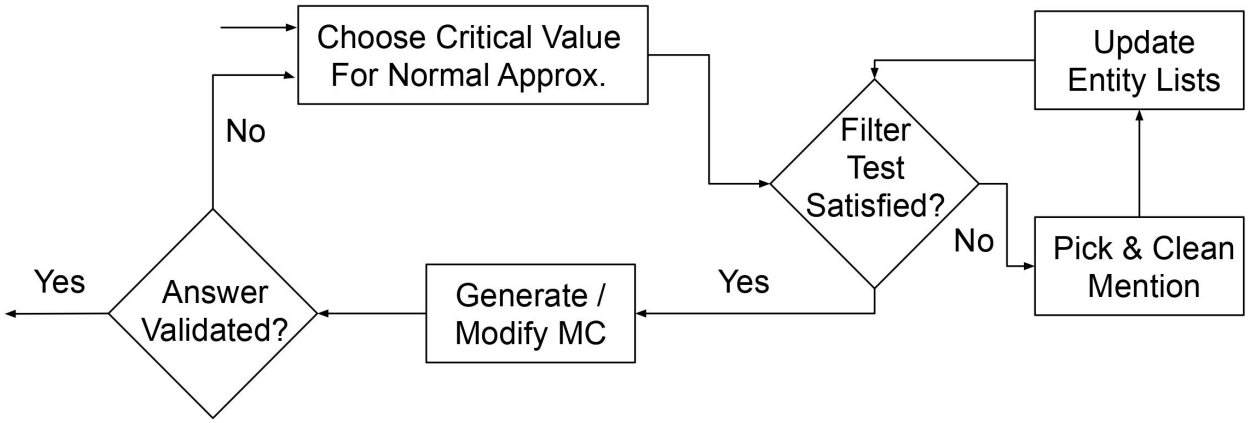

Figure 3.4: Stopping condition checking flow diagram

Effectively, our modified approach substitutes the more expensive MC simulation by a (much cheaper) filter until the algorithm reaches the point where based on filter execution we can be fairly confident that the stopping condition has been reached. To design an effective filter, we apply normal approximation statistics on that the linking probability of mentions associated with a specific entity e_x to compute the possible range of values for a given z-score (i.e., the z-value confidence interval, where z is a parameter associated with the filter). The confidence intervals can be computed efficiently while constructing the entity lists by computing the mean (μ) & standard deviation (σ) of e_x, as shown in equations 3.6 & 3.7. We treat the upper & lower bounds of the confidence intervals (shown in equations 3.8 &

3.9) as the approximate max & approximate min (AP_{max} & AP_{min}).

$$\mu_{l_{e_x}} = \sum_{m_j^i \in l_{e_x}} p(m_j^i, e_x) \tag{3.6}$$

$$\sigma^2_{l_{e_x}} = \sum_{m_j^i \in l_{e_x}} p(m_j^i, e_x) \times (1 - p(m_j^i, e_x)) \tag{3.7}$$

$$AP_{max} = \mu + x\sigma \tag{3.8}$$

$$AP_{min} = \mu - x\sigma \tag{3.9}$$

where x is the z-score associated with the desired confidence level. For example, for the confidence level of 68%, $x = 1$, while for 95%, $x = 1.96$, etc.

From the perspective of efficiency, the filter based on normal distribution assumption, takes roughly 1.7 ms to calculate the AP_{max} & AP_{min} for 10000 entities. In contrast, MC takes about 3.13 ms to generate only 1 world sample for 10000 mentions, which for a sample size of say a 1000 would take approximately 3 seconds.

After calculating the AP_{max} & AP_{min} for all entity lists, the filter checks if a solution is reached based on its calculated approximation. Given that we sort the entity lists descendingly based on their AP_{max} values, if there exists k entity lists with $AP_{min} \geq AP_{max}$ of the entity that is in the $k+1$ position then the condition is met and the suggested result is sent to MC simulation for validation.

Choosing the Confidence Intervals

Note that the choice of x in the equations 3.8 & 3.9 does not influence the correctness of the approach since, in our case, the filter is only used as a hint, and we still perform the Monte-Carlo simulation for validation of the top-k answer. However, the value x plays an important role in determining the number of calls to entity linking functions and the cost of generating and updating the MC simulation results. Smaller the value of x, smaller the confidence interval, and, lesser the number of calls to entity linking function in order to meet the filter condition (i.e., there exists k entities such that their $AP_{min} \geq AP_{max}$ of the rest of the possible entities).

Conversely, larger the value of x, more entity linking functions will need to be called before the filter condition is met. Thus, choice of x plays the key role in determining the number of entity linking functions called and the number of times MC simulation is invoked. An optimal value $x*$ would allow for enough entity linking functions to be called such that the MC test succeeds when called for validation.

To find such an $x*$, we first introduce an estimation of the number of calls to the entity linking function in order to find a candidate top-k answer (ENC(α_x)) given that x is the z-score for the confidence α_x. Given uncertainty of linking function, estimating such a number is complex. We use a heuristic estimation that greedily favours mentions that are associated with the k entities having the highest AP_{max} and estimate the number of mentions to be cleaned on those lists in order to report them as a candidate top-k result. The estimation is as follows:

$$ENC(\alpha_x) = \sum_{i=1}^{k} \frac{(maxof(AP_{max}(l_{e_{k+1}}, x) - AP_{min}(l_{e_i}, x),\ 0)}{1 - \frac{\mu_{e_i}}{max(e_i)}} \tag{3.10}$$

Where $AP_{max}(l_{e_i}, x)$ is the $AP_{max}(l_{e_i})$ if x is the z-value.

$ENC(\alpha_x)$ estimates the number of EL calls needed to shift the μ of the first k entities, given they are sorted on their AP_{max} values, so their $AP_{min} \geq AP_{max}(l_{e_{k+1}})$. Note that $ENC(\alpha_x)$ is monotonic in x, i.e. as x value ENC(α_x) also increases. To see this note that $AP_{max}(l_{e_i}, x)$ increases and $AP_{min}(l_{e_i}, x)$ decreases as value of x increases. Thus, the numerator in equation 3.10, increases with the increase in x while the denominator is a constant. Thus the functions is monotonic.

We, further, need to quantify the probability of success and failure when an arbitrary x value is chosen. Using normal approximation, let us say there are two entity lists l_{e_i} & l_{e_j} such that $AP_{min}(l_{e_i}) \leq AP_{max}(l_{e_j})$. The probability that its count is below $AP_{min}(l_{e_i}) = (\frac{1-\alpha}{2})$ if the mentions' probability follows normal distribution. Similarly the probability of the count of l_{e_j} to be above $AP_{max}(l_{e_j})$ is the same. We conservatively assume that when count of l_{e_i} is below $AP_{min}(l_{e_i})$ or the count of l_{e_j} is greater than $AP_{max}(l_{e_j})$, checking that filter test ($AP_{min}(l_{e_i})$ is $\geq AP_{max}(l_{e_j})$ would fail during validation stage. We, thus, estimate the probability of the success at the validation stage given the success of the filter to be: $(1 - 2 \times (\frac{1-\alpha}{2})) = \alpha$. Likewise, the probability of the validation failing given the success of the filter is (1- α).

We now calculate the cost of TQEL-approximate given an x value as $cost(\alpha_x)$, in the case that the validation step succeeds. Let C_{EL} be the cost of the entity linking function, C_{MC} be the cost of generating N MC simulations for one mention and C_V as the cost of running top-k query on all N worlds to verify the answer. $cost(\alpha_x)$ can be calculated using the following equation:

$$cost(\alpha_x) = C_{EL} \times ENC(\alpha_x) + C_{MC}(M - ENC(\alpha_x)) + C_V \tag{3.11}$$

We next consider the cost of TQEL-approximate for a given x value if the validation step fail. In such a case, we need to clean more mentions to get better results. We do that by

using a higher x value such that the confidence in our suggested top-k to succeed is higher. Such a scenario requires paying an overhead of linking more mentions, updating their linking results in the stored MC simulations and executing the verification process again. Consider that we begin the TQEL-approximate with a value of $x = x_1$, that results in failure at validation which prompts the algorithm to use $x = x_2$, which succeeds. We denote the cost of such an execution by $cost([\alpha_{x_1}, \alpha_{x_2}])$. We estimate the cost of choosing a value x_1 where the suggested top-k answer set based on x_1 estimation fails and we choose a higher value x_2 C$(\alpha_{x_1}, \alpha_{x_2})$ as follows:

$$\begin{aligned} &cost([\alpha_{x_1}, \alpha_{x_2}]) = cost(\alpha_{x_1}) \\ &+ (C_{EL} + C_{MC})(ENC(\alpha_{x_2}) - ENC(\alpha_{x_1})) + C_V \end{aligned} \tag{3.12}$$

In the formula above, note that the case when $\alpha_{x_2} = 1$ (that is, the confidence interval covers the entire distribution) is special. In such a case, to meet the filter condition, one will need to link all ambiguous entities and there will be no uncertainty in the top-k results. Thus, the process will not need to execute the final validation step. In such a case, the cost would be:

$$cost([\alpha_x, 1]) = cost(\alpha_x) + C_{EL}(M - ENC(\alpha_x)) \tag{3.13}$$

We next introduce a general equation for estimating the expected cost ($EC([\alpha_{x_1}, \alpha_{x_2}, ...\alpha_{x_n}])$) as follows:

$$\begin{aligned} &EC([\alpha_{x_1}, \alpha_{x_2}, ...\alpha_{x_n}]) = \alpha_{x_1} cost(\alpha_{x_1}) + (\alpha_{x_2} - \alpha_{x_1}) \\ &cost([\alpha_{x_1}, \alpha_{x_2}]) + ... + (\alpha_{x_{n-1}} - \alpha_{x_n}) cost([\alpha_{x_1}, \alpha_{x_2}, ...\alpha_{x_n}]) \\ &+ (1 - \alpha_{x_n}) cost([\alpha_{x_1}, ..., \alpha_{x_n}, 1) \end{aligned} \tag{3.14}$$

TQEL-approximate uses Algorithm 2 to find the optimal value of x. The algorithm searches for an x value such that EC$(\alpha_x) \leq$ EC$(\alpha_{x'})$ for any $x' \geq x$. The algorithm chooses an initial

Algorithm 2 Choosing the critical value

```
1: procedure ChoosingCriticalValue(M , C_EL, C_MC, C_V, τ, budget)
2:     α_x ← τ
3:     α_x' ← 1
4:     b ← 0
5:     while α_x < α_x' AND b < budget do
6:         if EC([α_x, α_x']) ≤ EC(α_x') then
7:             α_x' ← (α_x' + α_x)/2
8:             b ← b + 1
9:         else
10:            α_x ← α_x'
11:            α_x' ← 1
12:            b ← 0
13:    return α_x
```

x value such that α_x = user-defined τ. The intuition is that we want to choose an x such that the probability of the success when validating top-k results given the success of the filter $\geq$ user-defined τ. If we choose x $< \tau$, even if the filter succeeds the chance that the validation using MC simulation will fail is high. Given the monotonic nature of ENC(α_x), the algorithm performs a binary search to find a x‘ value such that EC($\alpha_{x'}$) < EC(α_x). If no such value x‘ is found before a searching budget b is exhausted, we choose x as the critical value. However, if an x‘ value is found before the b is exhausted, we choose x‘ as the new value for x and we rerun the algorithm.

Efficacy of Filter

The main role of the filter is to provide a cheaper mechanism for estimating the outcome of the query. However, the correctness of TQEL doesn't rely on this outcome for evaluating the top-k query but validates the query using MC simulations. Nonetheless, the efficacy of such filter is important as it effects the performance of the query (i.e. excessive EL calls and number of verification's performed).

Therefore, we have to propose a metric to measure how well this filter performs on different

queries, datasets and mention's probabilities distribution of the top-k entity lists. Please note that although the filter uses normal approximation it, however, doesn't require mentions probabilities to be normally distribution to work as it readjust itself by increasing the critical value when the validation step fails. In the extreme case, the critical value will be high enough to cover the entire distribution.

In order to properly measure the effectiveness of such a filter, we introduce a notion of accuracy in order to assess the goodness of using such a filter. The filter's test for expecting that an answer has been found can result in two answers: yes (in which case the validation using MC simulation is performed) or no (in which case TQEL will continue to link more mentions). Based on the outcome of the filter, and whether or not the MC stopping condition has been reached, we present 4 measurement labels: true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). TP refers to the number of times the filter test was accurately predicting the outcome of the validation step which is always 1. TN captures the number of times when both the validation using MC simulation and the filter ask for more linkings in order to validate the query (necessary number of EL calls). This can be calculated by calling the MC simulation after each linking to calculate exactly how many linkings are necessary. FP is captured by the number of times the filter test is satisfied but the validation using MC simulation is not. FN represents the excessive number of EL calls performed because of the filters failure of predicting that an answer has been reached using the validation step. This can be calculated by subtracting (the necessary EL calls) from the actual number of calls performed when using the filter. Hence, We define accuracy as the ratio of the sum of TP and TF determined by TQEL divided by the total sum of TP, TN, FP and FN. Accuracy is quite appropriate in our setting since both false positives and negatives result in expensive calls to MC simulation / EL function respectively. We report in experiment 7, a detailed accuracy evaluation for different baselines.

3.4.3 Monte-Carlo Simulation Implementation

In order to efficiently access the previous MC simulations, we keep the results of the previous runs in a vector of size N inside the mention object. For each value of the vector $v_1, v_2, ..., v_N$, we store the entity that is assigned to that mention for that specific sampled world. We, further, store the number of occurrences of each entity e_x in the N sampled worlds in a vector of size N where each value v_a corresponds to the number of assigned mentions for e_x in world w_a as shown in figure 3.3. We store such values to easily retrieve the number of occurrences for that entity in each sampled world w_a. We also store the minimum number of occurrences of all N worlds for each entity list.

In order to smartly generate MC samples for mentions in TQEL-approximate, we start by generating the samples for mentions of entity lists with the highest AP_{max} value. We iterate over the entity lists in a decreasing order of AP_{max} values in order to generate samples for the mentions which are associated with entities that are competing to be in the top-k answer. If k entity lists were found such that their minimum number of occurrences $\geq max(e_j)$ such that e_j is not an entity from the first k entities, we do not perform MC simulation process for e_j.

Answer Verification

We use equations 3.4 & 3.5 to calculate the lcb for the first k entities given that they are sorted based on their AP_{max} values. If for every entity, lcb $> \tau$, we return the first k entities as the answer to $\mathcal{Q}^k$. However, if the this step fails, we continue to link more mentions until the stopping condition is met.

3.4.4 Mention Selection

TQEL-approximate follows a benefit function approach similar to equation 3.3 where ENL is replaced by ENC. We define $ENC(S)$ as the expected number of calls such that the filter test passes given the current entity lists and $ENC(S^{m_j^i, e_x})$ as the expected number of calls needed for the filter to pass when m_j^i links to e_x. The benefit function finds a mention m_j^i such that if linked brings us closer to the top-k answer result. Note that chosen the entity with the highest linking probability, say 0.9, might not always be the best answer since the reduction in uncertainty, if it links to the desired entity, is quite low. Therefore, the benefit function takes into account the probability of success as well as the reduction in the expected number of EL calls.

3.4.5 Updating Lists and Approximations

AP_{min} and AP_{max} Maintenance

We can instantly calculate and update the AP_{min} & AP_{max} values based on the stored values of μ & σ for every entity list. Values of μ & σ are updated by adding the probability value of new mention m_j^i to μ and adding p(m_j^i)) * 1 - p(m_j^i)) to σ^2. Moreover, whenever we link mention m_j^i to an entity e_x, for every entity that was associated with mention m_j^i we update the values of μ & σ accordingly. To update entity e_x which is linked to mention m_j^i we add (1 - p(m_j^i)) to μ and subtract p(m_j^i)) * 1 - p(m_j^i)) from σ^2. For the other entities we subtract p(m_j^i)) from μ and subtract p(m_j^i)) * 1 - p(m_j^i)) from σ^2 as well.

Monte-Carlo Simulation Maintenance

In order to properly maintain the samples of the Monte-Carlo simulation algorithm, we store the results of the previous N runs to use ,if needed, in later stages. Whenever a mention m_j^i is linked to entity e_x, we update the number of occurrences of all the entities that could be referred to by m_j^i. We remove m_j^i from any entity that is not e_x and add it to e_x in all the runs too. If the mention m_j^i is not linked to any entity, then we remove m_j^i in all N runs of the possible entities.

3.5 Experiments

In this section we illustrate the wide range of experiments conducted using TQEL-exact & TQEL-approximate.

3.5.1 Experimental Setup

Datasets

We are using two tweets datasets that have been collected from Twitter's public API without any specification or focus on certain keywords, locations or topics. All these tweets are English tweets. We run the two heuristics of TQEL-exact & TQEL-approximate on both datasets and conduct different experiments to evaluate such approaches with different settings. We also use a synthetic dataset that have been generated by introducing a 10% uniform noise to the linking probability of every mention-entity pair using the small dataset. This can be done by multiplying a randomly generated real number between (0.9, 1.1) to every mentions linking probabilities. Afterwards, we normalize the probabilities in order to satisfy that the sum of probabilities equals 1. Our goal is to test the robustness of our

approach when the initial linking probability of a mention-entity pair is erroneous and study how TQEL-exact and TQEL-approximate would perform in such circumstances. For the TQEL-approximate we have chosen 10,000 to be the number of runs for the MC simulation for all of our experiments.

- **Small Dataset.** The first dataset contains 101,486 tweets and has been collected from April 6-April 7, 2018.
- **Large Dataset.** The second dataset contains 11,250,894 tweets and has been collected from May 30-June 9, 2019.

Approaches

In our experiments we use 6 baselines to compare them with TQEL-approximate approach and they are as follows:

- **Random Approach (random).** In this approach we will iteratively choose a random mention-entity pair to link until a solution is found. We enhance the efficiency of this algorithm by limiting the choice of mention from lists that are competing to be in the top-k answer.
- **TQEL-exact (Benefit Function) Approach** discussed in 3.3.
- **TQEL-exact (Greedy) Approach** discussed in 3.3.
- **TQEL-Approximate (Greedy) Approach.** In this approach, TQEL choose the mention with the highest probability rather than using the mention selection technique discussed in 3.4.

- **NOFILTER Approach.** In this approach, TQEL does not use the filter proposed in 3.4 but rather validates the query using MC Simulation after each EL call. This approach will follow the same mention selection technique for TQEL-Approximate.
- **MC-NOOPT Approach.** In this approach, TQEL does not apply the optimizations discussed in 3.4 but rather runs the Monte-Carlo simulation every time a validation is needed.

Knowledge Base

We have used Wikipedia [5] as the source of our KB in this experiment by indexing the whole Wikipedia dump using Apache Lucene [3] to make it easier to query titles, text bodies and other metadata. We have also tagged each Wikipedia article with categories fetched from DBpedia to be able to answer the top-k query with the category filter. We have only indexed the articles that are included in our queries and the indexing process took around a day and 20 hours.

Queries

The queries that are used for the experiments are top-k queries of different categories in DBpedia. We have used multiple categories from different levels in the DBpedia category hierarchy in order to control the selectivity of the query. The selectivity is corresponding to the number of mentions in all the tweets. For example, a 10% selectivity indicates that 90% of the mentions will be discarded as they do not have any possible entity that is associated with the category c_g. We also execute the query on different k-values and confidence scores. The categories that we used in these experiments have been selected from 3 different levels in the category hierarchy and it is as follows:

- **Top-level categories:** Agent, Work and Place. With an average selectivity of 55%.

- **Med-level categories:** Person, Musicalwork and Organization. The average selectivity is 31%.
- **Low-level categories:** Film, Song, Populatedplace, Artist, Athlete and Politician. The average selectivity is 13%.

Entity Extraction & Lookup Function:

In our experiment, we use simple dictionary-based entity lookup function that parses the tweet sequentially and identifies the largest sequence of words that matches an article in the KB. When there is a match, LU generates the possible candidates for the identified mention and gives each candidate a linking probability based on a score of different factors (e.g. page view count). LU takes roughly 0.5 milliseconds per tweet.

Entity Linking Function

In order to disambiguate a mention-entity pair we use TagMe API [36] that returns an entity for the mention in the tweet text along with a probability of linking. In our setting we use entity linking function as a determining function by assigning such mention to the entity if the linking function returns a probability of 0.5 or more and vice versa. The linking process takes on average 44 milliseconds.

3.5.2 Experiments Results

Experiment 1: Number of entity linking calls & execution time for different k values

In this experiment we show the effectiveness when using different strategies and the advantage that the TQEL-approximate approach provides as a function of k. We average the query

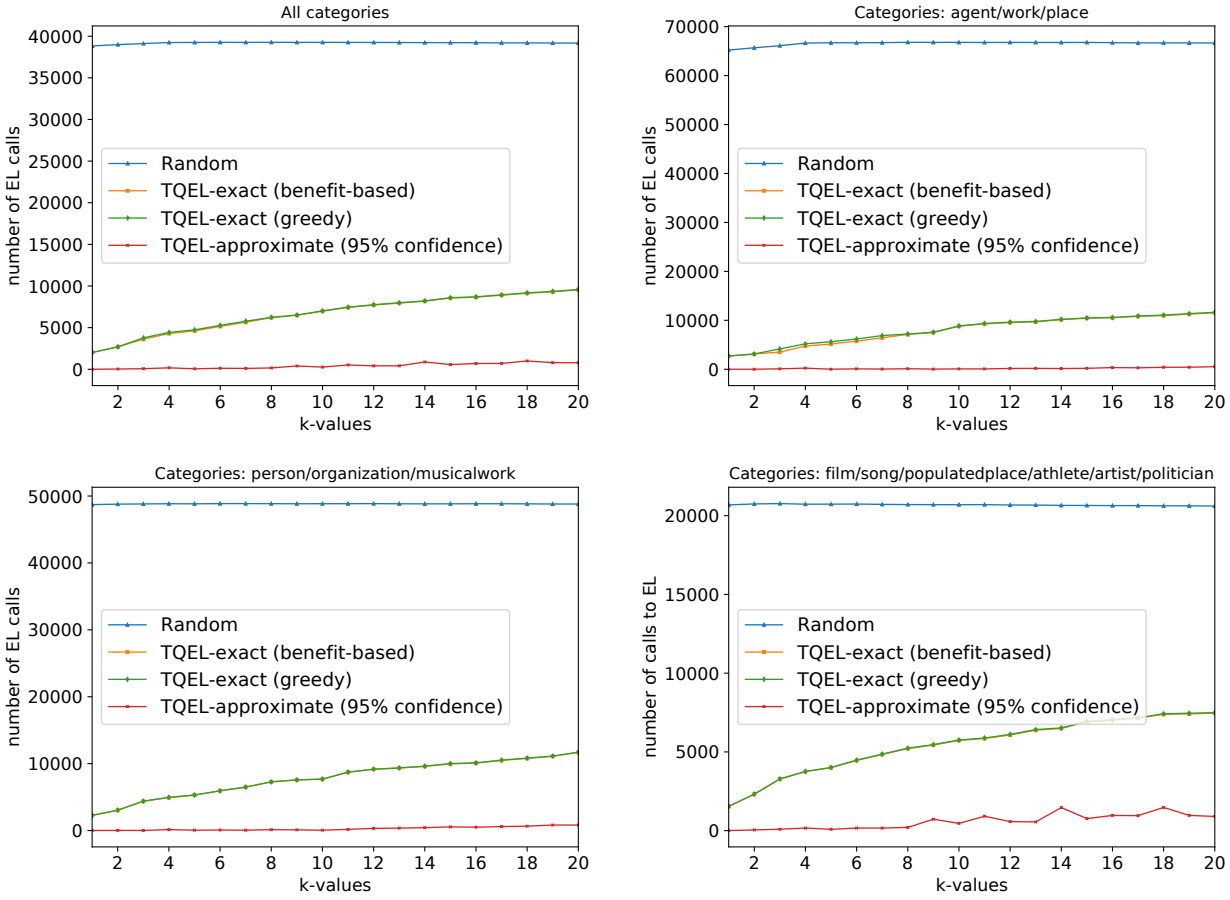

Figure 3.5: Comparing number of calls to entity linking function vs different k-values for multiple categories

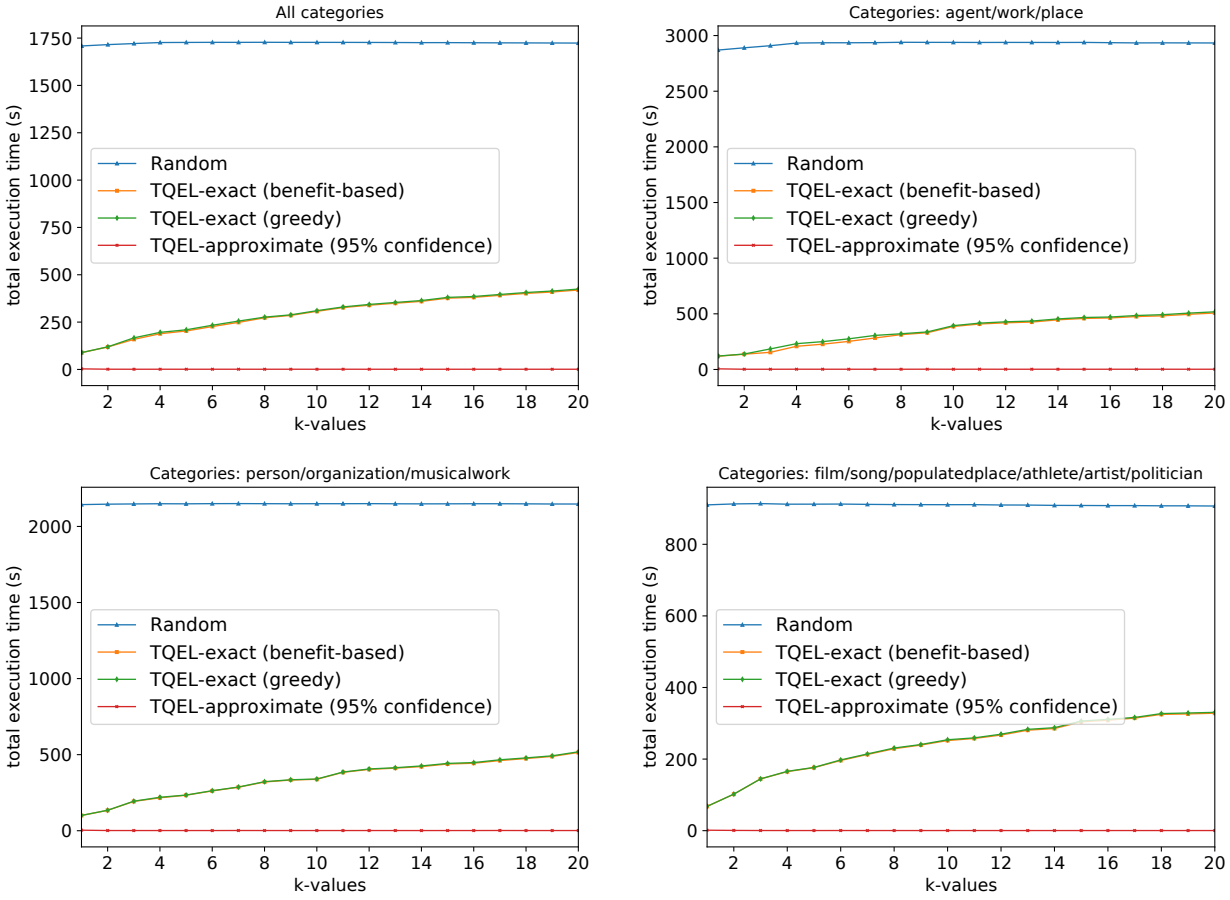

Figure 3.6: Comparing total execution time (seconds) vs different k-values for multiple categories

results based on the category used over the hierarchy levels. Total execution time for TQEL-exact consists of thinking time, benefit function calculation, and the time needed to resolve the chosen mentions. Moreover, for TQEL-approximate total execution time is calculated by adding critical factor choosing time, mention selection time, time to run MC simulation and time to validate the top-k result using the MC runs.

From figures 3.5 & 3.6, we can see that TQEL-approximate is showing promising results in terms of total of number of calls to the entity linking function and total query execution time. We noticed that in all the queries we are saving in term of calls to the EL function, which is a bottleneck for the query, due to the use of approximate query answering techniques. This is due to the fact that the variance in the number of occurrences of each entity in the top-k is quite large, and to find a top-k answer, we only require a much smaller number of entity linking calls.

We also see another trend in TQEL-approximate approach where the number of EL calls for $k = x$ is higher compared to $k = x - 1$ and $k = x + 1$, this happens because of having two or more contenders that have relatively comparable counts which in return forces more calls to EL function for mentions in the competing entities.

We illustrate the fact that TQEL-exact (for both heuristics) outperforms the random approach in general by a huge margin (130x saving) by smartly selecting mentions to disambiguate in an iterative fashion. The reason behind that is by focusing on proving that the k-highest entities in terms of max values. We also see the number of calls and total execution time is increasing whenever k increases while in TQEL-approximate that is not the case. The reason behind such results is that, if the approximation clearly shows that the top-k result without any competition from the $k + 1$ entities, then little work in terms of entity linking is needed regardless of k. However, for TQEL-exact, we need to pay the overhead of performing such EL calls to make sure that the top-k result is valid.

In the experiment, we do not include the execution time of the entity extraction and lookup function since it is shared by all strategies. Additionally, this process could be executed on the entire dataset during ingestion time since it is cheap. We also do not include the numbers of the naive approach that requires the cleaning of all the tuples before running the query although it is clear that our proposed strategies outperform such strategy. The naive approach takes around 145,023 EL calls and around 1.8 hours needed to execute any of the queries. We do not report the naive approach numbers as it is constant over all queries and stretches the figures making them less informative.

Experiment 2: Detailed analysis of TQEL-approximate performance

In this experiment we analyze the performance of the TQEL-approximate for different confidence levels that are given by the query. We will focus on 3 factors that are of interest: number of EL calls, number of mentions that have been sampled using MC technique and total execution time of the query. We are reporting also averaging the query results for categories in the same hierarchy level in this experiment.

In the left upper sub-figure of figure 3.7, we can see that the confidence level heavily affects the number of entity linking calls that we will end up being executed. The number of calls to EL function is growing as the confidence level rises. The number of calls to EL is also affected by the k value as it might cause a significant increase in EL calls due to the competitiveness of entities for that rank based on the number of occurrences .

We also report the difference in number of mentions required for the MC simulation in order to generate a top-k answer set. The right upper sub-figure in figure 3.7 illustrates the amount of savings that are achieved by smartly limiting the generation of MC simulations to mentions associated with competing entities as discussed in section 3.4. The number of sampled mentions also decreases when the number of resolved mentions increases as we will not need to generate samples for the linked mentions. In the lower sub-figure of figure 3.7,

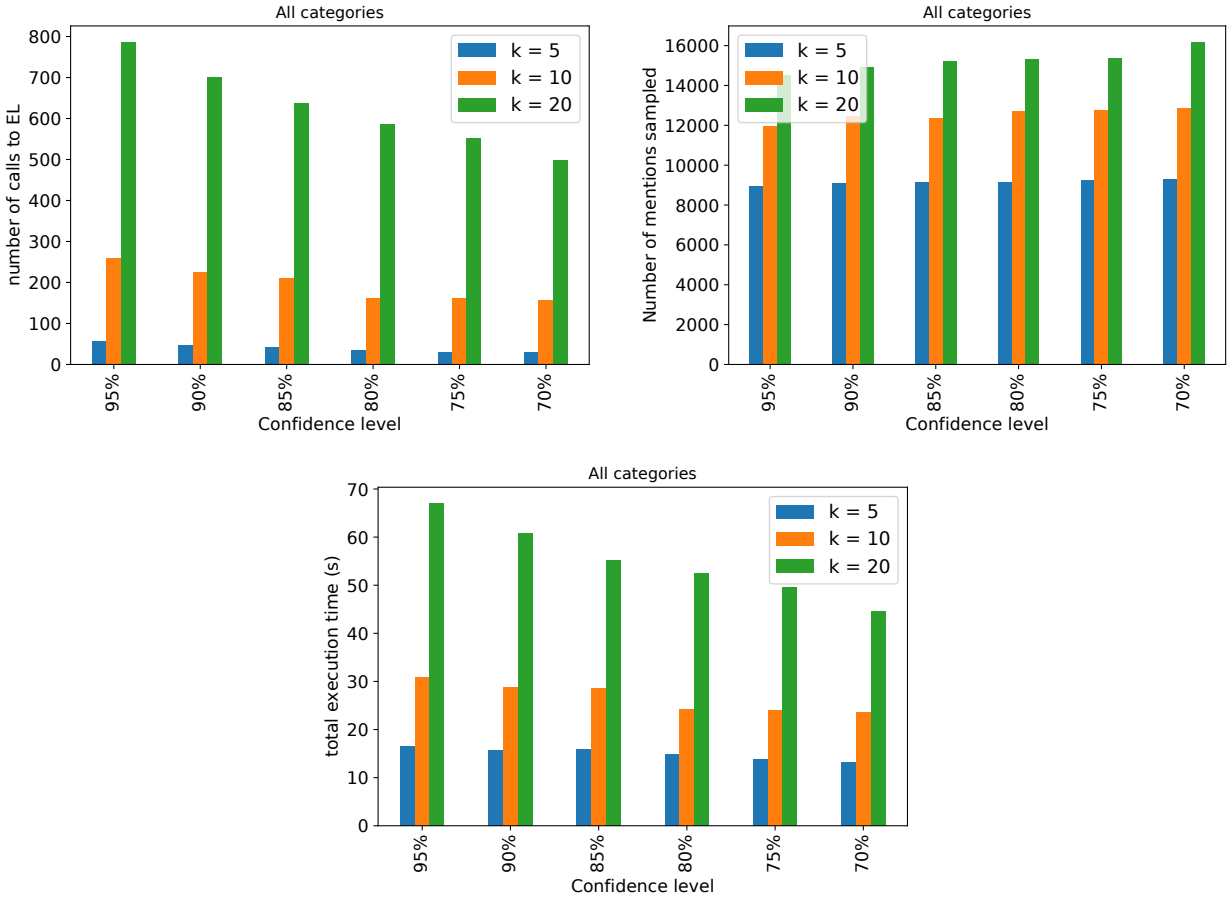

Figure 3.7: Detailed performance analysis of TQEL-approximate using different confidence levels.

we report the total execution time of the queries which consists of (critical factor choosing time, mention selection time, time to run MC simulation and validate top-k answer set using the MC runs). In this experiment we illustrate the fact that in general choosing a lower τ leads to less number of entity linking calls and therefore less total execution time in general given that the EL function execution is the bottleneck.

Experiment 3: Scalability of TQEL

This experiment illustrates the impact of large datasets on TQEL-exact & TQEL-approximate and that the amount of savings we are able to achieve using the TQEL-approximate is noticeable. We have run the query on the category "film" and reported the performance over multiple k values. The "film" category had roughly 10% selectivity in the large dataset. From figure 3.8, we are able to find the same pattern or trend where the savings by TQEL-exact approach in term of EL calls is more than 100x for the calls needed to fully disambiguate all the mentions in the tweets. TQEL-approximate results are also promising and are quite similar to results that have been achieved on the smaller dataset, even though in the large dataset we are dealing with a large number of tweets. This is due to the fact that the difference of number of occurances between entities that are in the top-k is quite large which is allowing TQEL-approximate to return the result with high confidence without much entity linking calls. This experiment clearly shows that in real-world datasets differences in the number of occurrences should be exploited by returning an answer with high confidence using approximation techniques rather than paying the huge overhead of reporting the exact answer.

Experiment 4: Score of TQEL-approximate results

In this experiment we measure the accuracy for the results returned by TQEL-approximate for different τ values. To evaluate the returned answer set we have used two metrics:

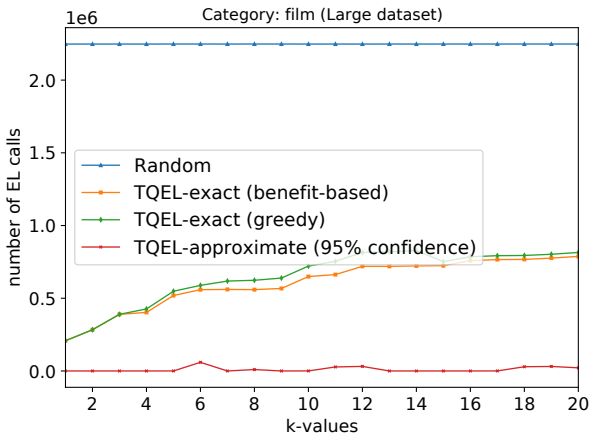

Figure 3.8: EL calls vs k-values

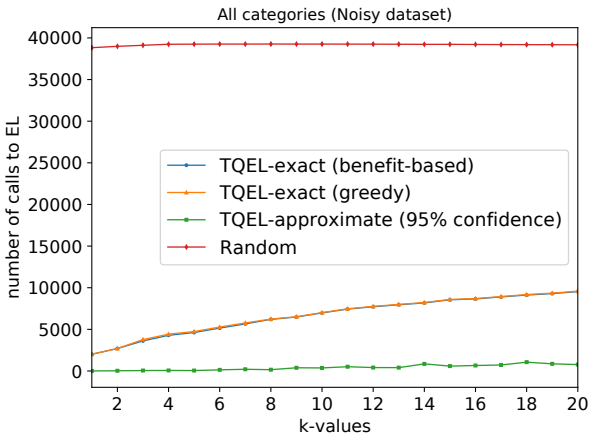

Figure 3.9: EL calls vs k-values

- **Precision:** Represents the fraction of elements in the approximate answer set compared to the exact top-k set.
- **Rank Distance:** A modified version of the footrule distance [34] to compute the distance of the inaccurate entity e_i in the approximate answer set to their exact rank.

To compute rank distance we compute the following: $\frac{1}{k}\sum_{i=1}^{k} maxof(exact_{e_i} - k, 0)$ where $exact_{e_i}$ is the exact rank of e_i. We modified the distance calculation since we only report the top-k answer set without any order between the top-k answer set.

In Table 3.2, we averaged the scores of each confidence levels over all categories and over k values (1 - 20). We see that rank distance is heavily influenced by the chosen τ and the difference in EL calls.

Confidence	95%	90%	85%	80%	75%	70%
Precision	0.95	0.93	0.92	0.92	0.91	0.90
Rank Distance	0.069	0.091	0.113	0.130	0.160	0.168

Table 3.2: Evaluation metrics for different confidence levels

Confidence	95%
Precision	0.90
Rank Distance	0.089

Table 3.3: Evaluation metrics for noisy dataset with 95% confidence level

Experiment 5: Robustness of TQEL

In this experiment we illustrate the effect of introducing noise to the linking probability to study the its impact on the query execution time and answer quality.

From figure 3.9 we have not noticed any major affect due to the introduced noise in terms of number of calls to EL as the TQEL-approximate approach is still dominating any other approach in terms of performance which is also reflected in the total execution time. However, the quality of the answer has dropped due to the introduced noise as shown in Table 3.3.

Experiment 6: Comparing TQEL with other approximate baselines

In this experiment we report the total execution time of the query for different proposed baselines. In figure 3.10, we illustrate the execution time of the EL functions, denoted as [baseline] EL, in a different color to show how different baselines perform. NOFILTER approach was as expected the best in terms of EL function time since we validate the query after each call resulting in not executing any unnecessary EL calls. However, in terms of overall execution of the query it performed the worst due to the extremely high number of expensive verification overhead (factor of the number of entity linking calls). We stopped the query execution after 5 minutes of execution due to the delay and overhead caused by some baselines.

On the other hand, MC-NOOPT approach verification cost was also high compared to TQEL-approximate since MC simulations will be run each time the validation step is performed. This clearly shows the effectiveness of our proposed MC optimizations. We see that

TQEL-approximate performed better than TQEL-approximate (Greedy), especially with the execution time of the EL functions caused by the 9% increase of EL calls compared to TQEL-approximate.

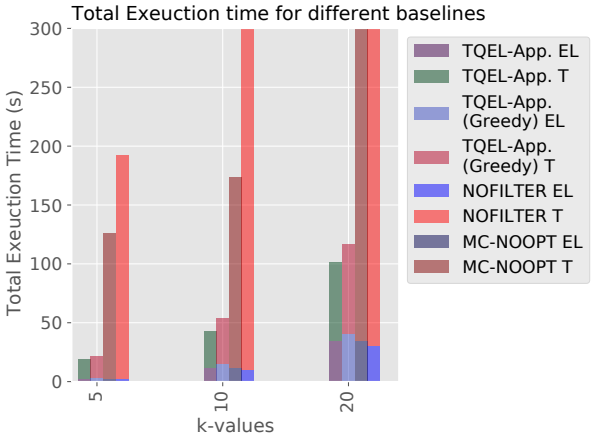

Figure 3.10: Execution Time

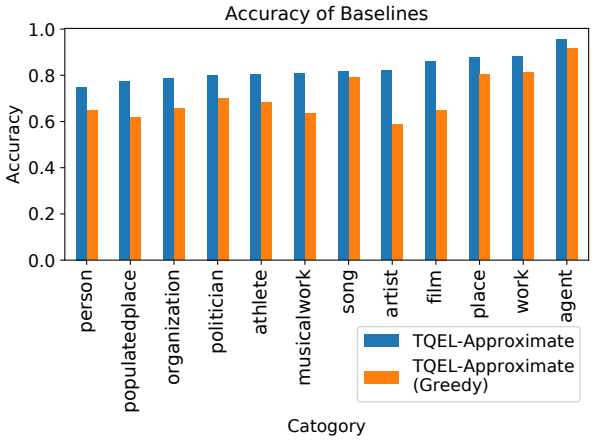

Figure 3.11: Accuracy

Experiment 7: Evaluating Filter's Efficacy.

This experiment shows the accuracy of TQEL-approximate, TQEL-approximate (Greedy) and NOFILTER baselines. In order to calculate the accuracy for the first two, we follow the measurement discussed in Section 3.4.

In the case of NOFILTER, since it never characterizes data as negative (it always invokes the expensive MC simulation), the values for TN and FN are 0 and the value of TP is 1, while the value of FP are number of EL calls - 1. As a result, its accuracy is very poor (below 1% for all categories) we do not show the results in Figure 3.11. Figure 3.11 illustrates the accuracy of TQEL-approximate & TQEL-approximate (Greedy) with a reported average of 83% and 64% accuracy, respectively, over all categories.

In the experiment we further analyzed and realized a correlation between the distribution of the top-k entity lists mentions' probabilities and the accuracy of the result illustrated in table 3.4.

Criteria for comparison of top-20 entity lists	Agent	Athelete
Average size of entity list	149.75	47.65
Average standard deviation of entity lists	4.68	1.7
Average percentage of the difference between mean calculated using normal approximation and mean calculated using MC	0.1%	4.1%
Average number of verifications	1.05	2.5
Average percentage increase of EL calls compared to NO-FILTER approach	5%	16%

Table 3.4: Evaluation metrics for different confidence levels

For the `agent` category where the size of the top-20 entity lists is relatively large, we get an accuracy of 95%. For agent, the average size of the top-20 lists was 149.75, the average σ based on normal approximation was 4.68 and the average μ difference between normal approximation and MC simulation was 0.1% (thus normal approximation is a better estimate), we get an accuracy of 95%. On the other hand, the same numbers for the `athlete` category were 47.56, 1.7 and 4.1% which effects the normal distribution statistics to be inaccurate resulting in a lower filter accuracy.

3.6 Conclusion

In this chapter, we presented TQEL, a framework to support queries that retrieve top-k entities belonging to a user-specified category in a collection $\mathcal{T}$. TQEL exploits the query semantics to reduce the number of entity linking function calls for mentions in $\mathcal{T}$. It provides the option of answering the query exactly with deterministic guarantees or approximately with probabilistic guarantees. The chapter focuses on the approximate approach that uses an implementation of Monte-Carlo simulation technique to efficiently evaluate the query with guarantees. To reduce the overhead of the Monte-Carlo simulations (which is expensive), TQEL-approximate uses normal approximation to estimate the count of each entity as a blocking function / filter. In order to reduce the number of calls to the entity linking

function, a benefit-based function is used to select mentions to link in each iteration.

Chapter 4

TQELX: Query-Driven Cleaning for Group-Based Aggregation Queries

Aggregate queries are a powerful tool for analyzing large sums of data to produce a single representative value. Aggregate queries are widely used in data analysis, decision-making and OLAP domains to offer a summarized version of a field or multiple fields. The presence of uncertainty in datasets hinders the benefits from performing data analytics tasks, including aggregate functions, where the query's result might be misleading and does not answer the query accurately.

For example, given the readings of table 2.1, an insurance company would like to quote a policy for a driver priced based on the average speed of the speed readings, with high confidence, of each driver. The readings were captured by a speed sensor device that produces uncertain readings. If the average speed for a car is above a specific threshold, then the price would be higher. A query corresponding to such an analysis can be as follows:

```
SELECT      License_plate
FROM        Speed_readings
```

```
GROUP BY    License_plate
HAVING      AVG(speed) > 100;
```

The readings in the dataset are not deterministic and an existence probability value accompanies each reading. Consider, for instance, that the video recording is extracted and analyzed to get a more accurate version of that reading (i.e., cleaning). One possible approach to solving this issue is to clean the entire dataset (i.e., consulting the video version for each reading instance). Another option is to apply uncertain query answering techniques to answer the query probabilistically.

Nonetheless, both approaches pose limitations when answering the query at hand. The first approach to cleaning the dataset would be costly and sometimes unnecessary (e.g., for readings that would not affect the predicate satisfaction outcome given if it were to be cleaned). On the other hand, we might not get a high confidence answer by purely using probabilistic query processing approaches. (e.g., the probability that a driver owning a car with license plate ABC satisfying the query is 0.4!). Therefore, a method that utilizes both approaches by cleaning only necessary readings needed to answer the query with high confidence would be more suitable for such queries.

In the previous chapter, we presented TQEL, a framework that finds the top-k entities in a collection of tweets by efficiently calling the entity linking function. The main idea was to exploit a cheap filter-like mechanism, based on normal approximation calculations, that identifies if a top-k answer is found before running the more expensive Monte-Carlo simulation technique for query evaluation. Let us consider the data in TQEL to be modeled using two tables 4.1 & 4.2, we can then view the TQEL top-k query to be a group-by COUNT query over the tables. In particular, the query to retrieve top-2 mentions for the category "films" can be viewed as the following group-by COUNT query:

Tweets				
Tweet_id	Tweeter	Text	Time	Location
t_1	u_1	Black panther has finally grossed $700 million domestically!	ts_1	l_1
t_2	u_2	Emmy Rossum in Beautiful Creatures is stunning	ts_2	l_2
...				

Table 4.1: Relation to represent tweets in table 3.1

Mentions						
Tuple_id	Tweet_id	Mention_id	Mention	Entity	Categories	P
tup_1	t_1	m_1	Black panther	Black Panther (2018 film)	{Film, Creative Work, Oscar Winner}	0.6
tup_2	t_1	m_1	Black panther	Black Panther (1977 film)	{Film, Creative Work}	0.1
tup_3	t_1	m_1	Black panther	Black Panther (Animal)	{Big Cat, Animal}	0.1
tup_4	t_2	m_1	Emmy Rossum	Emmy Rossum	{Entertainer, Agent}	1
tup_5	t_2	m_2	Beautiful Creatures	Beautiful Creatures (2013 film)	{Film, Creative Work}	0.8
tup_6	t_2	m_2	Beautiful Creatures	Beautiful Creatures (Novel)	{Novel, Creative Work}	0.15
...						

Table 4.2: Relation to represent mentions after running entity extraction & entity lookup function on tweets in table 4.1

```
SELECT      Entity
FROM        Mentions
WHERE       "Film" in Categories
GROUP BY    Entity
ORDER BY    COUNT(Entity) DESC
LIMIT       2;
```

In this chapter, we expand the techniques developed in the previous chapter to support a larger class of aggregations, particularly SUM and AVERAGE, in addition to the COUNT supported by the TQEL framework. We refer to the extended framework as TQELX. We first consider a top-k query with different aggregation functions, other than COUNT. We then further generalize the framework to include the other query, a group-by query with a having clause. For the top-k queries with generalized aggregation, we note that the method developed in the previous chapter using the normal distribution filter will need to be modified. Furthermore, as we will see, we can further optimize the Monte-Carlo simulation technique that reduces the cost of the query validation step.

For the group-by query with a having clause, we first formally define probabilistic semantics of such queries and formulate the precise query satisfaction requirement for the cleaning process. We devise a different approach for tackling the problem of integration of the cleaning process since the mechanism to integrate cleaning in top-k queries does not generalize to the queries with the having clause. We first use the normal approximation filter to estimate the aggregation value given the existence probabilities of each tuple. We then exclude groups with a high chance of satisfying the query's condition (i.e., the having clause) from the cleaning process. Furthermore, we eliminate the groups with a high probability of not satisfying the query's condition. Using such bounds, we can limit the cleaning process for each iteration to groups that are of interest and have a chance, if chosen for cleaning, to improve the answer's quality.

We empirically study the expanded framework, TQELX, on a synthetic dataset to evaluate each query's performance, the Monte-Carlo simulation's effects, and the impact of the normal approximation filter on each query.

In summary the main **contributions** of this chapter are:

- We develop a expanded framework TQELX, that integrates the cleaning process with

a different group-based aggregation queries. (section 4.1 & 4.2).

- We propose a probabilistic approach that relaxes the quality of the result by introducing a user-defined confidence threshold to expedite the overall running time of the query efficiently 4.3). (TQELX-probabilistic)

- We experimentally evaluate our framework using a synthetically generated dataset from the TPC-H dataset. We compare the results of the TQELX-probabilistic against the results of a deterministic approach. Moreover, we provide a detailed evaluation for the probabilistic approach (section 4.4).

4.1 Preliminaries

In this section, we will present the required preliminaries that form the basis for the TQELX framework.

4.1.1 Dataset and Required Functions

Dataset

Let $\mathcal{D}$ be a probabilistic database that consists of n probabilistic relations (x-relations) $\mathcal{R}_1, \mathcal{R}_2, ..., \mathcal{R}_n$. Each x-relation $\mathcal{R}_i$ consists of m x-tuples $x_1, x_2, ..., x_m$ and they are considered independent (i.e. the existence of an x-tuple in a x-relation instance is not dependent on the existence of another x-tuple). Each x-tuple x_j is composed of alternative tuples $t_{j,1}, t_{j,2}, ..., t_{j,|x_j|}$ which are normal tuples that follow the schema of the relation $\mathcal{R}_i$ but have an existence probability value associated with each one of them. The alternative tuples $t_{j,1}, t_{j,2}, ..., t_{j,|x_j|}$ of x-tuple x_j are mutually exclusive (i.e. at most one tuple can be present in any x-relation instance). Moreover, $\sum_{y=1}^{|x_j|} \mathrm{p}(t_{j,y}) \leq 1$ for every x-tuple. Our data model follows

the same probabilistic data model that was first introduced in Trio [88, 7].

Cleaning Function

A cleaning function in the TQELX framework $CLN(t_{j,1}, t_{j,2}, ..., t_{j,|x_j|})$ takes as input all alternative tuples of one x-tuple x_j. The output of cleaning function is a either a deterministic tuple that belongs to the x-tuple x_j (i.e. the existence probability of the returned tuple is 1) or no tuples indicating that x-tuple x_j has an existence probability of 0 for any x-relation instance.

The main goal for the cleaning function CLN is to remove the uncertainty of the x-tuple x_j, which in return reduces the uncertainty of the database $\mathcal{D}$. Entity linking is an example of the cleaning function where the combination of (Tweet_id and mention_id) in table 4.2 is the x-tuple while alternative tuples correspond to Tuple_ids that share the same combination of (Tweet_i and mention_id). Another example of the cleaning function is the process of identifying the accurate speed reading from table 2.1.

Query Model

Let $\mathcal{G}(\mathcal{Q})$ be the fields of the group-by clause such that the values of $\mathcal{G}(\mathcal{Q})$ are unique within the alternative tuples of the same x-tuple. Let $\mathcal{AF}(\mathcal{Q}))$ be the field used for aggregation and $agg(\mathcal{Q})$ be the aggregate function. Let $pred(\mathcal{Q})$ be the query's predicate, $having(\mathcal{G}(\mathcal{Q}))$ be the having clause and $order(\mathcal{G}(\mathcal{Q}))$ be the order-by clause of the query $\mathcal{Q}$. In the TQELX framework, we study the problem of integrating the cleaning process with two aggregation group queries, and they are as follows:

- **Top-k aggregation query:** a top-k query $\mathcal{Q}^k$ is evaluated on top of x-relation $\mathcal{R}_i$. Each query $\mathcal{Q}^k$ should include $order(\mathcal{G}(\mathcal{Q}^k))$, however, $pred(\mathcal{Q}^k)$ and $having(\mathcal{G}(\mathcal{Q}^k))$ are optional. For example, a top-k $\mathcal{Q}^k$ that finds the license plate with the highest sum

of recorded speeds from table 2.1. The query can be expressed as the following query:

```
SELECT      License_plate
FROM        Speed_readings
GROUP BY    License_plate
ORDER BY    SUM(speed)
LIMIT       1;
```

- **Group-based aggregation query with having clause:** a query $\mathcal{Q}$ is evaluated on top of x-relation $\mathcal{R}_i$. Each query $\mathcal{Q}$ has a having clause $having(\mathcal{G}(\mathcal{Q}))$ and an optional predicate $pred(\mathcal{Q})$. For example, a query $\mathcal{Q}$ retrieves the license plates that have more than one different speed readings from table 2.1. The query can be expressed as the following query:

```
SELECT      License_plate
FROM        Speed_readings
GROUP BY    License_plate
HAVING      COUNT(License_plate) > 1;
```

In TQELX, we only consider the set of aggregation functions {AVERAGE, SUM, COUNT}.

4.1.2 Probabilistic Query Definitions

Possible Worlds

A possible world w_a of x-relation $\mathcal{R}_i$ from the set of all possible worlds $\mathcal{W}$ is composed of at most one alternative tuple $t_{j,y}$ from each x-tuple x_j. The probability of the possible world p(w_a) can be computed using equation 4.1. Moreover, the summation of the probability of

all possible worlds $\mathcal{W}$ is defined in equation 4.2.

$$p(w_a) = \prod_{j=1}^{m} \mathcal{V} \qquad \text{s.t. } \mathcal{V} = \begin{cases} p(t_{j,y}) & \text{when } w_a \cup x_j = t_{j,y} \\ 1 - \sum_{y=1}^{|x_j|} p(t_{j,y}) & \text{when } w_a \cup x_j = \phi \end{cases} \tag{4.1}$$

$$p(\mathcal{W}) = \sum_{a=1}^{|\mathcal{W}|} p(w_a) \tag{4.2}$$

Probabilistic Group-Based Aggregation Query Evaluation

We now discuss how to evaluate the queries supported in TQELX. Given a possible world $w_a \in \mathcal{W}$, we evaluate the query on the world w_a. Let $ans(w_a)$ be the answer set for $\mathcal{Q}(w_a)$, then we define $satp(\mathcal{G}(\mathcal{Q}))$ as the probability that $\mathcal{G}(\mathcal{Q})$ is $\in ans(w_a)$ sets of $\mathcal{W}$, and can be calculated using the following equation:

$$satp(\mathcal{G}(\mathcal{Q})) = \sum_{a=1}^{|\mathcal{W}|} p(w_a) \cdot I(\mathcal{G}(\mathcal{Q}), w_a) \text{ s.t. } I(\mathcal{G}(\mathcal{Q}), w_a) = \begin{cases} 1 & \text{when } \mathcal{G}(\mathcal{Q}) \in ans(w_a) \\ 0 & \text{when } \mathcal{G}(\mathcal{Q}) \notin ans(w_a) \end{cases} \tag{4.3}$$

Answer Semantics

Let $A(\mathcal{Q})$ be the answer after evaluating query $\mathcal{Q}$ over x-relation $\mathcal{R}_i$ and τ is a user-defined confidence score threshold value. We now define the answer semantics for both queries as follows: We call $A(\mathcal{Q})$ a valid answer for the top-k query iff:

- $\forall\ \mathcal{G}(\mathcal{Q}) \in A(\mathcal{Q}), \quad satp(\mathcal{G}(\mathcal{Q})) > \tau.$

The top-k aggregation query evaluation semantic we adopt in TQELX is similar to the semantics of PT-k [49] where only group values that have a top-k probability, *satp*, higher

than a specific threshold τ will only be included in the answer set. The order among the returned top-k answer set is ignored in our setting. Note that in the case of ties (more than k group values satisfy the answer semantics), we return the only k group values ordered by their $satp(\mathcal{G}(\mathcal{Q}^k))$ score in a descending order.

Problem Definition

Given a group-based aggregation query $\mathcal{Q}$ on top of x-relation $\mathcal{R}_i$, a cleaning function CLN, a user-defined confidence score threshold value τ, TQELX provides an efficient probabilistic solution for the query $\mathcal{Q}$ that generates *A($\mathcal{Q}$)* such that *A($\mathcal{Q}$)* follows the predefined answer semantics for the probabilistic approach by reducing the number of x-tuples cleaned using the cleaning function CLN.

For the group-based aggregation query with having clause, a user specifies a cut-off confidence score threshold value β such that we discard any tuples for group $\mathcal{G}(\mathcal{Q})$ with $satp(\mathcal{G}(\mathcal{Q})) < \beta$ from the cleaning process.

4.1.3 Probabilistic Group-Based Aggregation Query Example Solution

SpeedReadings				
xid	Reading_id	License_plate	Speed	P
x_1	r_1	ABC	100	0.6
x_1	r_2	XYZ	80	0.3
x_1	r_3	MNO	30	0.1
x_2	r_5	XYZ	70	1
x_3	r_6	XYZ	90	1
x_4	r_7	ABC	110	0.4

Table 4.3: Partially cleaned speed readings table

Possible_world	Probability
r_1, r_5, r_6, r_7	0.24
r_1, r_5, r_6	0.36
r_2, r_5, r_6, r_7	0.12
r_2, r_5, r_6	0.18
r_3, r_5, r_6, r_7	0.04
r_3, r_5, r_6	0.06

Table 4.4: Partially cleaned possible worlds results.

SpeedReadings				
xid	Reading_id	License_plate	Speed	P
x_1	r_1	ABC	100	1
x_2	r_5	XYZ	70	1
x_3	r_6	XYZ	90	1

Table 4.5: Fully cleaned speed readings table

Let the query be the one presented as an example in the query model for the top-k aggregation query that finds the license plate with the highest sum of speeds that was recorded in table 2.1. Let the user-defined confidence score threshold value τ be 0.75. We first run the query in each possible world presented in table 2.2. After calculating *satp* for each license plate we get the following set (license_plate: *satp*): {ABC: 0.24, MNO: 0.138, XYZ: 0.622}. We see that we cannot return an to the top-k query unless we clean more tuples to return an answer to the query that satisfies the answer semantics. When we choose to clean alternative tuples of x-tuple x_2 we get the partially cleaned table 4.3. After evaluating the query in each possible world instance depicted in table 4.4 we get the following set (license_plate, *satp*): {ABC: 0.24, XYZ: 0.76}. We can now return {XYZ} as the $\mathcal{Q}^k$'s answer since it follows the answer semantics.

We next discuss the solution of the group-based aggregation with having clause query example that asks for the license plates with more than one recorded speed readings in table 2.1. Given that the confidence threshold confidence value τ is 0.85, the cut-off confidence threshold value β is 0.25 and after evaluating the group-based aggregation query and calculating *satp* for each group, we get the following set (License_plate: *satp*): {ABC: 0.24, MNO: 0.03, XYZ: 0.79}. Since there are no groups with $satp > \tau$, we cannot include any group in the answer set. However, the group with the value XYZ has *satp* 0.79 and can be a potential answer if some tuples are cleaned. From the partially cleaned table 4.3 and possible worlds illustrated in table 4.4, we get the following *satp* values after evaluating the query: {ABC: 0.24, MNO: 0.03, XYZ: 1}. Hence, we show that we have found a set {XYZ} that satisfies

the answer semantics and can be returned as the answer for the query $\mathcal{Q}$. Note that group values {ABC, MNO} will be ignored as potential answers and discarded from the cleaning candidate set since their $satp < \beta$.

Having formally defined the semantics of the group-by queries with aggregation and having queries, the following section discusses the overview of the TQELX implementation.

4.2 TQELX overview

The abstract flow of any query-driven cleaning model is illustrated in figure 1.1 as the uncertainty cleaning cycle where the cleaning step is iteratively executed until the quality of the answer satisfies the query's semantics. In TQELX, we generalize the uncertainty cycle to support probabilistic group-based aggregation queries where the query evaluation follows the approximate confidence computation approach. TQELX is a middleware framework that handles the probabilistic query evaluation in memory and returns the answer to the user. The general algorithm shown in algorithm 3 is an adaptive algorithm that analyzes the current state of the execution and adjusts the query plan accordingly. The algorithm consists of a preparatory phase where groups are created given the tuples that satisfy the query's predicate, a cleaning phase which identifies the set of tuples to be cleaned for the current iteration, and an evaluation phase that evaluates the query and return an answer if the stopping condition is met. The algorithm, in essence, is alike for both queries supported in TQELX. However, the aggregate functions' implementation differs (e.g., filter testing differs for both queries).

Algorithm 3 TQELX Approach

procedure GETQUERYANSWER($\mathcal{Q}$, $\mathcal{R}_i$)
    $satisfying\_tuples \leftarrow$ getSatisfyingTuples($\mathcal{Q}$, $\mathcal{R}_i$)
    $groups \leftarrow$ createGroupsFromTuples($satisfying\_tuples$)
    $answer\_found \leftarrow$ false
    while !$answer\_found$ **do**
        while !passFilterTest($\mathcal{Q}$, $groups$) **do**
            $t_{j,1}$, $t_{j,2}$,...,$t_{j,y} \leftarrow$ selectTuplesToClean($satisfying_tuples$, $groups$)
            $x_j \leftarrow$ CLN($t_{j,1}$, $t_{j,2}$,...,$t_{j,y}$)
            updateRelation(x_j, $\mathcal{R}_i$)
            updateGroups(x_j, $groups$)
        if stoppingConditionMet($\mathcal{Q}$) **then**
            $answer\_found \leftarrow$ true
    returnAnswer($\mathcal{Q}$, $groups$)

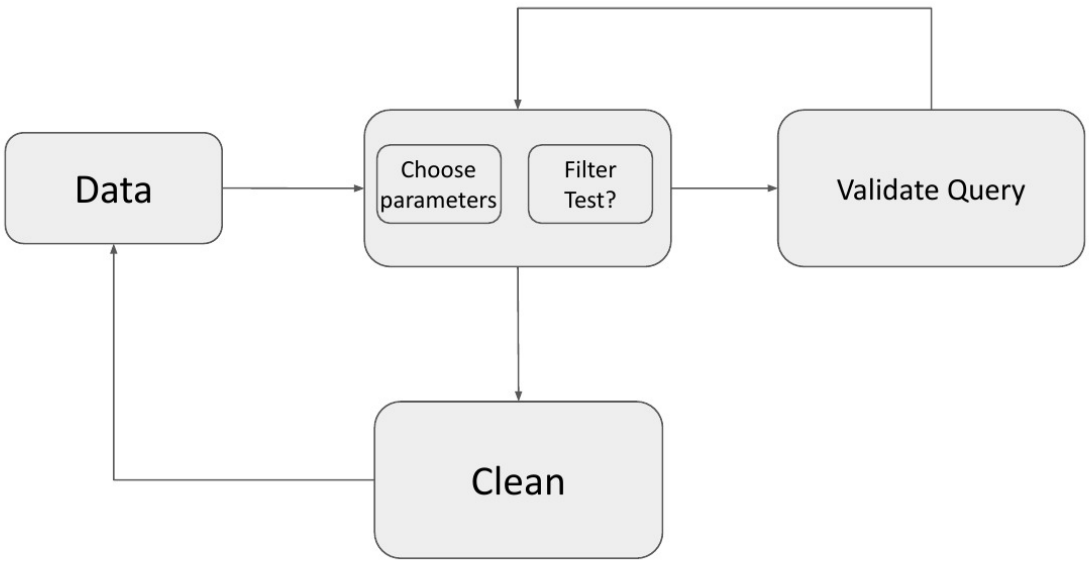

Figure 4.1: TQELX flow diagram

4.2.1 Preparatory Phase

To prepare tuples and groups for the next phases, algorithm 3 retrieves all the tuples that satisfy $pred(\mathcal{Q})$ from x-relation $\mathcal{R}_i$. Then a list of group objects is created such that each group object represents a possible value of $\mathcal{G}(\mathcal{Q})$ given the retrieved predicate satisfying tuples. Moreover, every group object holds a list named "dirty tuples list," which consists of pointers to all the uncleaned tuples (i.e., the probability of tuple < 1) that have the same group values. Tuples in the dirty tuples list are ordered based on their aggregation value in a descending order. Moreover, each group has a list named "clean tuples list," which stores the pointers to the cleaned tuples (i.e., the probability of tuple $= 1$) with the same group values. In addition, each group object has min & max counters. These correspond to the possible minimum and maximum values that an aggregate associated with the group might have after all tuples are cleaned.

In order to calculate the max counter for a group $\mathcal{G}_c$ when the aggregation function is COUNT or SUM, we execute the aggregation function on dirty tuples list + clean tuples list of the group $\mathcal{G}_c$ (e.g., for the aggregation function SUM, we calculate the summation of dirty tuples lists and add it to the summation of clean tuples list). However, for aggregate function AVERAGE, the calculation is more complicated. We need to consider the min average and the tuples that would increase the average score only. We show in algorithm 4 how to accurately calculate the max counter. On the other hand, when calculating COUNT and SUM for the min counter, we only apply the aggregate function to the clean tuples list and report the result as the min counter. For the AVERAGE aggregation function, we describe in algorithm 5 how to calculate it since calculating min value is not as straightforward by only increasing the aggregate values of dirty tuples that decreases the overall average.

Given the satisfying tuples, an object is created for each tuple that holds the values and a pointer to the x-tuple object it belongs to. We also maintain a list of x-tuple objects

Algorithm 4 Calculating max counter for AVERAGE aggregate function

```
1: procedure CalculateMaxCounterForAVG(G_c, AF(Q))
2:     max ← 0
3:     for each t ∈ G_c.CleanTuples do
4:         max ← t.AF(Q)
5:     count ← G_c.CleanTuples.size()
6:     max ← max / count
7:     groups ← createGroupsFromTuples(satisfying_tuples)
8:     stop_calculation ← false
9:     index ← 0
10:    while !stop_calculation & index < G_c.DirtyTuples do
11:        if max ¡ G_c.DirtyTuples[index].AF(Q) then
12:            max ← max * count + G_c.DirtyTuples[index].AF(Q)
13:            count ← count + 1
14:            max ← max / count
15:            index ← index + 1
16:        else
17:            stop_calculation ← true
```

Algorithm 5 Calculating min counter for AVERAGE aggregate function

```
1: procedure CalculateMinCounterForAVG(G_c, AF(Q))
2:     min ← 0
3:     for each t ∈ G_c.CleanTuples do
4:         min ← t.AF(Q)
5:     count ← G_c.CleanTuples.size()
6:     min ← min / count
7:     groups ← createGroupsFromTuples(satisfying_tuples)
8:     stop_calculation ← false
9:     index ← G_c.DirtyTuples.size() - 1
10:    while !stop_calculation & index > 0 do
11:        if min > G_c.DirtyTuples[index].AF(Q) then
12:            min ← min * count + G_c.DirtyTuples[index].AF(Q)
13:            count ← count + 1
14:            min ← min / count
15:            index ← index - 1
16:        else
17:            stop_calculation ← true
```

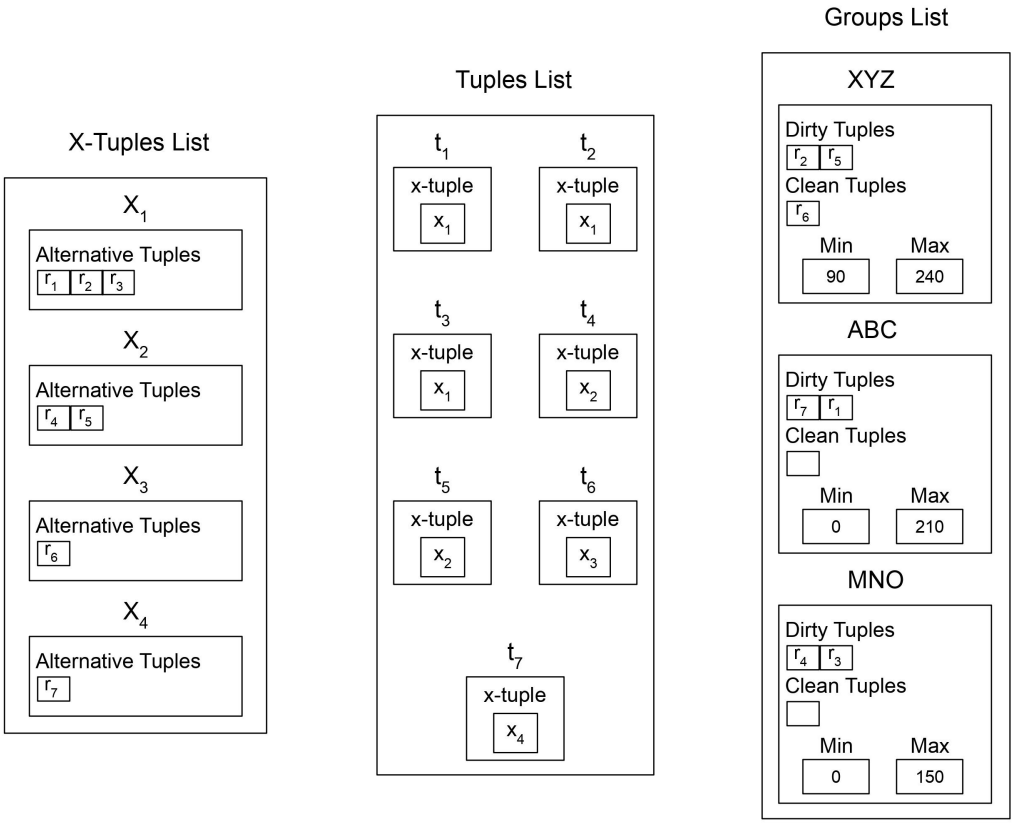

Figure 4.2: TQELX data structures.

where for each x-tuple object, we store a list of alternative tuples that point to the different tuples in the Tuples list. Figure 4.2 illustrates the various data structures that are used in the TQELX framework and how other objects connect. The central concept behind having different pointers in each object is to allow for instant access to any group, x-tuple, or alternative tuple.

4.2.2 Cleaning Phase

Tuples Selection for Cleaning

By exploiting the group-based query semantics, we can smartly select a set of tuples to

clean to satisfy the query evaluation steps' stopping condition efficiently. Cleaning the most relevant tuples yields savings in terms of the cost of cleaning and, hence, reduces the total execution time of the query. The method of choosing the set of tuples to clean is different when answering different queries supported in TQELX and will be explained in detail in the next section for each query.

Cleaning Function

When a set of alternative tuples, that belong to the same x-tuple x_j, are fed to the cleaning function CLN, the expectation is at most one tuple is returned as the clean version of x_j (i.e. the x-tuple value x_j will be unique within the relation). For example, if the framework decides to clean the alternative tuples of x-tuple x_1 in table 4.3, tuples $\{r_1, r_2, r_3\}$ are sent as input to the cleaning function $CLN(r_1, r_2, r_3)$. The output of the cleaning function, as depicted in the fully cleaned table 4.5, is the tuple r_1 and we can see in table 4.5 we only see one tuple representing the x-tuple x_1. The cleaning function process will be the same for any type of query used in TQELX framework.

Groups Update

In this step, the algorithm updates the different groups that were affected by the cleaning function executed on the selected tuples. For each tuple chosen, we remove its pointer from the dirty tuples lists and subtract the aggregation value of the tuple from the max counter. If the output returns a tuple as the cleaned version of the associated x-tuple, we add a pointer in the clean tuples list for the group that shares the same group values with the returned tuple. Moreover, the aggregate value of the clean version tuple is added to the min & max counters of that group.

For example, when choosing the alternative tuples $\{r_1, r_2, r_3\}$ of x-tuple x_1 for cleaning, r_1 pointer in the dirty tuples list of the group with value "ABC" and the new max counter

value is 110. we perform the same process for tuples r_2 & r_3 and their corresponding groups with values "XYZ", "MNO" respectively. Once r_1 is returned as the clean version of x-tuple x_1, we add it to the clean tuples list of group with value "ABC" along with adding the value 100 to both the min & max counter so they become 100 & 210 respectively.

4.2.3 Evaluation Phase

Filter Test

TQELX exploits a filter-based mechanism where its main job is to estimate if a potential answer is found for the probabilistic query $\mathcal{Q}$. Since the probabilistic evaluation of the stopping condition is expensive, the filter acts as a gatekeeper to prevent proceeding to the probabilistic query evaluation step until the chance of reaching the stopping condition is high. The following section explains the filter's features and how the test is performed for both queries supported in TQELX.

Stopping Condition

In this step, the algorithm checks if a solution for the group-based $\mathcal{Q}$ is found and can be validated. The implementation of the stopping condition function is different for the top-k aggregation query and the group-based aggregation with a clause query. The implementation of both queries will be discussed in detail in the next section.

4.3 TQELX-probabilistic Approach

In this section, we describe our approach to integrating the cleaning process with probabilistic query answering as the defined semantics in section 4.1. We discuss in detail how to implement the different functions of algorithm 3 in the context of both supported queries

in TQELX. We specifically study the implementation of tuples selection, updating groups, filter test and stopping condition functions.

To evaluate the query probabilistically, the generation of all possible worlds $\mathcal{W}$ is required and then evaluating the query on each world $w_a \in \mathcal{W}$ in order to retrieve the confidence score of each answer. However, as we explained in section 3.4 of chapter 3, this is infeasible to execute. We instead follow the approximation confidence computation semantics by sampling the set of all possible worlds $\mathcal{W}$ and then evaluating the query on each sampled world. By computing *satp* for each group, we can include groups with a more confidence score than τ.

4.3.1 Stopping Condition

Similar to the TQEL framework, Monte-Carlo simulation is used for the TQELX-probabilistic approach by generating random samples of the set of all possible worlds $\mathcal{W}$. A random world sample w_a is generated by choosing at most one alternative tuple $t_{j,y}$ for each x-tuple x_j based on the probability distribution of the alternative tuples $t_{j,1}, t_{j,2}, ..., t_{j,|x_j|}$. If the summation of the probabilities of alternative tuples $t_{j,1}, t_{j,2}, ..., t_{j,|x_j|}$ does not add up to 1, the x-tuple has a probability of 1 - $\sum_{y=1}^{|x_j|} \mathrm{p}(t_{j,y})$ which corresponds to the probability of the x-tuple x_j not existing in the database. We also generalize the formula to calculating the probability of group $\mathcal{G}_c$ showing up in the final answer $\hat{p_c}$ as follows:

$$\hat{p_c} = \frac{\text{number of times } \mathcal{G}_c \text{ appears in } A(\mathcal{Q})}{N} \tag{4.4}$$

where N is the number of sampled worlds.

TQELX-probabilistic checks the stopping condition by calculating and comparing lcb_c in

equation 3.5[1] against τ for every group $\mathcal{G}_c$. The stopping condition has a different implementation for the top-k aggregation query and group-based aggregation query with having clause. For the top-k aggregation query, if there are k groups whose lcb_c values $> \tau$, then query execution stops, and those k groups are returned as the answer for the top-k query.

For the group-based aggregation query with having clause, the query execution will not stop until all group values are either in the answer set (i.e., $lcb > \tau$) or discarded from the cleaning candidates given the cut-off confidence score threshold value β. We calculate a higher bound for the confidence cut-off bound ($hccb$) using equation 4.5. If $hccb_c < \beta$, we then refrain from cleaning any tuples associated with group $\mathcal{G}_c$ and delete its object from the groups' list. Hence, when the cleaning stops, every group in $\mathcal{G}(\mathcal{Q})$ has to be either in the answer set A($\mathcal{Q}$) or not considered as a potential answer and therefore deleted from the group list.

$$hccb_c = \hat{p_c} + z \times \sqrt{\frac{\hat{p_c} \times (1 - \hat{p_c})}{N}} \tag{4.5}$$

Given the similarity with TQEL-approximate, discussed in section 3.4, we propose similar approaches and optimizations from exploiting filters based on normal approximation calculation, mentions selection technique and other optimizations related to the Monte-Carlo simulation execution. However, a complete adoption is not simply possible. Thus, we need to modify some of the approaches, given that the query semantics are different and require multiple modifications to the previous solution. We discuss the necessary changes in the following subsections.

Filters

The main goal of the filter in TQELX is to provide a mechanism that can estimate the

[1] lcb as explained in chapter 3 is the lower confidence bound of the probability of group $\mathcal{G}_c$

outcome of the approximate query evaluation, using the Monte-Carlo simulation technique, in an efficient and fast way without the need to run the actual expensive simulation and evaluation. Hence, any filter with such properties can be used in our framework to speed up the total query execution time.

In TQELX, we use the normal approximation filter that was introduced in section 3.4 due to a degree of similarities between the two problem settings where both are answering a group-based aggregation query. To generalize the use of such filter, we have to redefine different terms such as the mean (μ), the standard deviation (σ), the approximate max (AP_{max}), the approximate min (AP_{min}) and $ENC(\alpha_x)$ that are used in section 3.4. We, further, revisit the implementation of the filter's test for both supported queries.

Let $\mu_{\mathcal{G}_c}$ be the mean of group $\mathcal{G}_c$, $\sigma_{\mathcal{G}_c}$ be the standard deviation of group $\mathcal{G}_c$, p($t_{j,y}$, $\mathcal{G}_c$) be the probability that $t_{j,y}$ exists in the database and the value of the group fields is $\mathcal{G}_c$ and $value(\mathcal{AF}_{t_{j,y}})$ is the value used by the aggregation field for $t_{j,y}$. In equations 4.6 & 4.7 we define μ_c & σ_c for different aggregate functions. Using those equations we will be able to calculate the approximate max & approximate min (AP_{max_c} & AP_{min_c}) for every group as formulated in equations 3.8 & 3.9 respectively.

$$\mu_c = \begin{cases} COUNT & \sum\limits_{\mathcal{G}(t_{j,y})\in\mathcal{G}_c} p(t_{j,y},\mathcal{G}_c) \\ SUM & \sum\limits_{\mathcal{G}(t_{j,y})\in\mathcal{G}_c} (p(t_{j,y},\mathcal{G}_c)\times value(\mathcal{AF}_{t_{j,y}})) \end{cases} \tag{4.6}$$

$$\sigma_c^2 = \begin{cases} COUNT & \sum\limits_{\mathcal{G}(t_{j,y})\in\mathcal{G}_c} (p(t_{j,y},\mathcal{G}_c)\times(1-p(t_{j,y},\mathcal{G}_c))) \\ SUM & \sum\limits_{\mathcal{G}(t_{j,y})\in\mathcal{G}_c} (p(t_{j,y},\mathcal{G}_c)\times(1-p(t_{j,y},\mathcal{G}_c))\times value(\mathcal{AF}_{t_{j,y}})^2) \end{cases} \tag{4.7}$$

In the case of AVERAGE, calculating the μ_c & σ_c^2 is not that straightforward given that we

cannot quickly know the tuples that will be counted in the average calculation and which will not. We, however, treat the AVERAGE aggregate function as a ratio sample estimator and use the DLTA method [20] to compute an estimate of the μ_c & σ_c^2. Although the estimates will be biased, when the number of tuples within each group is high, the bias effect will be small. Hence we define the μ_c & σ_c^2 of AVERAGE in equations 4.9 & 4.10 respectively. We, first, define the terms that will be used for equations 4.9 & 4.10.

$$\mathcal{M}_X = \frac{\mu_c \, of \, COUNT}{|\mathcal{G}_c|} \; , \; \mathcal{M}_Y = \frac{\mu_c \, of \, SUM}{|\mathcal{G}_c|} \; , \; \mathcal{S}_X = \frac{\sigma_c^2 \, of \, COUNT}{|\mathcal{G}_c|} \; , \; \mathcal{S}_Y = \frac{\sigma_c^2 \, of \, SUM}{|\mathcal{G}_c|}$$
$$\mathcal{S}_{XY} = \frac{1}{|\mathcal{G}_c|} \sum_{\mathcal{G}(t_{j,y}) \in \mathcal{G}_c} (p(t_{j,y}, \mathcal{G}_c) \times (1 - p(t_{j,y}, \mathcal{G}_c)) \times value(\mathcal{AF}_{t_{j,y}})) \tag{4.8}$$

$$\mu_c = \frac{\mathcal{M}_Y}{\mathcal{M}_X} \tag{4.9}$$

$$\sigma_c^2 = \frac{1}{|\mathcal{G}_c|}(\frac{\mathcal{S}_Y}{\mathcal{M}_X^2} - \frac{2 \times \mathcal{M}_Y \times \mathcal{S}_{XY}}{\mathcal{M}_X^3} + \frac{\mathcal{M}_Y^2 \times \mathcal{S}_X}{\mathcal{M}_X^4}) \tag{4.10}$$

For the top-k aggregation query $\mathcal{Q}^k$, we generalize the formula of the estimated number of cleanings (ENC) needed to pass the filter's test. With such generalization, we can apply it to SUM & AVERAGE aggregate functions rather than only COUNT as illustrated in equation 4.11. Note that groups are ordered in a descending order according to their AP_{max} value.

$$ENC(\alpha_x) = \sum_{c=1}^{k} \frac{maxof(AP_{max}(\mathcal{G}_{k+1}, x) - AP_{min}(\mathcal{G}_c, x), \; 0)}{max(\mathcal{G}_c) - \mu_c} \times TTC_{\mathcal{G}_c} \tag{4.11}$$

where $AP_{max}(\mathcal{G}_c, x)$ is the $AP_{max}(\mathcal{G}_c)$ if x is the z-value and $TTC_{\mathcal{G}_c}$ corresponds to the number of uncleaned tuples in $\mathcal{G}_c$.

For the group-based aggregation query with having clause, we calculate ENC for each group by calculating two estimates and choosing the minimum of both. The first estimate, illustrated in equation 4.12, calculates the estimated number of cleanings for $\mathcal{G}_c$ to prove it is in the answer set. On the other hand, equation 4.13 estimates the number of cleanings needed to prove that $\mathcal{G}_c$ is not in the answer set (i.e., to find enough cleanings that eliminate this group from the query's answer).

$$ENC(\alpha_x) = \sum_{c=1}^{|\mathcal{G}(\mathcal{Q})|} \frac{maxof(E - AP_{min}(\mathcal{G}_c, x),\ 0)}{max(\mathcal{G}_c) - \mu_c} \times TTC_{\mathcal{G}_c} \tag{4.12}$$

$$ENC(\alpha_x) = \sum_{c=1}^{|\mathcal{G}(\mathcal{Q})|} \frac{maxof(AP_{max}(\mathcal{G}_c, x) - E,\ 0)}{\mu_c - min(\mathcal{G}_c)} \times TTC_{\mathcal{G}_c} \tag{4.13}$$

where E is the numerical operand in the having clause.

We follow the rest of the normal approximation filter's implementation used in section 3.4 along with the algorithms and equations for choosing the filter's parameters.

4.3.2 Monte-Carlo Implementation

In TQELX, we leverage the previously generated Monte-Carlo runs by keeping the outcome of the runs inside the x-tuple object. each x-tuple x_j contains a vector of size N where for each vector value we store the alternative tuple $t_{j,y}$ that was assigned to x-tuple x_j for that run. If no alternative tuples were assigned, we place a null value for that run instead. We, further, have a vector with N values in each group $\mathcal{G}_c$ object that holds the aggregate

value that group $\mathcal{G}_c$ gets for that specific run. By doing so, we can effectively modify the previous aggregate values for solely the affected groups when an x-tuple is cleaned rather than regenerating the values from scratch for each query validation. We also store the minimum aggregate value of all N worlds for the group to answer the top-k query effectively.

Integration of Query Processing With Monte-Carlo Simulation

TQELX intelligently generates the Monte-Carlo simulation runs by checking the having clause condition within the generation process. For top-k queries, we follow the exact generation mechanism that was introduced in section 3.4.

For group-based queries aggregation queries with having clause, we keep two vectors with size N that hold the min & max values for each run such that min = max after generating all the runs for the dirty tuples for a group $\mathcal{G}_c$. The concept behind keeping min & max values for each run is to stop the generation of the rest of x-tuples if the group for this run will not satisfy the having clause condition. For example, in table 4.3, if the having clause condition for each license plate group to have a sum > 150, then if x_1 in run_1 is not randomly assigned to r_1 we do not need to generate the sample for x_4 since we already know that group "ABC" will not satisfy the having condition for run_1. Moreover, if the having clause checks whether the aggregate value is "greater than" or "greater than or equal" a numerical operand, we first generate the samples for tuples with the highest AF_c value (i.e., start from the first index in the dirty tuples list) and vice versa.

Answer Verification

After generating the sample runs for all the required x-tuples to evaluate the query as described above. We then calculate *satp* for groups that of interest depending on each query. For the top-k we follow the same answer verification in section 3.4.

For group-based aggregation query with having clause, we would need to calculate *satp*, *lcb* & *hccb* for each group as shown in equations 4.3, 3.5 & 4.5 respectively. If the lower confidence bound (lcb) $> \tau$, we add the group to the answer set. However, if the higher confidence cut-off bound ($hccb$) $< \beta$, we discard this group as a potential answer to the query and delete its object. Otherwise, the clean process continues until each group is either in the answer set or discarded from the potential answers (i.e., no groups that need cleaning).

4.3.3 Selecting Tuples for Cleaning

We propose a benefit-based method for identifying the next tuple to clean that would have a high chance of reducing the number of cleanings in future iterations. We use ENC as an indicator for the tuples that, if cleaned, would bring the cleanings process to an early stop. Let $ENC(S)$ be the estimated number of cleaning needed in the current state, $ENC(S, t_{j,y}, x_j)$ be to the estimated number of cleanings needed for the state when x_j is cleaned and the returned tuple is $t_{j,y}$ and let $ENC(S, NULL, x_j)$ is the number of needed cleanings when the cleaning function does not return any tuple for x_j. In each iteration, we choose the x-tuple with the highest benefit score calculated using the following equation:

$$Benefit(x_j) = ENC(S) - (\sum_{y=1}^{|x_j|}(p(t_{j,y}) \times ENC(S, t_{j,y}, x_j)) + (1 - \sum_{y=1}^{|x_j|} p(t_{j,y})) \times ENC(S, NULL, x_j)) \tag{4.14}$$

4.4 Experiments

In this section, we illustrate the wide range of experiments conducted to measure the effectiveness of the TQELX framework.

4.4.1 Experimental Setup

Datasets

Since dirty probabilistic datasets that require cleaning are not widely available to conduct performance experiments, we synthetically modify TPC-H data for experimental purposes. We first generated 1 GB of data from the TPC-H dataset using TPC-H's dbgen program. Given that TPC-H is a deterministic dataset used for benchmarking deterministic databases, we synthetically converted the dataset into a probabilistic dataset that follows the x-tuple model. As such, this resulted in creating $>$ 3GB of data which we use for our experiments.

When generating the probabilistic dataset, we first treated each original tuple as an x-tuple. Then, for each x-tuple, we added 1-4 alternative tuples that had random values different from the original x-tuple. After that, we randomly assigned each alternative tuple an existence score indicating its confidence to exist in the database. We also normalized the existence probability values for all alternative tuples that belong to the same x-tuple such that their $\sum \leq 1$. For instance, consider the following tuple in the "lineitem" relation (3666275, 133613, 8640, 3, 16.00, 26345.76, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular"). We first add an x-tuple id (xid) and a unique tuple id (tid), as the first two fields, so it becomes (1, 1, 3666275, 133613, 8640, 3, 16.00, 26345.76, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular") where the green colored values correspond to the newly added values.

After that we create the alternative tuples by generating random values for each field in the original tuple that is different from the original values. An example of the alternative tuples would be (1, 2, 3666275, 133613, 44, 3, 16.00, 2354.95, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular") & (1, 3, 3666275, 133613, 2567, 3, 16.00, 86674.82, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular") where the blue colored values correspond to randomly generated values. In this example, two alternative tuples are generated: a tuple with tid = 2 and another tuple with tid = 3 plus the tuple with tid = 1 that holds the original values. We, further, generate three random values within the range (0, 1) and normalize these values such that their summation is $\leq$ 1. Moreover, we assign the highest value to the tuple with tid = 1, by adding a probability field to the tuple, and assign the other two values to the other alternative tuples, tuples with tid=2 & tid = 3, randomly. An example of the resulting alternative tuples would be: (1, 1, 0.6, 3666275, 133613, 8640, 3, 16.00, 26345.76, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular"), (1, 2, 0.25, 3666275, 133613, 44, 3, 16.00, 2354.95, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular") & (1, 3, 0.15, 3666275, 133613, 2567, 3, 16.00, 86674.82, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular").

Note that the original tuple has to be one of the alternative tuples that are generated. In addition, we assign the existence probability with the highest value to the alternative tuples that hold the original tuple's values. We only randomly tamper with the values of the supplier keys (l_suppkey) and the total price of the line (l_extendedprice) in the "lineitem" because we use them in the group by field and the field used in the aggregation function accordingly.

Approaches

We evaluate our TQELX-probabilistic technique against multiple baselines. The first two approaches are deterministic, where they return exact answers rather than probabilistic ones. After ordering the groups based on their max counter value, if we find k groups whose min counter values $\geq$, the query processing stops, and the answer is returned. The exact approach's stopping condition for the top-k query is the same as the one discussed in section 3.3. The two exact baselines are as follows:

- **Random Approach (random).** In this approach, we randomly choose the alternative tuples of one x-tuple to clean in each iteration.
- **Greedy Approach (greedy).** In this approach, we eagerly choose tuples of the group with the highest max counter value for the cleaning process. We specifically choose the dirty tuple with the highest aggregate value and the alternative tuples of the same x-tuple within that group for cleaning.

We, further, compare our TQELX-probabilistic with two probabilistic baselines that follow the same answer semantics but differ in their techniques. They are as follows:

- **NOFILTER Approach.** We do not use the normal approximation filter for this approach but rather evaluate the query after each cleaning function call to clean the minimum number of tuples required.
- **MC-NOOPT Approach.** This approach restrains from using the Monte-Carlo optimizations mentioned in section 4.3 (i.e., the Monte-Carlo simulation is repeatedly run for each validation step).

Queries

We use three queries on the lineitem table with different selectivities to assess the performance of the TQELX-probabilistic approach. They are as follows:

- **Q1.** This query selects the line items of orders that have a commit date between "1994-01-01" and "1994-01-31". The number of tuples retrieved for this query is 230,875 tuples.

- **Q2.** The predicate for this query is the line items of orders with a commit date between "1995-01-01" and "1995-03-31" where we retrieve 672,865 in total.

- **Q3.** For this query, we look at line items of orders in the range of six months such that the commit date of the order is between "1995-07-01" and "1995-12-31". The number of selected tuples is 1,353,673.

We run top-k COUNT, top-k SUM, top-k AVERAGE on each query such that the aggregate field is l_extendedprice and the group-by field is l_suppkey. The top-k aims to retrieve the top-k suppliers that satisfy the top-k query's condition. An example of such a query is:

```
SELECT      l_suppkey
FROM        lineitem
WHERE       l_commitdate BETWEEN "1994-01-01" AND "1994-01-31"
GROUP BY    l_suppkey
ORDER BY    AVERAGE(l_extendedprice) DESC
LIMIT       5
```

For the group-based aggregation queries with the having clause, we find suppliers that satisfy the having condition. We use different values as the numerical operand of the having clause. The first value is the mean value of the aggregate function for all suppliers, where the other values are random numbers from the range of aggregate values. For example we evaluate the following query:

```
SELECT      l_suppkey
FROM        lineitem
WHERE       l_commitdate BETWEEN "1995-01-01" AND "1994-03-31"
```

```
GROUP BY    l_suppkey
HAVING      COUNT(l_extendedprice) >= 50
```

Cleaning Function

In our experiments, given that we have generated the alternative tuples synthetically from the original TPC-H dataset, the cleaning function CLN is equivalent to a lookup operation on the original relation in order to return the original tuple given the different alternative tuples for the same x-tuple. We estimate the cost of each cleaning function to be 50 milliseconds. For instance if the cleaning function CLN taken as input the following alternative tuples: (1, 1, 0.6, 3666275, 133613, 8640, 3, 16.00, 26345.76, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular"), (1, 2, 0.25, 3666275, 133613, 44, 3, 16.00, 2354.95, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular") & (1, 3, 0.15, 3666275, 133613, 2567, 3, 16.00, 86674.82, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular"), it returns the tuple (1, 1, 1, 3666275, 133613, 8640, 3, 16.00, 26345.76, 0.09, 0.07, "R", "F", "1993-04-11", "1993-06-04", "1993-05-01", "DELIVER IN PERSON", "REG AIR", "s. fluffily regular") as the output.

4.4.2 Experiments results

Experiment 1: Top-k Query (Number of cleanings & execution time for different k values)

In this experiment, we evaluate TQELX-probabilistic against the baselines specified earlier to measure our proposed technique's effectiveness in the case of different queries' selectivities. We also measure the impact of the value k for each aggregate function. We have run the top-

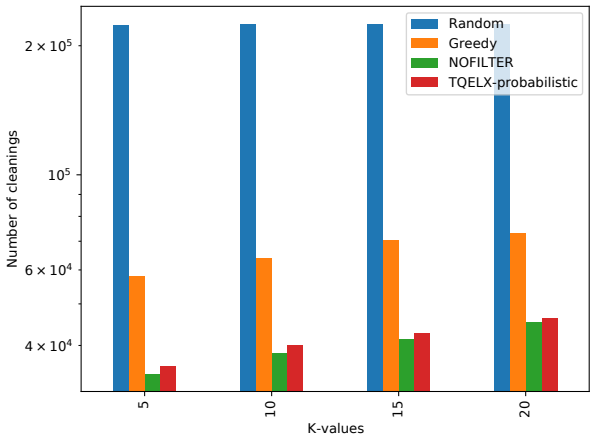

Figure 4.3: Number of cleanings for Top-k AVERAGE of line item prices per supplier

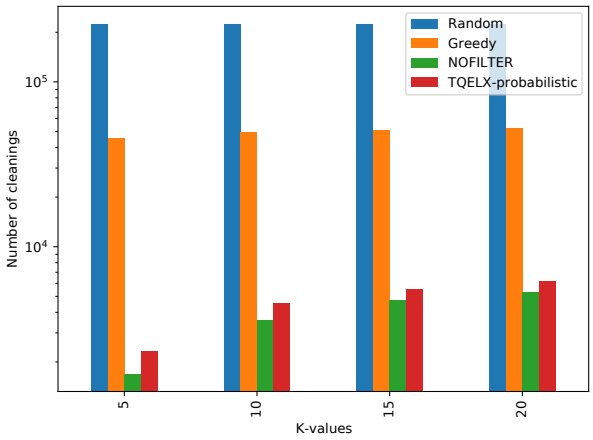

Figure 4.4: Number of cleanings for Top-k SUM of all line item prices per supplier query

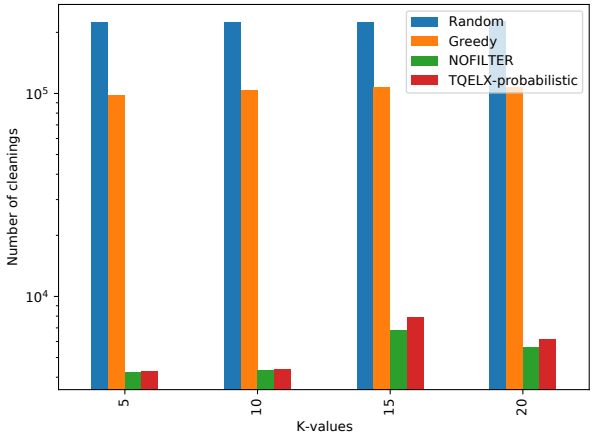

Figure 4.5: Number of cleanings for Top-k COUNT of all line items per supplier query

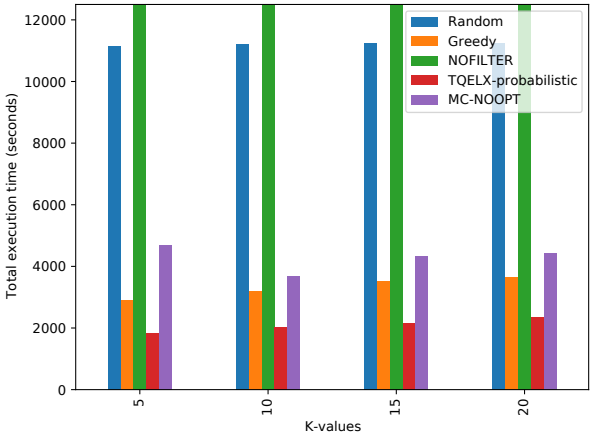

Figure 4.6: Total execution time of Top-k AVERAGE of line item prices per supplier

k query on the three proposed queries and averaged the results for each aggregate function. Given that the cleaning function is a bottleneck of any query-driven query, we measure the number of calls to the cleaning function CLN as well as the total execution time of the query. The total execution time of the query for any probabilistic approach consists of the cleaning time, the Monte-Carlo simulation time and the verification time. We only report the cleaning time for the deterministic approach since the query evaluation time is negligible in our setting.

The probabilistic evaluation approaches showed one order of magnitude in terms of savings

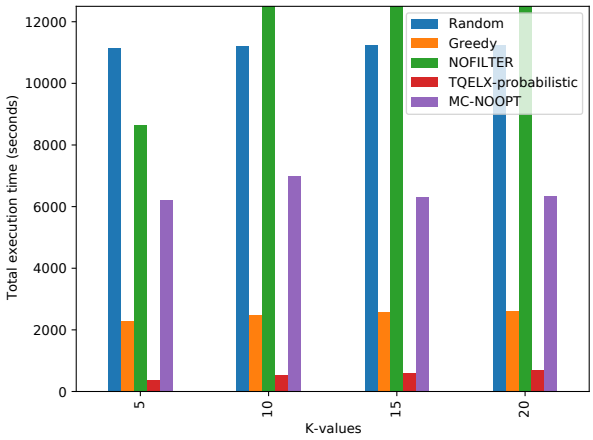

Figure 4.7: Total execution time of Top-k SUM of all line item prices per supplier query

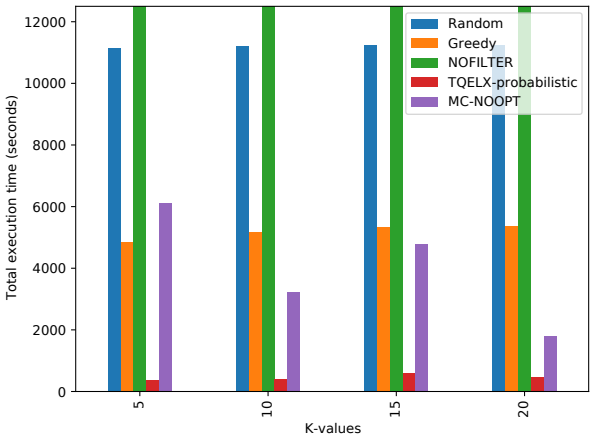

Figure 4.8: Total execution time of Top-k COUNT of line item prices per supplier query

in the cleaning cost compared to the deterministic ones. The main reason is that cleaning is not required for some cases when there is a high chance of a group being in the top-k answer set. On the other hand, the exact approaches will continue the cleaning process until that condition is guaranteed. We also see that as the k value increases, more cleanings would be required to find an answer for most cases. This, however, is not always true, as can be evidenced from the number of cleanings for the top-k count in figure 4.5. The figure shows that for $k = 20$, it required fewer cleanings than it required for $k = 15$. Such cases arise since, as explained in section 4.1, our answer semantics do not require the top-k results to be ordered. Thus, in some cases, less work may be required to find the best 10 items rather than finding the best 5 items.

As illustrated in figures 4.3, 4.4 and 4.5, we clearly see that the aggregate function has an effect on the number of cleanings which also influences the total execution time. Our technique, in general, offers higher savings in cleaning cost for Top-k COUNT queries compared to other aggregation functions. Top-k AVERAGE, in particular, required more cleanings due to two factors. The first factor is that AVERAGE is not monotonic. When a tuple is cleaned and assigned to a certain group, it is not always the case that the total value will increase due to that cleaned tuple being part of that group. For instance, adding the number

5 to an average value of 10 will lower the average rather than increase it and vice versa. The second factor corresponds to the dataset at hand, which has is quite dense in terms of the average value and cleaning the tuples is the only way to achieve the required confidence.

Figures 4.6, 4.7 and 4.8, shows the clear advantage of using TQEL-probabilistic compared to other probabilistic approaches. Although the NOFILTER approach achieved lower savings in cleanings, it failed terribly to provide acceptable results on time. This phenomenon is attributed to repeated evaluation after each cleaning, which required upwards of 2 hours to complete a single query, illustrating the effectiveness of adopting a filter approach that limits the number of evaluations. The MC-NOOPT approach, on the other hand, required more time for the query evaluation due to repeated Monte-Carlo simulation simulations every time the query was evaluated. The execution time of the MC-NOOPT reflects the number of verifications performed as it increases steadily with it. We do not report the number of cleanings for MC-NOOPT as it is the same as TQEL-probabilistic.

Experiment 2: Group-based Aggregation Query (Number of cleanings & execution time)

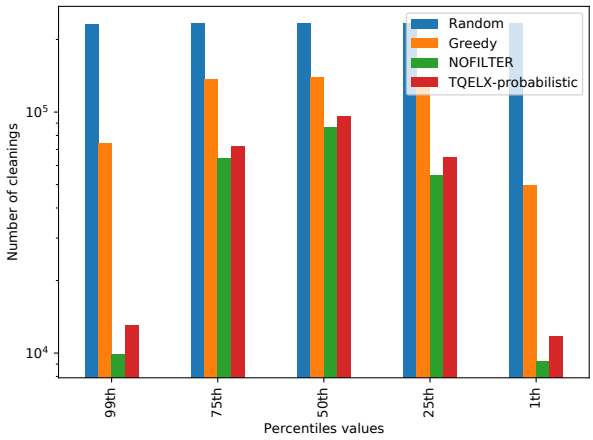

Figure 4.9: Number of cleanings for group-based query with AVERAGE as aggregation function

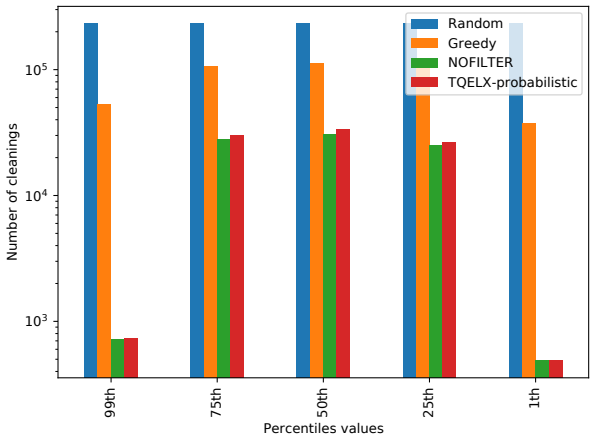

Figure 4.10: Number of cleanings for group-based query with SUM as aggregation function

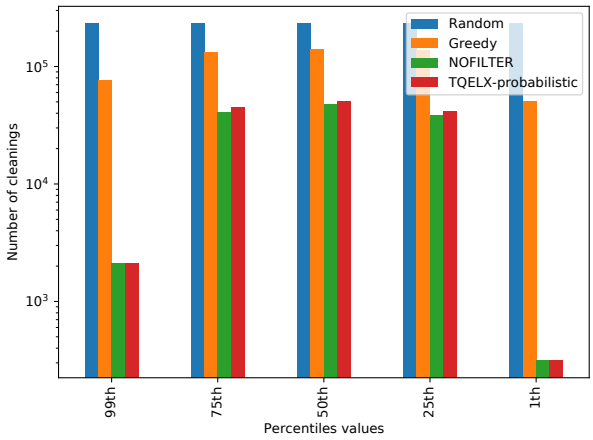

Figure 4.11: Number of cleanings for group-based query with COUNT as aggregation function

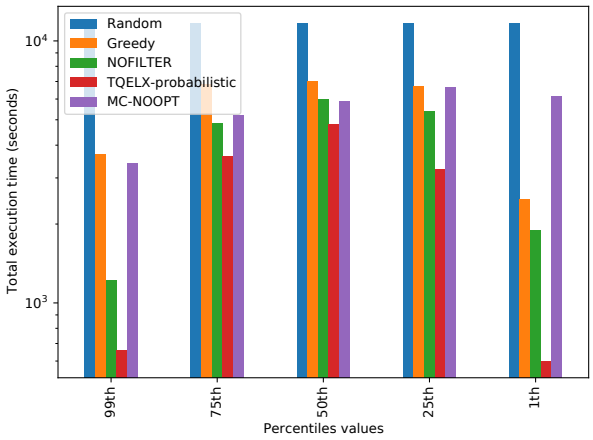

Figure 4.12: Total execution time for group-based query with AVERAGE as aggregation function

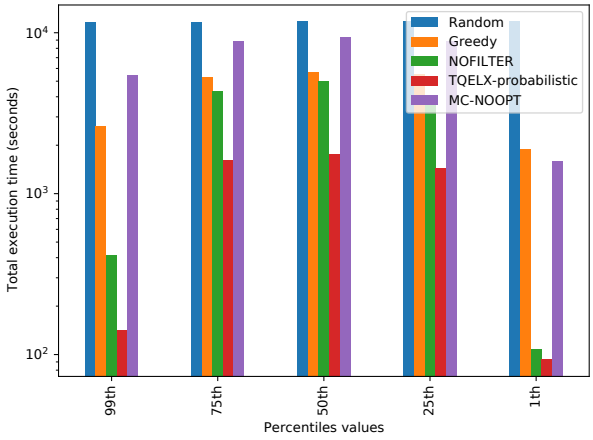

Figure 4.13: Total execution time for group-based query with SUM as aggregation function

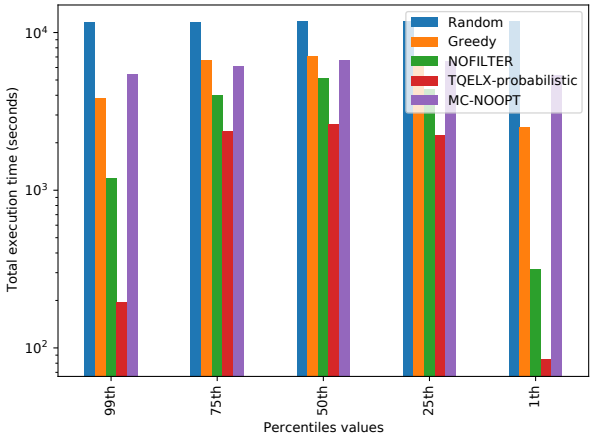

Figure 4.14: Total execution time for group-based query with COUNT as aggregation function

For the second experiment, we analyze the effectiveness of TQELX-probabilistic on group-based aggregation queries with a having clause and measure its performance against other baselines where τ is 95% and the β is 25%. Similar to the first experiment, we plot the number of cleanings and total execution time for each query. Given that the predicate and selectivity of Q1, Q2 & Q3 are different, we wanted to standardize the method for choosing the numerical operand value for each query. We chose five different values for each query using each supported aggregate function. The chosen values represent the values at the

border of percentile ranks (i.e., 99th percentile means that the chosen value is higher than 99% of the values). We average the result of the same aggregate function and the same percentile value to understand the behavior of TQELX-probabilistic with scenarios.

The performance of TQELX-probabilistic depended heavily on the value chosen for the query as it is illustrated in figures 4.9, 4.10 and 4.11. It required more cleanings whenever more groups had a chance of satisfying the query than not. For instance, for the 50th percentile, most groups were candidates to be in the answer set, which required more cleanings to prove the groups that truly are in the answer set. Moreover, the TPC-H dataset has low skewness, which does not help when choosing the mean value for all groups. On the other hand, for the 99th percentile and the 1st percentile, TQELX-probabilistic performed significantly better and had an acceptable query response time as well as a lower number of performed cleanings, which makes it an effective approach for such situations.

Although TQELX-probabilistic performed more cleanings for some queries, it still outperformed the other baselines with the help of the normal approximation filter and the Monte-Carlo simulation optimizations. For the NOFILTER baseline, we modify the execution such that the query executes on one group at a time (i.e., it cleans tuples of one query until it proves it is in the answer set or not). The modification is because the entire query evaluation is wasteful. The groups with no cleaned tuples from the last iteration will return the same answer result regardless of the new execution because we store the Monte-Carlo simulation result. However, the MC-NOOPT baseline does not take advantage of such a feature and will re-run the Monte-Carlo simulation technique whenever the validation step is performed.

Experiment 3: Detailed analysis of queries' performance on different confidence scores

Figure 4.18 illustrates the huge overhead entailed in the form of the number of cleanings when the chosen value is near the mean. This, in turn, affects the number of x-tuples that

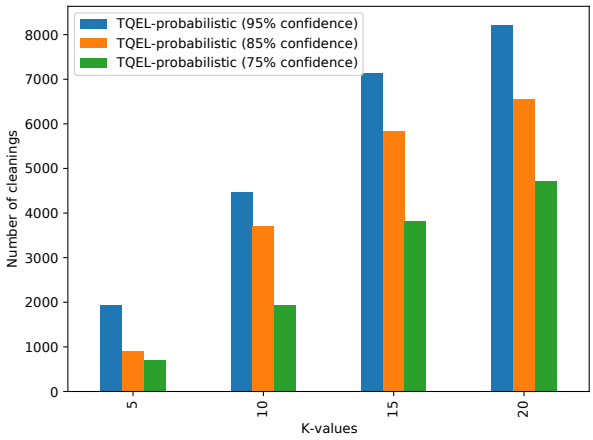

Figure 4.15: Number of cleanings for Top-k queries with different confidence scores

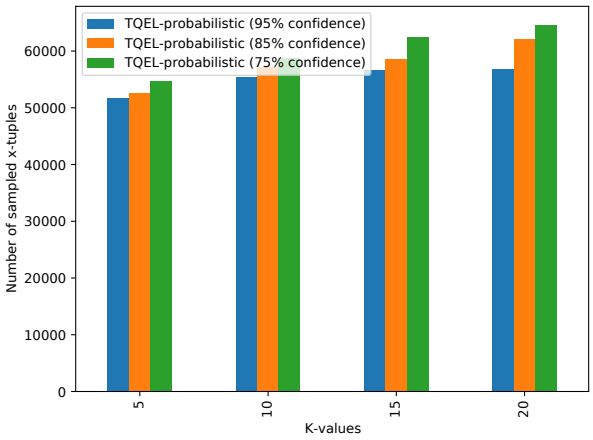

Figure 4.16: Number of x-tuples sampled for Top-k queries with different confidence scores

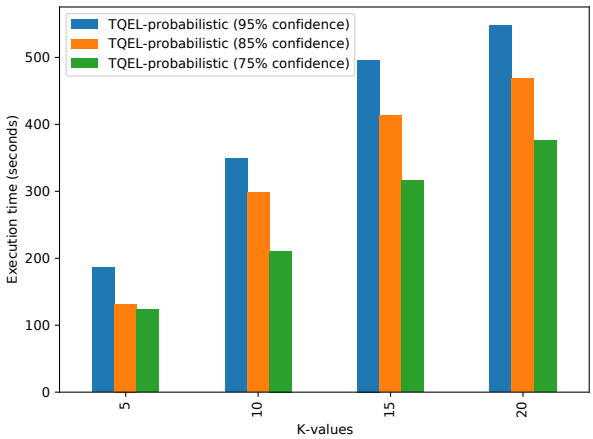

Figure 4.17: Total execution time for Top-k queries with different confidence scores

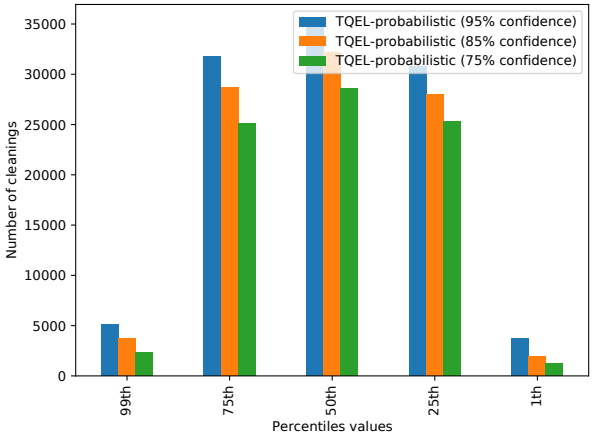

Figure 4.18: Number of cleanings for queries with having clause using different confidence scores

underwent the Monte-Carlo simulation step since the database version is less uncertain. For instance, if query lists groups with COUNT > 50, group G_{c_1} with min value as 49 and max value as 60 is expected to require less Monte-Carlo simulations than group G_{c_2} which has min value of 40 and max value of 60. Finally, the total execution time as shown in figure 4.20 is a reflection of the number of cleanings performed and follows the same trend as such.

In figure 4.15, we see that the confidence score level affects the number of cleanings performed for the top-k query. The reason is that by using a lower confidence score, the filter test will

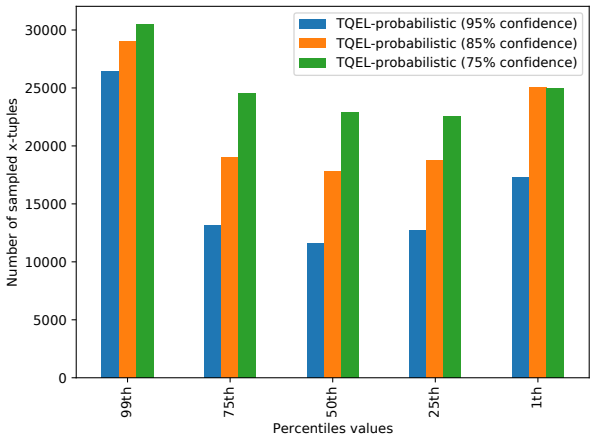

Figure 4.19: Number of sampled x-tuples for queries with having clause using different confidence scores

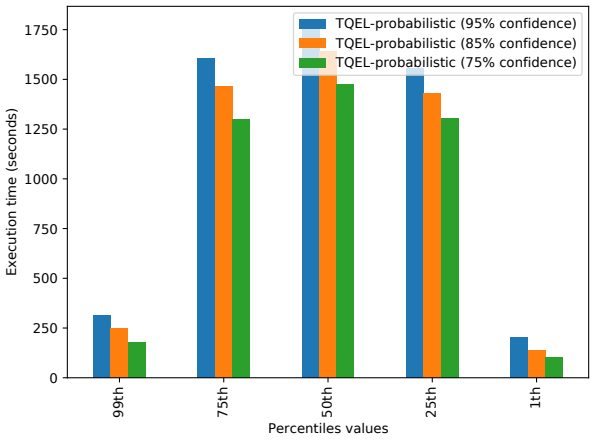

Figure 4.20: Total execution time for queries with having clause using different confidence scores

require fewer cleanings, in general, to pass it, given the lower confidence intervals representing the confidence score. Moreover, this entails more Monte-Carlo simulations, as in figure 4.16, to compensate for the fewer cleanings performed when the confidence score is lower. Figure 4.17 plots the different execution times were in general. The lower the confidence score, the faster the execution. However, this affects the quality of the results, as we will see in the following experiment.

Experiment 4: Accuracy of queries

We report in this experiment the accuracy of each query given the aggregate function and the confidence level. We define different metrics such that we can measure the accuracy of the returned answer, which are as follows:

- **Precision:** represents the quality of the returned answer. We calculate the precision of the answer using the ratio of the number of correct elements in the answer set by the total elements.
- **Recall:** measures the number of elements from the exact answer that TQEL-probabilistic

returns. It can be computed by calculating the ratio of correct elements returned by our technique divided by the number of elements in the exact answer.

- **Rank Distance:** We use the same rank distance proposed in section 3.5 which measures the distance of the return answer from the top-k query compared to the correct answer. The rank distance of 0 indicates that the returned probabilistic answer is the same as the exact answer.

Aggregate Function	Metric	95%	80%	75%
AVERAGE	Precision	0.92	0.77	0.604
AVERAGE	Rank Distance	0.3125	1.175	3.25
SUM	Precision	1	0.89	0.89
SUM	Rank Distance	0	0.433	0.816
COUNT	Precision	0.93	0.92	0.8625
COUNT	Rank Distance	0.27	0.508	0.675

Table 4.6: Evaluation metrics of different confidence levels for the top-k query using different aggregate functions

Aggregate Function	Metric	95%	80%	75%
AVERAGE	precision	0.99	0.95	0.92
AVERAGE	Recall	0.98	0.95	0.91
SUM	precision	1	1	0.95
SUM	Recall	0.97	0.93	0.9
COUNT	precision	1	1	1
COUNT	Recall	0.95	0.9	0.89

Table 4.7: Evaluation metrics of different confidence levels for the group-based query with a having clause using different aggregate functions

For the top-k query, we report the precision and rank distance in table 4.6. The precision of the answer and the rank distance were impacted by the confidence level as expected. The top-k AVERAGE measure was the worst compared to other aggregation functions because the aggregate values of the groups were less skewed than the other aggregate values. In addition, we report the precision and recall for the group-based aggregation queries with a having clause in table 4.7. In general, the scores were relatively high due to the high number of cleanings performed in our experiments for some queries.

4.5 Conclusion

This chapter presents TQELX, a framework that integrates the cleaning process with probabilistic query processing for group-based aggregation queries. The framework can handle top-k aggregation queries as well as group-based aggregation queries with a having clause. The probabilistic query processing follows an approximate confidence computation method by implementing the Monte-Carlo simulation technique. TQELX utilizes a normal approximation filter to estimate the aggregate value of each group in order to reduce the overhead of the probabilistic query evaluation. It leverages the previous work in Monte-Carlo simulation by storing the previously generated samples to speed up the evaluation process. TQELX cuts down the cleaning cost and the total execution time dramatically compared to approaches that return an exact answer. In addition, TQELX offers optimization to the Monte-Carlo simulation technique, which helps stop the query early while providing high confidence guarantees.

Chapter 5

PIVM: Probabilistic Incremental View Maintenance - A Monte-Carlo Approach

As we have seen in previous chapters, integrating the data cleaning mechanism into probabilistic query processing requires repeated query evaluation as the state of the database changes due to data cleaning. A change in the state due to cleaning, in turn, may invalidate the results computed prior to the state update. This is especially a challenge in probabilistic query processing since regenerating the Monte-Carlo samples can be very expensive. As illustrated from the experiments in section 3.5, the probabilistic query execution approach that re-executed the query after each cleaning call (entity linking call in this case) resulted in unacceptable delays where some queries had a total execution time exceeding 5 hours! We see clearly that the repeated executions are infeasible for such queries.

In the previous chapters we proposed two frameworks TQEL & TQELX that provided a mechanism to limit the number of query executions (i.e. the validation step by using Monte-

Carlo simulations and executing the query in each simulated world) by adopting a blocking-like filter based on normal approximation. The aim of the filter is to provide a fast and cheap mechanism for batching of multiple cleaning tasks on different tuples and estimating the validation step outcome with high accuracy. By doing so, most of the queries required three or less iterations of the validation step in order to return an answer for the group-based aggregation queries which was of the order of seconds, and hence tolerable and acceptable.

However, these two frameworks only cater to specific queries such as top-k aggregation and group-based aggregation queries that include a having clause. For more general queries, such as select-project-join (SPJ) queries that are set based, there is no counterpart to a normal approximation filter that can be used to reduce the number of calls to the validation step. We, thus, cannot use the schemes developed in previous chapters.

In this chapter, we focus on developing a new mechanism for SPJ style queries to deal with updates in the database, via cleaning, without having to pay the high cost of Monte-Carlo simulation repeatedly. The approach that we refer to as *Probabilistic Incremental View Maintenance* (PIVM) is based on incrementally updating the answer result. Similar to TQEL, we leverage previous Monte-Carlo simulation runs by temporarily storing the outcome of each sample. Given that each tuple is an independent event (i.e., its existence is not dependent on other tuples), we only need to regenerate the runs for the tuples that are affected by any updates. Moreover, the results generated for untouched tuples should also be preserved and the query should not be re-executed on top of those unchanged tuples.

For example, consider table 5.1 where data regarding cars' registration is collected from different sources and is ingested into a probabilistic database. Say, there are irregularities in the collected datasets such that in 30% of the sources assign the license plate "AAA" to an owner called "John" as seen in the tuple with tid = 1. On the other hand, 70% of the collected datasets have "AAA" license plate registered to another owner, "Amy", and had different car information as seen in the tuple with tid = 2. Table 5.2 has information about

CarRegistration						
xid	tid	License_plate	Owner	Price	Year	P
1	1	AAA	John	60,000	2019	0.3
1	2	AAA	Amy	65,000	2020	0.7
2	3	BBB	Emily	50,000	2018	0.2
2	4	BBB	Trent	45,000	2019	0.4
2	5	BBB	Sal	70,000	2021	0.4

Table 5.1: Dataset for car owners information collected from different sources.

Addresses						
xid	tid	Street	State	Zipcode	Occupant	P
11	11	1 first st	CA	90011	Sal	0.1
11	12	1 first st	CA	90011	Amy	0.9
12	13	2 second st	NY	10001	John	0.25
12	14	2 second st	NY	10001	Emily	0.75

Table 5.2: Addresses collected from different sources.

addresses associated with individuals that are also acquired from multiple different sources. The following query, shown below, lists addresses associated with the license plate of a car by joining the two tables.

```
SELECT  License_plate, Street, State, Zipcode
FROM    CarRegistration AS CR, Addresses AS A
WHERE   CR.owner = A.owner
```

In order to answer the above query using the approximate confidence computation approach (e.g., using Monte-Carlo simulation), samples of all possible worlds are randomly generated and the query is executed on each sampled world to calculate the confidence of each row in the result. In table 5.3, we report a possible answer that would be produced with the confidence of each row if we were to generate 100 world samples. Consider that more datasets about car registration become available, resulting in the probability of the tuples in the corresponding x-tuple to change. Say after acquiring more data; we update the existence probability of

License_plate	Street	State	Zipcode	Confidence
AAA	2 second st	NY	10001	0.09
AAA	1 first st	CA	90011	0.65
BBB	2 second st	NY	10001	0.17
BBB	1 first st	CA	90011	0.04

Table 5.3: Result for probabilistic join query

tuples with tid = 1 & tid = 2 such that their existence probabilities become 0.25 & 0.75 respectively. Such an update will result in the answers returned to the query above becoming obsolete. The update to the tuples would require us to rerun the same query and regenerate the random samples to answer the query. Instead, we could answer the query more efficiently by leveraging the previous runs and generated results. For instance, we only need to generate runs for tuples that share the same x-tuple with xid = 1 instead of generating the samples for all the x-tuples. This would only affect the first and second rows from the result to be recomputed, but no change is required for the third and fourth rows.

PIVM approach, built on top of PostgreSQL, models probabilistic relations and answers probabilistic queries using Monte-Carlo simulation by incrementally updating the results exploiting Incremental View Maintenance techniques [8]. When a query is submitted, in addition to executing the query, we also create temporary tables to store metadata generated after the query execution that will be exploited to speed up future iterations as the data changes (i.e., insert, delete or update). Moreover, PIVM, similar to other IVM approaches, such as DBtoaster [8], generates appropriate triggers that automatically deploy incremental changes to the query result based on updates to the base table without any user intervention. We believe that this is the first contribution to incremental view maintenance on top of probabilistic databases.

In summary the main **contributions** of this chapter are:

- We introduce PIVM, a solution that is built on top of PostgreSQL, which allows for

modeling, answering and incrementally updating the results of the query based on the changes to the probabilistic relations (sections 5.2 & 5.3).

- We discuss the different delta computations needed to efficiently update the previous Monte-Carlo simulation runs and incrementally update the results (section 5.4).
- We experimentally evaluate our solution using different select-project-join style queries (section 5.5).

5.1 Related Work

A view is a virtual relation that is constructed by a query on one or more physical base relations. Views are usually lazily invoked and the query is executed when necessary. On the other hand, a materialized view is eagerly executed, and the results are stored in the database for future queries. However, done naively, for both approaches, any change in the base relations requires the complete re-execution of the views.

Many works have looked at problems associated with the overhead of repeated execution of materialized views when there is an update to the underlying data in the context of deterministic databases. Naively implemented, such updates would require a re-execution of the query associated with the view in order to capture the new changes in the data. One of the first approaches to implement incremental evaluation for views is the counting algorithm [46] that addresses the problem of duplicates in views in case of deletion. In the counting algorithm, a count of each row's derivation in the view is stored in order to efficiently delete tuples in the base relations and propagate that change in the materialized view in the case of duplicates. For example, if the view projects the names of the employees without duplicates and there are two employees who share the same name, then the deletion of the record of one of them should not mistakenly delete the entry in the view given that the other record

is still in the base relation.

Another related early work is ViewCache [77], which uses stored pointers in order to propagate the changes in the underlying table in a lazy fashion. Given a single-block select-project-join style query, the work in [71] presents the concept of propagating the updates, insertions and deletions in the base relations using algebraic differential expressions. The work proposed in [44] follows the same concept and extends the range of supported queries to include, for instance, queries containing aggregation in the presence of duplicates.

Recent work, such as DBtoaster [8], is a database system that offers an approach for efficient delta computation by employing a recursive finite differencing mechanism that materializes the query results and other reciprocal views that are needed for maintaining the materialized query's view when a change happens by combining the analysis of the old and new data. By chopping the delta values into a set of views, the changes in the underlying views propagate to views of higher-order, causing the cost of maintaining the query result to be cheaper. When a query is submitted, triggers are created in order to properly cascade the required delta values in the chain of materialized views. LINVIEW [70] provides a different perspective for handling complex data analytical tasks such as in machine learning jobs. Since these analytical tasks are reduced to a set of linear algebra programs, The proposed framework utilizes matrix factorization approaches to limit the changes in the intermediate delta views in order to reduce the cost of the overall maintenance.

Most of the previous solutions are proposed to tackle the problem of high update rates in different applications that require fast responses. In addition, recent work has also used Incremental View Maintenance (IVM) in the context of query-driven cleaning such as EnrichDB [41], and in intermittent query processing [86].

EnrichDB is a DBMS technology that supports data enrichment on top of newly acquired data. By performing complex functions, EnrichDB prepares the data to be used for different

analytical queries. It integrates the data enrichment task with processing in order to progressively provide answers to the users' queries. It uses an IVM implementation [1] that is built on top of PostgreSQL, which handles the updates caused by the enrichment process in an incremental fashion, which drastically reduces the total time of execution.

While EnrichDB and intermittent query processing have considered using IVM techniques to speed up query processing, neither of these works has explored IVM in the context of probabilistic query processing as we do in PIVM. We note that the previously proposed solution to support incremental query processing using IVM cannot be quickly adopted when dealing with probabilistic relations and queries. We, thus, implement an IVM that is suitable for approximate confidence computation, using Monte-Carlo simulation, in probabilistic queries.

5.2 Preliminaries

In this section, we specify the data & query models and discuss how the PIVM is implemented on top of PostgreSQL.

Data Model

We adopt the same data model that was described in section 4.1 where each x-relation is composed of multiple x-tuples and each x-tuple has mutually exclusive alternative tuples such that for every possible world, there could be only one alternative tuple. We take advantage of the PostgresSQL DBMS powerful engines in order to create such x-relations and run probabilistic queries on top of them. When creating a new x-relation $\mathcal{R}_i$, we have to include an attribute of the "real" domain named "prob" which corresponds to the existence probability of each tuple in the x-relation $\mathcal{R}_i$.

Query Model

We extend the SQL language in order to support a wide range of queries. By adding the notion of confidence (conf()), the confidence of the probabilistic query results can be easily projected and filtered. Given that each tuple $t_{j,y}$ in the x-relation $\mathcal{R}_i$, has an associated existences probability, conf() refers to the confidence score that a row or tuple belongs to the answer of the query. Different SPJ style queries are supported[1] where the conf() of the answer can be either in the selection clause, where clause or/and the having clause (e.g., by limiting the answer of the query to only tuples that have a confidence score of 70% or more). Moreover, the query is associated with the number of Monte-Carlo runs used to compute each row's confidence in the result.

```
SELECT      License_plate, conf()
FROM        CarRegistration
WHERE       year > 2018 AND conf() > 0.7
GROUP BY    License_plate
HAVING      AVG(Price) > 58000
```

For example, the above query on table 5.1 can be expressed in our query model where we report the license plates that have an average price of more than $58,000 and the confidence of each row in the answer.

5.3 Query Processing Implementation

There are several probabilistic databases systems that follow the approximate confidence computation approach to answer queries. Examples include Trio [88], MCDB [52] and MayBMS [12]. We could have possibly built PIVM on top of such systems instead of building on top of PostgreSQL; however, existing systems either do not follow the approximate confidence computation approach for every supported query (i.e., they use exact confidence

[1]We currently do not support self joins, ranking queries and nested queries.

computation for safe-plan queries), execute the sample generation in memory and do not materialize the sample runs that are randomly generated to allow for the re-use of such runs or do not have proper functioning up-to-date code to use for building the Incremental View Maintenance technique.

Hence, we have built a simple query processing solution that rewrites the probabilistic query into multiple deterministic queries in order to support probabilistic query processing in PostgreSQL. Moreover, we have also implemented a Monte-Carlo random sample generator that generates sample runs based on the existence probabilities of the alternative tuples that share the same x-tuple.

A naive implementation of the query processing solution would execute the Monte-Carlo random generator code on every x-tuple in the base relations yielding N random instances of each x-relation. Afterward, the query would be executed on the deterministic instances that have been generated to find and materialize the answer set of each possible world. Then one can find the number of times a row in the answer set has appeared in the N answer sets that have been computed and report the confidence score from such calculations.

However, this implementation suffers from multiple problems as reported in [52]. One major issue with such an approach would be the massive amount of wasted space/time that could be potentially saved if the Monte-Carlo random generation execution was delayed until other operators are executed, which would result in excluding some tuples from the generation process. Another problem is related to the number of times we execute the query - if done naively would require 1 query execution per sample database, thus, resulting in N query executions for N instances that have been generated. Such a naive approach would be very costly.

We, thus, present an approach for probabilistic query execution that addresses both problems. Similar to the query processing in MCDB [52], we delay the Monte-Carlo simulation

process as late as possible in order to filter out tuples that do not satisfy the query's predicate by applying the selection and the join operators on the base x-relations early. Similar to MCDB, we propose a solution that runs the query only once on all the instances. We create a temporary table with the schema <xid, tid, run> such that each row stores the assigned tid for the x-tuple for each run. For example, if we generate a sample run for x-tuple with xid = 1 that randomly chooses tuple with tid = 2 for the third sample, we insert the row <1, 2, 3> in the temporary table. If the Monte-Carlo simulation generates a sample run that assigns xid = 1 to no tuple, we do not store that row in the table. After generating and storing all the required sample runs, one can calculate the number of times tid appears in sample runs by counting the number of times it appears in the temporary table.

We use plpython functions (function written in the Python language within PostgreSQL) to allow users to submit their probabilistic query on top of the x-relations. The function takes as input two parameters: the probabilistic query and the number of iterations for Monte-Carlo simulation N. After the query is received, the predicate is pushed to the base x-relations in order to only retrieve the tuples that satisfy the query's predicate. We next present a step-by-step explanation of the query processing technique.

Step 1: Selection Push-Down

Consider the following query on top of the relation CarRegistration shown in table 5.1:

```
SELECT  tid, conf()
FROM    CarRegistration
WHERE   Price < 55,000
```

Given the above query, we exclude any x-tuple that does not have an alternative tuple such that the value of Price $<$ 55,000. That means tuples with xid = 1 will never be in the answer because both alternative tuples with tid =1 and tid = 2 do not satisfy the query's predicate and therefore will not be used in the samples generation step. On the other hand,

JP					
xid1	tid1	p1	xid2	tid2	p2
2	3	0.2	12	14	0.75

Table 5.4: An example of a temporary table JP that holds the deterministic join result

tuples with xid = 2 will be used in the samples generation step because at least one of the alternative tuples satisfies the query predicate. In this case, both alternative tuples with tid = 3 and tid = 4 do. This process is done by executing a query with the same predicate "Price < 55,000" to retrieve the xid, tid and the existence probabilities of each tuple in the answer. For example, the result set of this step would be {(2, 3, 0.2), (2, 4, 0.4)}. Note that we do not include any clauses that has conf() or p as an operand (e.g., if predicate is "Price < 55,000 AND conf() > 0.5" we then discard "conf() > 0.5" in this step).

Step 2: Elimination of Unjoined Tuples

If the submitted query is a join query, we would perform a deterministic join and temporarily save the result in a relation called JP. In order to find the join results, we treat the x-relations $\mathcal{R}_1$, $\mathcal{R}_2$, ..., $\mathcal{R}_n$ as deterministic tables and join them by only projecting the xid, tid and existence probability of each tuple from each relation. This process identifies the tuples that could potentially be in the join result and excludes tuples that will not. Consider the following query:

```
SELECT  cr.tid, a.tid, conf()
FROM    CarRegistration AS cr, Addresses AS a
WHERE   cr.Owner = a.Occupant AND Price < 55,000
```

After executing the deterministic join on relations CarRegistration & Addresses, we get the following temporary table (JP) as illustrated in 5.4. By executing the deterministic join, we can eliminate tuples with xid = 1 in the CarRegistration relation as well as tuples with xid = 11 from the Addresses relation because they will never be in the answer set regardless of

CarRegistration_sim		
xid	tid	run
2	4	1
2	3	2
2	4	5
2	4	6
2	3	7
2	4	10

Table 5.5: An example of a simulation table for CarRegistration relation

their existence probability values. We then retrieve the set of tuples that will be used for the generation step for each relation. The set of retrieved tuples for the CarRegistration relation is {(2, 3, 0.2)} and the set of retrieved tuples for the relation Addresses is {(12, 14, 0.75)}. We then retrieve all the alternative tuples with xid = 2 that satisfy the query's predicate from the CarRegistration relation and all the alternative tuples with xid = 12 that satisfy the query's predicate from the Addresses relation. Including all the alternative tuples is necessary to recompute the query results in the event of updates to the base relations, as we will see in the following section.

Step 3: Monte-Carlo Simulation

In this step, we create a temporary simulation table $\mathcal{R}_i$_sim for each x-relation $\mathcal{R}_i$ in order to store the results of the Monte-Carlo simulation step. Note that if some of the alternative tuples of x-tuple x_j satisfy the query's predicate or are in the deterministic join result, we only store the runs that randomly choose the participating alternative tuples (i.e., if tuple $t_{j,y}$ is not a participating tuple and was randomly chosen as the exclusive tuple for x_j in the run r_1, then we do not store it in the $\mathcal{R}_i$_sim table). The schema of the simulation table $\mathcal{R}_i$_sim is <xid, tid, run>. Consider the same query as in step 1 and the number of Monte-Carlo samples is 10; one possible instance of the CarRegistration_sim would be illustrated in table 5.5.

In table 5.5 we see that the chosen alternative tuple for x-tuple with xid = 2 in the first sample run is tuple with tid = 4. In the second run, the chosen alternative tuple is the tuple with tid = 3. Moreover, if the chosen tuple is a tuple that does not satisfy the query's predicate, such as in the third run, we do not insert the data for that run since it will not be used in the next steps or future delta computations of queries. In the case of a join query, we create a simulation table for each base table.

Step 4: Aggregate Values Calculation

CarRegistration_sim		
xid	tid	run
1	1	1
1	2	2
1	2	3
1	2	4
1	1	5
1	2	6
1	1	7
1	2	8
1	2	9
1	2	10
2	4	1
2	3	2
2	5	3
2	5	4
2	4	5
2	4	6
2	3	7
2	5	8
2	5	9
2	4	10

Table 5.6: An example of a simulation table for the aggregation query on the CarRegistration relation

AG	
run	aggregation
1	105,000
2	115,000
3	135,000
4	135,000
5	105,000
6	110,000
7	110,000
8	135,000
9	135,000
10	110,000

Table 5.7: An example of a simulation table for the aggregation query on the CarRegistration relation

If a query is an aggregation query, we create another temporary table called *AG* that holds the aggregation results for each run, such as SUM, AVERAGE and MAX. We store such

values to avoid recomputing the aggregates for each sampled world in case of updates in the base relation. Note that we perform this only once by performing a group-by query where the grouping field is the run columns of the simulation table. We follow the same process for the group-based aggregation query by storing the aggregate value for each group and each run. Consider the following aggregation query:

```
SELECT  SUM(Price), conf()
FROM    CarRegistration
```

Table 5.6 is a possible instance of the simulation table for the CarRegistration relation and table 5.7 is the aggregate values table AG that has the aggregate value of each run. The query that populates the AG table is as follows:

```
SELECT      sim.run, SUM(cr.Price)
FROM        CarRegistration AS cr, CarRegistration_sim AS sim
WHERE       cr.tid = sim.tid
GROUP BY    run
```

Note that in the case of the aggregate function AVERAGE, we also add another column for storing the count of tuples in order to update the average value efficiently. If the query is a group-based query, we add the grouping fields as columns in the AG table and store the aggregate values for each group per run.

Step 5: View Creation

IV	
tid	conf
3	0.2
4	0.4

Table 5.8: An example of a view for the CarRegistration relation

Finally, we create an incremental materialized view named IV, which stores the query result

where each row has a confidence score calculated from the sampled worlds. In the case of aggregation queries, we calculate the confidence score from the AG table since it has the calculated aggregates for each sampled world. Considering the query in the first step and the simulation table 5.5; we can populate the materialized view IV in table 2 using the following query:

```
SELECT      tid, COUNT(*) / 10 AS conf
FROM        CarRegistration_sim
GROUP BY    tid
```

In the case of a condition on the confidence of the row in the elements to be more than a specified confidence score, we add a having clause that limits the results to be above a specific conf value (e.g., HAVING COUNT(*) / 10 $>$ 0.5).

If the query is a join query, we first create the JP temporary table, which holds the possible joins rows, then we create a simulation table for each of the base relations. Hence, to calculate the incremental materialized view IV for the join query, we use the following query:

```
SELECT      jp.tid1, jp.tid2, COUNT(*) / 10 AS conf
FROM        CarRegistration_sim AS sim1, Addresses_sim AS sim2,
            JP AS jp
WHERE       jp.tid1 = sim1.tid AND jp.tid2 = sim2.tid
            AND sim1.run = sim2.run
GROUP BY    jp.tid1, jp.tid2
```

Given that we have calculated the deterministic join results in JP, we guarantee that we only project tuples that satisfy the join condition by joining the tables. Moreover, we enforce that the join has appeared in the same sampled world by adding the "sim1.run = sim2.run" in the where clause.

For the aggregation queries, we follow the same process to populate the IV table. Consider

the aggregation query in step 4, the simulation in table 5.6 and the aggregation table AG in 5.7, then we execute the following query:

```
SELECT      aggregation, COUNT(*) / 10 AS conf
FROM        AG
GROUP BY    aggregation
```

Note that if the query is a group-based query and the grouping field is year then the query would be as follows:

```
SELECT      year, aggregation, COUNT(*) / 10 AS conf
FROM        AG
GROUP BY    year, aggregation
```

If the query projects fields other than the tid, we join the simulation table with the base table to retrieve the projected attributes' values. However, every result will require the projection of the tid attribute if the query is not an aggregation query.

5.4 Delta Computations of Queries

In this section we define the delta computations needed in order incrementally update the incremental materialized view IV.

We represent the delta computations as triggers that are created for the base x-relations of the query. We add two triggers, one for insertions to the base relation and the other for the deletion of tuples. We only consider the insertion and deletion of the entire alternative tuples of the same x-tuple since it changes the query result (e.g., when deleting $t_{j,y}$ we delete all alternative tuples that share the same x_j and vice versa).

Let $\Delta\mathcal{R}$ be the delta tuples that have been inserted or deleted from x-relation $\mathcal{R}$, $pred(\mathcal{Q})$ be

the predicate of the query $\mathcal{Q}$ without including the conf() clause. Moreover, let $\mathcal{R}$_sim be the Monte-Carlo simulation table for the x-relation $\mathcal{R}$ and IV to be the incremental materialized view for the query $\mathcal{Q}$. If an x-tuple is updated (e.g., changing the probability value of an alternative tuple), we model such event as a deletion then an insertion to the base relation.

Selection Query (Single Relation)

In the case of deletion, we simply delete $\Delta\mathcal{R}$ entries corresponding to tuples in $\mathcal{R}$_sim and from IV as follows:

$$\Delta\mathcal{R}\_sim = \Pi_{xid,tid,run}\Delta\mathcal{R} \bowtie \mathcal{R}\_sim$$

$$IV' = IV - \Pi_{tid,conf}(IV \bowtie \Delta\mathcal{R}\_sim)$$

$$\mathcal{R}\_sim' = \mathcal{R}\_sim - \Delta\mathcal{R}\_sim$$

where IV' is the new incremental materialized view and $\mathcal{R}$_sim' is the new simulation table for relation $\mathcal{R}$.

In the case of insertion, we first apply the predicate selection on $\Delta\mathcal{R}$. We then run the Monte-Carlo simulation technique on the satisfying tuples and add the results to IV along with the ratio of the number of times the results appear in $\Delta\mathcal{R}$_sim by N.

$$\Delta\mathcal{R}\_sim = MCSIM(\sigma_{pred(\mathcal{Q})}\Delta\mathcal{R})$$

$$\mathcal{R}\_sim' = \mathcal{R}\_sim \cup \Delta\mathcal{R}\_sim$$

$$IV' = IV \cup ({}_{tid}\mathcal{G}_{count(run)/N}\Delta\mathcal{R}\_sim)$$

Join Query

For join queries of $\mathcal{R}$ & $\mathcal{S}$, the deletion process is similar to the deletion process for selection queries where the entries of $\Delta\mathcal{R}$ are deleted from $\mathcal{R}$_sim and IV as well. The exact process is applied to $\mathcal{S}$. However, in the case of insertions, we create the temporary join table JP' using the following expression:

$$JP' = \sigma_{pred(\mathcal{Q})}\Delta\mathcal{R} \bowtie \sigma_{pred(\mathcal{Q})}\mathcal{S} \cup \mathcal{R} \bowtie \sigma_{pred(\mathcal{Q})}\Delta\mathcal{S} \cup \sigma_{pred(\mathcal{Q})}\Delta\mathcal{R} \bowtie \sigma_{pred(\mathcal{Q})}\Delta\mathcal{S}$$

In the case of multiple joins, the JP' table will hold tuples that deterministically join among all joined x-relations. Afterward, we create a delta simulation table for each base relation, including the new tuples the deterministic join added. If an x-tuple has been previously used for sampled generation appears in the JP' table, we do not need to rerun the Monte-Carlo simulation for such an x-tuple. On the other hand, newly added x-tuples will have to be used for the Monte-Carlo simulation samples generation. We also add the newely generated samples to the simulation tables of each base table.

$$IV' = IV \cup {}_{tid1,tid2}\mathcal{G}_{count(run)/N}(\mathcal{R}\_sim \bowtie_{run=run} \Delta\mathcal{S}\_sim \bowtie_{\mathcal{R}\_sim.tid=tid1 \wedge \Delta\mathcal{S}\_sim.tid=tid2} JP')$$

$$\cup {}_{tid1,tid2}\mathcal{G}_{count(run)/N}(\Delta\mathcal{R}\_sim \bowtie_{run=run} \mathcal{S}\_sim \bowtie_{\Delta\mathcal{R}\_sim.tid=tid1 \wedge \mathcal{S}\_sim.tid=tid2} JP')$$

$$\cup {}_{tid1,tid2}\mathcal{G}_{count(run)/N}(\Delta\mathcal{R}\_sim \bowtie_{run=run} \Delta\mathcal{S}\_sim \bowtie_{\Delta\mathcal{R}\_sim.tid=tid1 \wedge \Delta\mathcal{S}\_sim.tid=tid2} JP')$$

Aggregation Query

Delta queries for aggregation queries and group-based aggregation queries are more complex since we need to update the aggregates in the case of deletions as well as insertions. We first need to update the aggregates values for each run in the AG table and each group if groups are used before deleting or inserting the entries of $\Delta\mathcal{R}$.

$$\Delta AG = {}_{run[,group]}\mathcal{G}_{\alpha\ as\ aggregation}\Delta\mathcal{R}\_sim[\bowtie \Delta\mathcal{R}]$$

where α is the aggregation function.

In case of deletion, we subtract the ΔAG' table values for each run as follows:

$$AG' \leftarrow \Pi_{run[,group],AG.aggregation-\Delta AG.aggregation}(AG \bowtie_{run=run[\wedge group=group]} \Delta AG)$$

In case of insertion, we add the ΔAG' table values for each run as follows:

$$AG' \leftarrow \Pi_{run[,group],AG.aggregation+\Delta AG.aggregation}(AG \bowtie_{run=run[\wedge group=group]} \Delta AG)$$

Once the AG table is updated we then we then need to truncate the contents of IV and repopulate the results over again. In the case of group-based queries we only delete the results of the affected groups rather than deleting all the entries. The reason behind repopulating the view content rather than updating the current values is that the insert operation is faster than the update operation. The expression for populating the IV for aggregation over the entire table is as follows:

$$IV' = {}_{aggregation}\mathcal{G}_{count(run)/N \text{ as } conf} AG'$$

For group-based queries we first delete the entries in the results that contain the effected groups and then we add the new aggregate value instead.

$$IV' = IV - {}_{aggregation,group}\mathcal{G}_{count(run)/N} AG$$

$$IV'' = IV' \cup {}_{aggregation,group}\mathcal{G}_{count(run)/N} AG'$$

where IV" is the new incremental materialized view.

Note that when the aggregation function is either MAX or MIN we compare them to the current values in the AG table and only update them if the Δ values would effect the MAX

or MIN values.

5.5 Experiments

In this section we evaluate the performance of PIVM by analyzing different queries and their delta queries execution times.

5.5.1 Experimental Setup

Datasets

For experimental purposes, we have used the same dataset that was synthetically generated from the TPC-H dataset. The details of the generation process can be found in 4.4.1.

Database System

We have used PostgreSQL 13 to implement our functions for processing probabilistic queries following the approximate confidence calculation approach using the Monte-Carlo simulation technique. We also use plpython for generating triggers that allow for delta computation of inserted or deleted tuples. We have turned off the auto commit flags and disabled WAL mode to offer fast ingestion of the sample runs' tables.

5.5.2 Queries

Query 1: Selection Query

The first query finds the line items from orders committed in 1994 and have been flagged as a return order. The SQL for this query is as follows:

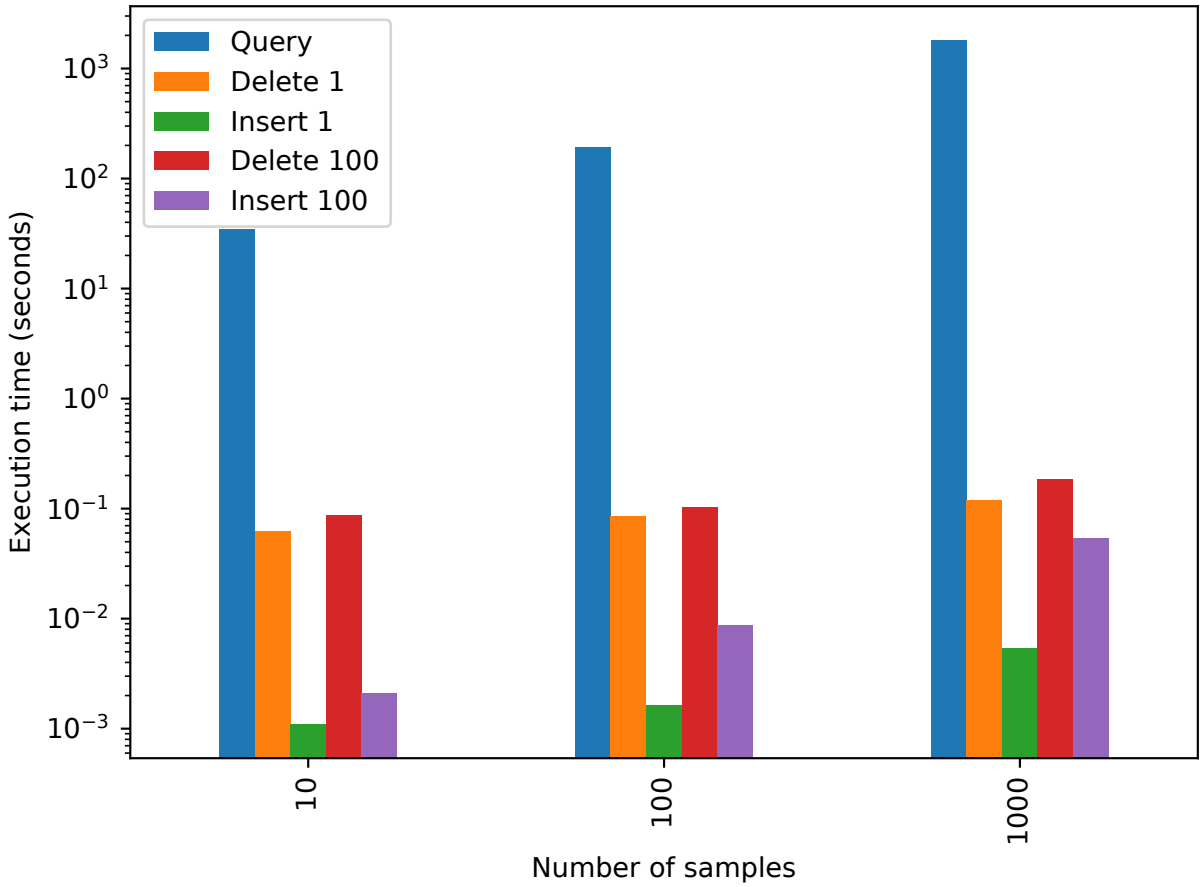

Figure 5.1: Execution times for query 1, insertion and deletion times.

```
SELECT   tid, conf()
FROM     lineitem
WHERE    l_commitdate BETWEEN "1994-01-01" AND "1994-12-31"
         AND l_returnflag = "R" AND conf() > 0.5
```

The resulting tuples from running this query without checking the confidence clause are 1,808,144, and we will only be running the Monte-Carlo simulation code on the filtered tuples. Also, the query will filter out 90% of the tuples in the base table. The disk space absorbed by each simulation table is 192 MB, 1921 MB and 19 GB for 10, 100 and 1000 runs, respectively. On the hand, the size of the view table was the same for different simulation runs which is 19 MB.

We plot in figure 5.1 the times for query execution, the insertion of one tuple, deletion of one tuple, and the insertions and deletions of 100 tuples. The y-axis shows the number of seconds it takes to execute the task, while the x-axis is the number of Monte-Carlo simulation samples. We see that the main factor that affects the query execution is the number of Monte-Carlo samples. On the other hand, the delta computation doesn't seem to be highly affected by the number of samples, given that we only insert or delete a small number of tuples even if the number of samples is 1000. The reason that the deletion takes a bit more time is due to the fact that deletion in PostgreSQL is slower compared to insertion.

Query 2: Aggregation Query

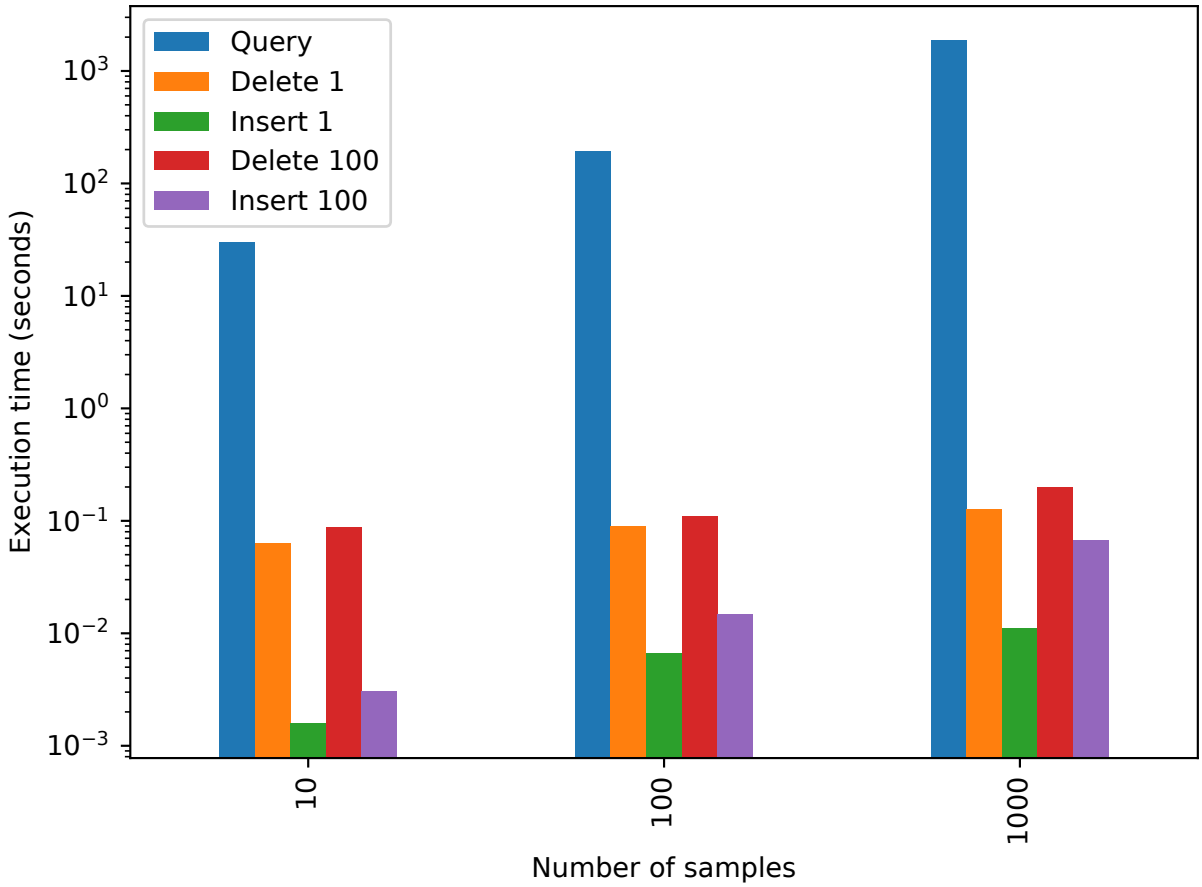

Figure 5.2: Execution times for query 2, insertion and deletion times.

We used the same predicate for the second query to have the same selectivity to have a relative comparison between the queries. The query returns the count of the number of tuples that satisfy the predicate and as follows:

```
SELECT   COUNT(tid), conf()
FROM     lineitem
WHERE    l_commitdate BETWEEN "1994-01-01" AND "1994-12-31"
         AND l_returnflag = "R" AND conf() > 0.5
```

Note that the confidence here will only return the count that appears in more of 50% of the worlds if any.

In figure 5.2 we see the same trend as query 1 as it shows lower running times for delta queries regardless of the number of samples generated. The only difference is that we have to update the temporary aggregation table after each insert or each delete so that we do not have to recompute the aggregation value for each sampled world each time. Moreover, since the number of runs or worlds is low, the update process takes no time for each delta tuples and therefore doesn't affect the total execution time. In terms of the size of the view, it was approximately 8 KB for 10 and 100 simulation runs and 40 KB for 1000 simulation runs. The AG table occupied the same disk space as well.

Query 3: Group-based Aggregation Query

For this query, we also use the same predicate as the previous query. This query calculates the average prices of line items per supplier for the same period along with confidence score of each average per supplier. The query is as follows:

```
SELECT       l_suppkey, AVG(l_extendedprice), conf()
FROM         lineitem
WHERE        l_commitdate BETWEEN "1994-01-01" AND "1994-12-31"
             AND l_returnflag = "R"
```

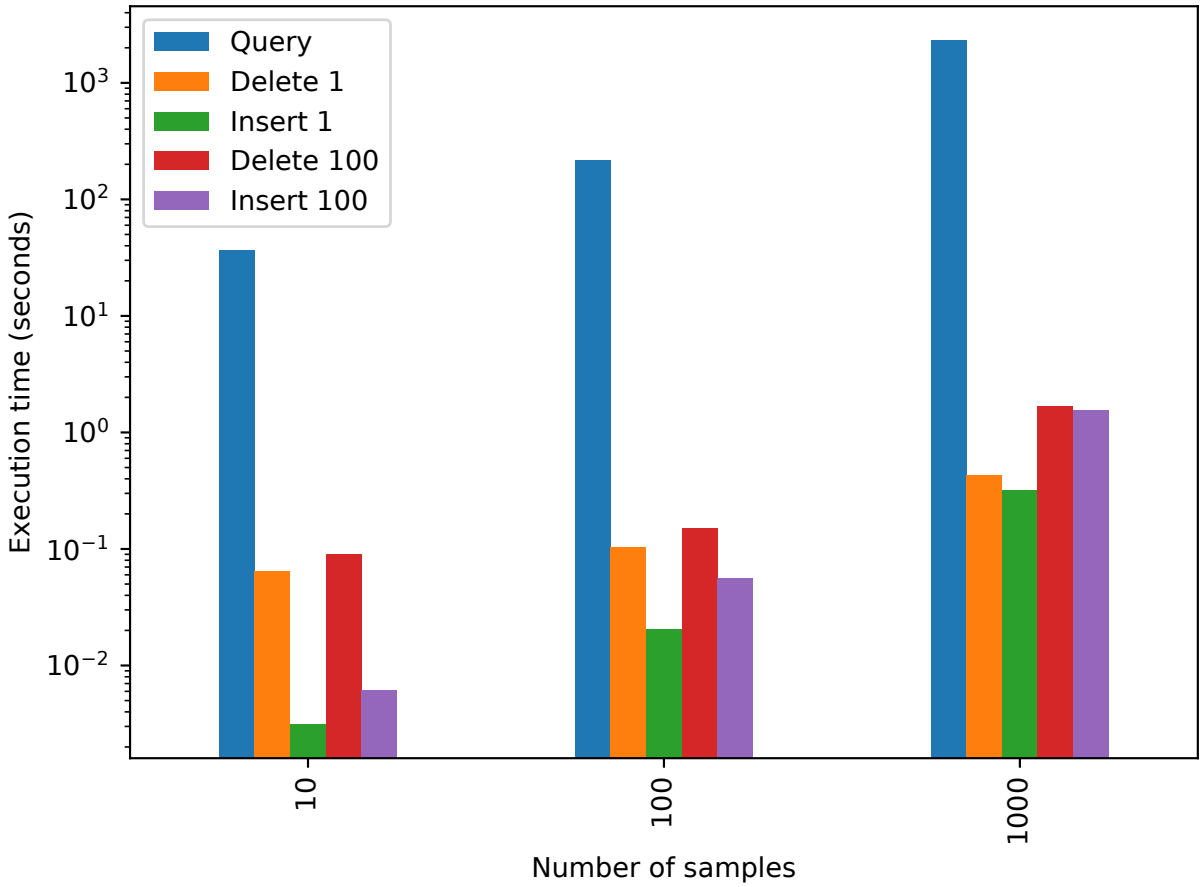

Figure 5.3: Execution times for query 3, insertion and deletion times.

```
GROUP BY    l_suppkey
```

In figure 5.3, the delta computation execution time is higher than the previous two queries. The cause of the increase in execution time is that the temporary aggregation table holds aggregate values for all groups per run. We do so to make it easier to update each group value without recomputing the rest of the group values in our query view. Therefore, the update will only affect the rows that hold the values for the groups of the delta tuples only. However, this requires updating all the groups involved, too, which entails more time for execution. The disk space allocated for the view in this experiment was 3544 KB, 35 MB and 340 MB where the number of samples is 10, 100 and 1000 respectively.

Query 4: Group-based Aggregation Query with Having Clause

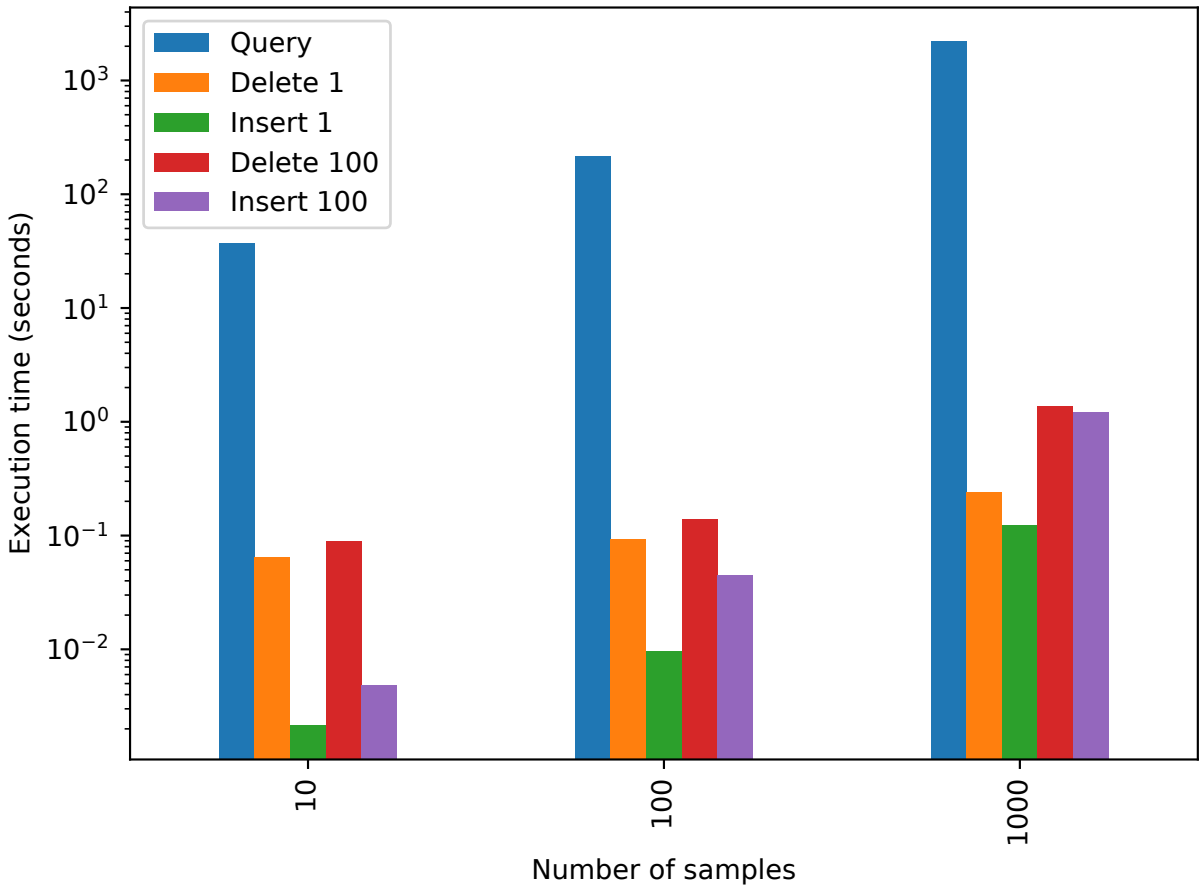

Figure 5.4: Execution times for query 4, insertion and deletion times.

For this query, we find the suppliers with the total sum of prices of returned line items sold in 1994 to be more than $1,600,000. We only return suppliers with %90 confidence that they satisfy the query's predicate. This query, in essence, is one of the queries that would be answered using the TQELX framework. The query is as follows:

```
SELECT      l_suppkey
FROM        lineitem
WHERE       l_commitdate BETWEEN "1994-01-01" AND "1994-12-31"
            AND l_returnflag = "R"
GROUP BY    l_suppkey
```

```
HAVING      SUM(l_extendedprice) > 1,600,000 AND conf() > 0.9
```

Note that we do not return the confidence score nor the sum of prices here as we are only interested in the suppliers that satisfy the condition.

The times reported for query 3 are similar to those reported for this query since they are both group-based aggregation queries. The key difference here is that we only report the suppliers that satisfy the having clause over all possible worlds rather than reporting every possible value as we did for query 3. For this query, the size of the view was approximately 232 KB for the different numbers of simulation samples.

Query 5: Join Query

In this query, we examine the performance of the join operator in the case of approximate confidence computation. We use the same predicate as in the previous queries for the line item table and join it with a probabilistic version of the supplier table. The query is as follows:

```
SELECT  r1.tid, t2.tid, conf()
FROM    lineitem AS r1, supplier AS r2
WHERE   r1.l_suppkey = r2.s_suppkey AND l_returnflag = "R"
        AND l_commitdate BETWEEN "1994-01-01" AND "1994-12-31"
```

We plot in figure 5.5 the execution times of the query, the insertion of one tuple to both tables and the deletion of one tuple from both tables as well. Moreover, we report the execution times for inserting 100 tuples into each table and deleting the same number of tuples from the tables. Contrary to the previous queries, we see that the insertion process takes more time to execute than the deletion process. This is due to the fact that when we delete tuples, we delete them from the temporary table that holds the simulation runs results and the view directly without the need for other actions. However, in the case of insertion, we need to join

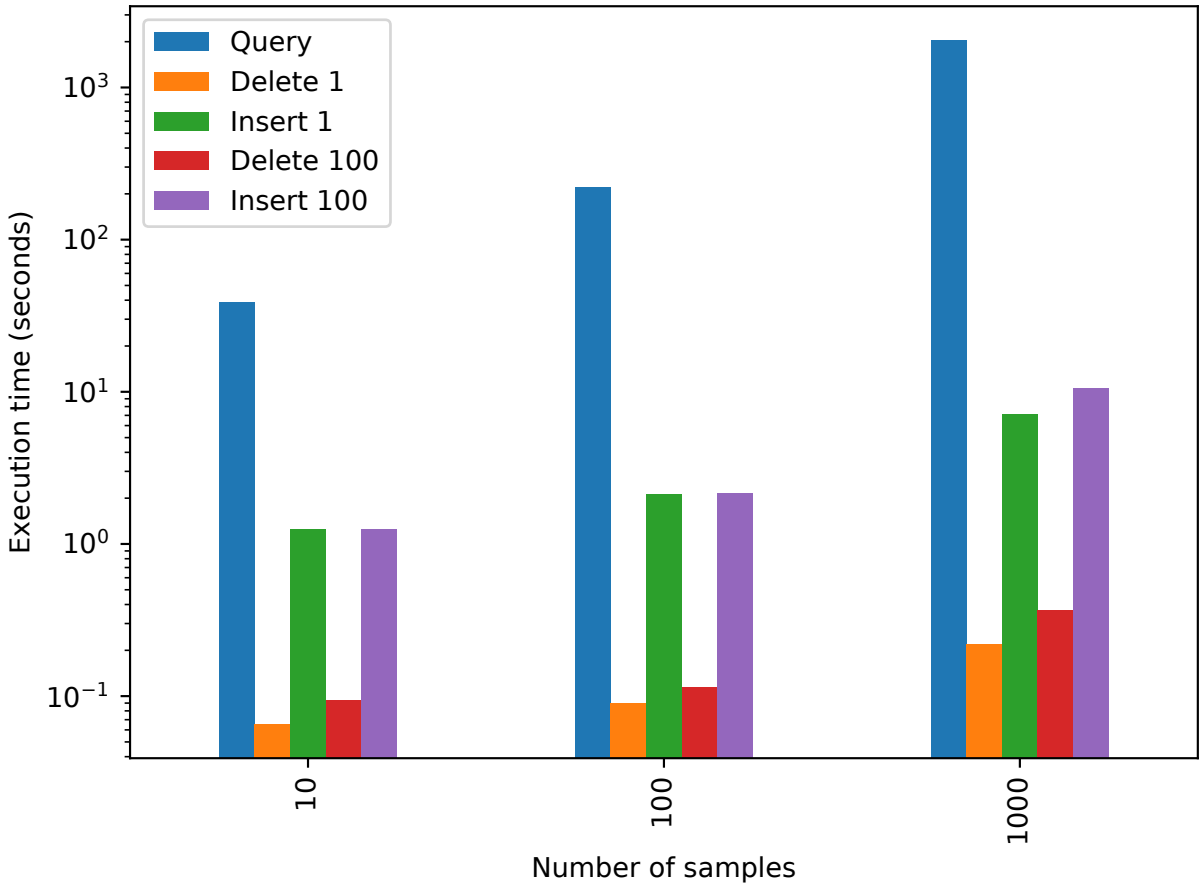

Figure 5.5: Execution times for query 5, insertion and deletion times.

the delta tuples with the base table of the other relations. In addition, we also need to join the sample runs tables in order to find the number of worlds this join actually occurs and add it to the view result. The reported size of the view was 19 MB for different simulation runs.

5.6 Conclusion

This chapter presents PIVM, a solution implemented on top of PostgreSQL, to represent uncertain relations and answer probabilistic queries following the approximate confidence computation method. Using the Monte-Carlo simulation technique, samples of the possible

worlds are created and used for probabilistic query evaluations. In addition, PIVM incrementally updates the query's result in the event of insertions, deletions and updates to the base relations. It deploys triggers depicting the different delta computation expressions to update the query's result without rerunning the entire query execution. Our experiments showed that the incremental view maintenance technique adopted by PIVM offers a tremendous speed-up for single or batched updates.

Chapter 6

Conclusions & Future Work

6.1 Conclusions

In this thesis, we addressed the problem of query-driven cleaning of group-based aggregation queries on top of probabilistic databases. In addition, we propose solutions that offer delta computation techniques to answer probabilistic queries incrementally.

Chapter 3 explored query-driven entity linking in the context of top-k queries on top of social media blogs. We introduced TQEL, a framework that offers an exact solution for answering the top-k query and an approximate one. TQEL-exact exact iteratively calls the entity linking task until an exact solution to the-k is found. On the other hand, TQEL-approximate uses the Monte-Carlo simulation technique to return an answer with guarantees. TQEL-approximate reduces the overheads incurred by the entity linking process as well as the repeated evaluation using the Monte-Carlo simulation technique. It utilizes a blocking-like filter based on normal approximation that estimates the count of each entity to prevent the repeated execution of the query by combining multiple entity linkings in this same iteration. It further leverages the previous Monte-Carlo runs to avoid the regeneration of such samples.

In addition, we empirically studied the performance of the two solutions on multiple Twitter datasets, which demonstrated the significant advantage of TQEL-approximate in terms of savings in the number of linkings compared to the other approaches.

In chapter 4, we extended the solutions of the previous chapter and introduced TQELX, a general analysis-aware framework for group-based aggregation queries. TQELX can answer top-k aggregation queries as well as a group-based aggregation query with a having clause on top of probabilistic databases. TQELX exploits the query semantics of both queries to efficiently select the least number of tuples to clean and return an answer with probabilistic guarantees. Like TQEL, it uses the Monte-Carlo simulation technique for the query evaluation step and implements multiple optimizations to speed up the execution. We have conducted multiple experiments to measure the effectiveness of TQELX for multiple aggregation functions on a synthetically generated dataset.

In chapter 5, we presented PIVM, a probabilistic Incremental View Maintenance technique built on top of PostgreSQL, which provides efficient delta computations in the case of updates to the database. PIVM allows for modeling uncertain relations and submitting select-project-join probabilistic queries that are incrementally updated by generating triggers on the base relations of the query. We tailor our solution to the probabilistic queries that follow the approximate confidence computation technique by implementing a Monte-Carlo simulation generation algorithm and storing the results of such runs for future iterations. Finally, We empirically conduct experiments to measure the speedup provided by PIVM using multiple queries. Our results show the effectiveness of adopting such solutions where the updates were incrementally reflected in the answers within milliseconds to seconds.

6.2 Future Work

In this thesis, we have only tackled the problems concerning the integration of cleaning within probabilistic group-based aggregation queries. However, these problems do not capture the complete picture, nor do they provide a comprehensive solution for the problem of cleaning in the context of probabilistic query processing. Different directions can be explored and researched in order to provide more solutions for such a domain.

One direction of potential work is exploring and proposing a different filter mechanism suitable for non-aggregation probabilistic queries. Such a filter would help cut down the expensive cost of the repeated execution for the probabilistic query processing as well as guide the cleaning process in order to limit the number of cleanings. For example, one possible filter that is applicable for a general class of queries is an incremental sampling filter. This can be achieved by running a small number of sampling runs for the entire dataset, say ten runs, then after that, a set of tuples is chosen for the cleaning stage given the deterministic query answers of the ten sampled worlds. By following this approach, we will aggressively eliminate tuples that are either are in the answer sets of all the worlds and tuples that are not in any of the answer sets from the cleaning process. Afterward, we incrementally call the Monte-Carlo simulation progressively for tuples of interest (i.e., high chance of appearing in the answer set after more samples are generated).

Furthermore, applying query-driven approaches in a progressive setting to answer probabilistic queries is an organic extension of TQEL and TQELX. That is, a user submits a query, and tuples are cleaned in iterations of epochs. At the end of each epoch, a probabilistic answer is given based on the current state of the data, where the main goal is to refine such given answers in future epochs. In this scenario, the user controls when the processing stops whenever he is satisfied with the returned result. This approach would hide some of the drawbacks of TQELX and TQELX in terms of total execution time, given that the processing

and cleaning will stop on demand and the best answer at that time will be returned. Such work can be integrated within EnrichDB [41] since they share the same cleaning-analysis model.

In the current implementation of PIVM, only single-block queries are supported, given that it does not contain self-joins or a distinct clause. One exciting direction would be to extend the support for nested queries and queries with a distinct clause. This will allow for supporting complex probabilistic queries and push for more experimentations in the area of query-driven cleaning of complex probabilistic queries because of the incremental query processing improvements. Another direction would be to optimize the costs associated with running and storing the Monte-Carlo sample results as they take most of the query processing time. Moreover, improvements concerning the sampled runs storage methods would be highly beneficial in reducing the amount of allocated disk space.

Bibliography

[1] Incremental view maintenance development for postgresql. https://github.com/sraoss/pgsql-ivm.

[2] Data never sleeps 8.0. `https://www.domo.com/learn/infographic/data-never-sleeps-8`, 2020.

[3] Apache lucene. `https://lucene.apache.org/`, 2021.

[4] Wikipedia. `https://www.wikipedia.org`, 2021.

[5] Wikipedia:database download. `https://en.wikipedia.org/wiki/Wikipedia:Database_download`, 2021.

[6] C. C. Aggarwal and S. Y. Philip. A survey of uncertain data algorithms and applications. *IEEE Transactions on knowledge and data engineering*, 21(5):609–623, 2008.

[7] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. *Proc. of VLDB 2006 (demonstration description)*, 2006.

[8] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *arXiv preprint arXiv:1207.0137*, 2012.

[9] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *Proceedings of the VLDB Endowment*, 7(11):999–1010, 2014.

[10] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. Query-driven approach to entity resolution. *Proceedings of the VLDB Endowment*, 6(14):1846–1857, 2013.

[11] H. Altwaijry, S. Mehrotra, and D. V. Kalashnikov. Query: A framework for integrating entity resolution with query processing. *Proceedings of the VLDB Endowment*, 9(3):120–131, 2015.

[12] L. Antova, C. Koch, and D. Olteanu. Maybms: Managing incomplete information with probabilistic world-set decompositions. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 1479–1480. IEEE, 2007.

[13] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. Dbpedia: A nucleus for a web of open data. In *The semantic web*, pages 722–735. Springer, 2007.

[14] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, 2000.

[15] D. Barbará, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on knowledge and data engineering*, 4(5):487–502, 1992.

[16] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *The VLDB Journal*, 18(1):255–276, 2009.

[17] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. Mystiq: a system for finding more answers by using probabilities. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 891–893, 2005.

[18] D. Burdick, P. Deshpande, T. Jayram, R. Ramakrishnan, and S. Vaithyanathan. Olap over uncertain and imprecise data. In *VLDB*, volume 5, pages 970–981. Citeseer, 2005.

[19] J. Cambronero, J. K. Feser, M. J. Smith, and S. Madden. Query optimization for dynamic imputation. *Proceedings of the VLDB Endowment*, 10(11):1310–1321, 2017.

[20] G. Casella and R. L. Berger. *Statistical inference*. Cengage Learning, 2021.

[21] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 313–324, 2003.

[22] R. Cheng, J. Chen, and X. Xie. Cleaning uncertain data with quality guarantees. *Proceedings of the VLDB Endowment*, 1(1):722–735, 2008.

[23] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 551–562, 2003.

[24] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 876–887, 2004.

[25] X. Chu, I. F. Ilyas, and P. Papotti. Holistic data cleaning: Putting violations into context. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 458–469. IEEE, 2013.

[26] X. Chu, I. F. Ilyas, P. Papotti, and Y. Ye. Ruleminer: Data quality rules discovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1222–1225. IEEE, 2014.

[27] E. Ciceri, P. Fraternali, D. Martinenghi, and M. Tagliasacchi. Crowdsourcing for top-k query processing over uncertain data. *IEEE Transactions on Knowledge and Data Engineering*, 28(1):41–53, 2015.

[28] G. Cormode, F. Li, and K. Yi. Semantics of ranking queries for probabilistic data and expected ranks. In *2009 IEEE 25th International Conference on Data Engineering*, pages 305–316. IEEE, 2009.

[29] M. Dallachiesa, A. Ebaid, A. Eldawy, A. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 541–552, 2013.

[30] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4):523–544, 2007.

[31] G. Demartini, D. E. Difallah, and P. Cudré-Mauroux. Zencrowd: leveraging probabilistic reasoning and crowdsourcing techniques for large-scale entity linking. In *Proceedings of the 21st international conference on World Wide Web*, pages 469–478, 2012.

[32] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 588–599, 2004.

[33] M. Fabian, K. Gjergji, W. Gerhard, et al. Yago: A core of semantic knowledge unifying wordnet and wikipedia. In *16th International World Wide Web Conference, WWW*, pages 697–706, 2007.

[34] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM Journal on discrete mathematics*, 17(1):134–160, 2003.

[35] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183–1210, 1969.

[36] P. Ferragina and U. Scaiella. Tagme: on-the-fly annotation of short text fragments (by wikipedia entities). In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 1625–1628, 2010.

[37] J. R. Finkel, T. Grenager, and C. D. Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 363–370, 2005.

[38] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems (TOIS)*, 15(1):32–66, 1997.

[39] A. Gattani, D. S. Lamba, N. Garera, M. Tiwari, X. Chai, S. Das, S. Subramaniam, A. Rajaraman, V. Harinarayan, and A. Doan. Entity extraction, linking, classification, and tagging for social media: a wikipedia-based approach. *Proceedings of the VLDB Endowment*, 6(11):1126–1137, 2013.

[40] E. Gelenbe and G. Hebrail. A probability model of uncertainty in data bases. In *1986 IEEE Second International Conference on Data Engineering*, pages 328–333. IEEE, 1986.

[41] D. Ghosh. A case for enrichment in data management systems. preprint on webpage at `https://github.com/DB-repo/enrichdb/blob/master/A%20Case%20for%20Enrichment%20in%20Data%20Management%20Systems_FullVersion.pdf`.

[42] S. Giannakopoulou, M. Karpathiotakis, and A. Ailamaki. Cleaning denial constraint violations through relaxation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 805–815, 2020.

[43] S. Giannakopoulou, M. Karpathiotakis, and A. Ailamaki. Query-driven repair of functional dependency violations. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1886–1889. IEEE, 2020.

[44] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 328–339, 1995.

[45] S. Guo, M.-W. Chang, and E. Kiciman. To link or not to link? a study on end-to-end tweet entity linking. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1020–1030, 2013.

[46] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993.

[47] X. Han and J. Zhao. Nlpr_kbp in tac 2009 kbp track: A two-stage method to entity linking. In *TAC*. Citeseer, 2009.

[48] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. *ACM Sigmod Record*, 24(2):127–138, 1995.

[49] M. Hua, J. Pei, W. Zhang, and X. Lin. Ranking queries on uncertain data: a probabilistic threshold approach. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 673–686, 2008.

[50] J. Huang, L. Antova, C. Koch, and D. Olteanu. Maybms: a probabilistic database management system. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1071–1074, 2009.

[51] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):1–58, 2008.

[52] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 687–700, 2008.

[53] M. A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of the American Statistical Association*, 84(406):414–420, 1989.

[54] T. Jayram, S. Kale, and E. Vee. Efficient aggregation algorithms for probabilistic data. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 346–355. Citeseer, 2007.

[55] H. Ji, R. Grishman, H. T. Dang, K. Griffitt, and J. Ellis. Overview of the tac 2010 knowledge base population track. In *Third text analysis conference (TAC 2010)*, volume 3, pages 3–3, 2010.

[56] R. M. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 56–64. IEEE Computer Society, 1983.

[57] R. M. Karp, M. Luby, and N. Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of algorithms*, 10(3):429–448, 1989.

[58] C. Koch. Approximating predicates and expressive queries on probabilistic databases. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 99–108, 2008.

[59] C. Koch and D. Olteanu. Conditioning probabilistic databases. *arXiv preprint arXiv:0803.2212*, 2008.

[60] H. Köpcke and E. Rahm. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69(2):197–210, 2010.

[61] L. V. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: A flexible probabilistic database system. *ACM Transactions on Database Systems (TODS)*, 22(3):419–469, 1997.

[62] R. Li, S. Wang, and K. C.-C. Chang. Towards social data platform: Automatic topic-focused monitor for twitter stream. *Proceedings of the VLDB Endowment*, 6(14):1966–1977, 2013.

[63] Y. Li, H. Wang, N. M. Kou, Z. Gong, et al. Crowdsourced top-k queries by pairwise preference judgments with confidence and budget control. *The VLDB Journal*, pages 1–25, 2020.

[64] X. Lin, J. Xu, H. Hu, and Z. Fan. Reducing uncertainty of probabilistic top-k ranking via pairwise crowdsourcing. *IEEE Transactions on Knowledge and Data Engineering*, 29(10):2290–2303, 2017.

[65] S. Lohr. For big-data scientists,'janitor work'is key hurdle to insights. *New York Times*, 17:B4, 2014.

[66] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178, 2000.

[67] R. Mihalcea and A. Csomai. Wikify! linking documents to encyclopedic knowledge. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 233–242, 2007.

[68] L. Mo, R. Cheng, X. Li, D. W. Cheung, and X. S. Yang. Cleaning uncertain data for top-k queries. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 134–145. IEEE, 2013.

[69] S. Monahan, J. Lehmann, T. Nyberg, J. Plymale, and A. Jung. Cross-lingual cross-document coreference with entity linking. In *TAC*, 2011.

[70] M. Nikolic, M. Elseidy, and C. Koch. Linview: incremental view maintenance for complex analytical queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 253–264, 2014.

[71] R. Paige. Applications of finite differencing to database integrity control and query/-transaction optimization. In *Advances in data base theory*, pages 171–209. Springer, 1984.

[72] F. Piccinno and P. Ferragina. From tagme to wat: a new entity annotator. In *Proceedings of the first international workshop on Entity recognition & disambiguation*, pages 55–62, 2014.

[73] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[74] D. Rao, P. McNamee, and M. Dredze. Entity linking: Finding extracted entities in a knowledge base. In *Multi-source, multilingual information extraction and summarization*, pages 93–115. Springer, 2013.

[75] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 886–895. IEEE, 2007.

[76] R. Ross, V. Subrahmanian, and J. Grant. Aggregate operators in probabilistic databases. *Journal of the ACM (JACM)*, 52(1):54–101, 2005.

[77] N. Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *ACM Transactions on Database Systems (TODS)*, 16(3):535–563, 1991.

[78] M. Sadri, S. Mehrotra, and Y. Yu. Online adaptive topic focused tweet acquisition. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 2353–2358, 2016.

[79] C. E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[80] W. Shen, J. Wang, and J. Han. Entity linking with a knowledge base: Issues, techniques, and solutions. *IEEE Transactions on Knowledge and Data Engineering*, 27(2):443–460, 2014.

[81] W. Shen, J. Wang, P. Luo, and M. Wang. Linden: linking named entities with knowledge base via semantic knowledge. In *Proceedings of the 21st international conference on World Wide Web*, pages 449–458, 2012.

[82] W. Shen, J. Wang, P. Luo, and M. Wang. Linking named entities in tweets with knowledge base via user interest modeling. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 68–76, 2013.

[83] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 896–905. IEEE, 2007.

[84] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Probabilistic top-k and ranking-aggregate queries. *ACM Transactions on Database Systems (TODS)*, 33(3):1–54, 2008.

[85] D. Suciu, D. Olteanu, C. Ré, and C. Koch. Probabilistic databases. *Synthesis lectures on data management*, 3(2):1–180, 2011.

[86] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Intermittent query processing. *Proc. VLDB Endow.*, 12(11):1427–1441, 2019.

[87] V. Verroios and H. Garcia-Molina. Top-k entity resolution with adaptive locality-sensitive hashing. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1718–1721. IEEE, 2019.

[88] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical report, Stanford InfoLab, 2004.

[89] W. Zhang, C. L. Tan, Y. C. Sim, and J. Su. Nus-i2r: Learning a combined system for entity linking. In *TAC*, 2010.

[90] X. Zhang and J. Chomicki. Semantics and evaluation of top-k queries in probabilistic databases. *Distributed and parallel databases*, 26(1):67–126, 2009.