UNIVERSITY OF CALIFORNIA,
IRVINE

Supporting Progressive Query Processing and Scalable Data Enrichment for Real time Analytic Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Dhrubajyoti Ghosh

Dissertation Committee:
Professor Sharad Mehrotra, Chair
Professor Michael J. Carey
Professor Chen Li
Professor Nalini Venkatasubramanian

2021

© 2021 Dhrubajyoti Ghosh

DEDICATION

To my beloved family.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vi**

ACKNOWLEDGMENTS **vii**

VITA **viii**

ABSTRACT OF THE DISSERTATION **xi**

1 Introduction **1**

2 Related Works **8**

3 EnrichDB Data Model **14**
3.1 Data Model 14
3.2 Query Model 20
3.2.1 Query Language 20
3.2.2 Determinization-Based Query Semantics 21

4 EnrichDB System Implementation **26**
4.1 Enriching Data During Query Processing 29
4.1.1 Query Processing in EQ_{LC} 31
4.1.2 Query Processing in EQ_{TC} 34
4.1.3 Comparison between EQ_{LC} and EQ_{TC} 36
4.2 Progressive Query Processing 38
4.2.1 Progressive Queries 39
4.2.2 State Management 42
4.2.3 Joint Enrichment and Query Execution 44
4.3 Experimental Evaluations 56
4.3.1 Experimental Setup 56
4.3.2 Experimental Results 58

5 Optimizing Enrichment with Progressive Query Processing **65**
5.1 Objective 65
5.2 Overview of the Algorithm 70

5.3 Candidate Tuple Set Selection 71
5.3.1 Choosing Thresholds for Each Relation. 72
5.4 Benefit Estimation 74
5.4.1 Selection Queries 75
5.4.2 Generalizing to Other Queries 83
5.5 Enrichment Plan Selection 85
5.6 Experimental Evaluation 85
5.6.1 Experimental Setup 86
5.6.2 Experimental Results 88

6 Use Cases of the System **96**
6.1 Social Media Analysis 96
6.2 Multi Media Analysis 99
6.3 IoT Application of Localization using WiFi data 101

7 Conclusions and Future Work **104**
7.1 Conclusions 104
7.2 Future Work 105

Bibliography **107**

Appendices **115**
A Twitter Analysis Application on Spark 115
B Proof of Theorem 5.2 122

LIST OF FIGURES

Page

4.1 Query processing in EQ_{LC} approach. 27
4.2 Query processing in EQ_{TC} approach. 28
4.3 The join graph of R_1, `probe queries` in EQ_{LC}, and the rewritten query in EQ_{TC} for the query of Figure 4.4(a). 28
4.4 Original query, query tree, and the rewritten query tree in both EQ_{LC} and EQ_{TC}. 30
4.5 Progressive Query Processing Strategy. 41
4.6 Updated probe query for R_1. 47
4.7 The incremental query used by IVM in EQ_{TC}^{P}. 50
4.8 Times in enrichment server and DBMS (Left bars=EQ_{LC}, Right bars=EQ_{TC}). 61
4.9 Progressive score achieved by queries in EQ_{TC} and EQ_{LC}. 61
4.10 Progressiveness achieved in EQ_{LC} and EQ_{TC} for (a) Q2, (b) Q3, and (c) Q4. 62
4.11 Comparing different plan generation strategies in EQ_{TC}: (a) Q2, (b) Q3, (c) Q4 (left to right). 63

5.1 Progressiveness Achieved. The dotted line shows a possible incremental strategy producing query answers using server-side cursors. 86
5.2 Performance results of different plan generation strategies in EnrichDB. . . 89
5.3 Cost vs. Quality (a) synthetic(lhs) and (b) real (rhs) functions. 93
5.4 Comparing plan generation on synthetic data. (a) linear (b) logarithmic (c) exponential correlations. 93
5.5 Time overhead. 94
5.6 Effect of epoch sizes (a) TTR 90% (lhs) (b) overhead (rhs). 94

6.1 EnrichDB web interface for submitting query. 97
6.2 Interface for visualizing progressively improving query results on image dataset. 97
6.3 EnrichDB web interface for submitting query on images. 100
6.4 Interface for visualizing progressively improving query results on image dataset. 101

LIST OF TABLES

Page

3.1 TweetData table in EnrichDB. Derived attributes are topic and sentiment, and the values are their determinized representations. 15
3.2 State output for derived attributes. 15
3.3 PresenceData table in EnrichDB. Derived attribute is the location attribute and the values are the determinized representations. 15
3.4 A part of DecisionTable. 19
3.5 TweetData table with 2 tuples and 1 derived attribute. 19
3.6 Possible World 1 (prob = 0.351). 19
3.7 PW2 (prob = 0.189). 19
3.8 PW3 (prob = 0.299). 19
3.9 PW4 (prob = 0.161). 19
3.10 Query Result of $\sigma_{topic=soc}$(TweetData) following PW semantics. 19
3.11 TweetData table after determinization. 19
3.12 Query Result of $\sigma_{topic=soc}$(TweetData) on determinized representation. 19
3.13 Truth table for evaluating complex conditions. 22

4.1 TweetData table where topic and sentiment are the derived attributes. . . 29
4.2 TweetDataState table (created for TweetData table). 43
4.3 PlanSpaceTable. 44
4.4 PlanTable. 44
4.5 Datasets used in experiments. 56
4.6 Query templates used. 57
4.7 Exp 1. Number of enrichments in EQ_{LC} and EQ_{TC}. 58
4.8 Number of enrichments saved in EQ_{TC} compared to EQ_{LC} with varying selectivity of precise condition (*i.e.*, TweetTime between (t_1, t_2)) in query Q3. . 59
4.9 Latency of queries in EQ_{LC} and EQ_{TC} approaches (in seconds). 60

5.1 TweetData table where topic and sentiment are the derived attributes. . . 67
5.2 DecisionTable. 74
5.3 Datasets used in experiments. 86
5.4 Queries used. 88
5.5 Time without progressive query execution (in minutes). 91
5.6 Max. storage overhead. 92
5.7 Storage cost of state. 92
5.8 Impact of optimization. 92

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor professor Sharad Mehrotra for advising me during my journey as a Ph.D. student. Prof. Sharad has taught me to identify new research problems, solve the problems in an innovative way, and express my thoughts in the form of writing research papers. I am glad that I had the opportunity to work with him and I am very grateful to him for his guidance.

I would like to thank all the members of my doctoral committee: Professor Michael Carey, Professor Chen Li, and Professor Nalini Venkatasubramanian, for their time for reading my thesis. I am very thankful to them for serving on my committee and providing me with valuable feedback for my thesis.

I am grateful to all the collaborators with whom I have worked during my study in UCI. I am very thankful to Peeyush Gupta, Shantanu Sharma, and Abdulrahman Alsaudi for working together in the EnrichDB project. The technical collaborations and the discussions during the project were very valuable to me and allowed me to improve my research and development skills. Peeyush's technical intellect on the system development were amazing and I learned a lot of development experiences while working with him on the EnrichDB project. Shantanu's research skills helped me to develop and implement new research ideas in this project. I learned a lot from him about writing research papers and describing and solving complex problems related to the project.

I would like to thank all the professors who were part of the TIPPERS project and with whom I have collaborated: Professor Sharad Mehrotra, Professor Nalini Venkatasubramanian, Professor Alfred Kobsa, and Professor Xi He. I would like to thank all the team members with whom I have worked and received valuable feedback in this project: Primal Pappachan, Roberto Yus, Peeyush Gupta, Shantanu Sharma, Abdulrahman Alsaudi, Sameera Ghayyur, Eun-Jeong Shin, Nisha Panwar, Yiming Lin, Guoxi Wang, and Georgios Bouloukakis. I would like to extend my gratitude to other collaborators of the TIPPERS project: Professor Chris Davison, Mamadou Diallo, Christopher Graves, Michael August, the team-members of NIWC and the participants of Trident Warrior 2019 and 2020 demonstrations.

I would like to thank the PIs and team members of the Synergy project: Professor Amit K. Roy-Chowdhury and Rameshwar Panda, for their collaboration with me. Their feedback and computer vision code on the project were very valuable to me.

I would like to acknowledge the research agencies that have funded my research work at UCI. The materials in this thesis are based on the research sponsored by DARPA under Agreement No. FA8750-16-2-0021. Furthermore, this thesis was partially supported by NSF Grants No. 1527536, 1545071, 2032525, and 2008993.

VITA

Dhrubajyoti Ghosh

EDUCATION

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

Master of Science in Mathematics and Computing **2013**
Indian Institute of Technology, Kharagpur *Kharagpur, India*

Bachelor of Science in Mathematics and Computing **2012**
Indian Institute of Technology, Kharagpur *Kharagpur, India*

RESEARCH EXPERIENCE

Graduate Research Assistant **2017–2021**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2014–2017**
University of California, Irvine *Irvine, California*

PUBLICATIONS

Prism: Private Verifiable Set Computation over Multi-Owner Outsourced Databases **2021**

Yin Li, Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, Nisha Panwar, and Shantanu Sharma.

ACM SIGMOD

A Case for Enrichment in Data Management Systems **2021**

Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, and Shantanu Sharma.

Under Submission

Implementing Enrichment in Data-Management Systems **2021**

Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, and Shantanu Sharma.

Under Submission

Optimizing Progressive Quality Improvement for Query Time Data Enrichment **2021**

Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, and Shantanu Sharma.

Under Submission

PANDA: Partitioned data security on outsourced sensitive and non-sensitive data **2020**

Sharad Mehrotra, Shantanu Sharma, Jeffrey D. Ullman, Dhrubajyoti Ghosh, Peeyush Gupta, and Anurag Mishra.

ACM Transactions on Management Information Systems

Transitioning from testbeds to ships: an experience study in deploying the TIPPERS Internet of Things platform to the US Navy **2020**

Dave Archer, Michael A August, Georgios Bouloukakis, Christopher Davison, Mamadou H Diallo, Dhrubajyoti Ghosh, Christopher T Graves, Michael Hay, Xi He, Peeter Laud, Steve Lu, Ashwin Machanavajjhala, Sharad Mehrotra, Gerome Miklau, Alisa Pankova, Shantanu Sharma, Nalini Venkatasubramanian, Guoxi Wang, and Roberto Yus.

The Journal of Defense Modeling and Simulation

IoT Expunge: Implementing Verifiable Retention of IoT Data **2020**

Nisha Panwar, Shantanu Sharma, Peeyush Gupta, Dhrubajyoti Ghosh, Sharad Mehrotra, and Nalini Venkatasubramanian.

ACM CODASPY

SCARF: a scalable data management framework for context-aware applications in smart environments **2019**
Eun-Jeong Shin, Dhrubajyoti Ghosh, Sharad Mehrotra, and Nalini Venkatasubramanian.

EAI MOBIQUITOUS

Towards Accuracy Aware Minimally Invasive Monitoring (MiM) **2019**
Sameera Ghayyur, Dhrubajyoti Ghosh, Xi He, and Sharad Mehrotra.

ACM CCS TPDP Workshop

SOFTWARE

TIPPERS middleware system `https://tippersweb.ics.uci.edu/`
A middleware system implemented in Java to provide semantic abstraction and scalable data management solutions for developing IoT applications. The system is very easy to use by providing several REST APIs for the end-users to develop application.

EnrichDB database system `https://github.com/DB-repo/enrichdb`
A database system that supports scalable enrichment of data using complex machine learning functions. The system allows enrichment of data in various stages of data processing pipeline, supports progressive query processing to hide latency of complex functions, and implements state management to eliminate redundant execution of complex enrichment functions.

ABSTRACT OF THE DISSERTATION

Supporting Progressive Query Processing and Scalable Data Enrichment for Real time Analytic Applications

By

Dhrubajyoti Ghosh

Doctor of Philosophy in Computer Science

University of California, Irvine, 2021

Professor Sharad Mehrotra, Chair

In this thesis, we propose EnrichDB, a new DBMS technology designed for emerging domains (*e.g.*, social media analytics and sensor-driven smart spaces) that require incoming data to be enriched using expensive machine learning/signal processing functions prior to its usage. To support online processing, today, such enrichment is performed outside of the database as a static data processing workflow prior to its ingestion to the database. Such a strategy could result in a significant delay from the time when data arrives and when it is enriched and ingested into the DBMS, especially when the enrichment complexity is high. Also, enrichment at ingestion could result in wastage of resources if applications do not use/require all data to be enriched. EnrichDB's design represents a significant departure from the above, where we explore seamless integration of data enrichment all through the data processing pipeline — at ingestion, triggered based on events in the background, and progressively during query processing. The cornerstone of EnrichDB is a powerful enrichment data and query model that encapsulates enrichment as an operator inside a DBMS enabling it to co-optimize enrichment with query processing. The first chapter of the thesis describes this data model of the system.

In the second chapter of the thesis, we describe two implementations of the EnrichDB

system. In the first implementation, we have taken a middleware-based approach where the database management system is treated as a black-box and enrichment is performed in a separate server, called the *enrichment server*. This is a simpler and portable implementation, since a user can utilize any DBMS as the storage system for storing the underlying data objects with a small code change. In the second implementation, we describe a layered implementation on top of PostgreSQL, where we have used the extensibility features of PostgreSQL to perform enrichment efficiently closer to the data where it resides. We used user-defined functions, stored procedures, indexes, and incremental materialized views to perform data enrichment and produce query results efficiently.

In the third part of my thesis, we present algorithms implemented in above systems that can optimize enrichment with query processing. We have chosen the quality metric of F_1-measure as the quality of set-based queries. Improving query results gradually by choosing online samples were explored in Online Approximate Query Processing systems. However, such systems did not consider data enrichment to improve the quality of query results and the underlying data was considered to be static. The goal of data enrichment in the EnrichDB system is to enrich objects in a way that can produce good quality results as soon as possible, formalized as progressive score. Experimental results on real world datasets and queries show that the algorithm performs significantly better than the traditional sampling based approaches.

Chapter 1

Introduction

Organizations, today, have access to potentially limitless data sources in the form of web data repositories, social media posts, and continuously generated sensory data [12]. Such data is often low-level/raw and needs to be enriched to be useful for analysis. Functions used to enrich data (referred to as *enrichment functions* in this thesis) could consist of (a combination of) custom-compiled code, declarative queries, and/or expensive machine learning (ML) techniques. Examples include mechanisms for sentiment analysis [103] over social media posts, named entity extraction [21] in text, and sensor interpretation such as face detection and face recognition [61, 83] from images, sensor data fusion [87], and data cleaning tasks such as missing value imputation in relational data [64].

Data enrichment could be performed as a periodic offline process prior to loading the data into the database for analysis. Such an offline approach to preparing data is indeed common practice in enterprise data processing pipelines. For instance, in the enterprise data warehouses setting, data collected from diverse transactional databases running core organizational operations is periodically loaded into an enterprise-wide warehouse through a extract-transform-load (ETL) process. The offline process, however, adds significant delay

between the time data arrives (or is created) to when it is available for analysis. This limits the ability of organizations to analyze data in (near) real-time as it arrives. We illustrate the challenge of executing expensive enrichment functions on large datasets based our experiences in developing analytic applications that have served as a motivation behind developing EnrichDB.

Motivating Example.

Consider an instrumented campus with 100 buildings and 50 WiFi access points and 10 surveillance cameras per building, resulting in a total of 5000 WiFi access points and 1000 surveillance cameras. Data produced from such devices (*e.g.*, events generated at the access points when mobile devices carried by individuals connect, images captured by cameras when people enter/exit spaces) can be used to localize individuals within building at different levels of granularity. Recent work by Lin et. al. [75] has shown that while coarse-level region localization based on Wi-Fi connectivity can be achieved relatively efficiently, *e.g.*, ≈10ms/event, on a server with 64 core Intel Xeon CPU E5-4640 with 2.40GHz clock speed and 128GB memory. ML techniques applied to the same data can help localize individuals at room-level granularity, at the expense of ≈200ms/event on same server. Likewise, video data with face detection & recognition (≈1s/image) could result in even higher accuracy. Now, consider analytical applications that perform tasks such as computing average occupancy level of a given location (meeting room), average time spent by people in the cafeteria during lunch, and number of people a person came in contact with during a time interval. Supporting such applications, require localizing people to appropriate rooms/regions based on WiFi and/or camera data.

At rates of ≈1K WiFi events/sec and ≈100 images/sec, analyzing six months of data requires enriching 15 billion WiFi events and 1.5 billion images. Complete enrichment of data would require several years on a suitably large server, which is not feasible. Further, while coarse WiFi-based localization could be performed at data ingestion, fine-grained localization (using

either WiFi or camera data) cannot be fully performed in its entirety (it would take ≈3 minutes of computation for one second of WiFi events generated, requiring a large server farm). Such computations would be wasted if the analyst is satisfied by coarse localization for most parts of the building and decides to restrict attention to only a small subset of data (*e.g.*, data from atrium) for finer analysis. EnrichDB is designed to support real-time analysis in such scenarios when it is expensive/infeasible to enrich data in its entirety. ■

In the enterprise context, such a challenge has been well recognized leading to the emergence of Hybrid Transaction/Analytical Processing (HTAP) technology over the past decade [94, 80] that exploits modern hardware and columnar storage to support both transactional and analytical capabilities in the same system. HTAP systems enable data analysis to be performed in real-time as it arrives. Motivated by a similar requirement to reduce latency between when data arrives/is created and when it is made available for analysis in data processing pipelines that require data to be enriched prior to use, this thesis explores effective ways to integrate data enrichment into query processing in databases.

Before we argue for the need for the integrated processing of data enrichment and query processing, we note that one possibility to overcome the latency challenge would be to enrich the data as it arrives into the system at the time of ingestion. Systems (*e.g.*, Spark Streaming [118] often used for scalable ingestion) are capable of executing enrichment functions on newly arriving data prior to its storage in a DBMS. Recently, in the context of an open-source big data management system AsterixDB [3, 23], Wang et. al. [113] has explored ways to optimize enrichment during ingestion by exploiting parallelism and batching of such operations.

Enriching data at arrival, however, exhibits several limitations. The approach could lead to significant avoidable overhead of redundantly enriching data if the analyst ends up using only (a small) portion of the data. In some situations, when workloads are predictable, one could potentially limit enrichment to only data that is expected to be used. However, accurate

prediction of the workload, especially when analyst may execute adhoc queries, can be very difficult, as argued in several prior works [94, 49]. Furthermore, enriching data at ingestion is only feasible if enrichment functions are computationally not resource intensive. If such functions require running one or many relatively complex machine learning models (*e.g.*, Multi-layer Perceptron, Random Forest) to interpret data, executing them during ingestion would create a bottleneck. Complex ML models (*e.g.*, for functions such as object detection, face recognition, or other such extraction and data interpretation tasks), often take 100s of millisecond per data item on modern processors. If we are to enrich all data as it arrives, it would limit the system to ingestion rates of only 10s of event per second.[1]

In this thesis, we develop a query time approach of enrichment which provides us with several benefits: (*i*) enrichment is performed only in the context of a query resulting in no wasted enrichment, (*ii*) data is ingested without any enrichment leading to higher ingestion performance, and (*iii*) scope for exploiting query semantics to reduce/eliminate expensive enrichment tasks that do not influence the query results. Integrating enrichment with query processing raises several key challenges: First, given a query, how do we determine which data items/objects need to be enriched to answer the query correctly. Second, where should such enrichment be performed — closer to the data at the database server possibly using stored procedures and user defined functions (UDFs), or outside the database in the application server. Both these options offer different advantages/disadvantages in terms of data movement, restricting resource wastage for enrichment, and the scope for parallelism in executing data enrichment. Finally, while ability of the system to enrich data at query time reduces the amount of work at ingestion, it potentially causes an increased query execution time for the individual queries (since data required to answer the query has to be enriched during query execution). Such an increased query latency may result in unacceptable query performance.

[1] The challenge of running complex ML functions on data as it arrives, was discussed extensively in the curated session on ML in databases in SIGMOD [84], leading to an observation that often organizations are forced to use simpler functions that can be performed at ingestion, even though such a choice results in poorer quality.

Motivated by the above-mentioned limitations, in this thesis, we make the following contributions:

- Integrating enrichment into query languages and semantics by extending the relational data model to support enrichment functions.
- Incorporating enrichment into databases by implementation of a new system using PostgreSQL DBMS.
- Overcoming latency in queries due to query-time enrichment by supporting progressive approach of query processing.

The above contributions are implemented inside a new database management system, called EnrichDB. Below, we describe the high-level design criteria of EnrichDB.

Semantic Abstraction. EnrichDB supports a declarative interface to specify and link enrichment functions with higher-level observations that the functions generate from raw data. Users may associate one or more such functions that differ in terms of quality (*e.g.*, uncertainty in the enriched value) and cost (*e.g.*, execution time of the function). In EnrichDB, developers do not need to deal with raw data directly — applications can be fully written based on higher-level semantic observations derived from raw data using enrichment functions. If the higher-level observation has not been derived through the enrichment process prior to the execution of the application, such enrichment would be automatically performed as part of query processing.

Transparency of Enrichment. In EnrichDB, the data enrichment process is transparent to application programmers, who view the data at a semantically higher level of abstraction. Programmers do not need to be concerned about what data needs to be enriched, using which functions, and at what stage of data processing. EnrichDB maintains the state of

enrichment of objects, performs enrichment automatically based on the current state and needs of applications, and updates the state appropriately.

Optimization of Enrichment. To mitigate the ingestion latency due to complex functions, EnrichDB allows enrichment all through the data processing pipeline. Enrichment can be performed at ingestion, triggered based on events, or during query processing. EnrichDB makes sure that enrichment of objects is performed optimally. During query time enrichment, EnrichDB exploits the query optimizer to prune away enrichment of objects that do not influence the query results. The developer does not have to write any separate code to prune such enrichment of objects. Furthermore, EnrichDB allows enrichment of data closer to where the data resides resulting in a low data movement.

Progressive Computation. EnrichDB produces answers progressively, as it executes enrichment functions as a part of query processing. A progressive query answering (motivated by Approximate Query Processing systems [58] — provided progressive query answering for aggregation queries) technique produces an initial set of answers that are improved over time as data is further enriched.

The overview of the thesis chapters are as follows: In **Chapter §2**, we explore previous works related to EnrichDB. In **Chapter §3**, we describe the data model of EnrichDB. In **Chapter §4**, we develop and discuss the advantages/disadvantages of two distinct solutions to support joint query processing and data enrichment: (*i*) a *loosely coupled* approach (referred to as $\boldsymbol{EQ_{LC}}$ approach) that performs data enrichment at an external server from the database server, denoted as *enrichment server*, and (*ii*) a *tightly coupled* approach (referred to as EQ_{TC} approach) that exploits the mechanisms for pushing code to the database server using stored procedures and UDFs to co-process enrichment and queries at the database. In **Chapter §5**, we describe a mechanism for optimizing enrichment with progressive query processing. We present the algorithm and the implementation details of the algorithm, using

the EQ_{LC} and EQ_{TC} approaches of implementation. In **Chapter §6**, we present several use-cases of EnrichDB in the domains of social-media analysis, multimedia analysis, and *Internet-of-things* (IoT) applications.

Chapter 2

Related Works

Data enrichment as described in Chapter 1, can be performed at data ingestion, or at query processing time or adaptively both at ingestion and at query processing. Recently, several industrial systems such as Apache Kafka [4], Storm [110], Spark [118], and Flink [39] have explored ways of data enrichment at scale during ingestion. *E.g.*, Apache Kafka is a centralized data pipeline that aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds. Kafka along with the stream processing systems of Apache Spark or Apache Flink are often used to develop scalable ingestion framework. The data feeds are enriched by the execution of Spark User Defined Functions (UDFs) before ingesting it into a data warehouse system. Apache AsterixDB [3], an open-source big data management system, supports efficient mechanisms for ingesting high velocity data using data feeds [55]. In AsterixDB, the data feeds support enrichment of data at ingestion, by allowing users to attach UDFs to the data ingestion pipeline. Wang et. al. [113] explored ways to make such enrichment of data feeds faster by exploiting parallelism and batching of such operations. While such systems can be used when simple functions (*e.g.*, that do not add significant latency) are executed at ingestion, they do not scale to situations when data analysis requires execution of a suite of computationally expensive enrichment functions.

In certain application scenarios, data enrichment can be performed periodically as an offline step. For instance, if data analysis does not need to be performed on the real time data as it arrives, one can choose to enrich such data offline and add the enriched data to the database when the insertion workload is not high. Analytical applications developed on top of data warehouse systems [2, 6, 14, 8] are often built for such situations. For such applications, the creator of a data warehouse, develops a well-defined ETL process, where the data is transformed using transformation functions such as data cleaning functions and functions for computing semantically higher level information.

A similar architecture is also possible for sensor-based applications where data from sensors are collected and stored and then loaded into a data warehouse system with appropriate enrichment at the non-peak time when the workload on the data warehouse is low. Such an approach suffers from several limitations as follows: (*i*) analysis on data as it arrives may not be available and (*ii*) ad-hoc queries cannot be answered unless they are enriched at the data load time.

The need for performing analytics on the data as it arrives, has been well recognized over the past decade and has been the driving force for the design of HTAP systems [94, 80]. Such systems have innovated extensively at the storage layer of the DBMS. To support real-time analytics for ad-hoc queries, these systems support columnar storage architectures, often maintained in the main memory. Data stored in columnar format are highly compressed as compared to the data stored in row format, since attribute values of same data types are stored together and can be compressed effectively [18]. Furthermore, the columnar stores exploit features of modern processors such as Single Instructions/Multiple Data (SIMD) operations and vector processing to ensure that the HTAP system can compute aggregates very efficiently in near-real time. This eliminates the need for pre-computing aggregates as would have been required if analysis was to be performed over row stores [18]. The careful design choices made by the HTAP systems, allowed them to support fast transactional

workloads with insertions and yet support near real-time analytics on the fast incoming data at high velocity. While HTAP systems today have focused on the storage layer and optimization of queries based on the new storage models, they have not considered supporting expensive enrichment and transformation that might be necessary to prepare the data for analysis, as is the case for the domains considered in this thesis. For instance, our example of fine-grained localization cannot be done with the HTAP systems available today. We believe that the technique we study in this thesis to scale enrichment can contribute to the next generation of HTAP systems that in addition to supporting real aggregation/OLAP, also support complex data enrichment.

Our work of query time data enrichment is also relevant to the data lake architectures. In data lake architecture, the incoming data from various data sources are stored into a large data lake system. The analysis/enrichment of the data is performed at the read/query time. Data lake architectures [5, 15, 16] are often realized using a data warehouse system through an extract-load-transform (ELT) pipeline, instead of the ETL process. But, to the best of our knowledge, none of the ELT based systems have considered real-time data enrichment.

Our approach of hiding increased query-time latency (due to performing enrichment at query time), is similar in motivation to the approximate query processing systems (AQP). Such systems are studied in the context of large data volume where queries need to be answered quickly at the expense of approximation of query results. The system provides an approximate answer with an error bound measured using a confidence interval. Such approximate results allow an analyst to get a quick insight of the data for future analysis that need to be performed on the data.

AQP systems can be categorized in two ways: (*i*) *Offline AQP* system [19, 92, 48, 41] and (*ii*) *Online AQP* system [58, 119, 43, 116]. Offline AQP systems maintain a set of pre-computed samples consisting of various column sets and size of samples. Given a query with an error or latency bound, the system finds out and executes the query on the samples that

can produce results with the given latency or error bound for the query. In contrast, online approximate processing systems compute such samples at the query execution time. Such systems are more closely related to our work on EnrichDB. Although such systems try to hide query latency due to large data volume, the challenge addressed in this thesis is about reducing query latency due to the requirement of performing complex enrichment on the data.

Our work in this thesis is also relevant to the works on **query time data cleaning** [54, 29, 99]. Several authors have shown that query context can be used to eliminate cleaning of object blocks (residing on disk) that cannot satisfy the query predicates. Authors in [29] have utilized an approximate statistic for the objects residing in each disk blocks. Such statistics are used during the query processing to dictate the cleaning tasks. However, such works considered cleaning functions to be deterministic and there were only one cleaning function for each cleaning tasks as compared to the enrichment functions considered by us in this thesis. Furthermore, such frameworks do not consider a progressive approach of query processing when the cost of cleaning functions are high.

Several system architectures were introduced to implement application logic on top of an existing database management systems. Some authors have employed a tightly coupled approach where application code (enrichment function code) is pushed down to the database servers using user-defined-functions (UDFs) and stored procedures [13, 59, 111]. Furthermore, several authors took a loosely coupled approach where the application logic is implemented inside a separate middleware system outside of the DBMS [92, 19, 34].

Progressive Approaches of Data Processing. Progressive and query-driven approaches have been studied in the past in several domains: entity resolution [79, 89, 24, 25, 28, 29], crowd sourced data processing tasks of filtering, sorting, ranking, clustering [91, 90, 30], online schema matching [81, 96], probabilistic databases [45, 98, 101]. Progressive approach of visualization to support interactive analysis of large-scale data was proposed by Jia et.

al. in [66]. Progressive approach of query processing along with enrichment have never been studied in the past.

Enrichment in Smart Application Domains. Enrichment of lower level data to semantically higher level data is required in various application domains of IoT and sensor-based applications. Examples of such application domains are smart water infrastructure [56, 112], smart city infrastructure [121, 60], smart healthcare [109], and advanced fire detection mechanisms [32]. Furthermore, enrichment of data is necessary for alerting/notification-based applications that sends messages based on contextual conditions on sensor data [33, 102]. EnrichDB can benefit such application domains by ensuring that enrichment is performed only in the context of queries. Furthermore, the progressive query processing mechanism of EnrichDB can provide real-time responses to queries.

Systems for Supporting ML using Databases. EnrichDB is related to, but complementary to the past research efforts of Apache MADlib [59], Bismarck [51], RioT [120], SystemML [53], and SimSQL [38]. Such systems are designed to learn ML models inside or on top of database systems. While such systems provide mechanisms to learn ML models (which can be used as enrichment functions) from data stored in databases, they do not focus on optimally enriching the data. EnrichDB focuses on optimally enriching the data that achieves progressive improvement in the quality of query answers. Furthermore, some recent systems address the problem of model selection and optimal query plans for ML inference queries [69, 44]. However, such systems do not support progressive execution of queries with enrichment as introduced by EnrichDB.

Expensive Predicate Optimization. These works optimize queries with expensive predicates [40, 57, 31, 68, 78]. In [100], authors surveyed several strategies of optimizing data flows that contain complex UDFs both in the context of relational databases and Map/Reduce-style data processing frameworks. A subset of these problems (more relevant to us) address *optimization of multi-version predicates* [73, 77, 67], where multiple versions

of predicates are created for an expensive predicate present in the queries. Less expensive predicates are evaluated first to reduce the number of tuples that are used to evaluate the expensive predicates. However, in these works, the expensive predicates were static and deterministic task. In contrast, EnrichDB can perform enrichment using a general class of deterministic and non-deterministic functions.

Delta Computation of Queries. Several authors in the past have proposed mechanisms for efficiently updating materialized views in databases: DBToaster [71], LINVIEW [86], and PostgreSQL IVM [9]. Furthermore, several intermittent query processing systems [106, 108, 107] were proposed in the past to support efficient ways of maintaining/computing delta answers to queries, while minimizing resource consumption of CPU and memory. Efficient delta computation has also been studied in *data flow systems* [85, 22, 82] that provide efficient state management techniques to speed up delta processing. While EnrichDB exploits delta computation of queries (it is built using IVM and could benefit from work such as [106, 108, 107]), its focus is on incremental enrichment, which is complementary to the above methods.

Chapter 3

EnrichDB Data Model

The cornerstone of EnrichDB is *Enrichment Data and Query Model* (EDQM) that integrates enrichment as a first-class operator in the database system. This data model is designed based on the design criterias of semantic abstraction, transparency of enrichment, scope for optimizing enrichment and progressive computation as described in Chapter §1.

3.1 Data Model

In EDQM, the data is modeled using relations where a relation can have two types of attributes: (*i*) ***derived*** attributes that require enrichment and (*ii*) ***fixed*** attributes that do not require enrichment. Each derived attribute is optionally associated with a domain size. If the domain size is not specified, then that attribute is considered to have a value from a continuous range. The command for specifying a relation in EnrichDB is shown below.

```
CREATE TABLE TweetData(tid char(8), userid
   char(20), Tweet text, feature float[], topic
   int derived:40, sentiment int derived:3,
```

tid	UserID	Tweet	feature	loc.	TweetTime	topic	sentiment
t_1	John	Uploading pics on Facebook.	[0.2, ..., 0.4]	US	16:08	soc	pos.
t_2	Mark	Listening...	[0.5, ..., 0.3]	US	16:48	ent.	NULL
t_3	Richard	Iran's ...	[0.6, ..., 0.4]	UK	11:48	pol.	neg.

Table 3.1: `TweetData` table in EnrichDB. Derived attributes are `topic` and `sentiment`, and the values are their determinized representations.

tid	topic	sentiment
t_1	soc:0.54, ent:0.46	pos:0.52, neu:0.48
t_2	ent: 0.65, art: 0.35	pos:0.5,neu:0.5
t_3	pol:0.8, art: 0.2	neu:0.3,neg:0.7

Table 3.2: State output for derived attributes.

tid	semantic_id	observation_time	location
t_1	125	10:04	room-1
t_2	127	10:06	room-4
t_3	128	10:07	room-8

Table 3.3: `PresenceData` table in EnrichDB. Derived attribute is the `location` attribute and the values are the determinized representations.

```
TweetTime timestamp, location text);
```

An example of the table created for tweet data is shown in Table 3.1. The `feature` attribute contains term frequency-inverse document frequency (tf-idf) vector [74] pre-extracted from tweets. The value of a derived attribute is determined using one or more ***enrichment functions*** associated with it. Another example of a table created in the context of an IoT application of location monitoring is shown in Table 3.3. In the majority part of the thesis, we are going to use the tweet analysis application as the driving use-case of EnrichDB. In Chapter 6, we describe several other use cases of EnrichDB.

Enrichment functions. EDQM supports a general class of enrichment functions (frequently used in real-world). The input to an enrichment function is a tuple and the output is either a single value, multiple values, or a probability distribution, as described below.

We categorize enrichment functions based on the output cardinality: (*i*) *single-valued*: outputting a single value, *e.g.*, a binary classifier [105], (*ii*) *multi-valued*: outputting a set of values as a prediction, *e.g.*, top-k classifiers [72], (*iii*) *probabilistic*: outputting a probability distribution over the possible values of a label, *e.g.*, probabilistic classifiers [42]. Also, enrichment functions can be categorized based on the output domain size: (*i*) *categorical*: predicts outputs from a finite set of possible values, *e.g.*, sentiment of positive/negative, and (*ii*) *continuous*: outputs a real number, *e.g.*, a weather of 72.8°F.

An enrichment function is associated with two parameters: (*i*) *cost*: the average execution time/tuple, and (*ii*) *quality*: a metric of the goodness (*i.e.*, accuracy) of enrichment function in determining the correct value of the derived attribute.

Training of enrichment functions. EDQM supports training procedures for enrichment functions where a user needs to specify an input table storing the training data. Below, we show an example of learning a machine learning model of Multi-Layer Perceptron (MLP) using a training procedure of `model_train`. This model is trained using data stored in `tweets_train` table and the name of the model is `sentiment_mlp`. It uses attribute values of `feature` as input to the model and outputs prediction for `sentiment` attribute. The model-specific parameters are passed as a string in `model_params`.

```
SELECT db.model_train('tweets_train', 'sentiment_mlp',
   'multi_layered_perceptron','sentiment',
   'feature', 'classification', model_params);
```

The *cost* and *quality* of enrichment functions can either be specified by the user or can be determined automatically by using several methods, *e.g.*, train/test split, k-fold cross-validation, and leave-one-out cross-validation, during the training phase. The set of enrichment functions for a derived attribute $\mathcal{A}_i$ are called ***function-family*** of $\mathcal{A}_i$. (We use ***calligraphic font*** for ***derived attributes***.) Outputs of enrichment functions in

a function-family are combined using a ***combiner function***. One can use weighted-average, majority-voting, or stacking-based [115] combiner functions in EnrichDB. As an example, shown below, the function-family of `sentiment` attribute is created using an `assign_enrichment_function` function. It uses `mlp_classifier` function with *cost* of 0.1 second/tuple and *quality* of 0.8 (measured in AUC).

```
SELECT db.assign_enrichment_function
  ('sentiment_fmly','TweetData', 'sentiment',
  ['mlp_classifier','sentiment_mlp',0.1,0.8]);
```

State of a Derived Attribute. Enrichment state or state of a derived attribute $\mathcal{A}_i$ in tuple t_k (denoted by $state(t_k.\mathcal{A}_i)$) is the information about enrichment functions that have been executed on t_k to derive $\mathcal{A}_i$. The state has two components: ***state-bitmap*** that stores the list of enrichment functions already executed on $t_k.\mathcal{A}_i$; and ***state-output*** that stores the output of executed enrichment functions on $t_k.\mathcal{A}_i$. *E.g.*, consider that there are four enrichment functions f_1, f_2, f_3, f_4, and out of which f_1, f_3 have been executed on $t_k.\mathcal{A}_i$. Also, assume that the domain of $\mathcal{A}_i$ contains three possible values: d_1, d_2, and d_3. Thus, the state-bitmap for $t_k.\mathcal{A}_i$ contains $\langle 1010 \rangle$, *i.e.*, only first and third functions are executed and the state-output of $t_k.\mathcal{A}_i$ contains: $\langle [0.7, 0.3, 0], [], [0.8, 0.1, 0.1], [] \rangle$, *i.e.*, the output of the first and third enrichment functions (remaining arrays are left empty).

The state-output stores a list of probability distributions when the enrichment functions are probabilistic classifiers, *e.g.*, $\langle [0.7, 0.3, 0], [], [0.8, 0.1, 0.1], [] \rangle$. For single-valued classifiers, clustering functions, and regression functions the state-output attribute stores the actual output of the function instead of a probability distribution, *e.g.*, $\langle [72.4], [], [76.8], [] \rangle$.

State of Tuples and Relations. The notion of state of derived attributes is generalized to the state of tuples and relations in a straightforward way. The state of a tuple t_k is the concatenation of the state of all derived attributes of t_k, *e.g.*, the state of a tuple t_k

of a relation R with three derived attributes $\mathcal{A}_p$, $\mathcal{A}_q$, and $\mathcal{A}_r$ is denoted by $state(t_k) = \langle state(t_k.\mathcal{A}_p)||state(t_k.\mathcal{A}_q)||state(t_k.\mathcal{A}_r)\rangle$.

Relative Ordering of Enrichment Functions. In EDQM, the user can specify (or can be learned by EnrichDB using a training dataset) the relative order in which enrichment functions need to be executed. This order is specified using the state of tuples for each derived attribute. Such relative ordering is important for ensembling different enrichment functions to be executed on a tuple. This ordering is stored in a table called `DecisionTable` (see Table 3.4).

This table, for each derived attribute of a relation, stores a map that — given the current state of a tuple with respect to the attribute — specifies the next function that should be executed to further enrich the attribute, as well as (optionally) the measure of ***benefit*** that is expected to result.

In Table 3.4, each row stores a map containing (state bitmap, entropy range) as keys and corresponding (next best function, benefit) pair as values. That is, given the state, the map specifies the function resulting in the maximum reduction of uncertainty per unit cost along with expected uncertainty reduction as a measure of benefit. For a probabilistic distribution over a set of domain values $D_1, \ldots, D_n$ with probabilities $p_1, \ldots, p_n$, the uncertainty of a tuple is often measured using entropy as follows: $\sum_{i=1}^{n} -p_i \log p_i$.

Consider the tuple t_1 of `TweetData` table (see Table 3.1) and assume that the sentiment state bitmap of t_1 is $[1, 0, 0]$ and the sentiment state output of t_1 is $[[0.94, 0.06, 0], [0, 0, 0], [0, 0, 0]]$. The entropy of t_1 is $(-0.94 \times \log(0.94) - 0.06 \times \log(0.06)) = 0.32$. From first row of Table 3.4, since entropy of t_1 is in the range (0.25-0.5), the decision table specifies that the next best function to execute on t_1 is f_3 and its benefit as 0.2.

Relation	Attribute	Map
TweetData	sentiment	$\langle 1,0,0\rangle$, [0-0.25) : $\langle f_2, 0.1\rangle$, $\langle 1,0,0\rangle$, (0.25-0.5) : $\langle f_3, 0.2\rangle$, $\langle 1,0,0\rangle$, (0.5-0.75) : $\langle f_2, 0.16\rangle$, $\langle 1,0,0\rangle$, (0.75-1] : $\langle f_2, 0.22\rangle$
TweetData	topic	$\langle 0,1,0\rangle$, [0-0.2) : $\langle f_4, 0.08\rangle$, $\langle 0,1,0\rangle$, (0.2-0.4) : $\langle f_6, 0.11\rangle$, $\langle 0,1,0\rangle$, (0.4-0.6) : $\langle f_4, 0.18\rangle$, $\langle 0,1,0\rangle$, (0.6-0.8) : $\langle f_6, 0.24\rangle$, $\langle 0,1,0\rangle$, (0.8-1]: $\langle f_6, 0.26\rangle$

Table 3.4: A part of DecisionTable.

tid	topic
t_1	soc:0.54, ent:0.46
t_2	art: 0.65, soc: 0.35

Table 3.5: TweetData table with 2 tuples and 1 derived attribute.

tid	topic
t_1	soc
t_2	art

Table 3.6: Possible World 1 (prob = 0.351).

tid	topic
t_1	soc
t_2	soc

Table 3.7: PW2 (prob = 0.189).

tid	topic
t_1	ent
t_2	art

Table 3.8: PW3 (prob = 0.299).

tid	topic
t_1	ent
t_2	soc

Table 3.9: PW4 (prob = 0.161).

tid	probability
t_1	0.54
t_2	0.35

Table 3.10: Query Result of $\sigma_{topic=soc}$(TweetData) following PW semantics.

tid	topic
t_1	soc
t_2	art

Table 3.11: TweetData table after determinization.

tid	topic
t_1	soc

Table 3.12: Query Result of $\sigma_{topic=soc}$(TweetData) on determinized representation.

3.2 Query Model

This section describes the query language of EnrichDB (§3.2.1) and the query semantics (§3.2.2) of EnrichDB.

3.2.1 Query Language

The query language of EnrichDB is an extended version of SQL. Queries in EnrichDB are associated with mandatory query semantics (which are required to deal with probabilistic values of derived attributes) and a (optional) quality parameter for the quality of the query results.

Two types of query semantics for probabilistic data have been proposed: (*i*) possible world (PW) semantics [62, 20] and (*ii*) determinization-based semantics [50]. In PW semantics, all possible worlds are generated (implicitly/explicitly) from probabilistic representation (as shown in Tables 3.6-3.9) and the query is executed in each world. The result consists of all possible tuples along with a probability value which is the summation of probability of all the worlds in which the tuple was present, as shown in Table 3.10. The determinization-based semantics converts probabilistic representation to a single or a small set of deterministic worlds as shown in Table 3.11. The query is executed in these worlds and a single deterministic answer is produced as shown in Table 3.12. The rationale of choosing one semantics over the other depends on the application scenarios. In some scenarios, an application can make good decisions by just using the most probable answers, whereas for some applications, it may require analysis of all possible answers along with their probability distribution. Due to simplicity, we have implemented the determinization-based query semantics in EnrichDB (the implementation of PW semantics is under development).

An example query in EnrichDB is shown below:

```
SELECT Tweet,location,TweetTime, topic,
sentiment FROM TweetData WHERE sentiment='pos'
AND topic='soc. media' AND TweetTime
BETWEEN('16:00','18:00') QUALITY 0.9
SEMANTICS DETERMINIZATION;
```

The `QUALITY` and `SEMANTICS` keywords specify the minimum quality requirement of query result and the semantics of query evaluation respectively. The possible value of `SEMANTICS` is either `DETERMINIZATION` or `PROBABILISTIC`.

3.2.2 Determinization-Based Query Semantics

In determinization-based query semantics, tuples of all participating relations in a query are determinized first before evaluating the query. The process of converting a probabilistic data representation, *i.e.*, the output of probabilistic enrichment functions, to a deterministic representation is referred to as the *determinization process*.

Consider a derived attribute $\mathcal{A}_i$ and a tuple t_k. The value of tuple t_k in attribute $\mathcal{A}_k$ (*i.e.*, $t_k.\mathcal{A}_i$) is determined using a *determinization function* ($DET(.)$) based on tuple's state. $DET(state(t_k.\mathcal{A}_i))$ returns a single or multiple values for $t_k.\mathcal{A}_i$ or a NULL value, representing a situation when state of the attribute does not provide enough evidence to assign any value for $t_k.\mathcal{A}_i$. Determinization concept naturally extends to a tuple and a relation. The determinized representation of a relation R is denoted by:

$$DET(R) = DET(state(t_i.\mathcal{A}_j)) \mid \forall t_i \in R, \forall \mathcal{A}_j \text{ of } R.$$

Example. Consider a relation `TweetData` with two derived attributes `topic` and `sentiment` (see Table 3.1) and assume state-output of derived attributes as shown in Table 3.2. Based

C_1	T	F	P	P	P	P	U
C_2	P	P	T	F	P	U	P
$C_1 \wedge C_2$	P	F	P	F	P	U	U
$C_1 \vee C_2$	T	P	T	P	P	P	P
NOT C_1	F	T	F	F	F	F	U

Table 3.13: Truth table for evaluating complex conditions.

on Table 3.2 and a top-1 determinization strategy, the determinized representation of `topic` and `sentiment` are shown in Table 3.1. ■

Since determinization of a tuple can result in either a set of values or NULL, evaluation logic of different conditions needs to be defined. EnrichDB extends the traditional three-valued logic used in relational operators, *i.e.*, with truth values of $\boldsymbol{T}$, $\boldsymbol{F}$, and $\boldsymbol{U}$ into a four-valued logic: **true ($\boldsymbol{T}$)**, **false ($\boldsymbol{F}$)**, **possible ($\boldsymbol{P}$)**, and **unknown ($\boldsymbol{U}$)**. Here, P represents that the condition is **possibly true** based on the current state of enrichment, whereas U (as in traditional setting) represents that the truth value is **unknown**, given the current level of enrichment. Similar to SQL, the DBMS implementing this data model does not have to return tuples that evaluate to unknown. However, the tuples evaluating to **possible** may or may not be returned. *E.g.*, the inclusion of such tuples in the answer could be based on the maximization of the quality of the query. We next discuss how we assign truth values to predicates/expressions.

Simple Predicates. Consider an expression $\boldsymbol{\mathcal{A}_i}$ `op` $\boldsymbol{a_m}$, where $\mathcal{A}_i$ is a derived attribute, `op` is an operator, and a_m is a possible value of $\mathcal{A}_i$. The operator `op` is one of the following operators: $\langle =, \neq, >, \geq, <, \leq \rangle$. If the output of $DET(state(t_k.\mathcal{A}_i))$ is NULL, then the expression evaluates to U. If $DET(state(t_k.\mathcal{A}_i))$ is a singleton set S and $x \in S$ such that x `op` a_m holds, then the expression evaluates to T; otherwise, F. If $DET(state(t_k.\mathcal{A}_i))$ is a multi-valued set (say S) and $\exists x \in S$ such that x `op` a_m holds, then it is possible that t_k satisfies the expression, and hence, it evaluates to P. However, if $\nexists x \in S$ for which x `op` a_m

holds, then the expression evaluates to F.

Consider an expression $\boldsymbol{\mathcal{A}_i}$ `op` $\boldsymbol{\mathcal{A}_j}$, where $\mathcal{A}_i$ and $\mathcal{A}_j$ are two derived attributes of (possibly different) relations and `op` is a comparison operator. If $DET(state(t_k.\mathcal{A}_i))$ or $DET(state(t_l.\mathcal{A}_j))$ is NULL, then the condition evaluates to U. If both $DET(state(t_k.\mathcal{A}_i))$ and $DET(state(t_l.\mathcal{A}_j))$ are singleton sets and for elements $x \in DET(state(t_k.\mathcal{A}_i))$ and $y \in DET(state(t_l.\mathcal{A}_j))$, x `op` y holds, then the condition evaluates to T; otherwise, F. In case one or both of $DET(state(t_k.\mathcal{A}_i))$ and $DET(state(t_l.\mathcal{A}_j))$ are multi-valued sets and $\exists x \in DET(state(t_k.\mathcal{A}_i))$ and $\exists y \in DET(state(t_l.\mathcal{A}_j))$, such that x `op` y holds, then the condition evaluates to P; otherwise, F.

Complex Predicates. Complex predicates are formed using multiple comparison conditions connected by Boolean operators (AND ($\wedge$), OR ($\vee$), and NOT ($\neg$)). Table 3.13 shows the truth table for such logical operators. This table only shows entries when one of the two expressions evaluates to P. When both expressions evaluate to either T, F, or U, we follow the same evaluation logic as in standard SQL.

Aggregation. Aggregation functions on fixed attributes are evaluated as in SQL, while, on a derived attribute, return a range of values $[l, u]$, denoting the lower and upper bounds of aggregated value. An aggregation function applied to all T tuples of a set produces the value of lower bound l, while applied to all T and P tuples together produces the upper bound u. The aggregation functions that are supported are as follows: *count*, *sum*, *min*, *max*, and *avg*.

Example 3.2.1. Consider a query that counts number of tweets with (`sentiment` = `positive`) from `TweetData` of Table 3.1, and assume that the table contains 250 tuples of which 100 tuples evaluate to T, while 20 of the remaining 150 tuples evaluate to P. Hence, the condition evaluation logic returns a range of $[100, 120]$. Likewise, group-by aggregation results in one such range per group.

Top-k Aggregation. EnrichDB first evaluates aggregation functions for each group-by key (as described above), and their outputs are ranked using a ranking function. The query result consists of a set of group-by keys with the top-k ranks. The purpose of the ranking function is to return a minimal answer set A, such that the real top-k groups are guaranteed to be part of A. EnrichDB sorts the group-by keys based on the lower bounds in a descending order and selects the first n (where $n \geq k$) group-by keys as the minimal answer set A such that the upper bound of $(n+1)$-th key is lower than the lower bound of the n-th key. This ensures that the $(n+1)$-th group-by key cannot be part of the top-k answer set.

Consider a query that returns top-2 topics with the highest tweet counts from Table 3.1. Suppose after applying $count()$, the topics had following bounds: social media: $[100, 150]$, entertainment: $[110, 120]$, politics: $[100, 115]$, and economy: $[80, 95]$. The answer of the query will be {social media, entertainment, politics} that guarantees that the actual top-2 groups (*i.e.*, social media and entertainment) are part of the answer. The group economy will be excluded from the answer, since the upper bound of this group is 95, which is lower than the lower bounds of groups inside the answers.

Queries Supported. We support any single-block *select-project-join-aggregation* query in this data model. The selection and join conditions on derived attributes are evaluated according to the four-valued logic described above of type $(\mathcal{A}_i \texttt{ op } a_m)$ and of type $(\mathcal{A}_i \texttt{ op } \mathcal{A}_j)$, respectively. The aggregation conditions and the top-k aggregation conditions are evaluated after evaluation of the rest of the query as shown in Example 3.2.1.

Extending to More General Class of Queries. Extending four-valued logic for more general class of queries such as `Union`, `Intersection`, `Set-difference`, `Division`, and `Nested` queries on probabilistic data are complex. For Union queries, we can use the semantics of UNION ALL as was used by the UADB [50] system. However, it is difficult to quantify the quality of query result for such queries.

Query Semantics. Now, based on the definition of determinization function and the predicate evaluation logic as described above, we define the query semantics as follows:

$$q(R_1, R_2, \ldots, R_n) = q'(DET(R_1), DET(R_2), \ldots, DET(R_n))$$

Here, $q(R_1, R_2, \ldots R_n)$ is a query on relations $R_1, \ldots, R_n$, $DET(R_i)$ is the determinized representation of the ith relation. Query q is rewritten as q' to be executed on the determinized representations of relations using the four valued logic as described above.

Chapter 4

EnrichDB System Implementation

In this chapter, we implement a system that can support joint query processing and data enrichment. We have developed this system to address the challenges of efficient enrichment using database systems as highlighted in §1. The chapter first develops and discusses the advantages/disadvantages of two distinct solutions to support joint query processing and data enrichment: (*i*) a ***loosely coupled*** approach (referred to as $\boldsymbol{EQ_{LC}}$ approach) that performs data enrichment at an external server from the database server, denoted as *enrichment server*, and (*ii*) a ***tightly coupled*** approach (referred to as EQ_{TC} approach) that exploits the mechanisms for pushing code to the database server using stored procedures and UDFs to co-process enrichment and queries at the database. Both strategies attempt to minimize the number of avoidable calls to the enrichment functions.

For a single query, EQ_{TC} is much faster than EQ_{LC} since EQ_{TC} performs enrichment closer to the data, hence there is no network latency, and EQ_{TC} uses query context to eliminate redundant enrichment of objects during the query time. However, the EQ_{TC} approach puts higher load to the database server as compared to EQ_{LC} approach. EQ_{LC} is more amenable for scale-out, *i.e.*, more machines can be easily added as the enrichment servers to execute

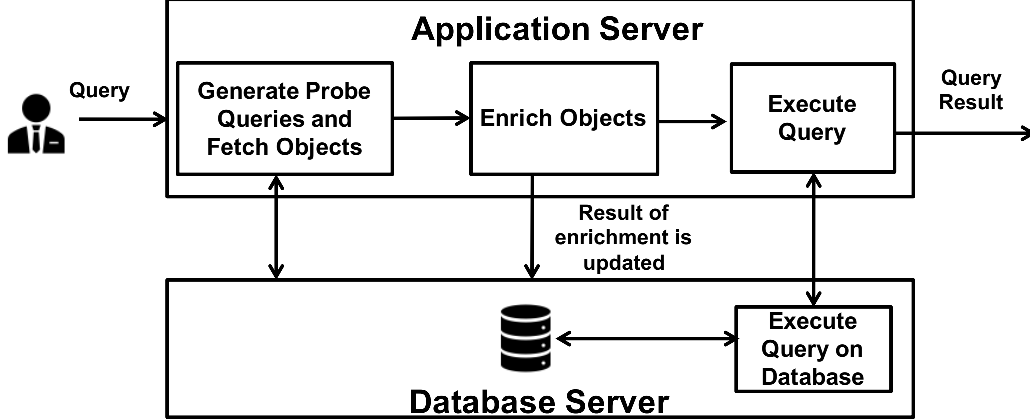


Figure 4.1: Query processing in EQ_{LC} approach.

enrichment in parallel. The EQ_{TC} approach can also be scaled out using parallel DBMSs such as AsterixDB [23] and Greenplum database [7], however, linear scaling of complex queries such as join queries with UDF is difficult.

This chapter, then, addresses the challenge of increased query latency, which arises due to enrichment functions executed during query processing, and develop an iterative/progressive approach to answer queries. Particularly, the approach exploits the fact that very often, enrichment of data can be performed using multiple enrichment functions, each of which may vary in their execution cost and quality of the output. Based on the query, simpler (*i.e.*, less costly), though possibly less accurate, models may be applied first, which can help to generate an approximate answer. More accurate and expensive enrichment functions can be used to refine such answers. once more accurate (though more expensive) models have executed. Such an approach to hiding query latency for complex queries has been extensively explored in approximate query processing [58, 97], where it is difficult to answer query in real-time due to the size of data involved. We adopt the similar methodology of answering queries approximately based on partially enriched data and progressively improving them as data is enriched further.

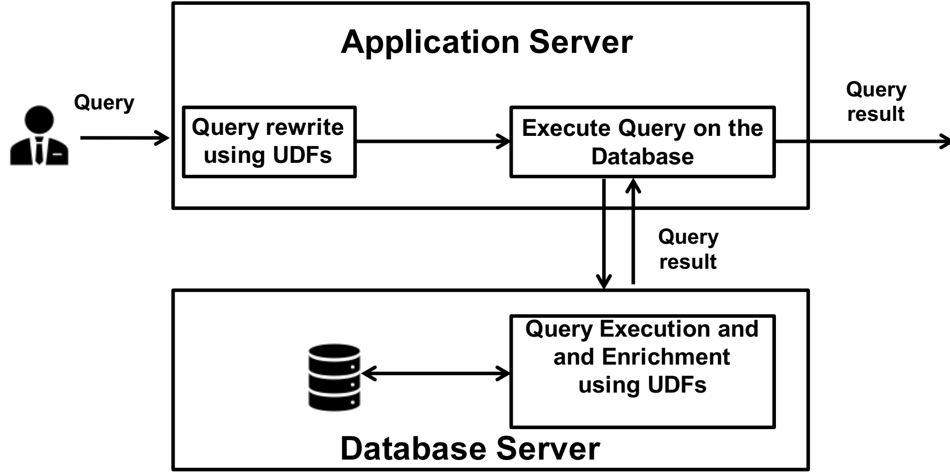
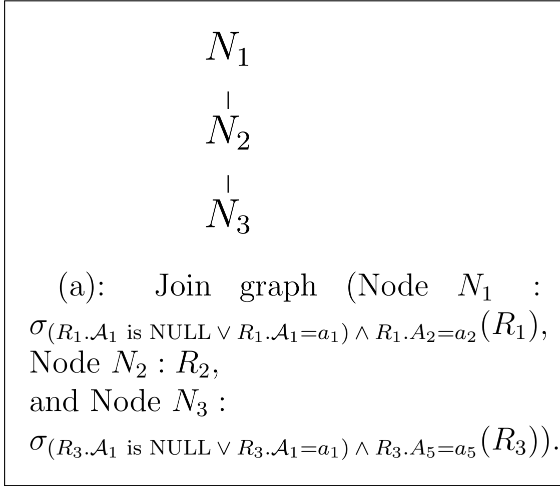
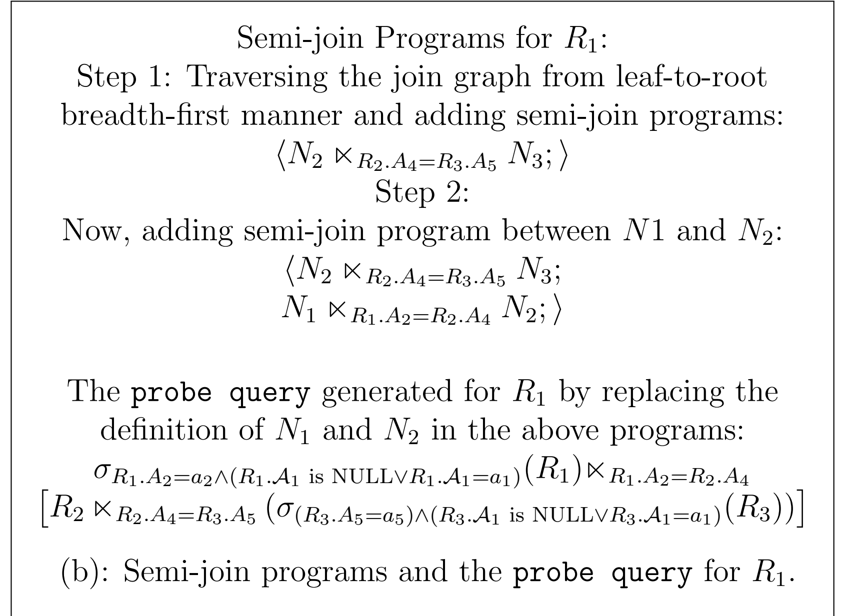


Figure 4.2: Query processing in EQ_{TC} approach.



(a): Join graph (Node N_1 : $\sigma_{(R_1.\mathcal{A}_1 \text{ is NULL} \vee R_1.\mathcal{A}_1=a_1) \wedge R_1.A_2=a_2}(R_1)$, Node N_2 : R_2, and Node N_3 : $\sigma_{(R_3.\mathcal{A}_1 \text{ is NULL} \vee R_3.\mathcal{A}_1=a_1) \wedge R_3.A_5=a_5}(R_3)$).



(b): Semi-join programs and the `probe query` for R_1.

SELECT * FROM R_1, R_2, R_3 WHERE $\omega_\sigma(R_1.\mathcal{A}_1 = a_1)$ AND $R_1.A_2 = a_2$ AND $\omega_\bowtie(R_1.\mathcal{A}_1 = R_2.\mathcal{A}_3)$ AND $R_1.A_2 = R_2.A_4$ AND $\omega_\sigma(R_3.\mathcal{A}_1 = a_1)$ AND $R_3.A_5 = a_5$ AND $\omega_\bowtie(R_3.\mathcal{A}_3 = R_1.\mathcal{A}_3)$ AND $R_3.A_5 = R_2.A_4$

(c): Rewritten query in EQ_{TC}, where ω_σ refers to rewrite logic of selections and $\omega_\bowtie$ refers to the rewrite logic of joins as described in §4.1.2.

Figure 4.3: The join graph of R_1, `probe queries` in EQ_{LC}, and the rewritten query in EQ_{TC} for the query of Figure 4.4(a).

tid	UserID	Tweet	feature	location	TweetTime	topic	sentiment
t_1	John	Uploading pics on Facebook.	[0.2, ..., 0.4]	US	16:08	social media	positive
t_2	Mark	Feeling great and listening to music.	[0.5, ..., 0.3]	US	16:48	ent.	NULL
t_3	Richard	Sad about current pandemic.	[0.6, ..., 0.4]	UK	11:48	NULL	NULL

Table 4.1: `TweetData` table where `topic` and `sentiment` are the derived attributes.

4.1 Enriching Data During Query Processing

Without loss of generality, we assume that all relations contain an `id` attribute that uniquely identifies the tuples present in them. *E.g.*, in a relation storing tweets, a derived attribute can be the tweet's `sentiment`, which is derived by executing sentiment analysis functions on the tweet. Likewise, in a relation storing images, the identity of person in images can be the derived attribute, which is derived by face recognition techniques for identifying person in an image.

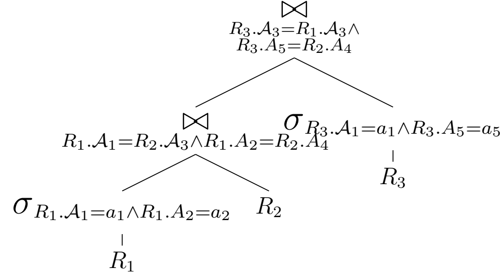
In general, several functions, called *enrichment function*, could be used either independently or in combination to determine the value of a derived attribute. If an enrichment function is executed on a tuple, the derived attribute will take the value of the function output. If the enrichment function is not executed so far, then the attribute value will be NULL. For example, in Table 5.1, the values of two derived attributes `topic` and `sentiment` are NULL in t_3, since they are not enriched yet.

In this section, we assume that only one enrichment function is associated with each derived attribute and outputs a single value from the domain of the derived attribute. This simplification makes the ***state*** representation of derived attributes easier, where the state corresponds to the set of enrichment functions that have been executed in the past and their outputs. §4.2 presents a more general model, where several enrichment functions could be associated with the derived attributes and the enrichment functions can output multiple

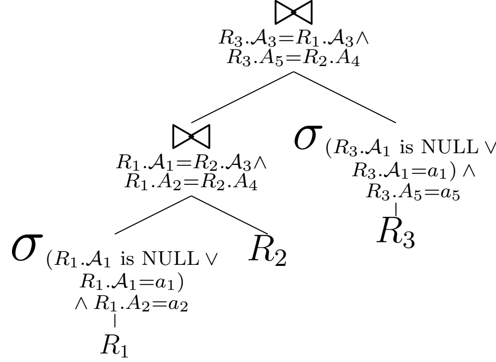
values or a probability distribution.

SELECT * FROM R_1, R_2, R_3 WHERE $R_1.\mathcal{A}_1 = a_1$ AND $R_1.A_2 = a_2$ AND $R_1.\mathcal{A}_1 = R_2.\mathcal{A}_3$ AND $R_1.A_2 = R_2.A_4$ AND $R_3.\mathcal{A}_1 = a_1$ AND $R_3.A_5 = a_5$ AND $R_3.\mathcal{A}_3 = R_1.\mathcal{A}_3$ AND $R_3.A_5 = R_2.A_4$

(a): Original query.



(b): Original query tree generated from the query in Figure 4.3(a).



(c): Rewritten query tree from the query tree of Figure 4.3(b).

Figure 4.4: Original query, query tree, and the rewritten query tree in both EQ_{LC} and EQ_{TC}.

We consider two ways of implementing enrichment during query processing: (i) loosely-coupled implementation (EQ_{LC}) that uses a separate enrichment engine (outside of databases) and (ii) tightly-coupled implementation (EQ_{TC}) that uses database programming features such as UDF to incorporate enrichment functions. Below, we discuss both such implementation approaches in details and illustrate using a query, as shown in Figure 4.3(a).

4.1.1 Query Processing in EQ_{LC}

EQ_{LC} executes enrichment at the enrichment server. Given a query q, EQ_{LC} generates a set of `probe queries`, denoted by $pq(R_i)$, one for each relation R_i in q. `Probe queries` are executed at the DBMS to retrieve a set of tuples that need to be enriched to execute q. These retrieved tuples by `probe queries` are enriched in the enrichment server, and the corresponding modified (enriched) values are updated at the DBMS. Finally, query q is executed at the DBMS.

The key step of EQ_{LC} is to generate `probe queries` that identify a "minimal" subset of tuples (as small a subset as possible) for each $R_i \in q$ that need to be enriched to execute q. For each relation R_i (whose derived attribute is part of q), EQ_{LC} exploits the following three strategies to identify such a minimal subset:

- *Exploiting Prior Work:* $pq(R_i)$ filters out all tuples of R_i that have been enriched earlier (*e.g.*, as part of prior queries), and hence, their derived attribute values in the database are not NULL. Such tuples do not need to be enriched again to execute q.

- *Exploiting Selection Conditions on Fixed Attributes:* $pq(R_i)$ filters all tuples of R_i that do not satisfy selection conditions over fixed attributes of R_i. *E.g.*, for the query of Figure 4.3(a), to identify the tuples of R_1 that require enrichment, $pq(R_1)$ retrieves only those tuples of R_1 that satisfy the condition $R_1.A_2 = a_2$.

- *Exploiting Join Conditions on Fixed Attributes:* $pq(R_i)$ filters out all tuples of R_i that would not join with any tuples in R_j, if there exists a join condition between R_i and R_j in q based on fixed attributes. Such tuples of R_i would not influence the final answer of q, and hence, do not need to be enriched. *E.g.*, for the query of Figure 4.3(a), tuples of relation R_1 that do not match with any tuples of R_3 based on the join condition of $R_3.A_1 = R_1.A_1$ do not need to be enriched.

Probe Query Generation Steps

The steps for generating `probe queries` (based on above three strategies) are as follows:

[**Step 1**]: ***Query Tree Generation*:** An input query q is first converted into a corresponding query tree, in which, selection conditions are pushed down as much as possible. The conditions present in selection and join nodes are converted into a conjunctive normal form (CNF), *i.e.*, ($C = C_1 \wedge C_2 \wedge \ldots \wedge C_z$). Each condition $C_i \in C$ is characterized as either a *fixed condition* (*i.e.*, a condition containing only fixed attributes) or a *derived condition* (*i.e.*, a condition containing only derived or both fixed and derived attributes). For example, Figure 4.3(b) shows the query tree generated from the query of Figure 4.3(a). In a CNF condition: ($R_1.\mathcal{A}_1 = a_1 \wedge R_1.A_2 = a_2$), the condition ($R_1.A_2 = a_2$) is a fixed condition while ($R_1.\mathcal{A}_1 = a_1$) is derived.

[**Step 2**]: ***Rewrite of Selection Condition ($\sigma_C(R)$)*:** Given a CNF condition C in a selection node, for each derived condition $C_i \in C$ over derived attribute(s) $\mathcal{A}_1, \ldots, \mathcal{A}_n$, this step finds only those tuples for which there exists an attribute A_i, $i \in [1, \ldots, n]$ that has not been enriched before. This filtering is achieved by replacing C_i by $\left[(\bigvee_{i=1}^{n} \mathcal{A}_i = \text{NULL}) \vee C_i\right]$. The fixed conditions are kept identical. *E.g.*, for the CNF expression of ($R_1.\mathcal{A}_1 = a_1 \wedge R_1.A_2 = a_2$) in Figure 4.3(b), using this step, it is rewritten as: (($R_1.\mathcal{A}_1$ is NULL $\vee R_1.\mathcal{A}_1 = a_1) \wedge R_1.A_2 = a_2$) in Figure 4.3(c). Note that only the first condition is modified as it is a derived condition, while the second condition is kept the same as it is fixed.

[**Step 3**]: ***Generating Join Graph***: This step and the next step 4 are performed to exploit the join conditions on fixed attributes in a query to filter out tuples of R_i that do not require enrichment. Given a query tree with selection conditions modified as in Step 2, a *join-graph* is generated from the tree. The purpose of the join graph is to find out for a relation R_i in the query: which join conditions (on fixed attribute) with other relations can

be utilized to reduce the number of tuples of R_i that require enrichment.

In join graph, the nodes correspond to *reduced* relations, *i.e.*, relations with the selection conditions applied on them. If there exists a join condition between the two relations in the original query, an edge between two nodes is present and shows the join conditions between two relations expressed in CNF form.

Next, from each edge of the join graph, all the derived join conditions are removed. If after removing all derived conditions of a join node, the final condition becomes empty (*i.e.*, all the conjuncts were on derived attributes), then that edge is deleted from the graph, *i.e.*, none of the join conditions between the two relations can be exploited to reduce the set of tuples that require enrichment.[1]

E.g., in Figure 4.2(a), we present a join-graph for the query tree shown in Figure 4.3(c). This graph contains three nodes: $\langle N_1, N_2, N_3 \rangle$, representing the reduced relations of $\langle R_1, R_2, R_3 \rangle$, respectively, *i.e.*, after applying the selection conditions on each relation. Here, the edge between N_1 and N_2 represents the join condition of $(R_1.A_2 = R_2.A_4)$ (after removing the join condition on $R_1.\mathcal{A}_1 = R_2.\mathcal{A}_3$) from Figure 4.3(c)).

[Step 4]: ***Semi-join-based Reduction***: Given the join graph as an input, for each node N_i in the graph, this step generates a set of semi-join programs for N_i to reduce the number of tuples of N_i that require enrichment. For N_i, semi-join programs are generated by exploiting join conditions among nodes of the graph. For a node N_i, this step starts from node N_i in the join graph and generates a spanning tree, denoted as $ST(N_i)$, that contains all nodes of the graph with minimum possible number of edges (using breadth-first traversal). From $ST(N_i)$, multiple semi-join programs are generated based on the join conditions in $ST(N_i)$.

Semi-join programs for a node N_i are generated in a bottom-up manner from $ST(N_i)$ starting

[1] If a query tree contains the operators of `union`, `set difference`, or `cross product`, then they are ignored, as such operators cannot be utilized to reduce the number of tuples in `probe queries` apart from the join conditions.

from the child nodes and reaching upto N_i. For each node encountered in the path, a semi-join program is generated. The nodes in $ST(N_i)$ are traversed in a breadth-first order from the leaf node to the root node. All the semi-join programs between the leaf node and their immediate parent nodes are created first. This step is continued until all the paths from the leaf node to the root node are consumed.

For example, $ST(N_1)$ for node N_1, is a tree with root as the node N_1, the node N_2 as the child of N_1, and the node N_3 as the child of N_2 (same as the graph shown in Figure 4.2(a)). In $ST(N_1)$ (Figure 4.2(a)), a semi-join between relations N_2 and N_3 is performed first to identify the tuples of relation R_2 (part of N_2) that may result in the join output of R_2 and R_3 (as shown in Figure 4.2(b) (top)). After this, a semi-join between R_1 and the tuples of R_2 output from previous semi-join, is performed. Using these two semi-join programs, this step can eliminate two types of tuples from R_1: (i) the tuples of R_1 that do not join with any tuple of R_2 and R_3, and (ii) the tuples of R_1 that may join with some tuples of R_2 but ultimately do not join with any tuple of R_3. This step for semi-join reduction we used is based on the seminal work on semi-join given in [35].

[Step 5]: ***Generating `probe queries`*:** Given the semi-join programs generated an input, for each relation $R_i \in q$ with predicates on derived attributes that require enrichment, this step generates a `probe query` based on the semi-join programs and the selection conditions on R_i in a straightforward manner. For example, in Figure 4.2(b) (bottom), we show the `probe query` generated for R_1, from the semi-join programs described in 4.2(b) (top) and the selection conditions added to the query tree of Figure 4.3(c) for R_1.

4.1.2 Query Processing in EQ_{TC}

In EQ_{TC}, a user query q is rewritten as query q' that enriches appropriate tuples while executing the query to generate answers to q. In EQ_{TC}, modified query q' checks whether

each derived attributes $\mathcal{A}_i \in q$ has been enriched earlier. If not, it invokes $read_u$ UDF that executes the enrichment function, updates the value of the derived attribute, and returns the value. The $read_u$ is implemented as a generic function that takes as input the name of the relation (*e.g.*, 'R_i'[2]), the name of the derived attribute (*e.g.*, '$\mathcal{A}_j$'), the tuple identity, and the identity of an enrichment function to execute.

Rewrite of Selection Condition: We rewrite each selection condition $(R.\mathcal{A}_i \ op \ a_i) \in q$ (where op is $\geq$, $>$, $=$, $\leq$, $<$, or $\neq$, and a_i is a constant value) that contains a derived attribute, by a modified selection condition denoted as $\omega_\sigma(R.\mathcal{A}_i \ op \ a_i)$, as follows:

$$\begin{aligned} & R.\mathcal{A}_i \ op \ a_i \\ \vee & \left[R.\mathcal{A}_i \text{ is NULL} \wedge read_u(\text{`}R\text{'}, \text{`}\mathcal{A}_i\text{'}, R.id, f_{\mathcal{A}_i}.id) \ op \ a_i\right] \end{aligned} \tag{4.1}$$

Here, $f_{\mathcal{A}_i}.id$ refers to the identity of an enrichment function for $\mathcal{A}_i$. In this rewritten condition, if a tuple's value in $\mathcal{A}_i$ is already enriched, then the original selection condition is evaluated (*i.e.*, $R.\mathcal{A}_i \ op \ a_i$). Otherwise, $read_u$ UDF is executed on the tuple to enrich the attribute $\mathcal{A}_i$ first, and then the selection condition is executed. Note that $read_u$ UDF is only invoked if the attribute value has not been enriched before.

Rewrite of Join Condition: We rewrite each join condition $R_p.\mathcal{A}_i \ op \ R_q.\mathcal{A}_j \in q$ that contains derived attributes $\mathcal{A}_i$ and $\mathcal{A}_j$, by a modified join condition, denoted as $\omega_\sigma(R.\mathcal{A}_i \ op \ a_i)$, based on whether one (or both) of the derived attributes in the condition have previously been enriched. If both the derived attribute values have been enriched, $(R_p.\mathcal{A}_i \ op \ R_q.\mathcal{A}_j)$ is executed with no modification. If one of the attributes (say $R_p.\mathcal{A}_i$) is not enriched, then $R_p.\mathcal{A}_i$ is replaced with a call to UDF $read_u$ on $R_p.\mathcal{A}_i$ in order to enrich the attribute as part of checking the join condition. If both of the attributes (*i.e.*, $R_p.\mathcal{A}_i$ and $R_q.\mathcal{A}_j$) are not enriched, then both attributes in the join condition are replaced by calls to the $read_u$ UDF.

[2] We use quotes to refer to the names of relations R_i and attributes $\mathcal{A}_j$

The modified join condition of $\omega_{\bowtie}(R_p.\mathcal{A}_i\ op\ R_q.\mathcal{A}_j)$ is shown below:

$$
\begin{aligned}
& R_p.\mathcal{A}_i\ op\ R_q.\mathcal{A}_j \text{ /*Both } \mathcal{A}_i \text{ and } \mathcal{A}_j \text{ are enriched*/} \\
& \vee \big[R_p.\mathcal{A}_i \text{ is not NULL} \wedge R_q.\mathcal{A}_j \text{ is NULL} \qquad \text{/* Only } \mathcal{A}_i \text{ is enriched */} \\
& \qquad \wedge\ read_u(`R_q\text{'}, `\mathcal{A}_j\text{'}, R_q.id, f_{\mathcal{A}_j}.id,)\ op\ R_p.\mathcal{A}_i\big] \\
& \vee \big[R_p.\mathcal{A}_i \text{ is NULL} \wedge R_q.\mathcal{A}_j \text{ is not NULL} \qquad \text{/*Only } \mathcal{A}_j \text{ is enriched*/} \\
& \qquad \wedge\ read_u(`R_p\text{'}, `\mathcal{A}_i\text{'}, R_p.id, f_{\mathcal{A}_i}.id)\ op\ R_q.\mathcal{A}_j\big] \\
& \vee \big[R_p.\mathcal{A}_i \text{ is NULL} \wedge R_q.\mathcal{A}_j \text{ is NULL} \qquad \text{/* None of } \mathcal{A}_i \text{ or } \mathcal{A}_j \text{ are enriched*/} \\
& \qquad \wedge\ read_u(`R_p\text{'}, `\mathcal{A}_i\text{'}, R_p.id, f_{\mathcal{A}_i}.id)\ op\ read_u(`R_q\text{'}, `\mathcal{A}_j\text{'}, R_q.id, f_{\mathcal{A}_j}.id)\big]
\end{aligned}
\tag{4.2}
$$

Figure 4.2(c) illustrates rewritten queries for the query of Figure 4.3(a) using modified selection and join conditions as described above.

4.1.3 Comparison between EQ_{LC} and EQ_{TC}

There are several advantages and disadvantages of EQ_{LC} and EQ_{TC} approaches. Below, we compare them on several criteria.

Load on the DBMS.

In EQ_{LC}, majority of query execution time is spent in the enrichment server as enrichment is performed there. Hence, in EQ_{LC}, the load on DBMS is much lower than EQ_{TC}. The DBMS in EQ_{LC} can support concurrent execution of larger number of queries than EQ_{TC}. In contrast, in EQ_{TC}, most of query execution time is spent on the DBMS. This causes high load on DBMS, making it a bottleneck when more queries arrive simultaneously.

Parallel computation. In EQ_{LC}, the enrichment tasks can be scaled out linearly very easily by adding more machines to the enrichment server. In EQ_{TC}, the scope for parallel

query execution in a multi-node environment, depends on the functionality of the database used for the implementation. If the database supports parallel query execution, then the queries containing UDFs can be executed in parallel as supported by the parallel DBMSs of AsterixDB [23] and Greenplum database [7]. However, linear scale out of such queries is difficult as compared to the EQ_{LC} approach.

Data movement. The movement of data between the DBMS and the enrichment server affects the query processing performance. In EQ_{LC}, the result of probe queries are transmitted from the DBMS to the enrichment server, resulting in a large data movement. In contrast, in EQ_{TC}, enrichment functions are executed using UDFs within the DBMS. Hence, EQ_{TC} approach has much lower data movement than EQ_{LC} approach.

Wasteful enrichments. It is important to limit the number of wasteful enrichments as much as possible during query processing. In EQ_{TC} (as discussed in §4.1.2), we rewrite the query by using enrichment function UDFs. If a tuple does not satisfy a predicate in the selection or join condition, then other predicates with enrichment function UDFs, are not evaluated and the tuple is filtered out. EQ_{LC} approach, although limits the number of wasteful enrichments using `probe queries` (§4.1.1), it is not able to leverage selection and join conditions fully, potentially causing more wasteful enrichments. Hence, EQ_{TC} has more potential in reducing wasteful enrichments during query execution. §5.6 will show this comparison experimentally.

Portability. This is a functional criterion for comparing these two approaches. EQ_{LC} approach can be developed in any programming language with appropriate interfaces for accessing databases and it can use any relational or non-relational database systems. In contrast, in EQ_{TC} approach, enrichment functions are executed as UDFs. Since syntax for writing UDFs is database dependent, the enrichment functions developed as UDF of a particular database system cannot be ported in another database system. Hence, the EQ_{LC} approach is more portable than EQ_{TC} approach.

4.2 Progressive Query Processing

While EQ_{LC} and EQ_{TC} reduce redundant enrichment of data and scale to higher ingestion rates (still supporting queries on data as it arrives), they result in an increased latency of queries, since enrichment (of objects/tuples relevant to query) is done at query time. To reduce query latency, this section explores ways to make EQ_{LC} and EQ_{TC} progressive that iteratively refines the query results as more enrichments are performed. A progressive approach allows data to be consumed by analysts right away and the analysts can stop computation any time they are satisfied with the results [26, 89, 79]. Progressive query processing has been extensively explored in approximate evaluation of aggregation queries (AQP) [58, 88]. AQP supports progressive computation to hide query latency arising from massive data sets on which the query is executed. In contrast, in this paper, the challenge is to hide query latency arising from the execution of expensive enrichment functions.

To develop a progressive approach, we exploit the observation that ML models often exhibit a tradeoff between cost and quality, wherein cheaper functions (*i.e.*, with low execution cost) produce prediction faster with low accuracy, as compared to more expensive functions (*i.e.*, with high execution cost) that produce slower but high quality predictions. *E.g.*, a random forest classifier (RF) implemented using a small number of decision tree models is cheaper but less accurate compared to a random forest classifier using high number of decision tree (DT) models.[3] Here, increased accuracy comes at increased complexity and, hence execution time. *E.g.*, in our experiments over Multi-PIE data [104], a DT classifier for classifying facial expression in an image with tree depth of 5 takes $\approx$ 12 ms/image, whereas a DT classifier with tree depth of 20 takes $\approx$ 35 ms/image while increasing the accuracy from 72% to 81%.

Such a tradeoff between cost and quality exhibited by enrichment functions helps supporting

[3] Similar tradeoff is exhibited by multi-layer perceptrons (where accuracy increases by additional layers and number of perceptrons/layer, until the model over-fits training data [47]), k-NN classifiers (accuracy increases with increasing k), or DT (accuracy increases with depth of tree).

progressive query answering. We can run cheap ML functions on (a subset of) data to generate an initial answer and subsequently select additional data to enrich and/or enrich the data already selected using more complex functions to refine the answers. Below, we develop a progressive approach of enriching data and answering queries for both EQ_{TC} and EQ_{LC} approaches. We begin by precisely defining the semantics of progressive query processing and then describe the ways to achieve progressiveness in the two strategies of EQ_{TC} and EQ_{LC} and denote them by the notation of EQ^P_{TC} and EQ^P_{LC} (*i.e.*, progressive versions). We use the notation of EQ^P to refer to both of them together, otherwise we explicitly use the notation of EQ^P_{TC} or EQ^P_{LC}.

4.2.1 Progressive Queries

Before we describe how EQ^P are executed progressively, we first need to describe some notations. Note that for developing a progressive approach, we now permit multiple enrichment functions to be associated with each derived attribute.[4] Suppose $\mathcal{A}$ is a derived attribute and $\{f_1, f_2, \ldots, f_n\}$ is the set of enrichment functions associated with $\mathcal{A}$. We refer to this set of enrichment functions as a ***function-family*** of $\mathcal{A}$. *E.g.*, for `sentiment` derived attribute in `TweetData` (Table 5.1), the function-family may consist of a decision tree (DT), a k-nearest neighbor (KNN), multi-layered perceptron (MLP), or a support vector machines (SVM) classifier.

At any instance of time, for a given tuple t and for a given derived attribute $\mathcal{A}$ of a relation, multiple enrichment functions might have been executed, resulting in the value for $\mathcal{A}$ in t. We refer to the set of functions in the function-family that have executed as the ***state of the derived attribute*** (denoted as $state(t.\mathcal{A})$) in the tuple t. The state of a derived attribute

[4] Progressive approach is still possible when there is a single enrichment function associated with derived attributes since the system can choose a subset of data to enrich progressively. However, it is much more effective when it is able to exploit the tradeoffs between execution time and quality.

$state(t.\mathcal{A})$ consists of two components: ***state-bitmap*** that stores a list of enrichment functions that have been executed on $t.\mathcal{A}$; and ***state-output*** that stores the output of executed enrichment functions on $t.\mathcal{A}$.

Each function-family is associated with a ***determinization function*** that finds the value of $\mathcal{A}$ in t based on $state(t.\mathcal{A})$. The determinization function (denoted by $DET(*)$) could use any ensemble technique [115, 76] for generating a value based on the classifiers executed so far, *e.g.*, it could use a most likely value, or a value based on majority consensus [76]. We treat the determinization function as a black-box and is independent of the specific function used. Note that $DET(state(t.\mathcal{A}))$ returns a single or a NULL value. NULL value represents a situation when state of the attribute does not provide enough evidence to the determinization function to assign any value for $t.\mathcal{A}$. As more functions execute, the state of $\mathcal{A}$ changes, $DET(state(t.\mathcal{A}))$ computes a new value of $\mathcal{A}$ in t.

The notion of state of a derived attribute generalizes to that of the state of tuples, relations, and database in a straightforward way. The state of a tuple t (or a relation R or a database D) denoted by $state(t)$ (or $state(R_i)$ or $state(D)$) is the concatenation of the state of all derived attributes of t (or the concatenation of the state of all tuples or the concatenation of the state of all relations). Likewise, the concept of determinization also generalizes to that of a tuple, a relation, and a database denoted by $DET(state(t))$, $DET(R_i)$, and $DET(D)$, respectively.

We can now develop the concept of progressive query processing more concretely. The complete strategy of progressive query processing is shown in Figure 4.5. In this strategy, the query execution time is discretized into multiple *epochs* (denoted by $\{e_0, e_1, e_2, \ldots, e_z\}$), where e_0 is a special epoch where some data structures are setup.[5] In each epoch e_k, we select a set of additional derived attribute instances and functions to execute to enrich those

[5]For simplicity, we will consider epochs $\{e_1, e_2, \ldots, e_z\}$ to be fixed size in the remainder of the paper, though, the approach does not require this to be the case.

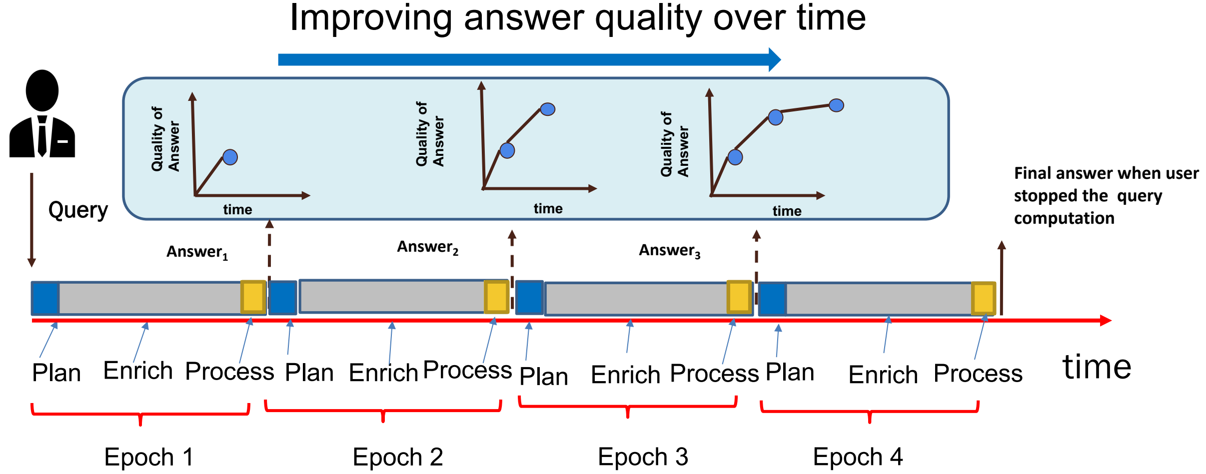


Figure 4.5: Progressive Query Processing Strategy.

attributes. Let q be a query, and let $R_1, R_2, \ldots, R_n$ be the set of relations referred to in q. Let $state(D, e_k)$ be the resulting state of the database based on all the enrichment functions that have executed at (or before) epoch e_k and, furthermore, let $DET(state(D, e_k))$ be the corresponding determinized representation of the database wherein all the derived attributes take a value based on their states.

A progressive query execution in epoch e_k returns the results of query executed over the determinized representation of data at epoch e_k, *i.e.*, it returns $q(DET(state(D, e_k)))$, where the determinized representation of D incorporates all the enrichment functions that have executed so far. Note that answers of q differ from epoch e_{k-1} to e_k, by changing the state of the database by executing the enrichment functions in epoch e_k.

Realizing progressive approach in EQ^P raises two related issues listed below that we address in the rest of the section.

- **Managing State.** State represents the current state of enrichment of all tuples in the database. By explicitly representing information about functions that have executed and their outputs, we can eliminate repeated execution of expensive enrichment functions on

objects. Since the number of objects can be large, and moreover, outputs of enrichment functions can be probability distribution (*e.g.*, capturing probability of an object being a specific value in an image and probability of a specific person being in an image), efficient ways to represent state need to be devised.

- **Incremental Execution of Enrichment and Queries.** In order to execute enrichment and queries in an incremental manner, the following problems need to be addressed: (i) selection of objects and enrichment functions to enrich and (ii) maintaining query results incrementally to avoid the overhead of computing query results from scratch. In the first problem, we need to select a set of ⟨object, enrichment function⟩ pairs that improve the quality of existing query result across different epochs. Sampling based approaches can be used to select objects (similar to AQP systems [19, 92]) and enrichment functions, or a benefit-based approach [52] can be used to optimize specific quality metric of results (*e.g.*, F_α-measure). In the second problem, a straightforward strategy to compute progressive answers is to simply execute q at the end of each epoch e_k over the determinized representation of the database after modification of the state of database due to enrichment. Such an approach, however, is wasteful, since the query is re-executed in each epoch without exploiting prior work performed to evaluate the query in previous epochs. Instead, we explore a strategy based on Incremental View Maintenance (IVM) [36, 86, 71] that are supported by several database engines. Such a strategy, instead of processing the query again on the new state of the data, computes answers as a delta answer over the previously reported query answers.

4.2.2 State Management

In EQ^P, the state of derived attributes of tuples for a relation R is stored as a separate table, $State(R)$. For each derived attribute, $State(R)$ contains a bitmap and an output state vector. Table 4.2 shows a state table for `TweetData` table (see Table 5.1) with two derived

tid	Topic BitMap	TopicOutput	Sentiment BitMap	Sentiment Output
t_1	[1,0,0]	[[0.18,0.64,0.05,...],[],[]]	[1,0,0]	[[0.94, 0.06,0], [], []]
t_2	[1,0,1]	[[0.5,0.2,0.1, ...],[],[0.1,0.6,0.1, ...]]	[1,0,1]	[[0.2,0.6,0.2],[], [0.86,0.1,0.04]]
t_3	[0,1,0]	[[], [0.78,0.06,0.02, ...], []]	[1,1,0]	[[0.1,0.7,0.2], [0.2,0.8,0],[]]

Table 4.2: `TweetDataState` table (created for `TweetData` table).

attributes `Topic` and `Sentiment`. The bitmap contains a bit for each enrichment function associated with the attribute, where a value of 1 means the function was already executed and 0 means it is yet to execute. `TweetDataState` table in Table 4.2 shows a bitmap for `Topic` and `Sentiment` derived attributes. For `topic` attribute of tuple t_2, enrichment functions corresponding to bits 1 and 3 have been executed while the function corresponding to bit 2 is not yet executed. The output state is an array of results of the execution of enrichment functions. In Table 4.2, both enrichment functions were probabilistic classifiers and their outputs were probability distributions ($[0.5, 0.2, 0.1, \dots]$ and $[0.1, 0.6, 0.1, \dots]$, respectively) over an ordered domain of values. Note that, the second element of `TopicOutput` for tuple t_2 is an empty array since the second enrichment function is not yet executed on it.

Compressed State Representation. If the domain size of a derived attribute is large, the columns corresponding to its state output can be large. *E.g.*, if domain size of `topic` in `TweetData` is 40 and there are 3 enrichment functions, then `TopicStateOutput` column (see Table 4.2) could contain 120 values in each row. Such a large domain could incur high storage overhead and read/write cost of states. Instead, EQ^P uses a compressed representation for state output when domain sizes are large. It sets a ***cutoff threshold*** and only stores the domain values whose probability is above that threshold. Domain values are appropriately mapped to integers using a dictionary encoding and the probabilities are stored as key-value pairs. The compressed representation saves us from having to store large tails of. probability distribution.[6] A similar strategy of *Uncertain Primary Indexing* (UPI) is used in [70] to store

[6] Though, at times, it may require re-execution of enrichment functions, if the determinization process requires a probability value from the corresponding enrichment function for the domain value that has been pruned out.

Rel.	TID	Attrs.
'R_1'	1	'$\mathcal{A}_1$', '$\mathcal{A}_3$'
...		
'R_1'	100	'$\mathcal{A}_1$', '$\mathcal{A}_3$'
'R_2'	1	'$\mathcal{A}_2$'
...		
'R_3'	200	'$\mathcal{A}_1$', '$\mathcal{A}_3$'

Table 4.3: `PlanSpaceTable`.

Rel.	TID	Attr-FID
'R_1'	2	$\langle$'$\mathcal{A}_1$', $f_2.id\rangle$, $\langle$'$\mathcal{A}_3$', $f_5.id\rangle$
'R_1'	3	$\langle$'$\mathcal{A}_1$', $f_4\rangle$, $\langle$'$\mathcal{A}_3$', $f_6.id\rangle$
'R_2'	1	$\langle$'$\mathcal{A}_2$', $f_7.id\rangle$
'R_3'	2	$\langle$'$\mathcal{A}_1$', $f_3.id\rangle$, $\langle$'$\mathcal{A}_3$', $f_5.id\rangle$

Table 4.4: `PlanTable`.

subset of tuples (with probability higher than a threshold) of a relation in a faster primary index and remaining tuples in a slower secondary index. In both EQ^P_{TC} and EQ^P_{LC}, state table is maintained in the database.

In EQ^P_{LC}, since enrichment is performed outside of the database to reduce number of database updates every time an enrichment is performed, an in-memory cache for the state table is maintained at the enrichment server. This cache only contains tuples that may need to be enriched during query execution (*i.e.*, the result of probe queries, as will be discussed in §4.2.3) and the updates are pushed to the database at the end of the epoch. EQ^P_{LC} makes sure that the same derived attribute of a tuple is never enriched using the same enrichment function multiple times.

4.2.3 Joint Enrichment and Query Execution

In EQ^P, new enrichment of tuples are performed in each epoch, that requires query results to be updated at the end of each epoch. Instead of re-executing the query to compute the modified answers, an incremental query processing approach based on Incremental View Maintenance (IVM) is used. Before we discuss how to exploit IVM to support incremental processing, we briefly review the IVM technique below.

Incremental View Maintenance (IVM). Given a view corresponding to a query q, for each table $R_i \in q$, IVM algebraically derives an incremental query Δq, that is executed (*e.g.*,

using triggers as in [71]) whenever the base tables change. The Δq query computes only the delta changes of the materialized view q. Correctness of IVM is characterized by ensuring that: $[q(D+\Delta D) = q(D)+\Delta q(D, \Delta D)]$, where D is an instantiation of a database, ΔD are the updates to D, $q(D)$ is the prior query results based on D, Δq is the modified query that is executed seeded based on ΔD, and the notation '+' in the expression $q(D)+\Delta q(D, \Delta D)$ refers to the way of combining answers of the two queries to generate the overall answer to q over the modified data.

A comprehensive description of how Δq can be algebraically derived from q appears in [71, 36, 86]). Below, we provide few examples of how operators are transformed to provide intuition. Let ΔR_1 and ΔR_2 be the set of tuples updated to relations R_1 and R_2, respectively.

- Let $q = \sigma_C(R_1)$ where C represents a set of selection conditions, then $\Delta q = \sigma_C(\Delta R_1)$, *i.e.*, the selection condition needs to be applied only on the updated tuples of R_1.
- Let $q = R_1 \bowtie R_2$, then $\Delta q = (\Delta R_1 \bowtie R_2 + R_1 \bowtie \Delta R_2)$, *i.e.*, only the updated tuples of R_1 needs to be joined with relation R_2 and the updated tuples of R_2 with R_1.
- Let $q = \gamma_g(R))$, then $\Delta q = \gamma_g(\Delta R_1)$, where γ is an aggregation function, g is a `group by` attribute. In this scenario, the aggregation function γ_g needs to be applied directly on the updated tuples.

IVM techniques have been integrated within several popular databases: PostgreSQL [9], Oracle [11], and Amazon Redshift [10]. IVM implementations can be significantly efficient than recomputing the original query. *E.g.*, the rewritten selection query above requires only selections to be performed on updated tuples (that may be few) as compared to having to re-execute the selection over the entire table. Likewise, incremental computation of joins and other operators may be significantly efficient compared to the naive implementation. Systems such as [71], have shown ≈ 90 times improvement for certain queries in TPC-H benchmark [17], in terms of the number of refreshes supported by IVM as compared to a full refresh of materialized view after each update of base tables.

Incremental Processing. EQ^P exploits IVM to incrementally compute the modified query answers because of enrichments performed on the data during an epoch. The query execution consists of four steps discussed in the following subsections. Out of the four steps, the query setup is performed once in the zero-th epoch e_0 (this is a special epoch where only query setup is performed) and all other steps are executed iteratively, *i.e.*, once per epoch (e_i, where $i \geq 1$).

4.2.3.1 Query Setup

During the query setup, EQ^P initializes a materialized view q_v for the query q based on the current state of the database. Results of q_v are incrementally updated as more data is enriched in future epochs.

In addition, during query setup, to determine the set of possible candidate enrichments to be performed in future epochs, probe queries $pq(R_i)$[7] are executed for each relation $R_i \in q$. The query $pq(R_i)$ needs to be appropriately modified since simply checking if the value of a derived attribute is not NULL, no longer suffices if a tuple is fully enriched. Instead, the probe queries need to exploit the state of derived attributes to determine if the derived attribute can be further enriched by exploiting other enrichment functions that have not been executed yet on the attribute value. The test for whether for a given attribute an enrichment function has not yet been executed (and hence the tuples can be further enriched) is performed by checking if the sum of the bits in the array of $\mathcal{A}j$*StateBitmap* column of a tuple is equal to the length of the array in $\mathcal{A}j$*StateBitmap* column.

Example 4.2.1. Considering the probe query of Figure 4.2(b) (bottom) for relation R_1, the modified probe query is presented in Figure 4.6. In the modified query, if sum of the bits in the array of $\mathcal{A}_1$*StateBitmap* column of a tuple is not equal to the length of the array in

[7] Discussed in §4.1.1.

$$\sigma_{\substack{R_1.A_2=a_2 \wedge (array_sum(\mathcal{A}_1 StateBitmap)!= \\ array_length(\mathcal{A}_1 StateBitmap))}} (R_1 \bowtie R_1State) \ltimes_{R_1.A_2=R_2.A_4} \Big[R_2 \ltimes_{R_2.A_4=R_3.A_5} (\sigma_{\substack{(R_3.A_5=a_5) \wedge (array_sum(\mathcal{A}_1 StateBitmap)!= \\ array_length(\mathcal{A}_1 StateBitmap))}} (R_3 \bowtie R_3State))\Big]$$

Figure 4.6: Updated probe query for R_1.

$\mathcal{A}_1 StateBitmap$ column, then that tuple is not completely enriched and hence it is returned in the probe query result.

`PlanSpaceTable`. The result of the probe queries are stored in a table entitled `PlanSpaceTable` an example of which is shown in Table 4.3. This table stores a set of candidate tuples of relations $R_i \in q$ that are considered for enrichment to answer q. Rows in `PlanSpaceTable` correspond to the name of the relation (R_i) included in q, the tuple ID, and the list of derived attributes for which the tuple needs to be enriched for q (see Table 4.3).

4.2.3.2 Enrichment Planning

At the beginning of each epoch, based on the state of the tuples, EQ^P moves a set of tuples from the `PlanSpaceTable` to a `PlanTable` for (potential) enrichment during this epoch. The `PlanTable` contains three columns: `RelationName`, `TID` (tuple identifier), and `Attr-FID` (stores a list of ⟨ name of derived attribute, enrichment function identifier⟩ pairs). For a tuple and each derived attribute that require enrichment in `PlanSpaceTable`, an enrichment function is chosen. The list of the derived attribute and enrichment function pairs are stored in the `Attr-FID` column. A sample `PlanTable` based on selecting tuples from the `PlanSpaceTable` in Table 4.3 is shown in Table 4.4.

In order to populate the `PlanTable` from `PlanSpaceTable`, we have to select a set of ⟨tuple, derived attribute, enrichment function⟩ triplets for enrichment during a given epoch. Such selection can be based on diverse criteria. Below, we describe a set of strategies that we will compare in the experiments in §5.6.

- **Sampling-based Function Ordered (SB(FO))** method, where a sample of tuples and derived attributes are selected from `PlanSpaceTable` using simple random sampling (*i.e.*, each tuple has equal probability of being part of the sample) and for each tuple in the sample, enrichment functions are chosen based on the order of $\frac{quality}{cost}$ that were not executed on the tuple earlier, where quality of the function is measured using any classifier metrics such as accuracy and AUC score and cost is measured using average execution time of the function per object.

- **Sampling-based Object-Ordered (SB(OO))** method, where a sample of tuples are chosen using simple random sampling, and each tuple is enriched completely using all the available enrichment functions for all the derived attributes that are required for q.

- **Sampling-based Random Ordered (SB(RO))** method, where a sample of tuples are selected using simple random sampling from `PlanSpaceTable` and for each tuple, a derived attribute and an enrichment function is selected randomly that is not executed on the tuple earlier (based on the state of tuple).

The cost of selected plan is the summation of the cost of enrichment functions part of `PlanTable`. Note that for the plan to be valid (*i.e.*, executable during the epoch), the cost of selected plan *must* be smaller than epoch duration. Apart from above strategies, other ways of selecting enrichment plan can also be devised, (*e.g.*, one could use a query cognizant approach that attempts to perform enrichments that improve quality of query results optimally).

4.2.3.3 Computing Progressive Answers

To compute q progressively, we need to compute delta answers for q_v based on the data modifications, resulted due to enrichment in the epoch. In EQ^P_{LC}, enrichments were performed

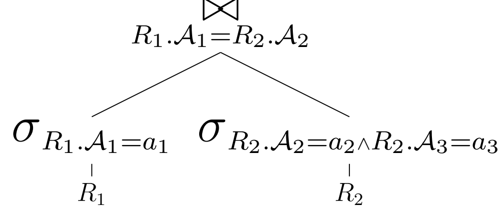
independently (at the enrichment server) and modified/updated attribute values of enriched tuples in the base relations, triggered the recomputation of query answers.

Computing delta answers in EQ^P_{TC} is significantly more complex than in EQ^P_{LC} which we discuss first.

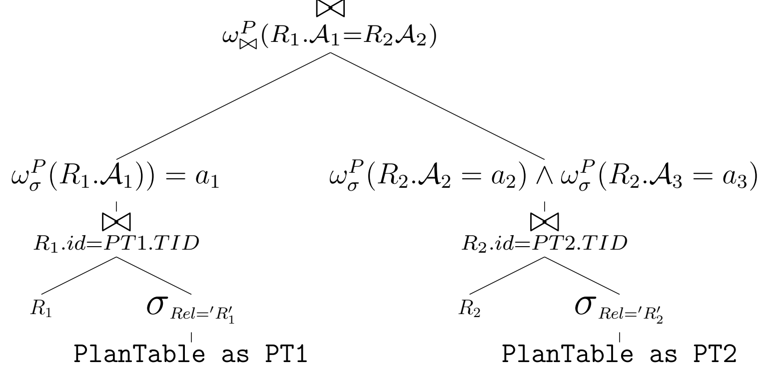
Progressive Answering in EQ^P_{LC}: In EQ^P_{LC}, enrichments are performed independently (at the enrichment server) and modified/updated attribute values of enriched tuples in the base relations, triggered the recomputation of query answers. In EQ^P_{LC}, the IVM query q_v is simply the original query q. During each epoch e_k, the $\langle$tuple, derived attribute, enrichment function$\rangle$ pairs of `PlanTable` are executed, followed by the execution of appropriate determinization function DET. The resulting updates are reflected in the database by replacing current values of the derived attributes of enriched tuples based on the functions executed upto that time. Hence, the determinized representation of the database changes from $DET(state(D, e_{k-1}))$ to $DET(state(D, e_k))$. Such an update, triggers IVM to update the materialized view based on ΔD that consists of all changes that took place in e_k (*i.e.*, $\big[DET(state(D, e_k))$ - $DET(state(D, e_{k-1}))\big]$). Specifically, $\Delta q(DET(D, e_{k-1}), DET(\Delta D))$ is executed to compute $q(DET(D, e_k))$, *i.e.*, the query result of e_k.

Progressive Answering in EQ^P_{TC}. Unlike EQ^P_{LC} where enrichments are performed separately at the enrichment server resulting in modifications to the base tables that triggers a recomputation of q_v, in EQ^P_{TC}, enrichment of tuples are performed within the query q as part of the UDF execution. Thus, the incremental evaluation of query results has to be triggered by updates to a different table. In particular, in EQ^P_{TC}, the query q_v uses updates to `PlanTable`, to support the incremental evaluation of the query results.

For this purpose, the query q is rewritten to use the `PlanTable` as follows: all relation $R_i \in q$, that require enrichment, R_i is replaced by the expression: $R_i \bowtie_{R_i.TID=PlanTable.TID}$ $(\sigma_{RelName=`R_i'}(PlanTable))$.



(a): Original query.



(b): Query used in IVM for EQ^P_{TC}.

Figure 4.7: The incremental query used by IVM in EQ^P_{TC}.

Example 4.2.2. Consider the query of Figure 4.6(a). The tuples of both R_1 and R_2 require enrichment because of the query conditions on attributes $\mathcal{A}_1$ and $\mathcal{A}_2$. In the rewritten query of q to support incremental evaluation, both the relations are joined with `PlanTable`, as shown in Figure 4.6(b) (other rewrites of selection and join conditions are denoted as ω^P_σ and $\omega^P_{\bowtie}$ will be clear in a later part of section).

The above rewrite triggers the incremental computation of the query result each time the `PlanTable` is modified.

Example 4.2.3. Consider the query of Figure 4.6(a) and the rewritten query of Figure 4.6(b). Suppose in epoch e_k, a $\langle$tuple, derived attribute, enrichment function$\rangle$ triplet is added to the `PlanTable` where the tuple belongs to relation R_1. The addition of this triplet triggers a view update of the query shown in Figure 4.6(b) as `PlanTable` is part of the view definition. ■

While the approach to create q_v by rewriting q using `PlanTable` results in desired incremental

updates, it suffers from a subtle complexity. Specifically, for a given tuple t, when enrichment functions execute, the change in state table, results in a new determinized value for the derived attribute $t.\mathcal{A}_j$ in R. If we update the current value of $t.\mathcal{A}_j$ in R, the change would cause the refresh to the view to cascade resulting in a duplicate update to the results. To see this, consider the following example.

Example 4.2.4. In Example 4.2.3, suppose a row $\langle t, \mathcal{A}_i, f_j \rangle$ where $t \in R$ is added to `PlanTable` in epoch e_k. Hence, the incremental query execution is triggered. During the execution of incremental query, enrichment function is executed on t and the condition of $(R.\mathcal{A}_1 = a_1)$ is evaluated on it. As a result, the state of t and the determinized value of $t.\mathcal{A}_1$ is updated. Now, if we update the new attribute value of t in R, it will cause another trigger to incremental query execution (as R is part of the q_v), causing duplicate results. ■

To prevent such a situation, we cannot update the value of attribute $t.\mathcal{A}_j$ directly during the execution of q_v. We instead store the value of determinized representation of $t.\mathcal{A}_j$ separately as part of state table R_i`State`. To do so, we extend the schema of R_i`State` to include a new field $\mathcal{A}_j$`Value` for each derived attribute in R_i.[8]

Since we do not modify the value of derived attribute in place, we need to define additional UDFs for q_v to read the value of attribute $\mathcal{A}_j$. Note that if $\mathcal{A}_j$ is modified (due to enrichment) during the execution of the query, its value resides in the $\mathcal{A}_j$`Value` column of the R_i`State` table. Otherwise, if $\mathcal{A}_j$ is not modified in an epoch, its most recent value is available in $\mathcal{A}_j$ column of table R_i. To enable q_v to correctly retrieve the value of $\mathcal{A}_j$, we define two UDFs entitled *CheckState* and *GetValue* as described below.

***CheckState* UDF**: The objective of this UDF is to check if a particular tuple was enriched for a particular derived attribute present in the query. The input to the UDF is a relation name, a derived attribute name, a tuple identity, and `Attr-FID` value retrieved from the

[8] R_i`State`, thus, contains three fields: $\mathcal{A}_j$`Bitmap`, $\mathcal{A}_j$`Output`, and $\mathcal{A}_j$`Value`, for attribute $\mathcal{A}_j$.

`PlanTable` for the corresponding tuple. `Attr-FID` column of `PlanTable`, is used to get the enrichment function that needs to be executed. If that enrichment function was executed before, then it returns true, otherwise false. Note that *CheckState* retrieves this information from the state bitmap column of the derived attribute in the state table §4.2.2.

***GetValue* UDF:** The purpose of *GetValue* UDF is to retrieve the latest value of a derived attribute for a given tuple. *GetValue* takes as input a relation name (*e.g.*, 'R_i''), attribute name (*e.g.*, '$\mathcal{A}_j$'), and a tuple identity and returns the determinized value of $\mathcal{A}_j$ stored in the $\mathcal{A}_j$`Value` column of R_i`State` table.

Apart from the above two UDFs, we further need to appropriately modify the $read_u$ function shown in §4.1.2 (that enriches derived attributes as a side effect of reading them) to account for the way the state and data value are stored when multiple enrichment functions can be associated with a derived attribute.

Modified $read_u$ UDF: Given a tuple, a derived attribute $\mathcal{A}_j$, and an enrichment function, the modified $read_u$ UDF executes the enrichment function on the tuple, updates the state, and returns the determinized representation of the tuple for the derived attribute.

The modified $read_u$ function takes the following inputs: the name of the relation, the name of the derived attribute, the tuple identity, and the list of ⟨derived attribute, function ID⟩ pairs (stored in *Attr_FID* column of `PlanTable`). It executes the enrichment function on the input tuple for the derived attribute (parsed from *Attr_FID*), updates the state of the tuple and returns the derived attribute value. In the state table, it updates the state bitmap, state output, and attribute-value columns. The state-bitmap is updated by setting the bit corresponding to the enrichment function executed on it. The state-output is updated by the output of the executed enrichment function. The attribute-value column is updated by the latest determinized representation of the derived attribute value of the tuple. The determinized representation of the derived attribute is returned by the $read_u$ function.

Using $read_u$ UDF as shown above, we now describe how query q_v is rewritten. Essentially, the selection conditions are rewritten appropriately to check if the corresponding derived attribute was enriched during the epoch from the state table (enrichment is only performed if the enrichment function was not executed earlier). This check is performed using *CheckState* and *GetValue* UDFs as defined earlier. The complete rewrite logic of a selection condition $(\omega_\sigma^P(R.\mathcal{A}_i \; op \; a_i))$ is presented below:

$$
\begin{aligned}
&\big[\mathit{CheckState}(\text{`}R\text{'}, \text{`}\mathcal{A}_i\text{'}, R.id, \mathit{Attr_FID}) \quad \text{/* } \mathcal{A}_i \text{ is enriched.*/} \\
&\qquad \wedge \mathit{GetValue}(\text{`}R\text{'}, \text{`}\mathcal{A}_i\text{'}, R.id) \; op \; a_i\big] \\
&\vee \big[!\mathit{CheckState}(\text{`}R\text{'}, \text{`}\mathcal{A}_i\text{'}, R.id, \mathit{Attr_FID}) \quad \text{/* } \mathcal{A}_i \text{ is not enriched.*/} \\
&\qquad \wedge \mathit{read}_u(\text{`}R\text{'}, \text{`}\mathcal{A}_i\text{'}, R.id, \mathit{Attr_FID}) \; op \; a_i\big]
\end{aligned}
\tag{4.3}
$$

In the above rewrite logic, the rewritten condition first checks if the tuple is already enriched or not for a given derived attribute $\mathcal{A}_i$ using the *CheckState* function. If it is already enriched, then the *GetValue* UDF is used to retrieve the latest attribute value of the tuple and then the selection condition is executed. This rewrite logic is achieved by: $\big[\mathit{CheckState}(\text{`}R\text{'}, \text{`}\mathcal{A}_i\text{'}, R.id, \mathit{Attr_FID}) \wedge \mathit{GetValue}(\text{`}R\text{'}, \text{`}\mathcal{A}_i\text{'}, R.id) \; op \; a_i\big]$. If a tuple is not enriched before (*i.e.*, *CheckState* UDF output is 0), then the $read_u$ UDF is executed to enrich the tuple and finally the selection condition is executed based on the output of $read_u$ UDF. It is achieved by the condition of $\big[!\mathit{CheckState}(\text{`}R\text{'}, \text{`}\mathcal{A}_i\text{'}, R.id, \mathit{Attr_FID}) \wedge \mathit{read}_u(\text{`}R\text{'}, \text{`}\mathcal{A}_i\text{'}, R.id, \mathit{Attr_FID}) \; op \; a_i\big]$.

The corresponding rewrite logic for join condition $\omega^{P}_{\bowtie}(R_p.\mathcal{A}_i\ op\ R_q.\mathcal{A}_j))$ is shown below (similar to rewrite logic of selection condition).

$$
\begin{aligned}
&\big[\mathit{CheckState}(\text{`}R_p\text{'}, \text{`}\mathcal{A}_i\text{'}, R_p.id, \mathit{Attr_FID}) \\
&\quad \wedge \mathit{CheckState}(\text{`}R_q\text{'}, \text{`}\mathcal{A}_j\text{'}, R_q.id, \mathit{Attr_FID}) \qquad \text{/* Both of } \mathcal{A}_i \text{ and } \mathcal{A}_j \text{ are enriched*/} \\
&\quad \wedge \mathit{GetValue}(\text{`}R_p\text{'}, \text{`}\mathcal{A}_i\text{'}, R_p.id)\ op\ \mathit{GetValue}(\text{`}R_q\text{'}, \text{`}\mathcal{A}_j\text{'}, R_q.id)\big] \\
\vee &\big[\mathit{CheckState}(\text{`}R_p\text{'}, \text{`}\mathcal{A}_i\text{'}, R_p.id, \mathit{Attr_FID}) \\
&\quad \wedge !\ \mathit{CheckState}(\text{`}R_q\text{'}, \text{`}\mathcal{A}_j\text{'}, R_q.id, \mathit{Attr_FID}) \qquad \text{/* Only } \mathcal{A}_i \text{ is enriched.*/} \\
&\quad \wedge \mathit{GetValue}(\text{`}R_p\text{'}, \text{`}\mathcal{A}_i\text{'}, R_p.id)\ op\ \mathit{read}_u(\text{`}R_q\text{'}, \text{`}\mathcal{A}_j\text{'}, R_q.id, \mathit{Attr_FID})\big] \\
\vee &\big[!\ \mathit{CheckState}(\text{`}R_p\text{'}, \text{`}\mathcal{A}_i\text{'}, R_p.id, \mathit{Attr_FID}) \\
&\quad \wedge \mathit{CheckState}(\text{`}R_q\text{'}, \text{`}\mathcal{A}_j\text{'}, R_q.id, \mathit{Attr_FID}) \quad \text{/* Only } \mathcal{A}_j \text{ is enriched.*/} \\
&\quad \wedge \mathit{read}_u(\text{`}R_p\text{'}, \text{`}\mathcal{A}_i\text{'}, R_p.id, \mathit{Attr_FID})\ op\ \mathit{GetValue}(\text{`}R_q\text{'}, \text{`}\mathcal{A}_j\text{'}, R_q.id)\big] \\
\vee &\big[!\ \mathit{CheckState}(\text{`}R_p\text{'}, \text{`}\mathcal{A}_i\text{'}, R_p.id, \mathit{Attr_FID}) \\
&\quad \wedge !\ \mathit{CheckState}(\text{`}R_q\text{'}, \text{`}\mathcal{A}_j\text{'}, R_q.id, \mathit{Attr_FID}) \quad \text{/* None of } \mathcal{A}_i \text{ and } \mathcal{A}_j \text{ are enriched.*/} \\
&\quad \wedge \mathit{read}_u(\text{`}R_p\text{'}, \text{`}\mathcal{A}_i\text{'}, R_p.id, \mathit{Attr_FID})\ op\ \mathit{read}_u(\text{`}R_q\text{'}, \text{`}\mathcal{A}_j\text{'}, R_q.id, \mathit{Attr_FID})\big]
\end{aligned}
\tag{4.4}
$$

In the above rewrite logic, given a join condition of $(R_p.\mathcal{A}_i\ op\ R_q.\mathcal{A}_j)$, for a tuple pair, the rewritten condition first checks if both the derived attributes were enriched (*i.e.*, *CheckState* returning true for both tuples). If they were, then the join condition is executed on the output of the *GetValue* function as it returns the latest attribute value of the tuples. In second and third condition of rewrite, only one tuple of the tuple pair is enriched. For the tuple that was enriched before, the value was retrieved using *GetValue* UDF. The other tuple is enriched first using the modified $read_u$ function and the join condition between the tuple pair is executed. In the fourth condition of rewrite, both the tuples were not enriched in the epoch. Hence, both the tuples were enriched using the modified $read_u$ function and the join condition is executed.

Based on the query rewrite logic described above, now we rewrite the query of q_v that is used to incrementally maintain the result of query q. Considering the query shown in Figure 4.6(a), the rewritten query of q_v is shown in Figure 4.6(b). In the rewritten query, the selection and join conditions are rewritten using the rewrite logic of ω_σ and $\omega_\bowtie$ as we described above.

The query of q_v as described above contains the `PlanTable`. In an epoch e_k, when a set of $\langle$tuple,derived attribute, enrichment function$\rangle$ triplets are added to `PlanTable`, a re-execution of query q_v is triggered. During the execution of q_v, the enrichment of triples in the `PlanTable` take place and the state of the tuples are updated (*i.e.*, using $read_u$ UDF), and the IVM maintained using query q_v is updated. Hence, the delta query used to compute the delta changes to the query result of q at the end of each epoch e_k is created using the same query of q_v.

4.2.3.4 Fetching Results

In EQ^P, users can fetch complete query results at the end of an epoch by querying the IVM. If complete answer set is large, users can retrieve delta changes of answers, *i.e.*, inserted/deleted/updated tuples from previous epoch. Current implementation allows users to fetch delta answers only from last epoch. Fetching delta answers from any arbitrary epoch using a cursor is complex (will be supported in future version), since the query processing in EQ^P are not demand-driven, as in SQL databases. The refined answers due to $\Delta q(DET(D, e_{k-1}), DET(\Delta D))$, may result in retraction of previously returned tuples, or addition of new tuples, or updates to the previously reported answers.

Relation	#tuples	Size(GB)	Derived attributes	Functions used
TweetData	11M	10.5	sentiment(3)	GNB,KNN,SVM,MLP
			topic(40)	GNB,KNN,LDA,LR
MultiPie[104]	100K	16.9	gender(2)	DT,GNB,KNN,MLP
			expression(5)	DT, GNB, RF, KNN

Table 4.5: Datasets used in experiments.

4.3 Experimental Evaluations

This section evaluates the performances of both EQ_{LC} and EQ_{TC} approaches. Specifically, we address the following questions:

- How does EQ_{TC} perform compared to EQ_{LC} approach in terms of savings in the enrichment? Does exploiting query semantics during enrichment really pay off?
- How does progressive query processing help in reducing query response time for queries with expensive UDFs compared to the traditional approaches of query processing?
- Which approach between EQ_{TC} and EQ_{LC} is well suited for supporting progressive query processing? What are their advantages and limitations in terms of overheads?
- How do enrichment plan generation strategies affect progressive query processing? Are there scopes for improvement?

4.3.1 Experimental Setup

Datasets. We used two datasets to evaluate the performances of EQ_{LC} and EQ_{TC} approaches: (*i*) TweetData collected using APIs with 11 million rows, two derived attributes: ⟨`sentiment` and `topic`⟩, and six fixed attributes: ⟨`tid`, `UserID`, `Tweet`, `feature`, `location`, and `TweetTime`⟩ (*ii*) MultiPie [104] dataset with 100K facial images, two derived attributes:

ID	Queries	Application	Query Type
Q1	SELECT * from MultiPie where gender=1 and CameraID $< c_1$	Images	Selection
Q2	SELECT * from MultiPie where gender = 1 and expression = 2 and CameraID $< c_1$	Images	Selection
Q3	SELECT tid, UserID, Tweet, location, TweetTime from TweetData where sentiment = s_1 and topic = t_1 and TweetTime between(t_1,t_2)	Tweets	Selection
Q4	SELECT * from TweetData T1, TweetData T2 where T1.sentiment = T2.sentiment and T1.topic = T2.topic and T1.TweetTime between(t_1,t_2) and T2.TweetTime between (t_1, t_2)	Tweets	Join
Q5	SELECT * from MultiPie M1, MultiPie M2 where M1.expression = M2.expression and M1.gender = M2.gender and M1.CameraID $< c_1$ and M2.CameraID $< c_1$	Images	Join
Q6	SELECT * from MultiPie M1, MultiPie M2 where M1.gender = M2.gender and M1.expression = 1 and M2.expression = 2 and M1.CameraID $< c_1$ and M2.CameraID $< c_1$	Images	Join
Q7	SELECT * from TweetData T1, State S where T1.location = S.city and S.state='California' and T1.sentiment = 1 and T1.TweetTime between(t_1,t_2)	Tweets	Join
Q8	SELECT topic, count(*) from TweetData where T1.TweetTime between(t_1,t_2) group by sentiment	Tweets	Aggregation

Table 4.6: Query templates used.

⟨`gender` and `expression`⟩, and five fixed attributes: ⟨`ImageID`, `UserID`, `CameraID`, `Image`, and `ImageTime`⟩ (see Table 4.5).

Enrichment Functions. We used the following probabilistic classifiers as enrichment functions: Gaussian Naïve Bayes (GNB), Decision Tree (DT), Support Vector Machine (SVM), K-Nearest Neighbor (KNN), Multi-Layered perceptron (MLP), Linear Discriminant Analysis (LDA), Logistic Regression (LR), and Random Forest (RF). GNB classifier was calibrated using isotonic-regression model [117] and remaining classifiers were calibrated using Platt's sigmoid model [93] during cross-validation to the output probability distribution.

Queries. Table 4.6 shows nine queries, where *Q1-Q3* are *selection queries*, ***Q4-Q7*** are *join queries*, and ***Q8*** is an ***aggregation query***. At any instance of time in EQ_{LC} and in EQ_{TC}, only one query was executed at the servers.

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Baseline	100K	200K	22M	22M	200K	200K	11M	11M
EQ_{LC}	10K	20K	200K	200K	20K	20K	4K	100K
EQ_{TC}	10K	13705	164210	127550	12943	12340	4K	100K

Table 4.7: Exp 1. Number of enrichments in EQ_{LC} and EQ_{TC}.

4.3.2 Experimental Results

For the experiments, we setup the database on an AWS server with 16 core 2.50GHz Intel Xeon CPU, 64GB RAM, and 1TB SSD. We used another AWS server with the same configuration as the enrichment server. The datasets were stored in two tables of a PostgreSQL database (version 13). In EQ_{TC}, the enrichment functions were implemented as PostgreSQL UDFs, and in EQ_{LC}, the enrichment functions were implemented as Python functions.

4.3.2.1 EQ_{TC} VS EQ_{LC}

To compare EQ_{TC} and EQ_{LC} using the criterias of §4.1.3, we used MLP for `sentiment` (6.26 ms/tweet), GNB for `topic` (7.82 ms/tweet), MLP for `gender` (32.6 ms/image), and RF for `expression` attribute (28.7 ms/image).

Exp 1: Number of enrichments. Table 4.7 shows the total number of enrichments for EQ_{LC}, EQ_{TC}, and the baseline approach that performs complete enrichment at the time of data ingestion. Both EQ_{TC} and EQ_{LC} perform significantly well as compared to the baseline approaches as enrichment is performed only in the context of the query leading close to 90% savings in enrichment for all queries. Note that this number also depends on the selectivity of the query as will be shown in the next experiment. Between EQ_{TC} and EQ_{LC}, EQ_{TC} performs either same or much lower number of enrichments than EQ_{LC} since EQ_{TC} exploits query semantics to remove redundant enrichments. In EQ_{TC}, for the queries with multiple predicates (*i.e.*, Q2-Q6) on derived attributes, the tuples that did not satisfy one predicate were not enriched for the remaining derived attributes, that resulted in savings

Approach	Selectivity of TweetTime	topic $\leq$ 10	topic $\leq$ 20	topic $\leq$ 30	topic $\leq$ 40
Baseline	1%	22M	22M	22M	22M
EQ_{LC}	1%	20K	20K	20K	20K
EQ_{TC}	1%	11.4K	14.7K	15.4K	16.8K
Baseline	10%	22M	22M	22M	22M
EQ_{LC}	10%	200K	200K	200K	200K
EQ_{TC}	10%	100.1K	103.9K	104.6K	139K
Baseline	100%	22M	22M	22M	22M
EQ_{LC}	100%	22M	22M	22M	22M
EQ_{TC}	100%	11.16M	12.14M	13.6M	15.8M

Table 4.8: Number of enrichments saved in EQ_{TC} compared to EQ_{LC} with varying selectivity of precise condition (*i.e.*, TweetTime between (t_1, t_2)) in query Q3.

on enrichment. However, in Q1, Q7, and Q8, the number of enrichments were same for both EQ_{LC} and EQ_{TC}, as Q1 and Q7 had a single predicate on derived attribute and Q8 was an aggregation query with a selection condition on a fixed attribute. Hence, for these queries, during query execution, EQ_{TC} did not have opportunity to eliminate redundant enrichment based on predicates on derived attributes.

Number of enrichments with varying selectivity. We define selectivity as the ratio of input-cardinality to the output-cardinality of a predicate. We used Q3 for this experiment where we replace the predicate of (topic $= t_1$) with the predicate of (topic $\leq k$) where k is varied from 10 to 40 to control the selectivity of topic predicate and varied the selectivity of TweetTime predicate to control the number of tweets that may require enrichment. Table 4.8 shows the results, where we observe that when predicate selectivity increases (*i.e.*, passes fewer input tuples), the savings in terms of enrichment for both EQ_{LC} and EQ_{TC} is high as compared to the baseline approach (as shown in the first three rows of the Table 4.8 with selectivity of 10%). The improvement in EQ_{TC} as compared to EQ_{LC} is that all the tuples that do not satisfy the predicate of (topic $\leq k$) are not further enriched for the attribute of sentiment. When the selectivity increases to 100%, then EQ_{LC} performs the same as the baseline approach. EQ_{LC} can still save more enrichments by exploiting query semantics on the derived attribute. Therefore, for queries with high selectivity, both EQ_{LC}

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
EQ_{LC}	378.18	684	1291	1319.5	736	705	32.72	693
EQ_{TC}	306.7	572.73	944.68	905.3	627.51	582.37	28.17	612

Table 4.9: Latency of queries in EQ_{LC} and EQ_{TC} approaches (in seconds).

and EQ_{LC} performs much better compared to EQ_{LC}. Furthermore, from this experiment, we can conclude that exploiting query semantics is quite useful to reduce enrichment cost.

Execution time/Server load. Table 4.9 shows the latency of queries Q1 to Q8 in EQ_{LC} and EQ_{TC}, where latency is the average execution time of 50 queries generated from each template of Table 4.6. *E.g.*, we created 50 queries of $Q2$ by setting different values in `CameraID` predicate. As expected, the latency in EQ_{TC} is much lower than EQ_{LC}, as the total number of enrichments performed in EQ_{TC} is much lower than EQ_{LC}.

Figure 4.8 shows the execution time spent in enrichment server and DBMS for EQ_{LC} and EQ_{TC}. In EQ_{LC}, the majority of the query execution time was spent at enrichment server, whereas in EQ_{TC}, majority of the time was spent at DBMS. Also, it shows that total execution time in EQ_{LC} is higher than EQ_{TC}, as EQ_{LC} enriches more tuples, as shown in Table 4.7. The network latency is higher in EQ_{LC}, since the result of probe queries are transmitted from DBMS to enrichment server. *E.g.*, in $Q2$ and $Q3$, average network latencies were 72 seconds and 37.1 seconds, respectively.

Since in EQ_{TC}, the majority of query execution time is spent in DBMS, it can become a bottleneck when more queries execute concurrently. Furthermore, adding more machines does not improve EQ_{TC}, since enrichment functions are executed at DBMS. A data partitioning-based approach can be used to scale-out DBMS but it requires a distributed database and expensive data partitioning. Hence, when query workload is low, one should use EQ_{TC}, and then switch to EQ_{LC} when workload increases.

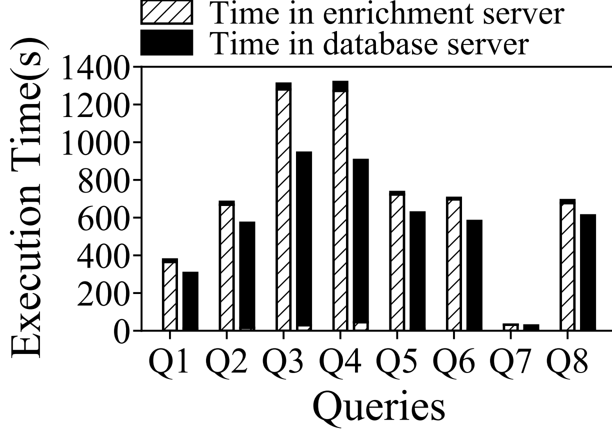


Figure 4.8: Times in enrichment server and DBMS (Left bars=EQ_{LC}, Right bars=EQ_{TC}).

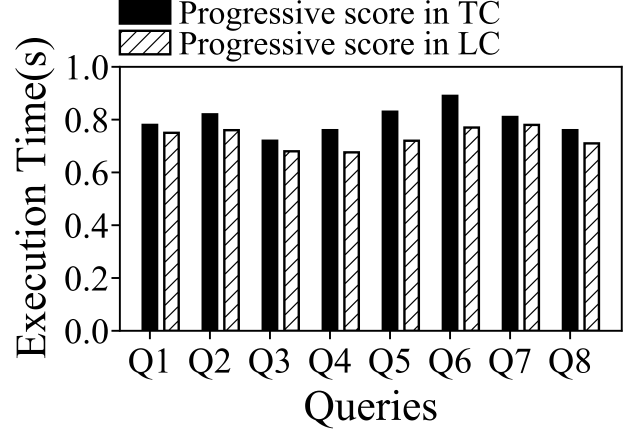


Figure 4.9: Progressive score achieved by queries in EQ_{TC} and EQ_{LC}.

4.3.2.2 Progressiveness

This section considers all enrichment functions for the derived attributes as shown in Table 4.5. We present progressive quality improvement of queries Q1-Q8 in two ways: (i) plotting the quality of query results with respect to time and (ii) quantifying the quality improvement over time using a metric of *progressive score*, denoted by $\mathcal{PS}()$. This metric was used in previous literature to measure progressiveness [89, 26].

$$\mathcal{PS}(Ans(q,E)) = \sum_{i=1}^{|E|} W(e_i) \cdot [\mathcal{Q}(Ans(q,e_i)) - \mathcal{Q}(Ans(q,e_{i-1}))] \tag{4.5}$$

where, ($E = \{e_1, e_2, \ldots, e_z\}$) are the epochs, $W(e_i) \in [0,1]$ is the weight allotted to epoch e_i, $W(e_i) > W(e_{i-1})$), $\mathcal{Q}$ is the quality of answers, and $[\mathcal{Q}(Ans(q,e_i)) - \mathcal{Q}(Ans(q,e_{i-1}))]$ is the improvement in the quality of answers occurred in the epoch e_i.

Exp 2: Progressiveness of different queries. Figure 4.10 evaluates EQ^P_{LC} and EQ^P_{TC} approaches, in terms of progressive quality improvement achieved. Figures 4.10(a), 4.10(b), and 4.10(c) show the results for queries Q2, Q3, and Q4, where the quality of answers is

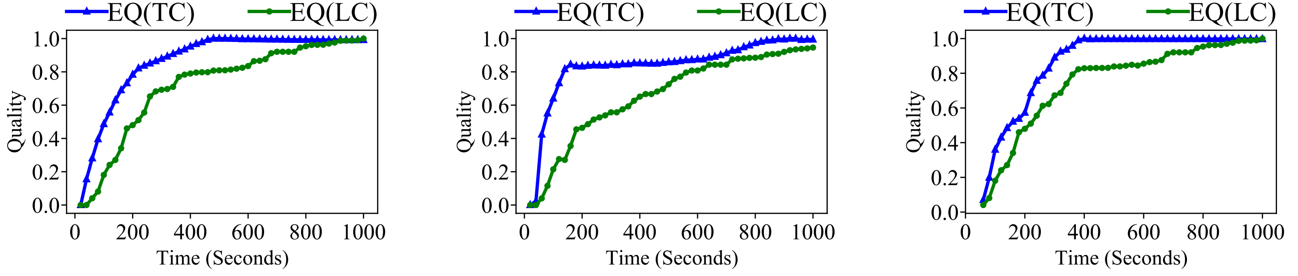


Figure 4.10: Progressiveness achieved in EQ_{LC} and EQ_{TC} for (a) Q2, (b) Q3, and (c) Q4.

measured using ***normalized F_1 measure*** *i.e.*, F_1/F_1^{max}, where F_1^{max} is the maximum F_1 measure achieved during query execution. The results of remaining queries and ***normalized Jaccard's similarity*** results are available in [1]. For aggregation query Q8, the quality is measured using ***normalized root mean square error*** (RMSE). We plot normalized measures as a function of time to emphasize the rate at which the quality of query results are improved across different queries and datasets, instead of actual F_1-measures. Actual F_1-measure varies across different queries based on the quality of classifiers chosen for enrichment (*e.g.*, maximum F_1 measures for different queries were: for Q1 0.73, Q2 0.81, Q3 0.74, Q4 0.89, Q5 0.78, Q6 0.83, Q7 0.72). The minimum RMSE achieved for Q8 was 1.78 starting with an RMSE of 46.18 in the first epoch. From Figures 4.10(a), 4.10(b), and 4.10(c), we observe that both EQ^P_{LC} and EQ^P_{TC} achieve a high-quality improvement within the first few epochs of query execution. The progressive scores achieved for queries (measured using Equation 4.5) are presented in Figure 4.9. EQ^P_{TC} achieves a higher progressive score since the number of redundant enrichments in EQ^P_{TC} is lower than EQ^P_{LC}. Comparing the latency between Table 4.9 and Figure 4.10, we can conclude that the progressive approach of query processing with enrichment can be very beneficial since, the progressive approach achieves high-quality results within a few epochs, without the need for users to wait for complete enrichment.

Exp 3: Effect of Different Plan Generation Strategies. Figure studies different plan generation strategies (as described in §4.2.3.2) and their impact on progressiveness. Figure 4.11 plots progressive improvement of quality for three queries: Q2, Q3, and Q4. Figures

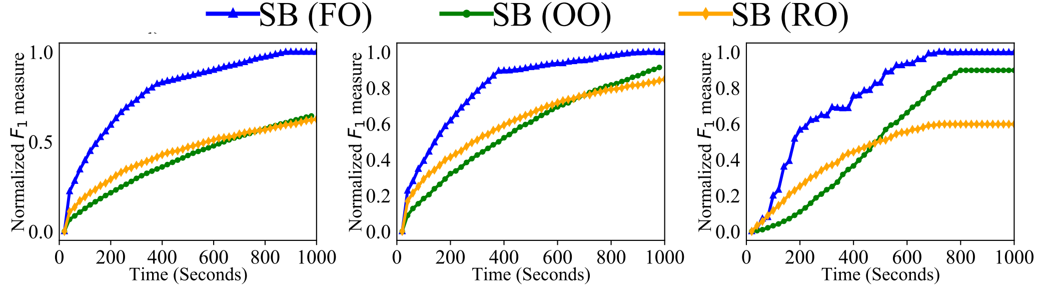


Figure 4.11: Comparing different plan generation strategies in EQ_{TC}: (a) Q2, (b) Q3, (c) Q4 (left to right).

4.11(a), 4.11(b), and 4.11(c) show that SB(FO) performs the best and SB(OO) performs the worst since function order chooses functions with highest quality per unit cost before other functions. SB(RO) performs only marginally better than SB(OO).

4.3.2.3 System Overhead

We measure the overhead incurred by progressive query processing in both EQ_{LC}^P and EQ_{TC}^P approaches.

Exp 4: Time overhead measures the amount of time spent in *non-enrichment tasks* (*i.e.*, query setup, plan selection, delta computation, and state update) to compare against the time involved in data enrichment. Particularly, across all epochs, the total time in query setup, plan selection, delta answer computation, and state update took at most 3s, 4s, 5s, and 17s, respectively, while the total time spent across all epochs in enrichment was 1000s. This result shows that both EQ_{LC}^P and EQ_{TC}^P have low overhead in terms of non-enrichment tasks performed during query processing.

Exp 5: Storage overhead measures the size of all temporary tables (`PlanSpaceTable` and `PlanTable`), IVM, and the state tables used during query processing to compare against the size of data tables. The maximum storage overheads of `PlanSpaceTable`, `PlanTable`, and the IVM at any epoch for the queries of Q1-Q8 were 1.48 MB, 56 KB, and 1.2 MB

respectively. The state table sizes for TweetData and Multi-Pie were 2.4GB and 101MB, respectively, that are much smaller than the data tables. Furthermore, using the state cutoff (§4.2.2) strategy, state storage overhead was reduced significantly. For TweetData table, the state overhead was reduced from 2.4 GB to 0.9 GB, due to large domain size (*i.e.*, 40) of `topic` derived attribute.

Chapter 5

Optimizing Enrichment with Progressive Query Processing

In this chapter, we explore a mechanism of ordering ⟨object, enrichment function⟩ pairs of an enrichment plan (as described in the previous chapter) that optimizes the quality improvement of query results with respect to time. We use the same data model of Chapter 3 to describe the concepts in this chapter. The query execution model is the progressive execution model defined by us in Chapter 4. Our experimental results show that our approach achieves its goal of progressive improvement of quality of answers during query processing.

5.1 Objective

In Chapter 3, we described the data model of EnrichDB that follows an extended relational data model. In this model, some attributes of a relation are fixed (do not require enrichment) and the remaining attributes are derived (that require enrichment). Each derived attribute is associated with a set of enrichment functions.

An *enrichment function* takes as input a tuple t_k and a derived attribute $\mathcal{A}_i$ and outputs a probability distribution over the tags of $\mathcal{A}_i$. In the rest of the chapter we refer to a enrichment function as a *function*. The functions for an attribute $\mathcal{A}_i$ are denoted by $F^i = \{f_1^i, f_2^i, \ldots, f_k^i\}$. Each function f_j^i is associated with a *quality* (denoted by $\mathfrak{q}_j^i$) and a *cost* (denoted by c_j^i). The quality measures the accuracy of the function in enrichment the tag of a derived attribute. The cost of a function represents the average execution cost of the function on a single tuple.

A query (denoted by q) is expressed using SQL as follows:

```
SELECT Tweet,location,TweetTime, topic,
sentiment FROM TweetData WHERE sentiment='positive'
AND topic= 'social media' AND TweetTime
BETWEEN('16:00','18:00');
```

Listing 5.1: A query submitted to EnrichDB.

In the above query, one predicate is on fixed attribute (referred to as fixed predicate) and two are on derived attributes (referred to as derived predicate). As described in Chapter 4, given a query q, we generate a set of probe queries to fetch the minimum set of tuples that require enrichment from each relations in q. The resulted tuples are fetched to the application server in $\mathbb{EQ}_{\mathbb{LC}}$. The $\mathbb{EQ}_{\mathbb{TC}}$ implementation does not fetch any tuples to the application server, however the resultant tuple IDs of probe queries are used for generating the tuples in `PlanSpaceTable`. Given the above query, the probe query for the `TweetData` relation will be as follows:

```
SELECT Tweet,location,TweetTime, topic, sentiment FROM
    TweetData WHERE TweetTime
BETWEEN('16:00','18:00');
```

Listing 5.2: The probe query executed in DBMS.

tid	UserID	Tweet	feature	loc.	Time	topic	sentiment
t_1	John	Uploading pics...	$[0.2, ..., 0.4]$	US	16:08	social media	positive
t_2	Mark	Feeling great...	$[0.5, ..., 0.3]$	US	16:48	entertainment	NULL
t_3	Richard	Sad about current pandemic.	$[0.6, ..., 0.4]$	UK	11:48	NULL	NULL

Table 5.1: `TweetData` table where `topic` and `sentiment` are the derived attributes.

In the following we briefly define the concept of epochs, quality metric of the query answer, and our objective of data enrichment.

Epochs. The epochs are designated for the purpose of performing data enrichment over a lesser number of tuples. Particularly, the query execution time is discretized into different time slots, called epochs, in which data enrichment is performed. At the end of each epoch, the query results are returned to the users. An epoch, denoted by e_i, is identified by a range of time. We assume that each epoch is of equal duration, and it is specified as the *epoch_duration* parameter of the query. Query answer returned at the end of epoch w is referred to as Ans_w. Each query has a parameter, *num_epoch*, that denotes the maximum number of epochs for which the query will run. Query answer returned at the end of epoch w is denoted as Ans_w. Note that, in a progressive query processing strategy, a query answer returned to the user in an earlier epoch, may be retracted in a later epoch due to the result of enrichment.

Measuring Quality. The quality of an answer represents how close the answer is is to the ground truth of G. We use F_α-measure [95] and Jaccard's similarity [65] as the quality metric of a query answer as they are widely used. They combine two other set-based quality metrics of precision and recall. Precision (Pre) is defined as the fraction of correct tuples in Ans_w to the total number of tuples in Ans_w whereas recall (Rec) is defined as the fraction of correct tuples in Ans_w to the total number of tuples in G. The F_α-measure and Jaccard

similarity are computed as follows:

$$
\begin{aligned}
F_\alpha(Ans_w) &= \frac{(1+\alpha)\cdot Pre(Ans_w)\cdot Rec(Ans_w)}{(\alpha\cdot Pre(Ans_w)+Rec(Ans_w))} \\
\mathcal{J}(Ans_w) &= \frac{|Ans_w \cap G|}{|Ans_w \cup G|} = [\frac{1}{Pre(Ans_w)} + \frac{1}{Pre(Ans_w)} - 1]
\end{aligned}
\tag{5.1}
$$

where $Pre(Ans_w) = |Ans_w \cap G|/|Ans_w|$, $Rec(Ans_w) = |Ans_w \cap G|/|G|$, and $\alpha \in [0,1]$ is the weight factor assigned to precision in calculating F_α-measure.

We assume that ground truth values G are not available for the tuples. Hence, our approach measures quality of the answer set in an expected sense. Although, in experiments of §5.6, we plot graphs using actual F_α measure of the answer calculated from available ground truth, our approach never uses it for guiding the enrichment process. It performs enrichment that optimizes the following expected quality measure in each epoch:

$$
\begin{aligned}
&E(Pre) = \frac{\sum_{i=1}^{m}\mathcal{P}_i}{m}, E(Rec) = \frac{\sum_{i=1}^{m}\mathcal{P}_i}{\sum_{j=1}^{n}\mathcal{P}_j}, E(F_\alpha(Ans_w)) = \frac{(1+\alpha)\sum_{i=1}^{m}\mathcal{P}_i}{\alpha\sum_{j=1}^{n}\mathcal{P}_j + m}, \\
&E(\mathcal{J}(Ans_w)) = [\frac{m}{\sum_{i=1}^{n}\mathcal{P}_i} + \frac{\sum_{j=1}^{n}\mathcal{P}_j}{\sum_{i=1}^{m}\mathcal{P}_i} - 1]
\end{aligned}
\tag{5.2}
$$

where, $\mathcal{P}_i$ represents the probability of a tuple t_i to be a part of the real answer set G (*i.e.*, query executed on ground truth), m is the cardinality of Ans_w, and n is the cardinality of the probe query results (described in detail in §5.2). Based on the definition of F_α measure of Equation 5.1, expected F_α measure is calculated by the weighted harmonic mean of precision and recall as shown in Equation 5.2. In order to find Jaccard's similarity between the answer set returned (*i.e.*, Ans_w) and the ground truth set (*i.e.*, G), we utilize the metric of precision and recall as shown Equation 5.1. The expected Jaccard's similarity is presented in Equation 5.2.

Progressive Score. Since in EnrichDB, users may stop query evaluation at any instance of time, performing enrichments that impact the answer quality as early as possible is desirable. EnrichDB's effectiveness is measured using the following progressive score (similar to other progressive approaches [89, 24, 27]):

$$\mathcal{PS}(Ans(q,E)) = \sum_{i=1}^{|E|} W(e_i) \cdot [Qty(Ans(q,e_i)) - Qty(Ans(q,e_{i-1}))] \tag{5.3}$$

where $E = \{e_1, e_2, \ldots, e_k\}$ is a set of epochs, $W(e_i) \in [0,1]$ is the weight allotted to the epoch e_i, $W(e_i) > W(e_{i-1})$), $\mathcal{Q}$ is the quality of answers, and $[\mathcal{Q}(Ans(q,e_i)) - \mathcal{Q}(Ans(q,e_{i-1}))]$ is the improvement in the quality of answers occurred in epoch e_i.

In Equation 5.3, we observe that the optimization of enrichment in each epoch, ensures that the progressive score is optimized for the query. Since the improvements in qualities of epochs are added in the progressive score (using a weighted summation), highest improvement in quality for each epoch ensures that the maximum possible progressive score is achieved.

$$\begin{aligned} &Maximize(\mathcal{PS}(Ans(q,E))) \Rightarrow \\ &Maximize(Qty(Ans(q,e_i)) - Qty(Ans(q,e_{i-1}))) \end{aligned} \tag{5.4}$$

The quality Qty in Equation 5.3, for a set-based query answer can be measured using set-based quality metrics: precision, recall, F_α-measure, or Jaccard similarity coefficient as described earlier. The quality of an aggregation query can be measured using root-mean-square error [63] or mean-absolute-error [114]. For the purpose of enrichment, aggregation queries on derived attributes are treated as a set-based query where the aggregation function is applied on the set of results obtained from the rest of the query evaluation. Estimating

Algorithm 1: Overall Algorithm.

```
   Inputs: Query Q and the duration of each epoch epoch_duration.
   Outputs: An enrichment plan for each epoch that optimizes progressive score.
1  Function Optimize_Enrichment() begin
2      CandidateSet ← ∅
3      Ans ← ∅
4      for each R_i ∈ Q do
5          pq(R_i) ← GenerateProbeQuery(Q, R_i)
6          CandidateSet ← CandidateSet ∪ Execute(pq(R_i))
7      for each epoch e_i do
8          curr_cost ← 0, curr_benefit ← 0
9          Thresholds ← ComputeThreshold(Q, Ans)
10         CandidateSet ← PruneTuples(CandidateSet)
11         for all C ∈ CandidateSet do
12             C.benefit ← estimate_benefit(Q, C)
13         Sort_{Benefit/Cost}(CandidateSet)
14         while curr_cost ≤ epoch_duration do
15             ⟨tuple, function⟩ ← CandidateSet.pop()
16             ExecuteEnrichment(tuple, function)
17         Ans ← UpdateQueryResult(Q)
18     Return Ans
```

the above quality metrics for queries on probabilistic data is not straight-forward. In later sections, we describe how to estimate these quality metrics.

Problem Statement. For a given query Q, where a set of ⟨tuple, enrichment function⟩ pairs that are not yet executed till the previous epochs of $e_1, e_2, \ldots, e_{w-1}$, the objective in epoch e_w is to maximize the improvement in quality as shown in Equation 5.4.

5.2 Overview of the Algorithm

The problem of optimizing progressive score for a query, is a budgeted Knapsack problem, which is NP-hard [37]. We use a greedy approach for solving this problem. In each epoch, this approach chooses a set of tuples for enrichment from each relation. For each tuple, it chooses a derived attribute to enrich for which a condition (selection or join) is present

in query Q. Next, for each tuple and derived attribute pair, the algorithm chooses an enrichment function that is based on reduction of uncertainty of the attribute value of the tuple. Finally, the generated ⟨tuple, attribute, enrichment function⟩ triplets are ordered based on their capability of improving the quality of the query result from previous epoch.

The complete algorithm for generating and executing an enrichment plan during query processing is shown in Algorithm 1. In the zero-th epoch (Line 2-6), the approach first determines a minimal set of objects to enrich. Naive way all can be enriched. executes a set of *probe queries* (will be described in detail later), one for each relation present in query, that identifies a minimal subset of tuples from each relations that may require enrichment based on the predicates on fixed attributes. The algorithm for probe query generation is presented in the previous section of §4.1.1. The later epochs (*i.e.*, $e_1, e_2, \ldots e_z$) consist of four steps: candidate tuple set selection (Line 10), computing benefit of the chosen tuples and enrichment functions (Line 11 - 12), selection and execution of an enrichment plan (Lines 13-16), and generation of the query result (Line 17).

5.3 Candidate Tuple Set Selection

Given a (super)set of tuples that may need enrichment – (*i.e.*, result of probe queries) – the goal is to select the right set of tuples to enrich in each epoch. The result of probe queries can still be large depending on the selectivity of the predicates on fixed attributes. The system uses a greedy algorithm to choose tuples for enrichment, that did not contribute to the answer set in the previous epoch. For this purpose, the algorithm determines a threshold for each relation present in the query Q. The purpose of the threshold selection is two-fold: (i) choosing an answer set to be returned to the user at the zero-th epoch and at the end of each epoch based on enrichment performed and (ii) prune the tuples of each relation further that are considered for enrichment, from the results of probe queries. Below, we describe

the threshold selection algorithm.

5.3.1 Choosing Thresholds for Each Relation.

The selection of the threshold for a relation at a particular epoch is based on the following theorem.

Theorem 5.1. *Let L be the list of tuples sorted in a decreasing order of their probability values in satisfying the selection condition on derived attributes and let L^k be the list that consists of the first k tuples in L. The expected quality of the possible subsets of L follows a monotonically increasing pattern with respect to k, up to a certain value of k and beyond that it decreases monotonically, i.e., it follows the pattern: $E(Qty(L^1)) < E(Qty(L^2)) < \ldots < E(Qty(L^{\tau-1})) > E(Qty(L^{\tau})) > E(Qty(L^{\tau+1})) > \ldots > E(Qty(L))$.*

Proof. We prove this theorem by using F_α-measure as the quality metric of the answer set. We show that if $E(F_\alpha)$ measure of the answer decreases for the first time, due to the inclusion of a particular tuple in answer set, then it keeps decreasing monotonically with the inclusion of any further tuple.

Let us denote the $E(F_\alpha)$ measure of the answer set, if τ-th tuple is included in the answer set as F_τ. Similarly, the $E(F_\alpha)$ measures corresponding to the inclusion of $\tau+1$-th and $\tau+2$-th tuple are denoted as $F_{\tau+1}$ and $F_{\tau+2}$ respectively. We show that for a particular value of τ, if $F_{\tau+1} < F_\tau$, then it implies $F_{\tau+2} < F_{\tau+1}$.

$$F_\tau = \frac{(1+\alpha).\frac{k_1}{\tau}.\frac{k_1}{k_2}}{\alpha \cdot \frac{k_1}{\tau} + \frac{k_1}{k_2}} = \frac{(1+\alpha)\cdot k_1}{\alpha \cdot k_2 + \tau}, k_1 = \sum_{i=1}^{\tau} \mathcal{P}_i, k_2 = \sum_{i=1}^{n} \mathcal{P}_i \tag{5.5}$$

Similarly, the values of $F_{\tau+1}$ and $F_{\tau+2}$ are as follows:

$$F_{\tau+1} = \frac{(1+\alpha)(k_1+\mathcal{P}_{\tau+1})}{(\alpha k_2+\tau+1)}, F_{\tau+2} = \frac{(1+\alpha)(k_1+ \mathcal{P}_{\tau+1}+\mathcal{P}_{\tau+2})}{(\alpha k_2+\tau+2)} \tag{5.6}$$

$$\begin{aligned} F_{\tau+1} < F_{\tau} &\Rightarrow \frac{(1+\alpha)\cdot(k_1+\mathcal{P}_{\tau+1})}{(\alpha k_2+\tau+1)} < \frac{(1+\alpha)\cdot k_1}{\alpha k_2+\tau} \\ &\Rightarrow (k_1+\mathcal{P}_{\tau+1})(\alpha k_2+\tau) < k_1(\alpha k_2+\tau+1) \\ &\Rightarrow \alpha k_1 k_2 + k_1\tau + \alpha k_2 \mathcal{P}_{\tau+1} + \tau\mathcal{P}_{\tau+1} < \alpha k_1 k_2 + k_1\tau + k_1 \end{aligned} \tag{5.7}$$

Simplifying some more steps, we derive the following condition: $\frac{(k_1+\mathcal{P}_{\tau+1}+\mathcal{P}_{\tau+2})}{(\alpha k_2+\tau+2)} < \frac{(k_1+\mathcal{P}_{\tau+1})}{\alpha k_2+\tau+1}$, *i.e.*, $F_{\tau+2} < F_{\tau+1}$. □

Based on above theorem, our approach only adds the tuples that have probability higher than the threshold probability to the answer set of an epoch. All the tuples with probability lower than the threshold probability are considered for enrichment and hence, added to the *CandidateTuple* set of the epoch. The reason behind this selection is formalized in the following theorem.

Theorem 5.2. *Enriching a tuple t_k corresponding to a composite-tuple that was not part of the answer set in the previous epoch (i.e., in Ans_{w-1}) ensures that the quality of the answer set increases with respect to previous epoch. That is, $E(Qty(Ans_w)) >= E(Qty(Ans_{w-1}))$ irrespective of the outcome of enrichment.*

Proof Sketch. For a tuple that was not part of any answer-tuple of the previous epoch of $(w-1)$, the execution of a triple containing the tuple can cause either an increment or decrement of its probability value of satisfying the selection conditions. In the scenario where it increases, it can either become higher than the probability value of the threshold tuple of

Derived Attribute	**State**	**Entropy Ranges**	**Next Function**	**Δ Uncertainty**
`sentiment`	[1, 0, 0, 0]	[0 - 0.1), [0.1-0.2), $\cdots$,[0.8-0.9), [0.9-1]	f_2^1, f_4^1, $\cdots$,f_3^1, f_3^1	-0.04, -0.10, $\cdots$, -0.2, -0.26
`sentiment`	[1, 1, 0, 0]	[0 - 0.25), [0.25-0.5), [0.5-0.75), [0.75-1]	f_3^1, f_4^1, $\cdots$, f_3^1	-0.03, -0.11, -0.15,-0.18
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
`topic`	[1, 0, 0, 0]	[0 - 0.25), [0.25-0.5), [0.5-0.75), [0.75-1]	f_2^2, f_4^2, f_4^2,f_3^2	-0.04, -0.12, -0.16, -0.22
`topic`	[0, 0, 1, 1]	[0 - 0.1), [0.1-0.2), $\cdots$, [0.9-1]	f_1^2, f_1^2, $\cdots$, f_2^2	-0.02, -0.11, $\cdots$, -0.28

Table 5.2: `DecisionTable`.

the previous epoch or stay lower than it. In both the cases, the quality of the answer (*i.e.*, $E(Qty(Ans_w))$ measured using Equation 5.2) increases or remains the same with respect to previous epoch (*i.e.*, $E(Qty(Ans_{w-1}))$). Similarly, when the probability value of a tuple decreases, the $E(Qty(Ans_w))$ value stays the same or it increases, depending on the amount of decrement. The complete proof of this theorem is available in Appendix B.

5.4 Benefit Estimation

The algorithm chooses the ⟨object, enrichment function⟩ pairs as candidate set, based on *benefit*, per unit cost. Benefit, discussed formally below, corresponds to expected improvement in quality of the answers compared to the quality of results in the previous epoch. We restrict the choice of objects to enrich to only those that are not in the answer set to reduce complexity of repeatedly computing benefits for objects that have already appeared in the answer. We do so, since the expected benefit by further enriching an object that already appears in the answer is significantly lower compared to those that are not in the answer. Later in the paper we will formally justify this decision and will, furthermore, show performance improvement due to reduced overhead of computing benefit in the experiments.

Given the database state at the end of epoch e_{w-1}, the query is executed on the database

to produce the result of Ans_{w-1}. The quality is measured for the answer set Ans_{w-1}. Let f_m be an enrichment function that is not yet executed on tuple t_k in the previous epochs. The benefit of executing function f_m in epoch e_w can be computed as the improvement in the quality of the query result if we were to execute f_m on t_k in the current epoch of e_w. In particular, let $State(D, e_{w-1})$ be the state of the database at the end of epoch e_{w-1} and $(state(D, e_{w-1}) \oplus (t_k, f_m))$ be the expected state of the database after execution of f_m on t_k. If the query Q is executed on this new state of the database, the resulting improvement in the quality of the answer set from the previous epoch is used to measure the benefit of enriching t_k with f_m. Specifically, the benefit of enrichment is defined as follows:

$$\begin{aligned} Benefit(t_k, \mathcal{A}_l, f_m) = E(Qty(Q(state(D, e_{w-1}) \oplus (t_k, f_m))))- \\ E(Qty(Q(state(D, e_{w-1})))) \end{aligned} \tag{5.8}$$

where $E(Qty(Q(state(D, e_{w-1}) \oplus (t_k, f_m))))$ is the expected quality of the query answer if the tuple t_k is enriched using the enrichment function f_m and $E(Qty(Q(state(D, e_{w-1}))))$ is the quality of the query result at the end of previous epoch.

Thus, to determine the benefit of enriching a tuple, we need to estimate (i) the quality of the answer after epoch e_{w-1} and (ii) the expected quality of the answer set if that tuple is enriched in the current epoch. Before we consider how we can estimate these two metrics for general queries, we first describe how to estimate them for selection queries.

5.4.1 Selection Queries

The estimation of quality of the answer in the previous epoch is performed as follows:

Estimating Quality of Answer. The algorithm calculates the probability of the tuples of a relation to be part of the query result. If the probability of the tuples that were part of the answer in previous epoch were $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_m$, and the probability of the tuples outside of the answer were $\mathcal{P}_{m+1}, \mathcal{P}_{m+2}, \ldots, \mathcal{P}_n$, then expected F_1 measure of the answer (*i.e.*, using Equation 5.2) can be calculated as follows: $E(F_\alpha(Ans_{w-1})) = \frac{(1+\alpha)\sum_{i=1}^{m}\mathcal{P}_i}{\alpha\sum_{j=1}^{n}\mathcal{P}_j + m}$. In this equation, precision is calculated by the summation of probabilities of tuples that are part of the answer set and the size of the answer set. The recall is calculated by the ratio of summation of probabilities of all tuples that are part of answer set to the probabilities of all the tuples that are part of the probe query result. Using the harmonic mean of precision and recall, the $E(F_\alpha(Ans_{w-1}))$ of Ans_{w-1} is $E(F_\alpha(Ans_{w-1})) = \frac{(1+\alpha)\sum_{i=1}^{m}\mathcal{P}_i}{\alpha\sum_{j=1}^{n}\mathcal{P}_j + m}$.

Example. Consider the selection query as shown in Code Listing 5.1 on `TweetData` table (as in Table 5.1). Suppose at the end of epoch e_{w-1}, the tuples t_1 and t_2 were part of the query result. Since, t_1 has a probability of 0.54 to satisfy the query condition on `topic` and a probability of 0.94 to have a `sentiment` of `positive`, the combined probability of the tuple satisfying the query is $(0.54 \times 0.94) = 0.51$. Similarly, the probability of t_2 to satisfy the query is $(0.65 \times 0.5) = 0.325$. The expected precision of the answer is calculated as follows: $(0.51 + 0.32)/2 = 0.415$. The recall calculation requires probability of the tuples that are part of the answer as well as of the tuples that are outside of the answer. The probability of tuple t_3 to be part of the query result is $(0.4 \times 0.7) = 0.28$.

The algorithm for computing the benefit of executing a function that is not yet executed on the tuple is presented in Algorithm 2. The input to the algorithm is a set of ⟨tuple, derived attribute, enrichment function⟩ triples. Estimating the quality improvement after executing an enrichment function on a tuple, the algorithm uses the information about the amount of uncertainty reduction caused by the enrichment function. As a first step, the algorithm computes the uncertainty of the tuple in epoch e_{w-1} (say $\mathcal{E}_{w-1}$) in the derived attribute value

Algorithm 2: Benefit Calculation.

Inputs: A tuple of a relation, a derived attribute, the next best function for the tuple at that state of the attribute.

Outputs: The benefit of the $\langle$tuple t_i, derived attribute $\mathcal{A}_j$, enrichment function $f_k\rangle$ triplet.

1 Function *Estimate_Benefit*() **begin**
2 $PreviousQuality \leftarrow ComputeQuality(Ans_{w-1})$
3 $PreviousEntropy \leftarrow ComputeEntropy(t_i, \mathcal{A}_j)$
4 $ExpectedEntropy \leftarrow (PreviousEntropy - ComputeDeltaEntropy(t_i, f_j))$
5 $SelectionProbability \leftarrow ComputeInverseOfEntropy(ExpectedEntropy)$
6 $ExpectedAnswerQuality \leftarrow ComputeQuality(SelectionProbability, Ans_{w-1})$
7 $Benefit(t_l, f_j) \leftarrow (ExpectedAnswerQuality - PreviousQuality)$
8 Return $Benefit(t_l, f_j)$

of the tuple and finds out the expected amount of uncertainty reduction that will be caused by executing the next best enrichment function, say Δ_w. From the new entropy value, the expected probability of the tuple satisfying the query is computed using inverse equation of entropy.

$$\mathcal{E}_{w-1} - \Delta_w = -p \cdot log(p) - (1-p) \cdot log(1-p) \tag{5.9}$$

Note that the above equation of inverse entropy for finding out probability values, has two possible solutions. After the actual execution of enrichment function, the probability of the tuple may increase or decrease as compared to the previous epoch. While estimating the quality of the query result, we use the estimated probability of the tuple which is higher than the current value, as that probability contributes to the quality of query result.

Example. Consider the state of tuple with $tid = 2$ in Table 4.2. Since the tuple's state for `sentiment` derived attribute is [1,1,0] (*i.e.*, functions f_1^i and f_2^i already executed) and the uncertainty is 0.92 (computed from the values in `SentimentStateOutput`),[1] our approach

[1]Combining the output of functions using an average based combiner produce the following outputs: [(0.3+0.5)/2, (0.4+0.3)/2, (0.3+0.2)/2], *i.e.*, [0.4,0.35, 0.25]. Calculating entropy $h(t_k, \mathcal{A}_p)$ using the equation of $h(t_k, \mathcal{A}_p) = -\sum_i p_i \cdot log(p_i)$, will produce the output of 0.92.

will retrieve the corresponding record from decision table using the entropy range of $[0.75-1]$. The next function to be executed on the tuple is f_4^i and the expected decrease in entropy is 0.28. The new entropy of the tuple with respect to the derived attribute is $(0.92 - 0.28) = 0.64$.

Estimating Improvement in Quality. Using the new probability value of the tuple if it is enriched in the current epoch and considering that the probability of all the remaining tuples remain the same, we compute the new quality of the answer set. The benefit of enrichment of the tuple is measured by Equation 5.2 and Equation 5.8. While measuring this quality, the algorithm needs to measure the new threshold probability to determine which tuples remain in the answer and then compute the $E(F_\alpha)$-measure of the answer set. This way of computing benefit has a time complexity of $\mathcal{O}(n^2 log(n))$ for a tuple, since the algorithm needs to sort all the tuples based on their probability values $(\mathcal{O}(nlog(n)))$ and then find out the threshold value using a linear scan $(O(n))$, where n is the number of tuples in the relation. Hence, the time complexity of computing the benefit of a single tuple takes $(\mathcal{O}(nlog(n)+n))$ time. Hence, for computing the benefit of n tuples, the time complexity of $\mathcal{O}(n(nlog(n)+n))$, *i.e.*, $\mathcal{O}(n^2 log(n))$. We reduce the time complexity of this step by deriving a new metric of *RelativeBenefit* to order the tuples in linear time. According to this metric, if one tuple has higher relative benefit than another tuple, then the first tuple will always have higher benefit according to Equation 5.8, irrespective of the probability values of the tuple (in epoch e_{w-1} or in epoch e_w) and whether such probabilities affect the threshold of the answer.

Our approach calculates a *RelativeBenefit* value for each triple $(t_k,\ \mathcal{A}_l,\ f_m)$ in epoch w as follows:

$$RelativeBenefit(t_k, \mathcal{A}_l, f_m) = \frac{\mathcal{P}_k(\mathcal{P}_k + \Delta\mathcal{P}_k)}{c_m} \tag{5.10}$$

where $\mathcal{P}_k$ is the probability of the tuple satisfying the selection conditions and $\mathcal{P}_k + \Delta\mathcal{P}_k$ is the new probability of tuple t_k if it is enriched in the current epoch.

We derive the above metric of computing relative benefit that is independent of the previous probability of a tuple, the expected amount of probability increment in the current epoch, based on the following theorem.

Theorem 5.3. *A triple (t_k, $\mathcal{A}_l$, f_m) has higher benefit than a triple of (t_q, $\mathcal{A}_t$, f_v) in epoch w irrespective of the values of $\mathcal{P}_k$, $\mathcal{P}_q$, $\Delta\mathcal{P}_k$ and $\Delta\mathcal{P}_q$, if the following condition is satisfied:*

$$\gamma_k \cdot \frac{\mathcal{P}_k(\mathcal{P}_k + \Delta\mathcal{P}_k)}{c_m} > \gamma_q \cdot \frac{\mathcal{P}_q(\mathcal{P}_q + \Delta\mathcal{P}_q)}{c_v} \tag{5.11}$$

Proof. We prove this theorem as follows: for a given values of each $\mathcal{P}_k$, $\mathcal{P}_q$, $\Delta\mathcal{P}_k$ and $\Delta\mathcal{P}_q$, there can be four possible orders among them. They are as follows: (i) $\mathcal{P}_k > \mathcal{P}_q$ and $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q$, ($ii$) $\mathcal{P}_k > \mathcal{P}_q$ and $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$, ($iii$) $\mathcal{P}_k < \mathcal{P}_q$ and $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q$, ($iv$) $\mathcal{P}_k < \mathcal{P}_q$ and $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$. In each of these orders, we determine the answer set by calculating the new threshold probability value. Once the answer set is determined, the quality of the answer set is calculated separately when the triple (t_k, R_l^i, f_m^i) and the triple (t_q, R_t^s, f_v^s) is executed. We measure their improvement in F_1 measure using Equation 5.2. By simplifying the equation, it is observed that the first triple will have higher benefit than the second one when the condition in Equation 5.11 is satisfied. Below, we show the derivation for each of the scenarios described above.

Suppose m_1 is the number of tuples that are part of Ans_{w-1} and moves out of Ans_w as a result of changing the probability of the tuple t_k from $\mathcal{P}_k$ to $\hat{\mathcal{P}}_k$, where $m_1 \geq 0$ (as shown in Lemma 4). Furthermore, suppose $m_2 \geq 0$ is the number of tuples that are part of Ans_{w-1} and moves out of Ans_w as a result of changing the probability of the tuple t_q from $\mathcal{P}_q$ to $\hat{\mathcal{P}}_q$.

The possible values of m_1 and m_2 can be as follows. Both m_1 and m_2 can be greater than zero and m_1 is greater than m_2. Both of m_1 and m_2 can be greater than zero and m_1 is less than or equal to m_2, or both m_1 and m_2 are equal to zero which implies that the number of tuple in both Ans_{w-1} and Ans_w are the same. Given these three cases, we make the following observations about the benefit values of the triples in TS_w.

Given three possible cases of m_1 and m_2, we consider the possible combinations (16 possible combinations) of the values of $\mathcal{P}_k$, $\mathcal{P}_q$, $\Delta\mathcal{P}_k$, and $\Delta\mathcal{P}_q$ and show if Equation 5.11 holds, then the benefit of (t_k, R^i_l, f^i_m) will be higher than the benefit of (t_q, R^s_t, f^s_v). In the following we provide the proof of these scenarios:

For ease of notation, we denote the threshold $\mathcal{P}^{\tau}_{w-1}$ of epoch $w-1$ as $\mathcal{P}_\tau$ in this proof. Since, $E(F_\alpha(Ans_{w-1})) = \frac{(1+\alpha)(\mathcal{P}_1+\cdots+\mathcal{P}_\tau)}{\alpha(\mathcal{P}_1+\mathcal{P}_2+\cdots+\mathcal{P}_{|O|})+\tau}$, the notation can be simplified as $\frac{X}{Y+\tau}$. We denote the value of $\frac{\mathcal{P}_k}{c^l_n}$ by ν_k and the value of $\frac{\mathcal{P}_q}{c^s_v}$ by ν_q.

Simplifying the expression of benefit the triples, benefit value of triple (t_k, R^l_m, f^l_n) will be higher than the triple (t_q, R^s_t, f^s_v), when the following condition holds:

$$\begin{aligned}
&\nu_k\Big(\frac{X-(1+\alpha)\cdot(\mathcal{P}_\tau+\mathcal{P}_{\tau-1}+...\mathcal{P}_{\tau-(m_1-1)})+(1+\alpha)\cdot(\mathcal{P}_k+\Delta\mathcal{P}_k)}{Y+(\tau-m_1)+\alpha\cdot\Delta\mathcal{P}_k}\Big) > \\
&\nu_q\Big(\frac{X-(1+\alpha)\cdot(\mathcal{P}_\tau+\mathcal{P}_{\tau-1}+...\mathcal{P}_{\tau-(m_2-1)})+(1+\alpha)\cdot(\mathcal{P}_q+\Delta\mathcal{P}_q)}{Y+(\tau-m_2)+\alpha\cdot\Delta\mathcal{P}_q}\Big)
\end{aligned} \tag{5.12}$$

Case 1: $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$**,** $\mathcal{P}_k+\Delta\mathcal{P}_k > \mathcal{P}_q+\Delta\mathcal{P}_q$**, and** $m_1 > m_2$.

Comparing both the denominators of Equation 5.12, we can see that $(\tau-m_1) < (\tau-m_2)$ and $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$. This implies that the denominator on the L.H.S. is smaller than the denominator on the R.H.S. In the numerator of L.H.S., the value of $(\mathcal{P}_\tau+\mathcal{P}_{\tau-1}+...\mathcal{P}_{\tau-(m_1-1)})$ is higher than $(\mathcal{P}_\tau+\mathcal{P}_{\tau-1}+...\mathcal{P}_{\tau-(m_2-1)})$ as m_1 is higher than m_2. Furthermore, if $\nu_k(\mathcal{P}_k+\Delta\mathcal{P}_k) > \nu_q(\mathcal{P}_q+\Delta\mathcal{P}_q)$ then the numerator of the L.H.S. will be higher than the numerator of

the R.H.S. Thus we conclude that Equation 5.12 is satisfied when condition $\nu_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > \nu_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ is satisfied.

Case 2: $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q$**,** $\mathcal{P}_k + \Delta\mathcal{P}_k > \mathcal{P}_q + \Delta\mathcal{P}_q$**, and** $m_1 > m_2$. In Equation 5.12, the value of $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)})$ on the L.H.S is higher than $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)})$ as m_1 is higher than m_2. In the denominator, although the value of $\Delta\mathcal{P}_k$ is higher than $\Delta\mathcal{P}_q$, the total value of $(\tau - m_1 + \alpha \cdot \Delta\mathcal{P}_k)$ is lower than $(\tau - m_2 + \alpha \cdot \Delta\mathcal{P}_q)$ as both $\Delta\mathcal{P}_k$ and $\Delta\mathcal{P}_q$ are less than one.

Case 3: $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q$**,** $\mathcal{P}_k + \Delta\mathcal{P}_k < \mathcal{P}_q + \Delta\mathcal{P}_q$**, and** $m_1, m_2 = 0$. Let us compare the L.H.S and R.H.S. of Equation 5.12. In the numerator, if the term of $\nu_k(\mathcal{P}_k + \Delta\mathcal{P}_k)$ is higher than the value of $\nu_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$, then the numerator of L.H.S will be higher than the numerator of R.H.S. Hence, the value of the expression in the left-hand side will be higher.

Case 4: $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$**,** $\mathcal{P}_k + \Delta\mathcal{P}_k < \mathcal{P}_q + \Delta\mathcal{P}_q$**, and** $m_1, m_2 = 0$. In Equation 5.12, after simplifying some steps further, we derive that the condition in which the L.H.S. will be higher than the R.H.S. is as follows: $\nu_k(\mathcal{P}_k + \Delta\mathcal{P}_k)\Delta\mathcal{P}_q > \nu_q(\mathcal{P}_q + \Delta\mathcal{P}_q)\Delta\mathcal{P}_k$. According to the assumption $\Delta\mathcal{P}_q$ value is higher than the value of $\Delta\mathcal{P}_k$. This implies that, if the condition $\nu_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > \nu_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ is satisfied, then L.H.S. will be higher than the R.H.S.

The above proofs will also hold for the scenarios where $m_1 = m_2$ and $m_1 > 0$. Only difference will be as follows: an additional constant term (*i.e.*, $\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + \cdots + \mathcal{P}_{\tau-(m_1-1)}$) will be added to the numerators of both the sides of Equation 5.12. The remaining steps will remain the same as the proofs of Cases 1-4.

Case 5: $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$**,** $\mathcal{P}_k + \Delta\mathcal{P}_k > \mathcal{P}_q + \Delta\mathcal{P}_q$**, and** $m_1 < m_2$. Comparing both the denominators of Equation 5.12, we can see that $(\tau - m_1) < (\tau - m_2)$ and $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$. This implies that the denominator of L.H.S. is lower than the denominator of R.H.S. In the numerators, the value of $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)})$ is lower than $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)})$

as m_1 is less than m_2. This condition makes the numerator of L.H.S higher than R.H.S. Furthermore, if $\nu_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > \nu_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ is satisfied then the numerator of L.H.S. is higher than R.H.S.

Case 6: $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q$**,** $\mathcal{P}_k + \Delta\mathcal{P}_k > \mathcal{P}_q + \Delta\mathcal{P}_q$**, and** $m_1 < m_2$. From Equation 5.12, we can derive the following equation:

$$\begin{aligned}
&\nu_k(X - (1+\alpha)\cdot(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)}) + (1+\alpha)\cdot \\
&(\mathcal{P}_k + \Delta\mathcal{P}_k))\cdot(Y + (\tau - m_2) + \alpha\cdot\Delta\mathcal{P}_q) > \nu_j(X- \\
&(1+\alpha)\cdot(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)}) + (1+\alpha)\cdot \\
&(\mathcal{P}_q + \Delta\mathcal{P}_q))\cdot(Y + (\tau - m_1) + \alpha\cdot\Delta\mathcal{P}_k)
\end{aligned} \tag{5.13}$$

In the above equation, the value of $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_1-1)})$ is lower than $(\mathcal{P}_\tau + \mathcal{P}_{\tau-1} + ...\mathcal{P}_{\tau-(m_2-1)})$ as m_1 is smaller than m_2. This favors the value in the left-hand side of the equation. Furthermore, if the value of $\nu_k(\mathcal{P}_k+\Delta\mathcal{P}_k)$ is higher than the value of $\nu_q(\mathcal{P}_q+\Delta\mathcal{P}_q)$, then the benefit of the first triple will be higher than the second triple.

Case 7: $\Delta\mathcal{P}_k > \Delta\mathcal{P}_q$**,** $\mathcal{P}_k + \Delta\mathcal{P}_k < \mathcal{P}_q + \Delta\mathcal{P}_q$**, and** $m_1 < m_2$. Comparing the L.H.S. of Equation 5.13 with R.H.S. of the equation, we observe that if the value of $\nu_k(\mathcal{P}_k + \Delta\mathcal{P}_k)$ is higher than the value of $(\mathcal{P}_q + \Delta\mathcal{P}_q)$, then the whole expression of L.H.S. becomes higher and hence Equation 5.12 is satisfied.

Case 8: $\Delta\mathcal{P}_k < \Delta\mathcal{P}_q$**,** $\mathcal{P}_k + \Delta\mathcal{P}_k < \mathcal{P}_q + \Delta\mathcal{P}_q$**, and** $m_1 < m_2$. Let us consider Equation 5.12 and compare the L.H.S. with the R.H.S. After simplifying them, we can derive that the condition in which the left-hand side will be higher than the right-hand side is as follows: $\nu_k(\mathcal{P}_k + \Delta\mathcal{P}_k)\Delta\mathcal{P}_q > \nu_q(\mathcal{P}_q + \Delta\mathcal{P}_q)\Delta\mathcal{P}_k$. According to the assumption of this case, $\Delta\mathcal{P}_q$ value is higher than the value of $\Delta\mathcal{P}_k$. This implies that, if the condition $\nu_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > \nu_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ holds, then Equation 5.12 is satisfied and the benefit of first triple is higher than the second triple.

The above proofs (*i.e.*, the proofs of Cases 5-8) will also hold for the scenarios where $m_1 > m_2$, due to the symmetric nature of the assumptions. Based on the proofs of Cases 1-8, we conclude that given two triples (t_k, R_m^l, f_n^l) and (t_q, R_t^s, f_v^s), if $\nu_k(\mathcal{P}_k + \Delta\mathcal{P}_k) > \nu_q(\mathcal{P}_q + \Delta\mathcal{P}_q)$ is satisfied then the first triple has higher benefit then the second triple.

□

5.4.2 Generalizing to Other Queries

To estimate benefit computation to general queries, we have to extend the model for both estimating the expected quality of the query result in the previous epoch and also the benefit of executing the next best function on tuples in the current epoch. Let us consider a query Q with conditions on n relations of R_1, R_2, ..., R_n. For each R_i, there could be selection conditions on derived and fixed attributes and then there can be multiple join conditions with other relations.

At any epoch, the tuples of R_i could be classified as one of the two types: (i) the tuples who have met the selection condition on R_i, denoted by R_i^σ and (ii) the tuples who have not met such conditions, denoted by $R_i^{\neg\sigma}$. Of the tuples that are in R_i^σ, we further classify such tuples to be part of the answer set and not part of the answer set based on if there exists a tuple in answer of the query so far. Initially, none of the tuples are part of the query answer.

We first define the probability of the tuple satisfying the selection conditions of the query. This probability is based on the output of previous enrichment functions and their combined output as described earlier for selection queries. Next, we estimate the size of the result-tuples in the query result generated by a tuple that is in R_i^σ in the current epoch. We estimate this value by computing the ratio of the result-tuple size by the tuples in R_i^σ to the size of the query result in the previous epoch. This ratio signifies the average number of

result-tuples that are generated from a tuple in R_i^{σ}.

Also, for each result-tuple in the query answer A, we generate a probability of that result tuple to be part of the ground truth result. This probability is based on the probability of the tuple satisfying individual predicates of P_1, P_2, ..., P_n, of the query. Each predicate can be a selection condition or a join condition on the derived attributes. The computation of probability for selection conditions is performed the same way as we described for selection queries as described above. The computation of probability for a join condition is based on choosing the highest probability in the distribution of the derived attribute value of the tuple. By computing the probability of the tuples in the answer, we can compute the precision of the query result by simply using Equation 5.2. Measuring recall is harder since it requires the sum of probability of all the tuples that are part of the answer as well as that are not part of it. The sum of probability of the tuples that are in the answer is computed the same way as precision calculation. The sum for tuples outside of the answer is computed by maintaining the sum of the selection probabilities of the tuples in R_i^{σ} and in $R_i^{\neg\sigma}$. The sum of probabilities of all the tuples that were part of the answer and not part of the answer, is simply the product of probability sums of the individual relations.

The tuples of a relation R_i that are enriched are chosen from the set of $R_i^{\neg\sigma}$. The benefit computation has two components: (i) if a tuple is enriched with respect to a derived attribute present in the selection condition, how much probability of the tuple will be improved and (ii) if a tuple in $R_i^{\neg\sigma}$ is pushed up the query tree, how many extra answer-tuples it will produce with respect to the answer of previous epoch. The first step is performed the same way as the selection queries as described previously in §5.4.1. The second step is performed by calculating the total number of tuples that are in the answer set of e_{w-1} that contained the tuples from the set of R_i^{σ}. This provides an estimated number of answer tuples that will be added to the answer set of epoch e_w.

5.5 Enrichment Plan Selection

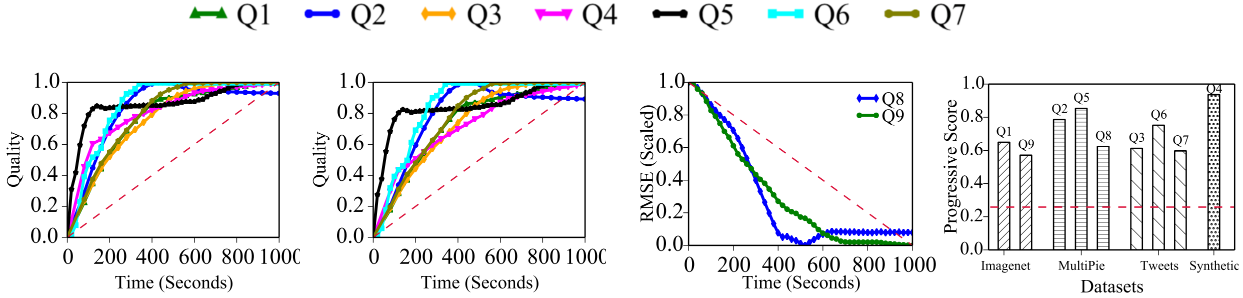
This step chooses a set of ⟨ tuple, enrichment function⟩ pairs as the enrichment plan of the epoch. The problem of selecting an enrichment plan is a budgeted Knapsack problem. The algorithm needs to choose a set of ⟨tuple, enrichment function⟩ with highest summation of *RelativeBenefit* value and a total cost not exceeding the duration of the epoch. We use a greedy approach to choose this enrihcment plan for an epoch. The ⟨ tuple, enrichment function⟩ pairs generated, as described in the previous step, are sorted based on their *RelativeBenefit* values.

Since, the pairs are ordered based on the *RelativeBenefit* metric as descirbed in the previous section, total cost of an enrichment plan is less than or equal to *epoch_duration* with the maximum sum of benefit values among all possible subset of ranked tuples. This problem is a budgeted Knapsack problem. We use a greedy approach to solve this problem. The triples in TS_w are sorted in decreasing order of their benefit values. Enrichment plan is chosen from the sorted set starting from the triple with highest benefit. Once the running total cost of the chosen triples exceeds *epoch_duration*, the WSPT algorithm terminates, and the chosen set is considered as the enrichment plan.

5.6 Experimental Evaluation

This section empirically evaluates our approach using three real datasets and real enrichment functions for different derived attributes of the datasets. We implemented this approach in EQ_{TC} implementation. In these experiments, we want to address following questions:

- How does the benefit-based approach perform as compared to the traditional sampling-based approaches of enrichment plan generation?



(a): F_1 measure. (b): Jaccard's similarity. (c): RMSE. (d): Progressive Score.

Figure 5.1: Progressiveness Achieved. The dotted line shows a possible incremental strategy producing query answers using server-side cursors.

Relation	#tuples	Size(GB)	Derived attrs.	Functions used
TweetData	11M	10.5	sentiment(3)	GNB,KNN,SVM,MLP
			topic(40)	GNB,KNN,LDA,LR
ImageNet[46]	100K	19	ObjectClass(100)	DT,GNB,RF,MLP
MultiPie[104]	100K	16.9	gender(2), expression(5)	DT,GNB,KNN,MLP
SyntheticData	100M	35	$\mathcal{A}_1(5)$	f_1,f_2, ... , f_{10}.

Table 5.3: Datasets used in experiments.

- What are the overhead of benefit-based approaches?
- How does the benefit-based approach perform when the data size, query selectivity, number of enrichment functions are varied?
- How does the benefit-based approach perform when some of the previous enrichment results are already cached?

5.6.1 Experimental Setup

Datasets. We used four datasets (see Table 5.3) to evaluate the performance of EnrichDB. The datasets corresponded to:(*i*) TweetData collected using Twitter APIs containing 11 million rows (size = 10.5 GB) and derived attributes of `sentiment` (dom. size = 3) and `topic` (dom. size =40), (*ii*) ImageNet [46] dataset consisting of 100K images of objects (size = 19 GB) and derived attributes of `ObjectClass` (dom. size =100), (*iii*) MultiPie [104] dataset contains 100K facial images (size = 16.9 GB) with derived attributes `gender` (dom.

size = 2) and `expression` (dom. size = 5), and (*iv*) a large synthetic dataset with 100M tuples (size = 35 GB) with a derived attribute $\mathcal{A}_1$ (dom. size = 5) for evaluating the scalability of EnrichDB.

Enrichment Functions. In the real datasets, we used the following probabilistic classifiers: Gaussian Naïve Bayes (GNB), Decision Tree (DT), Support Vector Machine (SVM), k-Nearest Neighbor (KNN), Multi-Layered perceptron (MLP), Linear Discriminant Analysis (LDA), Logistic Regression (LR), and Random Forest (RF); as enrichment functions. The GNB classifier was calibrated using isotonic-regression model [117] and all the other classifiers were calibrated using Platt's sigmoid model [93] during cross-validation. After calibration, each classifier outputs a probability distribution. For the synthetic dataset, we used synthetic functions each associated with varying levels of quality and cost. For synthetic functions, quality 0.8 means predicting correct values 80% of the time. We used `weighted average` as the combiner function, where weights are proportional to the quality of enrichment functions learned based on the mechanism of §3.1.

Queries. As shown in Table 5.4, we selected nine queries, where ***Q1-Q4*** are ***selection queries***, ***Q5*** is a ***conjunctive query***, ***Q6-Q7*** are ***join queries***, and ***Q8-Q9*** are ***aggregation queries*** with number of groups as 2 (low) and 100 (high) respectively. The ***epoch size*** for all queries in all experiments (except Exp 5) was ***20 seconds***.

Plan Generation Strategies. For experiments, four different plan generation strategies were used: (*i*) *Benefit-based approach using Decision Table* (**BB(DT)**): that selects a set of ⟨tuple, function⟩ pairs with the highest benefit value based on the decision table. (*ii*) *Sample-based strategy with Object Order* (**SB(OO)**): that randomly selects tuples from the set of tuples satisfying predicates on fixed attributes. Selected tuples are completely enriched by executing all enrichment functions available for derived attributes present in the query. (*iii*) *Sample-based strategy with Function Order* (**SB(FO)**): that selects enrichment

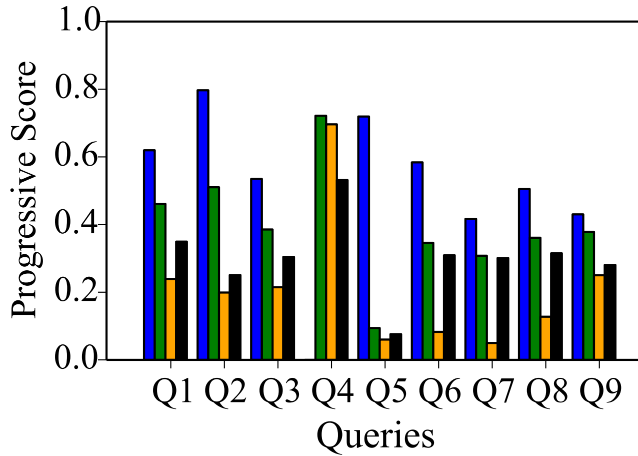
ID	Query	Application	Query Type
Q1	SELECT * from **ImageNet** where ObjectClass=2 where ImageID between (20000,30000)	Image	Selection
Q2	SELECT * from **MultiPie** where gender=1 and CameraID < 12	Image	Selection
Q3	SELECT tid, UserID, Tweet, location, TweetTime from **TweetData** where sentiment = 1 and TweetTime between(‘16:00’,‘18:00’)	Tweets	Selection
Q4	SELECT * from **SyntheticData** where $\mathcal{A}_1$=1 and $A_2 < 100000$	Synthetic	Selection
Q5	SELECT * from **MultiPie** where gender = 1 and expression =2 and CameraID < 12	Image	Selection
Q6	SELECT * from **TweetData** T1, **TweetData** T2 where T1.sentiment = T2.sentiment and T1.TweetTime between(‘16:00’,‘18:00’) and T2.TweetTime between (‘16:00’,‘18:00’)	Tweets	Join
Q7	SELECT * from **TweetData** T1, State S where T1.location = S.city and S.state=’California’ and T1.sentiment = 1 and T1.TweetTime between(‘16:00’,‘18:00’)	Tweets	Join
Q8	SELECT gender, count(*) from **MultiPie** where CameraID < 12 group by gender	Image	Aggregations
Q9	SELECT ObjectClass, count(*) from **ImageNet** where ImageID between (20000,30000) group by ObjectClass	Image	Aggregations

Table 5.4: Queries used.

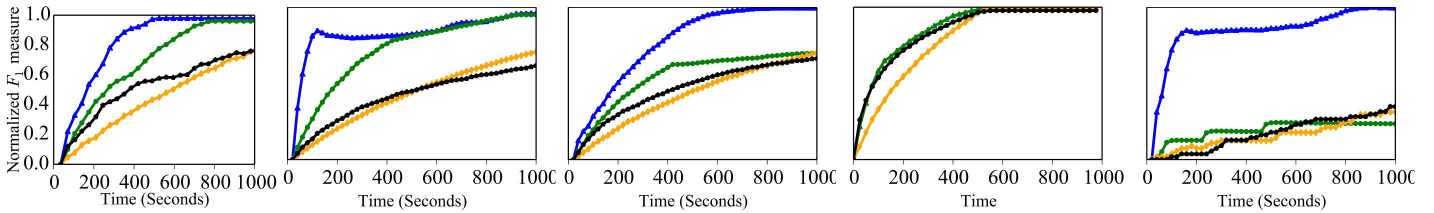
functions based on the decreasing order of their $\frac{quality}{cost}$ values, as described in §3.1. The top-most function from the sorted enrichment functions is executed on all tuples (obtained after checking the predicates on fixed attributes), before executing the next function. (*iv*) *Sample-based Random Order* (**SB(RO)**): that selects both tuples and enrichment functions randomly from a set of ⟨tuple, function⟩ pairs after checking predicates on fixed attributes.

5.6.2 Experimental Results

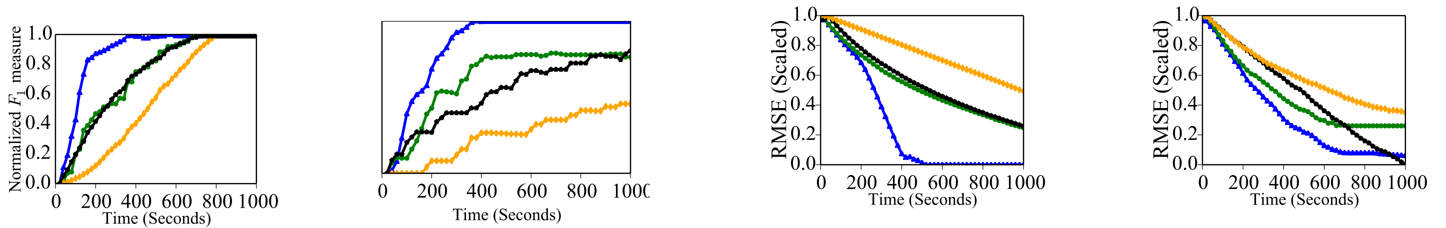
The experiments were performed on an AWS server with 16 core 2.50GHz Intel Xeon CPU, 64GB RAM, and 1TB SSD. **Exp 1-2** are progressiveness experiments, **Exp 3-4** are performance experiments and **Exp** 5 is an experiment for query parameter selection.



(a): Progressive Score.



(b): Imagenet (Q1). (c): Multi Pie (Q2). (d): Tweets (Q3). (e): Synthetic (Q4). (f): Conjunctive (Q5).



(g): Tweets Join (Q6). (h): Static Table Join (Q7). (i): MultiPie (Q8). (j): ImageNet (Q9).

Figure 5.2: Performance results of different plan generation strategies in EnrichDB.

Exp 1: Progressiveness of different queries. Figure 5.1 evaluates EDB in terms of progressive quality improvement achieved for different types of queries (Q1-Q9). For real datasets, we used the BB(DT) method, whereas for synthetic dataset we used SB(FO) plan generation method. Figure 5.1(a) shows the results for set based queries Q1-Q7, where the quality of answers is measured using ***normalized*** $\boldsymbol{F_1}$ *i.e.*, F_1/F_1^{max}, where F_1^{max} is the maximum F_1 measure achieved during the query execution. The ***normalized Jaccard's similarity*** results are shown in Figure 5.1(b). For aggregation queries Q8 and Q9, the quality is measured using ***normalized root mean square error*** (RMSE); Figure 5.1(c). We plot normalized measures as a function of time to emphasize the rate at which EnrichDB improves the quality of query results across different queries and datasets instead of the actual F_1-measures. Actual F_1-measure varies across different queries (that belong to different datasets) based on the quality of classifiers chosen for enrichment (*e.g.*, the maximum F_1 measures for different queries were: for Q1 0.54, Q2 0.78, Q3 0.56, Q4 1, Q5 0.4, Q6 0.58, Q7 0.52). From Figures 5.1(a) 5.1(b), and 5.1(c), we observe that EnrichDB achieves a high quality improvement (95% of maximum F_1 measure and 95% reduction of RMSE) within the first few epochs of query execution. Also, EnrichDB performs much better than a possible iterative strategy, where a database system chooses a tuple, completely enriches it, and if it satisfies the query predicate, returns it as an answer to the user (shown as the dotted line in Figure 5.1). Since for queries with blocking operators (joins and group-by aggregation such as queries Q6-Q9) all tuples need to be enriched completely before evaluating join or aggregation operators, such strategy cannot be devised.

Figure 5.1(d) shows the progressive score achieved for each query. Observe that EnrichDB achieves a high progressive score for all queries as compared to the possible iterative strategy (dotted line in Figure 5.1(d)). Table 5.5 shows the query execution time for Q1-Q9, when they are executed after complete enrichment of tuples. Comparing Table 5.5 with Figure 5.1, we observe that EnrichDB returns high quality results within a few epochs without requiring users to wait for complete enrichment.

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Time	31	44.5	22.1	14	67.1	39.2	22.8	45.1	31.8

Table 5.5: Time without progressive query execution (in minutes).

Exp 2: Effect of Different Plan Generation Strategies. Figure 5.2 studies different plan generation strategies and their impact on progressiveness. Figure 5.2(a) plots progressive score of all nine queries Q1-Q9. Figures 5.1(a) - 5.1(j) show BB(DT) performs better than sampling based approaches highlighting that the decision table learned by EnrichDB using the validation dataset accurately represents the benefit of the chosen execution order of enrichment functions. Among sample-based strategies, SB(FO) performs the best and SB(OO) performs the worst since function order chooses functions with highest quality per unit cost before other functions. SB(RO) performs only marginally better than SB(OO). In the following experiments, we use BB(DT) as a plan generation strategy, except for synthetic dataset for which we do not store benefit values.

Exp 3: System Overhead. This experiment measures the following two overheads incurred by EnrichDB:

(***i***) ***Time overhead*:** measures the amount of time spent in *non-enrichment tasks* (*i.e.*, query setup, benefit calculation, plan selection, delta computation, and state update) to compare against the time involved in data enrichment. Figure 5.5 shows that the time overhead is significantly lower than the time spent in enrichment. Particularly, across all epochs, the total time in query setup, benefit calculation, plan selection, delta computation, and state update took at most 3s, 90s, 4s, 5s, 17s, respectively, while the total time spent across all epochs in enrichment was 1000s. Majority of the time for benefit calculation was spent in the first epoch due to the calculation of benefit of all tuples satisfying query predicates on fixed attributes.

(***ii***) ***Storage overhead*:** measures the size of all temporary tables and IVM, used during query processing to compare against the size of EnrichDB tables. The maximum storage

overheads of the benefit table, plan table, and IVM at any epoch for the queries of Q1-Q9 were 1.48 MB, 56 KB, and 1.2 MB respectively (see Table 5.6). Also, the combined size of all metadata tables, (*i.e.*, `RelationMetadata`, `FunctionMetadata`, `FunctionFamily`, and `DecisionTable`) is less than 10 MB. These overheads are significantly lower than the size of EnrichDB tables. The state table sizes for TweetData, Imagenet, Multi-Pie and Synthetic datasets were 2.4GB, 6.8GB, 101MB and 246MB respectively (as shown in Table 5.7) which are much smaller than the corresponding EnrichDB tables. Furthermore, using state cutoff (§4.2.2) strategy, state storage overhead reduced significantly. For ImageNet the state cutoff representation reduced the size of state table from 6.8 GB to 50 MB. For TweetData the state overhead reduced from 2.4 GB to 0.9 GB. The improvement was more in ImageNet because the `ObjectClass` had a larger domain of size 100 as compared to `topic` attribute with domain size of 40.

Query	PlanSpaceTable	PlanTable	IVM
Q1	536 kB	48 kB	168 kB
Q2	776 kB	56 kB	232 kB
Q3	816 kB	48 kB	1168 kB
Q4	840 kB	36 kB	216 kB
Q5	1488 kB	8.2 kB	48 kB
Q6	264 kB	56 kB	112 kB
Q7	824 kB	48 kB	126 kB
Q8	832 kB	48 kB	0.25 kB
Q9	528 kB	48 kB	12 kB

Table 5.6: Max. storage overhead.

Relation	State overhead	Overhead after state cutoff
TweetData	2.4 GB	936 MB
ImageNet	6.8 GB	50 MB
Multi-Pie	101 MB	-
Synthetic	246 MB	-

Table 5.7: Storage cost of state.

Query	#rows in `PlanSpaceTable`	#rows in `PlanSpaceTable` with optimization
Q1	100K	10,000
Q7	11M	50,000

Table 5.8: Impact of optimization.

Exp 4: Impact of Optimizations. §4.2.3.1 presents two re-rewrite optimizations to reduce the complexity of plan generation step and enrichment. We selected Q1 (filter on

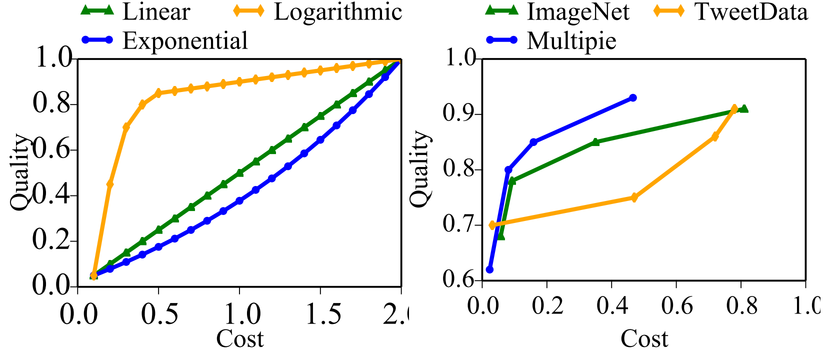


Figure 5.3: Cost vs. Quality (a) synthetic(lhs) and (b) real (rhs) functions.

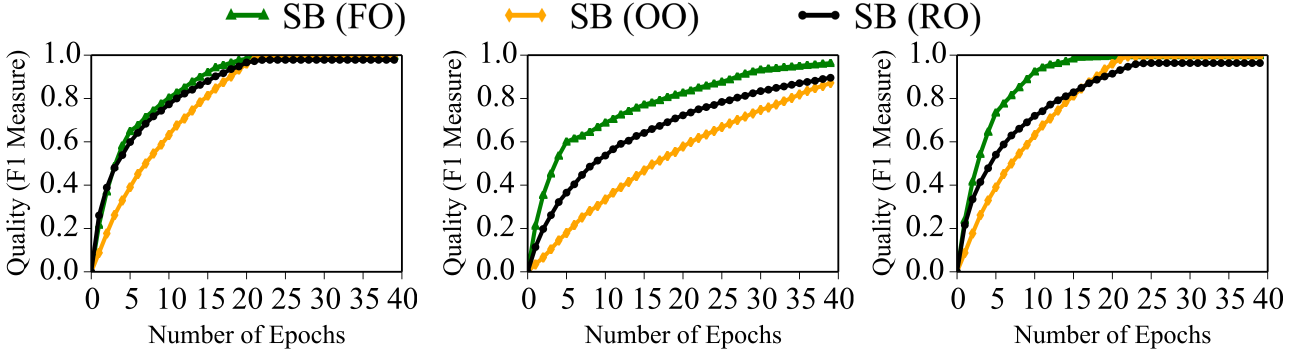


Figure 5.4: Comparing plan generation on synthetic data. (a) linear (b) logarithmic (c) exponential correlations.

fixed attribute) and Q7 (join on fixed attribute) to evaluate these optimizations. The results are presented in Table 5.8, that shows that the tuples considered for enrichment were reduced significantly due to these optimizations. Furthermore, due to the reduction of benefit table size, the complexity of the enrichment plan generation phase reduced significantly.

Exp 5: Effect of Epoch Size. Figure 5.6(a) plots time to reach (TTR) 90% quality for Q2 with respect to epoch size. As epoch size reduces from 50 to 20, TTR reduces since smaller epoch size causes more frequent plan generation resulting in accelerated improvement of answer quality. Reducing epoch size further (*i.e.*, 5 or 10) results in increase in TTR since increased overhead of frequent plan generation begins to overshadow improvements in quality achieved (Figure 5.6(a)). Figure 5.6(b) shows the overhead of plan generation as a function of epoch size. The effect of epoch size to other queries (besides Q2) is similar.

Exp 6: Impact of enrichment function correlations. This experiment evaluates the

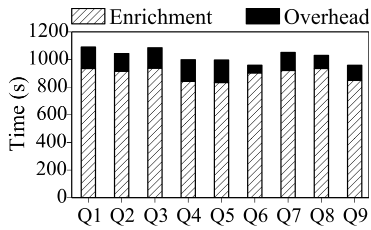


Figure 5.5: Time overhead.

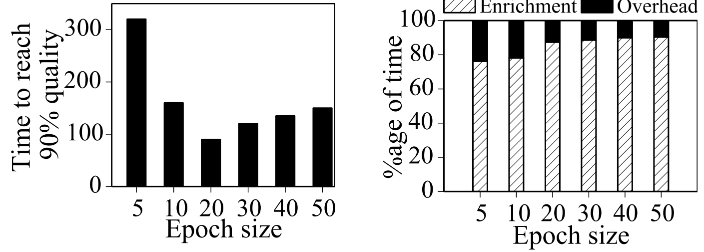


Figure 5.6: Effect of epoch sizes (a) TTR 90% (lhs) (b) overhead (rhs).

impact of variations in cost and quality of different enrichment functions on the sample based plan generation strategies (*i.e.*, SB(OO), SB(FO), SB(RO)) of EnrichDB. Note that, we do not experiment with BB(DT) strategy, since creating a decision table for the synthetic functions would be artificial and hence would not provide much insight. For this experiment, we used 10 synthetic functions that have the following correlations in their cost and quality (shown in Figure 5.3): (*i*) *linear correlation* where the quality and cost of functions vary linearly, (*ii*) *exponential correlation* where the quality increases exponentially with respect to the cost (the enrichment functions used in `TweetData` follow such correlation), and (*iii*) *logarithmic correlation* where quality increases logarithmically with respect to the cost (the enrichment functions used in MultiPie and ImageNet dataset follow such correlation).

For each enrichment function correlation, we compared all SB plan generation strategies (as described in Exp 1) on query Q4 using F_1 measure (Figure 5.4). Since the function with highest $\frac{quality}{cost}$ is always chosen first in SB (FO) strategy, it always outperforms both SB(OO) and SB(RO). However, SB (RO) becomes almost as good as SB(FO), when the correlation is linear (Figure 5.4(a)), since all enrichment functions have the same $\frac{quality}{cost}$, when the correlation is linear. SB (OO) performs worst as it executes all enrichment functions (including functions with low $\frac{quality}{cost}$), before enriching another tuple.

The correlations of the functions in real datasets are shown in Figure 5.3 (b). From Figures 5.2(b), 5.2(c), and 5.2(d) of Exp 2, we observe that the quality of answer for BB(DT) improves very aggressively in MultiPie as compared to ImageNet and TweetData. The

reason is that the quality of functions used in MultiPie has highest positive curvature in quality-cost graph as compared to ImageNet and TweetData functions.

Chapter 6

Use Cases of the System

User Interaction/Usability. Users interact with EnrichDB using a command line or a web-application based interface. Users are able to (*i*) create a new table, (*ii*) specify derived attributes that require enrichment, (*iii*) add data to the table, (*iv*) add new enrichment functions and associate them with derived attributes, (*v*) use training feature to train new ML models and use them in enrichment functions, (*vi*) pose queries on added data. Users are able to visualize query results using EnrichDB web interface based on the type of queries (set/aggregation). Query results are updated automatically at the end of each epoch. Newly added tuples are highlighted using green boxes and the deleted tuples are highlighted using red boxes in the interface. Furthermore, users will be able to pause, resume or stop an ongoing query. We consider the data stored in EnrichDB to be append-only.

6.1 Social Media Analysis

Let us consider an application, where analysts are interested to find the reaction of the public about the most recent presidential debate from tweets. We develop this application using

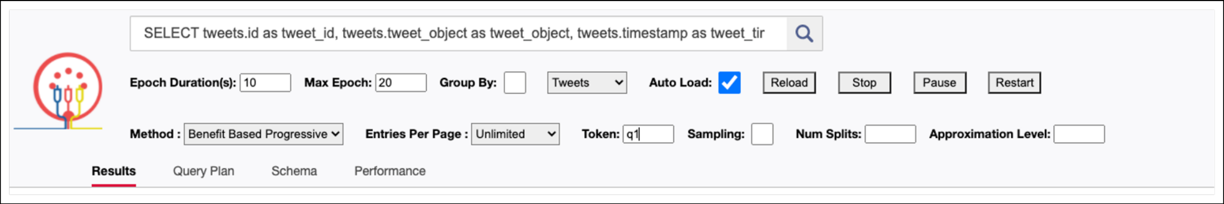


Figure 6.1: EnrichDB web interface for submitting query.

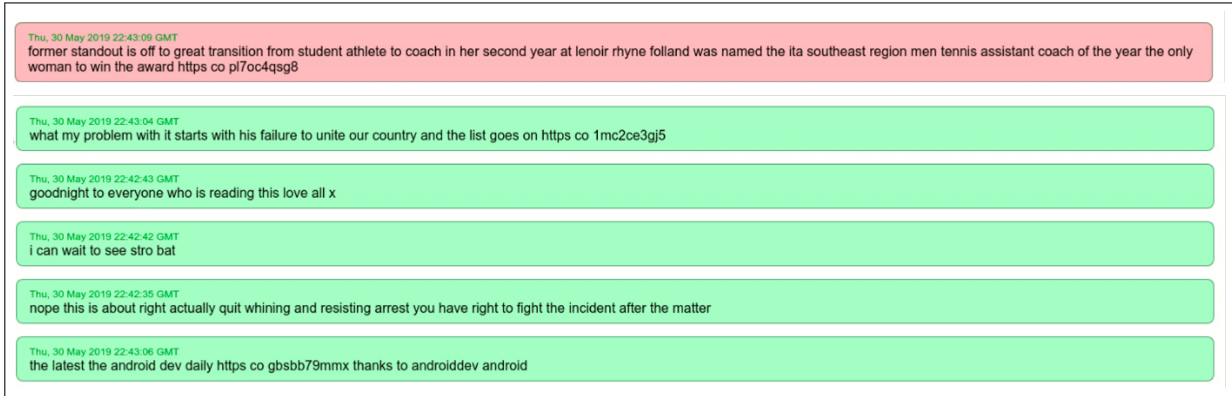


Figure 6.2: Interface for visualizing progressively improving query results on image dataset.

EnrichDB. To do so, analysts store the tweets after the end of the presidential debate in a table `TweetData`. Here, analysts must train multiple machine learning models for detecting the sentiment and topic of such tweets based on a past dataset (stored in `TweetsTrain` table) that was collected during the previous presidential debates. Finally, analysts pose a query on `TweetData` table to find out all tweets with positive sentiment and the topic of the presidential debate. In order to achieve these functionalities, the steps that analysts must take in EnrichDB are presented below. Note that EnrichDB-based implementation is much simpler (≈12 lines of code) as compared to any loosely coupled implementation, where enrichment is performed outside of DBMS and requires much more lines of code (≈130 lines, as shown in Appendix A).

```
1 -- Creating a new table
2 CREATE TABLE TweetData(tid char(8),
3    userid char(20),Tweet text,feature float[],
4    topic int derived:40,sentiment int
5    derived:3, Time timestamp,location text);
```

```
6  -- Training ML Models
7  SELECT db.model_train('TweetsTrain',
8     'sentiment_dt', 'decision_tree',
9     'sentiment', 'feature[]', model_params);
10 -- Associating functions with 'sentiment'
11 SELECT db.assign_enrichment_functions(
12    'tweets',
13    [['sentiment',1,'sentiment_dt',0.8,0.7],
14    ['sentiment',2,'sentiment_fo',0.9,0.8],
15    ['sentiment',3,'sentiment_mlp',0.9, 0.9]]);
16 -- Setting up decision table
17 SELECT db.learn_decision_table('TweetData',
18    'sentiment','TweetValidationSet');
19 --Adding data
20 SELECT db.enriched_insert
21    ('INSERT INTO TweetData (id, tweet_object,
22    topic,sentiment,timestamp,location) VALUES
23    (1,''is very happy for my friend'',NULL,
24    NULL,"2021-02-06 12:29:06","LO")');
25 -- Executing Queries
26 CALL db.exec_driver_udf('SELECT id,
27    tweet_object,location,timestamp, sentiment,
28    topic FROM TweetData WHERE sentiment = 0
29    AND topic = 2 AND id < 10000', 20, 0.9);
```

In the following, we describe the implementation of the above application using Spark.

6.2 Multi Media Analysis

In this scenario, we use two image datasets: (i) **Imagenet** dataset containing images from 1000 image categories and (ii) **Multi-PIE** dataset containing face images of people. We have implemented multiple enrichment functions to derive the **object type** from Imagenet dataset and to derive **gender** and **expression** of people from Multi-Pie dataset. In this application, users will pose queries to find out images of smiling males or finding images of sharks on these datasets. Figure 6.3 shows the web interface for submitting a query on image data (same as earlier, only the drop down menu for the visualization needs to modified to "Images"). Furthermore, Figure 6.4 shows the web-interface for visualizing the results on Multi-PIE dataset.

```
1  -- Creating a new table
2  CREATE TABLE ImageData(id int precise,
3     image_blob  bytea precise,
4     feature float[] precise,
5     gender  int imprecise:2,
6     expression  int precise,
7     age  int precise,
8     timestamp  timestamp precise,
9     location  text precise,
10    cameraid  int precise);
11
12 -- Training ML Models
13 SELECT db.model_train('ImagesTrain',
14    'gender_dt_model', 'decision_tree',
15    'gender', 'feature[]', model_params);
16
```

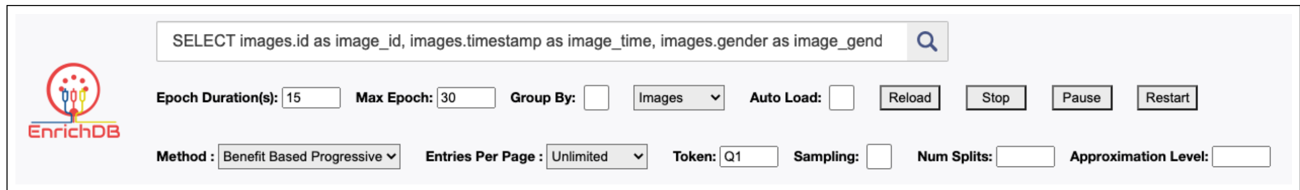


Figure 6.3: EnrichDB web interface for submitting query on images.

```
17 -- Associating functions with derived attribute ‘gender’
18 SELECT db.assign_enrichment_functions(
19     'ImageData',
20     [['gender', '1', 'gender_dt_model', '0.80', '0.7', ''],
21      ['gender', '2', 'gender_mlp_model', '0.84', '0.78', ''],
22      ['gender', '3', 'gender_knn_model', '0.98', '0.9', '']]);
23
24 -- Setting up decision table
25 SELECT db.learn_decision_table('ImageData',
26    'gender','ImageDataValidation');
27
28 --Adding data
29 SELECT db.enriched_insert('INSERT INTO ImageData (id,feature,gender,
     timestamp,location,cameraid) VALUES (1, ''
     {0.014,0.0,0.006,0.003,0.94,0.0023,0.0071}'', NULL, ''2020-02-04
      16:28:47.261182'', ''L0'', 0)');
30
31 -- Executing Queries
32 CALL db.progressive_exec_driver_udf('SELECT images.id as image_id,
     images.timestamp as image_time, images.gender as image_gender
33     @FROM images @WHERE images.gender = 0', 20, 5);
```

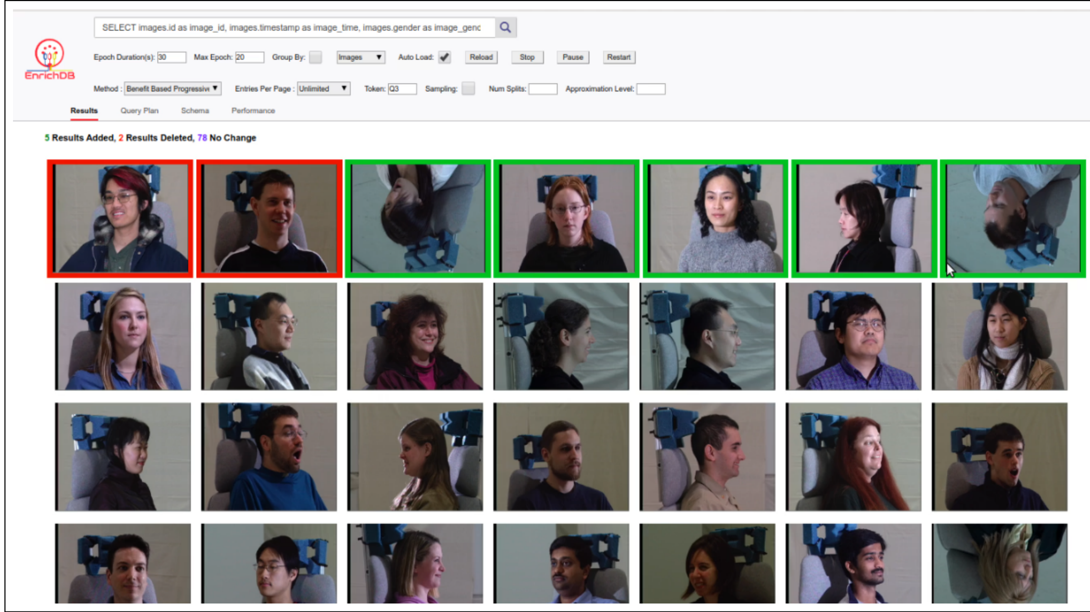


Figure 6.4: Interface for visualizing progressively improving query results on image dataset.

6.3 IoT Application of Localization using WiFi data

In this application, we use the connectivity data of users' mobile devices with WiFi access points installed inside a building. Such connectivity data is used to localize individuals in the building at different levels of granularity (region and room level) and accuracy [75]. We have implemented five enrichment functions for **localization** based on the LOCATER algorithms, proposed in [75]. Each of the algorithms have different cost quality tradeoff where the cheapest function has lowest accuracy, and the most expensive function has highest accuracy. In this application, analysts pose queries to (*i*) find out the location of users (in floor level and region level) with respect to time and (*ii*) find out the occupancy of the floors and regions of the building at each time instants within two time intervals. The analyst needs to submit the query, provide an epoch duration, type of query (for visualization), and an optional maximum number of epochs till which the query needs to be executed. In the following we describe the complete application logic implemented using EnrichDB.

```
1 -- Creating a new table
```

```
2 CREATE TABLE PresenceData(
3   id int precise,
4   semantic_id int precise,
5   observationtime timestamp precise,
6   location int imprecise:10);
7
8 -- Training ML Models
9 SELECT db.model_train('PresenceTrain',
10   'location_dt_model', 'decision_tree',
11   'location', 'feature[]', model_params);
12
13 -- Associating functions with derived attribute ‘gender’
14 SELECT db.assign_enrichment_functions(
15    'PresenceTrain',
16    [['location', '1', 'location_dt_model', '0.80', '0.7', ''],
17     ['location', '2', 'location_mlp_model', '0.84', '0.78', ''],
18     ['location', '3', 'location_knn_model', '0.98', '0.9', '']]);
19
20 -- Setting up decision table
21 SELECT db.learn_decision_table('PresenceData',
22   'location','PresenceDataValidation');
23
24 --Adding data
25 select db.enriched_insert('INSERT INTO presence (id, semantic_id,
    observationtime, location) VALUES (1, 40, ''2020-02-04 16:28:47'
    ', NULL)')
26
27 -- Executing Queries
```

```
28 call db.progressive_groupby_exec_driver_udf('SELECT PresenceData.
     timestamp as time_, count(PresenceData.id) as count_, count(
     PresenceData.location) as count_1
29 FROM PresenceData WHERE PresenceData.observationtime < ''2020-02-01
     08:00:00'' AND PresenceData.observationtime < ''2020-02-01
     17:00:00'' GROUP BY PresenceData.observationtime', 10, 1000);
```

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we proposed EnrichDB — a new system for supporting data enrichment inside a single data management system. The cornerstone of EnrichDB is a powerful *enrichment data model* that encapsulates enrichment as an operator inside a DBMS enabling it to co-optimize enrichment with query processing. EnrichDB data model provides semantic abstraction and transparency of enrichment that allows application developers to write application logic using semantically higher level data. Furthermore, developers do not have to be concerned about how EnrichDB enriches the underlying data while developing their application logic.

In the second chapter, we described the implementation of EnrichDB system, developed using a loosely-coupled and a tightly-coupled approach. The loosely-coupled architecture provides scope for parallelism of the enrichment process by using multiple servers as enrichment server instead of one. In contrast, the loosely coupled approach is very beneficial in terms of exploiting query context to eliminate redundant enrichment of objects. We have

experimentally shown that such an approach can save large amount of enrichment of objects. Furthermore, we showed that both the approaches can support progressive approach of query processing. In this approach, initially, approximate results are returned that are improved over time as more relevant data is enriched. While the system is implemented using a tightly couple approach (*i.e.*, $\mathbb{EQ}_{\mathbb{TC}}$) and a loosely coupled approach (*i.e.*, $\mathbb{EQ}_{\mathbb{LC}}$), exploring additional optimizations that are possible at the storage layer and modifying query processing layer of DBMS, are interesting future directions for us.

In the third chapter, we described a mechanism for optimizing enrichment with progressive query processing. We presented the algorithm and its implementation using the $\mathbb{EQ}_{\mathbb{LC}}$ and $\mathbb{EQ}_{\mathbb{TC}}$ approaches. We showed that EnrichDB achieves very high quality query result within the first few epochs of query execution. Both $\mathbb{EQ}_{\mathbb{LC}}$ and $\mathbb{EQ}_{\mathbb{TC}}$ had very low overhead in terms of the time spent in non-enrichment tasks and the storage overhead of the temporary tables and data structures used during query processing.

7.2 Future Work

There are several directions in which EnrichDB can be extended as described below.

Support for Partial Execution of Enrichment Functions. In EnrichDB, we have assumed that enrichment functions execute atomically. However, with the development of deep neural networks such as convolutional neural networks, execution of an enrichment function on a single tuple can be very expensive. In such scenarios, partial execution of enrichment functions (*i.e.*, execution of only a part of neural net) can be used to get a quick insight on the tuple's derived attribute value. Supporting such partial execution of enrichment functions will require a different way of state management and query processing in EnrichDB and is an interesting direction of future work.

Support for Continuous Queries. In EnrichDB, we have supported adhoc queries that require enrichment of the underlying data. Supporting scalable enrichment for continuous queries is an interesting direction of future work. Furthermore, optimizing such enrichment tasks with query processing of continuous queries, is another direction of future work. *E.g.*, considering a continuous query with sliding window-based semantics, the benefit of enriching a tuple in a window, can be beneficial for answering queries in multiple windows.

Support for Progressive Training of Enrichment Functions. In EnrichDB, we have considered that the training of the enrichment functions is performed before any queries are posed. However, often in real life scenarios, machine learning models need to be re-trained on the live data when the accuracy of machine learning models fall below a certain accuracy. During progressive query processing, enabling EnrichDB to trigger such training mechanisms based on the result of latest query answers, can be an interesting direction of future work.

Optimization of Multiple Queries. In this thesis, we have considered that optimization of enrichment is performed only in the context of a single query, executed at the EnrichDB server. However, optimizing system resources to support enrichment for multiple application queries running concurrently, is an interesting direction of future work.

Extending EnrichDB to General Class of Sensor-Driven Smart Applications. In the current use-cases of EnrichDB, we have tested it in the domains of social-media analysis, multimedia analysis, and occupancy analysis of users in an IoT application. Extending EnrichDB to more general class of applications that support smart-water infrastructure, smart-cities, advanced healthcare facilities, or detection of wildfire progression on a real-time basis can be an interesting direction of future work. Enrichment functions in such scenarios can be based on time series analysis of sensor data in various time windows, execution of various simulation codes, or pattern recognition of user's behaviour in the past.

Bibliography

[1] EnrichDB code. https://github.com/DB-repo/enrichdb.

[2] Amazon Redshift. https://aws.amazon.com/redshift/.

[3] Apache asterixdb. https://asterixdb.apache.org/.

[4] Apache kafka. https://kafka.apache.org/23/documentation/streams/.

[5] Etl vs elt: 5 critical differences. https://www.xplenty.com/blog/etl-vs-elt/#elt.

[6] Google bigquery data warehouse. https://cloud.google.com/bigquery.

[7] Greenplum database. https://greenplum.org/.

[8] Ibm db2. https://www.ibm.com/analytics/db2.

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

[10] Incremental view maintenance for amazon redshift. https://docs.aws.amazon.com/redshift/latest/dg/materialized-view-refresh-sql-command.html.

[11] Incremental view maintenance for oracle database. https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_6002.html.

[12] Internet live stats. http://www.internetlivestats.com.

[13] Sap hana core data services (cds). https://help.sap.com/viewer/09b6623836854766b682356393c6c416/2.0.02/en-US/b710731496cf43b7ba76e15a928f1a80.html.

[14] Snowflake data warehouse. https://www.snowflake.com/workloads/data-warehouse-modernization/.

[15] Top 10 elt tools. https://hevodata.com/learn/best-elt-tools.

[16] Top 10 elt tools. https://hevodata.com/.

[17] Transaction processing performance council. tpc-h specification. `http://www.tpc.org/tpch/`.

[18] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Found. Trends Databases*, 5(3):197–280, 2013.

[19] S. Agarwal et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. EuroSys '13.

[20] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154. ACM, 2006.

[21] S. Agrawal et al. Scalable ad-hoc entity extraction from text collections. *Proc. VLDB Endow.*, 1(1):945–957, Aug. 2008.

[22] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[23] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. R. Borkar, Y. Bu, M. J. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. J. Tsotras, R. Vernica, J. Wen, and T. Westmann. Asterixdb: A scalable, open source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, 2014.

[24] Y. Altowim et al. Progressive approach to relational entity resolution. *VLDB'14*.

[25] Y. Altowim et al. Progresser: Adaptive progressive approach to relational entity resolution. *ACM Trans. Knowl. Discov. Data*, 12(3), Mar. 2018.

[26] Y. Altowim, D. V. Kalashnikov, and S. Mehrotra. Progressive approach to relational entity resolution. *Proc. VLDB Endow.*, 7(11):999–1010, 2014.

[27] Y. Altowim and S. Mehrotra. Parallel progressive approach to entity resolution using mapreduce. In *ICDE*, pages 909–920. IEEE Computer Society, 2017.

[28] H. Altwaijry et al. Query-driven approach to entity resolution. *PVLDB*, 2013.

[29] H. Altwaijry et al. QuERy: A framework for integrating entity resolution with query processing. *PVLDB*, 9(3):120–131, 2015.

[30] Y. Amsterdamer et al. Crowd mining. In *SIGMOD 2013*, pages 241–252, 2013.

[31] R. Avnur et al. Eddies: Continuously adaptive query processing. *SIGMOD 2000*.

[32] K. E. Benson, G. Bouloukakis, C. Grant, V. Issarny, S. Mehrotra, I. D. Moscholios, and N. Venkatasubramanian. Firedex: a prioritized iot data exchange middleware for emergency response. In *Middleware*, pages 279–292. ACM, 2018.

[33] K. E. Benson, C. Fracchia, G. Wang, Q. Zhu, S. Almomen, J. Cohn, L. D'arcy, D. Hoffman, M. Makai, J. Stamatakis, and N. Venkatasubramanian. SCALE: safe community awareness and ale rting leveraging the internet of things. *IEEE Commun. Mag.*, 53(12):27–34, 2015.

[34] L. Berg, T. Ziegler, C. Binnig, and U. Röhm. Progressivedb: Progressive data analytics as a middleware. *Proc. VLDB Endow.*, 12(12):1814–1817, Aug. 2019.

[35] P. A. Bernstein and D. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.

[36] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, June 1986.

[37] P. Brucker. *Scheduling Algorithms*. Springer Publishing Company, 2010.

[38] Z. Cai, Z. Vagena, L. L. Perez, S. Arumugam, P. J. Haas, and C. M. Jermaine. Simulation of database-valued markov chains using simsql. In *SIGMOD Conference*, pages 637–648. ACM, 2013.

[39] P. Carbone et al. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[40] S. Chaudhuri et al. Optimization of queries with user-defined predicates. *TODS '99*.

[41] S. Chaudhuri et al. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2):9, 2007.

[42] W. Cheng, E. Hüllermeier, and K. J. Dembczynski. Bayes optimal multilabel classification via probabilistic classifier chains. In *ICML*, pages 279–286, 2010.

[43] T. Condie et al. Mapreduce online. pages 313–328. USENIX NSDI, 2010.

[44] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627. USENIX Association, 2017.

[45] N. Dalvi et al. Efficient query evaluation on probabilistic databases. *VLDB '07*.

[46] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and F. Li. Imagenet: A large-scale hierarchical image database. In *CVPR*, pages 248–255. IEEE Computer Society, 2009.

[47] T. G. Dietterich. Overfitting and undercomputing in machine learning. *ACM Comput. Surv.*, 27(3):326–327, 1995.

[48] B. Ding et al. Sample + seek: Approximating aggregates with distribution precision guarantee. In *SIGMOD Conference*, pages 679–694. ACM, 2016.

[49] F. Färber, May, et al. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[50] S. Feng et al. Uncertainty annotated databases - A lightweight approach for approximating certain answers. In *SIGMOD*, 2019.

[51] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a unified architecture for in-rdbms analytics. In *SIGMOD Conference*, pages 325–336. ACM, 2012.

[52] D. Ghosh et al. Progressive evaluation of queries over untagged data. *CoRR*, abs/1805.12033, 2018.

[53] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242. IEEE Computer Society, 2011.

[54] 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*, SIGMOD '20, page 805–815, New York, NY, USA, 2020. Association for Computing Machinery.

[55] R. Grover and M. J. Carey. Data ingestion in asterixdb. In *EDBT*, pages 605–616. OpenProceedings.org, 2015.

[56] Q. Han, S. Mehrotra, and N. Venkatasubramanian. Aquascale - exploring resilience of community water infrastructures. In *Middleware Demos/Posters*, pages 7–8. ACM, 2019.

[57] J. M. Hellerstein et al. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD*, 1993.

[58] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997.

[59] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library or MAD skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.

[60] Y. Hsieh, H. Hong, P. Tsai, Y. Wang, Q. Zhu, M. Y. S. Uddin, N. Venkatasubramanian, and C. Hsu. Managed edge computing on internet-of-things devices for smart city applications. In *NOMS*, pages 1–2. IEEE, 2018.

[61] R.-L. Hsu et al. Face detection in color images. *IEEE Tran. on Pattern Analysis and Machine Intelligence*, 2002.

[62] J. Huang, L. Antova, C. Koch, and D. Olteanu. Maybms: a probabilistic database management system. In *SIGMOD Conference*, pages 1071–1074. ACM, 2009.

[63] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679 – 688, 2006.

[64] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Found. Trends Databases*, 5(4):281–393, 2015.

[65] P. JACCARD. Etude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

[66] J. Jia, C. Li, and M. J. Carey. Drum: A rhythmic approach to interactive analytics on large data. In *IEEE BigData*, pages 636–645. IEEE Computer Society, 2017.

[67] M. Joglekar et al. Exploiting correlations for expensive predicate evaluation. SIGMOD '15, New York, NY, USA, 2015. ACM.

[68] K. Karanasos et al. Dynamically optimizing queries over large scale data platforms. SIGMOD, 2014.

[69] K. Karanasos, M. Interlandi, F. Psallidas, R. Sen, K. Park, I. Popivanov, D. Xin, S. Nakandala, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino. Extending relational query processing with ML inference. In *CIDR*. www.cidrdb.org, 2020.

[70] H. Kimura, S. Madden, and S. B. Zdonik. UPI: A primary index for uncertain databases. *Proc. VLDB Endow.*, 3(1):630–637, 2010.

[71] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[72] M. Lapin et al. Top-k multiclass SVM. In *NIPS 2015*.

[73] I. Lazaridis et al. Optimization of multi-version expensive predicates. SIGMOD'07.

[74] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[75] Y. Lin, D. Jian, R. Yus, G. Bouloukakis, A. Chio, S. Mehrotra, and N. Venkatasubramanian. Locater: Cleaning wifi connectivity datasets for semantic localization.

[76] N. Littlestone and M. K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 1994.

[77] Y. Lu et al. Accelerating machine learning inference with probabilistic predicates. SIGMOD '18, New York, NY, USA, 2018. ACM.

[78] V. Markl et al. Robust query processing through progressive optimization. SIGMOD '04.

[79] D. Marmaros et al. Pay-as-you-go entity resolution. *IEEE TKDE*, 2013.

[80] N. May et al. Sap hana–the evolution of an in-memory dbms from pure olap processing towards mixed workloads. *BTW 2017*.

[81] R. McCann et al. Matching schemas in online communities: A web 2.0 approach. In *ICDE*, pages 110–119, April 2008.

[82] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*. www.cidrdb.org, 2013.

[83] K. Mikolajczyk et al. Human detection based on a probabilistic assembly of robust part detectors. In *ECCV 2004*.

[84] U. F. Minhas and A. Kumar. SIGMOD 2021 curated session: Data management for ML. `https://2021.sigmod.org/program/program_tuesday.shtml`.

[85] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

[86] M. Nikolic, M. Elseidy, and C. Koch. LINVIEW: incremental view maintenance for complex analytical queries. In *SIGMOD Conference*, pages 253–264. ACM, 2014.

[87] R. Olfati-Saber et al. Consensus filters for sensor networks and distributed sensor fusion. CDC '05, Dec 2005.

[88] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *Proc. VLDB Endow.*, 4(11):1135–1145, 2011.

[89] T. Papenbrock et al. Progressive duplicate detection. *IEEE TKDE*, 2015.

[90] A. Parameswaran et al. Crowdscreen: Algorithms for filtering data with humans.

[91] A. Parameswaran et al. Optimal crowd-powered rating and filtering algorithms. *Proc. VLDB Endow.*, 7(9):685–696, May 2014.

[92] Y. Park et al. Verdictdb: Universalizing approximate query processing. SIGMOD'18.

[93] J. C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *ADVANCES IN LARGE MARGIN CLASSIFIERS*.

[94] H. Plattner. The impact of columnar in-memory databases on enterprise systems. *Proc. VLDB Endow.*, 7(13):1722–1729, 2014.

[95] D. Powers et al. Evaluation: From precision, recall and f-measure to roc, informedness, markedness and correlation. *J. Mach. Learn. Technol*, 2:2229–3981, 01 2011.

[96] E. Rahm et al. A survey of approaches to automatic schema matching. *VLDB '01*.

[97] V. Raman and J. M. Hellerstein. Partial results for online query processing. In *SIGMOD Conference*, pages 275–286. ACM, 2002.

[98] C. Re et al. Efficient top-k query evaluation on probabilistic data. In *ICDE 2007*.

[99] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: Holistic data repairs with probabilistic inference. *Proc. VLDB Endow.*, 10(11):1190–1201, 2017.

[100] A. Rheinländer, U. Leser, and G. Graefe. Optimization of complex dataflows with user-defined functions. *ACM Comput. Surv.*, 50(3):38:1–38:39, 2017.

[101] P. Sen et al. Representing and querying correlated tuples in probabilistic databases. In *ICDE '07*.

[102] E. Shin, D. Ghosh, S. Mehrotra, and N. Venkatasubramanian. SCARF: a scalable data management framework for context-aware applications in smart environments. In *MobiQuitous*, pages 358–367. ACM, 2019.

[103] N. F. F. D. Silva et al. A survey and comparative study of tweet sentiment analysis via semi-supervised learning. *ACM Comput. Surv.*, 2016.

[104] T. Sim et al. The cmu pose, illumination, and expression (pie) database. In *Int. Conf. on Automatic Face Gesture Recognition*, 2002.

[105] J. A. K. Suykens and J. Vandewalle. Least squares support vector machine classifiers. *Neural Process. Lett.*, 9(3):293–300, 1999.

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

[107] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Crocodiledb in action: Resource-efficient query execution by exploiting time slackness. *Proc. VLDB Endow.*, 13(12):2937–2940, 2020.

[108] D. Tang, Z. Shang, A. J. Elmore, S. Krishnan, and M. J. Franklin. Thrifty query execution via incrementability. In *SIGMOD Conference*, pages 1241–1256. ACM, 2020.

[109] S. Tian, W. Yang, J. M. L. Grange, P. Wang, W. Huang, and Z. Ye. Smart healthcare: making medical care more intelligent. *Global Health Journal*, 3(3):62–65, 2019.

[110] A. Toshniwal et al. Storm@twitter. SIGMOD '14.

[111] M. Vagac and M. Melichercík. Improving image processing performance using database user-defined functions. In *ICAISC (1)*, volume 9119 of *Lecture Notes in Computer Science*, pages 789–799. Springer, 2015.

[112] P. Venkateswaran, M. A. Suresh, and N. Venkatasubramanian. Augmenting in-situ with mobile sensing for adaptive monitoring of water distribution networks. In *ICCPS*, pages 151–162. ACM, 2019.

[113] X. Wang and M. J. Carey. An idea: An ingestion framework for data enrichment in asterixdb. *VLDB '19*.

[114] C. J. Willmott and K. Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1):79–82, 2005.

[115] D. H. Wolpert. Stacked generalization. *Neural Networks*, 1992.

[116] S. Wu et al. Continuous sampling for online aggregation over multiple queries. In *SIGMOD Conference*, pages 651–662. ACM, 2010.

[117] B. Zadrozny and C. Elkan. Transforming classifier scores into accurate multiclass probability estimates. KDD, 2002.

[118] M. Zaharia et al. Discretized streams: A fault-tolerant model for scalable stream processing, 2012.

[119] K. Zeng et al. G-OLA: generalized on-line aggregation for interactive analysis on big data. In *SIGMOD Conference*, pages 913–918. ACM, 2015.

[120] Y. Zhang, W. Zhang, and J. Yang. I/o-efficient statistical computing with RIOT. In *ICDE*, pages 1157–1160. IEEE Computer Society, 2010.

[121] Q. Zhu, M. Y. S. Uddin, Z. Qin, and N. Venkatasubramanian. Data collection and upload under dynamicity in smart community internet-of-things deployments. *Pervasive Mob. Comput.*, 42:166–186, 2017.

Appendices

A Twitter Analysis Application on Spark

The codebase for the twitter analysis application implemented using Spark is shown below.

```
import time
import re
import sys
import numpy as np
import pickle
import pyspark
from pyspark import SQLContext
from pyspark.sql.types import StructType, StructField, IntegerType,
    FloatType, StringType, ArrayType
from pyspark.sql.functions import udf
from pyspark.sql import Row
from pyspark.sql.functions import col
from pyspark.sql import SparkSession
from pyspark import SparkContext
from pyspark import SparkConf
import pyspark.sql.functions as F
```

```
sys.path.insert(0, '/home/ubuntu/functions/backend/load')

url = 'jdbc:postgresql://localhost:5432?user=postgres&password=
    postgres'
table = 'tweets'
conf = SparkConf()
conf.setMaster("local[*]")
conf.setAppName('pyspark')


properties = {
    'user': 'postgres',
    'password': 'postgres',
    'driver': 'org.postgresql.Driver',
    'spark.jars':'org.postgresql:postgresql:42.2.12'
}

sc = pyspark.SparkContext.getOrCreate()

spark = SparkSession.builder.appName("Python Spark SQL").config("
    spark.jars","/home/ubuntu/java/postgresql-42.2.18.jar").
    getOrCreate()

df = spark.read.format("jdbc").option("url", "jdbc:postgresql://
    localhost:5432/test").option("dbtable", "tweets").option("user",
     "postgres").option("password", "postgres").load()
```

```
clf_dt = pickle.load(open('/home/ubuntu/tweet_clfs/
    tweet_dt_sentiment_calibrated.p', 'rb'))
clf_gnb = pickle.load(open('home/ubuntu/tweet_clfs/
    tweet_gnb_sentiment_calibrated.p', 'rb'))
clf_lda = pickle.load(open('home/ubuntu/tweet_clfs/
    tweet_lda_sentiment_calibrated.p', 'rb'))
clf_mlp = pickle.load(open('home/ubuntu/tweet_clfs/
    tweet_mlp_sentiment_calibrated.p', 'rb'))

def execute_mlp(rl):
    gProb = clf_mlp.predict_proba(rl)
    return gProb[0]

def execute_dt(rl):
    gProb = clf_dt.predict_proba(rl)
    return gProb[0]

def execute_gnb(rl):
    cProb = clf_gnb.predict_proba(rl)
    return cProb[0]

def execute_lda(rl):
    gProb = clf_lda.predict_proba(rl)
    return gProb[0]

def execute_svm(rl):
    gProb = clf_svm.predict_proba(rl)
    return gProb[0]
```

```
def generateCombinedProbability(functionBitmap, probability2dArr):

    output_arr = [0] * len(probability2dArr[0])
    num_possible_tag = len(probability2dArr[0])

    weights = [1,2, 3 ,6]

    for i in range(num_possible_tag):
        sum_val = 0.0
        count_val = 0
        for j in range(len(functionBitmap)):
            if functionBitmap[j] == 1:
                sum_val += weights[j]*probability2dArr[j][i]
                count_val += weights[j]
        if count_val > 0:
            output_arr[i] = 1.0 * (sum_val / count_val)
        else:
            output_arr[i] = 0

    return output_arr

def _response(input_features):
    fcname = 'tweet_sentiment_all'
    response = None
    proba = None

    if fcname == 'tweet_sentiment_all':
        bitmap = [1,1,1,1]
```

```
        prob2DArray =[]

        features =  input_features
        proba = execute_dt([features[:1000]])
        prob2DArray.append(proba)

        proba = execute_gnb([features[:1000]])
        prob2DArray.append(proba)

        proba = execute_lda([features[:1000]])
        prob2DArray.append(proba)

        proba = execute_mlp([features[:1000]])
        prob2DArray.append(proba)

        output = generateCombinedProbability(bitmap, prob2DArray)
        proba = output
        response = 0

    res = [round(v,6) for v in output]
    max_val = max(res)
    label = res.index(max_val)
    return label


if __name__ == "__main__":

    start_id = 50000
    end_id = 60000
```

```
_select_sql = "(select t1.id,t1.tweet,t1.feature, t2.sentiment
from tweets t1, tweets_full t2 where t1.id = t2.id and t1.id >"
+ str(start_id)+ " and t1.id <" + str(end_id) +  " and t2.id >"
+ str(start_id) +" and t2.id  <" + str(end_id) + ") as my_table"
df_select = spark.read.jdbc(url="jdbc:postgresql://localhost
:5432/test",table=_select_sql,properties=properties)
df_select.show()
truth_list = df_select.select('sentiment').rdd.flatMap(lambda x:
 x).collect()
all_f1 = []
for i in range(len(truth_list)):
    if truth_list[i] == 1:
            truth+=1
prev_recall =0
max_time = 50
ep_size = 10
quality_requirement = 0.9

for i in range(max_time):
    t1 = time.time()
    start_id = 0 + i * epoch_size
    end_id = start_id + epoch_size
    _select_sql = "(select t1.id,t1.tweet,t1.feature, t2.
sentiment from tweets t1, tweets_full t2 where t1.id = t2.id and
 t1.id >" + str(start_id)+ " and t1.id <" + str(end_id) +  " and
 t2.id >" + str(start_id) +" and t2.id  <" + str(end_id) + ") as
 my_table"
    df_select = spark.read.jdbc(url="jdbc:postgresql://localhost
:5432/test",table=_select_sql,properties=properties)
```

```
    df_select.show()
    output_udf_float = udf(_response, IntegerType())
    df4 = df_select.withColumn('exec',output_udf_float('feature'
).alias('exec_output'))
    df4.show()
    query_res = df4.select('exec').rdd.flatMap(lambda x: x).
collect()
    count = 0
    correct = 0
    for j in range(len(pred_list)):
        if query_res[j] == 1:
            count+=1
        if query_res[j] == truth_list[j + i*ep_size] and
pred_list[j] == 1:
            correct+=1

    prec = correct*1.0/count
    recall = correct* 1.0 /truth
    total_recall = prev_recall + recall
    prev_recall = total_recall
    if (prec + total_recall) != 0:
        f1 = 2* prec * total_recall / (prec + total_recall)
    else:
        f1 = 0
    all_f1.append(f1)
  if f1 >= quality_requirement:
    break
query_res.show()
```

B Proof of Theorem 5.2

Lemma 1. *If the probability of a tuple $t_k \in Ans_{w-1}$ increases in epoch w, then the threshold $\mathcal{P}_\tau^w$ remains the same as $\mathcal{P}_\tau^{w-1}$ or increases (some tuples which were part of Ans_{w-1} can move out of Ans_w). In both cases, the $E(F_\alpha)$ measure of the answer set increases from $E(F_\alpha)$ of Ans_{w-1}.*

The threshold of epoch w (*i.e.*, $\mathcal{P}_\tau^w$) satisfies the following two conditions:

$$\mathcal{P}_\tau^w > \frac{\mathcal{P}_1 + \mathcal{P}_2 + ...\mathcal{P}_{\tau-1}}{\tau - 1 + k_2} \tag{B.1}$$

$$\mathcal{P}_{\tau+1}^w < \frac{\mathcal{P}_1 + \mathcal{P}_2 + ...\mathcal{P}_{\tau}}{\tau + k_2} \tag{B.2}$$

where k_2 is defined as Equation 5.5. Suppose the selection probability of a tuple $t_i \in Ans_{w-1}$ increased in epoch w from $\mathcal{P}_i$ to $\mathcal{P}_i + \Delta$. Depending on the value of Δ, if threshold remained the same, then $E(F_\alpha)$ measure of the answer set (as shown in Equation 5.5) increases as the denominator increases. If threshold changes, then it only increases with respect to $\mathcal{P}_\tau^{w-1}$. This in turn ensures that $E(F_\alpha)$ measure is higher than epoch $w-1$ as it starts to drop from an earlier tuple. In the following part of the proof, we denote the threshold probability of $\mathcal{P}_\tau^{w-1}$ by the notation $\mathcal{P}_m$ and the probability of $\mathcal{P}_{\tau+1}^{w-1}$ to be $\mathcal{P}_{m+1}$.

In the new epoch, considering the new probability value of t_i, the following condition holds:

$$
\begin{aligned}
&\mathcal{P}_m > \frac{(\mathcal{P}_1 + \mathcal{P}_2 + .. + \mathcal{P}_i + \Delta + ...\mathcal{P}_{m-1})}{(m-1+k+\Delta)}, where, k = \sum_{j=1}^{N} \mathcal{P}_j \\
&\Rightarrow \mathcal{P}_m(m-1+k+\Delta) > (\mathcal{P}_1 + \mathcal{P}_2 + .. + \mathcal{P}_i + \Delta + ...\mathcal{P}_{m-1}) \\
&\Rightarrow \mathcal{P}_m(m-1+k) + \Delta\mathcal{P}_m > \mathcal{P}_1 + \mathcal{P}_2 + ... + \mathcal{P}_{m-1} + \Delta \\
&\Rightarrow \mathcal{P}_m(m+k) - (\mathcal{P}_1 + \mathcal{P}_2 + ... + \mathcal{P}_{m-1}) > \Delta(1 - \mathcal{P}_m)
\end{aligned} \tag{B.3}
$$

From Equation B.2, we get that $\mathcal{P}_m(m-1+k) - (\mathcal{P}_1 + \mathcal{P}_2 + ... + \mathcal{P}_{m-1}) > 0$. Equation B.3 can be simplified as follows:

$$
\begin{aligned}
&\Delta(1 - \mathcal{P}_m) < \mathcal{P}_m(m+k) - (\mathcal{P}_1 + \mathcal{P}_2 + ... + \mathcal{P}_{m-1}) \\
&\Rightarrow \Delta < \frac{\mathcal{P}_m(m+k) - (\mathcal{P}_1 + \mathcal{P}_2 + ... + \mathcal{P}_{m-1})}{1 - \mathcal{P}_m}
\end{aligned} \tag{B.4}
$$

Hence, the threshold remains constant as long as Δ value is less than $\frac{\mathcal{P}_m(m+k) - \sum_{i=1}^{m-1} \mathcal{P}_i}{1-\mathcal{P}_m}$, otherwise it will be increased. In both these cases, the numerator of $E(F_\alpha)$-measure of Equation 5.2 increases resulting in the increment of the F_1 measure of the answer set.

Lemma 2. *If the selection probability value of $t_k \in Ans_{w-1}$ decreases in epoch w but still remains higher than $\mathcal{P}_\tau^{w-1}$, then the threshold $\mathcal{P}_\tau^{w}$ remains the same as $\mathcal{P}_\tau^{w-1}$ or decreases. This implies that the tuples which were already part of Ans_{w-1} will still remain in Ans_w and some new tuples might be added to Ans_w. In both the cases, the $E(F_\alpha)$ measure of Ans_w is lower than $E(F_\alpha)$ of Ans_{w-1}.*

Suppose the probability of a tuple t_i decreases from $\mathcal{P}_i$ to $\mathcal{P}_i - \Delta$. According to Equation B.1, if the tuple t_m is still the threshold tuple in the new epoch, then the following condition

must hold:

$$\mathcal{P}_m > \frac{(\mathcal{P}_1 + \mathcal{P}_2 + ...\mathcal{P}_{m-1} - \Delta)}{(m - 1 + k - \Delta)} \tag{B.5}$$

The right-hand side of the above inequality $\frac{(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_{m-1}-\Delta)}{(m-1+k-\Delta)}$ is reduced from the previous epoch, *i.e.*, $\frac{(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_{m-1})}{(m-1+k)}$, this inequality will always hold. This implies that no further tuples are included in the answer set but the joint probability value of one of the existing tuple is reduced. This reduces the $E(F_1)$ measure of the answer set as the value in the numerator is reduced from the previous epoch.

Lemma 3. *If the selection probability value of a tuple $t_i \in Ans_{w-1}$ decreases in epoch w and becomes less than $\mathcal{P}_\tau^{w-1}$, then the threshold $\mathcal{P}_\tau^{w}$ can increase or decrease. In both the cases, the $E(F_\alpha)$ measure of Ans_w is lower than $E(F_\alpha)$ of Ans_{w-1}.*

Suppose the selection probability of tuple t_i is reduced from $\mathcal{P}_i$ to $\mathcal{P}_i - \Delta$ in the new epoch. The new probability $\mathcal{P}_i - \Delta$ is lower than the previous threshold probability of epoch $w - 1$. Suppose t_m was the threshold tuple, hence, $\mathcal{P}_m$ was higher than $\frac{(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_{m-1})}{(m-1+k)}$ (according to Equation B.1). Let us denote the numerator by X and the denominator by Y.

Since t_i moved out of the answer set, the new numerator will be $X - \mathcal{P}_i$ and new denominator will be $Y - 1 - \Delta$. So the new value in the right-hand side of inequality B.1 becomes $\frac{X-\mathcal{P}_i}{Y-1-\Delta}$. If the value $\frac{X-\mathcal{P}_i}{Y-1-\Delta}$ becomes greater than $\frac{X}{Y}$, then the inequality B.1 will not hold and threshold value will be increased. This condition arises when Δ value is more than $\frac{\mathcal{P}_i}{\mathcal{P}_m} - 1$. Note that $\frac{\mathcal{P}_i}{\mathcal{P}_m}$ is more than 1, since $\mathcal{P}_i > \mathcal{P}_m$ as the tuple with probability $\mathcal{P}_m$ was the last tuple in the answer set. This implies that the threshold either remains the same and one of the previous tuples moved out of the answer set. The numerator in Equation 5.2 decreases resulting in the reduction of $E(F_1)$ measure.

Based on above lemmas we conclude that, given a tuple $t_i \in Ans_{w-1}$, the $E(F_\alpha)$ measure of Ans_w will increase with respect to Ans_{w-1} only if the selection probability of t_i increases in

epoch w.

Lemma 4. *If the selection probability value of $t_i \notin Ans_{w-1}$ increases in epoch w and becomes higher than $\mathcal{P}_\tau^{w-1}$, then the threshold $\mathcal{P}_\tau^{w}$ remains the same as $\mathcal{P}_\tau^{w-1}$ or increases (i.e., some tuples which were part of Ans_{w-1}, might move out of Ans_w). In both the cases, the $E(F_\alpha)$ of Ans_w will be higher than $E(F_\alpha)$ of Ans_{w-1}.*

Let t_m be the threshold tuple of epoch $(w-1)$. Hence, $\mathcal{P}_m$ was higher than $\frac{(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_{m-1})}{(m-1+k)}$, according to Equation B.1. Let us denote this fraction by $\frac{X}{Y}$. Suppose the selection probability value of a tuple $t_i \notin Ans_{w-1}$, increases from $\mathcal{P}_i$ to $\mathcal{P}_i + \Delta$ in epoch w. Hence, in epoch w, the right-hand side of Equation B.1 will be updated to $\frac{X+\mathcal{P}_k}{Y+1+\Delta}$. If this new value is more than previous value of $\frac{X}{Y}$, and $\mathcal{P}_m$ is less than this new value, then the tuple t_m moves out of the answer set. Now we show that, in both cases, where threshold remains same or gets increased, F_1 measure of the answer set will increase. First we consider the case where threshold is increased.

Suppose t_m was the threshold tuple in previous epoch (*i.e.*, epoch *(w-1)*), and in epoch w it moved out of the answer set due to the inclusion of tuple t_k. This implies that in epoch $w-1$, the answer set consisted of the objects with probability values $(\mathcal{P}_1, \mathcal{P}_2, ...\mathcal{P}_{m-1}, \mathcal{P}_m)$ and in epoch w, the answer set became $(\mathcal{P}_1, \mathcal{P}_2, ...(\mathcal{P}_k + \Delta), ...\mathcal{P}_{m-1})$. The $E(F_1)$ measure of epoch $i-1$ will be $\frac{(1+\alpha)(\mathcal{P}_1+\mathcal{P}_2...\mathcal{P}_m)}{\alpha(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_N)+m}$, and let us denote this fraction by $\frac{X}{Y}$. The new F_1 measure of epoch w will be $\frac{(1+\alpha)(\mathcal{P}_1+\mathcal{P}_2...+\mathcal{P}_k+\Delta+...\mathcal{P}_{m-1})}{\alpha(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_N)+m+\Delta}$, which can be written as $\frac{X+(\mathcal{P}_k+\Delta)-\mathcal{P}_m}{Y+\Delta}$.

As tuple t_m was the threshold in epoch $w-1$, the Equation B.1 holds for tuple t_m as shown below.

$$\begin{aligned} \mathcal{P}_m &> \frac{(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_{m-1})}{(m-1+(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_N))} \\ \Rightarrow \mathcal{P}_m &> \frac{(X-\mathcal{P}_m)}{(Y-1)}, \quad \Rightarrow \mathcal{P}_m > \frac{X}{Y} \end{aligned} \tag{B.6}$$

In epoch $w-1$, if tuple t_m was not part of the answer set, which implies that in epoch $w-1$, the following condition was true:

$$\begin{aligned}
&\mathcal{P}_m < \frac{(\mathcal{P}_1 + \mathcal{P}_2 + ...\mathcal{P}_{m-1} + \mathcal{P}_k + \Delta)}{(m-1+(\mathcal{P}_1 + \mathcal{P}_2 + ...\mathcal{P}_N) + \Delta)} \\
&\Rightarrow \mathcal{P}_m < \frac{(X - \mathcal{P}_m + (\mathcal{P}_k + \Delta))}{(Y-1+\Delta)}, \quad \Rightarrow \mathcal{P}_m < \frac{X + \mathcal{P}_k + \Delta}{Y + \Delta}
\end{aligned} \tag{B.7}$$

Using Equations B.6 and B.7 and simplifying the expression of F_1 measures (Equation 5.2) of epoch w and $w-1$, we can show that $E(F_1)$ measure of epoch w is higher than $E(F_1)$ measure of epoch $w-1$.

Lemma 5. *If the selection probability value of $t_k \notin Ans_{w-1}$ increases in epoch w but does not become higher than $\mathcal{P}_\tau^{w-1}$, then the threshold $\mathcal{P}_\tau^{w}$ remains the same as $\mathcal{P}_\tau^{w-1}$ or decreases. In both the cases, the $E(F_\alpha)$ of Ans_w will remain the same as that of Ans_{w-1}.*

Let t_m be the threshold object of the answer set in the previous epoch. This implies that $\mathcal{P}_m$ is greater than $\frac{(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_{m-1})}{(m-1+k)}$, according to Equation B.1. Let us denote this fraction by $\frac{X}{Y}$. Let us consider an object o_k which was outside of the answer set at the end of epoch $w-1$. Let the joint probability value of tuple t_k is increased from $\mathcal{P}_k$ to $\mathcal{P}_k + \Delta$ in epoch w. In the new right-hand side of Equation B.1, numerator stays the same as previous epoch, as no extra tuples were added to the answer set, whereas the denominator is increased from Y to $Y + \Delta$. Hence, the new right-hand side $\frac{X}{Y+\Delta}$ becomes smaller than the previous right-hand side value of $\frac{X}{Y}$, which implies that the tuple t_m will remain in the answer set.

Let us consider the condition of threshold, related to tuple t_{m+1} (Equation B.2). In epoch $w-1$, the following condition was true for object o_{m+1}: $\mathcal{P}_{m+1} < \frac{(\mathcal{P}_1+\mathcal{P}_2+...\mathcal{P}_m)}{(m+k)}$, where, $k = \sum_{i=1}^{N} \mathcal{P}_i$. Let us consider the numerator and denominator as X and Y respectively. In new epoch (*i.e.*, epoch w), the denominator Y will be increased as the joint probability of object

$\mathcal{P}_k$ is increased from $\mathcal{P}_k$ to $\mathcal{P}_k+\Delta$. This implies that the right-hand side $\frac{X}{Y+\Delta}$ became smaller as compared to the previous value of $\frac{X}{Y}$. We can conclude that there is a possibility that t_{m+1} can become part of the answer set if $\mathcal{P}_{m+1}$ becomes more than the value of $\frac{X}{Y+\Delta}$. This implies that the threshold will remain the same or it will include more objects to the answer set. If the answer set remains the same then the $E(F_1)$ remains the same as there is no new addition. If the answer set includes more tuples then the expression of $E(F_1)$ measure of the answer set increases.

Lemma 6. *If the selection probability value of $t_k \notin Ans_{w-1}$ decreases in epoch w, then the threshold $\mathcal{P}_\tau^w$ remains the same as $\mathcal{P}_\tau^{w-1}$ or increases. In these cases, the $E(F_\alpha)$ measure of Ans_w is higher than or equal to the $E(F_\alpha)$ measure of Ans_{w-1}.*

Let t_m be the threshold tuple of epoch $w-1$. The value of $\mathcal{P}_m$ was higher than the value of $\frac{(\mathcal{P}_1+\mathcal{P}_2+\ldots\mathcal{P}_{m-1})}{(m-1+k)}$, according to Equation B.1. Let us denote it by $\frac{X}{Y}$. Let t_k be the tuple which was not in the answer set in epoch $(w-1)$ and the probability value of this tuple is decreased from $\mathcal{P}_k$ to $\mathcal{P}_k - \Delta$. In the new right-hand side of Equation B.1, the numerator stays the same as no extra tuple was added to the answer set, the denominator is reduced from Y to $Y-\Delta$. The new value of right-hand side $\frac{X}{Y-\Delta}$ increases from the previous right-hand side $\frac{X}{Y}$. This implies that there is a possibility that the tuple t_m moves out of the answer set in the new epoch. If the answer set remains the same, then in the expression of $E(F_1)$ measure of the answer set (*i.e.*, Equation 5.2), the denominator reduces. The numerator remains the same as the answer set remains the same. If it cause the tuple t_m to be out of the answer set in epoch e_w, then the numerator reduces but at the same time denominator reduces by larger amount as the size of the answer set is decreased. From the expression of $E(F_1)$ measure, we observe that it increases the $E(F_1)$ measure of the answer.

From Lemmas 4, 5, and 6, given a tuple $t_k \notin Ans_{w-1}$, we conclude that the $E(F_\alpha)$ measure of Ans_w increases or remains the same with respect to Ans_{w-1} but it never decreases.