UNIVERSITY OF CALIFORNIA,
IRVINE

Optimizing Query Processing for Data-Intensive Computation

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Yiming Lin

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

2023

DEDICATION

To my beloved family.

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

ACKNOWLEDGMENTS

It is a long but a joy and rewarding journey for my Ph.D., thanks to the support of many individuals who've been instrumental in guiding me across the finish line.

First and foremost, I am profoundly thankful to my advisor, Professor Sharad Mehrotra, whose patience, guidance, and mentorship have been invaluable to both my personal and professional development. His continuous support, encouragement, and insightful advice have allowed me to delve into the exhilarating world of research and explore new territories with confidence. It has been an immense privilege to work under his guidance. I have learned a multitude of invaluable things from Sharad. Through our academic discussions, he has taught me how to develop complex ideas from simple beginnings, how to think creatively, and how to approach challenges with an enthusiastic and positive attitude. Moreover, his patient mentorship has empowered me to guide others, just as he has guided me. In terms of my career development, Sharad's support has been indispensable. He has not only guided me through the intricacies of academic research but also help me formulate a clear and focused path towards my professional goals. I am very fortunate to have had such opportunities and am deeply grateful for Sharad's mentorship. I extend my best wishes to Sharad and express my sincere hope to maintain our friendship in the future.

I would also like to express my deep gratitude to Professor Michael J. Carey, Professor Chen Li, and Professor Nalini Venkatasubramanian for their valuable contributions as part of my committee in advancement, thesis topic defense, and final thesis review. I had the privilege of working with Professor Venkatasubramanian on the LOCATER and T-Cove projects, and I have continually been impressed by her ability to approach problems from a multitude of creative perspectives, which has greatly inspired me. I wish to extend heartfelt thanks for the invaluable feedback on my work, advice on presentation skills, and recommendations for further study from Nalini, Mike, Chen, and Faisal. Their guidance has significantly enriched my academic journey.

UCI ISG Group is a brilliant and supportive group of academics, and I consider myself fortunate to have been an ISGer over the past years. This experience has enriched me with both valuable research collaborations and lasting friendships. GuangXue, a wonderful friend, has made even the most mundane conversations enjoyable, spanning topics from research to daily life, and encouraged me during challeging times. Peeyush has amazing DBMS knowledge and engineering skills, and I am fortunate to work with him on several projects. Peeyush is also a very good friend with a quirky sense of humor. Dhrub is always smiling and is supportive of everything. I would also like to thank Guoxi, Nada, Sameera, Roberto, Primal, Sriram, Rithwik, Ashwin, Yicong, Qiushi, Pratyoy, Abdul, Daokun, Fangqi, Vishal and Shantanu. Each of you has significantly enriched my journey during my Ph.D., and our shared experiences have indeed been unforgettable. Thank you for the camaraderie, the stimulating conversations, and the memories we've created together. I will miss you all and wish you the very best in everything that comes your way.

Outside the ISG group, I would also like to express my words of gratitude to Yeye He, my

mentor during my internship at Microsoft Research. I had the privilege of being mentored by a remarkable individual. His patient guidance, profound knowledge base, and insightful suggestions were instrumental in my progress and successful completion of a significant project, AutoBI.

Lastly, I wish to express my profound gratitude to my parents, Ying and Shuhui, whose support has been unyielding. Their daily concern for my well-being, their encouragement during challenging times, and their interest in sharing every aspect of life with me have been nothing short of inspirational. Their unwavering belief in my decisions and their immense understanding has played a significant role in my journey. Without their relentless encouragement, the completion of my Ph.D. would have been a much more formidable task.

I would like to thank VLDB Endowment for permission to include the portions of Chapter 3, Chapter 5, and Chapter 6 as part of my thesis, which were originally published in the Proceedings of the VLDB Endowment. The work reported in this thesis is based on research sponsored by HPI and DARPA under Agreement No. FA8750-16-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. The work is partially supported by NSF Grants No. 1527536, 1545071, 2032525, 1952247, 1528995 and 2008993.

Portion of Chapter 3 of this dissertation is a reprint of the material as it appears in Proceedings of the VLDB Endowment, volume 17, issue 3, 2024, used with permission from VLDB Endowment. The co-authors listed in this publication are Yiming Lin, Sharad Mehrotra.

Portion of Chapter 5 of this dissertation is a reprint of the material as it appears in Proceedings of the VLDB Endowment, volume 14, issue 3, 2021, used with permission from VLDB Endowment. The co-authors listed in this publication are Yiming Lin, Daokun Jiang, Roberto Yus, Andrew Chio, Georgios Bouloukakis, Sharad Mehrotra, Nalini Venkatasubramanian.

Portion of Chapter 6 of this dissertation is a reprint of the material as it appears in Proceedings of the VLDB Endowment, volume 14, issue 12, 2021, used with permission from VLDB Endowment. The co-authors listed in this publication are Yiming Lin, Pramod Khargonekar, Sharad Mehrotra, Nalini Venkatasubramanian.

VITA

Yiming Lin

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of Calfornia Irvine *Irine, California*

Master of Science, Computer Science and Technology **2017**
Harbin Institute of Technology *Harbin, China*

Bachelor of Science, Computer Science and Technology **2015**
Harbin Institute of Technology *Harbin, China*

RESEARCH EXPERIENCE

Graduate Research Assistant **2019–2023**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2017–2019**
University of California, Irvine *Irvine, California*

PUBLICATIONS

Efficient Entity Resolution on Heterogeneous Records. **2019**
Yiming Lin, Hongzhi Wang, Jianzhong Li, Hong Gao

IEEE Transactions on Knowledge and Data Engineering

Data Source Selection for Information Integration in Big Data Era. **2019**
Yiming Lin, Hongzhi Wang, Jianzhong Li, Hong Gao

Information Sciences

Semiotic: Bridging the Semantic Gap in IoT Spaces. **2019**
Sumaya Almanee, Georgios Bouloukakis, Daokun Jiang, Sameera Ghayyur, Dhrubajyoti Ghosh, Peeyush Gupta, Yiming Lin, Sharad Mehrotra, Primal Pappachan, Eun-Jeong Shin, Nalini Venkatasubramanian, Guoxi Wang, Roberto Yus

ACM BuildSys

LOCATER: Cleaning WiFi Connectivity Datasets for Semantic Localization. **2021**
Yiming Lin, Daokun Jiang, Roberto Yus, Andrew Chio, Georgios Bouloukakis, Sharad Mehrotra, Nalini Venkatasubramanian

Proceedings of the VLDB Endowment

T-Cove: An exposure tracing System based on Cleaning Wi-Fi Events on Organizational Premises. **2021**
Yiming Lin, Pramod Khargonekar, Sharad Mehrotra, Nalini Venkatasubramanian

Proceedings of the VLDB Endowment

Auto-BI: Automatically Build BI-Models Leveraging Local Join Prediction and Global Schema Graph. **2023**
Yiming Lin, Yeye He, Surajit Chaudhuri

Proceedings of the VLDB Endowment

Robust Occupancy Computation Based on WiFi Connectivity Events. **2023**
Rithwik Kerur, Yiming Lin

IEEE International Conference on Data Engineering (ASTRIDE Workshop)

ZIP: Lazy Imputation during Query Processing. **2024**
Yiming Lin, Sharad Mehrotra

Proceedings of the VLDB Endowment

PLAQUE: Automated Predicate Learning at Query Time. **2024**

Yiming Lin, Sharad Mehrotra

Technical Report in University of California, Irvine

SOFTWARE

LOCATER `https://github.com/yiminl18/LOCATER.git`

An indoor localization system implemented in Java based on data cleaning technologies. LOCATER is able to predict the location of a person inside a building using WiFi connectivity events without the need for additional hardware and software.

ZIP `https://github.com/yiminl18/QDMIDB.git`

ZIP implements a query-time data cleaning system in Java that is able to clean minimal missing data to answer the given query.

PLAQUE `https://github.com/yiminl18/filterplusdb.git`

PLAQUE is a data processing system that takes a query plan by a given optimizer as input, and learns new predicates to skip unnecessary computations at query run time.

ABSTRACT OF THE DISSERTATION

Optimizing Query Processing for Data-Intensive Computation

By

Yiming Lin

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Sharad Mehrotra, Chair

Today, data-driven analysis and applications exploit vast streams of data that are perpetually generated and collected from numerous data sources. Such a surge in data production, which has reached over 10,000 Exabytes [26], is driving transformative advancements in sectors such as transportation, emergency services, and health and wellness. Before data is queried and used by downstream data-driven analytical tasks, various computationally-intensive computations often need to be performed. Such tasks include data cleaning, data integration, and/or data enrichment operations that often execute expensively AI/ML models incur non-trivial costs. Such computationally expensive tasks can often not be performed at data ingestion time due to the rate at which data is produced. Periodic, offline computation is also infeasible due to the volume of data. Query processing in such a situation requires careful incorporation and co-optimization of computationally expensive operations into the query engine that can streamline query analysis and enhance execution efficiency in terms of time and resources.

The goal of this thesis is to develop mechanisms to support computationally expensive operations (e.g., enrichment, imputations, information extraction, data interpretation) within data management systems in order to support interactive analysis. While the techniques developed in the thesis have wide applicability, our focus is on emerging smart space appli-

cations. Smart spaces consist of sensor-embedded physical spaces that capture and represent the dynamic state of the physical infrastructure, and that of people interacting with the physical infrastructure, and with each other. Data management in smart spaces opens several new challenges one of which is the ability to support interactive analytics on very large volumes of data being captured at large velocities. The problems studied in this thesis draw its motivation from such challenges.

In Chapter 3 we develop a query-time missing value imputation framework, entitled La*Z*y *I*mputation during query *P*rocessing (ZIP), that modifies relational operators to be imputation-aware in order to minimize the joint cost of imputing and query processing. The modified operators use a cost-based decision function to determine whether to invoke imputation or to defer it to downstream operators to resolve missing values. The modified query processing logic ensures results with deferred imputations are identical to those produced if all missing values were imputed first. ZIP includes a novel outer-join-based approach to preserve missing values during execution, and a bloom filter-based index to optimize the space and running overheads. Extensive experiments on both real and synthetic data sets demonstrate orders-of-magnitude improvements in query processing when ZIP-based deferred imputations are used.

In Chapter 4, we present a system for automated *P*redicate *L*e*A*rning at *QU*ery tim*E* (PLAQUE), that automatically infers new predicates while running queries in order to accelerate query execution. PLAQUE represents a significant departure from prior work on learning predicates which are either limited to queries containing selection conditions on certain columns (e.g., columns involved in a join in the query), or require statistics to be collected and maintained from data, such as range set. We identify several opportunities to learn predicates from various query conditions, such as aggregation, equi join, theta join, and group by/having conditions at query time. In PLAQUE, learned predicates are pushed down to the optimal positions in a given query plan tree in order to maximize their bene-

fits. A novel partial-order based approach is developed for such a purpose. Furthermore, we introduce a pre-learning technique for predicate inference before query optimization, which synergistically combines with the runtime learning approach of PLAQUE to further enhance performance. Comprehensive evaluations on both synthetic and real datasets demonstrate that our learned predicates accelerate query execution by an order of magnitude, and the improvements are even higher (two orders of magnitude) when computationally expensive operators (imputations/enrichment) in the form of User-Defined Functions (UDFs) are utilized in queries. PLAQUE, thus, significantly benefits data-driven analytical applications.

In Chapter 5, we shift the interest to applying techniques developed in the thesis to data processing and query analytics in smart spaces. We first develop LOCATER, an indoor localization solution based on WiFi connectivity data. LOCATER is zero-cost, accurate (90% accuracy), and passive without the need to install any new hardware in the building or new software on users' phones. LOCATER has already been deployed and is operational in the USA and India, across three distinct locations (UCI, BSU, Plaksha), and in over 40 UCI buildings for four years. LOCATER serves as a representative compute-intensive task in building smart space applications. We conduct a case study by building two applications using LOCATER, occupancy, and contact tracing applications. Our case study clearly demonstrates the benefits of using our query processing techniques – ZIP and PLAQUE in building campus scale smart space applications. For instance, we show queries without these optimizations, which were impractical to execute interactively can be used for interactive analytics.

Chapter 1

Introduction

In a traditional data processing pipeline, the system transforms and integrates data from multiple sources before storing it in a database. Nowadays, streams of data are continuously generated and captured by various data sources, such as devices, sensors, applications, and services, all connected across multiple heterogeneous networks. Numerous studies indicate a rapid surge in the data created by hundreds of billions of interconnected devices used by several billion people, leading to the generation of over 10,000 Exabytes of data [26]. The large volume of data opens new opportunities for transformational improvements in domains like transportation, emergency response, and health and wellness.

Before such abundant data can be queried and used for downstream data-driven analytical tasks, various computationally-expensive operations may need to be performed. For instance, the real-world data is often dirty with possible missing data, incorrect data, inconsistent data, or duplicated data [71], and thus data cleaning [78, 116, 102, 33] as a potentially compute-intensive task is often performed to improve the quality of data to enable data-driven decision making. Another compute-intensive task is data enrichment [48], which associates a tag/label with a data object to support the downstream analytics. Data

enrichment is often expensive since it requires the use of computation and I/O intensive machine learning/artificial intelligence (ML/AI) model training and deployment [82, 87, 35].

Performing compute-intensive tasks given large and rapidly generated data sets poses significant challenges. Consider a use case which motivates our work. At UC Irvine, for the past 4 years, we have been using continuously generated WiFi connectivity data captured over a campus wireless networks for fine-grained localization using a WiFi-based localization framework entitled LocatER [79]. LocatER exploits a person's recent connection history to predict the room a person is in given the access point a person is connected to (which corresponds to an imputation problem [79]). LocatER takes roughly 400ms per event for such an imputation.[1] With 1000s of WiFi access points, about 30,000+ individuals connected to the network and tens of thousands of WiFi events per second, it would take over one hour to process one second of data collected from the WiFi infrastructure during peak load. Clearly, it would be impossible to perform missing location enrichment/cleaning tasks on the complete data sets or as data arrives on-the-fly when the compute-intensive tasks are expensive.

In this thesis, our objective is to develop solutions to support expensive compute-intensive tasks for downstream query processing. We study the problem in the context of relational databases (RDB) and SQL as a query language for specifying analytic tasks. We focus on the following two main directions of exploration. First, we consider a lazy approach to implementing expensive pre-processing tasks, such as enrichment and/or imputations. Such tasks are delayed and lazily evaluated during query processing, such that any complex computation is only performed on a subset of data required for analysis. Second, we develop novel approaches to learning new query predicates during query execution to accelerate query processing. The learned predicates allow us to significantly reduce query processing costs by skipping unnecessary work at query time.

[1] This is based on a 16 core 2.50GHz Intel Xeon CPU, 64GB RAM, and 1TB SSD.

Lazy Execution. In this work, we consider a common data cleaning problem – missing value imputation as a compute-intensive task, which could be probably expensive. As discussed above, using LOCATER to impute one second of data collected from the WiFi infrastructure during peak load would take over one hour of processing for an offline cleaning approach, which is prohibitively costly if the volume of data is large and cost per imputation is high. Likewise, an online approach that imputes location value as soon as data is ingested is clearly infeasible. Instead, we adopt an alternate query-time approach that cleans data lazily when the need arises. Motivated by similar requirements as the example above, a query-time approach to cleaning/imputation has become popular in several recent studies [25, 24, 49, 34, 39, 48].

Query time cleaning offers several benefits. It significantly reduces the wasted effort and computational resources by cleaning only parts of the data actually needed in analysis instead of indiscriminately cleaning the entire dataset. This is especially important when cleaning is expensive and/or datasets are very large, making cleaning the data fully infeasible. Predicting the dataset analysts might use (and cleaning only such data as a pre-analysis step) is often not feasible (e.g., when a common analysis operation consists of ad-hoc queries on the data) [28]. In such situations, the only recourse is to support data cleaning with query processing.

Query-time data cleaning opens new challenges, the prominent part of which is to minimize cleaning performed during query processing to reduce latency. We develop *ZIP*, a la*Z*y *I*mputation query *P*rocessing approach that exploits query semantics to reduce the cleaning overhead. When processing records with missing values, ZIP may delay imputations until later - such a lazy approach to imputing can be beneficial if the record with the missing value gets eliminated in the query tree, thus, avoiding imputations unnecessary for answering the query. Delaying imputations will increase processing costs if imputation could not be avoided. ZIP, given a query plan for an SQL query, develops an execution strategy that minimizes

the overall (joint) cost of imputing missing data and executing the query.

Learning Predicates. In this part of the thesis, we seek a new approach to query processing, entitled *P*redicate *LeA*rning at *QU*ery tim*E* (PLAQUE), that automatically learns predicates *during* query execution (beyond those listed explicitly) in order to filter out tuples that would not result in any query results as early as possible during query processing. To illustrate the key idea behind PLAQUE, we examine a slightly modified and simplified version of TPC-H Query Q-10 that includes a theta-join condition. In this query, the predicates o_orderdate $<$ '1993-01-01' and p_brand = ':10' can be pushed down to orders and part tables. However, the query contains no predicates on the lineitem table that could be used to prune away non-matching tuples in lineitem that do not result in any query results. Thus, any query plan will scan over all records in the lineitem table.

SELECT MAX(l_discount)
FROM part, lineitem, orders
WHERE p_retailprice $<$ l_extendedprice **AND** o_orderkey = l_orderkey **AND** o_orderdate $<$ '1993-01-01' **AND** p_brand = ':10'

Consider that all records in the lineitem table that result in an answer satisfies a predicate $l\_discount > 0.7$. If the system could learn such predicates, it could significantly accelerate query execution by pushing such predicates down in the query tree in order to filter records in the lineitem table. Only a small fraction of lineitem records will need to join with the orders and parts tables resulting in significant savings.

Several prior works have explored learning additional predicates to reduce downstream processing, other than those specified explicitly in queries. Several such approaches learn predicates prior to query execution based on exploiting query predicates on join columns (e.g., [29, 88, 52, 107]. For instance, if there were a query predicate l_orderkey $<$ 5) in a TPCH query, such techniques would infer a predicate such as o_orderkey $<$ 5) to speed up query

execution. Such techniques are, however, of limited applicability since in queries with equi joins, additional predicates on the join column are rare, as observed in several real datasets and synthetic benchmarks. For instance, none of the queries in TPC-H [19] or TPC-DS [18] contain additional predicates in join columns in equi joins. As such, for such queries, above mentioned techniques do not help reduce query execution costs. Another work [67] maintains data statistics (e.g., min and max of columns) at the data block level. Such statistics are used for sideways information passing over equi joins and are effective in accelerating query execution, especially in big-data systems such as Hive [106] or Pig [91] where data is partitioned across clusters. The work, however, is limited to equi joins. An alternate strategy is to infer predicates to add to queries on the fly during query execution. Such a strategy has previously been explored in [62, 121, 93] largely in the context of hash-joins in the main-memory database setting. In particular, a hash join $R \bowtie S$ over relations R and S first creates a hash table for one relation, (a.k.a., build relation). For each tuple t from the other relation, (a.k.a., probe relation), the hash join will use the join key value in t to retrieve the matching tuples in the hash table. Such strategies build summarization data structures, such as bloom filters [7], for the build table and use them to skip tuples in the probe table. We note that approaches that learn predicate prior to execution [29, 88, 52, 107], and those that learn predicates during execution, can be considered complementary - they can both be used in conjunction.

In Chapter 4 of the thesis, we propose PLAQUE that like [62, 121, 93] learns predicates to add to query during query processing. Unlike prior work on which it builds, PLAQUE considers a much more aggressive and adaptive approach to learning predicates. PLAQUE devises ways to infer new predicates from *a range of relational operators* with a variety of implementations (not limited to just hash-joins). In PLAQUE predicates are inferred from aggregation operators such as min and max, theta-joins, equi-joins, group-by operators, and having conditions, within a given query. PLAQUE does not rely on the existence of predicates on the join columns in the original query (as is the case with [29, 88, 52, 107]),

enabling it to be applied to a broad set of queries including TPCH and TPC-DS queries. We conduct extensive experiments demonstrating that adding the learned predicates using our strategies can achieve significant improvement ranging from 2x-30x, especially in queries containing expensive User-Defined-Functions (UDFs) where improvement can be up to 100x in the SmartBench benchmark [54].

While the above techniques are general and useful in several domains where data-intensive computation is used, our particular interest is the smart space context, wherein smart physical infrastructure, such as a smart building and/or smart campus, interacts with humans immersed in the physical environment for a variety of tasks. Such tasks range from real-time control (e.g., improved occupancy-based heating/air-conditioning control, locating/finding people and/or resources such as empty rooms, dynamic contact tracing to determine who came across who at different places in campus) to analytical tasks such as evaluating building management policies, understanding group interactions between building/campus occupants, to exploiting workplace social networks to improve productivity. A key challenge in all of these applications is the need to localize a person both outside and more importantly inside buildings. Accurate locations of people enable a variety of essential location-based applications in smart spaces, such as determining occupancy of rooms, thermal control based on occupancy [21], determining density of people in a space and areas/regions of high traffic in buildings —applications that have recently gained significance for COVID-19 prevention and monitoring in workplaces [109, 55]. The technical challenge in localizing people inside the buildings is how to develop a zero-cost, accurate, and off-the-shelf solution without the additional hardware and software to enable a potentially widely used solution. To this end, we develop a series of data cleaning solutions based on WiFi connectivity data to locate users to *semantic* indoor locations such as buildings, regions, and rooms.

LOCATER: A Semantic Indoor Localization Solution using Data Cleaning

In Chapter 5 of the thesis, we offer a zero-cost indoor localization solution by exploring the

data cleaning challenges that arise in using WiFi connectivity data to locate users to semantic indoor locations such as buildings, regions, and rooms. WiFi connectivity data consists of sporadic connections between devices and nearby WiFi access points (APs), each of which may cover a relatively large area within a building. Our system, entitled semantic LOCATion cleanER (LOCATER), postulates semantic localization as a series of data cleaning tasks - first, it treats the problem of determining the AP to which a device is connected between any two of its connection events as a missing value detection and repair problem. It then associates the device with the semantic subregion (e.g., a conference room in the region) by postulating it as a location disambiguation problem. LOCATER uses a bootstrapping semi-supervised learning method for coarse localization and a probabilistic method to achieve finer localization. LOCATER does not require any new hardware to be installed in the environment or any participation of end users to install software on the phones. LOCATER is able to achieve 90% accuracy which is quite usable for many location-based applications. It is worth mentioning that The LOCATER has already been deployed and is operational in the USA and India, across three distinct locations (UCI, BSU, Plaksha), and in over 40 UCI buildings for four years. LOCATER performs similarly to a commercial system Occuspace in a UCI Library testbed.

While LOCATER is accurate, using LOCATER to impute/enrich one missing location is not cheap, which takes around 400ms. LOCATER provides a motivating context and use case for the research developed in this thesis. In particular, we conduct a case study to show how to use the lazy approach (i.e., ZIP) and the learned predicates (i.e., PLAQUE) to make location-based data processing and query processing interactive, without sacrificing the quality of the query results.

In this thesis, we make the following concrete contributions:

- We develop a query-time missing value imputation framework, entitled ZIP, that mod-

ifies relational operators to be imputation-aware in order to minimize the joint cost of imputing and query processing without sacrificing the quality of query results.

- We build PLAQUE that is able to learn new predicates from various query conditions, such as aggregate, equi join, theta join, and group by/having conditions during the execution of the query. The predicates learned during query execution are pushed down immediately to optimal positions in the query tree in order to benefit the remainder of the query execution by skipping rows early based on a novel partial-order based approach.

- We develop a zero-cost, accurate, and passive indoor localization system based on WiFi connectivity events by using data cleaning technologies, entitled LOCATER.

- We build two real applications using LOCATER, occupancy and contact tracing, and optimize the data/query processing inside these applications by using the proposed lazy approach ZIP and the learned predicates in PLAQUE to improve the performance of these applications in a real-world UCI WiFi testbed.

The remainder of the thesis is structured as follows. Chapter 2 describes the related work. In Chapter 3, we show the design and experimental results of ZIP. Chapter 4 describes the learned predicates in PLAQUE. In Chapter 5 we present our LOCATER technology. Chapter 6 conducts a case study where we build two applications using LOCATER, and use ZIP as well as PLAQUE to accelerate these real applications in the UCI campus testbed.

Chapter 2

Related Work

In this chapter of the thesis, we describe several works related to our thesis. First of all, we introduce the concept of data cleaning as well as several problems/challenges in data cleaning area in Section 2.1. Data cleaning is closely related to ZIP since ZIP explores a lazy approach to reducing the overhead of missing value imputation, which is a common data cleaning problem. LOCATER develops a set of data cleaning technologies, and it would benefit from the background of the data cleaning area. Second, we describe the query-time data cleaning in Section 2.2, the focus of which is the query-time data processing serving as an important related work to ZIP. Third, we describe the previous work of predicate inference in Section 2.3 to develop strategies to learn new predicates, which is closely related to PLAQUE. Finally, we describe two data systems that support query-time data processing and enrichment, TippersDB and EnrichDB, in Section 2.4.

2.1 Data Cleaning

Data warehousing, analysis, and mining technologies have brought about revolutionary changes to numerous scientific and business domains. Data-driven applications are increasingly popular to gain a better understanding and insight into new discoveries in disciplines such as medical imaging, biomedical engineering, earth sciences, and so on. The effectiveness of such data-driven technologies as decision support tools, data exploration, and scientific discovery tools, is closely tied to the quality of data to which such techniques are applied. It is well recognized that the outcome of the analysis is only as good as the data on which the analysis is performed [60]. Errors in analysis due to dirty data can snowball into incorrect decisions and, as a result, dirty data can be costly.

Data quality challenges can be both at the schema level or at the instance/data level. We focus below on instance-level problems that refer to the errors and inconsistencies in the actual data content which cannot be visible at the schema level. In this thesis, we mainly focus on instance/data level quality. Several well-recognized data cleaning problems at the instance level are entity resolution, data consistency, missing value imputation, etc.

Entity Resolution

The goal of entity resolution (ER) problem [45] is to find records in a data set that refer to the same object/entity across possibly different sources (e.g., product catalogs, web tables, files, etc.). Such a problem often arises when we merge multiple tables (from different data sources) to create a single merged table. Objects, in different data sources may not be referred to using a common identifier leading to the ER challenge.

To see a concrete example in Table 2.1, consider merging product information across multiple websites and/or product tables. The same products may be referred to very differently in

Table 2.1: Camera Products

	battery	Year	Price	provider
EOS 5D3	LP-E6	2014	1899	BestBuy
EOS 5D mark III	LPE6	2013	1934	eBay
EOS 5Diii	LPE6	2014	2010	amazon

.

two listings. Consider, for example, a comparison-shopping application that has crawled product information from different vendors such as eBay, BestBuy and Amazon to create an integrated product table. Such an application will require mechanisms to determine which products in the different sites refer to the same real-world entity. The table below shows possible entries for the same entity, Canon EoS 5D Mark 3 – collected from different sources. The representations of data across data sources are different and must be reconciled to support comparison shopping.

Missing Value Imputation

A large number of real-world datasets contain missing values. Reasons include human/-machine errors in data entry, unmatched columns in data integration, etc. Imputation approaches can roughly be characterized as statistics-based, rule-based, master-data-based, time-series-based, or learning-based approaches [78].

Statistics-based Imputation Approaches. Missing data can be missing completely at random or conditioned on existing values (observed and missing). Methods in the statistical community hinge on leveraging the inherent statistical properties and interdependencies within the dataset to estimate and replace missing values. One of the fundamental techniques of this approach is mean or median imputation [23]. In this technique, the missing value of a specific variable is replaced with the mean or median of the known values of that variable. This method provides a quick and easy way to handle missing data, however, it can lead to an underestimation of variances and correlations if the percentage of missing data is high.

Another line of methods is regression imputation [96, 100], which relies on the relationships between variables to estimate missing values. This method develops a regression model using the variable with missing data as the dependent variable and other related variables as predictors. This model is then used to predict the missing values. While regression imputation can offer more accurate estimations than mean or median imputation, it assumes a perfect correlation between variables, which might not always be the case, leading to over-fitted results. Multiple imputation [97] is a more advanced technique. It creates several imputed datasets, performs analysis on each of them, and then consolidates the results to generate a single inference. This method is particularly beneficial as it introduces variability across the imputed datasets, acknowledging and accounting for the uncertainty associated with imputed values.

Master-based Imputation Approaches. Knowledge-based missing value imputation leverages external databases or knowledge repositories [37] to fill in the gaps in a dataset. For instance, if a certain dataset has missing information about a person's job title, an external professional database like LinkedIn could be used to find and fill in that missing data. This method requires that there is a reliable and accurate external source of data available that's relevant to the missing values in the dataset. One major advantage is that it can provide highly accurate imputations if a good knowledge base is used. However, privacy concerns, accessibility, and compatibility between the dataset and the knowledge base are challenges that need to be considered. Crowdsource-based missing value imputation [114, 115, 95], on the other hand, utilizes the collective intelligence of a large group of people, typically via the internet, to fill missing values. This approach is often employed when the task requires human insight, and judgment, or when no automated imputation method is suitable. Crowdsourcing can be particularly effective when dealing with data that requires specific expertise to interpret, or when the missing values can be determined from the context provided by other data points. However, the accuracy of this method depends on the crowd's expertise, and maintaining quality control can be challenging.

Time Series Data Imputation. Imputation strategies in time series data [70, 30, 79] are often performed by learning patterns over *historical* data to forecast *current* missing values or using the correlation across the time series. Missing value imputations on time series data can be classified as either matrix-based or pattern-based, according to the underlying method they use. Matrix-based algorithms [74] operate on the assumption that the original data matrix, even with missing values, can be represented or approximated as a low-rank matrix. The goal then is to find this low-rank matrix that best fits the observed (non-missing) entries. Techniques often used for matrix completion include Singular Value Decomposition (SVD) and its variants, which decompose a matrix into the product of simpler matrices, filling in the missing values during the process. Pattern-based recovery techniques [83] assume that a high degree of similarity exists between series. When a block is missing in a base series, an algorithm would leverage the similarity to any number of reference series. The observed values in the reference series are treated as a query pattern. Any blocks matching that pattern may reveal candidate replacement values in the base series. An example is LOCATER [79] that imputes each missing location of a user at a time stamp by learning the users' pattern from historical data, such as the most often visited places and the closest group at certain time intervals.

Rule-based Missing Value Imputation. Rule-based imputation methods based on differential dependencies [102] or editing rules [43] often impute missing values by replacing them with corresponding values of similar objects. In particular, [102] tries to fill the missing data with the values of its neighbors who share the same information. Such neighbors could either be identified certainly by editing rules or statistically by relational dependency networks. In particular, [102] identifies a more extensive class of similarity neighbors, with value similarity relationships identified by similarity rules. By tolerance of small variations, the enriched (similarity) neighbors can fill in more missing data that are not revealed by the very limited equality neighbors. Based on the similarity rules, they further devise a randomized algorithm with an expected performance guarantee to impute the missing data.

Learning-based Imputation Approaches. Several learning-based techniques are used to impute missing data by learning the pattern or features from a clean training data set. Such techniques, as in [87], may use models such as generative adversarial network (GAN) [27] that can take over hours to train. Such techniques are, hence, not suitable for being called from within online analysis queries due to long latency due to training prior to imputation. Given such a limitation, several prior works have explored reducing the training time of learning-based methods. Miao et al [87] select a small representative sample (about 5%) to speed training by about 4x while maintaining imputation accuracy guarantees. Two widely used learning approaches, XGboost [35] and LightGBM [69] use histograms to speed up training. (Their APIs are available in standard Python packages [6].) Using histograms can reduce training time dramatically - e.g., Xgboost [6] using histograms takes 69.8s on a one-million size table with 10 attributes using 700k training samples and achieves an accuracy of 0.985, while it takes only 1s while training on the 20k samples used by Miao's approach [87] to achieve 0.971 accuracy. Sampling and histogram methods make learning-based methods amenable to online processing by dramatically reducing learning costs.

2.2 Query-time Data Cleaning

The query-time strategy has been explored in several data cleaning problems, such as conflicting values detection and repair using denial constraints [49, 38], entity resolution problems [24, 25].

Daisy [49] performs probabilistic repair of denial constraint violations on-demand, driven by the exploratory analysis that users perform. In particular, Daisy seamlessly integrates data cleaning into the analysis by relaxing query results and executes analytical query workloads over dirty data by weaving cleaning operators into the query plan.

QDA [24] studies the Query-Driven Entity Resolution problem in which data is cleaned "on the fly" in the context of a query. It developed a query-driven entity resolution framework that efficiently issues the minimal number of cleaning steps solely needed to accurately answer the given selection query. In particular, QDA exploits the specificity and semantics of the given SQL selection query to reduce the cleaning overhead by resolving only those records that may influence the query's answer. It computes answers that are equivalent to those obtained by first using a regular cleaning algorithm, and then querying on top of the cleaned data.

While QDA focuses on the selection query, QuERy [25] extends the query-time entity resolution solution to Selection-Projection-Join (SPJ) queries. The objective of QuERy is to efficiently and accurately answer complex SPJ queries issued on top of dirty data. The predicates in those queries may be associated with any attribute in the entity sets being queried. In particular, QuERy leverages the selectivities offered by the query predicates to reduce the amount of cleaning (by only deduplicating those parts of data that influence the query's answer) and thus, minimizes the total execution time of the query. There are two variants of QuERy, lazy-QuERy and adaptive-QuERy. The former uses a lazy architecture that attempts to avoid cleaning until it is necessary for the system to proceed. The latter is an adaptive cost-based technique that tries to devise a good plan to decide when to perform cleaning.

ImputeDB [34] explores a dynamic optimization strategy to design query plans for queries over relations with missing data. In particular, ImputeDB introduces two new operators - drop and impute. For any predicate where the condition being evaluated is over an attribute that may contain missing values, ImputeDB introduces one of these two operators. For any tuple that passes through the impute operator, ImputeDB will call the corresponding imputation function to resolve the tuple prior to passing it to the predicate in the original operator in the query tree. In contrast, for a drop operator, it will simply drop the tuples

whose corresponding attribute contains a missing value. Note that the placement of impute/-drop operators explores a trade-off between the accuracy of results and the corresponding overheads, especially when imputations can be expensive and dominate the query evaluation cost. For instance, if we only introduce the drop (impute) operator, query processing will be fast (slow).

2.3 Predicate Inference

Predicate inference refers to the problem of discovering new predicates besides a given SQL query to extend the predicate pushdown benefit to speed up query execution.

Some research works study sideways information passing (SIP) [62, 93, 67] over joins. [62, 93] focus on SIP during query execution by building summarization data structure such as bloom filter on the build table which is used to skip rows in the probe table. [67] studies SIP by creating data-induced predicates (diPs) from data before query execution by maintaining the range set data structure for each data block, and is suitable for equi join conditions. In particular, diPs are suitable for big-data systems, e.g., SCOPE, Spark, Hive, or Pig clusters that run SQL-like queries over large datasets, where data statistics such as the maximum and minimum value of each column are at different granularities of the input. Using data statistics, diPs convert predicates on a table to data skipping opportunities on the joining tables at query optimization time. The method begins by using data statistics to eliminate partitions on tables that have predicates. Second, using the data statistics of the partitions that satisfy the local predicates, the proposed diPs capture all of the join column values contained in such partitions. All of these steps happen during query optimization where the optimizer effectively replaces each table with a partition subset of that table; the reduction in input size often triggers other plan changes (e.g., using broadcast joins which eliminate a partition-shuffle) leading to more efficient query plans. diPs are applicable when data

is stored in blocks/clusters, and certain statistics such as maximum and minimum values are maintained for each data block. diPs focus on learning new predicates from equi join conditions, and they are not able to learn predicates from other query operators/conditions.

Another line of the work move predicates using magic sets [29, 88], algebraic equivalence [52], value-based pruning [107], or infer new predicates based on syntax-driven rewrite rules [120]. These techniques are applicable when there exist predicates on join columns. However, this case is not often seen especially for queries containing equi joins as observed in several real data sets and synthetic benchmarks since join columns tend to be opaque as system-generated identifiers [1] One representative work Sia [120] explores synthesized predicates to move predicates around the query blocks. Consider a SQL query:

```
SELECT * FROM A,B WHERE A.id=B.id
AND A.val+10>B.val+20 AND B.val+10>20
```

For this query, the optimizer may only move the third predicate (B.val + 10 > 20) below the join operator. It cannot push down the second predicate (A.val + 10 > B.val + 20) below the join operator since it depends on columns from both tables A and B. Sia deduces a new predicate $A.val > 20$ which can be inferred from the original query, and the new predicate is weaker than the original predicates (i.e., it accepts all the tuples that the original predicates accept). Adding this new predicate into the query will allow the optimizer to effectively push down the new predicate into table A and thus achieve a potentially better plan. Sia only works when there exist predicates on the join columns, such as $B.val + 10 > 20$.

Prior research has also explored the use of data properties, such as functional dependencies and column correlations, to accelerate query processing [32, 61, 72]. However, determining these properties can be computationally expensive (e.g., [61] employs a student t-test for each column pair). Moreover, it remains uncertain whether these properties can be sustained

[1] There do not exist predicates in all queries in TPC-H [19] and TPC-DS [18].

as data evolves. Additionally, imprecise data properties may have limited utility in query optimization (e.g., a soft functional dependency, which does not retain set multiplicity, cannot ensure the accuracy of specific plan transformations involving group-bys and joins).

2.4 Systems for Query-Time Enrichment

Overcoming the challenge/difficulty of on-the-fly enrichment/cleaning of data as it arrives when data rates are very high have previously been studied in systems including **TippersDB** [56] that supports sensor-based applications integrating sensor data processing with online data analytics, and **EnrichDB** [48, 47] that supports query-time data enrichment.

EnrichDB is a data management system that seamlessly integrates *data enrichment* during online data analytics. Enrichment often consists of complex compiled code, declarative queries, and/or expensive machine learning/signal processing code. EnrichDB is motivated by emerging application domains (such as sensor-driven smart spaces, IoT applications, and social media analysis) that require incoming data to be appropriately enriched prior to being used. Many industrial as well as research systems have started to explore effective ways to execute data enrichment in order to support online data processing, but much of this effort has focused on optimizing enrichment at the time of data ingestion. EnrichDB's design represents a significant departure from the above in that it explores 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. EnrichDB is based on the premise that enriching data in its entirety at ingestion can be wasteful (if applications do not use require all of the data to be enriched to the highest level possible), may result in unacceptable latencies (if the speed of data arrival is higher than can be enriched), or not be feasible (if enrichment functions are learned and incorporated into the system at a later time after ingestion during analysis). By pushing enrichment to query time, EnrichDB scales data

processing to environments where data rates prevent enrichment on data ingestion. While the system scales to fast and large datasets, it, nonetheless, comes at the cost of increased latency of queries which now need to enrich data during query processing. EnrichDB addresses the latency challenge by supporting progressive query processing. In EnrichDB, cheap and fast (though may not be very accurate) enrichment functions are used to enrich data which, in turn, is used to generate initial results for queries. The system progressively executes more expensive (and more accurate) functions to enable progressive refinement of query results. As a result, users/analysts can have access to approximate query answers quickly which can then be refined over time. EnrichDB uses a clever integration of incremental view maintenance (IVM) technology [103] to support the progressive generation of results. The existing implementation of EnrichDB is based on a carefully designed middleware that coordinates the execution of queries over the underlying database engine. We note that the goals of EnrichDB and the research performed in this thesis are very related. Indeed, our approach to lazy imputation of missing value during query processing explored in ZIP can be incorporated into EnrichDB to further improve on the latency challenge it addresses. Likewise, predicate learning in PLAQUE to accelerate queries could also be incorporated into EnrichDB to improve its performance. One can view our thesis contributions as exploring optimizations/techniques that are complementary to the progressive computation approaches currently implemented in EnrichDB.

TippersDB [56] is another system developed in our research group related to our work on query-time imputation. TippersDB has been designed to support data virtualization for smart space applications. In particular, TippersDB offers several unique features, semantic abstraction, transparent translation, query-driven translation, and progressive query processing.

Semantic abstraction. TippersDB supports a novel two-tier data model that separates the sensor data from the higher-level semantic data. TippersDB models the physical world/-

domain as physical entities and relationships. TippersDB uses an Entity-Relational Model suitably extended to support the dynamic nature of an evolving smart space for this purpose. In particular, attributes of entities (and relationships between them) may be static or dynamic that may change over time. For instance, a person's name may be static, but his/her location may change with time. Likewise, relationships between entities may be dynamic. E.g., if the system captures the fact of persons entering into rooms, then based on the movement of a person, a new relationship between a person and a room may dynamically emerge. In addition to representing data at the semantic level, TippersDB represents data at the sensor level in the form of data streams. It provides mechanisms for the specification of functions to translate data at the sensor level into higher-level semantic abstraction. Such a layered data model decouples application logic from sensors and alleviates the burden of dealing with sensor heterogeneity from application programming which greatly reduces the complexity of developing smart-space applications.

Transparent translation. TippersDB optimizes the translation of sensor data to generate semantic/application-level information. TippersDB associates observing functions with the dynamic properties, which observe the "value" of the attribute/relationships through sensors. Also, TippersDB maintains a representation of sensors and their coverage (i.e., what entities in the physical world they can observe) as a function of time.

Query Driven Translation. Sensor data translation can be done at ingestion or during query execution. In IoT-based systems, sensors continuously generate data, causing the data arrival rates to be very high. Processing sensor data at ingestion using streaming systems [41] leads to significant overhead. Therefore, complete sensor data translation at ingestion is not viable. The alternate strategy of translating sensor data at query time is more suitable. Not only does it avoid the large ingestion time delay, but it also reduces redundant translation of sensor data if the applications end up querying (a small) portion of the data. This query-time translation is also consistent with the modern data lake view architecture where we store and

process only data that is needed. TippersDB uses such an architecture and performs query-driven translation by adding translation as an operator inside the query plan, co-optimizing translation, and query processing.

Dynamic translation of sensor data to higher-level observations is akin to data enrichment and TippersDB exploits and expands upon the EnrichDB codebase to support progressive query processing. In particular, TippersDB develops progressive query processing techniques that progressively translate data and provide early results to the user. To progressively return results, TippersDB returns results of lower quality or at a coarse level early and keeps returning results of better quality and at a finer-grained level as more and more time passes. The design of TippersDB is complicated by an additional challenge beyond the way progressiveness is implemented in EnrichDB. In the EnrichDB setting, exact functions are readily available to enrich and interpret raw data. However, in the context of TippersDB's automated translation, the exact sensor data needed for generating higher-level data may not be known immediately. For instance, if the query is exploring the location of a person P, while the system may know that the location can be determined using techniques such as LOCATER (see Chapter 5) on WiFi access point logs, it is not immediately obvious that the logs of which access point may have information to locate a person. In the worst situation, one may have to process data from all sensors to be able to locate a person P. TippersDB contains several techniques to improve upon such a translation task by learning semantic filters from data that can be used to prioritize (or filter away) data from sensors to reduce the complexity of translation.

TippersDB, like EnrichDB shares significant objectives with the research described in this thesis. Indeed, one of the key chapters of the thesis on LOCATER, a WiFi-based indoor localization technology has been incorporated into TippersDB and the resulting system has been transitioned to US Navy as well as deployed in several sites as will be discussed in Chapter 5.

Chapter 3

ZIP: Lazy Imputation during Query Processing

3.1 Introduction

A large number of real-world datasets contain missing values. Reasons include human/machine errors in data entry, unmatched columns in data integration [71], etc. Failure to clean the missing data may result in the poor quality of answers to queries that may, in turn, negatively influence tasks such as machine learning [76], data analytics, summarization [53, 59], etc. built on top of data.

Missing value imputation has been extensively studied in the literature, especially from the perspective of ensuring accuracy [78, 116, 102, 33]. Traditionally data cleaning (including missing value imputation) is performed during the Extract, Transform, and Load (ETL) data processing pipeline as an offline data preparation process that cleans all the data prior to loading it into the warehouse to make it available for analysis. Such a data preparation step is often costly if the data is large and if the cost per imputation is high. In situations where

a database continuously ingests new data containing missing values, performing imputation at the time of ingestion becomes impractical, especially when data arrives at rates such that imputing missing values cannot be done cost-effectively. For example, LOCATER (as described in Chapter 1) takes roughly 400ms to impute one missing location on WiFi connectivity data. With 1000s of WiFi access points, about 30,000+ individuals connected to the network and tens of thousands of WiFi connectivity events per second, it would take one hour of processing per one second of data during peak load. Carrying out such operations during ingestion is clearly infeasible. Similar observations of impossibility to perform enrichment/imputations on data as it arrives have also been made in prior works such as [48, 39] in different contexts - e.g., when enriching tweets with sentiment tags that may require executing complex sentiment analysis[1]. As a result, recent works have explored alternate data cleaning pipelines that clean data lazily at the time of query processing when the need for clean data arises [25, 24, 49, 34].

As discussed earlier in Chapter 1, query time cleaning offers several benefits. It significantly reduces wasted effort and computational resources by cleaning only parts of the data actually needed in analysis instead of indiscriminately cleaning the entire dataset. This is especially important when cleaning is expensive and/or datasets are very large, making cleaning of the data fully infeasible. Predicting the dataset analysts might use apriori so as to clean such data as a pre-analysis step is often not feasible (e.g., when a common analysis operation consists of ad-hoc queries on the data) [28]. In such situations, the only recourse is to support data cleaning with query processing.

Query-time data cleaning opens new challenges, the prominent of which is to minimize cleaning performed during query processing to reduce latency. This chapter develops *ZIP*, a la*Z*y *I*mputation query *P*rocessing approach that exploits query semantics to reduce the cleaning overhead. When processing records with missing values, ZIP may delay imputations

[1] [39] shows that sentiment inference on 1.6 million tweets in Sentiment140 dataset [50] takes hours to execute.

sid	faceID	Time	location
1	20	12pm	2206
2	41	2pm	NULL ($N_1 = 3001$)
3	20	1pm	NULL ($N_2 = 2206$)
4	35	3pm	NULL ($N_3 = 2099$)
5	NULL ($N_4 = 26$)	1pm	3119
6	NULL ($N_5 = 55$)	2pm	2214

Table 3.1: Camera-Snapshots (C)

Name	Type	faceID
Mike	faculty	NULL (N_6 =20)
Robert	graduate	65
John	faculty	NULL (N_7 =55)

Table 3.2: User (U)

until later - such a lazy approach to imputing can be beneficial if the record with the missing value get eliminated in the query tree, thus, avoiding imputations unnecessary for answering the query. Delaying imputations, comes at an increase in processing cost, if imputation could not be avoided. ZIP, given a query plan for an SQL query, develops an execution strategy that minimizes the overall (joint) cost of imputing missing data and executing the query. We illustrate the key intuition behind ZIP through an example below.

3.1.1 A Case for Lazy Imputation

Consider a real camera-based localization application in Donald Bren Hall building, UCI, which is instrumented with the camera used to locate people. A tuple in Camera-Snapshots (Table 3.1) stores the location (i.e., room) of a person (i.e., faceID determined using face recognition) at a given time (i.e., the timestamp). The faceID of a person could be determined by matching camera data with picture(s) of a person stored in the database or through a model trained using such pictures. User (Table 3.2) and Space (Table 3.3) tables store the metadata about registered users and space. There are 10 missing values (shown as NULL) in the 3 tables, and we also display the corresponding imputed values (shown as blue color in

Room	Building
2214	NULL (N_8 =DBH)
2206	DBH
2011	DBH
3119	NULL (N_9 =ICS)
2065	NULL (N_{10} =DBH)

Table 3.3: Space (S)

the bracket). Let us consider a simple query, find all snapshots (sid) for graduate students in room 2099. Such a query joins Camera-Snapshots with the User table, after selecting tuples matching query predicates on each table as shown in Figure 3.1.

Let us consider various possible query-time imputation strategies in different query plans. Figure 3.1-a) is the plan where all selections are pushed down. In such a plan, all the missing values under *location* column (i.e., N_1, N_2, N_3) must be imputed since the selection operator $\sigma_{location='2099'}$ requires missing values to be imputed prior to execution. After imputations, only one tuple with sid 4 satisfies the selection condition, and will thus be passed onto the join operator. Since the faceID of this tuple (i.e., 35) does not match any faceID in Table 3.2, the query execution will terminate.

One may be tempted to consider the additional imputation overhead (i.e., N_1, N_2, N_3) of Plan 1 to be a result of pushing selections to the leaf level. This raises the issue whether the savings resulting from modifying the operators could be achieved simply by making the optimizer aware of the expensive nature of imputations which may, then, consider imputation required by the selection operator $\sigma_{location='2099'}$ as expensive (as in [58]) resulting in the operator to be pulled above the join condition, such as Plan 2 in Figure 3.1-b). Even such a plan would still require two imputations for N_4, N_5. Furthermore, such a plan would incur significant execution overhead for tuples for which attribute values are not missing, since the input size to join from table C will be the cardinality of C table without any filtering. Thus, the benefits that can be achieved by modifying the operator implementation cannot simply be mimicked by changing the optimizer.

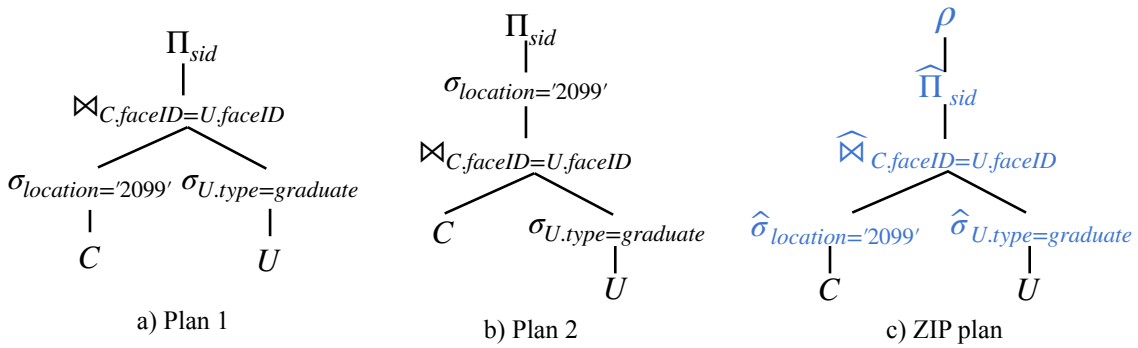


Figure 3.1: Imputation in Different Query Plans.

Now let us now consider the strategy illustrated in Figure 3.1-c) wherein each operator o is replaced by a corresponding imputation-aware operator $\widehat{o}$. A modified operator $\widehat{o}$ behaves exactly the same as the original operator o for tuples that do not contain missing values. For instance, for tuples with sid 1, 5, and 6 for which location attribute is not missing, $\widehat{\sigma}_{location='2099'}$ evaluates the predicate right away (and drops the tuples since they do not match the predicate). For tuples with missing values (i.e., tuples with sid 2, 3, and 4), the modified operator $\widehat{\sigma}_{location='2099'}$ may decide to either impute the missing value and compute the predicate, or delay the imputation for the downstream operator to perform. Delaying imputation can prevent unnecessary imputations, if such a tuple (whose imputations are delayed) does not satisfy predicates associated with the downstream operators. In our example, if $\widehat{\sigma}_{location='2099'}$ forwards the tuples with sid 2, 3, 4 in Table 3.1 to the downstream join without imputing N_1, N_2, N_3, it would have resulted in the savings of all three imputations since the tuples do not meet the join condition (the only graduate student in the User table has a faceID of 65 which does not match the faceID of tuples with sid 2,3, and 4)!

Such a lazy strategy would possibly minimize the imputation costs without sacrificing the quality of the result. In the example above, saving two or three imputations may appear to be of little benefit compared to the additional complexities that could arise in maintaining state and modifying operators, in practice, when tables are large and imputation costs are relatively expensive such savings quickly add up. For instance, even for the simple query

discussed above if Camera-Snapshots contains millions of rows, imputing all the missing locations would be very expensive.

3.1.2 Challenges in Supporting Laziness

ZIP uses such a lazy strategy that enables operators to defer imputations to later in the hope that the need for imputation may become unnecessary if the tuple with the missing value gets eliminated by downstream operators thereby reducing imputation overhead. A strategy that defers imputation to downstream operators, such as ZIP, has to be designed carefully. If a tuple whose imputation was delayed is not eliminated (e.g., due to conditions associated with downstream operators) rises to the top, it will finally be imputed by a newly added operator in ZIP (ρ) which will result in the evaluation of all the query predicates associated with the missing value and possible generation of tuples that emanate from the tuple containing the imputed value. Of course, the modified operators that had been executed before must have saved enough state to perform delayed predicate evaluation, and to generate all the answers the tuple would have been generated had it been imputed eagerly. In the remainder of the chapter, we develop ZIP that supports modified operators with the ability to defer imputations to later to benefit from the pruning power of downstream operators to reduce imputations during query processing. ZIP always updates imputed values in the database, avoiding repeated imputations for the same missing value, which offers an advantage for a query workload that frequently visits the same subset of data.

3.1.3 Contributions

The chapter introduces a ZIP framework to answer SQL queries over data that may contain missing values. The primary contributions include (a) a simple modification to the logic of relational operators that empowers operators to choose to either impute or delay missing

values, (b) a decision function to enable operators to determine whether the imputation should be performed right away or delayed based on a cost-based analysis of tradeoffs between the two choices, (c) efficient mechanisms to maintain the state of the execution and the modified query processing logic to continue execution over imputed values so as to generate the right query results. ZIP designs modified operator logic for a wide range of operators, such as selection, join, projection, aggregate-group by, union, and set minus and can handle a large class of queries of significant complexity including nested queries. Extensive experiments on both real and synthetic data sets demonstrate orders-of-magnitude improvements in query processing when ZIP-based deferred imputations are used.

In the rest of the chapter, in Section 3.2, we overview of the approach. Section 3.3 to 3.6 describe ZIP algorithm. Section 5.6 evaluates ZIP, and Section 5.7 concludes this chapter.

3.2 ZIP Overview

This section provides an overview of how ZIP achieves delayed imputation by appropriately modifying the relational operators. We will use the query shown in Figure 3.2 to illustrate ZIP. We shift to this query instead of a simpler query we used in the previous section to contrast ZIP from ImputeDB since the simpler query will no longer suffice to illustrate all the cases ZIP needs to handle to ensure correct execution. The Figure 3.3-a) shows the query tree generated by a third-party optimizer, (e.g., a standard commercial system such as PostgreSQL or a specialized plan generated by ImputeDB). ZIP modifies such a plan by replacing operators by their modified versions and by adding a new operator ρ at the top of the tree as shown in Figure 3.3-b) that imputes missing values whose imputation has been delayed by previous operators. ZIP has been implemented in the context of pipeline query execution using an iterator interface. The execution starts from the root of the query tree by calling a root.getNext() that retrieves tuples from the child nodes that satisfy the associated

```
SELECT C.sid FROM C, U, S
WHERE C.faceID = U.faceID AND
C.location = S.room AND S.building = 'DBH' AND
C.location in {2065, 2011, 2082, 2206}
```

Figure 3.2: Query

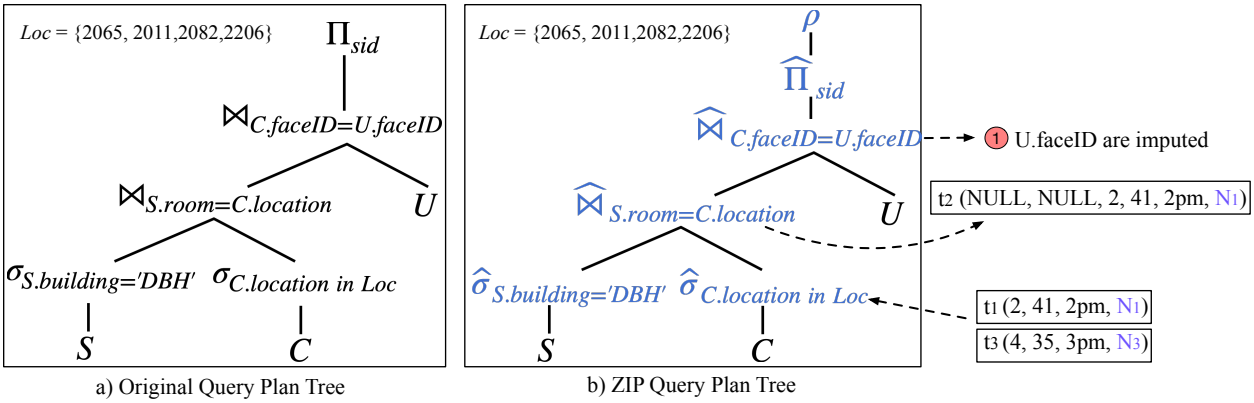


Figure 3.3: ZIP query plan.

conditions. Child nodes, in turn, recursively call the getNext() operator on their children. ZIP modifies the relational operators to process incoming tuples that contain missing values differently. Other tuples (that do not contain missing values) are processed exactly as they would be by the original operator. In particular, ZIP does not change the underlying operator implementation - for instance, the relational operator can continue to use hash/sort/nested loop/index-based operator implementations supported in the underlying database without change. ZIP simply routes tuples containing missing values through a sequence of steps (i.e., filter, verify, decision function, and generate) that will be discussed soon. Thus, besides code to implement such steps, ZIP only changes the routing logic of operators which requires a very small amount of new code (approximately 500+ lines) while preserving the existing code of the database.

Missing Value Representation: Before we discuss how modified operators are implemented in ZIP, we first specify how ZIP represents missing attribute values. In ZIP missing values are represented using NULLs. However, to differentiate between a value of an attribute being NULL or missing, the relational schema is extended with an additional attribute that

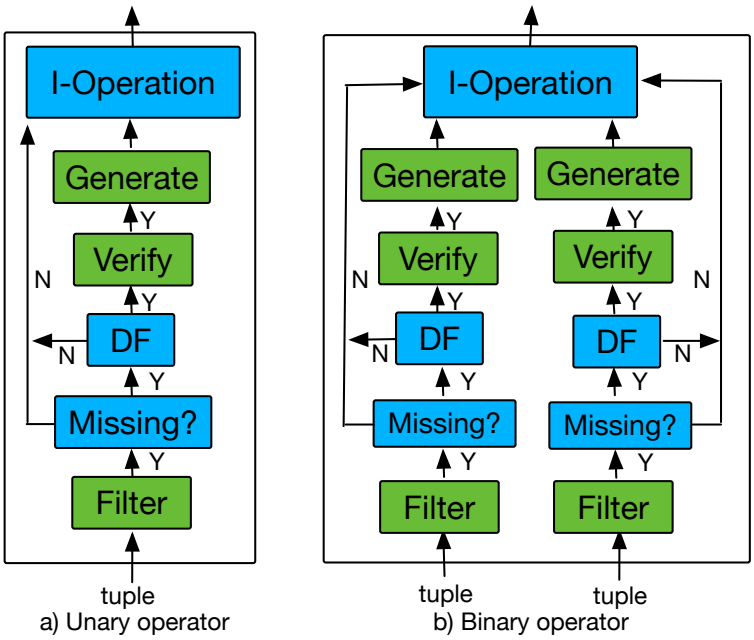


Figure 3.4: Modified Operators.

contains a bit per attribute of the relation. If the value of attribute a in a tuple t is missing, its value is NULL and its corresponding bit is set to 1. If attribute a is NULL but its bit is 0, then a is not missing, instead, it is NULL.

Routing logic of modified operator. Figure 3.4 shows the modified logic of the unary and binary operators. The tuple incoming to the operator first passes through a filter step (the purpose of which will become clear momentarily). It is then checked to determine if the attribute value (on which the operator is defined) is missing (by checking for the appropriate bit in the additional attribute stored in the tuple). If the value is not missing, the tuple is directed to the I-operation, i.e., *Imputation-aware operation*, which, for such tuples, implements the exact same logic as the original operator. If the value is missing, the tuple is diverted through a decision function (DF) which may decide to either impute or delay the imputation of the missing value. If imputation is delayed, the tuple again is routed to the I-operation which, in turn, forwards the tuple to downstream operators with missing values preserved without checking the associated predicate, if any, with the operator. For unary operators (e.g., selection), the tuple is forwarded as is, and for binary operators (e.g., join)

the tuples are forwarded to the next operator in the pipeline in a way similar to the way they are in an outer-join as illustrated in the example below. For projection operators, ZIP preserves all attributes in a tuple that contain missing values and that may be imputed later. Tuples that the decision function decides to impute are first routed to the verify and the generate steps. The goal of the verify step is to determine if the imputed value satisfies all the predicates associated with the previous operators (and hence the tuple would have made it to the current operator). If a tuple t passes the verify step, the generate step is invoked on t. This step generates all additional tuples that would have resulted by executing the logic of upstream operators had t been imputed at the very beginning of query execution. The generated tuples, now with missing values imputed, are passed through the operator logic and processed just the same way the the original unmodified operator would have processed tuples. The imputed value of an attribute a in a tuple t may also be present in other tuples in multi-join queries. When a is imputed, all the tuples with the imputed values will be forwarded to generate step in order to generate all the results as we discussed in Section 3.5.

To see how data flows through modified operators in ZIP, consider a tuple $t_1 = (2,41,2\text{pm},N_1)$ in table C with a missing value in the location field for a query illustrated in Figure 3.3-b. Further, let us assume that the selection operator delays imputation. Thus, t_1 is passed to the join operator as it is. The modified join operator, which is also defined on the location field, will decide whether to impute the missing location field or to delay its imputation further. If the join decides to delay, it preserves the missing value in a location in a way similar to the way outer joins preserve tuples. In particular, it generates a tuple $t_2 = (\text{NULL},\text{NULL},2,41,2\text{pm},N_1)$ where N_1 is the preserved missing value and the NULLs represent that the values of those fields are NULL. Here we denote N_i by missing values and the associated value of NULL for null values.

We next explain the roles verify, filter, generate and decision function (DF) play in the implementation of the modified operator.

Verify: The verify operator is invoked whenever a missing value is imputed in the current operator to check if, had it been imputed earlier, it would have caused the tuple to be eliminated by a prior upstream operator. In such a case, the tuple can be dropped since such a tuple would not have passed the logic of a prior operator and would, thus, have not reached the current operator. To see such a case, consider the missing value N_1, and assume the decision function decides to delay its imputation in $\widehat{\sigma}_{C.location\,in\,Loc}$. If this value is imputed later during query processing (e.g., in operator ρ), (② in Figure 3.3), the imputed value must satisfy every predicate that applies to the imputed value in the upstream operators prior to ρ.

Filter: Filter operator in ZIP works in a manner dual to verify - while verify is used to check if a tuple whose missing value imputation was delayed in a prior (upstream) operator can be pruned after imputation since it would have failed predicates in prior operators, the filter test is used to prune tuples based on predicates associated with future downstream operators that the tuple will not satisfy. Filter test associated with an operator o can, thus, result in early pruning of tuples saving imputations. To see how the filter works, consider Figure 3.3. Assume that the decision function associated with the join operator $\widehat{\bowtie}_{C.faceID=U.faceID}$ decides to impute all missing values of $U.faceID$. Further, assume that the $\widehat{\bowtie}_{C.faceID=U.faceID}$ is implemented as a hash join and U is the build table used in the hash join. In such a case, all possible values $U.faceID$ (i.e., $\{20, 65, 55\}$) could take would have been determined early in the pipeline query processing as soon as the build phase of the hash join is complete. Now consider tuples in table C (e.g., t_3) passing through $\widehat{\sigma}_{C.location\,in\,Loc}$. Such tuples will be pruned if their faceID is not in $\{20, 65, 55\}$. In this example, t_3 with faceID 35 will not join any tuples in U and will, thus, not be part of the answer.

Generate: In operator o, the generate step is responsible to generate possible tuples that satisfy all the previous upstream predicates of o. For instance, in the query tree in Figure 3.3, if the imputation of N_1 is delayed by the join operator until later (say, until ρ executes),

necessary joining tuples that could have resulted from t_1 will need to be generated. To this end, ZIP needs to maintain the states of all tuples that flow through the join operator and uses the state to support a carefully designed mechanism that ensures correct query answer even when imputations are delayed. When and how the generate step executes will be described in Section 3.5.

Decision function: ZIP creates a decision function associated with each operator to determine whether to impute the missing values or delay imputation. Intuitively, it is tempting to delay imputing in operator o if imputations are expensive and the downstream operators of o are selective. If the tuple is eliminated by a downstream operator, the imputation required to execute o would be saved. On the other hand, if ZIP decides to impute missing values right away, the imputed tuple will have a chance to be eliminated by the current operator saving execution cost. The decision function is a cost-based solution to estimate the expected execution cost of imputing right away versus delaying the imputation, and chooses the option with lower cost. We discuss the decision function in Section 3.6.

ρ **Operator**: ZIP adds a new operator ρ at the top of the tree which imputes all missing values in the attributes associated with query predicates that have not been imputed so far. The structure of the ρ operator is the same as that of unary operator with the difference that for ρ the decision function is always set to impute[2]. Like other unary operators, once a tuple is imputed in the ρ operator, it goes through the verify step, and if passing verification, goes through the generate step. Since ρ is the final operator, the way ρ executes the generate differs slightly compared with other operators as will be discussed in Section 3.5. We note that ρ will impute any missing values in the projected attributes if any and removes all attributes in the imputed tuples that were not part of the projection in the query. [3]

[2]We could alternatively, also consider drop operator, similar in spirit to ImputeDB, which will allow our technique to explore the cost-quality tradeoff as well.

[3]When a tuple with multiple missing values reaches ρ, ZIP simply prefers imputing attributes in selection conditions. An alternative strategy can be first imputing the missing value with the lowest estimated imputation cost.

Roadmap: In the remainder of this chapter, we first describe the implementation of the verify and filter operators in Section 3.3. The imputation-aware operations (i.e., I-operation) and answer generation (i.e., the generate step) are described in Section 3.4 and Section 3.5 respectively. Finally, we show the design of the decision function (i.e., the DF step) in Section 3.6. We restrict the discussion to the modified versions of the select, project, and join operators and illustrate query processing in ZIP through the SPJ queries.

3.3 Verify and Filter Steps

Implementation of Verify and Filter steps of an operator o requires ZIP to maintain several data structures that we discuss next.

3.3.1 Data Structures

Verify Set. Verify set for operator o consists of all the predicates over the attribute A_o which are associated with all the *upstream* operators (i.e., those that appear below o in the query tree), where A_o are the attributes associated with the predicate in o. Figure 3.5 shows the verify sets for all operators in the query tree in Figure 3.3.

Filter Set. A filter set for an operator o consists of predicates defined over attributes associated with the tuples that are input to o. These predicates correspond to conditions associated with operators that are *downstream* to o (i.e., are higher up in the query tree) and are defined over attributes other than A_o on which o is defined. As an example, consider selection operator $o = \sigma_{S.Building='DBH'}$ in Figure 3.3-b) where $A_o = \{S.Building\}$. We add the predicate $\{S.room = C.location\}$ from the downstream join operator to the filter set since it is defined on the attribute $S.room$ which is different from the attribute $S.Building$ on which the selection operator is defined. The filter set for $o = \sigma_{S.Building='DBH'}$ can be

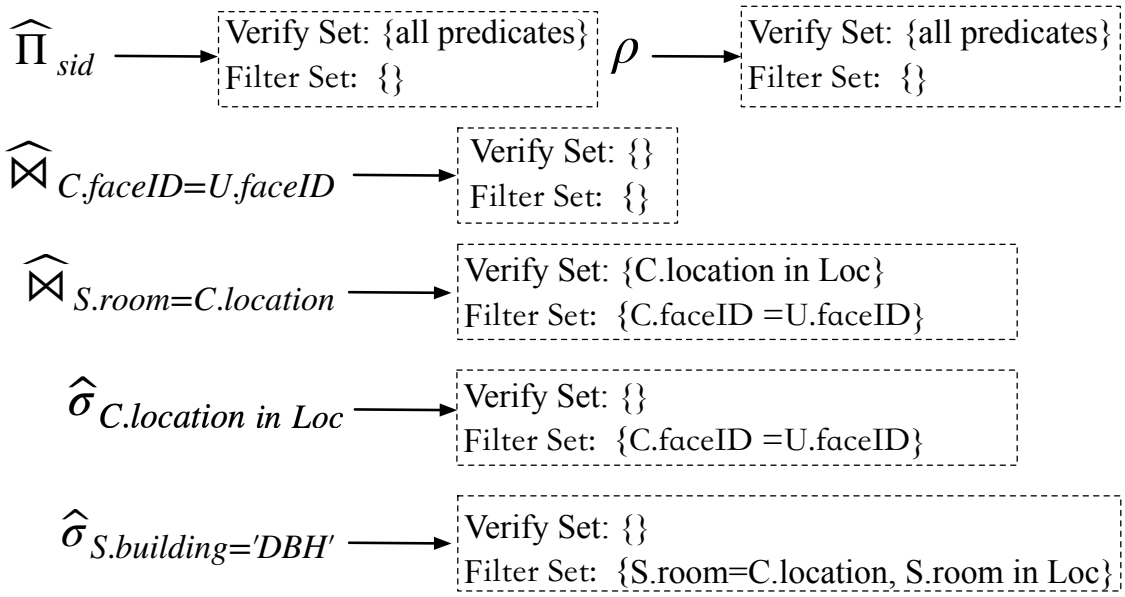


Figure 3.5: Verify Set and Filter Set.

expanded further by additional predicates which can be inferred from the current filter set. In the example, the predicate $\{C.location\ in\ Loc\}$ coupled the filter $\{S.room = C.location\}$ enable filter set of $o = \sigma_{S.Building='DBH'}$ to be expanded to $\{S.room = C.location, S.Room\ in\ Loc\}$. The conditions in the filter set are used in ZIP to eliminate tuples earlier that will eventually be eliminated by downstream operators, thereby saving unnecessary imputations.

Bloom Filters. ZIP constructs a bloom filter [7] for each join attribute in the equi-join operator in a query Q. Such a bloom filter, $BF(a)$ for the attribute a is constructed incrementally as the tuples are processed by the modified join operator. That is, when the modified operator processes a (non-missing) attribute value, it stores the value in the bloom filter $BF(a)$. Likewise, whenever a missing value in a tuple for a join attribute is imputed (either as part of the join or a further downstream operator) and passes the verification test, it is added to the corresponding bloom filter. The bloom filters help prune/filter tuples early in upstream operators based on downstream join conditions. For instance, in the example above, using the bloom filter, the operator $\sigma_{S.Building='DBH'}$ could use the join condition in its filter set (e.g., $\{S.room = C.location\}$ to check if the *room* associated with the current

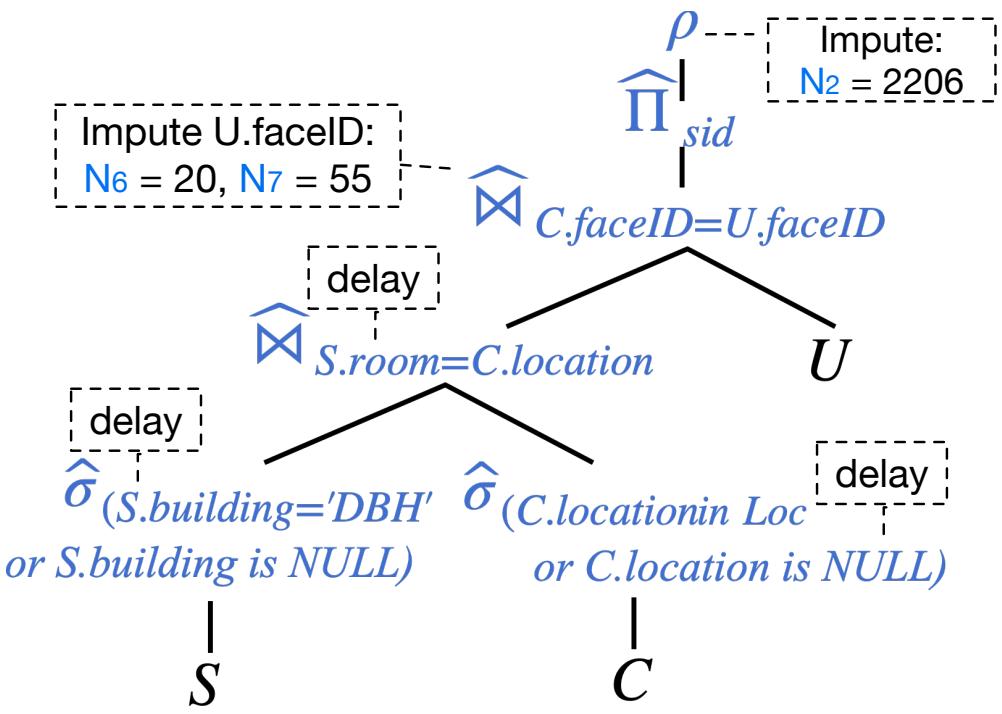


Figure 3.6: ZIP Query Plan Tree.

tuple matches *any* $C.location$ using the bloom filter $BF(C.location)$. In addition to helping implement the filter step, ZIP uses bloom filter in the modified join will also be used to support the modified join operator (as will be discussed in Section 3.4).

Bloom Filter Completeness. A bloom filter $BF(a)$ for a join attribute a in a query Q is said to be *complete* with respect to Q if $BF(a)$ contains all values of a that could result in tuples in the answer set of Q. Note that the completeness condition does not require all values of a to be in $BF(a)$. Tuples that are filtered away by the selection/join operators may not be in $BF(a)$ for it to be considered complete. More formally, let Q be a query over relations $R_1, R_2, \ldots, R_n$. W.L.O.G, let a be an attribute in R_1 that participates in a join predicate in Q. The bloom filter $BF(a)$ is said to be complete w.r.t. Q if for all tuples $t_1 \in R_1$, such that there exist tuples $t_2, t_3, \ldots, t_n$ in $R_2, R_3, \ldots, R_n$ that along with t_1 produce a tuple in the answer set (i.e., tuples $t_1, t_2, \ldots t_n$ together satisfy all the predicates associated with Q), $BF(a)$ contains $t.a$.

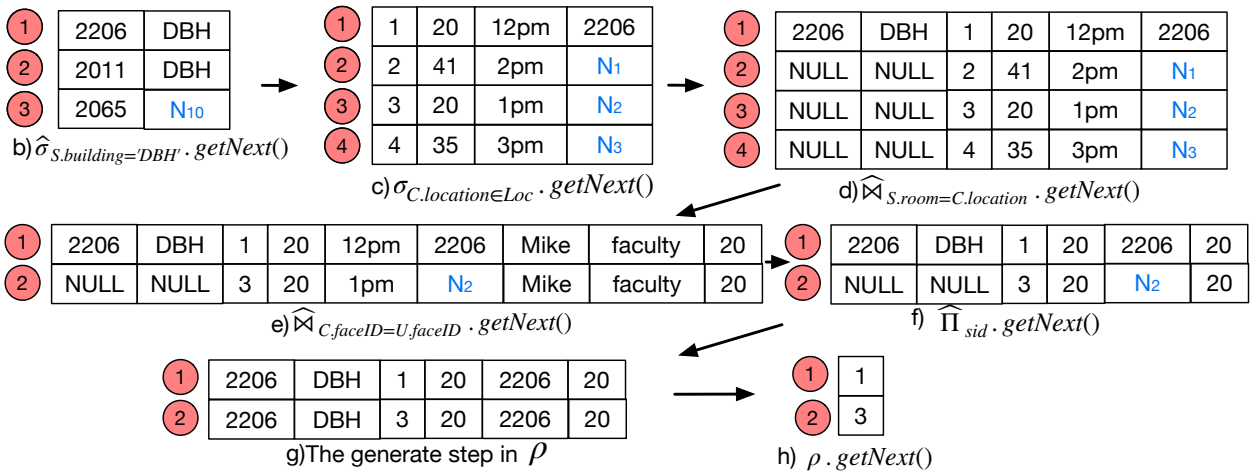


Figure 3.7: Pipeline ZIP.

We denote the event during query processing that causes the bloom filter $BF(a)$ to become complete as $BFC(a)$ and we refer to it as a completeness event for $BF(a)$. ZIP will need to test when the bloom filter completeness for an attribute a in a join is reached to correctly generate query answers. For $BFC(a)$ to be reached, two conditions should be held. First, all the tuples with missing values in a should have been imputed or eliminated and there should be no missing values. To test such a condition, for a query Q, ZIP maintains a missing value counter $MC(a)$ that records the number of missing values for each attribute a in Q. Such an array is initialized using the metadata or statistics maintained by in database. Whenever a missing value in attribute a is imputed or dropped, (e.g., as a result of a filter operator), ZIP reduces the count of $MC(a)$ appropriately.

Second, reaching $BFC(a)$ further depends upon the specific join algorithm used to compute a join. Consider a join $R^L.a \bowtie R^R.b$, where R^L and R^R are the left and right relations respectively, and a and b are join attributes. When there is no ambiguity, we will refer to R^L and R^R simply as L and R. If this join ($L.a \bowtie R.b$) is implemented using nested loop, for inner relation R, bloom filter $BF(R.b)$ contains all values in $R.b$ (i.e., $BFC(R.b)$ is reached) when there are no more missing values of $R.b$ and the first pass of relation R has been processed. For outer relation L, such a condition becomes true only when all tuples

have been processed through the join operator. For hash joins, similar to nested loop, the bloom filter contains all values as soon as the hash table based on build relation has been built and for outer relation such a condition is reached when all tuples have been processed. For sort merge, or multi-pass hash join, the bloom filters for both relations L and R contain all values as soon as the sort or hash table build is finished. ZIP maintains for each attribute a in a join a boolean, entitled $JC(a)$ that becomes true when all the values in attribute a have been processed. For instance, if a is in the build relation of a join, $JC(a)$ becomes true when the build phase is fully executed. ZIP modifies the scan operator to detect and set JC conditions when all tuples in a relation have been consumed. [4]

Thus to determine $BFC(a)$ ZIP simply needs to check when both $MC(a) = 0$ and $JC(a) = true$ has been reached.

3.3.2 Verify and Filter Implementation

To implement verify and filter operation, for incoming tuples, ZIP only needs to check conditions stored in verify and filter set to determine if the tuples satisfy them or not. If the conditions are selections, tuples can be evaluated right away. For join conditions, we check if the bloom filters of the join attributes are complete or not. If they are complete in pipeline query processing, we use the bloom filter to test if the tuple satisfies this join condition. For instance, consider operator $\widehat{\sigma}_{S.building='DBH'}$ whose filter set contains a join condition $S.room = C.location$ in Figure 3.5. For tuple t received by $\widehat{\sigma}_{S.building='DBH'}$, if $S.room$ is not missing, and the bloom filter $BF(C.location)$ is complete, ZIP uses $BF(C.location)$ to check if $S.room$ has any matched values in $BF(C.location)$. If $BF(C.location)$ returns false, we drop tuple t. This check operation is safe because the bloom filter does not have a false negative. Otherwise, we do nothing and let t pass.

[4] In the case of index scan, JC becomes true when all the tuples that satisfy scan condition are consumed.

3.4 Impute-Aware Operators

In this section we describe the impute-aware operation, i.e., I-operation in Figure 3.4, for selection, projection, join and ρ operators.

Unary Operators: I-operation for the select, project, and ρ operators are straightforward. For selection, I-operation simply evaluates the selection predicate if the corresponding attribute value is not missing. Otherwise, it forwards the tuple to the next operator. I-operation for the projection operator, besides forwarding attributes in the projection, also preserves values associated with attributes in query predicates for tuples that have missing values in those attributes. The I-operation for the ρ operator at the top of the tree returns the tuples after projecting to the attributes in the final results. We illustrate the execution using an example in Figure 3.6 and Figure 3.7. Figure 3.6 is the ZIP query plan for the query in Figure 3.2, and the decisions taken by the decision functions in each operator are marked. In Figure 3.7-b) to g), the numbered red circle represents the tuples returned by $getNext()$ for each operator. Assume ZIP decides to delay imputations in two selection operators $\widehat{\sigma}_{S.Building='DBH'}$ and $\widehat{\sigma}_{C.location\,in\,Loc}$, and their getNext() tuples are shown in Figure 3.7-b) and Figure 3.7-c), respectively. The projection operator $\widehat{\Pi}_{sid}$ returns tuples in Figure 3.7-f), it not only projected *sid*, but also all the attributes in query predicates.

Join Operator: The I-operation for the join operator is more complex. Consider a modified join operation $\widehat{\bowtie}_{L.a=R.b}$, and a tuple t that reaches I-operation of the join in either relation L or R. Note that such a tuple t has passed through the filter, decision function, verify, and generate steps in Figure 3.4. W.L.O.G, let t belongs to relation L. First, if the attribute value a in t, $t.a$, is not missing, ZIP adds $t.a$ into the bloom filter $BF(a)$ and simply uses the original join implementation to join t with tuples in R whose b values are not missing. For instance, if the query plan specified a hash (or an index, or nested-loop) join, ZIP simply continues to use the original code for such joins that were part of the database prior to

modifying the operators to be impute-aware. If $t.a$ is missing, however, ZIP bypasses the original join code and instead generates a new output tuple that contains all the attribute values of t including the missing value, and NULL values for all the attributes of the other relation. ZIP preserves the missing value of $t.a$ for later query processing by creating a tuple similar to the tuple created by the left-outer join. Likewise, if $t \in R$, then ZIP creates a corresponding tuple by concatenating NULLs for the attributes in L.

In addition, for one of the two inputs (i.e., L or R) for the join operator $o = \widehat{\bowtie}_{L.a=R.b}$, ZIP maintains a list of tuple identifiers of the base relations from which the missing value of $L.a$ or $R.b$ originated. These lists are denoted by $\mathcal{L}(o, a)$ and $\mathcal{L}(o, b)$ respectively. ZIP only populates one of $\mathcal{L}(o, a)$ and $\mathcal{L}(o, b)$ leaving the other empty. ZIP chooses the list that is expected to be smaller (e.g., with a lower number of missing values in the corresponding base relation) to reduce overhead. Thus, if either of the two inputs do not contain missing values, ZIP will choose that attribute, and hence both lists would be empty. These lists, as we will see in Section 3.5, are required to ensure result tuples are generated only once with no duplicates.

To illustrate the modified join operator, consider the join operator $\widehat{\bowtie}_{S.room=C.location}$ in Figure 3.7-d), where only *C.location* has missing values. We assume the decision function decides to delay imputation in this join operator. The tuple ① in Figure 3.7-d), is a joined tuple from tuple ① in S relation in Figure 3.7-b) and tuple ① in C relation in Figure 3.7-c). All the other tuples, i.e., tuples ②-④ in Figure 3.7-e), are the right outer join results of S and C where the missing values N_1, N_2, and N_3 are preserved with NULLs in columns in S side.

Note that a missing value may appear in multiple tuples if one tuple t matches multiple tuples in the join operation. In Figure 3.8, in the join operation $\widehat{\bowtie}_{C.faceID=U.faceID}$, N_1 appears more than once in the join result. To prevent having to impute the same missing value more than once, when a missing value is imputed, references to the value in all tuples

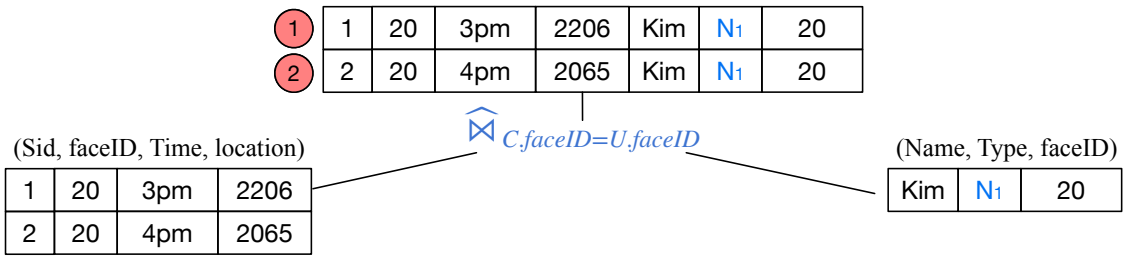


Figure 3.8: Example of Missing Value Duplication.

are replaced at the same time. For this purpose, we maintain a link in main memory from the missing values to all tuples in which they appear. When a missing value is imputed, all of these tuples with the missing attribute imputed will be passed on to the generate step to compute the corresponding results.

3.5 The Generate Step

This section describes how ZIP generates tuples when a tuple with an attribute (whose imputation had been delayed by a previous operator) is imputed as part of a downstream operator (e.g., another relational operator or the ρ operator). Let the *generate* step be invoked when a missing value in attribute a of a tuple t is imputed and passes the verify step for an operator o. The generate step reconstructs all the tuples that would be present in the output of o to which $t.a$ would have contributed, had $t.a$ been imputed earlier. The essential idea in the generate step is to replay the joins on tuple t contributed by the imputation of $t.a$, as shown in Algorithm 1.

Given operator o, let $UJ(o, a)$ be the set of predicates associated with join operators upstream of o whose associated predicate contains attribute a. $UJ(o, a)$ can be identified from the join predicates in the verify set of o. For instance, $UJ(\rho, C.location) = \{S.room = C.location\}$. Note that $UJ(\rho, C.location)$ does not contain join condition $C.faceID = U.faceID$ even though it is also an upstream join of ρ since it does not contain $C.location$. If $UJ(o, a)$

Algorithm 1: *generate* step in operator o

```
   Input: o, t, a
1  T ← {t}
2  if CHECK_REPLAY_READY_(o, a) then
3  |   new_T ← ∅
4  |   for o_j ∈ UJ(o, a) do
5  |   |   for t_i ∈ T do
6  |   |   |   new_T ← new_T ∪ REPLAY(t_i, o_j, a)
7  |   |   T ← new_T
8  return T to the I-operation of o
```

is empty, then tuple t is forwarded to the I-operation in o. (Line 8) Otherwise , for each predicate p in $UJ(o, a)$, the generate step first checks if the attribute (other than a) in p has reached its bloom filter completeness (i.e., $BFC(b)$ is true, where b is an attribute in p, and $b \neq a$) by calling CHECK_REPLAY_READY_(o, a). In such a case it generates all the tuples that would have resulted from the imputed value of a in t by replaying the joins (Line 3-7). Note that if any attribute presented in predicates in $UJ(o, a)$ (other than a) is not bloom filter complete, the join processing for the tuple containing the imputed value cannot be processed fully right now. Hence, the original generate step after imputation would simply forward the tuple to the I-operation of o which will forward it to downstream operators for future processing similar to the way I-operators pass tuples containing missing values (Line 8). If o is the ρ operator (and, thus, there is no further downstream operator for o to push the tuple whose join processing is not complete), ZIP banks such tuples whose CHECK_REPLAY_READY_(o, a) fails until the time the condition becomes true. Once the condition becomes true, the tuple is rerouted to generate all the relevant results using the replay function. For tuples not delayed by ρ, all the generated answers are returned as output by passing the tuples to the corresponding I-operation of ρ.

Note that for all tuples delayed in the ρ operator, eventually the CHECK_REPLAY_READY_(o, a) will become true, which requires that the $BFC(b)$ be true for any attribute b that is an attribute in any predicate p in $UJ(o, a)$ other than a. Reaching $BFC(b)$ requires $MC(b) = 0$ and $JC(b) = true$. The condition $JC(b)$, as discussed in Section 3.3.1, is a property of the

join algorithm used and will eventually always be met for all attributes as the scan for the base relation containing b has processed all the tuples that satisfy the predicates (if any) associated with the scan. The condition $MC(b)$ will also be eventually reached as the ρ operator continues to impute the remaining missing values.

Replay Function: We now explain the REPLAY function in Algorithm 2. Consider executing the replay function for a tuple t in operator o, and assume the join condition is $a = b$. ZIP first checks if the imputed value $t.a$ is in the bloom filter of attribute b, i.e., $BF(b)$. If a matched value is not found, then the tuple t will not join with any tuple in the current join operator o and thus an empty set is returned. (Line 2-3) Otherwise, if the bloom filter matches the imputed value of a, ZIP first retrieves all the tuples in the relation that match with t on the join attribute a using the index built on a (Line 4), and removes the tuples stored in the $\mathcal{L}(o, b)$ to prevent from generating possible duplicated join answers. Its correctness will be clear in a later discussion part in this section. ZIP then updates each such matched tuple to t and returns the results by using the merge function. (Line 7-8) [5]

We illustrate the Generate Step (including the replay and merge functions) using an example in Figure 3.7. Consider the generate step in ρ operator in Figure 3.7-g. When the input tuple to ρ (tuple ② in Figure 3.7-f), $t = \{NULL, NULL, 3, 20, N_2, 20\}$ with missing C.location N_2 is imputed as 2206, ZIP generates the answers resulting from this imputation for all the join conditions containing C.location in the tree. In this query tree, $\widehat{\bowtie}_{S.Room=C.location}$ is the only upstream join operator of ρ that is applicable to $C.location$. We further assume that the bloom filter of $S.room$ is complete. For instance, assume $S.room$ does not have missing values and is the build side of join. When ZIP replays $t = \{NULL, NULL, 3, 20, N_2, 20\}$ using join condition $S.Room = C.location$, ZIP checks $BF(S.room) = \{2206, 2011, 2065\}$ and the imputed value 2206 is found. ZIP then retrieves the matched tuple in relation S which is $\{2206, DBH\}$, and updates t to $\{2206, DBH, 3, 20, 2206, 20\}$, as shown as the tuple

[5] ZIP requires indices on all join attributes. If such an index does not exist, ZIP will create a hash index as part of the execution of the join operator.

Algorithm 2: REPLAY

Input: t, o, a

b : the join attribute in o other than attribute a
if $t.a$ *not in* $BF(b)$ **then**
    return $\emptyset$
else
    $T_{matched} \leftarrow look\_up(t.a)$
    $T_{matched} \leftarrow T_{matched} \setminus \mathcal{L}(o, b)$
    $Ans \leftarrow \emptyset$
    for $t_i \in T_{matched}$ **do**
        $Ans \leftarrow Ans \cup merge(t, t_i)$
    return Ans

② in Figure 3.7-g. Note that such updates can be easily achieved since the schema of each composite tuple is maintained in each operator and we could project the matched tuples into corresponding fields in t by aligning their schema.

Discussion:

The correctness of ZIP requires 1) soundness: the tuples returned by ZIP would have been returned had we imputed in the base relations prior to executing the query; 2) completeness: ZIP will not miss a result; 3) non-duplicates: ZIP will not generate duplicated results. We focus on the join execution since proving the correctness of the other operators, i.e., unary operators, is simpler. Consider join operation $L.a = R.b$, let $\mathcal{T}_d^L$ be the tuples in L that pass filter and verify steps and have missing values in $L.a$, and $\mathcal{T}_c^L$ be the tuples in L that pass filter and verify steps and do not have missing values in $L.a$, respectively. Likewise, $\mathcal{T}_d^R$ and $\mathcal{T}_c^R$ are similarly defined. $L.a \bowtie R.b$ can be rewritten as $(\mathcal{T}_c^L \cup \mathcal{T}_d^L) \bowtie (\mathcal{T}_c^R \cup \mathcal{T}_d^R)$. In join operator $o = \bowtie_{L.a=R.b}$, the I-operation of o will implement $\mathcal{T}_c^L \bowtie \mathcal{T}_c^R$ as normal join. Tuples in $\mathcal{T}_d^L$ and $\mathcal{T}_d^R$ will be pushed to the downstream operators by appending NULLs for the attributes in the other relation. In later query processing, when the bloom filter of $R.b$ is complete, $\mathcal{T}_d^L \bowtie R.b$ will be computed by the generate step. Similarly, when $L.a$ reaches its bloom filter completeness condition, $L.a \bowtie \mathcal{T}_d^R$ will be generated. Note that this may result in the duplicated results for $\mathcal{T}_d^L \bowtie \mathcal{T}_d^R$. Recall that ZIP maintains a list $\mathcal{L}(o, a)$ (or $\mathcal{L}(o, b)$)

in every join operator o to prevent the generation of such duplicated tuples. (See Line 6 in Algorithm 2)

3.6 Decision Function

In the decision function in ZIP, the decision of whether a missing attribute value should be imputed prior to the execution of the operator or should imputation be delayed depends upon whether the imputation method is non-blocking or blocking. We focus on an *adaptive* cost-based solution for non-blocking imputations, denoted by ZIP-adaptive. For blocking imputations, we use a *lazy* strategy, denoted by ZIP-lazy, which always delays imputing until the tuple with the missing value reaches the imputation operator ρ. Alternatively, we can use an eager strategy, wherein imputations are performed as soon as required.

3.6.1 Obligated Attributes

Non-blocking imputations in ZIP can be placed anywhere in the query tree since ZIP, through operator modification, decouples imputation from the operator implementation. To guide the actions of each operator, we first define a concept of *o*bligated attributes for relations in query Q. Intuitively, an attribute a in R is obligated if missing values of a in R "must" be imputed in order to answer the query, i.e., for such attributes, its values cannot be eliminated as a result of other query conditions or due to imputation of other missing values.

Definition 3.1. (Obligated Attributes) *Given the set of attributes in the predicate set of a query Q (denoted by A_Q), an attribute a in relation R is said to be obligated if*

- *attribute a appears in a predicate in Q, i.e., $a \in A_Q$, or a is one of the attributes listed in a projection operator; and*

- *all attributes of R (other than attribute a) do not appear in any predicate in Q. That is, $\forall a' \in R - a$, $a' \notin A_Q$.*

If an attribute $a \in R$ is neither in the projection list nor in A_Q, imputing its missing values will not be required to answer Q and hence a would not be obligated. Likewise, if a predicate in A_Q contains an attribute b which is also in R, it is possible that such a predicate may result in the tuple of R to be eliminated thereby making imputation of the corresponding a value (in case it was missing) unnecessary. Thus, again, such a possibility would prevent a from being classified as obligated. As an example in Table 3.2, U.faceID is an obligated attribute for query Q in Figure 3.2 because other attributes U.name and U.type are not in any predicate of query Q and U.faceID is in join predicate U.faceID=T.faceID.

Since missing values of obligated attributes must always be imputed, there is no benefit in delaying their imputations. In contrast, imputing could potentially reduce the number of tuples during query processing. As a result, the decision function in the ZIP operators never delay such imputations. For the remaining attributes, ZIP performs a cost-benefit analysis to decide whether to impute a missing value or delay its imputation.

3.6.2 Decision function

For each operator o in the query tree, ZIP associates a decision function $df(a, o)$ for all attributes a that appear in the predicate associated with o. The decision to delay/impute missing values has implications on both imputation and query processing costs. Consider a tuple t in relation $R = (a, b, c, d)$ and a query tree in Figure 3.9-a). Say $t_1 = (N_1, 1, 2, 3)$ (N represents missing value), if we delay imputing $t_1.a$, and $t_1.b$ does not join with any tuples in the other relation, we can avoid imputing $t_1.a$. On the other hand, imputing $t_2.a$ for $t_2 = (N_1, N_2, 2, 3)$, could prevent imputation of $t_2.b$, if the imputed value of $t_2.a$ is filtered in the selection operator. Imputing $t_2.a$ may also reduce query processing time since it does

not require the operator on attribute b to be executed.

Since decisions on whether to impute/delay are made per tuple containing missing values locally by the operator, the decision function must not incur significant overhead. In making a decision for operator o_1 over attribute value $t.a$ of a tuple t, ZIP assumes if t contains other missing values in attributes, say $t.b$ on which predicates are defined in downstream operators, say o_2, those operators will decide to impute $t.b$ if the tuple t reaches those operators. For instance, in the query tree in Figure 3.9-a), in making a decision for imputing /delaying $t.a$, i.e., N_1, in developing a cost model we assume that the missing value N_2 ($t.c$) will be imputed right away. This prevents ZIP from recursively considering a larger search space that enumerates, which leads to a potentially exponential number of other possibilities wherein downstream operators may delay/impute.

We build a cost model below to estimate the impact of delay/impute decision on both the imputation cost and the query processing cost based on which the operators make decisions in ZIP. To compute the imputation and query processing costs associated of the decision for an operator, ZIP maintains the following statistics:

• $impute(a)$: Cost of imputing a missing value of attribute a, computed as an average over all imputations performed so far for missing values of a.

• Selectivity of selection operator o_i, $S_{\emptyset_i} = \frac{|T_s|}{|T_c|}$, where T_c (T_s) are tuples that are processed (satisfy) the predicate associated with $\emptyset_i$.

• Selectivity of join operator between relation L and R computed as $S_{o_i} = \frac{|T_s|}{|T_L||T_R|}$, where T_L (T_R) are tuples in relation L (R) and T_s are tuples that satisfy o_i [6]

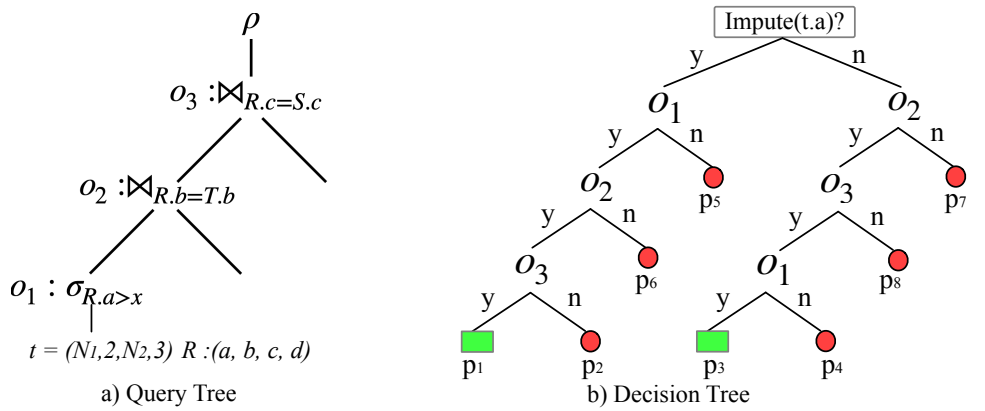
• $TTJoin_o$: the average time to join tuples in (join) operator o; [7]

• $\mathcal{T}_o$: the average number of evaluation tests to perform per tuple in operator o for tuples without missing values in the attribute to be evaluated in o. [8] If o is a join operator,

[6] We exclude those tuples containing missing values from T_s, T_c, T_L and T_R.

[7] We also use the notation $TTjoin_o$ for selection operator, in this case, $TTjoin_o = 0$.

[8] A tuple with missing value in the attribute that passes through o will be pushed to the above operator



path	Expected Imputation Cost
p5	impute(a)
p6, p1, p2	So1*So2*impute(c) + impute(a)
p7, p8	So2*impute(c)
p3, p4	So2*So3*impute(a) + So2*impute(c)

c) Expected Imputation Cost

path	Probability	Query Processing Cost (QP)
p5	(1-So1)	To1*TTJoino1
p6	So1(1-So2)	To1 *To2* TTJoino2+QP(p5)
p1	So1So2So3	To1 *To2* To3 * TTJoino3+QP(p6)
p2	So1So2(1-So3)	To1 *To2* To3 * TTJoino3+QP(p6)
p7	(1-So2)	To2*TTJoino2
p8	So2(1-So3)	To2 *To3* TTJoino3+QP(p7)
p3	So2So3So1	To2 *To3* To1 * TTJoino1+QP(p8)
p4	So2So3(1-So1)	To2 *To3* To1 * TTJoino1+QP(p8)

d) Expected Query Processing Cost

Figure 3.9: Decision Function Example.

evaluation tests refer to join tests. Otherwise, if o is selection operator, we set $\mathcal{T}_o = 1$.

To bootstrap the process of statistics collection, ZIP initially delays all imputations forcing tuples to rise up to the top of the tree, and these tuples can be dropped if they fail some predicates en route. During this process, ZIP collects imputed tuple samples to compute $impute(a)$ and to determine other statistics such as $\mathcal{T}(o)$, join cost $TTJoin_o$ and selectivity S_{o_i}. These statistics are then adaptively updated during query processing.

Cost Model for Imputations.

We illustrate how to estimate the imputation cost.

Consider a query tree in Figure 3.9-a), and a tuple $t = (N_1, 2, N_2, 3)$. To decide whether to impute or delay missing value $t.a$ (N_1), ZIP estimates the total imputation cost in case it chooses to impute or to delay imputing $t.a$. The set of possible executions that may result for either of the decisions are illustrated in the decision tree shown in Figure 3.9-b). Each

without evaluation immediately, and thus $\mathcal{T}_o$ for such tuple is 1.

path of the tree corresponds to a possible outcome based on the decision to impute/delay imputing $t.a$. For instance, in path p_5, $t.a$ is imputed but fails the predicate in o_1, while in path p_3, $t.a$ is delayed and t passes the predicates associated with o_2 and o_3, and reaches the imputation operator ρ, where $t.a$ is imputed and evaluated in ρ using predicate associate with o_1. The estimated imputation cost in the case of imputing (delaying) $t.a$, is the summation of the *expected* imputation cost of all paths in the left (right) side of the tree, i.e., p_1, p_2, p_5, p_6. (p_3, p_4, p_7, p_8). The expected costs of various paths (shown in Figure 3.9-c) are computed as a weighted sum of imputations along the path, where the weight corresponds to the probability of execution of that imputation. For instance, for path p_1, we impute $t.a$ with the probability of 1, and, subsequently impute $t.c$ with the probability of $S_{o_1}S_{o_2}$. Thus, the cost of path p_1 is $impute(a) + S_{o_1}S_{o_2}impute(c)$.

Cost Model for Query Processing. Since join costs dominate query execution over the unary operators in most cases, ZIP estimates query processing costs by the corresponding join costs. Consider the same decision tree in Figure 3.9-b). The expected query processing cost if we impute (delay) $t.a$ is the sum of the expected query processing costs for all the paths on the left (right) side of the tree. Figure 3.9-d) lists the probability of each path, and also, its query processing cost. The probability is estimated based on the selectivity of the predicates along the path, and the cost is estimated by summing execution cost of execution of operators along the path incurred in processing tuple(s) that are generated as a result of processing t. Take p_6 as an example. Its corresponding probability is $S_{o_1}(1 - S_{o_2})$ since t passes o_1 but fails o_2. The estimated cost for processing t (shown in Figure 3.9-a) in operator o_1, denoted by $QP(o_1)$, is $\mathcal{T}_{o_1} * TTJoin_{o_1}$ which is 0 in this example since o_1 is a selection operator for which $\mathcal{T}_o$ is 1 and $TTJoin_{o_1} = 0$. The cost $QP(o_2) = \mathcal{T}_{o_1}\mathcal{T}_{o_2} * TTJoin_{o_2}$ since o_2 is a join operator and $\mathcal{T}_{o_1}\mathcal{T}_{o_2}$ is the estimated number of join tests to perform in o_2. The decision function will decide to impute missing values if the estimated cost of imputation is lower. Otherwise, if the estimated cost of query processing is lower, the imputation will be delayed.

3.7 Experimental Evaluation

In this section, we evaluate ZIP over two real data sets and one synthetic data set. We implemented ZIP on top of a database prototype system, SimpleDB [4] [9]. Note that the ImputeDB optimizer is also implemented in SimpleDB. We did so, so that we can directly measure the improvements due to ZIP on ImputeDB query plans.

3.7.1 Experimental Setup

We used the following three data sets to evaluate ZIP.

WiFi. The first data set consists of WiFi connectivity events at University of California, Irvine. WiFi-based occupancy determination has recently received a lot of attention due to the pandemic with several companies offering related products [12, 16, 15] and research projects [73, 79, 77]. The database consists of three tables, *users*, *wifi* and *occupancy* with 4018, 240,065 and 194,172 number of tuples, and total 383,676 missing values respectively. WiFi records the continuous connectivity data of devices - i.e., which device is at which location in which time interval. *occupancy* records the number of people at different locations over time.

CDC NHANES. We use the subset of 2013–2014 National Health and Nutrition Examination Survey (NHANES) data collected by the U.S. Centers for Disease Control and Prevention (CDC) [1]. [10] The CDC data set has three tables, demo, exams, and labs, which are extracted from a larger complete CDC data set. demo, exams and labs have 10175, 9813, and 9813 tuples, respectively, and all of them have 10 attributes. Among them, there are a total of 24 attributes that contain missing values, whose missing rate ranges from 0.04% to

[9] SimpleDB, developed at MIT has been used for research purposes at several universities including MIT, University of Washington, and Northwestern University.

[10] We thank ImputeDB [34] for providing this data set whose link can be found in [3].

97.67%, with a total of 81,714 missing values.

Smart Campus. To test ZIP on larger datasets, we generated a smart-campus data set consisting of 3 synthetic sensor tables, WiFi, Bluetooth, Camera, a space table (that connects sensors to locations) user table (that connects a user to a device mac-address). In addition, two additional tables are derived from the sensor data. The first table location consists of the location of users over time and the second table occupancy consists of the number of people at a given location over time. smart-campus data set has totally 1,892,500 tuples and 1,634,720 missing in occupancy and location and occupancy tables.

Query Set We create three query workloads to evaluate ZIP, *random* (with random selectivity), *low-selectivity* and *high-selectivity*. In each query workload, the majority of queries are SPJ-aggregate queries that contain *select*, *project*, *join*, *aggregate* (*group by*) operations. SP queries are also included. Each query workload contains 20 queries.

Imputation Methods Note that any imputation approach could be approapiately used in ZIP, and we choose several popular and easily used methods, which could be called in standard Python library or be well-packaged in Github. For the CDC NHANES dataset we use three imputation approaches, Top-k nearest neighbor [13] (KNN), XGBoost [35, 6], and histogram-based mean value imputation [34]. Of these, the first two are blocking while the third is non-blocking. KNN, XGBoost, and mean value imputations are widely used and their implementations are available in standard Python packages, such as sklearn or xgboost. For the WiFi and Smart Campus data set to impute location and occupancy values, in addition to using the above three approaches, we further use a proprietary non-blocking imputation method LOCATER [79] (LOC in short). We now describe the complexity of each imputation method to impute one missing entry. Mean takes constant time $O(C)$ to find the mean value in a histogram. LOC uses $O(Nk)$ where N is the average number of rows of the queried person and k is the iteration number, recommended as 10. KNN takes $O(Nd)$ where N and d are the number of rows and columns in the table to be imputed. XGboost

		Time(secs)			# of Imp ($*10^3$)		
Data Sets	Imputation	QTC-Eager	ZIP-lazy	ZIP-adaptive	QTC-Eager	ZIP-lazy	ZIP-adaptive
WiFi	Mean	1.6	2.3	1.6	79	3.2	74
	LOC	393.4	19.1	19.2	78	3.4	3.6
	KNN	769.2	30.6	-	79	3.3	-
	XGboost	144.7	126.2	-	79	3.2	-
CDC	Mean	0.12	0.12	0.12	12.1	1.3	10.7
	KNN	9.63	1.02	-	12	1.3	-
	XGboost	42.5	37.8	-	11.9	1.3	-
Smart Campus	Mean	3.8	9.6	3.8	16	4.7	15.9
	LOC	97.8	32.5	27.2	16	4.7	4.8
	KNN	157.1	44.3	-	16	4.6	-
	XGboost	101.8	72.6	-	16	4.7	-

Table 3.4: ZIP VS QTC-Eager.

(Seconds)	Min	Max	Avg	Avg Speed Up
WiFi	3.9	46.3	19.2	1200X
WiFi-Large	6.2	74.5	33.6	19607X

Table 3.5: ZIP VS Offline.

takes $O(tdxlogn)$ to train, and $O(td)$ to infer one missing value, where t is the number of trees, d is the average height of the tree and n is the number of non-missing values in training data.

Strategies Compared We evaluate the two versions of ZIP - *ZIP-lazy* and *ZIP-adaptive* as defined in Section 3.6 with a baseline query-time strategy, *QTC-Eager*, that imputes missing values eagerly without delay during query execution as soon as the imputed value is required during query processing. Comparing ZIP to QTC-Eager will show the benefits of the lazy imputation strategy. We also compare against the offline approach that first imputes all missing data and then runs queries. In Experiment 1 to 5 below, we use the query plan generated by PostgreSQL as the input for ZIP to execute. [11] In Experiment 6, we incorporate ZIP with query plans generated from ImputeDB and explore the performance of the combination.

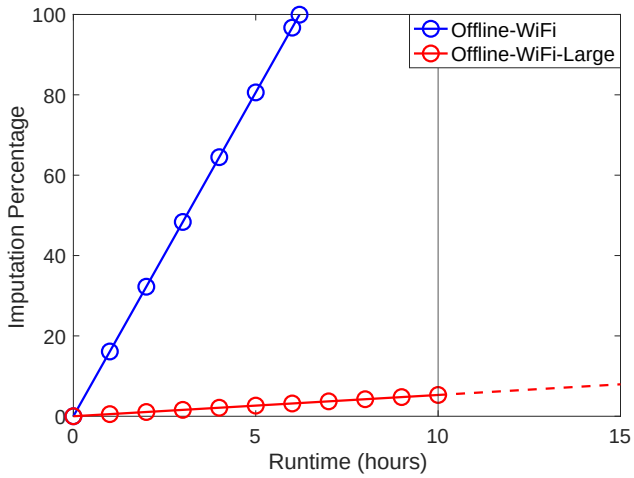


Figure 3.10: ZIP VS Offline.

3.7.2 Evaluation

Experiment 1: ZIP VS Offline. To show the needs of query-time imputation (i.e., ZIP) by comparing with the offline approach, we collected a larger real data set WiFi-large by extending the WiFi data set from one building to 40+ buildings over campus, and we report the runtime of ZIP and the offline approach in Table 3.5 and Figure 3.10. In Figure 3.10, the offline approach takes 6.4 hours in WiFi data set and estimated 183 hours in WiFi-large [12] data set, respectively. In contrast, in Table 3.5, ZIP has only 19.2 and 33.6 seconds of run time in WiFi and WiFi-large data sets, and it speeds up the offline approach by 1200X and 19607X in WiFi and WiFi-large data sets.

Experiment 2: ZIP VS QTC-Eager. In Table 3.4 we report the runtime (in seconds) and number of imputations, i.e., the number of missing values imputed for *QTC-Eager*, *ZIP-lazy* and *ZIP-adaptive* approaches using the *random* query set. Note that LOC is applied on WiFi and Smart-Campus since it is not applicable on CDC data set. We compare the performance of QTC-Eager with ZIP-lazy and ZIP-adaptive in WiFi, CDC, and Smart-Campus in Figure 3.11, Figure 3.12, and Figure 3.13, respectively. We show the percentage of

[11] We implement an API in simpleDB that could read and translate the PostgreSQL plan to be executed in its executor.

[12] We stop cleaning at 10 hours and the offline approach only imputes around 5% missing data.

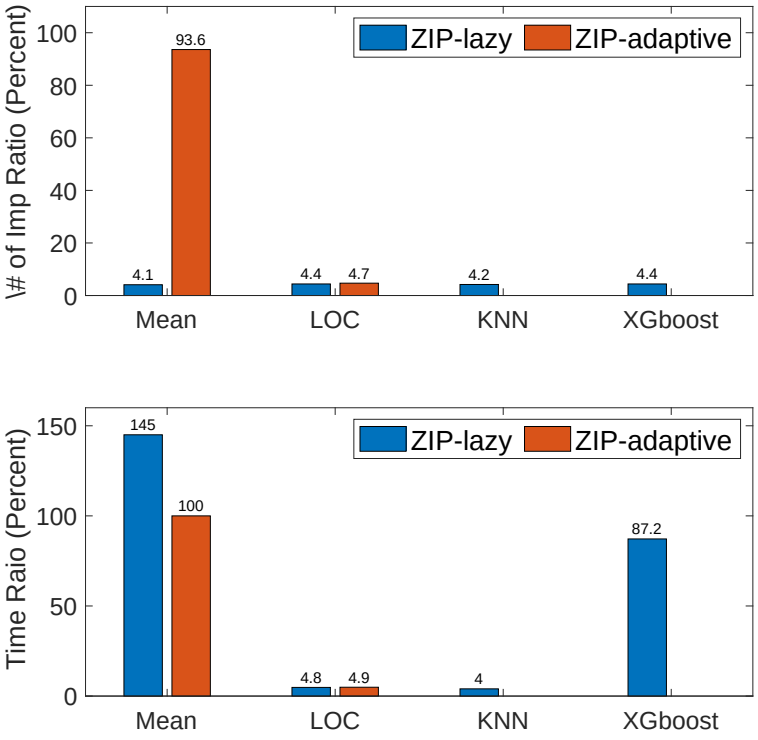


Figure 3.11: ZIP VS QTC-Eager in WiFi Data.

run time and number of imputations performed by ZIP-lazy and ZIP-adaptive compared to QTC-Eager. *time ratio* is the run time of ZIP-lazy (or ZIP-adaptive) divided by the time of QTC-Eager times 100 (percentage). Similarly, # *Imp ratio* is the percentage of imputation numbers.

We make several observations. First, ZIP-lazy and ZIP-adaptive perform similarly and they both outperform QTC-Eager by around 20x when expensive imputations are used, which demonstrates that delaying imputations significantly improves performance when imputations are expensive. Second, when cheap imputations are used such as Mean imputation, ZIP-adaptive tends to impute data first since doing so will potentially save the query processing time by reducing temporary tuples, and thus has similar imputations as QTC-Eager. This shows that the decision function in ZIP-adaptive works correctly and is able to actively adjust its decision based on the cost of imputations. Third, ZIP-lazy requires slightly fewer imputations than ZIP-adaptive since it always delays imputation to the end of processing, while ZIP-adaptive performs similarly as ZIP-lazy in WiFi and CDC and ZIP-adaptive slightly

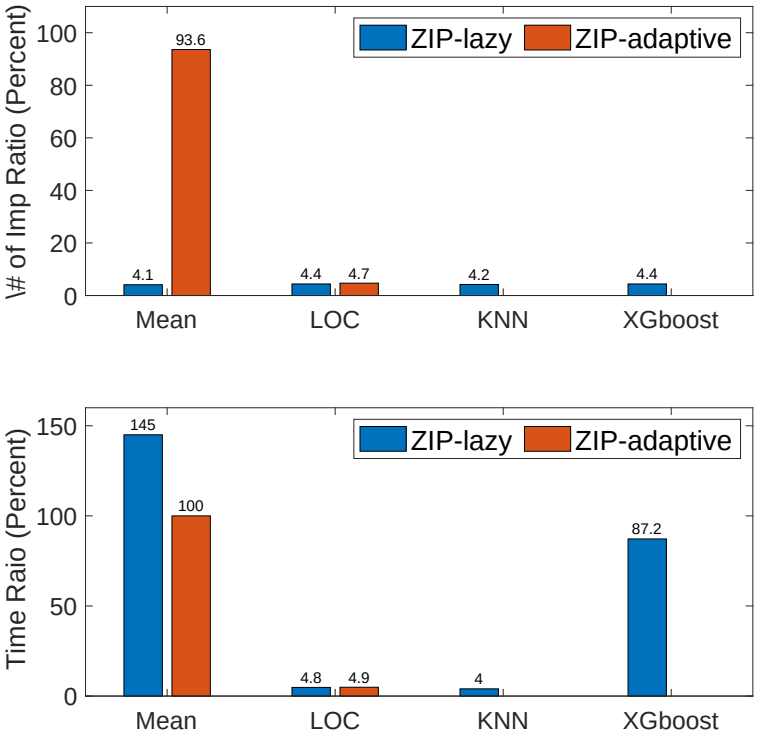


Figure 3.12: ZIP VS QTC-Eager in CDC Data.

outperforms ZIP-lazy in Smart Campus due to the more complex join workload (higher join selectivity). Fourth, as expected, when learning approaches are used whose training time dominates the inference costs (e.g., as in XGboost), reducing the number of imputations will not offer a big improvement. For instance, ZIP-lazy imputes 11.6% of imputations needed for QTC-Eager, but it takes 89% run time of QTC-Eager (i.e., saving only 11%).

Experiment 3: Quality of Query Answer. The quality of query answer depends upon the imputation method used and the query itself. Figure 3.14 plots the accuracy of imputation methods (Acc_I) and that of the corresponding query answers (Acc_Q). Acc_I is the percentage of correctly imputed values. For aggregate queries, Acc_Q is measured as $|\frac{\mathcal{A}^{\mathcal{T}}-\mathcal{A}}{\mathcal{A}^{\mathcal{T}}}|$, where $\mathcal{A}^{\mathcal{T}}$ is the true answer and $\mathcal{A}$ is the answer returned using an imputation method. For set-based queries, Acc_Q is measured using Jaccard similarity. Using different imputations with different accuracies, the accuracy of query answers is also different. The experiment above highlights the impact of the choice of the imputation method on query answer quality. To measure the *difference* between answers returned by ZIP and the offline approach, we use

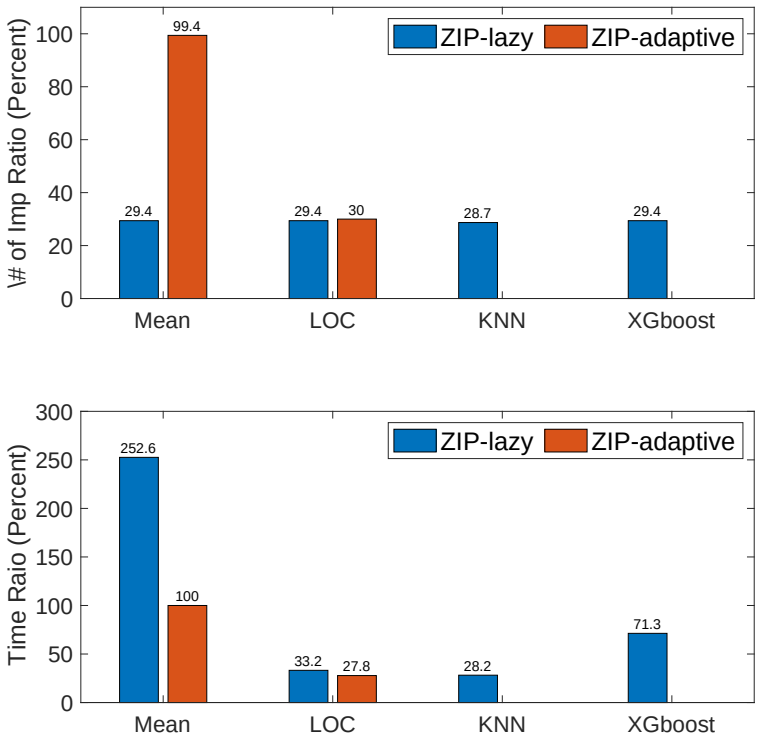
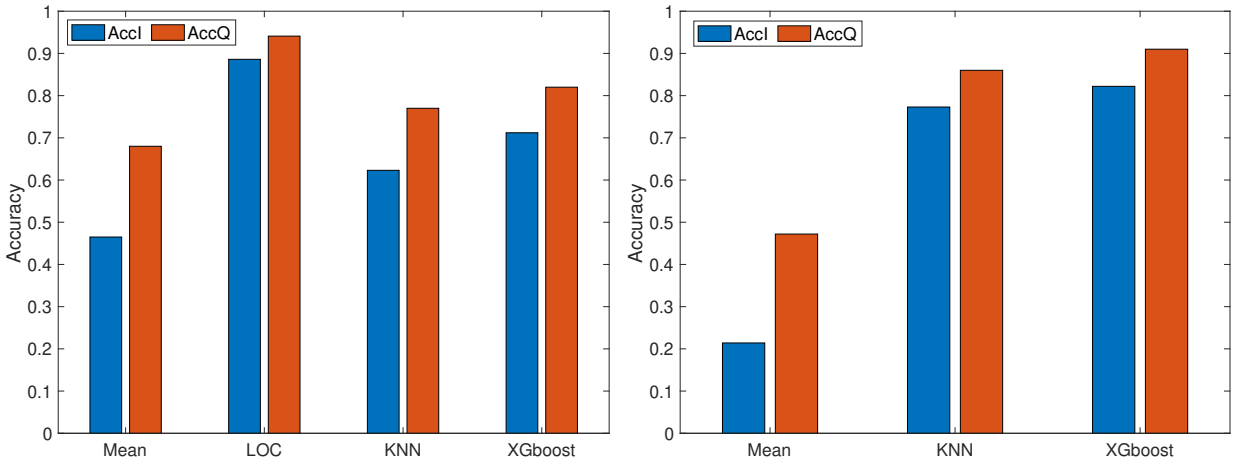


Figure 3.13: ZIP VS QTC-Eager in Smart Campus Data.



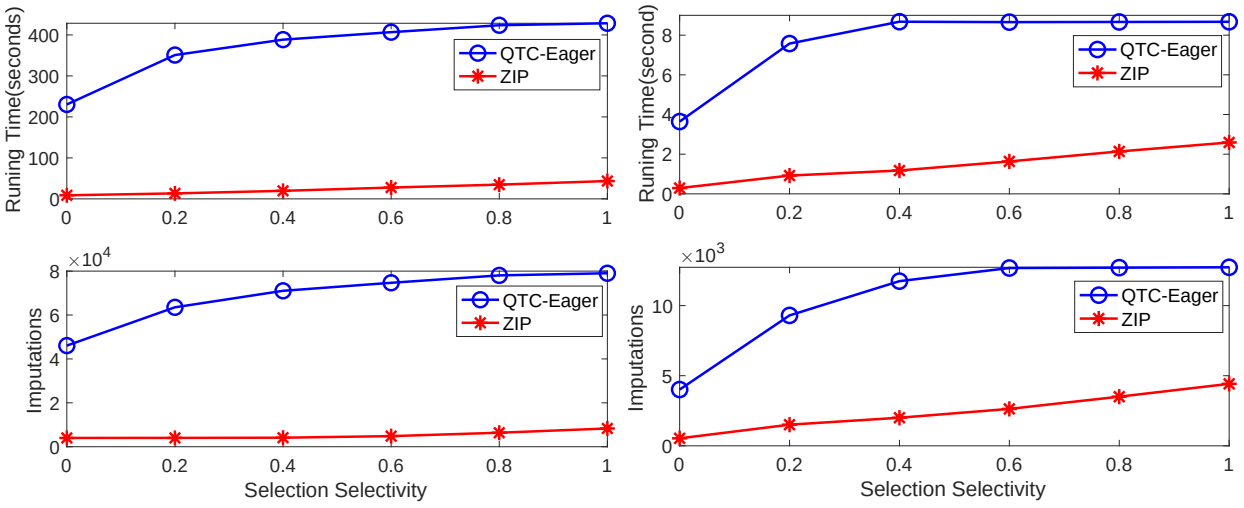
(a) WiFi (b) CDC

Figure 3.14: Accuracy of Imputations and Query Answer

Symmetric-Mean-Absolute-Percentage-Error (SMAPE) [85] (also used by ImputeDB [34]). SMAPE is computed as a tuple-wise absolute percentage deviation between ZIP & offline approach. Since ZIP-lazy and ZIP-adaptive all return exactly the same answers as offline, the SMAPE value for all is 0.

Experiment 4: Query Selectivity Effects. We use the query template below to generate *low-selectivity* and *high-selectivity* query workloads: SELECT a, AVG(b) FROM $R_1, ..., R_n$ WHERE [$Pred_J$] [$Pred_S$] GROUP BY a., where $Pred_J$ and $Pred_S$ are join and selection predicates. We varied the selectivity of each selection predicate as 0, 0.2, 0.4, 0.6, 0.8, 1, and the selectivity of join predicate is set to be low and high by modifying the matching numbers of joined attribute values. KNN is applied in CDC data set, while in WiFi and Smart-Campus data set, LOC is used to impute location and occupancy, and Mean-value is used to impute other missing values. In CDC and WiFi data set, we report the effect from the selectivity of selection predicates in Figure 3.15, and the effects from both join and selection selectivity are evaluated in the synthetic data set in Figure 3.16. In this and later experiments, if no ambiguity, we call ZIP the hybrid of ZIP-lazy and ZIP-adaptive – ZIP always uses ZIP-adaptive for non-blocking imputations and ZIP-lazy for blocking imputations.

The number of imputations and running time increase for both QTC-Eager and ZIP when the selectivity of predicates increases, though ZIP has considerably lower overhead and running time at all selectivity levels. In CDC, since KNN is costly and join operations are selective, ZIP delays imputations in selection operators. Join predicates help eliminate tuples reducing imputations needed thus reducing cost. In WiFi where join attributes (e.g., location) have missing values, instead of paying expensive imputations (e.g., LOC) to impute all missing values in join attributes as QTC-Eager, ZIP delays the imputations of the missing values in join and allow the downstream operators to wisely choose cheap imputations (e.g., Mean) for missing values that are not in join attributes to help eliminate tuples if the corresponding predicates are selective. In the synthetic data set, when the join operators are highly



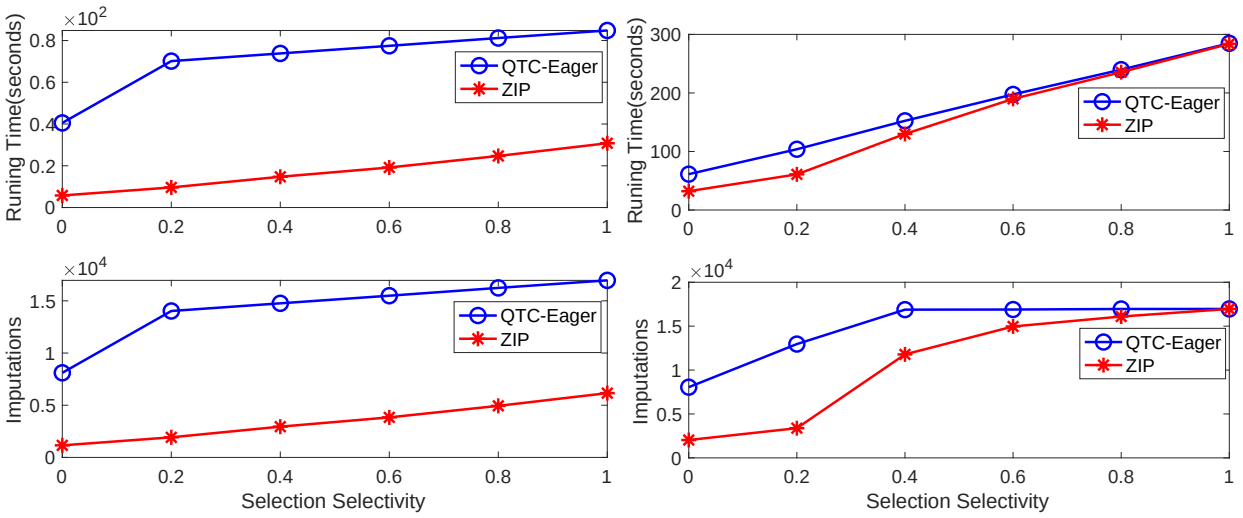
(a) WiFi Data Set. (b) CDC Data Set.

Figure 3.15: Selectivity Effects on Real Data Set.

selective, (i.e., selectivity is low) as in Figure 3.16(a), ZIP delays imputing missing values in selection operators to exploit join operators to help eliminate most tuples and thus save most imputations. When join is less selective (i.e., more result tuples in Figure 3.16(b)) but the selection predicates are selective, ZIP might choose to partially delay the imputations in join attributes and the downstream operators that could be selective due to other selections, to help remove tuples to avoid expensive imputations in join attributes (e.g., location). However, when the selection predicates are also less selective and query processing overhead is higher than imputation costs, ZIP prefers imputing missing values immediately same as QTC-Eager.

Experiment 5: The effect of Missing Rates. Figure 3.17 shows how missing rates affect the query performance. With the increasing missing rates, the runtime and imputation times taken by ZIP slightly increase and tend to converge, since the amount of imputations needed to answer a given query depends on the selectivity of the query. For the set of queries with fixed selectivities, their performance will not be dramatically affected by the number of missing data in the data, which demonstrates the robustness of ZIP.

Experiment 6: ZIP with ImputeDB plans. We investigate ZIP using ImputeDB-



(a) Low Join Selectivity.

(b) High Join Selectivity.

Figure 3.16: Selectivity Effects on Synthetic Data Set.

generated query plans. ImputeDB [34] adds *impute* and *drop* operators to a query plan based on a parameter α ($0 \leq \alpha \leq 1$) that balances efficiency and quality. Higher the value of α, more drop operators are used causing more tuples with missing values to be dropped. To incorporate ImputeDB plans in ZIP, we added the drop operators that mirror the drop operator in ImputeDB. Given an ImputeDB plan, the *impute* operators are treated as regular while ZIP impute operators that could be evaluated lazily based on the decision function. The *drop* operators check if a tuple contains missing values in the appropriate attribute and, if so, drop the tuple. We generated several plans based on varying α from 0 to 1 and execute the plan both in ImputeDB and in ZIP modified as above to support ImputeDB plans with *drop* operator.

Figure 3.18 shows the average quality (1-SMAPE) versus runtime of queries for the CDC and WiFi data using the KNN imputation approach on *random* query sets for ImputeDB plans with and without ZIP. Each line plot shows 6 points corresponding to $\alpha = 1, 0.8, 0.6, 0.4, 0.2,$ and 0 from left to right. The improvements are 10x to 25x when $\alpha = 0$ (i.e, when ImputeDB plan optimizes for quality, choosing *impute* over *drop*). When α increases, the relative improvements due to ZIP reduces. This is expected since ZIP only applies to tuples that are imputed. With increasing α more tuples are dropped reducing the need for ZIP. When

the cost-quality tradeoff across ImputeDB plans with and without ZIP, to achieve the same level of quality, plans with ZIP require significantly less time. Furthermore, ImputeDB plans when executed with ZIP achieve significantly higher quality in the same amount of time spent in query processing. Thus, the execution of ImputeDB plans with ZIP significantly dominates the ImputeDB plans without ZIP. The experiment clearly establishes that even in use cases where we explore cost-quality tradeoffs, ZIP-optimization can help improve systems such as ImputeDB that explore such tradeoffs, and ZIP and ImputeDB are complementary approaches integration of which provides a powerful query time imputation approach.

Experimental Summary. We below summarize the main outcomes of the experiments in ZIP and what we learned.

- The offline approach that first imputes all the missing data and then performs query processing is not practical when the size of the data is large and the imputations are expensive as shown in Experiment 1. In such a case, ZIP is able to support interactive analysis and outperforms the offline approach by up to 19607X in a large WiFi data set.

- By comparing with another baseline approach QTC-Eager which is the underlined query processing technique in the state-of-the-art system, EnrichDB, ZIP outperforms QTC-Eager by around 20X when expensive imputations are used, such as LOCATER or KNN. It shows the benefits of the lazy imputation strategy.

- ZIP will generate the exact same query answers as the offline approach.

- ZIP provides higher improvement when the queries are more selective, and ZIP is relatively robust to the missing rates in the dataset.

- ZIP used together with the ImputeDB plan (the state-of-the-art system to generate query plans that optimize for missing value imputation), outperforms the ImputeDB

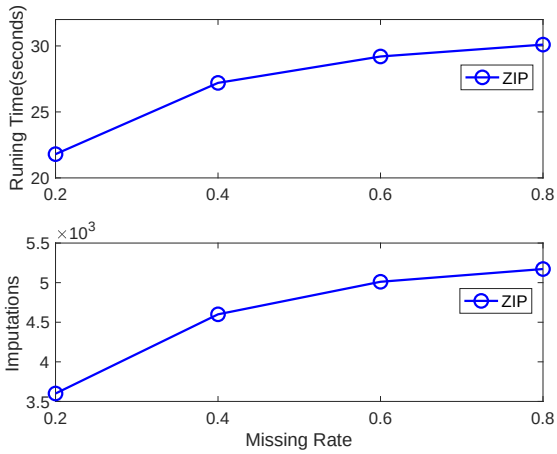


Figure 3.17: Missing Rate Effects.

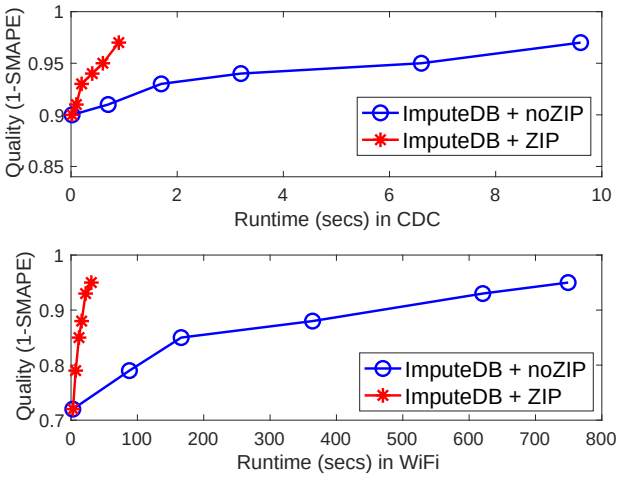


Figure 3.18: ZIP + ImputeDB.

by 10X to 25X, which shows that ZIP and ImputeDB can be used complementarily to further improve the performance of the system.

3.8 Conclusion

This chapter studies query-driven missing value imputation and proposes ZIP, a technique to intermix query processing and missing value imputation to minimize query overhead. Specifically, ZIP co-optimizes imputation cost and query processing cost, and proposes a new implementation based on outer join to preserve missing values in query processing. Extensive experiments on both real and synthetic data sets show that ZIP outperforms the offline approach by up to 19607 times in a real WiFi data set.

Chapter 4

PLAQUE: Automated Predicate Learning at Query Time

In this chapter, we introduce our second strategy to accelerate query execution to support interactive analysis when computationally expensive operations are used, by learning new predicates at query runtime.

4.1 Introduction

Predicate pushdown based on selectivity and cost estimates is a key strategy used to optimize queries in relational databases. Pushing predicates down in a query tree could lead to significant savings by reducing the size of data that migrates to downstream operators. In this chapter, we seek a new approach to query processing, entitled PLAQUE, automated *P*redicate *LeA*rning at *QU*ery tim*E*, that learns selective predicates *during* query execution beyond those listed explicitly in order to filter out tuples that would not result in any query results as early as possible during query processing. To illustrate the key idea behind

PLAQUE, we examine a slightly modified and simplified version of TPC-H Query Q-10 that includes a theta-join condition. [1] In this query, the predicates o_orderdate < "1993-01-01" and p_brand = ":10" can be pushed down to the orders and part tables. However, the query contains no predicates on the lineitem table that could prune non-matching records in the lineitem table that do not result in any query answers. Thus, any query plan will scan over all records in the lineitem table.

SELECT MAX(l_discount)
FROM part, lineitem, orders
WHERE p_retailprice < l_extendedprice **AND** o_orderkey = l_orderkey **AND** o_orderdate < "1993-01-01" **AND** p_brand = ":10"

Consider that all records in the lineitem table that result in an answer satisfy a predicate $l\_discount > 0.7$ - we will momentarily see how PLAQUE learns such predicates. Query execution can be significantly accelerated by pushing such predicates down in the query to filter records in the lineitem table. Only a small fraction of records in the lineitem table will need to join with the orders and part tables resulting in significant savings.

Several prior works have explored learning predicates, other than those specified explicitly in queries, to reduce downstream query processing. Such approaches [29, 88, 52, 107], typically learn predicates prior to the execution of the query, especially based on exploiting query predicates on join columns. For instance, if the query above contained a predicate l_orderkey < 5) in addition to the other predicates listed in the query, techniques such as [120] could infer a new predicate o_orderkey < 5) which could then be used to filter tuples from the orders table to speed up the query execution. Such prior work on learning predicates, however, is of limited applicability since queries containing equi joins seldom contain additional selection predicates on the join column. This can be observed by examining such equi join queries over several real datasets and synthetic benchmarks such as TPC-H [19] or TPC-DS [18] in

[1] The similar query is used in previous works [111, 119] to evaluate theta-join in TPC-H benchmark.

which none of the equi join queries contain additional predicates on the join columns. As such, above mentioned techniques rarely result in a significant reduction of the execution cost in the benchmark queries. An alternate strategy that empowers predicates learned ahead of query execution has been explored in [67]. In this strategy, the system maintains data statistics (e.g., min and max of columns) at the data block level which is used for sideways information passing over equi joins to accelerate query execution, especially in big-data systems such as Hive [106] or Pig [91] where data is partitioned across clusters. The work, however, is limited to equi joins.

In this chapter, similar to [62, 121, 93], we propose PLAQUE that learns predicates during query processing. In contrast to them, PLAQUE takes a much more comprehensive, as well as, an adaptive approach to learning and using predicates in query execution. PLAQUE infers new predicates not just during the execution of hash-join as in [62, 121, 93], but based on a range of relational operators including aggregation operators such as min and max, theta joins, equi joins, group-by operators, and having conditions in queries. In PLAQUE, as the query execution proceeds and records pass through operators in the query tree, the system learns new predicates to reduce the downstream data processing. Such predicates learned are further refined as query processing proceeds and more data is seen, resulting in improved filters. Furthermore, based on the operator, predicate learning in PLAQUE occurs not just when the system uses a hash-based operator implementation, such as in hash-joins, but also when nested-loop or sort-merge algorithms are used. In PLAQUE, predicate learning and maintenance including predicate refinement is performed efficiently and remains a negligible part of query execution cost. PLAQUE in addition to saving computation cost by pruning unnecessary records, also supports checking of newly-learned predicates using an index-based implementation to reduce I/O cost, and decides on the optimal placement of the learned filters. The placement of the operator depends upon when it is learned and a simple rule such as pushing the predicate as far down the tree as possible may not be optimal. Finally, PLAQUE uses a pre-learning approach to learning new predicates before the execution of

the actual query. This approach fully leverages the power of the optimizer to create a good query plan early, and it is complementary to and combined with the run time predicates learning approach to form a hybrid strategy to boost query performance.

Comprehensive experiments on two benchmarks (TPCH [19] and SmartBench [54]) and one real dataset (IMDB) [14] in Section 5.6 demonstrate that adding the learned predicates using PLAQUE can achieve significant improvement ranging from 2x-33x, especially in queries containing expensive User-Defined-Functions (UDFs) where improvement can be up to 100x in SmartBench [54].

Specifically, this chapter makes the following contributions.

- We build a system entitled PLAQUE that explores a set of novel approaches to infer predicates during query execution from the aggregate, equi join, theta join, group-by, and having conditions in the given query.
- PLAQUE places the learned predicates wisely in the given query tree to maximize the benefits from predicates pushdown using a partial-order based graphical approach.
- PLAQUE exploits the learned predicates using index and in-memory predicates to effectively save both I/O cost and memory footprint.
- We conduct a set of comprehensive experiments on both real and synthetic benchmarks to evaluate the effectiveness of our learned predicates. We further test the learned predicates on queries with UDFs to demonstrate their broader applicability.

4.2 PLAQUE Overview

PLAQUE learns predicates that act as filters to reduce the load on the downstream operations to accelerate query processing. Before we discuss how PLAQUE works, we briefly discuss opportunities that exist during query processing and can be exploited to learn pred-

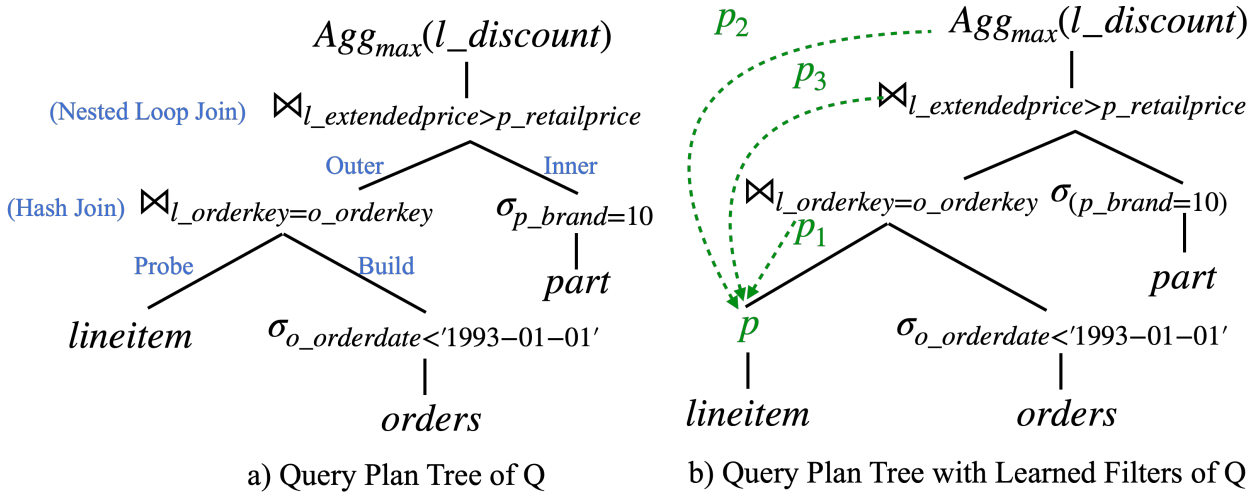


Figure 4.1: Learned Predicates in PLAQUE.

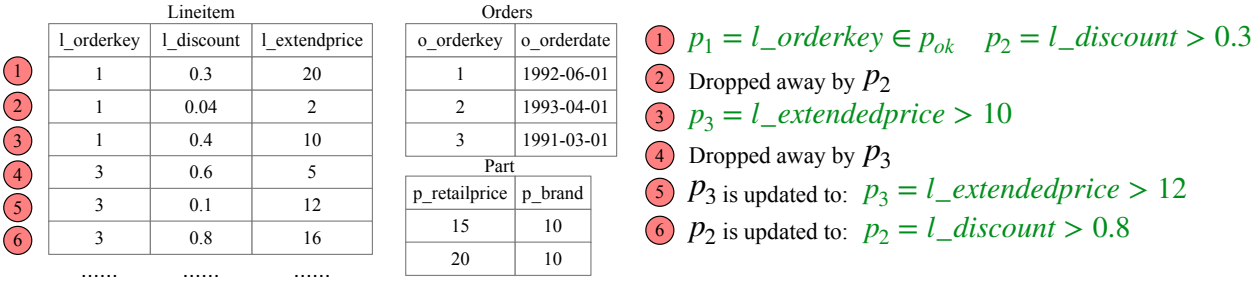


Figure 4.2: Data Observations to Learn Predicates in PLAQUE.

icates.

Opportunities to Learn Predicates

Consider a pipeline query plan tree illustrated in Figure 4.1-a) that corresponds to the query plan generated by PostgreSQL (Version 14.6) for the TPC-H query in Section 1. In this plan, $\bowtie_{l\_orderkey=o\_orderkey}$ is implemented as a hash join and nested loop join is used for $\bowtie_{l\_extendedprice>p\_retailprice}$. One such opportunity to learn a predicate to accelerate query execution is to exploit the hash join implementation of $\bowtie_{l\_orderkey=o\_orderkey}$ that was proposed in the previsous works such as [62, 93]. In particular, since the orders table is the build table and the lineitem table is the probe table, once the hash table on the orders table has been built, all values of the join column of the orders table are known when they have been read during the build phase. Such information can be used to learn a predicate $p_1 = l\_orderkey \in$

p_{ok} that corresponds to all values of *o_orderkey* in the build table (i.e., the orders table) as shown in Figure 4.2. p_1 can be used to filter tuples in the probe side (i.e., the lineitem table). p_1 in this example is a membership predicate that is effective to reduce the size of tuples for the downstream operators. We can alternatively implement the predicate learned from the equi join as a range predicate, which is amenable to support index scan to bring the additional I/O savings.

Besides exploiting equi joins (hash joins in particular) to learn filters, let us explore how other relational operators offer additional opportunities. We continue to use the example in Figure 4.2. For ease of illustration, we use small instances of the lineitem, part and orders tables, respectively.

After the execution of the build phase for the orders table, during the probe phase over the lineitem table, assume a tuple (1, 0.3, 20) (① in Figure 4.2) rises to the join operator $\bowtie_{l\_orderkey=o\_orderkey}$ where it joins appropriate records in the orders and part tables to reach the aggregate operator $Agg_{max}(l\_discount)$. At this stage, we can establish that the final query answer (i.e., $MAX(l\_discount)$) is at least 0.3, since 0.3 is the current maximum *l_discount* in the quantifying tuples reaching the aggregate operator so far. We can, thus, create a new predicate p_2, i.e., $l\_discount > 0.3$, and push this predicate down to p in Figure 4.1-b). Such a predicate can potentially reduce the query execution cost significantly, especially if the maximum value of *l_discount* in the lineitem table is close to 0.3 in an ideal setting. In this case, all the future tuples in the lineitem table can be eliminated from consideration! In our current example, the second tuple ② in the lineitem table will be dropped away by p_2 since its *l_discount* value is 0.04 which is less than 0.3.

Consider now the third tuple ③ in the lineitem table that joins with the record in the orders table to reach the nested loop join with the part table. Assume it fails to meet the condition $l\_extendedprice > p\_retailprice$ in the theta join for every record in the part table. As a result, we can learn a new predicate $p_3 = l\_extendedprice > 10$, since the failure of tuple

③ with $l\_extendedprice = 10$ to join any tuple in the part table establishes all values of $p\_retailprice$ must be greater than or equal to 10. Thus all values in $l\_extendedprice$ must be greater than 10 in order to successfully join and produce an answer. Such a filter will allow the tuple ④ in the lineitem table to be eliminated since it violates p_3, which implies that it must fail the theta join operator.

Note that the predicates learned above can be refined to more selective predicates as data processing proceeds. To see this, consider the fifth tuple ⑤ in the the lineitem table. When it joins with the orders table and reaches $\bowtie_{l\_extendedprice > p\_retailprice}$, if it fails to join with any tuple in the part table, we can thus update p_3 to a more selective predicate as $l\_extendprice > 12$. Likewise, when the tuple ⑥ in the lineitem table reaches the aggregate operator with $l\_discount$ as 0.8, we can similarly update p_2 to be a more selective predicate $l\_discount > 0.8$. The more selective predicates can prune additional tuples early further to reduce query execution costs.

The example above illustrates several opportunities to learn predicates that can serve as the filters to accelerate query processing from different relational operations – from equi join (p_1), theta join (p_3), and MIN/MAX (p_2). In Section 4.3 we consider a more comprehensive set of relational operators that can help determine predicates. We note that several predicates we learn can be refined as query processing proceeds as illustrated above - e.g., the predicates learned from theta join conditions, aggregations such as MIN/MAX. Furthermore, different types of predicates can be learned from equi join conditions (e.g., range filters or membership filters), and such predicates can be implemented in different ways - as filters in memory or using an index, in which case, it could potentially reduce I/O cost of reading a relation from disk. Finally, note that the predicates learned from equi joins could potentially provide more benefit if they are learned from a more downstream join operator. For instance, consider the theta join condition $l\_extendedprice > p\_retailprice$ in Figure 4.1-b), if we modify it to be an equi join $l\_extendedprice = p\_retailprice$ and assume it uses hash join with the part

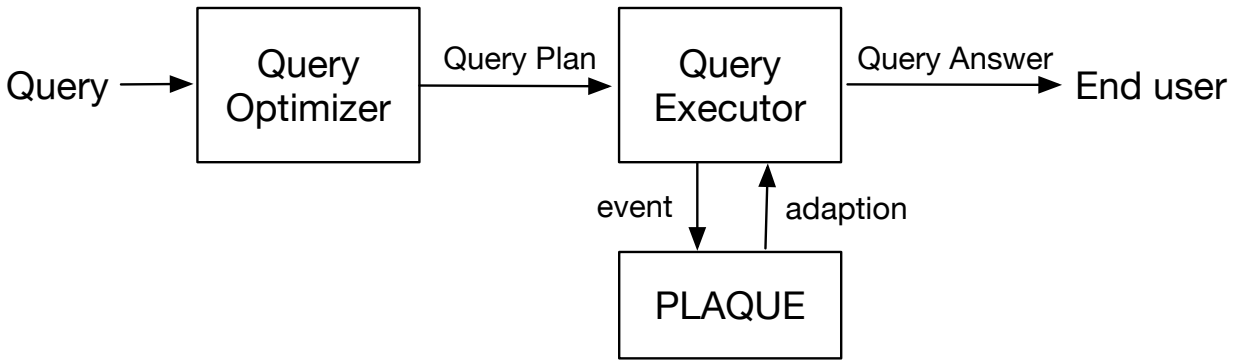


Figure 4.3: PLAQUE Architecture

table as the build table. The predicates learned from *p_retailprice* when the hash table of the part table is built can be pushed further down to the scan of the lineitem table. As a part of p in Figure 4.1-b), such a predicate would prune tuples early by using the condition from a downstream join operator.

We have highlighted several opportunities to infer predicates during query execution that can help accelerate query execution. To our best knowledge, PLAQUE is the first such comprehensive attempt to explore learning and refining predicates during query processing to prune away redundant tuples which do not result in query results. Before we discuss PLAQUE architecture, we first highlight some key challenges that arise in learning predicates that will be addressed by PLAQUE.

Challenges

Learning predicates and using them to accelerate query execution leads to several challenges. One such challenge is to devise ways to infer and refine predicates by exploiting semantics and implementing various relational operators that comprise a query. The learned predicate should be selective so that it prunes away as many records as possible. However, the predicate must simultaneously be correct in the sense that its usage does not change query results. Second, where should we insert the learned predicates in the query tree? Pushing the predicates down to the scan (leaves) of the query tree might not always be the best

option, as we will show later in Section 4.5. Third, it is critical to implement the learned predicates carefully such that applying them will not introduce significant overhead while ensuring the correctness of the learned predicates.

PLAQUE Design

PLAQUE addresses the above challenges by making careful design choices. The overview of the architecture of PLAQUE is in Figure 4.3. Given a SQL query, in PLAQUE a query optimizer first generates a query plan that is sent to the query executor. During query execution, PLAQUE will capture certain events to either learn new predicates, or update/refine predicates that have been learned previously. PLAQUE ensures that such additions and refinements of the predicates do not change the results of the original query. Learning or refining predicates in PLAQUE are implemented using ECA rules [17] based on the state of execution of the query. An ECA rule consists of three components: an event is defined as [**WHEN:** event, **IF:** condition, **THEN:** action]. For example in Figure 4.2, consider the first tuple ① in the the lineitem table, and the following event: [**WHEN**: the tuple ① reaches the aggregate operator $Agg_{max}(l\_discount)$, **IF** the tuple ① is the first tuple reaching $Agg_{max}(l\_discount)$, **THEN** a predicate $l\_discount > 0.3$ is created. Similarly, PLAQUE will detect events to learn/update new predicates from MIN/MAX, theta join, equi join, HAVING/GROUP BY conditions in Section 4.3.

Once the new predicates are learned, PLAQUE applies the learned predicates in the query executor to speed up the execution. This is achieved in PLAQUE through two subtasks:

• Deciding how to implement the learned predicates in the executor, i.e., whether or not to use index-scan. (discussed in Section 4.4)

• Deciding where to place the learned predicates in the given query plan tree. PLAQUE makes the decision based on evaluating the dependencies among different query blocks of a query tree and determining a placement strategy designed to maximize the benefit of predi-

cate placement (discussed in Section 4.5).

Finally, the executor returns the query answer to the end users or the end applications.

PLAQUE has been implemented in the Apache project VanillaDB [20, 112], which consists of several key components, such as query executor and optimizer, which were suitably modified for a reference implementation of PLAQUE. In particular, PLAQUE added the code to implement the learned predicates in VanillaDB by creating in-memory predicates or index predicates, which require minimal modifications to current DBMS codes with negligible overhead. The in-memory predicate is implemented as an in-memory checker that is directly applied to the data flow among operators during query execution to eliminate any tuple that fails the corresponding predicate, while the index predicate is implemented as index-scan to fetch tuples using an index. We discuss the two implementations in detail in Section 4.4. Furthermore, the mechanisms to add and dynamically refine predicates during query execution using ECA rules were added to the codebase. Overall, PLAQUE was implemented by adding approximately 500+ lines of code on VanillaDB. We anticipate a similar relatively modest effort extending other open-source DBMSs, such as PostgreSQL, to support learned predicate based to accelerate query execution as a part of our future work.

4.3 Predicate Creation

In this section we describe how PLAQUE learns predicates from various relational operations in a given query, including MIN/MAX aggregate, theta join, equi join, and group by/having conditions. In particular, PLAQUE aims to learn two types of predicates during query execution, i.e., *range predicate* and *membership predicate* which are of the form $[a\ op\ v]$ and $[a \in V]$ respectively, where *op* is a relational operator such as $>$, $\geq$, etc., v is a value in the

domain of attribute a, and V is a set of such values.

Predicates learned in PLAQUE that are used to modify the query do not result in a change of the final answers returned by the query. Furthermore, PLAQUE uses a **monotonic refinement** approach to modifying the learned predicates wherein a predicate p may be replaced by a predicate p' learned later if p' is more selective compared to p, i.e., $p' \rightarrow p$. For example, a predicate $a > 10$ may be replaced by $a > 20$ since the latter is more selective. PLAQUE uses such a monotonic refinement strategy to filter more tuples thereby improving performance. Monotonic refinement of the learned predicates does not jeopardize the correctness of the approach, which produces exactly the same results as that produced by the original query without the learned predicates.

4.3.1 MIN/MAX Aggregation

Consider an aggregate query with max or min conditions on attribute a, $MAX(a)$, or $MIN(a)$. Let t be a tuple and $t.a$ be the attribute value of a in tuple t. We first describe the event that causes the corresponding ECA rule (discussed in Section 4.2) to trigger the creation of a predicate learned from extremal aggregate operators. We restrict our discussion to the MAX operator. The logic for MIN is very similar and follows directly from the discussion below.

Event 1. Predicate Creation from MAX Operator.
WHEN: $MAX(a)$ operator receives a tuple t
IF: t is the first tuple $MAX(a)$ receives
THEN: a predicate p, $a > \$a$, is created, where $\$a = t.a$.

Note that $a > \$a$ satisfies the predicate correctness since none of the records with values of $a \leq \$a$ would satisfy the query answer. As an example in Figure 4.2, consider the first tuple

① in the lineitem table. A predicate $l\_discount > 0.3$ is created when the tuple ① reaches $Agg_{max}(l\_discount)$ with $l\_discount$ as 0.3. Eliminating records with $l\_discount \leq 0.3$ will not change the query results.

Once a predicate is learned from MAX aggregate operator, it may be updated later during query processing. Such a refinement is captured by the following event.

Event 2. Predicate Refinement from MAX Operator.
WHEN: $MAX(a)$ aggregate operator receives a tuple t
IF: the predicate p associated with $MAX(a)$ exists and $t.a > \$a$
THEN: update p to be $a > \$a$, where $\$a = t.a$.

The predicate refinement based on the $MAX(a)$ operator defined above is *monotonic* and hence the refinement may filter additional records since the corresponding predicate is more selective. We note that the predicates learned from a MAX operator would be the most effective if the true maximum value or a value close to it appears early in the lineitem table, which will then allow early pruning of the most tuples in the lineitem table that would fail the aggregate operator.

4.3.2 MIN/MAX with GROUP BY

Let us now consider MIN and MAX predicates in conjunction with GROUP BY operators. For now, let us assume there is no HAVING clause in the query which is addressed separately in Section 4.3.3. Let a be the attribute on which the MAX or MIN value is computed, and b is the attribute used to create groups, e.g., SELECT MAX(a), b, FROM..., WHERE..., GROUP BY b. For such a GROUP BY aggregate operator, PLAQUE adds an initial predicate p as follows at the beginning of the query processing.

Event 3. Predicate Initialization MIN/MAX GROUP BY.

WHEN: at start of query execution

THEN: Add a predicate $p = \neg(b \in \$groups)$, where $\$groups = \emptyset$.

Predicate p initially will return true for any tuple since $\$groups = \emptyset$. When a tuple t reaches the aggregation operator, the predicate p is appropriately modified by adding a new predicate p_i as a disjunct, where p_i corresponds to a predicate for the group (i.e., the b value) associated with the tuple t.

Event 4. Predicate Addition MIN/MAX GROUP BY.

WHEN: $MAX(a)$ operator receives a tuple t

IF: t is the first tuple $MAX(a)$ receives in the group whose group value $b = t.b$

THEN: create a predicate $p_i = (b = b_i) \wedge (a > \$a_i)$, where $\$a_i = t.a$. Modify the variable $\$groups$ in the predicate p associated with the aggregation to $\$groups \cup \{b_i\}$. Finally, add p_i as a disjunct to p creating a modified/extended version of p. More formally, let $p = \neg(b \in \$groups\} \vee p'$. [2] The predicate p is modified to be: $p = \neg(b \in \{\$groups \cup \{b_i\}\} \vee p' \vee ((b = b_i) \wedge (a > \$a_i))$.

Consider the modified TPCH query in Section 4.1 where the aggregate attribute $a = l\_discount$ and the group attribute is $l\_shipmode = \{$"Air", "Mail", ...$\}$. When the first tuple t reaches the aggregate operator whose $t.l\_shipmode =$"Air" and $t.l\_discount = 0.3$, the predicate p is updated to $\neg(b \in \{$"Air"$\}) \vee ((b =$"Air"$) \wedge (a > 0.3))$. At this time instance, if we were to apply the learned predicate p on a new tuple t' to check if t' can be skipped or not, and assume $t'.l\_shipmode =$ "Mail", the predicate returns true and tuple t' will pass since its group does not associate with any filtering condition.

The newly learned disjunct to the predicate p associated with the GROUP BY aggregation

[2] Note that after initialization, when $\neg(b \in \$groups)$, then p' is empty. As more disjuncts get added to the predicate p, the subsequent value of predicate p has a non-empty p' which itself contains one disjunct for each group that has been observed so far.

operator contains a filtering condition ($a > \$a_i$) which is further refined as more tuples of the same group b_i are seen as the query execution proceeds.

Event 5. Predicate Refinement from MIN/MAX GROUP BY.

WHEN: $MAX(a)$ operator receives a tuple t

IF: t is in group b_i where $b_i \in \$groups$, and $t.a > \$a_i$

THEN: update p to $p = \neg(b \in \$groups) \vee p' \vee ((b = b_i) \wedge (a > \$a_i)$, where $\$a_i = t.a$.

When a new tuple t reaches the aggregate operator whose $t.l\_shipmode =$ "Air" and $t.l\_discount = 0.8$, the predicate p is refined to $\neg(b \in \{$ "Air" $\}) \vee ((b =$ "Air" $) \wedge (a > 0.8))$. For each group b_i, $\$a_i$ is the maximum value in this group observed so far during execution. In Section 4.4, we will detail how to implement such a disjunction of predicates.

We note that the above strategy of maintaining a predicate for each group to filter tuples may introduce non-trivial storage and processing overhead when the number of groups is large. PLAQUE uses several optimizations to reduce such overhead. To reduce the overhead of maintaining and checking a disjunction for each group, PLAQUE maintains predicates for a small set of k groups. We choose the k groups for which to maintain predicates based on estimating the size of different groups by a bootstrapping approach that processes an initial sample of records without any predicates. From the sample, we determine the top-k largest groups and then subsequently learn filters on the chosen b_i values for those top-k groups. The intuition behind the choice is that predicate-based filtering will be the most effective on such groups with larger size. We can further reduce the overhead of checking if a value of b in a tuple has/has not been previously observed (i.e., $\neg(b \in \{\$groups)$ by maintaining $\$groups$ as a bloom filter. Note that the false positives in a bloom filter do not jeopardize the correctness - it only implies that PLAQUE will not be able to form a predicate on b_i if the bloom filter indicates that b_i is already in $\$groups$ as a false positive.

4.3.3 Conditions in HAVING Clause

Consider a query with having condition, **SELECT** Agg(a), b **FROM** $R_1, \ldots, R_n$ **WHERE** ... **GROUP BY** b **HAVING** Agg(a) *op* v, where a is the aggregate attribute and b is the group attribute. *op* is one of $> \mid \geq \mid < \mid \leq \mid =$, v is a value, and $Agg = max \mid min \mid sum \mid count$.

During the query execution, the aggregate operator maintains the aggregated value $Agg(a)$, such as $SUM(a)$, for each group. $Agg(a)$ will be updated when any new tuple reaches the aggregate operator.

Consider the scenario where *Agg* is *count*, and HAVING condition is *count(a)* $<$ *100*. If the HAVING condition becomes false, i.e., *count(a)* $\geq$ *100*, it will *always* remain false during later query execution for that group. On the other hand, for the HAVING condition *count(a)* $>$ *100*, once it becomes true, it will always remain true in the future when more tuples are processed. We capture such a concept by defining *in-preserving* and *out-preserving* properties for the condition in the HAVING clause. Subsequently, we describe how to learn predicates that can be used to filter tuples based on the conditions in the HAVING clause.

Definition 4.1 (In/Out-Preserving Property of Having Condition)**.** A condition $H = [Agg(a)$ *op Constant*] in the HAVING clause is *in-preserving*, if H becomes *true* at any instance t during query execution based on partially observed tuples, H always remains *true* at any instance t' where $t' > t$, when more tuples have been observed. On the other hand, H is *out-preserving*, if H is *false* at an instance t during query execution, it remains *false* at any future instance t' where $t' > t$ when more data has been observed. □

Given the above concepts of In/Out preserving conditions, we can now define the event to create the corresponding predicate.

Event 6. Predicate Learned From Having.

WHEN: $Agg(a)$ in a HAVING condition is updated for the group with group value b_i

IF: the HAVING condition is *out-preserving*, and $Agg(a)$ fails the condition

THEN: a membership predicate $p_i = \neg\{b_i\}$ is created.

Whenever an out-preserving having condition becomes false during query execution in the group whose group value is b_i, PLAQUE learns the predicate p_i to skip all later tuples in the same group. In particular, for any tuple t, if $t.b = b_i$, t fails the predicate p_i and it will be skipped. Note that the In/Out-preserving property of HAVING containing *MIN*, *MAX* or *COUNT* aggregation can be decided together with *op* in advance of the query execution. For instance, $max(a) > 100$ is in-preserving, and $max(a) < 100$ is out-preserving. As for the *sum* aggregate operation, if the data statistics of attribute a are known in advance that all values in a are non-negative, then the out-preserving property of sum can also be determined prior to the query execution. For instance, $sum(a) < 100$ is out-preserving if $\forall v \in Vals(a), v \geq 0$.

4.3.4 Learning from Theta Join

We show how to learn predicates from theta join conditions in the given query during the query execution. Let R be a relation. Consider a theta join condition between relations R_1 and R_2, $R_1 \bowtie_{a\, op\, b} R_2$, where a and b are two attributes, and $op := > \mid \geq \mid < \mid \leq$. To better illustrate the idea, without loss of generality, let us assume *op* is $>$, i.e., the theta join condition is $a > b$. We first define the event to trigger the creation of predicates learned from a theta join operator.

Event 7. Predicate Creation From Theta Join Operator.

WHEN: tuple $t \in R_1$ arrives at a theta join operator, $R_1 \bowtie_{a>b} R_2$

IF: t is the first tuple that *fails to join with any tuples* in R_2, [3]

THEN: a predicate p, $a > \$a$, is created, where $\$a = t.a$

[3]The above observation can be easily captured during query execution since the join output for t in the current theta join operator will be empty (NULL).

A tuple $t \in R_1$ failing to join with any tuple in R_2 implies that for any tuple $t^{'} \in R_2$ that comes to this theta join operator, the attribute value of $t^{'}.b$ must be *greater than or equal* to $t.a$, i.e., $b \geq t.a$. Since $a > b$, this naturally implies $a > t.a$. Consider the tuple ③ in the lineitem table in Figure 4.2. As shown in Section 4.2, the theta join $\bowtie_{l\_extendedprice > p\_retailprice}$ learns a predicate $p\_retailprice \geq 10$ when the tuple ③ of the lineitem table fails to join any tuple in the part table.

Once the predicate $p = a > \$a$ is learned, it could be updated during later query execution when $\$a$ is updated to a larger value. In particular, we define the event of predicate refinement from a theta join operator below.

Event 8. Predicate Refinement From Theta Join Operator.

WHEN: a tuple $t \in R_1$ arrives at a theta join operator, $R_1 \bowtie_{a > b} R_2$

IF: $p = a > \$a$, t fails to join with any tuples in R_2, and $t.a > \$a$

THEN: the predicate p is updated to be, $a > \$a$, where $\$a = t.a$.

The predicate refinement discussed above is monotonic. The operand $\$a$ in predicate $a > \$a$ is the maximum value of attribute a in the tuple from R_1 that failed the join test in the theta join operator so far. So the failure of a larger a value to join any tuple in the theta join can be used to refine the predicate to a more selective predicate while ensuring the correctness of the execution. This was illustrated in Figure 4.2 by refining the predicate from $l\_extendedprice > 10$ to $l\_extendedprice > 12$ when processing the tuple ⑤ of the lineitem which also failed to join with any tuples in the part table in theta join operator $\bowtie_{l\_extendedprice > p\_retailprice}$.

Likewise, when the *op* in a theta join condition is $\geq$, we follow Event 7 and Event 8 to learn exactly the same predicate as the one when *op* is $>$. In contrast, when the *op* is $<$ or $\leq$, the learned predicate is $a < \$a$, where $\$a$ is the minimum value of the attribute a in the tuple from R_1 that failed the join test in the theta join operator so far.

Theta join	$R_1 \ltimes_{a\ op\ b} R_2$			
Outer relation	R_1		R_2	
op	$>$ or $\geq$	$<$ or $\leq$	$>$ or $\geq$	$<$ or $\leq$
Predicates	$a > max_a$	$a < min_a$	$b < min_b$	$b > max_b$

Table 4.1: Learned Predicates for Theta Join

Symmetrically, for a theta join $R_1 \ltimes_{a>b} R_2$, if there is a tuple coming from the right side of the join, i.e., R_2 and it fails the join test, we create a predicate $b < \$b$, where $\$b$ is the minimum value of $R_2.b$ in the tuples from R_2 that fail the join test in this theta join operator so far during query execution.

In a nested loop implementation of a theta join $R_1 \ltimes_{a>b} R_2$, if a tuple rises from R_1 and the join algorithm checks the entire R_2 relation to perform the join, we refer to R_1 as the outer relation and R_2 as the inner relation. Table 4.1 summarizes all the predicates that can be learned from a theta join condition based on the *op* and on which side the tuple t ascends into the join. In Table 4.1, we denote max_a and min_a by the maximum and minimum values in the attribute a that fail the join test so far during query execution. It can easily be shown that the process of replacing predicates added to the query tree earlier by the stronger predicates discovered later in the execution is monotonic, thus ensuring the correctness of the execution.

Analysis: The effectiveness of the predicates learned from theta joins in accelerating query processing depends on the implementations of the joins. The most commonly used theta join implementations can be categorized as (block-based) nested loop join, index nested loop join or sort merge joins with a few variants, such as ripple join [57], that performs join $R \bowtie S$ by sampling tuples from both relations simultaneously. Consider a theta join operator $R_1 \ltimes_{a>b} R_2$ and let the join be implemented using nested loop join or index nested loop join. Without loss of generality, assume $R_1.a$ is the outer relation where each tuple t in R_1 reaches the operator, and then the operator checks if t matches any tuples in the inner relation R_2. [4]

[4]For index join, the inner relation corresponds to the one that performs the index scan for each tuple coming from the outer relation. Ripple join switches the inner and outer relations during join execution.

The learned predicates from such a join operator are expected to provide improvement when (block-based) nested loop join, index join and ripple join are used and the values in $R_1.a$ reaching the theta join operator is *not sorted*. The advantage of the predicate would not benefit a sort merge join implementation since tuples are processed in the sorted order and all the remaining tuples yet to be processed satisfy the learned predicate and would not be pruned further.

4.3.5 Learning from Equi Join

Equi join is one of the most commonly used SQL operators. We start with identifying several opportunities to learn predicates from an equi join.

To identify the opportunities to learn predicates from an equi join, we first need to define the concept of a *convergence point*.

Definition 4.2 (Convergence Point). Let R be a relation that participates in a join in a query Q. A convergence point for R with respect to the join operator is defined to be the earliest point in the query execution when all the possible tuples of R that could possibly participate in the join have passed through the join operator at least once. □

The convergence point of a relation participating in a join depends upon the join algorithm used. For instance, in the case of a hash join, the convergence point of the build relation occurs when the corresponding hash table has been built. In the query plan tree in Figure 4.1, the convergence point of the orders table is reached after the hash table of the orders table has been built. Similarly, the convergence point of the part table is reached when the first outer loop of the lineitem table is complete in the join operator $\bowtie_{l\_extendedprice>p\_retailprice}$ when a nested loop join is used. At the convergence point for a relation in a join, all possible values of the relation that could participate in the join have already been observed and this

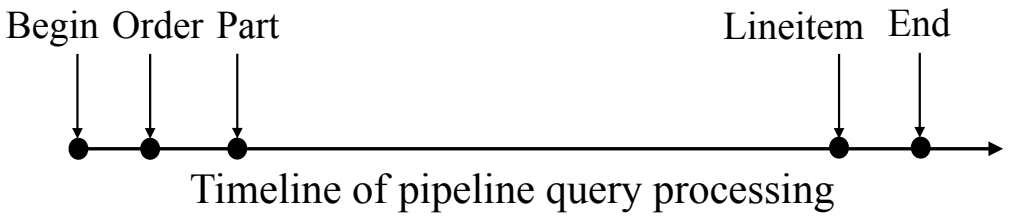


Figure 4.4: Convergence Points of Relations.

information can be exploited in learning the appropriate predicates from equi joins. Note that different relations reach convergence points at different time instances, based on the join implementations. For instance, in Figure 4.4 the convergence point for the lineitem table occurs close to the end of query execution since the lineitem table is the probe table and we have to consume all tuples from the lineitem table to complete the query. In general, for a one-pass hash join or nested loop join, their build table or inner table will potentially reach the convergence points early during query execution when the build phase is complete or the first outer loop is complete. For multi-pass hash join, such as a grace hash join, and sort merge join, both relations will reach their convergence points when the scan or sort for both relations is complete.

We can learn either membership or range predicates from equi joins at the convergence point of the participating relations.

Event 9. Predicate Creation from Equi Join.

WHEN: R_1 reaches its convergence point

THEN: define a predicate p on relation R_2 on the join column $R_1.a$. p is either a set of range predicates that cover the attribute values in $R_1.a$, or p is a membership predicate $a \in Vals(R_1.a)$, where $Vals(R_1.a)$ consists of all values in $R_1.a$.

We next discuss how membership/range predicates are learned.

Learning Membership Predicate from Equi Join: When a relation R reaches its convergence point early during query execution, we can learn a membership predicate p_m from join attribute in R. We adopt the choice of bloom filters to implement membership predicates as

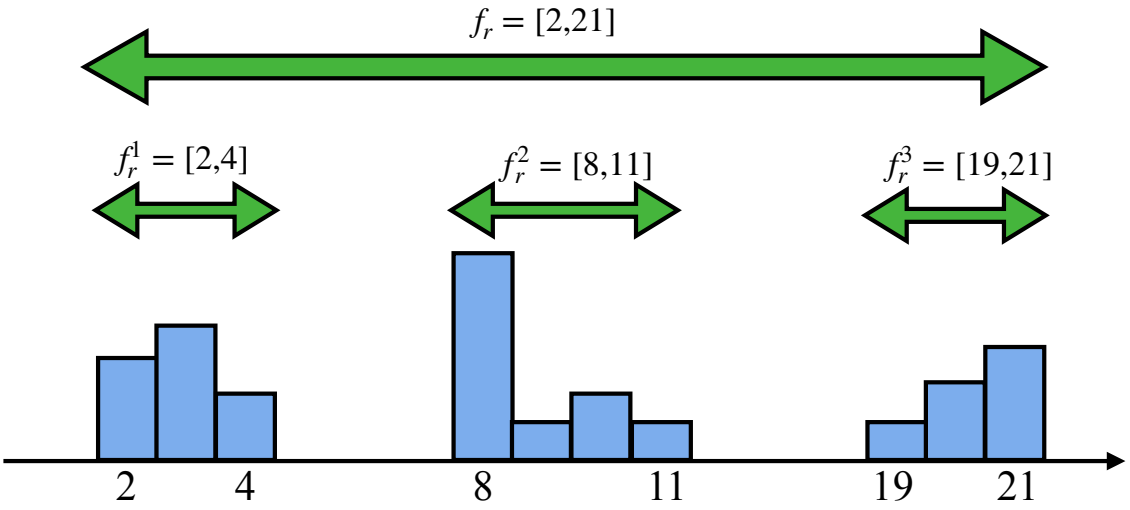


Figure 4.5: Range Predicates Learned From Equi Join.

in the previous work [93, 121, 62] to enable more efficient filtering due to the succinct nature of the bloom filters.

Membership predicates can save the computational cost of computing joins by filtering records that will not join with records in the other relation. Such functions, however, do not save the I/O cost of reading tuples from disk. For such a benefit, we instead can learn index-based range predicates. Below we describe the range predicate learning strategy used in PLAQUE that brings about 3x I/O savings compared with the membership filter as shown in Section 5.6. We will show how to implement index scan using the range predicates learned from equi join conditions in Section 4.4.

Learning Range Predicate from Equi Join: Consider an equi join operator $\bowtie_{R_1.a=R_2.b}$, where a and b are join attributes in R_1 and R_2. Assume that R_1 reaches its convergence point early during query execution (e.g., R_1 is the build table), and thus all values of $R_1.a$ are known early at the convergence point. Our goal is to learn a set of range predicates from the values in $R_1.a$ that can be pushed down to the other relation R_2 relation (e.g., the probe side) in the query plan tree to help filter away tuples using an index on R_2, thus saving I/O costs. Figure 4.5 shows the histogram of values in $R_1.a$ (blue buckets) [5] where the size of

[5]Assume the value type of $R_1.a$ is an integer, which can easily be relaxed to double or float.

the bucket is the unit size, i.e., 1. Let $P_r = \{p_r^i\}$ be a set of range predicates. Our approach to learning range predicates does not explicitly construct the histogram for $R_1.a$, and we use the histogram in Figure 4.5 for better illustration in determining the range predicate. To learn P_r from a set of values in $R_1.a$ in the equi join condition $\bowtie_{R_1.a=R_2.b}$, several factors are considered.

- *Completeness*: the learned range predicates P_r should not introduce false negatives, i.e., P_r should contain all values of $R_1.a$. Otherwise using P_r on the probe side will incorrectly filter potentially correct values in the query answer.
- *Effectiveness*: P_r should not result in a large number of false positives. One possible learned predicate for $R_1.a$ in Figure 4.5 is $p_r = \{[2, 21]\}$, which has zero false negatives. However, p_r is not effective since it has large false positives. (e.g., $[5, 7], [12, 18]$) Instead, $P_r = \{[2, 4], [8, 11], [19, 21]\}$ may be a better set of predicates learned from $R_1.a$ since it does not introduce any false positives nor false negatives.
- *Complexity*: the number of predicates in P_r should be constrained. If we simply learn the unit predicates by creating one predicate for one unit value, such as $[2, 2], [3, 3], ...[21, 21]$, P_r downgrades to a membership-like predicate but using a less efficient implementation. In this case, $|P_r|$ will be equal to the number of distinct values in the column $R_1.a$, which increases the complexity of predicate implementation as we will show in Section 4.4.

The predicate p_r we learn from an equi join condition on an attribute a has the format $[l, u]$, where l and u represent the lower and the upper values in attribute a. [6] Let $l(p_r)$ be the number of domain values covered by the predicate, where $l(p_r) = u - l + 1$. For a value v, we denote by $v \in p_r$ if v is in the interval of p_r. Formally, we define the Range Predicate Learning (RPL) problem as follows.

Definition 4.3 (Range Predicate Learning)**.** Given a set of values V in the join attribute and k, the maximum number of range predicates to learn, RPL aims to find a set of range

[6] $a \in [l, u]$ is equivalent to $a \geq l \wedge a \leq u$.

Algorithm 3: k-Max-Gap

Input: V, k
Output: a set of range predicates, P_r

```
1  Sort V in the ascending order
2  Initialize a max-heap, h ← ∅
3  for v_i ∈ V do
4      d_i = v_{i+1} − v_i
5      g_i = (v_i, v_{i+1})
6      Construct key-value pair kv = [d_i → g_i]
7      Insert kv in h
8  G = {g_i} ← h_{top−(k−1)} // Return the set of gaps with top(k-1) largest
       distance
9  P_r ← DropGap(V, G)
10 return P_r
```

predicates $P_r = \{p_r^i\}$, such that,

$$(RPL) \min \sum_{p_r^i \in P_r} l(p_r^i) \tag{4.1}$$

$$s.t. \quad |P_r| \leq k \tag{4.2}$$

$$\forall v \in V, v \in P_r. \tag{4.3}$$

The condition 4.2 in the RPL problem guarantees that the number of learned range predicates is at most k (i.e., complexity), and the condition 4.3 makes sure the learned predicates will contain all attribute values and thus no false negatives (i.e., completeness). By minimizing the total length of range predicates, we are able to maximize the effectiveness of the predicates since a lower number of false positives will be introduced by more concise predicates.

The RPL problem is NP-hard which can be proven using a reduction from the Size-constrained weighted set cover problem [51], a generalization of the weighted set cover problem. In particular, each value $v \in V$ is an object, and each predicate $p_r \in P_r$ is a set with weight as $l(p_r)$. Size-constrained weighted set cover problem seeks to select at most k sets to cover all objects and the sum of the weights of selected sets is minimized. Correspondingly, RPL aims to choose at most k predicates (efficiency) to cover all values (completeness) with minimum

total predicate length (effectiveness).

We below present an efficient greedy solution to RPL, k-Max-Gap, in Algorithm 3. Given a set of values V in the join attribute in an equi join operator and k, the maximum number of range predicates to learn, Algorithm 3 returns at most k range predicates. Let gap g_i be the distance between two consecutive values v_i and v_{i+1}, where $v_i \in V, v_{i+1} \in V$. k-Max-Gap seeks to find the top-(k-1) gaps with the largest distances in V, and drop these gaps from V to generate k range predicates. In particular, we sort all values in V and initialize a max heap h. (Line 1-2) We compute the distance d_i for every consecutive value pair (v_i, v_{i+1}), which are inserted into a max heap. (Line 3-6) We retrieve top-(k-1) gaps G with the largest distance d_i from the max heap. (Line 7) Finally, we generate P_r by dropping the set of gaps G from V.

Consider the example in Figure 4.5. Assume $k = 3$, i.e., $|P_r| \leq 3$, meaning that we try to find at most 3 range predicates. We first sort the values in the join key, and find the top-2 largest gaps between two consecutive values in the join key, and they are $(11, 19)$ and $(4, 8)$. Dropping these two gaps from the value interval of the join key leads to three predicates, $[2, 4]$, $[8, 11]$ and $[19, 21]$.

4.3.6 Sideway Information Passing

In the previous sections, we specify how PLAQUE learns predicates from different relational operators. In addition, PLAQUE uses a sideway information passing (SIP) approach to learn new predicates based on the predicates learned from relational operators when queries involve joins (equi joins or theta joins).

SIP via Equi Join. Consider an equi join $\bowtie_{R_1.a=R_2.b}$. Without loss of generality, assume we have learned a predicate p_1 which is applicable in the join column $R_1.a$, e.g., $p_1 = R_1.a >$

10, PLAQUE learns a new predicate p_2 in $R_2.b$ by passing p_1 via equi join condition, i.e., $p_2 = R_2.b > 10$.

SIP via Theta join. Consider a theta join $\bowtie_{R_1.a\ op\ R_2.b}$, where $op := > \mid \geq \mid < \mid \leq \mid \neq$. Without loss of generality, assume we have learned a predicate p_1 which is applicable in the join column $R_1.a$. If p_1 is a membership predicate, i.e., $R_1.a \in Vals(R_1.a)$, and op is the operator $\neq$, then PLAQUE learns a new predicate p_2 on $R_2.b$ where p_2 is $R_2.b \notin Vals(R_1.a)$. Alternatively, if op is $>$, then PLAQUE can learn a predicate $R_2.b <= x$, where x is the maximum of the elements in $Vals(R_1.a)$. Similar predicates can be learned for other instantiation of the operator, e.g., if op is $<$, then we can learn the predicate $R_2.b >= x$, where x is the minimum value in $Vals(R_1.a)$.

Likewise, PLAQUE learns appropriate predicates on $R_2.b$ values based on a set of range predicates learned over $R_1.a$. Consider a predicate p_1 consisting of a set of ranges: $p_1 = p_{r1} \vee p_{r2} \vee ... \vee p_{rn}$, where $p_{ri} = [l_i, u_i]$ and $u_i < l_{i+1}$ learned over $R_1.a$. Based on the operator op in the theta join $\bowtie_{R_1.a\ op\ R_2.b}$, PLAQUE learns predicates on R_2 as follows. If op is $>$, then the predicate learned on $R_2.b$ corresponds to $R_2.b \leq x$, where $x = u_n$, and u_n is the largest value in the range predicates covering $R_1.a$ values. Likewise, if op is $<$, we add a predicate $R_2.b \geq x$, where $x = l_1$, where l_1 is the smallest value in the range predicates covering $R_1.a$. Note that in both the above cases, if l_1 or u_n are not bounded, we do not learn any predicate on R_2. For instance, if the first range predicate on $R_1.a$ corresponded to $R_1.a \leq 5$, then its range is $(-\infty, 5]$. Thus, in such a situation, since l_1 is not bounded, no predicate on R_2 will result from the above join condition.

Above, we have specified a few cases of how SIP predicates are learned in the case of theta joins. The comprehensive set of learned predicates depends upon the set of operators in the theta join, but the essential logic is similar to the one highlighted above.

4.4 Predicate Implementation

In this section, we discuss how PLAQUE implements the learned predicates as either an *in-memory predicate* or as an *index predicate*.

In-memory Predicate. The in-memory predicates can be either a range predicate, a membership predicate, or a disjunction of range predicates as in the MIN/MAX with GROUP BY (see Section 4.3.2).

A membership predicate is implemented by converting the value set to a bloom filter. Note that the bloom filter will not have false negatives but may introduce false positives. Such a false positive may result in tuples going through but such tuples will be eliminated by the downstream operators, and thus will not generate wrong answers. In-memory range predicates are simply implemented as range conditions. The disjunction of range predicates learned from MIN/MAX with GROUP BY is converted into a map, where the key is the group value and the value corresponds to the filtering condition in the corresponding group.

Index Predicate. In-memory predicates are easy to implement and can be placed everywhere in the query tree. While they offer great flexibility and are able to eliminate tuples early during query execution, they do not help reduce the I/O cost of query execution. The alternate implementation of the learned predicates using an index can additionally offer I/O saving. The index-based implementation of the learned predicates is, however, more complex since refining predicates dynamically during query execution with more selective predicates becomes more complex when using an index-based implementation. Consider the case with the predicates learned from MIN/MAX aggregate operators and theta join operator in Section 4.3 for instance. Furthermore, shifting the original scan in the given query plan tree to an index scan of learned predicates at query execution time, if not carefully implemented, will

generate duplicated query answers, as will be clear shortly. The index-based implementation of predicates needs to be implemented carefully only when it will bring obvious performance improvement.

We thus consider implementing the index-based predicate p when the following conditions are satisfied:

- The index of the attribute that a learned predicate p operates on already exists in the database.
- p is able to be pushed down to just above the scan of relation R that p is applicable in the query plan tree.
- The original scan of R is not an index scan. Otherwise, the benefit from p using index scan would be diminished, and implementing index scan using more than one predicate adds high complexity to the executor, and thus it is not worthy.
- p is a range predicate instead of a membership predicate.

We begin with a short bootstrapping phase to estimate the selectivity of a learned predicate p, i.e., the percentage of the tuples satisfying p over all the sampled tuples so far during the bootstrapping phase. During this stage, p is implemented as an in-memory predicate. If the selectivity of p is lower than a predefined threshold [7] (i.e., p is selective), we shift p from an in-memory predicate to an index predicate. Let T be the timestamp when the index predicate is built and operated, and $Tuples$ be a set of tuples in R that have been already processed during query execution before T. An index scan p on R typically retrieves all tuples satisfying p, which might contain a subset of $Tuples$, leading to potential duplication of query answers. PLAQUE uses a low-overhead strategy to prevent duplication. For the table scan on relation R, the rows are accessed in the increasing order of the record id (RID) for efficient sequential I/O. Let cur_RID be the RID of the row in R at the time T when

[7] The choice of index predicate also depends on the size of relation that the predicate is applied on. The larger the size, the more improvement from the index predicate with a reasonable selectivity is expected.

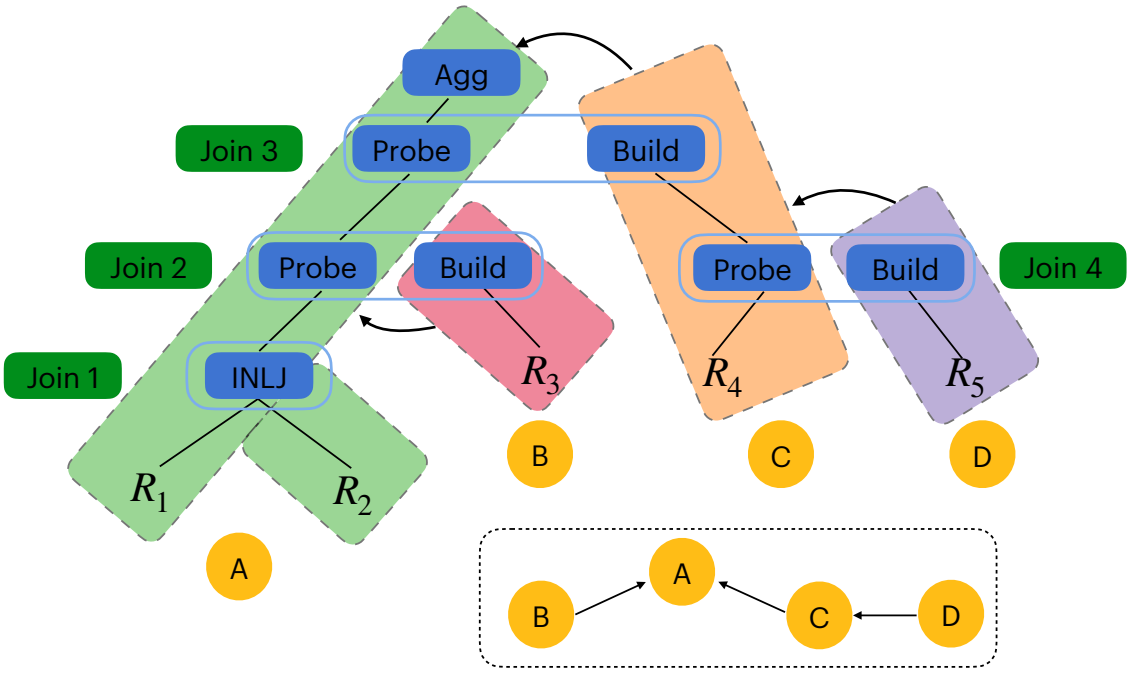


Figure 4.6: Partial Order Execution Graph.

the index predicate p is built. We add a predicate $RID > cur\_RID$ implemented as an in-memory predicate in p immediately to prevent the duplication of the already processed rows whose RID is smaller than or equal to cur_RID.

4.5 Predicate Placement

We next discuss the strategy used in PLAQUE to place the learned predicates during query execution in a given query tree so as to maximize its benefit of filtering away spurious tuples. Predicate placement in the traditional context before query execution is relatively straightforward. The query optimizers typically push down predicates as far down the query tree and as close to the relational scan as possible. Interestingly, when predicates are learned mid-flight during query execution, their placement as far down the query tree as possible might *not* be a good strategy. For example, consider a scenario where PLAQUE learns a new predicate $R.a > val$ at a given stage during execution. Assume that the relation R was part of a join condition and was designated as a build table in a hash-join. Pushing the predicate

below the hash function in such a case, if the build process has already occurred by the time the predicate is learned, would not help since the hash table based on R is already built. In contrast, placing the predicate perhaps above the hash-join to reduce the number of tuples that reach downstream operators could still be very useful in accelerating query execution. In general, one has to be careful on where and how to place operators in the query tree, when predicates are not known apriori and are learned during query execution. Our goal in this section is to develop a strategy that maximizes the impact of the predicate by placing it at an appropriate location in the query tree.

In a pipeline query plan, a query is often executed in several blocks based on the specific implementations in relational operators, where all operations in one block are pipelined. Consider a four-way join aggregate query plan tree in Figure 4.6, where R_1 and R_2 are joined using Index Nested Loop Join (INLJ), and all the other joins are hash joins. For each hash join, the probe phase is executed after the build phase is complete, leading to naturally two blocks of execution, i.e., build and probe.

Figure 4.6 shows four execution blocks in the query tree, represented by nodes A, B, C, D with different colors. Let b_i be a block where all the operations can be executed using *pipelining*. In Figure 4.6, node A is one block where the INLJ, two probing operations, and aggregate operation can be pipelined together, while the build operation in Join 2 is in one individual block, i.e., node B in Figure 4.6. Similarly, the probe phase of Join 4 can be pipelined with the build phase of Join 3 in node C, while the build of R_5 in Join 4 is an individual block.

We formulate the query execution on a given query plan tree as a partial order graph to capture the order of block executions.

Definition 4.4 (Join Graph)**.** Let $G = (\mathcal{V}, \mathcal{E})$ be a directed graph, where each $b_i \in \mathcal{V}$ represents a block and $e_{ij} \in \mathcal{E}$ denoting that the execution of block b_j must be executed after

Algorithm 4: Predicate Insertion

Input: $p, G = (\mathcal{V}, \mathcal{E})$

1 $b :=$ node where p is created
2 $Des_G(b) :=$ the set of descendants nodes of b in G
3 **for** $b_i \in Des_G(b) \cup \{b\}$ **do**
4 $\quad pushdown(p, b_i)$

b_i is complete. $\square$

For example, the join graph in Figure 4.6 has four nodes, A, B, C, and D. The edge from B to A denotes that all the operations in A can be executed only when the build table of R_3 is complete.

Formulating query execution as a partial order join graph is helpful in identifying where to place the learned predicates. On one hand, we wish to push the learned predicates as low as possible in the query tree to maximize their benefits to potentially skip more rows early during query execution. On the other hand, it is not beneficial to place the predicates in a block whose execution has already been completed before the time when the predicates are learned. The partial order join graph provides a way forward in determining where to place the predicates as shown in the example below.

Assume a range predicate p_r is learned from the max operator (e.g., Agg in Figure 4.6), and it is applicable in R_1, then the best location to place p_r is just above the scan of R_1, since doing so will prune tuples earliest. However, if p_r is applicable to R_3, inserting p_r above the scan of R_3 will not help remove tuples since the build phase of R_3 is complete before the p_r is learned in the aggregate operator. Instead, the best location for p_r is R_1 if R_3 is joined with R_1. Intuitively, we wish to push the learned predicate as deep as possible in the query plan tree, while ensuring the predicate will effectively prune tuples.

We formally describe the algorithm to place any learned predicate in the query plan tree in Algorithm 4. Given a learned predicate p and the partial order execution graph $G = (\mathcal{V}, \mathcal{E})$, we first identify the node $b \in \mathcal{V}$ where p is created. (Line 1) Second, we identify the set of

descendant nodes of b in graph G, $Des_G(b)$, i.e., the set of nodes that are reachable from b. (Line 3-4) Finally, in block b and each block in $Des_G(b)$, we push down the predicate p into each such block if p is applicable in the corresponding relations in the block.

We below use an example to illustrate the Algorithm 4. In Figure 4.6, assume a predicate p is learned in node D (i.e., after the hash table of R_5 is complete). Obviously, p cannot be pushed down further in D. Consider the set of descendants of D, i.e., $\{C, B\}$. p can be pushed down in C to the probe of R_4, and it can also be immediately pushed down above the scan of the applicable base relation in node A via Join 3, for instance, if Join 3 is $R_1 \bowtie R_4$, then p can be pushed down above the scan of R_1. Similarly, a predicate p learned in the node C after the build table of R_4 is complete can be passed through Join 3 and pushed down to the node A. Note that p learned in C will not help eliminate tuples in its ancestor nodes in the graph G, such as D, since p is learned after the block D is fully executed. When the predicate p is learned from Agg in the node A (e.g., max or min operator), the only node we can push p down is A since A does not have any child nodes in the graph. In particular, p can be pushed down appropriately into the different execution points in A based on the applicable attribute of p. For instance, if p is applicable in either R_1 or R_2, then p can be pushed down to the scan of R_1 or R_2. If p is applicable in R_3, then p will be placed in probe phase in Join 2 in the node A, which is not the deepest location in A, but will help eliminate tuples early for downstream operations in A, such as the probe phase in Join 3 and aggregation.

4.6 A Pre-Learning Approach

We present a *complementary* approach to learning predicates *prior to the query execution* which we refer to as, *PreL*. With the addition of PreL, the modified data flow in PLAQUE is shown in Figure 4.7. Note that the predicates learned by PreL for a query Q are appended

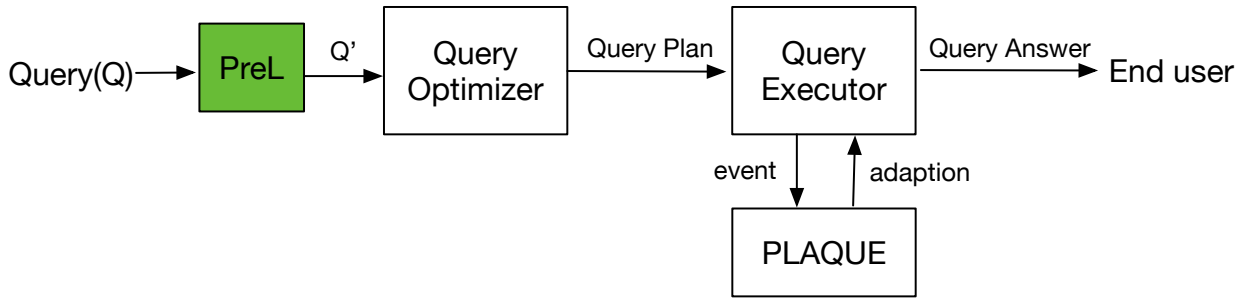


Figure 4.7: PLAQUE Architecture.

to Q. The modified query Q' is optimized to empower the optimizer to exploit the newly learned predicates. For instance, the optimizer may push down the predicate learned by PreL during optimization.

The key idea of the pre-learning approach before the query run is *walking through a set of possibly small relations without finishing the query run* to learn predicates fast. Consider the query in Section 4.1. Can we simply scan the part table applied by p_brand = ":10" or the orders table applied by o_orderdate < "1993-01-01" to learn useful predicates based on the join column values prior to query execution? To this end, we propose a strategy by modifying the nested-loop-join algorithm.

The pre-learning approach is executed in the following steps. Given a query plan tree T, let o be the *lowest* operator above any *join* operator in plan tree T, [8] such as the projection operator $\Pi_{R_1.a}$ in Figure 4.8. We simply modify this operator o to o' such that it will terminate after it receives the *first* tuple. Second, we force the optimizer to use a *nested loop join* for every join operator $\bowtie$ and constraint the shape of the tree to be left deep tree [9]. For each join operator $\bowtie$, we slightly modify it to be $\bowtie'$ by adding one condition: when $R_1 \bowtie' R_2$ receives the *first* tuple t from its outer relation R_1, $\bowtie'$ forces to iterate all tuples in its inner relation R_2, *union* t with a random tuple in R_2, and pushed the composite tuple above. [10] The goal of the pre-learning approach is not to generate correct query answers, but to walk

[8] If there is no join in the query, such as a selection-projection query, it is often trivial and fast to run without the need of predicate optimization.

[9] This can be achieved by adding hints in the database, such as PostgreSQL.

[10] In our implementation, we simply choose the first tuple in R_2 to be unioned with t.

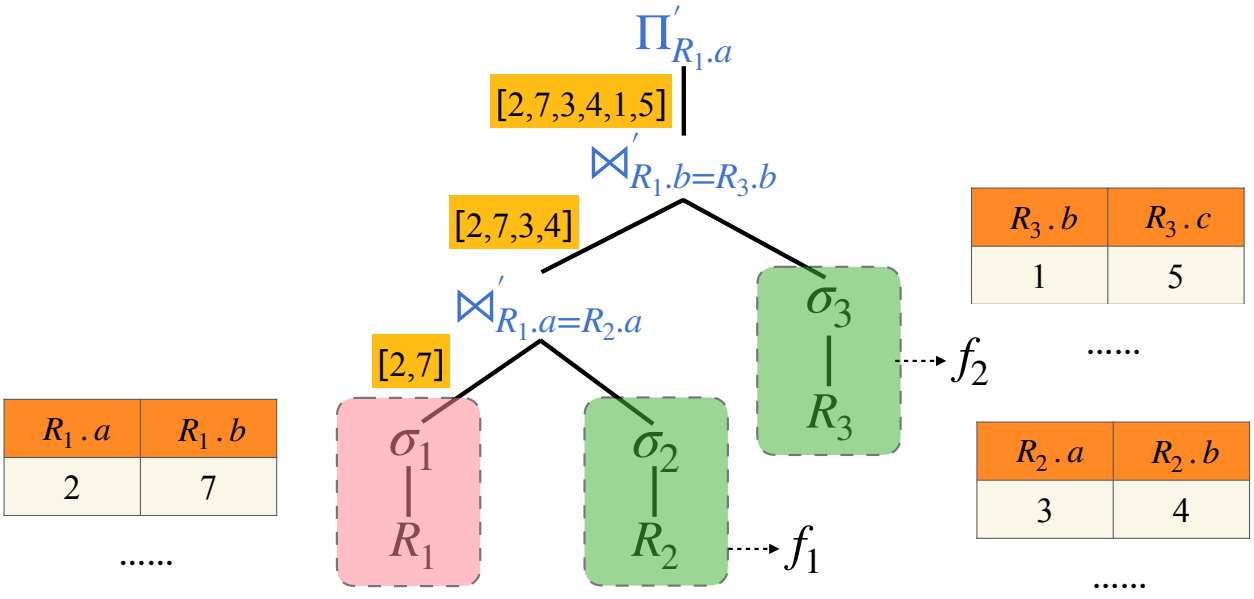


Figure 4.8: Pre-Learning Approach.

through the inner relations as much as possible in join operators early, such that we can learn predicates from the scanned inner relations. Note that the output of the pre-learning approach PreL is a set of learned predicates, instead of the query answers from the modified nested loop join, and thus PreL is correct since the approach to learning predicates from join conditions as used in Section 4.3 is correct.

We below use an example in Figure 4.8 to illustrate PreL. Consider the query plan tree in Figure 4.8, where every join operator uses a nested loop join, and R_1 is the driving (outer) relation, R_2, R_3 are the inner relations. When the first tuple $[2, 7]$ from R_1 arrives at the join operator $\bowtie'_{R_1.a=R_2.a}$. The inner relation R_2 will be scanned one pass. Instead of finding any matched tuples in the inner relation as a standard nested-loop-join does, we modify the action to let the join operator pick a random tuple (e.g., $[3, 4]$ in R_2) from the inner relation and push the composite tuple (e.g., $[2, 7, 3, 4]$) above. Similarly, when the composite tuple arrives at $\bowtie'_{R_1.b=R_3.b}$, the inner relation R_3 is scanned and the composite tuple $[2, 7, 3, 4, 1, 5]$ is sent above. The query execution terminates when the projection operator $\Pi'_{R_1.a}$ receives the first tuple $[2, 7, 3, 4, 1, 5]$. During this process, R_2 and R_3 are scanned exactly one pass and there exists only one composite tuple flowing during query execution.

PreL is a *one pass* algorithm, and all the inner relations will be scanned only one time, leading to efficient query execution. Note that the size of the inner relations is often smaller than that of the driving relations since we force the optimizer to choose a nested loop join plan. This observation often leads to fast execution of PreL in practice.

During the execution of PreL, we are able to learn a set of predicates P fast on their corresponding join attributes from all the inner relations (as colored in the green box in Figure 4.8). For each join condition, if it is an equi join, a set of range predicates can be learned given the available values in the join attribute in the inner relation using the predicate learning approaches described in Section 4.3. For a theta join, we can easily get the maximum max_a or minimum value min_a of the join attribute a in the inner relation after the corresponding selection predicates are applied. For example, consider the max_a and min_a computed after σ_2 as shown as the lowest green box in Figure 4.8. max_a and min_a are *valid* since they are computed based on the tuples in R_2 satisfying the selection predicates σ_2. Based on the certain theta join condition, we could develop the corresponding range predicates based on Table 4.1.

We further propose a hybrid approach by setting a time budget T for the pre-learning approach to execute and will stop pre-learning to use all the predicates it has learned so far within the time budget T.

4.7 Experimental Evaluation

We evaluate the ability of PLAQUE to accelerate query execution using two synthetic data sets and one real data set. PLAQUE along with the techniques for learning predicates are implemented in an Apache project VanillaDB [20, 112].

4.7.1 Methodology

Data Sets

• TPC-H. We use TPC-H (SF=1) as our first data set generated using standard datagen [19] which creates uniformly distributed data. The default TPC-H does not represent a practical data distribution, which is often skewed. Therefore, we further use a modified datagen [2] to create TPC-H datasets with different amounts of skew, i.e., Zipf factor as 1 and 2, respectively.

• SmartBench. To evaluate the learned predicates on User-Defined-Functions (UDFs) benchmark where UDFs are used in queries, we choose SmartBench [54] which is derived from a smart space sensor system and focuses on analytics of IoT data. Smartbench contains multiple sensor tables, such as Bluetooth, WiFi, or cameras as well as a space table that connects sensors to locations. In SmartBench, several UDFs are supported, such as location and occupancy computations.

• IMDB. We finally use a real data set IMDB [14], which contains files with a total size of around 4GB.

Queries

In TPC-H we use query Q2, Q3, Q4 and Q5. [11] In addition to testing the query Q_i to test the effect of MIN and MAX optimizations, we also test PLAQUE for Q_i with the aggregate operation modified to MAX and MIN, denoted by Q_i-max and Q_i-min to evaluate the predicates learned from MIN/MAX conditions. Furthermore, we construct two new queries $Q23$ and $Q24$ to increase the complexity of the query workload. In the SmartBench we use three queries, $Q3$, $Q5$, and $Q10$, where two UDFs are used in the queries, computing

[11] Minor query modifications or rewriting are made while preserving the query complexities and semantics such that they can be supported by VanillaDB which does not support specific SQL features, such as nested queries.

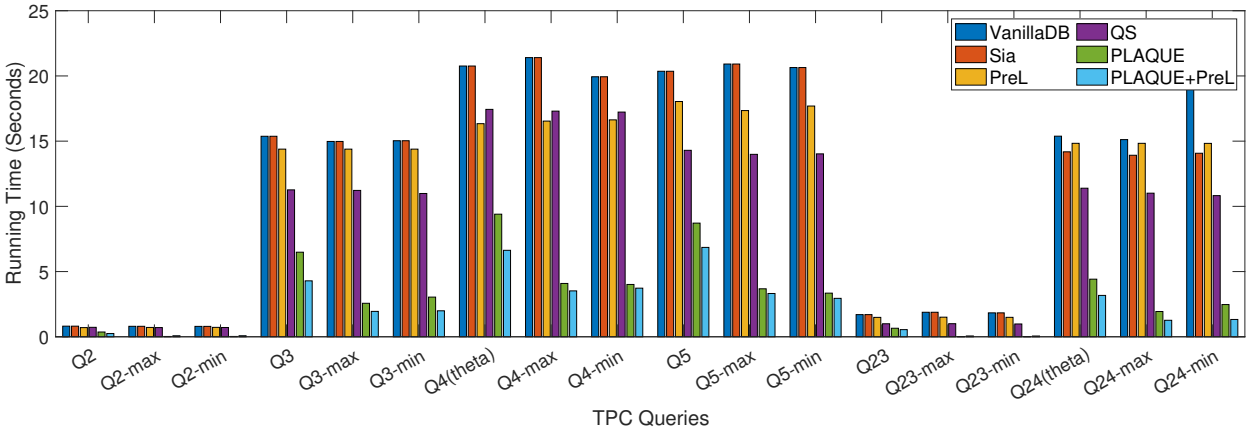


Figure 4.9: Query Run Time on TPCH.

location of a person [79] and occupancy of a room [80]. In the IMDB data set, we manually create four selection-projection-join-aggregate queries (i.e., $Q1$-$Q4$), and for each query Q_i we modify the aggregate condition to be MAX and MIN, and thus creating additional two queries Q_i-max and Q_i-min for each Q_i.

Compared Approaches

We compared the performance of the following six strategies. (1) **VanillaDB**: standard query optimizer and executor implemented in VanillaDB [20, 112]. (2) **Sia** [120]: Sia learns synthesized predicates given a SQL query before query execution. (3) **PreL**: the pre-learning approach described in Section 4.6. (4) **QuickStep (QS)** [93]: QS builds bloom filters for the build table in a hash join and use them to filter (5) **PLAQUE**. (6) **PLAQUE +PreL**: the hybrid approach described in Section 4.6 that combines the predicates learned in PreL and PLAQUE.

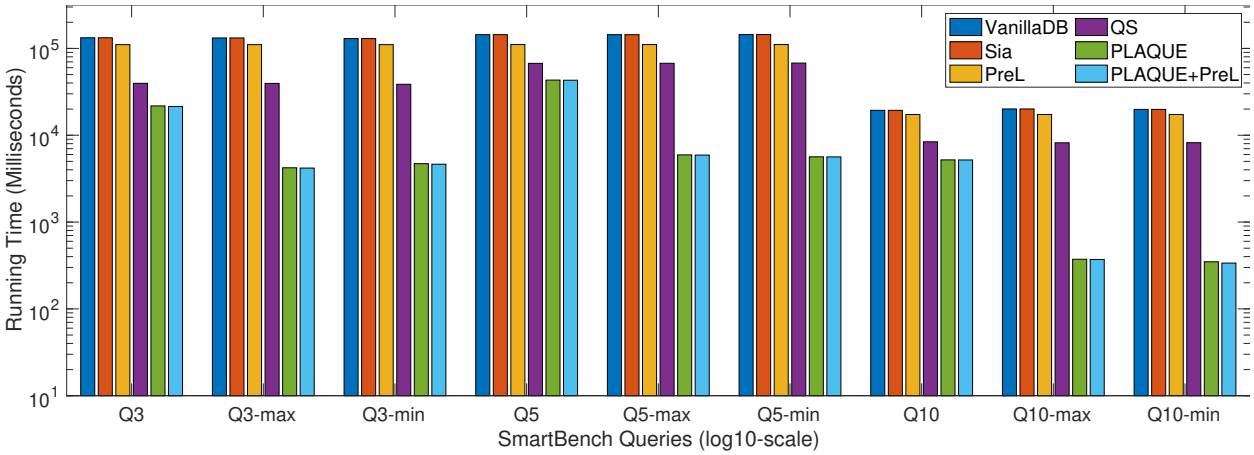


Figure 4.10: Query Run Time on SmartBench.

4.7.2 Experimental Results

Performance of Learned Predicates

We start with reporting the performance of our learned predicates on TPCH (Zipf is 0 using the standard datagen [19]), SmartBench and IMDB data sets in Figure 4.9, Figure 4.10 and Figure 4.11, respectively.

Performance on TPCH. In Figure 4.9, the learned predicates in queries except for MIN/-MAX queries in PLAQUE are able to achieve an improvement over the VallinaDB from 2.3x ($Q2$) to 3.5x ($Q24$), and these improvements become 5.3x ($Q4$-min) to 33.5x ($Q23$-min) for MIN/MAX queries. This observation shows that the learned predicates could significantly speed up query execution especially when the MIN/MAX is used as the aggregate condition, the learned predicates are able to skip a large number of tuples to be processed, thus leading to significant savings.

Performance on SmartBench. In Figure 4.10, we use log10 scale to plot the query running time. The improvement of PLAQUE over the standard query executor, VallinaDB, is up to 6.6x for non-MIN/MAX queries and 58x for MIN/MAX queries. It demonstrates that, when queries contain expensive UDFs, the impact of filtering tuples as done by learned

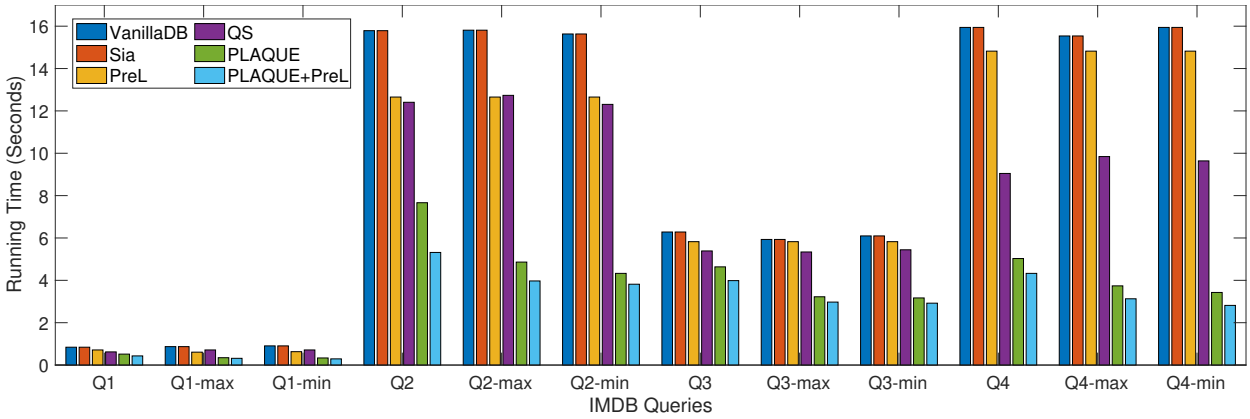
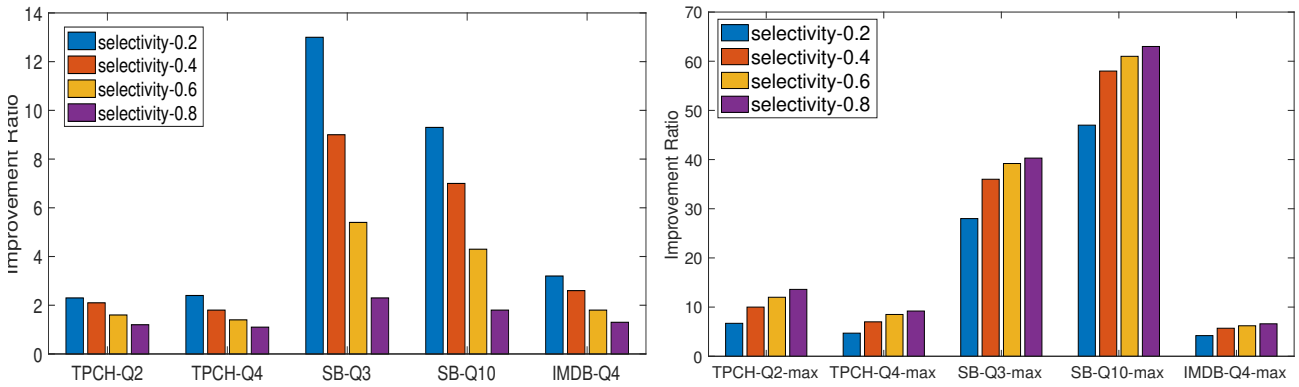


Figure 4.11: Query Run Time on IMDB.



(a) Learned Predicates from Join.

(b) Learned Predicates from MAX.

Figure 4.12: Improvement Ratio on Different Selectivities.

predicates is even more significant.

Performance on IMDB. In the IMDB data set in Figure 4.11, we made similar observations. The predicates learned in PLAQUE improve the standard query executor, VallinaDB, by around 2x ($Q1$) to 3.7x ($Q4$) for join queries without MIN/MAX aggregate conditions, and the improvement goes up from 2.2 ($Q3$-max) to 5.7x ($Q4$-min).

Comparison with Sia. Among 18 queries in the TPCH benchmark, Sia is only able to generate new predicates for Q24, by leveraging the condition l_shipdate < "1992-07-01" and l_shipdate > l_commitdate, to create a new predicate l_commitdate < "1992-07-01". Sia fails to learn new predicates for all queries in SmartBench and IMDB workloads. Sia works well when queries contain additional predicates on join columns. PLAQUE works in a much wider

spectrum of queries and achieves higher performance improvements. Sia is complementary to PLAQUE and the predicates learned by Sia before query execution could be combined with the one learned by PLAQUE during query run time.

Comparison with QS. QS focuses on the hash join and builds the bloom filter for the hash table, which is used to filter tuples in the probe table. The filtering approach of QS is included already in PLAQUE. Its counterpart is learning the membership predicate from a hash join. However, PLAQUE expands the opportunities to learn predicates in several ways based on a much larger repertoire of operators and supports both main memory and index-based implementation of the predicate. The experimental results clearly demonstrate that PLAQUE significantly outperforms QS by 6.3X, 10.4X, and 2.4X in TPCH, SmartBench and IMDB, respectively.

Comparison with PreL. PreL by itself improves the performance by around 1.3X, 2.2X, and 1.2X compared with the standard query execution without any optimization (i.e., VanillaDB), in TPCH, SmartBench and IMDB data sets, respectively. However, our adaptive approach PLAQUE outperforms PreL considerably by around 16.3X, 21.6X, and 2.36X in TPCH, SmartBench, and IMDB, respectively. It demonstrates that PLAQUE can learn more effective predicates than PreL which is only able to learn predicates from equi join and theta conditions on the small relations within the time budget.

Comparison with PLAQUE +PreL. Interestingly, we observe a slight improvement in PLAQUE when appending the predicates learned from PreL. In particular, the hybrid approach PreL+PLAQUE outperforms PLAQUE by 1.12X, 1.01X, and 1.17X in TPCH, SmartBench and IMDB data sets, respectively. This points to an interesting future direction of work that explores a much more carefully designed pre-learning approach that can boost the performance of PLAQUE even further.

The Effect of Query Selectivity

Figure 4.12 examines the performance of PLAQUE over the standard query executor on queries with different selectivities. We select five queries from TPCH, SmartBench (SB for short), and IMDB data sets, i.e., TPCH-Q2, TPCH-Q4, SB-Q3, SB-Q10, and IMDB-Q4, and report the results in Figure 4.12(a). We also select their corresponding MAX queries in Figure 4.12(b). We vary the selectivity of a query to be $0.2, 0.4, 0.6, 0.8$. A query with a lower selectivity indicates it is more selective because a lower number of tuples are in the results. We plot the improvement ratio under various selectivities of the query workloads. The improvement ratio corresponds to the run time of VallinaDB (i.e., standard query executor) divided by that of the PLAQUE, i.e., $\frac{Time(VallinaDB)}{Time(PLAQUE)}$. For join queries without MIN/MAX aggregate conditions, when queries are more selective (low selectivity value), the improvement due to PLAQUE is larger. This is because for any equi join operator o, if one of its inputs, such as the left side of o, o_L, is highly selective, then PLAQUE would be able to learn selective predicates from the tuples coming to o from its left side o_L, and pass the learned selective predicates along the query plan tree using the algorithm in Section 4.5, leading to larger improvement.

For aggregate queries with MAX conditions in Figure 4.12(b), interestingly, we have made a different observation. The improvement from the learned predicates is larger when the query is less selective (higher selectivity value). This is because when the query is less selective, the tuples will probably reach the aggregate operator at an earlier time and thus the predicates can be learned earlier and updated to be more selective in the aggregate operator using the predicate creation algorithm in Section 4.3. On the other hand, when the query is less selective, the improvement brought by the predicates learned from join operators is smaller as observed in Figure 4.12(a). It turns out the improvement due to the predicates from MAX operator is more significant than the one learned from join operators, thus leading to an overall performance improvement with the increase of selectivity. This observation

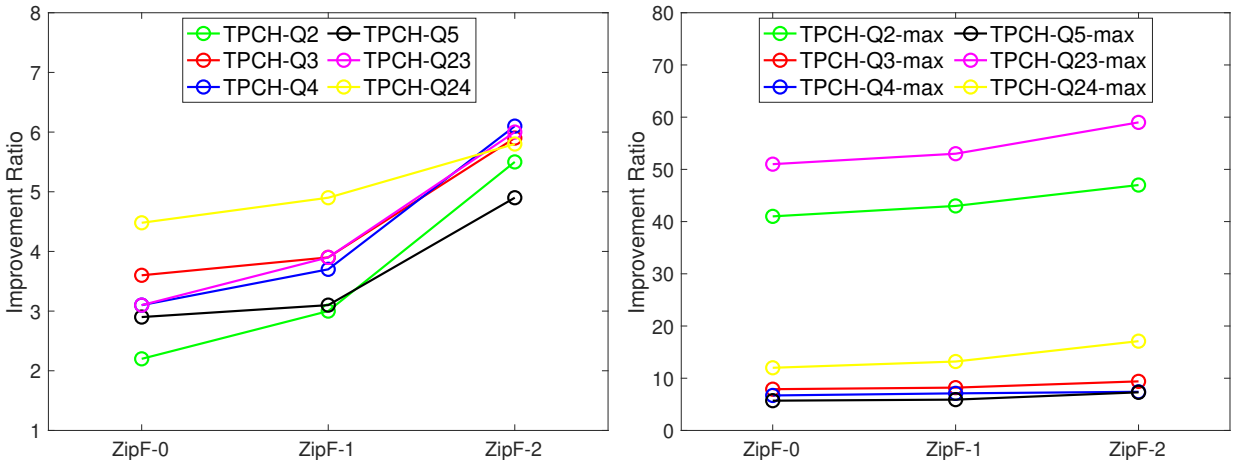


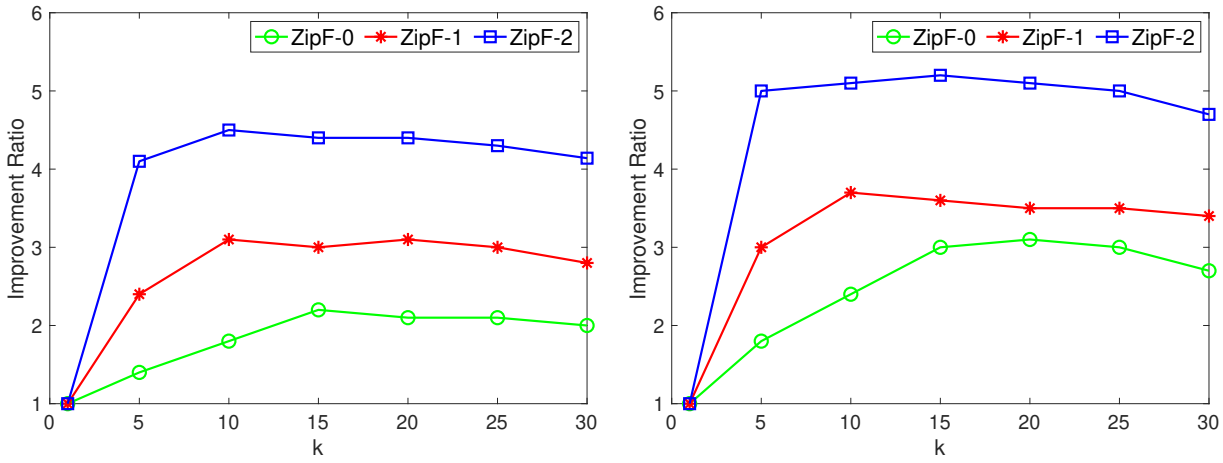
Figure 4.13: Improvement Ratio on Different Data Distributions.

indicates that the predicates learned in MIN/MAX aggregate conditions will work better for slow queries that are less selective, which demonstrates that such learned predicates are even more suitable for long-running queries with significant overheads.

The Effect of Data Distributions

In this experiment, we explore the effect of data distributions on the query performance in Figure 4.13. In particular, we use a modified datagen [2] to create TPC-H datasets with different amounts of skew, i.e., Zipf factor as 1 and 2, respectively. The standard TPC-H [19] comes with a Zipf as 0, which means that the data values have a uniform distribution in each column. We report the improvement ratio of PLAQUE over the VallinDB, and discuss the result for queries with and without MAX aggregate conditions in Figure 4.13.

For join queries without MIN/MAX aggregate conditions (left picture in Figure 4.13), the improvement due to the learned predicates becomes larger on a more skewed data set with a higher Zipf value. This is expected because using the algorithm in Section 4.3.5 to learn new predicates, we are able to learn a more selective predicate from equi join conditions when values are more skewed. We have similar observations for the MAX aggregate queries. The improvement from the learned predicates slightly increases on a more skewed data set, which



(a) TPCH-Q2 (b) TPCH-Q4

Figure 4.14: Improvement Ratio on Various k for Predicates Learned from Equi Join.

is primarily contributed by the predicates learned from equi join conditions, and it turns out the predicates learned from MAX aggregate condition are less sensitive to the skewness of the data set than the predicates learned from join conditions.

Parameter Selection in Join Queries

In Section 4.3.5, when we learn range predicates from equi join conditions, we use k to specify the maximum number of range predicates we wish to learn from an equi join condition. In this experiment, we explore the effect of k on the query performance of TPCH-Q2 and TPCH-Q4, by varying k from 1 to 30, and report the improvement ratio in Figure 4.14.

When k is 1, both queries $Q2$ and $Q4$ have the same run time as the standard query executor without improvement. In this case, one range predicate learned from the equi join condition contains the maximum and minimum values on one side of the join input, which will not help eliminate any tuples, and thus will not improve the query performance. When k increases, the improvement ratio quickly increases and then flattens out when k reaches about 10 for both $Q2$ and $Q4$. When k is too large, such as 30, the improvement ratio is slightly lower. This is because learning too many range predicates, which will although improve

the selectivity of the overall learned predicates marginally, leads to additional complexity of applying the learned predicates in the query processing. Empirically, we recommend k as 10 to be the ideal setting when we learn predicates from equi join conditions.

Experimental Summary.

We below summarize the main outcomes of the experiments in PLAQUE and what we learned.

- PLAQUE provides the highest improvement in the MAX/MIN (aggregate/group-by) queries by up to 33.5X in TPCH, 58X in SmartBench, and 5.7X in IMDB.
- PLAQUE also achieves considerable improvements in the queries involving equi-join or theta-join conditions without MIN/MAX operations by 3.2X, 4.8X, and 2.9X in TPCH, SmartBench, and IMDB datasets on average, respectively.
- For the queries with equi-join or theta-join conditions without MIN/MAX, lower selectivity of the queries, larger improvement brought by PLAQUE, while for the queries with MIN/MAX conditions, larger improvements are observed on less selective queries.
- PLAQUE tends to perform better in more skewed data sets when the query predicates will become more selective due to the skewness of data.
- We recommend learning at most 10 range predicates when learning predicates from equi-join conditions.

4.8 Conclusion

In this chapter, we studied the predicate inference problem at query run time. We proposed a set of approaches to learn new predicates from aggregate, equal join, theta join, group

by/having conditions, and further place the learned predicates wisely in the given query plan tree to maximize their benefit of skipping rows early during query execution, leading to possibly significant improvement. The learned filters exhibit monotonic properties, becoming increasingly selective during query processing. we further introduced a pre-learning technique for predicate inference before query optimization, which is synergistically combined with the run time learning approach PLAQUE to enhance performance. Experiments on both synthetic and real datasets demonstrated that our learned predicates can accelerate query execution by up to 33x, and this improvement increased to up to 100x when User-Defined Functions (UDFs) are utilized in queries.

Chapter 5

LOCATER: Cleaning WiFi Connectivity Datasets for Semantic Localization

In the previous chapters, we developed ZIP and PLAQUE techniques to support the interactive analysis at query time when computationally expensive operations are required in the data processing pipeline. While these techniques are general and useful in several domains where data-intensive computations are used, our particular interest is in emerging smart space applications. Data management in smart spaces opens several new challenges, and one of the most prominent challenges is to support interactive analytics on very large volumes of data being captured at large velocities.

In smart space infrastructure, such as a smart building or smart campus, there are a variety of tasks ranging from real-time control, such as occupancy-based heating/air-conditioning control, to analytical tasks, such as evaluating building management policies. A key challenge in all of these applications is the need to localize a person both outside and more importantly

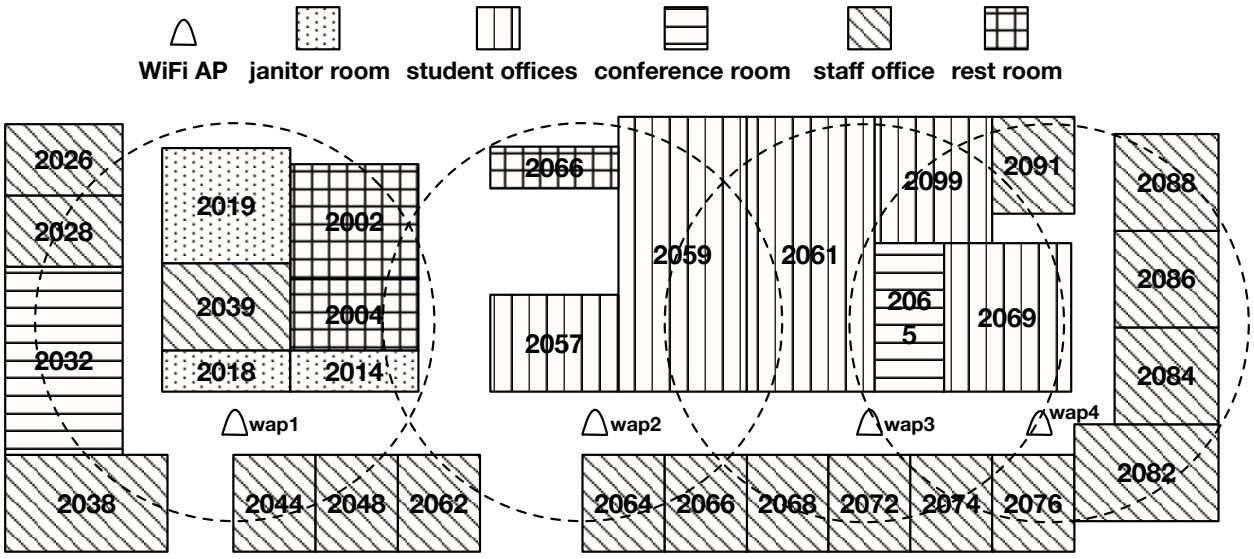


Figure 5.1: A Space with Different Types of Room and WiFi APs.

inside buildings.

In this chapter, we describe an indoor localization solution, entitled LOCATER, to localize a person inside buildings based on WiFi connectivity data by using a series of data cleaning technologies. We further build a real case study in Chapter 6 later to show how ZIP and PLAQUE can benefit the real applications built using LOCATER in the UCI campus testbed.

5.1 Introduction

This chapter studies the challenge of cleaning connectivity data collected by WiFi infrastructures to support *semantic localization* inside buildings. By semantic localization, we refer to the problem of **associating a person's location to a semantically meaningful spatial extent such as a floor, region, or a room.**

Semantic localization differs from and complements the well-studied problem of indoor positioning/localization [94, 40] that aims to determine the exact physical position of people inside buildings (e.g., coordinate (x,y) within radius r, with z% certainty). If indoor positioning/physical localization could be solved accurately, it would be simple to exploit knowledge

about the building's floor plan and layout to determine the semantic location of the device. However, despite over two decades of work in the area [81, 40, 118], and significant technological progress, accurate indoor positioning remains an open problem [118]. Among others, the reasons for this include technology limitations such as costs associated with the required hardware/software [84, 117, 113, 99], the intrusive nature and inconvenience of these solutions for users [68, 40, 94] who require specialized hardware/software, and algorithmic limitations to deal with dynamic situations such as occlusions, signal attenuation, interference [89, 110, 77]. As a result, applications that depend upon accurate positioning and those that could benefit from semantic localization have faced challenges in effectively utilizing indoor localization technologies.

While indoor localization methods have targeted applications such as indoor navigation and augmented reality that require highly accurate positioning, semantic localization suffices for a broad class of smart space applications such as determining occupancy of rooms, thermal control based on occupancy [21], determining density of people in a space and areas/regions of high traffic in buildings —applications that have recently gained significance for COVID-19 prevention and monitoring in workplaces [109, 55], or locating individuals inside large buildings [64, 89]. Despite the utility of semantic localization, to the best of our knowledge, semantic localization has never before been studied as a problem in itself.[1]

This chapter proposes a location cleaning system, entitled *LOCATER* to address the problem of semantic localization. LOCATER can be viewed as a system, the input to which is a log of coarse/inaccurate/incomplete physical locations of people inside the building that could be the result of any indoor positioning/localization strategy or even the raw logs collected by WiFi APs. The output of LOCATER is a clean version of such a log with the semantically

[1]Prior work on indoor localization [65, 66] have evaluated their positioning techniques by measuring the accuracy at which devices can be located physically inside/outside a room. Such work has neither formulated nor addressed the semantic localization challenge explicitly. Instead, naive strategies such as degree of spatial overlap/random selection of an overlapping room out of the several choices are used for their experimental study.

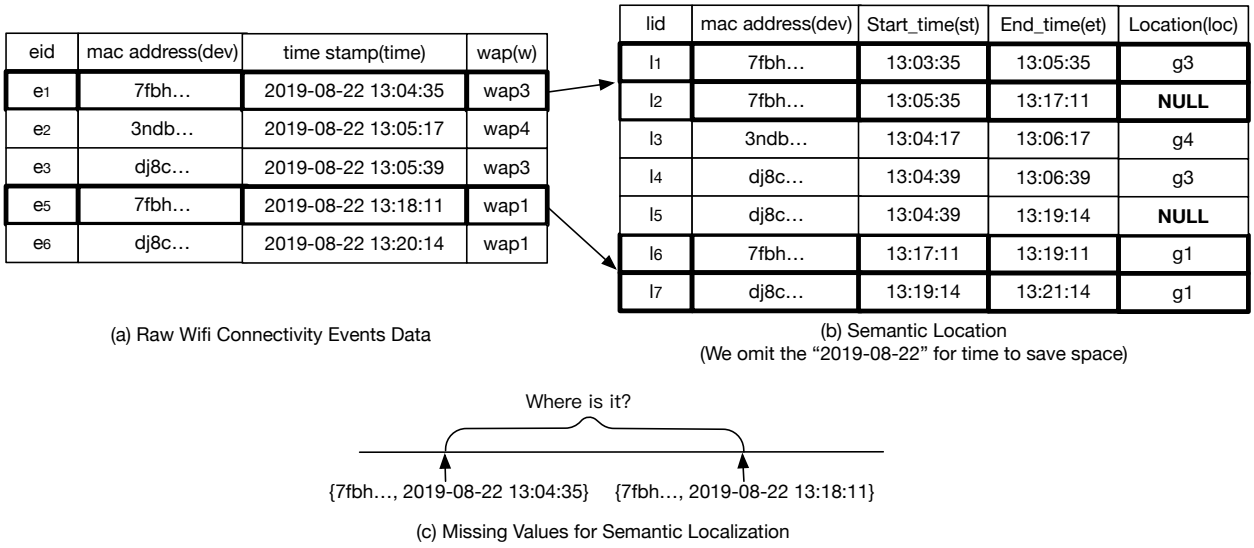


Figure 5.2: Example.

meaningful geographical location of the device in the building – viz., a floor, a region, or, at the fine granularity, a room. Current solutions determine the physical location of a device and use simple heuristics (e.g., the largest overlap with the predicted region) for room-level localization. In contrast, LOCATER postulates associating a device to a semantic location as a data cleaning challenge and exploits the inherent semantics in the sensor data capturing the building usage to make accurate assessments of device locations. LOCATER, we believe, is the first such system to study semantic localization as a problem in its own right.

While LOCATER could be used alongside any indoor positioning/localization solutions, we built LOCATER using a localization scheme that uses connectivity events between devices and the WiFi hardware (viz., access points –APs–) that constitute the WiFi infrastructure of any organization. Such connectivity events, generated in the network when devices connect to an AP, can be collected in real-time using a widely used SNMP (Simple Network Management Protocol), a more recent NETCONF [42], a network management protocol, or from network Syslog [44] containing AP events. Connectivity events consist of observations in the form of ⟨mac address, time stamp, wap⟩ which correspond to the MAC of the WiFi-enabled connected device, the timestamp when the connection occurred, and the WiFi AP (wap) to which the device is connected. Since APs are at fixed locations, connectivity events can

be used to locate a device to be in the region covered by the AP. In Figure 5.2(a) an event e_1 can lead to the observation that the owner of the device with mac address 7bfh... was located in the region covered by wap3, which includes rooms 2059, 2061, 2065, 2066, 2068, 2069, 2072, 2074, 2076, and 2099, in Figure 5.1(a)) at the time stamp 13:04:35.

Using WiFi infrastructure for coarse location, as we do in LOCATER, offers several distinct benefits. First, since it is ubiquitous in modern buildings, using the infrastructure for semantic localization does not incur any additional hardware costs either to users or to the built infrastructure owner. Such would be the case if we were to retrofit buildings with technologies such as RFID, ultra-wideband (UWB), Bluetooth, camera, etc. [81]. Besides being (almost) zero cost, another artifact of the ubiquity of WiFi networks is that such a solution has wide applicability to all types of buildings - airports, residences, office spaces, university campuses, government buildings, etc. Another key advantage is that localization using WiFi connectivity can be performed passively without requiring users to either install new applications on their smartphones or actively participate in the localization process.

Challenges in exploiting WiFi connectivity data. While WiFi connectivity datasets offer several benefits, they offer coarse localization – e.g., in a typical office building, an AP may cover a relatively large region consisting of dozens of rooms, and as such, connectivity information does not suffice to build applications that need semantic localization. Using WiFi connectivity data for semantic localization raises the following technical challenges:

• *Missing value detection and repair.* Devices might get disconnected from the network even when the users carrying them are still within the space. Depending on the specific device, connectivity events might occur only sporadically and at different periodicities, making prediction more complex. These lead to a *missing values* challenge. For example, in Figure 5.2(b) we have the raw connectivity data for device 7fbh at time 13:04:35 and 13:18:11. Location information between these two consecutive time stamps is missing.

• *Location disambiguation.* APs cover large regions within a building that might involve multiple rooms and hence simply knowing which AP a device is connected to may not offer a room-level localization. For example, in Figure 5.2, the device 3ndb connects to wap2, which covers rooms: 2004, 2057, 2059,..., 2068. These values are *dirty* for room-level localization. Such a challenge can be viewed as a location disambiguation challenge.

•*Scalability.* The volume of WiFi data can be very large. For instance, on the campus of University of California, Irvine, with over 200 buildings and 2,000 plus APs, we generate several million WiFi connectivity tuples in one day on average. Thus, data cleaning technique needs to be able to scale to large data sets.

To address the above challenges, LOCATER uses an iterative classification method that leverages temporal features in the WiFi connectivity data to repair the missing values. Then, spatial and temporal relationships between entities are used in a probabilistic model to disambiguate the possible rooms in which the device may be. LOCATER cleans the WiFi connectivity data in a dynamic setting where we clean objects on demand in the context of queries. In addition, LOCATER caches cleaning results of past queries to speed up the system. Specifically, we make the following contributions:

- We propose a novel approach to semantic indoor localization by formalizing the challenge as a combination of missing value cleaning and disambiguation problems in Section 5.2.
- We propose an iterative classification method to resolve the missing value problem in Section 5.3 and a novel probability-based approach to disambiguate room locations without using labeled data in Section 5.4.
- We design an efficient caching technique to enable LOCATER to answer queries in near real-time in Section 5.5.

- We validate our approach in a real-world testbed and deployment. Experimental results show that LOCATER achieves high accuracy and good scalability on both real and simulated data sets in Section 5.6.

5.2 Semantic Localization Problem

The problem of semantic localization consists of associating each device with its location at any instance of time at a given level of spatial granularity.

5.2.1 Space Model

LOCATER models space at three levels of spatial granularity[2]:

Building: The coarsest building granularity B takes the values $B = B_1, ..B_n, b_{out}$, where $B_i = 1...n$ represents the set of buildings and b_{out} represents the fact that the device is not in any of the buildings. We call a device inside a building as *online* device and outside as *offline* device.

Region: Each building B_i contains a set of regions $G = \{g_j : j \in [1...|G|]\}$[3]. We consider a region g_j to be the area covered by the network connectivity of a specific WiFi AP [104] represented with dotted lines in Figure 5.2(a). Let $WAP = \{wap_j : j \in [1...|WAP|]\}$ be the set of APs within the building. Hence, $|G| = |WAP|$ and each wap_j is related to one and only one g_j. Interchangeably, we denote by $Cov(wap_j)$ as the region covered by wap_j. In Figure 5.2(a), there exist four APs $wap_1, ..., wap_4$ and thus there exist four regions such that $G = \{g_1, g_2, g_3, g_4\}$. Regions can often overlap.

[2]The technique can be easily adapted to other spatial models conforming to the nature of the underlying space.

[3]We drop the parameter from $G(B_i)$ and simply refer to it as G since we are dealing with inside a given building.

Table 5.1: Model variables and shorthand notation.

Variable(s)	**Definition/Description**
$B = \{B_1, .., B_n, b_{out}\}$; $g_j \in G$; $r_j \in R$	buildings; regions; rooms
$R(g_j)$	set of rooms in region g_j
$wap_j \in WAP$; $d_i \in D$	WiFi APs; devices
$\delta(d_i)$; $gap_{t_s,t_e}(d_i)$	time interval validity of d_i; gap associated to d_i in $[t_s, t_e]$
$l_i \in L$	semantic location relation

Room: A building contains a set of rooms $R = \{r_j : [1...|R|]\}$ where r_j represents the ID of a room within the building – e.g., $r_1 \rightarrow 2065$. Furthermore, a region g_i contains a subset of R. Let $R(g_i) = \{r_j : [1...|R(g_i)|]\}$ be the set of rooms covered by region g_i. Since regions can overlap, a specific room can be a part of different regions if its extent intersects with multiple regions. For instance, in Figure 5.2(a) room 2059 belongs to both regions g_2 and g_3.

We consider that rooms in a building have metadata associated. In particular, we classify rooms into two types: (i) *public*: shared facilities such as meeting rooms, lounges, kitchens, food courts, etc., that are accessible to multiple users denoted by $R^{pb} \subseteq R$; and (ii) *private*: rooms typically restricted to or owned by certain users such as a person's office, denoted by $R^{pr} \subseteq R$ such that $R = R^{pb} \cup R^{pr}$.

5.2.2 WiFi Connectivity Data

Let $D = \{d_j : j \in [1...|D|]\}$ be the set of devices and $TS = \{t_j : j \in [1...|TS|]\}$ the set of time stamps.[4] Let $E = \{e_i : i \in [1...|E|]\}$ be the WiFi connectivity events table with attributes $\{eid, dev, time, w\}$ corresponding to the event id, device id ($dev \in D$), the time stamp when it occurred ($time \in TS$), and the WiFi AP that generated the event ($w \in WAP$). (As shown in Figure 5.2(b)) For each tuple $e_i \in E$, we will refer to each attribute (e.g., dev) as

[4]The granularity of t_j can be set on various scenarios.

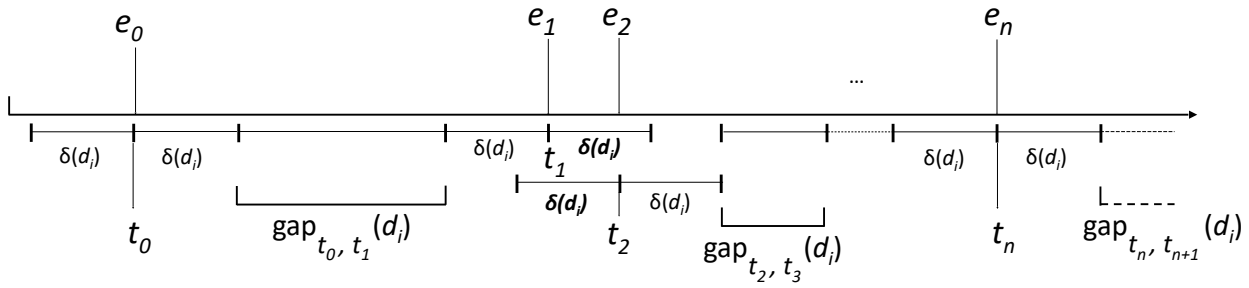


Figure 5.3: Connectivity events of device d_i and their validity.

$e_i.dev$.[5]

Connectivity events occur stochastically even when devices are stationary and/or the signal strength is stable. Events are typically generated when (i) a device connects to a WiFi AP for the first time, (ii) the OS of the device decides to probe available WiFi APs around, or (iii) when the device changes its status. Hence, connectivity logs do not contain an event for every instance of time a device is connected to the WiFi AP or located in space. Because of the sporadic nature of connectivity events, we associate to each event a *validity* period denoted by δ. The value of δ depends on the actual device d_i and is denoted by $\delta(d_i)$. In Figure 5.3 we show some sample connectivity events of device d_i. Let the Valid Interval for an event e_i be $VI_i = \{VI_i.st, VI_i.et\}$, where $VI_i.st$ ($VI_i.et$) is the start (end) time stamp of this interval. Considering the connectivity events of device d_i, the valid interval for event e_i can be considered in three ways. 1) If the subsequent (previous) event e_j of the same device happens after (before) $e_i.time + \delta(e_i.dev)$ ($e_i.time - \delta(e_i.dev)$), then $VI_i.et = e_i.time + \delta(e_i.dev)$ ($VI_i.st = e_i.time - \delta(e_i.dev)$); (e.g., event e_0 in Figure 5.3) 2) Otherwise if the subsequent (previous) event e_j happens close to e_i ($|e_i.time - e_j.time| < \delta(e_i.dev)$), $VI_i.et = e_j.time$ ($VI_i.st = e_i.time$). (e.g., e_1 is valid in $(t_1 - \delta(d_i), t_2)$, and e_2 is valid in $(t_2, t_2 + \delta(d_i))$ in Figure 5.3). While we assume that an event is valid for δ period, there can be portions of time in which no connectivity event is valid in the log for a specific device. We refer to such time periods as *gaps*. Let $gap_{t_s,t_e}(d_i)$ be the gap of device d_i that starts at t_s and ends at t_e time stamp. In Figure 5.3, $gap_{t_0,t_1}(d_i)$ represents a gap of d_i whose time interval is $[t_0, t_1]$.

[5]We use the device's unique MAC address to represent it.

5.2.3 Semantic Location Table

The semantic localization challenge of determining the location of device d_i at any time t_j at a given spatial granularity can be viewed as equivalent to creating a *Semantic Location Table*, $L = \{l_i : i \in [1...|L|]\}$. The table L has the attributes $\{lid, dev, loc, st, et\}$, where the device dev is in the location loc from time st to et. The table L is such that for any device dev and any time t, there exists a tuple in L such that $st \leq t \leq et$.

We can form the table L from the event table E as follows: for each event $e_i \in E$ we create a corresponding tuple $l_j \in L$, where $l_j.dev = e_i.dev$, $l_j.loc = Cov(e_i.w)$, and its start and end times correspond to the validity interval of the event e_i, i.e., $l_j.st = VI_i.st$ and $l_j.et = VI_i.et$. We further insert a tuple l_j corresponding to each gap in the event table E. For each gap $gap_{t_s,t_e}(d_i)$, we generate a tuple $l_j \in L$ such that $dev_j = d_i$, $st_j = t_s$, $et_j = t_e$, $loc_j =$ NULL. Furthermore, let $L^c = \{l_i : loc_i \neq \text{NULL}\}$ be the set of tuples whose location is not NULL, and $L^d = L \setminus L^c$ be the set of tuples whose location is NULL. We further define $L(d_j) = \{l_i : dev_i = d_j\}$ as the set of tuples of device d_j and L_T be the set of tuples of device d_i happening in time period T.

In Figure 5.2(c), we transform the raw WiFi connectivity data to a semantic location table. In this example, we assume $\delta = 1$ minute for all devices. e_1 in Figure 5.2(b) corresponds to l_1 in Figure 5.2(c), where timestamp is expanded to a valid interval, and the gap between e_1 and e_5 in Figure 5.2(b) corresponds to the tuple l_2 in Figure 5.2(c).

5.2.4 Data Cleaning Challenges

The table L, which captures the semantic location of individuals, contains two data cleaning challenges corresponding to the coarse and fine-grained localization.

Coarse-Grained Localization: Given a tuple l_i with $l_i.loc = \text{NULL}$, consists of imputing the missing location value to a coarse-level location by replacing it with either $l_i.loc = b_{out}$ or $l_i.loc = g_j$ for some region g_j in building B_k.

Fine-Grained Localization: Given a tuple l_i with $l_i.loc = g_j$, consists of determining the room $r_k \in R(g_j)$ the device $l_i.dev$ is located in and updating $l_i.loc = r_k$.

We can choose to clean the entire relation L or clean it on demand at query time. In practice, applications do not require knowing the fine-grained location of all the users at all times. Instead, they pose point queries, denoted by $Query = (d_i, t_q)$, requesting the location of device d_i at time t_q. Hence, we will focus on cleaning the location of the tuple of interest at query time.[6] Thus, given a query (d_i, t_q), LOCATER first determines the tuple in L for the device d_i that covers the time t_q. If the location specified in the tuple is NULL, the coarse-level localization algorithm is executed to determine first the region in which the device is expected to be. If fine-grained location is required, the fine-grained localization algorithm is executed to disambiguate amongst the rooms in the region.

5.3 Coarse-Grained Localization

LOCATER uses an iterative classification algorithm combined with bootstrapping techniques to fill in the missing location of a tuple l_m with $l_m.loc = \text{NULL}$ for the device $l_m.dev$. In the following, we will refer to such a tuple as a *dirty* tuple. For simplicity, we use dev_i, st_i, et_i and loc_i to denote $l_i.dev, l_i.st$, $l_i.et$, $l_i.loc$, respectively.

The algorithm takes as input, $L_T(dev_m)$, a set of historical tuples of the device dev_m in the time period T consisting of N past days before the query time, where N is a parameter set

[6]Notice that we could use query-time cleaning to clean the entire relation L by iteratively cleaning each tuple, though if the goal is to clean the entire table better/more efficient approaches would be feasible. Such an approach, however, differs from our focus on real-time queries over collected data. Similar query-time approaches have been considered recently in the context of online data cleaning [49, 24].

experimentally (see Section 5.6). For a tuple l_i, let $st_i.time$ be the time part of the start timestamp, $st_i.date$ be the date part of the start timestamp, and $st_i.day$ be the day of the week. [7] We define the following features for each tuple $l_i \in L_T(dev_i)$:

- $st_i.time$, $et_i.time$: the start and end time of tuple l_j.
- *duration* $\delta(l_j)$: the duration of the tuple (i.e., $et_i.time - st_i.time$).
- $st_i.day$ ($et_i.day$): the day of the week in which tuple l_j occurred (ended).
- loc_{i-1}, loc_{i+1}: the associated region at the start and end time of the tuple.
- *connection density* ω: the average number of logged connectivity events for the device dev_i during the same time period of l_i for each day in T.

The iterative classification method trains two logistic regression classifiers based on such vectors to label gaps as: 1) Inside/outside and 2) Within a specific region, if inside.

Bootstrapping. The bootstrapping process labels a dirty tuple as inside or outside the building by using heuristics that take into consideration the duration of the dirty tuple (short duration inside and long duration outside). We set two thresholds, τ_l and τ_h, such that a tuple is labeled as b_{in} if $\delta(l_j) \leq \tau_l$ and as b_{out} if $\delta(l_j) \geq \tau_h$. If the duration of a tuple is between τ_l and τ_h, then we cannot label it as either inside/outside using the above heuristic. Such dirty tuples are marked as *unlabeled*. We partition the set of dirty tuples of device d_i, $L_T^d(dev_m)$, into two subsets – $\mathcal{S}_{labeled}$, $\mathcal{S}_{unlabeled}$. For tuples in $\mathcal{S}_{labeled}$ that are classified as inside of the building, to further label them with a region at which the device is located, the heuristic takes into account the start and end region of the gap as follows:

- If $loc_{j-1} = loc_{j+1}$, then the assigned label is loc_{j-1}. In particular, if the regions at the start and end of the tuple are the same, the device is considered to be in the region for the entire duration.
- Otherwise, we assign as label a region g_k which corresponds to the most visited region

[7] We assume that gaps do not span multiple days.

Algorithm 5: Iterative classification algorithm.

```
Input: S_labeled, S_unlabeled
while S_unlabeled is not empty do
    classifier ← TrainClassifier(S_labeled);
    max_confidence ← −1, candidate_tuple ← NULL;
    for tuple ∈ S_unlabeled do
        prediction_array, label ← Predict(classifier, tuple);
        confidence ← variance(prediction_array);
        if confidence > max_confidence then
            max_confidence ← confidence;
            candidate_tuple ← tuple;
    S_unlabeled ← S_unlabeled − candidate_tuple ;
    S_labeled ← S_labeled + (candidate_tuple, label) ;
return classifier;
```

of dev_j in connectivity events that overlap with the dirty tuple whose connection time is between $st_j.time$ and $et_j.time$.

Iterative Classification. We use iterative classification to label the remaining (unlabeled) dirty tuples $\mathcal{S}_{unlabeled}$, as described in Algorithm 5. For each device d_i, we learn logistic regression classifiers on $\mathcal{S}_{labeled}$ by using the function `TrainClassifier`($\mathcal{S}_{labeled}$) in Algorithm 5. The learned classifiers are then used to classify the unlabeled dirty tuples associated with the device.[8]

Algorithm 5 is firstly executed at the building level to learn a model to classify if an unlabeled dirty tuple is inside/outside the building. To this end, let $\mathcal{L}$ be the set of possible training labels - i.e., inside/outside the building. The method `Predict`($classifier, gap$), returns an array of numbers from 0 to 1, where each number represents the probability of the dirty tuple being assigned to a label in $\mathcal{L}$ (all numbers in the array sum up to 1), and the label with the highest probability in the array. In the array returned by `Predict`, a larger variance means that the probability of assigning a certain label to this dirty tuple is higher than other dirty tuples. Thus, we use the variance of the array as the confidence value of each prediction. In

[8]We assume that connectivity events exist for the device in the historical data considered, as is the case with our data set. If data for the device does not exist, e.g., if a person enters the building for the first time, then, we can label such devices based on aggregated location, e.g., the most common label for other devices.

each outer iteration of the loop (Line 1-11), as a first step, a logistic regression classifier is trained on $\mathcal{S}_{labeled}$. Then, it is applied to all tuples in $\mathcal{S}_{unlabeled}$. For each iteration, the dirty tuple with the highest prediction confidence is removed from $\mathcal{S}_{unlabeled}$ and added to $\mathcal{S}_{labeled}$ along with its predicted label. This algorithm terminates when $\mathcal{S}_{unlabeled}$ is empty and the classifier trained in the last round will be returned. The same process is followed to learn a model at the region level for dirty tuples labeled as inside the building. In this case, when executing the algorithm $\mathcal{L}$ contains the set regions in the building (i.e., G). The output is a classifier that labels a dirty tuple with the region where the device is located.

Given the two trained classifiers, for a dirty tuple l_m, we first use the inside/outside classifier to classify l_m as inside or outside of the building. If the tuple l_m is classified as outside, then $loc_m = b_{out}$. Otherwise, we further classify the tuple l_m using the region classifier to obtain its associated region. Then, the device will be located in such region and LOCATER will perform the room-level fine-grained localization as we will explain in the following section.

5.4 Fine-Grained Localization

Given a query $Q = (d_i, t_q)$ and the associated tuple l_m whose location has been cleaned by the coarse-level localization algorithm, this step determines the specific room $r_j \in R(l_m.loc)$ where d_i is located at time t_q. As shown in Figure 5.2(c), tuples l_1, l_3, are logged for two devices d_1 and d_2 with MAC addresses 7fbh and 3ndb, respectively. Assume that we aim to identify the room in which device d_1 was located at 2019-08-22 13:04. Given that d_1 was connected to wap3 at that time, the device should have been located in one of the rooms in that region g_3 – i.e., $R(g_3) = \{2059, 2061, 2065, 2069, 2099\}$. These are called *candidate rooms* of d_1, and we omit the remaining candidate rooms 2066, 2068, 2072, and 2074 for simplicity. The main goal of the fine-grained location approach is to identify in which candidate room d_1 was located.

5.4.1 Affinity Model

LOCATER's location prediction is based on the concept of *affinity* which models relationships between devices and rooms.

- *Room affinity*: $\alpha(d_i, r_j, t_q)$ denotes the affinity between a device d_i and a room r_j (i.e., the chance of d_i being located in r_j at time t_q), given the region g_k in which d_i is located at time t_q.
- *Group affinity*: $\alpha(D, r_j, t_q)$ represents the affinity of a set of devices D to be in a room r_j at time t_q (i.e., the chance of all devices in D being located in r_j at t_q), given that device $d_i \in D$ is located in region g_k at time t_q.

Note that the concept of group affinity generalizes that of room affinity. While room affinity is a device's conditional probability of being in a specific room, given the region it is located in, group affinity of a set of devices represents the probability of the set of devices being co-located in a specific room r_j at t_q. We differentiate between these since the methods we use to learn these affinities are different, as will be discussed in the following section. We first illustrate how affinities affect localization prediction using the example in Figure 5.4, which shows a hypergraph representing room and group affinities at time t_q. For instance, an edge between d_1 and the room 2065 shows the affinity $\alpha(d_1, 2065, t_q) = 0.3$. Likewise, the hyperedge $\langle d1, d2, 2065\rangle$ with the label 0.12 represents the group affinity, represented as $\alpha(\{d_1, d_2\}, 2065, t_q) = 0.12$. If at time t_q device d_2 is not online (i.e., there are no events associated with d_2 at t_q in that region), we can predict that d_1 is in room 2061 since d_1's affinity to 2061 is the highest. On the other hand, if d_2 is online at t_q, the chance that d_1 is in room 2065 increases due to the group affinity $\alpha(\{d_1, d_2\}, 2065, t_q) = 0.12$. The location prediction for a device d_i, thus, must account for both the room and group affinity.

Room Probability. Let $Pr(d_i, r_j, t_q)$ be the probability that a device d_i is in room r_j at time t_q. Given a query $Q = (d_i, t_q)$ and its associated tuple l_m, the goal of the fine-grained

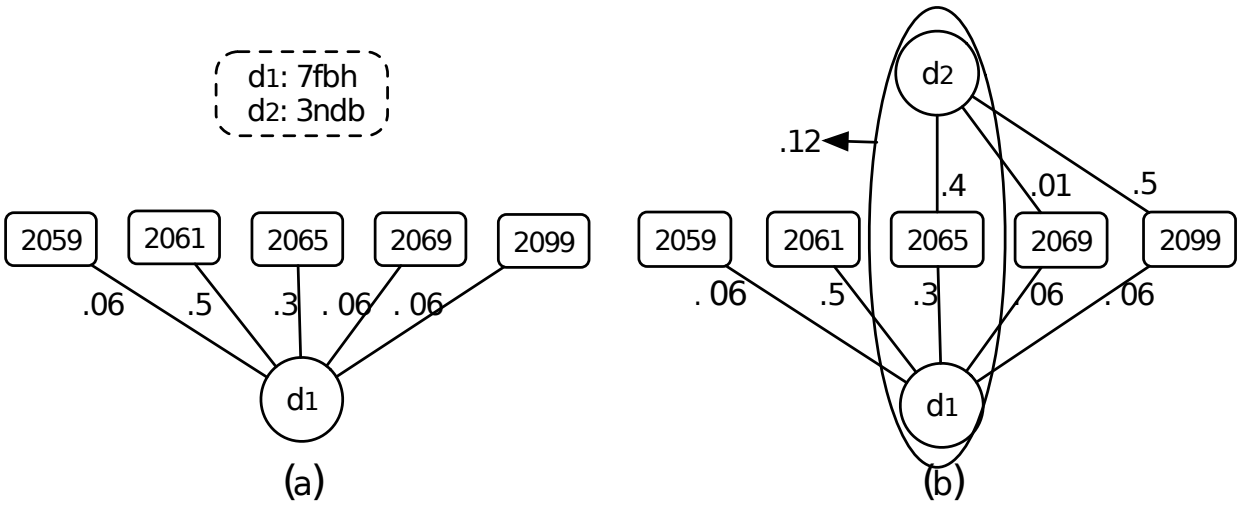


Figure 5.4: Graph view in fine-grained location cleaning.

location prediction algorithm is to find the room $r_j \in R(l_m.loc)$ of d_i at time t_q, such that r_j has the maximum $Pr(d_i, r_j, t_q), \forall r_j \in r_j \in R(l_m.loc)$. We first describe how affinities are estimated.

5.4.2 Affinity Learning

Learning Room Affinity. One of the challenges in estimating room affinity is the potential lack of historical room-level location data for devices - collecting such data would be prohibitively expensive, especially when we consider large spaces with tens of thousands of people/devices. Our approach, thus, does not assume the availability of room-level localization data which could have been used to train specific models.[9] Instead, we compute it based on the available background knowledge and space metadata.

To compute $\alpha(d_i, r_j, t_q)$, we associate for each device d_i a set of preferred rooms $R^{pf}(d_i)$ – e.g., the personal room of d_i's owner based on space metadata, or the most frequent rooms d_i's owner enters based on background knowledge. $R^{pf}(d_i)$ is an empty set if d_i's owner does not have any preferred rooms. If r_j is one the preferred rooms of d_i, i.e., $r_j \in R^{pf}(d_i)$,

[9]Extending our approach to handle when such data is obtainable for at least a subset of devices is interesting and part of our future work.

we assign to r_j the highest weight denoted by w^{pf}. Similarly, if r_j is a public room, i.e., $r_j \in (R(g_x) \cap R^{pr}) \diagdown R^{pf}_{d_i}$, we assign to r_j the second highest weight denoted by w^{pb}. Finally, if r_j is a private room, i.e., $r_j \in (R(g_x) \cap R^{pr}) \diagdown R^{pf}(d_i)$, we assign to r_j the lowest weight denoted by w^{pr}. In general, these weights are assigned based on the following conditions: (1) $w^{pf} > w^{pb} > w^{pr}$ and (2) $w^{pf} + w^{pb} + w^{pr} = 1$. The influence of different combinations of w^{pf}, w^{pb}, w^{pr} is evaluated in Section 5.6.

We illustrate the assignment of these weights by using the graph of our running example. As already pointed out, d_1 connects to wap3 of region g_3, where $R(g_3) = \{2059, 2061, 2065, 2069, 2099\}$. In addition, d_1's office, room 2061, is the only preferred room ($R^{pf}(d_1) = \{2061\}$) and 2065 is a public room, such as a meeting room. Hence, the remaining rooms in $R^{pf}(d_1)$ are other personal offices associated with other devices. Based on Figure 5.4, a possible assignment of w^{pf}, w^{pb}, w^{pr} to the corresponding rooms is as follows: $\alpha(d_1, 2061, t_q) = \frac{w^{pf}}{1} = 0.5$, $\alpha(d_1, 2065, t_q) = \frac{w^{pb}}{1} = 0.3$, and any room in $R(g_3) \setminus (R^{pf}(d_1) \cup R^{pb})$ – i.e., {2059, 2069, 2099} shares the same room affinity, which is $\frac{w^{pr}}{3} = 0.066$.

Note that since the room affinity is not data-dependent, we can pre-compute and store it to speed up the computation. Furthermore, preferred rooms could be time-dependent. For instance, the user is expected to be in the break room during lunch, while being in the office during other times. Such a time-dependent model would potentially result in more accurate room-level localization if such metadata is available.

Learning Group Affinity. Before describing how we compute group affinity, we first define the concept of *device affinity*, denoted by $\alpha(D)$, which intuitively captures the probability of devices/users being part of a group and be co-located which serves as a basis to compute group affinity. Consider all the tuples in L. Let $L(d_i) = \{l_j : dev_j = d_i\}$ be the set of tuples corresponding to device $d_i \in D$, and $L(D)$ be the tuples of devices in D. Consider the set of semantic location tuples such that for each tuple $l_a \in L(d_i)$, belonging to that set, and for every other device $d_j \in D \setminus d_i$, there exists a tuple $l_b \in L(d_j)$ where devices $l_a.dev$

and $l_b.dev$ are in the same region at approximately the same time, i.e., $TR_a \cup TR_b \neq \varnothing$ and $l_a.loc = l_b.loc$ (not NULL). Intuitively, such a tuple set, referred to as the intersecting tuple set, represents the times when all the devices in D are in the same area since they are connected to the same WiFi AP. We compute device affinity $\alpha(D)$ as a fraction of such intersecting tuples among all tuples in $L(D)$.

Given device affinity $\alpha(D)$, we can now compute the group affinity among devices D in room r_j at time t_q, i.e., $\alpha(D, r_j, t_q)$. Let R_{is} be the set of intersecting rooms of connected regions for each device in D at time t_q: $R_{is} = \bigcap R(l_i.loc), l_i \in L_{t_q}(D)$. If r_j is not one of the intersecting rooms, $r_j \notin R_{is}$, then $\alpha(D, r_j, t_q) = 0$. Otherwise, to compute $\alpha(D, r_j, t_q)$, we first determine conditional probability of a device $d_i \in D$ to be in r_j given that $r_j \in R_{is}$ at time t_q.

Let $@(d_i, r_j, t_q)$ represent the fact that device d_i is in room r_j at time t_q, and likewise $@(d_i, R_{is}, t_q)$ represent the fact that d_i is in one of the rooms in R_{is} at t_q. $P(@(d_i, r_j, t_q)|@(d_i, R_{is}, t_q))$ $= \frac{P(@(d_i, r_j, t_q))}{P(@(d_i, R_{is}, t_q))}$, where $P(@(d_i, R_{is}, t_q)) = \sum_{r_k \in R_{is}} P(@(d_i, r_k, t_q))$. We now compute $\alpha(D, r_j, t_q)$, where $r_j \in R_{is}$ as follows:

$$\alpha(D, r_j, t_q) = \alpha(D) \prod_{d_i \in D} P(@(d_i, r_j, t_q)|@(d_i, R_{is}, t_q)) \tag{5.1}$$

Intuitively, the group affinity captures the probability of the set of devices being in a given room based on the room level affinity of individual devices, given that the individuals carrying the devices are co-located, which is captured using the device affinity.

We explain the notation using the example in Figure 5.4(b). Let us assume that the device affinity between d_1 and d_2 is .4, i.e., $\alpha(\{d_1, d_2\}) = .4$. The set $R_{is} = \{2065, 2069, 2099\}$. We compute $\alpha(\{d_1, d_2\}, 2065, t_q)$ as $P(@(d_1, 2065, t_q)|@(d_1, R_{is}, t_q)) = \frac{.3}{.3+.06+.06} = .69$. Similarly,

$P(@(d_2, 2065, t_q)|@(d_2, R_{is}, t_q)) = \frac{.4}{.4+.01+.5} = .44$. Finally, $\alpha(\{d_1, d_2\}, 2065, t_q) = .4 * .69 * .44 = .12$.

5.4.3 Localization Algorithm

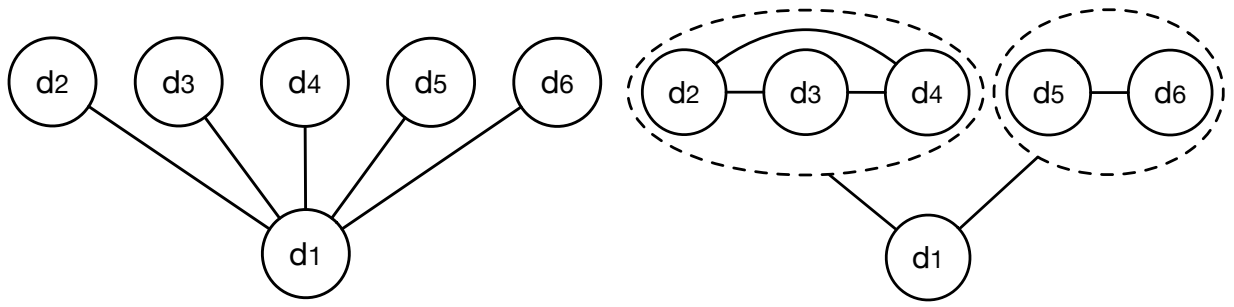
Given a query $Q = (d_i, t_q)$, its associated tuple l_m, and candidate rooms $R(l_m.loc)$, we compute the room probability $Pr(d_i, r_j, t_q)$ for each $r_j \in R(l_m.loc)$ and select the room with the highest probability as an answer to Q. We first define the concept of the set of *neighbor* devices of d_i, denoted by $D_n(d_i)$. A device $d_k \in D_n(d_i)$ is a *neighbor* of d_i if: (i) d_k is online at time t_q (inside the building); (ii) $\alpha(\{d_i, d_k\}, r_j, t_q) > 0$ for each $r_j \in R(l_m.loc)$; and (iii) $R(l_m.loc) \cap R(g_y) \neq \varnothing$, where $R(g_y)$ is the region in which d_k is located. In Figure 5.4(b), d_2 is a neighbor of d_1. Essentially, neighbors of a device d_i could influence the location prediction of d_i since they will contribute a non-zero group affinity for d_i.

Since we use the concept of neighbor always in the context of a device d_i, we will simplify the notation and refer to $D_n(d_i)$ as D_n. Since processing every device in D_n can be computationally expensive, the localization algorithm considers the neighbors iteratively until there is enough confidence that the unprocessed devices will not change the current answer. Let $\bar{D}_n \subseteq D_n$ be the set of devices that the algorithm has processed. We denote as $P(r_j|\bar{D}_n)$ the probability of r_j being the answer of Q given the devices and their locations in $\bar{D}_n$[10] that have been processed by the algorithm so far. Using Bayes's rule:

$$P(r_j|\bar{D}_n) = \frac{P(\bar{D}_n|r_j)P(r_j)}{P(\bar{D}_n|r_j)P(r_j) + P(\bar{D}_n|\neg r_j)P(\neg r_j)} \tag{5.2}$$

where we estimate $P(r_j)$ using the room affinity $\alpha(d_i, r_j, t_q)$.

[10]We could express the above, as explained in Section 5.4.2, as $P(@(d_i, r_j, t_q)|\bar{D}_n)$ but we simplify the notation for the brevity of the following formulas. r_j being the answer of query Q means d_i is in r_j at time t_q, and we write r_j here for simplicity.



(a) Independent Neighbor Set. (b) Dependent Neighbor Set.

Figure 5.5: Graph view in fine-grained location cleaning.

We first compute $P(r_j|\bar{D}_n)$ under a simplified assumption that the probability of d_i to be in room r_j given any two neighbors in D_n is conditionally independent. Then, we consider that multiple neighbor devices may together influence the probability of d_i to be in room r_j.

Independence Assumption. Since we have assumed conditional independence: $P(\bar{D}_n|r_j) = \prod_{d_k \in \bar{D}_n} P(@(d_k, r_j, t_q)|r_j)$ where $@(d_k, r_j, t_q)$ represents that d_k is located in r_j at time t_q. By definition, $P(@(d_k, r_j, t_q)|r_j) = \frac{P(@(d_k, r_j, t_q), r_j)}{P(r_j)}$. The numerator represents the group affinity, i.e., $P(@(d_k, r_j, t_q), r_j) = \alpha(\{d_k, d_i\}, r_j, t_q)$. Similarly, $P(@(d_k, r_j, t_q), \neg r_j) = 1 - \alpha(\{d_k, d_i\}, r_j, t_q)$.

$$P(r_j|\bar{D}_n) = 1/\left(1 + \frac{\prod_{d_k \in \bar{D}_n}(1 - \alpha(\{d_k, d_i\}, r_j, t_q))}{\prod_{d_k \in \bar{D}_n} \alpha(\{d_k, d_i\}, r_j, t_q)}\right) \tag{5.3}$$

To guarantee that our algorithm determines the answer of Q by processing the minimum possible number of devices in $\bar{D}_n$, we compute the expected/max/min probability of r_j being the answer based on neighbor devices in D_n. We consider the processed devices $\bar{D}_n$ as well as unprocessed devices $D_n \setminus \bar{D}_n$. Thus, we consider all the possible room locations (given by coarse-localization) for unprocessed devices. We denote the set of all possibilities for locations of these devices (i.e., the set of possible worlds [5]) by $\mathcal{W}(D_n \setminus \bar{D}_n)$. For each possible world $W \in \mathcal{W}(D_n \setminus \bar{D}_n)$, let $P(W)$ be the probability of the world W and $P(r_j|\bar{D}_n, W)$ be the probability of r_j being the answer of Q given the observations of processed devices $\bar{D}_n$ and the possible world W. We now formally define the expected/max/min probability of r_j given

all the possible worlds.

Definition 1. *Given a query $Q = (d_i, t_q)$, a region $R(g_x)$, a set of neighbor devices D_n, a set of processed devices $\bar{D}_n \subseteq D_n$, and the candidate room $r_j \in R(g_x)$ of d_i, the expected probability of r_j being the answer of Q, denoted by $expP(r_j|\bar{D}_n)$, is defined as follows:*

$$expP(r_j|\bar{D}_n) = \sum_{W \in \mathcal{W}(D_n \setminus \bar{D}_n)} P(W)P(r_j|\bar{D}_n, W) \tag{5.4}$$

The maximum probability of r_j, denoted by $maxP(r_j|\bar{D}_n)$, is:

$$maxP(r_j|\bar{D}_n) = \max_{W \in \mathcal{W}(D_n \setminus \bar{D}_n)} P(r_j|\bar{D}_n, W) \tag{5.5}$$

The minimum probability can be defined similarly.

The algorithm terminates the iteration only if there exists a room $r_i \in R(g_x)$, for any other room $r_j \in R(g_x), r_i \neq r_j$, such that $minP(r_i|\bar{D}_n) > maxP(r_j|\bar{D}_n)$. However, it is often difficult to satisfy such strict condition in practice. Thus, we relax this condition using the following two conditions:

1. $minP(r_i|\bar{D}_n) > expP(r_j|\bar{D}_n)$(or $P(r_j|\bar{D}_n)$)
2. $expP(r_i|\bar{D}_n)$(or $P(r_i|\bar{D}_n)$) $> maxP(r_j|\bar{D}_n)$

In Section 5.6 we show that these loosen conditions enable the algorithm to terminate efficiently without sacrificing the quality of the results.

A key question is, *how do we compute these probabilities efficiently?* To compute the maximum probability of d_i being in r_j, we can assume that all unprocessed devices are in room r_j as described in the theorem below.

Algorithm 6: Fine-grained Localization

Input: $Q = (d_i, t_q), D_n, L, l_m$

```
Stop_flag ← false;
D̄_n ← ∅;
for d_k ∈ D_n do
    D̄_n ← d_k;
    for r_j ∈ R(l_m.loc) do
        Compute P(r_j|D̄_n);
    if D_n independent then
        Find top-2 probability P(r_a|D̄_n), P(r_b|D̄_n);
        Compute minP(r_a|D̄_n), maxP(r_a|D̄_n), expP(r_a|D̄_n);
        Compute minP(r_b|D̄_n), maxP(r_b|D̄_n), expP(r_b|D̄_n);
        if minP(r_a|D̄_n) ≥ expP(r_b|D̄_n) or expP(r_a|D̄_n) ≥ maxP(r_b|D̄_n) then
            Stop_flag ← true;
    if D_n dependent then
        if ∀D̄_nl ⊆ D̄_n, α({D̄_nl, d_i}, r_j, t_q) = 0 then
            Stop_flag ← true;
    if Stop_flag == true then
        break;
    return r_a;
```

Theorem 1. *Given a set of already processed devices $\bar{D}_n$, a candidate room r_j of d_i ,and the possible world W where all devices $D_n \setminus \bar{D}_n$ are in room r_j, then, $maxP(r_j|\bar{D}_n) = Pr(r_j|\bar{D}_n, W)$.*

Proof of Theorem 1. Consider another possible world W_0 where some unseen devices are not in r_j. We denote by $W_0(d)$ the room where d is located in W_0. We can transform W to W_0 step by step, wherein each step for a device that is not in r_j in W_0, we change its room location from r_j to $W_0(d)$. Assuming the transformation steps are W, W_n, ..., W_1, W_0, we can prove easily: $Pr(r_j|\bar{D}_n, W) > Pr(r_j|\bar{D}_n, W_n) > ... > Pr(r_j|\bar{D}_n, W_1) > Pr(r_j|\bar{D}_n, W_0)$.

Likewise, to compute the minimum probability, we can simply assume that none of the unprocessed devices are in room r_j. The following theorem states that we can compute the minimum by placing all the unprocessed devices in the room (other than r_j) in which d_i has the highest chance of being at time t_q.

Theorem 2. *Given a set of already processed devices $\bar{D}_n$, a candidate room $r_j \in R(g_x)$, $r_{max} = argmax_{r_i \in R(g_x) \setminus r_j} P(r_i|\bar{D}_n)$, and a possible world W where all devices in $D_n \setminus \bar{D}_n$ are in room r_{max}, then, $minP(r_j|\bar{D}_n) = P(r_j|\bar{D}_n, W)$.*

Proof of Theorem 2. Consider another possible world W_0 where some unseen devices are not in room r_{max} and d is located in room $W_0(d)$. We can transform W to W_0 step by step, wherein each step for a device which is located in room $W_0(d) \neq r_{max}$ in W_0, we change its value from r_{max} to $W_0(d)$. Assuming the transformation steps are W, W_n, ..., W_1, W_0, we can prove easily: $Pr(r_j|\bar{D}_n, W) < Pr(r_j|\bar{D}_n, W_n) < ... < Pr(r_j|\bar{D}_n, W_1) < Pr(r_j|\bar{D}_n, W_0)$.

For the expected probability of r_j being the answer of Q, we prove that it equals to $P(r_j|\bar{D}_n)$.

Theorem 3. *Given a set of independent devices D_n, the set of already processed devices $\bar{D}_n$, and the candidate room r_j, then,*
$expP(r_j|\bar{D}_n) = P(r_j|\bar{D}_n)$.

Proof of Theorem 3. We compute each possible world's probability based on the probabilities of the rooms being the answer, which are computed based on observations on $\bar{D}_n$.

$$
\begin{aligned}
expPr(r_j|\bar{D}_n) &= \sum_{W \in \mathcal{W}(D_n \setminus \bar{D}_n)} Pr(W) Pr(r_j|\bar{D}_n, W) \qquad (5.6)\\
&= \sum_{W \in \mathcal{W}(D_n \setminus \bar{D}_n)} Pr(W|\bar{D}_n) \frac{Pr(r_j, \bar{D}_n, W)}{Pr(\bar{D}_n, W)} \\
&= \sum_{W \in \mathcal{W}(D_n \setminus \bar{D}_n)} Pr(W|\bar{D}_n) \frac{Pr(\bar{D}_n) Pr(r_j, W|\bar{D}_n)}{Pr(\bar{D}_n) Pr(W|\bar{D}_n)} \\
&= \sum_{W \in \mathcal{W}(D_n \setminus \bar{D}_n)} Pr(W|\bar{D}_n) \frac{Pr(\bar{D}_n) Pr(r_j|\bar{D}_n) Pr(W|\bar{D}_n)}{Pr(\bar{D}_n) Pr(W|\bar{D}_n)} \\
&= \sum_{W \in \mathcal{W}(D_n \setminus \bar{D}_n)} Pr(W|\bar{D}_n) Pr(r_j|\bar{D}_n) \\
&= Pr(r_j|\bar{D}_n)
\end{aligned}
$$

Relaxing the Independence Assumption. We next relax the conditional independence assumption we have made so far. In this case, we cannot treat each neighbor device independently. Instead, we divide $\bar{D}_n$ into several *clusters* where every neighbor device in a cluster has non-zero group affinity with the rest of the devices. Let $\bar{D}_{nl} \subseteq \bar{D}_n$ be a cluster where $\forall d_k, d_k^{'} \in \bar{D}_{nl}, \alpha(\{d_k, d_k^{'}\}, r_j, t_q) > 0$. In addition, group affinity of devices of any pair of devices in different clusters equals zero, i.e., $\forall d_k \in \bar{D}_{nl}, d_k^{'} \in \bar{D}_{nl'}$, where $l \neq l^{'}$, $\alpha(\{d_k, d_k^{'}\}, r_j, t_q) = 0$. In Figure 5.5(b), $\bar{D}_{n1} = \{d_2, d_3, d_4\}$ and $\bar{D}_{n2} = \{d_5, d_6\}$. Naturally, we have $\bar{D}_n = \bigcup_l \bar{D}_{nl}$. In this case, we assume that each cluster affects the location prediction of d_i independently.

Thus, probability $P(\bar{D}_n|r_j) = \prod_l P(\bar{D}_{nl}|r_j)$. For each cluster, we compute its conditional probability $P(\bar{D}_{nl}|r_j) = \frac{P(\bar{D}_{nl}, r_j)}{P(r_j)}$, where $P(\bar{D}_{nl}, r_j) = \alpha(\{\bar{D}_{nl}, d_i\}, r_j, t_q)$. The reason is that $P(\bar{D}_{nl}, r_j)$ is the probability that all devices in $\bar{D}_{nl}$ and d_i are in room r_j, which equals $\alpha(\{\bar{D}_{nl}, d_i\}, r_j, t_q)$ by definition. Thus,

$$P(r_j|\bar{D}_n) = 1/(1 + \frac{1 - \prod_l \alpha(\{\bar{D}_{nl}, d_i\}, r_j, t_q)}{1 - \alpha(d_i, r_j)}) \quad (5.7)$$

the algorithm terminates when the group affinity for any cluster turns zero.

Finally, we describe the complete fine-grained location cleaning algorithm in Algorithm 6. Given $Q = (d_i, t_q)$, we observe only the neighbor devices at time t_q (Line 4-5). Next, we compute the probability of $P(r_j|\bar{D}_n)$ for every candidate room in $R(l_m.loc)$ (Line 7-8). If devices are independent, we select two rooms with top-2 probability and use loosen stop condition to check if the algorithm converges (Line 10-14). Otherwise, we check if all clusters have zero group affinity (Line 15-17). Finally, we output the room when the stop condition is satisfied (Line 13-16).

5.5 System Implementation

We describe the prototype of LOCATER built based on the previous coarse and fine-grained localization algorithms. Also, we describe a caching engine to scale LOCATER to large connectivity data sets.

Architecture of LOCATER. Figure 5.6 shows the high-level architecture of the LOCATER prototype. LOCATER ingests a real-time stream of WiFi connectivity events (as discussed in Section 5.2). Additionally, LOCATER takes as input *metadata about the space* which includes the set of *WiFi APs* deployed in the building, the set of *rooms* in the building, including whether each room is a public or private space, the *coverage of WiFi APs* in terms of a list of rooms covered by each AP, and the *temporal validity of connectivity events* per type of device in the building.

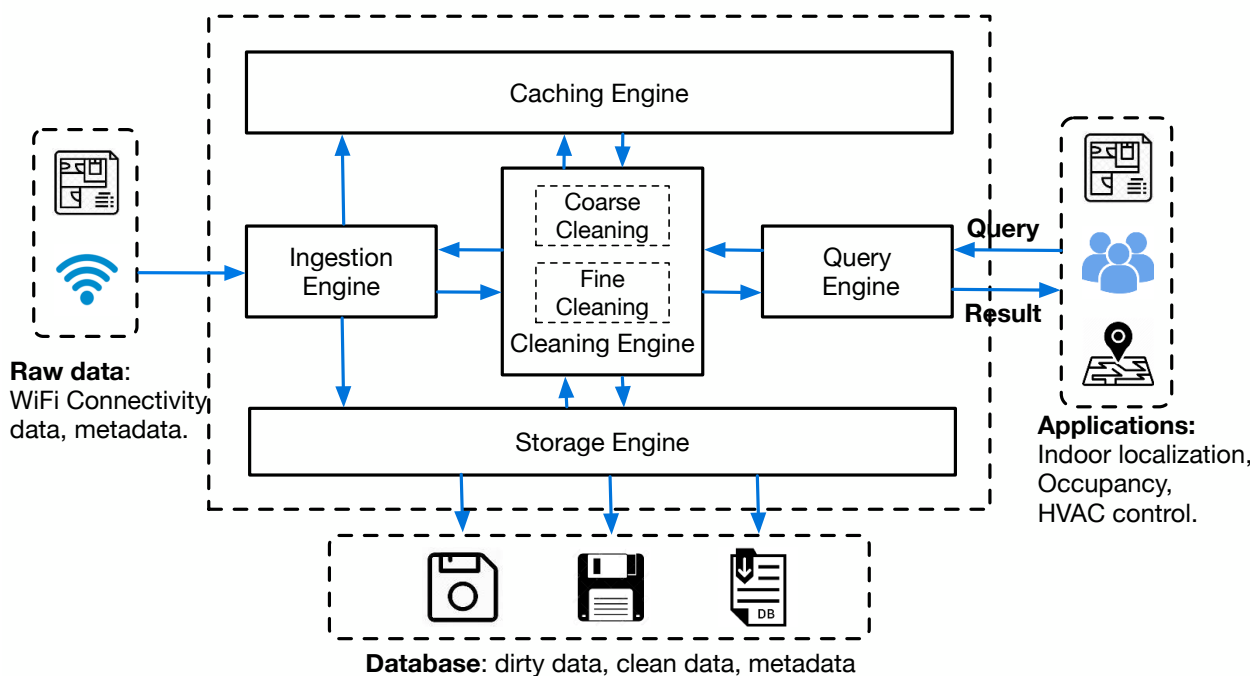


Figure 5.6: Architecture of LOCATER.

LOCATER supports queries $Q = (d_i, t_q)$ that request the location of device d_i at time t_q, where t_q could be the current time (e.g., for real-time tracking/personalized services) or a past timestamp (e.g., for historical analysis). Given Q, LOCATER's *cleaning engine* determines if t_q falls in a gap. If so, it executes its coarse-grained localization (Section 5.3). If at t_q, d_i was inside the building, the cleaning engine performs the fine-grained localization (Section 5.4). Given a query with associated time t_q, LOCATER uses a subset of historical data, such as X days prior to t_q, to learn both room and group affinities. We explore the impact of the amount of historical data used on the accuracy of the model learned in Section 5.6.

Scaling LOCATER. The cleaning engine computes room and group affinities which require time-consuming processing of historic data. Algorithm 6 iteratively performs such computations for each neighbor device of the queried device. In deployments with large WiFi infrastructure and a number of users, this might involve processing large sets of connectivity events which can be a challenge if the applications expect real-time answers. LOCATER caches the computations performed to answer queries and leverages this information to answer subsequent queries. Such cached information constitutes what we will refer to as a

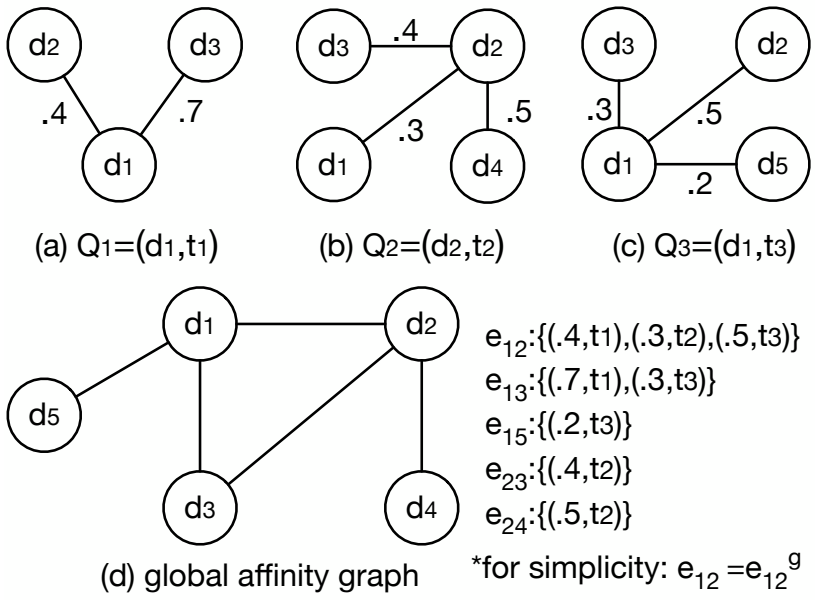


Figure 5.7: Generation of global affinity graph (d) from local affinity graphs (a,b,c).

global affinity graph $\mathcal{G}^g = (\mathcal{V}^g, \mathcal{E}^g)$, where nodes correspond to devices and edges correspond to pairwise device affinities. Given a query $Q = (d_i, t_q)$, LOCATER uses the global affinity graph $\mathcal{G}^g$ to determine the appropriate order in which neighbor devices to d_i have to be processed. Intuitively, devices with a higher device affinity with respect to d_i have a higher impact on the computation of the fine-grained location of d_i. For example, a device that is usually collocated with d_i will provide more information about d_i's location than a device that just appeared in the dataset. We empirically show in our experiments that processing neighbor devices in decreasing order of device affinity instead of a random order makes the cleaning algorithm converge much faster.

(1) Building the local affinity graph. The affinities computed in Section 5.4 can be viewed as a graph, which we refer to as *local affinity graph* $\mathcal{G}^l = (\mathcal{V}^l, \mathcal{E}^l)$, where $\mathcal{V}^l = \bar{D}_n \cup d_i$. In this time-dependent local affinity graph, each device in $\bar{D}_n$, as well as the queried device d_i, are nodes and the edges represent their affinity. Let $e^l_{ab} \in \mathcal{E}^l$ be an edge between nodes d_a and d_b and $w(e^l_{ab}, t_q)$ be its weight measuring the probability that d_a and d_b are in the same room at time t_q. The value of $w(e^l_{ab}, t_q)$ is computed based on Algorithm 6 as $w(e^l_{ab}, t_q) = \frac{\sum_{r_j \in R(g_x)} \alpha(\{d_a, d_b\}, r_j, t_q)}{|R(g_x)|}$.

(2) Building the global affinity graph. After generating a local affinity graph for d_i at time t_q, this information is used to update the global affinity graph. We will illustrate the process using Figure 5.7. Given the current global affinity graph $\mathcal{G}^g = (\mathcal{V}^g, \mathcal{E}^g)$ and a local affinity graph $\mathcal{G}^l = (\mathcal{V}^l, \mathcal{E}^l)$, the global affinity graph $\mathcal{G}^g = (\hat{\mathcal{V}}^g, \hat{\mathcal{E}}^g)$ is updated by setting $\hat{\mathcal{V}}^g = \mathcal{V}^g \cup \mathcal{V}^l$ and $\hat{\mathcal{E}}^g = \mathcal{E}^g \cup \mathcal{E}^l$. Note that, as the affinity graphs are time-dependent, in the global affinity graph we associate each edge included from an affinity graph with its timestamp t_q along with its weight. Hence, in the global affinity graph, the edge in between two nodes is a vector that stores the weight-timestamp pairs associated with different local affinity graphs $v^g_{ab} = \{(w(e^l_{ab}), t_1), ..., (w(e^l_{ab}), t_n)\}$. When merging the edge set, we merge the corresponding vectors – i.e., $v^g_{ab} = v^g_{ab} \cup w(e^l_{ab}, t_q)$ for every $e^g_{ab} \in \mathcal{E}^g$. For instance, in the global affinity graph in Figure 5.7(d), which has been constructed from three different local affinity graphs in Figure 5.7(a),(b),(c), the edge that connects nodes d_1 and d_2 has the weight-timestamp values extracted from each local affinity graph $(.4, t_1), (.3, t_2), (.5, t_3)$. To control the size of the global affinity graph, we could delete past affinities stored in the graph $(w(e^l_{ab}), t_i)$, $\tau - t_i > T_s$, where τ is the current time and T_s is a threshold defined by users, e.g., 3 months.

(3) Using the global affinity graph. When a new query $Q = (d_i, t_q)$ is posed, our goal is to identify the neighbor devices that share high affinities with d_i and use them to compute the location of d_i using Algorithm 6. Given the set D_n of devices that are neighbors to d_i at time t_q, we compute the affinity between d_i and each device $d_k \in D_n$, denoted by $w(e^g_{ik}, t_q)$, using the global affinity graph. As each edge in the global affinity graph contains a vector of affinities with respect to time, we compute the affinity by assigning a higher value to those instances that are closer to the query time t_q as follows: $w(e^g_{ik}, t_q) = \sum_{j=1}^{j=n} l_j w(e^l_{ik}, t_j)$, where l_j follows a normal distribution, $\mu = t_q$ and $\sigma^2 = 1$ that is normalized. Finally, we create a new set of neighbor devices $\mathcal{N}^g(d_i)$ and include each device $d_k \in D_n$ in descending order of the computed affinity $w(e^g_{ik}, t_q)$. This new set replaces D_n in Algorithm 6. Thus, the algorithm processes devices in descending order of the affinity in the global affinity graph.

5.6 Experimental Evaluation

We implemented a prototype of LOCATER and performed experiments to test its performance in terms of the quality of the cleaned data, efficiency, and scalability. The experiments were executed in an 8 GB, 2 GHz Quad-Core Intel Core i7 machine with a real dataset as well as a synthetic one. We refer to the implementation of LOCATER's fine-grained algorithms based on the independent and dependent assumptions as I-FINE and D-FINE. Correspondingly, we will refer to the system using those algorithms as I-LOCATER and D-LOCATER, respectively.

5.6.1 Experimental Setup

Dataset. We use connectivity data captured by the TIPPERS system [86] in Donald Bren Hall (DBH) building at University of California, Irvine, with 64 WiFi APs, 300+ rooms including classrooms, offices, conference rooms, etc., and average daily occupancy of about 3,000. On average, each WiFi AP covers 11 rooms. The dataset, *DBH-WIFI*, contains 10 months of data, from Sep. 3rd, 2018 to July 8th, comprising $38,670,714$ connectivity events for $66,717$ different devices.

Ground truth. We collect fine-grained locations of 28 distinct individuals as the ground truth. We asked 9 participants to log their daily activity within the building for a week. Such activity includes the rooms where they were located and how much time they spent in them. The participants filled in comprehensive and precise logs of their activity amounting to 422 hours in total. We also selected three cameras in the building that cover different types of spaces, i.e., faculty offices area, student offices area, and lounge space. We manually reviewed the camera footage to identify individuals in it. The area covered is in the portion of the DBH building, so we identified 26 individuals and 7 of them were also participants in

the daily activity logging. We requested the identified individuals for their MAC address. If a person p with MAC address m was observed to enter a room r at time t_1 and left the room at time t_2, we created an entry in our ground truth locating m in room r during the interval (t_1, t_2).

Queries. We generated a set of $10,028$ queries, denoted by $\mathcal{Q}$, related to individuals in the ground truth. There are a total of $3,129$ queries for participants that logged their activities and $6,899$ queries for individuals detected in the camera images. The number of queries per individual is approximately the same, as far as differences in the labeled elements per user allow it.

Baselines. The traditional indoor localization algorithms are either based on active localization or passive localization using information such as signal strength maps. Hence, we defined two baselines used in practice for the coarse and fine-grained localization based on connectivity logs and background information. The baselines are defined as follows: *Baseline1* and *Baseline2* use *Coarse-Baseline* for coarse localization and for fine-grained localization they use *Fine-Baseline1* and *Fine-Baseline2*, respectively. In *Coarse-Baseline*, the device is considered outside if the duration of a gap is at least one hour, otherwise, the device is inside and the predicted region is the same as the last known region. *Fine-Baseline1* selects the predicted room randomly from the set of candidates in the region whereas *Fine-Baseline2* selects the room associated with the user based on metadata, such as his/her office.

Quality metric. LOCATER can be viewed as a multi-class classifier whose classes correspond to all the rooms and a label for outside the building. We use the commonly used *accuracy* metric [105], defined next, as the measure of quality.[11] Let $\mathcal{Q}$ be the set of queries, $\mathcal{Q}_{out}, \mathcal{Q}_{region}, \mathcal{Q}_{room}$ be the subset of queries for which LOCATER returns cor-

[11] Accuracy, as defined in the chapter, is exactly the same as other micro-metrics such as micro-precision, recall, and F-measure [101]. Micro-level metrics are, often, more reflective of the overall quality of the multi-level classifier, such as LOCATER, when the query dataset used for testing is biased towards some classes.

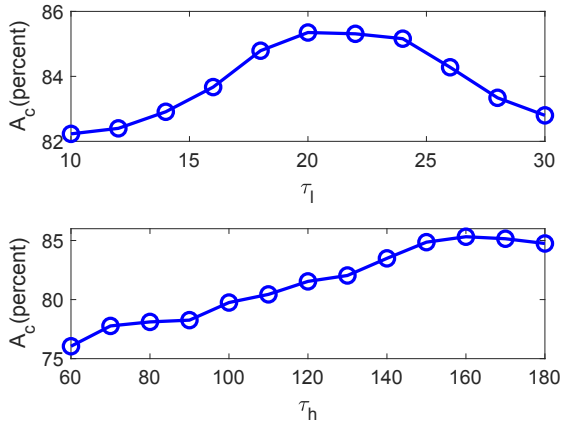


Figure 5.8: Thresholds tuning.

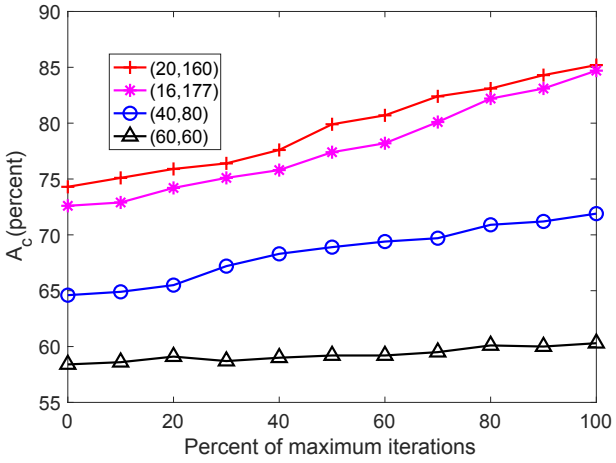


Figure 5.9: Iteration.

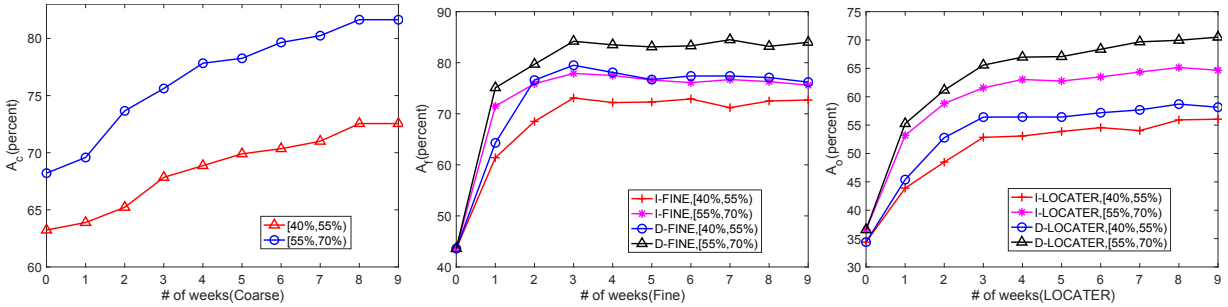


Figure 5.10: Impact of historical data used on accuracy.

rectly the device's location as being *outside*, in a specific region, and a specific room, respectively. Accuracy of the coarse-grained algorithm can then be measured as: $A_c = (|\mathcal{Q}_{out}| + |\mathcal{Q}_{region}|)/|\mathcal{Q}|$. Likewise, for fine-grained and overall algorithms, accuracy corresponds to $A_f = |\mathcal{Q}_{room}|/|\mathcal{Q}_{region}|$, and $A_o = (|\mathcal{Q}_{room}| + |\mathcal{Q}_{out}|)/|\mathcal{Q}|$, respectively.

5.6.2 Evaluation

Accuracy on DBH-WIFI Dataset

We first test the performance of LOCATER, in terms of accuracy, for the DBH-WIFI dataset. As LOCATER exploits the notion of recurring patterns of movement/usage of the space, we analyze the performance w.r.t. the level of *predictability* of different user profiles. We consider

the fact that some people spend most of their time in the building in the same room such as their offices as a sign of predictable behavior. We can consider this as their "preferred room". We group individuals in the dataset into 4 classes based on the percentage of time they spend in their preferred room: $[40, 55)$, $[55, 70)$, $[70, 85)$ and $[85, 100)$, where $[40, 55)$ means that the user spent 40-55 percent of the time in that room. In other words, no user in the ground truth data spent less than 40% of his/her time in a specific room.

Impact of thresholds in coarse localization. The coarse-level localization algorithm depends upon two thresholds: τ_l and τ_h. We use k-fold cross-validation with $k = 10$ to tune them. We vary τ_l's value from 10 to 30 minutes and τ_h's value from 60 to 180 minutes. We fix $\tau_h = 180$ when running experiments for τ_l and fix $\tau_l = 20$ when running experiments for τ_h. From Figure 5.8 we observe that, with the increasing of τ_l, the accuracy increases first and then slightly decreases after it peaks at $\tau_l = 20$. For τ_h, when it increases, accuracy gradually increases and levels off when τ_h is beyond 170. We also test the parameters computed by a confidence interval, which are $\tau_l = 16.4$ and $\tau_h = 177.3$. The accuracy achieved by this parameter setting is 84.7%, which is close to the best accuracy (85.2%) achieved by parameters tuned based on cross-validation.

Iterative classification for coarse localization We test the robustness of the iterative classification method. We vary the quality of the initial decisions of the heuristic strategy without iterations by setting the parameters (τ_l, τ_h) to $(20, 160)$, $(16, 177)$, $(40, 80)$, and $(60, 60)$. For each query, we terminate the coarse localization algorithm at different stages as a percentage of the maximum iterations the algorithm would perform, and report A_c in Figure 5.9. We observe that for a high quality initial decision, the iterative classification improves the accuracy significantly with an increasing number of iterations. Also, for those relatively bad initial decisions with the initial accuracy 58% and 65%, the improvement achieved by the iterative classification is small but it always increases. We also show that for the parameters decided by the Gaussian confidence interval method i.e., $(16, 177)$, which

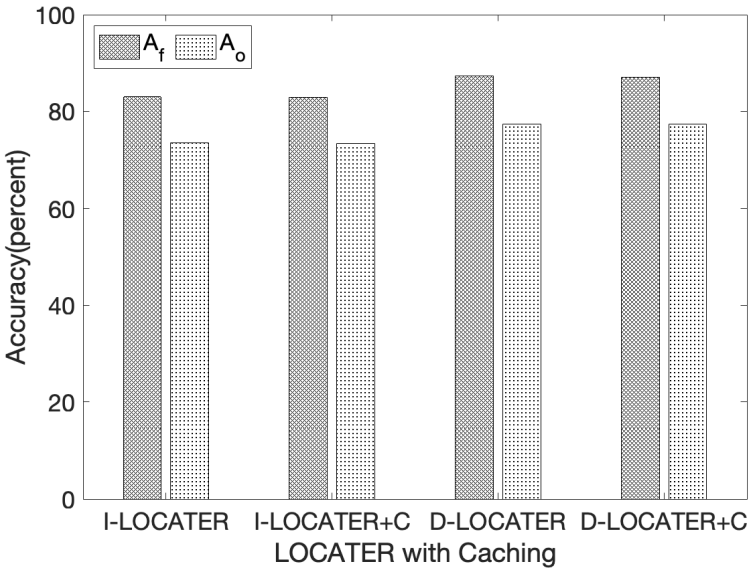


Figure 5.11: Caching accuracy.

does not rely on the ground truth data, the iterative classification method works very well.

Impact of weights of room affinity. We examine the impact of weights used in computing room affinity, w^{pf}, w^{pb}, w^{pr}. We report the fine accuracy of the four weight combinations satisfying the rules defined in that section: $C1 = \{0.7, 0.2, 0.1\}$, $C2 = \{0.6, 0.3, 0.1\}$, $C3 = \{0.5, 0.3, 0.2\}$, and $C4 = \{0.5, 0.4, 0.1\}$. For C_1, C_2, C_3, C_4, A_f of I-FINE is 81.8, 83.4, 82.3, 82.4, and A_f of D-FINE is 86.1, 87.5, 86.6 and 86.4, respectively. We observe that all the combinations for both I-FINE and D-FINE obtain a similar accuracy with $C2$ achieving a slightly higher accuracy. Hence, the algorithm is not too sensitive to the weight distributions in this test. Also, D-FINE outperforms I-FINE by 4.6% on average.

Impact of historical data. We use historical data to train the models in the coarse algorithm and to learn the affinities in the fine algorithm. We explored how the amount of historical data used affects the performance of LOCATER. We report the coarse, fine, and overall accuracy for the [40,55)% and [55,70)% predictability groups, in Figure 5.10(a), Figure 5.10(b), and Figure 5.10(c), respectively. The graphs plot the accuracy of the algorithm with increasing amounts of historical data, from no data at all up to 9 weeks of data. The accuracy of the coarse-grained algorithm increases with increasing amount of historical data

and it reaches a plateau when 8 weeks of data are used. The reason is that the iterative classification algorithm used to train the model becomes more generalized the more data is used for the training. The performance of the fine-grained algorithm is poor when no historical data is used since this effectively means selecting the room just based on its type. However, when just one week of historical data is used the performance almost doubles. The accuracy keeps increasing with an increasing number of weeks of data though the plateau is reached at 3 weeks. The results show that the kind of affinities computed by the algorithm are temporally localized. The overall performance of the system follows a similar pattern. With no data, mistakes made by the fine-grained localization algorithm penalize the overall performance. With increasing amounts of historical data, the performance increases due to the coarse-grained algorithm labeling gaps more correctly. In all the graphs, the performance of the overall system and its algorithms increases with the increasing level of predictability of users.

Robustness of LOCATER with respect to room affinity. LOCATER's approach to disambiguating locations exploits the prior probability of individuals being in specific rooms (room affinity). In this experiment, we explore the robustness of LOCATER when we only know the prior for a smaller percentage of people. We randomly select users for whom we compute and associate a room affinity to each candidate room based on historical data and room metadata. For the rest, we consider a uniform room affinity for all the candidate rooms. We repeat the experiment 5 times and report the average fine accuracy: A_f. We set the percentage of users with refined room affinities to 0%, 25%, 50%, 75%, and 100%, and the corresponding A_f is 6.2, 57.1, 71.3, 81.1, 87.1. We observe that the accuracy is poor when equally distributed affinity is considered for all users. When a refined room affinity is computed for a small portion of users (25%), the accuracy increases significantly to 57.1. Increasing the number of users with refined room affinity makes the accuracy converge to 87.1. Thus, we expect LOCATER to work very well in scenarios where the pattern of building usage and priors for a significant portion of the occupants is predictable.

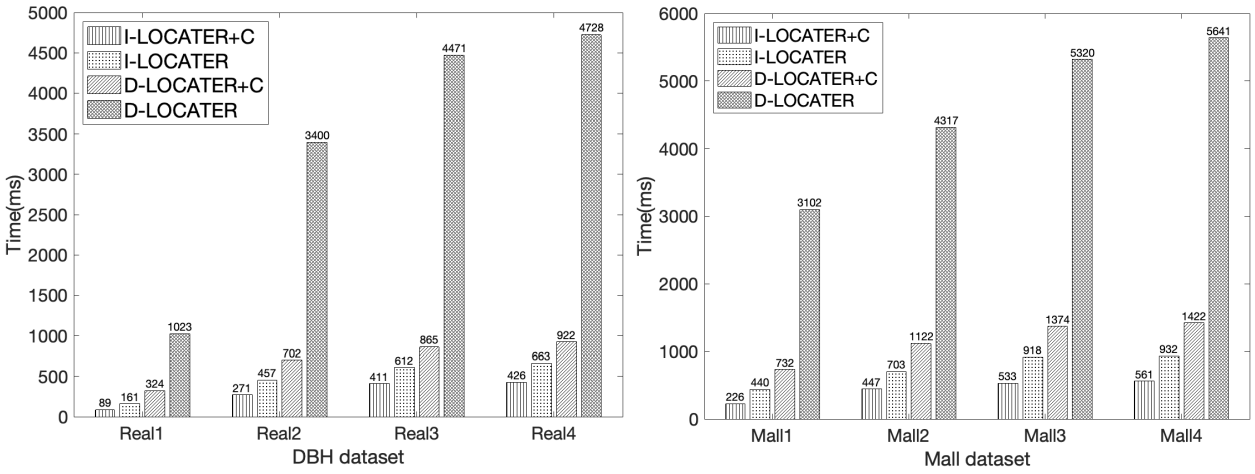


Figure 5.12: Scalability testing

Impact of caching. We examine how the fine-grained algorithm's caching technique (see Section 5.4) affects the accuracy of the system. We compute the accuracy of both I-LOCATER and D-LOCATER compared to their counterparts using caching I-LOCATER+C and D-LOCATER+C. Figure 5.11 plots the overall accuracy of the system averaged for all the tested users. We observe that adding caching incurs a reduction of the accuracy from 5%-10%, which does not significantly affect the performance. This means that the device processing order generated by the caching technique maintains a good accuracy while decreasing the cleaning time (see Section 5.6.2).

Probability distribution of results. We show the probability distribution computed by LOCATER for each of the rooms in the set of candidate rooms for a given query. In particular, we plot the highest probability value associated with any room (Pr_h), the difference between the highest and second highest probability (ΔPr), and the summation of the remaining probabilities ($\sum_r$). We report the statistics over all the queries in Table 5.2. We observe a long tail distribution for the set of different rooms output by LOCATER. In particular, there are 69% queries whose highest probability is in $[.4, .6)$, 43% queries whose difference of the highest and second highest probability is $[.2, .3)$ and 51% queries where the sum of top-2 probabilities is greater than .6.

Table 5.2: Probability distribution of rooms.

Pr_h	$[0, .2)$	$[.2, .4)$	**[0.4,.6)**	$[.6, .8)$	$[.8, 1)$
Percent of queries	0	19	**69**	12	0
ΔPr	$[0, .1)$	$[.1, .2)$	**[.2,.3)**	$[.3, .4)$	$[.4, .5)$
Percent of queries	4	17	**43**	20	16
$\sum_r$	$[0, .2)$	**[.2,.4)**	$[.4, .6)$	$[.6, .8)$	$[.8, 1)$
Percent of queries	32	**51**	15	2	0

Table 5.3: Accuracy for different predictability groups.

$A_c\|A_f\|A_o$	$[40, 55)$	$[55, 70)$	$[70, 85)$	$[85, 100)$	$\mathcal{Q}$
Baseline1	56|10|24	63|8.0|25	67|10|26	73|12|28	**64|10|26**
Baseline2	62|45|39	67|63|50	69|75|57	76|93|72	**68|67|53**
I-LOCATER	76|72|61	83|78|70	87|84|77	93|87|84	**85|83|75**
D-LOCATER	76|77|63	83|82|72	87|87|79	93|92|88	**85|87|79**

Comparison with baselines. We compare the accuracy of LOCATER vs. baselines for different predictability groups, and the average of accuracy for all people as $\mathcal{Q}$ in Table 5.3, where each cell shows the rounded-up values for A_c, A_f, and A_o. We observe that both I-LOCATER and D-LOCATER significantly outperform *Baseline1* regardless of the predictability level of people. This is due to the criteria to select the room in which the user is located when performing fine-grained localization. Deciding this at random works sometimes in situations where the AP covers a small set of large rooms but incurs errors in situations where an AP covers a large set of rooms. For example, in our dataset up to 11 rooms are covered by the same AP. *Baseline2* uses a strategy where this decision is made based on selecting the space where the user spends most of his/her time if that space is in the region where the user has been localized. This strategy only works well with very predictable people. Hence, LOCATER outperforms *Baseline2* in every situation except for the highest predictable group where *Baseline2* obtains a slightly better accuracy. The accuracy of D-LOCATER is consistently higher than I-LOCATER. Both of them perform significantly better than the baselines except for the situation highlighted before.

Macro results. We report macro precision, recall, and F-1 measure for *Baseline1*, *Baseline2*, I-LOCATER, and D-LOCATER, respectively. Macro precision (recall) is defined as

Table 5.4: Macro results of LOCATER for different methods.

	Precision	Recall	F-1
Baseline1	21.8	33.5	26.4
Baseline2	58.7	46.2	51.7
I-LOCATER	78.2	73.7	76.7
D-LOCATER	81.3	76.4	78.8

the average of precision (recall) of all classes. As shown in Table 5.4, LOCATER achieved significantly better precision and recall than baselines and the performance of D-LOCATER is slightly better than I-LOCATER.

Efficiency and Scalability

We first examine the efficiency of LOCATER on the DBH-WIFI dataset. We report the average time per query when the system uses or does not the stopping conditions described in Section 5.4. With the stop condition, LOCATER takes 563 milliseconds while it takes 2,103 milliseconds without it. Without stop conditions, I-LOCATER has to process all neighbor devices, whereas with the stop conditions the early stop brings a considerable improvement in the execution time.

We conduct scalability experiments both on real and synthetic data. We randomly select a *subspace* of a building by controlling its size using as parameters the number of WiFi APs, rooms, and devices. For the real dataset, DBH-WIFI, we extract four datasets, $Real_1$, ..., $Real_4$. The number of WiFi APs for these four datasets are 10, 30, 50, 64, and the number of rooms is 46, 152, 253, and 303, and the number of devices is 41,343, 60,885, 63,343, 64,717, respectively. To test the scalability of LOCATER on various scenarios, we generated four synthetic datasets simulating the following environments, which we list in order of increasing predictability: airport, mall, university, and office. For each of them, we used a real blueprint, Santa Ana's airport for the first scenario, and created types of people, such as TSA staff, passengers, etc., and events they attend such as security checks,

and boarding flights, based on our observations. We report the running time of LOCATER on the Mall scenario. In particular, we generated four synthetic datasets, $Mall_1$, ..., $Mall_4$.

We plot the average time cost per query on the DBH-WIFI and the Mall data sets in Figure 5.12. The main observations from the results on both datasets are: 1) The caching strategy decreases the computation time of D-LOCATER significantly, and D-LOCATER performs slightly better than I-LOCATER; 2) With the caching technique LOCATER has great scalability when the size of space increases to large scale to support a near-real-time query response, in particular, around 1 second for D-LOCATER and half a second for I-LOCATER.

Experimental Summary.

We below summarize the main outcomes of the experiments in PLAQUE and what we learned.

- The accuracy of LOCATER to predict a coarse-level (region) and fine-level (room) location is 85% and 87% on average in a real DBH-WiFi data set when using 3 weeks of historical data.
- LOCATER performs better for those more predictable groups with an accuracy over 90% on average, while LOCATER performs less accurately for very unpredictable users with mostly random trajectories.
- LOCATER is able to predicate a coarse-level (region) and fine-level (room) location within 73 milliseconds and 563 milliseconds on average.
- LOCATER predicts the locations of the users based on their historical trajectories as well as the space metadata without the installation of the new hardware or software. LOCATER is mostly suitable for location-based applications that need more aggregated information derived from the locations of the individuals, such as occupancy, or

the semantic locations instead of pin-point physical locations.

5.7 Conclusion

In this chapter, we propose LOCATER which cleans existing WiFi connectivity datasets to perform semantic localization of individuals. The key benefit of LOCATER is that it: 1) Leverages existing WiFi infrastructure without requiring the deployment of any additional hardware, such as monitors typically used in passive localization; 2) Does not require explicit cooperation of people, like active indoor localization approaches. Instead, LOCATER leverages historical connectivity data to resolve coarse and fine locations of devices by cleaning connectivity data. Our experiments on both real and synthetic data show the effectiveness and scalability of LOCATER. Optimizations made LOCATER achieve near real-time response.

LOCATER's usage of WiFi events, even though it does not capture any new data other than what WiFi networks already capture, still raises privacy concerns since such data is used for a purpose other than providing networking. Privacy concerns that arise and mechanisms to mitigate them, are outside the scope of this work and are discussed in [36, 46, 92]. For deployments of LOCATER, we advocate performing data collection based on informed consent allowing people to opt-out of location services if they choose to.

Chapter 6

Case Study

6.1 Introduction

In this chapter, we describe a case study based on the smart space to highlight the contributions of all three technical parts of our work in this thesis, ZIP, PLAQUE, and LOCATER.

Recall that in the previous chapters, we explored two techniques ZIP and PLAQUE to optimize query processing for data-intensive computations. In particular, ZIP explores a lazy approach to impute minimal missing data to answer a given query, and PLAQUE identifies a set of opportunities to learn new predicates at query runtime to accelerate query execution. In Chapter 5, we further introduced an indoor localization technology, LOCATER, to find the indoor location of an individual at any given time instance based on data cleaning technologies using WiFi connectivity data. Although LOCATER provides several unique benefits, e.g., LOCATER is passive, free, and accurate, using LOCATER to predict indoor location is not cheap, and it takes around 0.4 seconds on average to impute one missing location.

macAddress	timeStamp	WiFiAP
9867...	2019-04-25 15:03:02	3142-clwa-2099
9867...	2019-04-25 15:07:13	3142-clwa-2059
9867...	2019-04-25 15:09:22	3142-clwa-2059

Table 6.1: Raw WiFi Connectivity Data.

In this case study, first, we describe two applications built using LOCATER technology, occupancy computation and, contact tracing. Both applications result in SQL queries as we will show in Section 6.4 and Section 6.3. Next, we show how ZIP and PLAQUE make both contact tracing and occupancy applications execute at interactive rates in Section 6.5. We start with specifying the setup of the case study on the campus of University of California, Irvine (UCI).

6.2 Setup

The first data set we use in this case study is the real WiFi connectivity events data (see Chapter 5 for the detailed description), which is collected by OIT (Office of Information Technology) at the University of California, Irvine. We store the WiFi connectivity data in the TIPPERS database system [86].

WiFi connectivity data, as shown in Table 6.1, consists of the following three key fields, i.e., the mac address of a device that uniquely identifies a device, the timestamp of connection, and the identifier of a WiFi Access Point (WiFiAP). In the first tuple in Table 6.1, a device with a mac address starting with 9867 connects to WiFiAP 3142-clwa-2099 at the time stamp 2019-04-05 15:03:02. The mac address of a hand-held device, such as a mobile phone, serves as a surrogate for the owner of the device.

Besides the raw WiFi connectivity data, we also create a User table as shown in Table 6.2 and a Space table as shown in Table 6.3. The User table stores the user profiles, such as name, the mac address of the device they registered in TIPPERS, and the type of user such

Name	macAddress	type
Mike	9867...	faculty
Robert	3b26...	graduate
John	7145...	faculty

Table 6.2: User.

spaceID	spaceName	spaceType	area
23	3142-clwa-2099	region	60
26	3142-clwa-2059	region	60
113	2065	room	25
116	2099	room	35
201	Donald Bren Hall	Building	450

Table 6.3: Space.

macAddress	timeStamp	regionLocation	roomLocation
9867...	2019-04-26 15:03:02	23	NULL
9867...	2019-04-26 15:07:13	26	NULL
9867...	2019-04-26 15:09:22	26	NULL

Table 6.4: Semantic WiFi Connectivity Data.

as graduate or faculty. The Space table stores the metadata of the spaces with different granularities. For instance, in the first tuple in Table 6.3, a location with spaceID 23 is a region whose name is 3142-clwa-2099 and has a physical area of 60 m^2.

With the Space table, we transform the raw WiFi connectivity data received from OIT into a semantic WiFi connectivity table as shown in Table 6.4. Each connectivity log in the semantic WiFi connectivity table has a region location (i.e., *regionLocation*) and a room location (i.e., *roomLocation*), where the region location is the area/location covered by the corresponding WiFiAP in the raw connectivity table, and initially, the room locations for all connectivity events are missing represented as NULL values in the semantic WiFi connectivity table. Note that the region location and room location correspond to the spaceIDs of the corresponding locations stored in the Space table. For instance, the region location in the first tuple in the semantic WiFi connectivity table is 23 corresponding to the region 3142-clwa-2099, which is the WiFiAP covering this region location.

The last data set we use in the case study is the Presence table (see Chapter 5 for a detailed description), as shown in Table 6.5, which contains the region and room locations of people during different time intervals. For instance, in the first tuple in the Presence table in Table 6.5, a device with a mac address starting with 9867 is in the region covered by WiFiAP

macAddress	startTime	endTime	regionLocation	roomLocation
9867...	2019-04-26 15:02:02	2019-04-26 15:04:02	23	NULL
9867...	2019-04-26 15:04:02	2019-04-26 15:06:13	NULL	NULL
9867...	2019-04-26 15:06:13	2019-04-26 15:08:13	26	NULL
9867...	2019-04-26 15:08:13	2019-04-26 15:08:22	NULL	NULL
9867...	2019-04-26 15:08:22	2019-04-26 15:10:22	26	NULL

Table 6.5: Presence.

3142-clwa-2099 with a spaceID as 23 in the Space table from 2019-04-05 15:02:02 to 2019-04-05 15:04:02, but its room location is missing.

Chapter 5 describes how to construct the Presence table from the semantic WiFi connectivity data in detail. We briefly summarize the construction of the Presence table below to keep this chapter self-contained. First, for each connectivity event in the semantic WiFi connectivity table at timestamp t, the Presence table creates a new tuple storing the information that the same device is in the region covered by the corresponding WiFiAP one minute before and after the connection time stamp t. For instance, the first, third, and fifth tuples in the Presence table in Table 6.5 are transformed from the first three tuples in the WiFi connectivity table, respectively. For the same device, the gap between two consecutive tuples in the Presence table will correspond to a new tuple whose region and room location are missing. For example, the second and the fourth tuples in the Presence table are the tuples representing such gaps.

We will show later in building contact tracing and occupancy applications, we need to write SQL queries on the Presence table, semantic WiFi connectivity table, User table, and Space table to realize the key functionalities in both applications. In the SQL queries, LOCATER is used as a User-Defined-Function (UDF) to impute any missing location if required.

We formally specify the LOCATER UDF below. Let LOCATER(loc,mac,t) be the LOCATER UDF, which takes as input a location *loc*, the mac address of a device *mac* as well as a timestamp t. If the location *loc* is not missing, then LOCATER(loc,mac,t) simply returns *loc* as the predicted location. Otherwise, LOCATER(loc,mac,t) will call LOCATER technology de-

scribed in Chapter 5 to predict the location of the device with mac address *mac* at timestamp *t*.

6.3 Occupancy

Occupancy, i.e., the number of people in a certain location at a given time, is essential for many applications, such as building energy control [90] and space planning [98]. For example, Agarwal, et al. [22] observed that the heating, ventilating, and air-conditioning (HVAC) energy consumption was reduced between 10% to 15% based on the occupancy detection in offices. Leephakpreeda, et al. [90] proposed an occupancy-based lighting control, and showed that the energy consumption of the application can be reduced between 35% to 75%. Additionally, occupancy is also a key mitigation strategy for COVID-19 [80] as users could use occupancy-based applications to be aware of the people's density inside buildings to reduce the chance of exposure.

We build the occupancy application by leveraging the LOCATER technology described in Chapter 5. LOCATER is able to predict a semantic location, such as floor/region/room, of an individual at a given time instance. However, knowing the location of each individual is not enough to generate an accurate occupancy as there are several challenges in estimating the occupancy based on WiFi connectivity data. First of all, one person often has multiple devices, such as a phone, laptop, and iPad, so simply counting the number of appearances of devices in the WiFi connectivity data would lead to over-counting of the actual occupancy. Second, not all connectivity logs are generated by *people*, and could instead be generated by *static devices*, such as a printer, or computers in a lab/office. Finally, many passer-by devices that connected to some WiFi AP but left the corresponding region immediately should not be counted in the occupancy of the region of interest. Detecting passer-by devices from WiFi connectivity data is also an important factor to ensure an accurate occupancy estimation.

```
SELECT DISTINCT COUNT(WiFi.mac)
FROM WiFi
WHERE WiFi.timeStamp > st AND WiFi.timeStamp < et
AND LOCATER(WiFi.regionLocation,WiFi.mac,WiFi.timeStamp) = loc
```

Figure 6.1: Occupancy Query

We describe the logic of the basic occupancy application in Section 6.3.1. We improve the approach described in Section 6.3.1 to develop a more accurate model of occupancy in Section 6.3.2. Finally, we show a use case scenario for occupancy application in Section 6.3.3.

6.3.1 Basic Occupancy Application Logic

The occupancy application takes the semantic WiFi connectivity events as the input as shown in Table 6.4, wherein each WiFi connectivity event, a device with mac address mac_i is associated with a region location and room location at time t_i.

To ask for the occupancy in a location in the given time range inside a building, one can issue a point query $Q = (st, et, loc)$, where st, et represent the start and end time stamps, and loc corresponds to the location of interest, such as region or room. The output of the occupancy application will be an occupancy count, i.e., number of people, in location loc in the given time interval (st, et).

In order to determine the occupancy of a location loc in the time interval (st, et), the occupancy application generates the *occupancy query* in Figure 6.1 in the semantic WiFi connectivity table (WiFi table for short) as shown in Table 6.4.

In the occupancy query, if the given location loc is room, the LOCATER UDF would be LOCATER(WiFi.roomLocation,WiFi.mac,WiFi.timeStamp).

The occupancy application allows more complex queries to be asked. For instance, if we wish to know the count of people of a given type, such as faculty or graduate, we can do so by

augmenting the above SQL query with additional predicates, such as User.type = 'graduate'. In general, the occupancy application allows the predicates on any of the fields listed in the User Table 6.2 or Space Table 6.3.

6.3.2 Improvements

We note that the answer to the occupancy SQL query in Figure 6.1 can be inaccurate when comparing it with the ground truth. First, the connectivity logs could be generated from a person who passes by the location specified in the occupancy query, without entering the location. Such a person should not be included in the occupancy counts. For example, a person who passes by a building B and is outside of the building should not be counted as the occupant of B. We develop an algorithm to detect and eliminate passer-by devices. Second, a user may carry more than one device which connects to the same WiFi AP, which results in over-counting of occupancy. For instance, if one carries both a tablet and a mobile phone which connect to the same WiFi AP, the occupancy count for this individual should be one instead of two. We develop an algorithm to deduplicate such duplicate devices. Third, the environment may contain static devices connecting to the network, which could introduce false positives. For instance, the connected printers should not be counted as occupants. We address this challenge by developing a method to detect static devices.

We next describe our approach to detecting *passer-by devices*, *duplicate devices*, and *static devices*. Each of the algorithm described below has been incorporated as UDFs which can either be invoked during the processing of the occupancy query in Figure 6.1 to determine a more accurate occupancy.

Passer-by Devices

When a person carrying WiFi-equipped devices passes by a region r covered by some WiFi

AP without staying in the region r, such devices are called *passer-by* devices, and they should not be counted into occupancy of the region r. For example, students walking by a building without entering it are false positives for occupancy estimation of the building and sub-regions inside that building.

To compute the occupancy of a location *loc* that is covered by a WiFi AP *ap* during the time interval (st, et), we observe that the WiFi connectivity events associated with *ap* for a passer-by device concentrate on a fairly short time interval with limited numbers. For example, most passer-by devices generate less than 5 connectivity logs associated with *ap* within 1 minute based on our empirical study. We express the logic of the algorithm to detect passer-by devices using ECA rule [17] as [**WHEN**, **IF**, **THEN**].

WHEN: a device d connects to the same WiFi AP of the location *loc* in the occupancy query. Let $E_d = \{e_1, e_2, ..., e_n\}$ be the set of connectivity events of d during the time interval (st, et) in increasing order of the timestamp of the events

IF: $|E_d| < k$ and $|e_n.timeStamp - e_1.timeStamp| < T$

THEN: d is a passer-by device.

The above ECA rule captures the connectivity pattern of a typical passer-by device in our empirical studies, where we set $k = 5$ and $T = 1$ minute, turning out to be an effective filter to detect passer-by devices.

We implement the algorithm to detect passer-by devices as a tabular UDF [11, 10]. Let Passerby(R, WiFi) be a tabular UDF of the removal of the passers-by devices, which takes as input a single-column relation R with a field WiFi.mac and the semantic WiFi relation WiFi. The output of Passerby(R, WiFi) is a single-column relation R' with the same field WiFi.mac, where the device in each tuple in R' is not a passer-by.

To incorporate Passerby(R, WiFi) at query time into the occupancy query in Figure 6.1,

we create a view, Occupancy-NoPasserby, and implement Passerby(R, WiFi) below. When the occupancy query returns a single-column relation with WiFi.mac, for each device d in WiFi.mac, Passerby(R, WiFi) first retrieves all the connectivity tuples associated with d in WiFi relation, and uses the defined ECA rule to determine if d is a passer-by or not. The tuple corresponding to d will be eliminated if d is a passer-by device.

```
CREATE VIEW Occupancy-NoPasserby AS
SELECT Passerby(WiFi.mac,WiFi)
FROM WiFi
WHERE WiFi.timeStamp > st AND WiFi.timeStamp < et
AND LOCATER(WiFi.regionLocation,WiFi.mac,WiFi.timeStamp) = loc
```

Duplicate Device Detection

If one person carries multiple devices connecting to the same WiFi AP, counting the connections for each of such devices would over-count the occupancy. The task of *duplicate device detection* is to determine a set of devices that belong to the same person and remove the duplicates when counting occupancy.

The algorithm of duplicate device detection consists of two phases, the offline phase, and the online phase. In the offline phase, the algorithm learns a *similarity graph* based on the Presence table, wherein each node represents a device and each edge corresponds to the similarity between two devices. As will be clear shortly, similarity is a metric to measure the likelihood of two devices belonging to the same individual. We determine the likelihood by estimating the similarity of the trajectories of two devices over time. In the online phase, given a set of devices D, the algorithm deduplicates the devices based on the similarity graph and returns a set of devices $D' \subseteq D$ such that there do not exist two devices $d_i, d_j \in D'$ belonging to the same user.

Offline Phase. We start by building a similarity graph based on a subset of the Presence table during the time interval (st, et) as shown in Table 6.5. Note that the similarity graph will be updated offline periodically by using the newer data in the Presence table.

Let $G(V, E)$ be the similarity graph, where V is the set of devices represented by their unique mac addresses, and E is a set of edges. Let $e_{ij} \in E$ be the edge between device d_i and d_j. Let $Sim(i, j)$ be the similarity of trajectories between devices d_i and d_j, which is also the weight of edge e_{ij}. The trajectory of device d_i consists of a set of consecutive tuples of device d_i in the Presence table during the time interval (st, et). $Sim(i, j) = \frac{CD(i,j,st,et)}{et-st}$, where $CD(i, j, st, et)$ is the total length of the duration time when devices d_i and d_j are in the same region during time range (st, et). [1] After constructing the similarity graph, we cluster the nodes in the graph using k-means clustering algorithmk [2] with a constraint that the number of nodes in any cluster can be no larger than k. k is the bound of the maximum number of devices a person can carry. [3] In the resulting clusters, the devices in one cluster represent the devices belonging to the same user since they share the highest similarities with each other. Note that if a user u has self-reported and registered all her devices in TIPPERS, we can use this information to form the corresponding cluster in the similarity graph without performing clustering of the devices reported by the user u.

Online Phase. We implement the algorithm of duplicate device detection as a tabular UDF [11, 10], which is executed at the query time. Let Deduplication(R,G) be the tabular UDF of device deduplication. Deduplication(R,G) takes as input a single-column relation R with a field WiFi.mac and the similarity graph G. The output of Deduplication(R,G) is another single-column relation R' with the same field WiFi.mac. For any two tuples r_i, r_j in R', let d_i, d_j be their corresponding attribute value in the field WiFi.mac. For any such devices

[1] We choose region since the region-level localization by LOCATER is much cheaper than the room-level localization, and thus enables efficient computation of similarity.

[2] Any other clustering algorithm such as DBSCAN [31] would suffice.

[3] In our implementation, we set k as 5, which turns out to be effective enough to enable accurate duplicate device detection.

d_i, d_j, they are not in the same cluster in the similarity graph G. When there exist multiple devices in one cluster, Deduplication(R,G) picks a random representative of the cluster among these devices and returns it.

To further remove the duplicate devices from the occupancy count, we create a view, Occupancy-NoPasserby-NoDuplication based on the view Occupancy-NoPasserby below.

```
CREATE VIEW Occupancy-NoPasserby-NoDuplication AS
SELECT Deduplication(WiFi.mac,G)
FROM Occupancy-NoPasserby
```

Static devices

The connectivity logs generated by the static devices, such as printers and computers in a computer lab/personal offices, should not be counted as occupancy. Thus we need to detect static devices based on their connectivity patterns. We developed a simple offline strategy to identify and maintain a list of static devices. First, we identify a set of devices D_1, each of which predominately connects to a *unique* WiFi AP for a relatively long time. They may sporadically connect to the neighboring WiFi APs, and connect back to the WiFi AP that is mostly often connected. In contrast, mobile devices carried by people tend to connect a larger number of WiFi APs in a larger area more frequently. Second, we collect a set of devices D_2 that constantly generate connectivity logs at night, such as from 3:00 am to 6:00 am as we set in our implementation since such connectivity events would not possibly be continuously generated by a mobile device carried by a person for a long time. Let the list of static devices we identified be $SD = D_1 \cap D_2$, which is periodically updated.

Let StaticDevices(R,SD) be the tabular UDF of the removal of static devices. StaticDevices(R,SD) takes as input a single-column relation R with a field WiFi.mac and the maintained static devices SD, and it returns another single-column relation R' by removing the

tuples from R whose WiFi.mac is in the static devices SD. Note that the step of removing static devices can be done prior to query execution or at query time since it is cheap. To incorporate the StaticDevices(R,SD), we create a view Occupancy-NoPasserby-NoDuplication-NoStatic based on the defined view Occupancy-NoPasserby-NoDuplication below.

```
CREATE VIEW Occupancy-NoPasserby-NoDuplication-NoStatic AS
SELECT StaticDevices(WiFi.mac,SD)
FROM Occupancy-NoPasserby-NoDuplication
```

Finally, by incorporating three UDFs in the occupancy query in Figure 6.1, we get the *advanced occupancy query* in Figure 6.2.

```
SELECT COUNT(*)
FROM Occupancy-NoPasserby-NoDuplication-NoStatic
```

Figure 6.2: Advanced Occupancy Query.

6.3.3 Use Case Scenario

We have had the occupancy application deployed and running in more than 40 buildings at University of California, Irvine (UCI) for over four years. Figure 6.3 is our occupancy application that displays the occupancy for each floor in engineering buildings in low/medium/high levels on the UCI campus. The application could also display the exact occupancy number for other granularities of locations, such as building/floor/region/room, at any customized time range. It is worth mentioning that the occupancy information displayed in the dashboard could be updated automatically based on the window query, such as giving the occupancy of a given location in the last 10 minutes, by ingesting the streaming data and updating the occupancy numbers in near real-time.

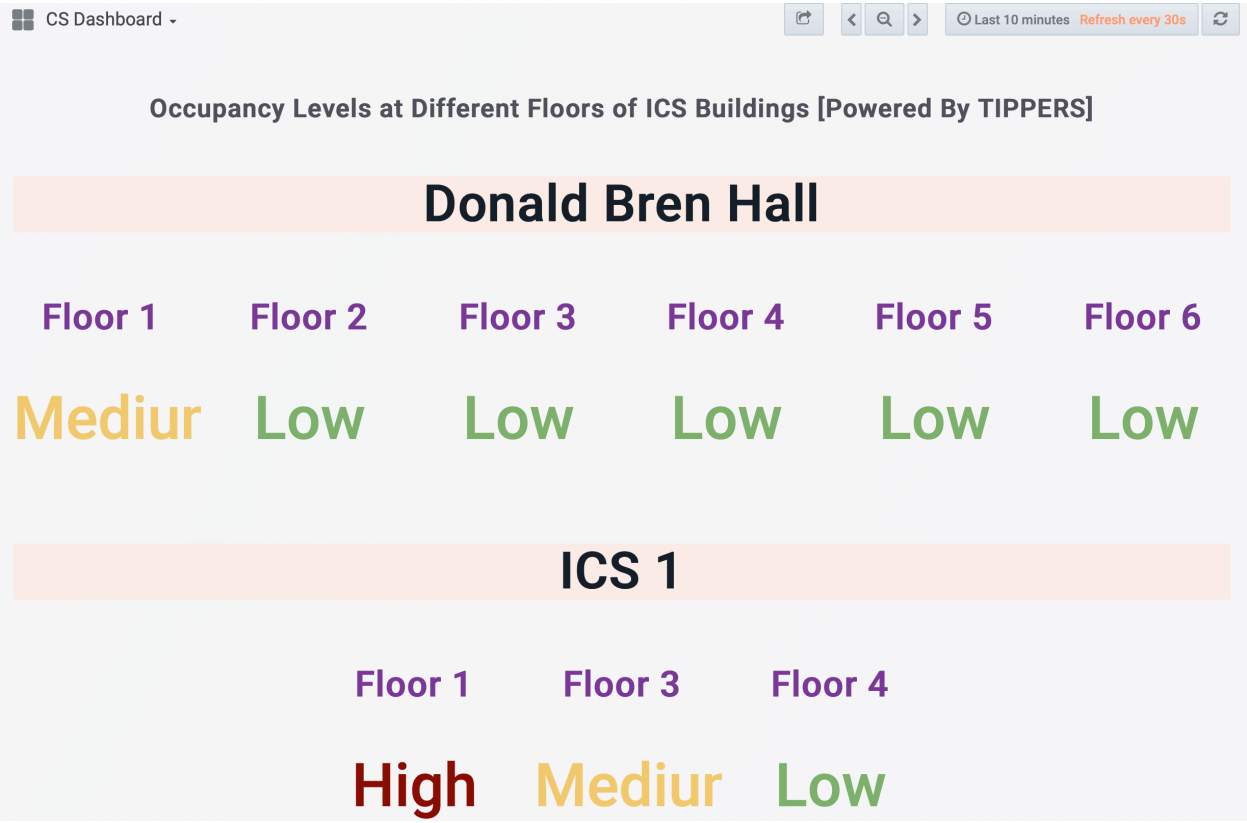


Figure 6.3: Occupancy Use Case Scenario.

6.4 Contact Tracing

We now introduce the second application, contact tracing, which we built using LOCATER technology.

Contact tracing technology has emerged as a key mitigation strategy for COVID-19 [108]. Several systems based on technologies ranging from Bluetooth [8], GPS, and WiFi [75] have been developed and widely adopted worldwide. The success of such technology, however, depends upon the participation by a large segment of the population (some estimates suggest $> 80\%$ [9]), while several studies [63] have shown that the adoption rate of existing technologies remains much lower, limiting their effectiveness. While WiFiTrace [109] also builds a contact tracing application using WiFi connectivity data, they assume data is clean without resolving data cleaning challenges.

In contrast to contact tracing systems such as the above that either require users to download apps/install new software or operating system, and trust the third parties with their location/proximity data, we build a new system, *T-COVE*, which is passive (does not require users to actively participate in the protocol), does not capture any additional information about individuals other than what is already captured by WiFi networks, and targets the technology at the organizational level. While T-Cove is designed to support the organization level mitigation of COVID-19, the underlying technology can be used for several distinct applications including smart occupancy-based HVAC control, estimating occupancy during disasters for evacuation planning, understanding individuals' behavior as related to space, etc.

6.4.1 Contact Tracing Model

Before we discuss contact tracing, let us first specify how we model contact. Different countries have different protocols for defining contact. For instance, in the USA in the context of COVID-19, contact is defined as being within 6 feet of an affected person for a cumulative total of 15 minutes or more over a 24-hour period [108]. We define *contact* as the user and the affected person being in the same room for a cumulative total of τ_1 minutes or more over a τ_2-hour period. Although two people in the same room might not be within 6 feet (i.e., false negatives), our definition does not introduce false positives and it is easy to be used for practical deployments where several follow-up steps are normally taken to ascertain contact.

Given the above definition of contact tracing model, we capture the essence of contact tracing through the following three queries:

- *ReportQuery* ($Q_R = \{name, mac, st, et\}$): that given the name of an affected person carrying a device with the mac address *mac*, determines the locations (i.e. regions/-rooms) and times the person visited those locations. We ask the user to report the mac address of their phone, and we store this information in the User table.
- *CheckQuery* ($Q_C = \{name, st, et\}$): that allows a user to check if he or she came in contact with any affected users during a given time interval $\{[st, et)\}$.
- *ContactQuery* ($Q_T = \{st, et\}$): that returns the set of people who have been exposed to any affected user during the time period $\{[st, et)\}$.

The queries above taken together form the basic contact tracing application that we have built using LOCATER.

The fundamental SQL query for all the ReportQuery, CheckQuery, and ContactQuery is to

```
SELECT LOCATER(Presence.roomLocation,Presence.macAddress,t) AS Location, Presence.startTime AS Start time, Presence.endTime AS End time
FROM Presence, User
WHERE Presence.macAddress = User.macAddress
AND User.name = name
AND Presence.startTime ≥ st AND Presence.endTime ≤ st
AND Presence.startTime ≥ et AND Presence.endTime ≤ et
```

Figure 6.4: Presence Query.

find the trajectories, i.e., (location, time interval), for an individual. We write the *Presence Query* in Figure 6.4 to compute the trajectories at room level, i.e., the room locations for a person whose name is name during the time interval (st,et).

Note that in the presence query, the parameter t inside the LOCATER UDF can be any time instance at the corresponding time interval (Presence.startTime, Presence.endTime), and t is set as (Presence.startTime+Presence.startTime)/2, since we assume that in each tuple (time interval) in the Presence table, the room location of a person is same.

ReportQuery directly uses the query answer of the presence query as the trajectories of a user which are then stored in the *Affected Trajectories* table which has the same schema as the Presence table. The Affected Trajectories table essentially stores the trajectories of all affected users. CheckQuery compares the results of the presence query with the trajectories of the affected people stored in the Affected Trajectories table to identify if a person contacts any affected user or not. ContactQuery will call CheckQuery for a group of people to perform contact tracing.

6.4.2 Use Case Scenario

To test ZIP and PLAUQE, we implemented the contact tracing application over the data captured from the Donald Bren Hall (DBH), at University of California, Irvine. DBH houses the School of Information and Computer Sciences. DBH has 64 WiFi AP each covering

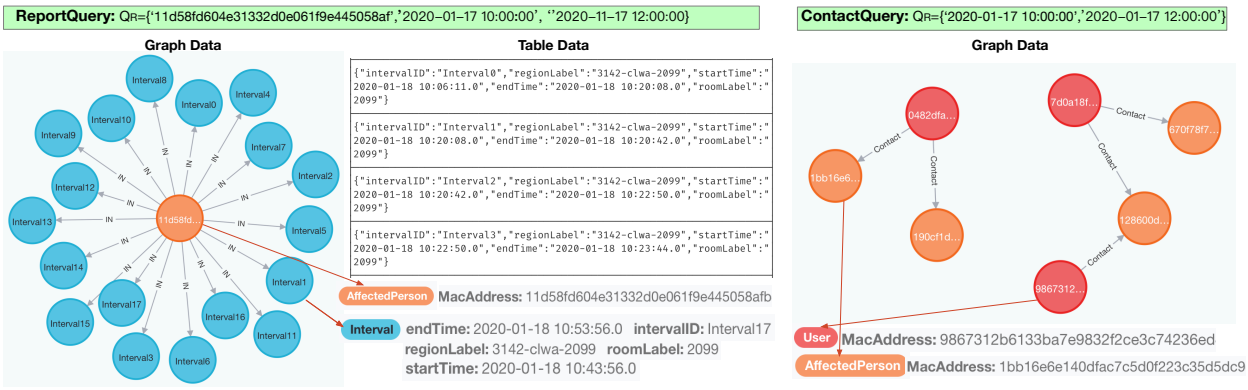


Figure 6.5: Contact Tracing Use Case Scenario.

approximately 11 rooms, 300+ rooms, and the average number of people in the building is greater than 1000. The dataset contains 10 months of data, from Sep. 3rd, 2019 to July 8th, 2020, comprising $38,670,714$ connectivity events for $66,717$ different devices. For the purpose of the study, we randomly select 100 devices to mimic the affected people.

In Figure 6.5, we show the output of ReportQuery, and ContactQuery.

- **ReportQuery**: An affected user carrying a device with mac address starting with 11d5 reports herself in the time interval from "2020-01-17 10:00:00" to "2020-01-17 12:00:00". The left graph in Figure 6.5 shows the predicated trajectories (blue nodes) of the affected person (orange node). One blue node represents a part of the trajectories by storing the following key fields, the region location (in *regionLabel*) and room location (in *roomLabel*) of a person during a time interval (*startTime*, *endTime*). The trajectories reported by a person can also be represented in a table in Figure 6.5.

- **ContactQuery**: The right graph in Figure 6.5 shows the set of people (red nodes) who were in contact with any affected person (orange nodes) during the time interval from "2020-01-17 10:00:00" to "2020-01-17 12:00:00".

6.5 Evaluation

We evaluate ZIP and PLAQUE in the context of contact tracing and occupancy applications as part of this case study. Note that the improvement due to ZIP and PLAQUE is evaluated independently, since ZIP and PLAQUE have been implemented in different database systems, and it is thus difficult to evaluate the performance due to ZIP and PLAQUE techniques being used simultaneously. [4]

Data Sets. We use two real data sets in UC Irvine campus to evaluate the effect of ZIP, WiFi and WiFi-large. WiFi consists of one week of WiFi connectivity events at the Donald Bren Hall Building (DBH) in UCI. WiFi records the continuous connectivity data of devices - i.e., which device is at which location in which time instance, and WiFi has a total of 325,925 connectivity logs. We further collected a larger real data set WiFi-large by extending the WiFi data set from one building to 40+ buildings over campus within one week with 8,799,975 connectivity logs.

Query Workload. We generate a query workload that contains 20 queries, where 10 queries are based on the presence query in Figure 6.4, and others are based on the occupancy query in Figure 6.2. Both the occupancy query and presence query can join with the Space and User tables to support more complex queries. In the generated query workload, we developed more complex selection-projection-join queries with optional aggregations developed from the basic presence query and occupancy query. In particular, in the generated query workload, we select a random user in the User table for the presence queries and a random location in the Space table for the occupancy queries. The time interval (st, et) in the query workload is generated randomly below. Let t_{min} and t_{max} be the minimum timestamp and the maximum timestamp in the data set. [5] We first generate two random timestamps t_1 and t_2 in the value

[4] We note that it is an important future work to implement ZIP and PLAQUE in one database system to maximize the performance of the improvement from both technologies.

[5] Both WiFi and WiFi-large data sets consist of one week of connectivity events from '2018-05-01' to '2018-05-07'. Thus we set t_{min} ='2018-05-01 00:00:00' and t_{max} ='2018-05-07 11:59:59'.

range $[t_{min}, t_{max}]$. Next let $st = min(t_1, t_2)$ and $et = max(t_1, t_2)$ be a randomly sampled start time and end time of a time interval used in the queries.

We now explore the optimization opportunities by using ZIP and PLAQUE to accelerate the query processing in the generated query workload.

6.5.1 ZIP Optimization

As described in Chapter 3, given a large data set that contains missing data, instead of performing imputations for all missing entries in the data set, ZIP performs a lazy execution to impute minimal missing values to answer a given query. ZIP is a query time approach that co-optimizes the missing value imputations and query execution. The query answer returned by ZIP is exactly the same as the one returned by the offline approach that first imputes all missing data and then runs query processing.

We now explore the optimization opportunity brought by ZIP to speed up the occupancy and contact tracing applications by optimizing the generated query workload.

Compared Strategies.

- *Offline* is the approach that first imputes all missing locations in the WiFi data, and then performs query execution.
- *QTC-Eager* is another query-time strategy that imputes missing values eagerly without delaying imputations during query execution as soon as the imputed value is required during query processing. Comparing ZIP to QTC-Eager will show the benefits of the lazy imputation strategy. QTC-Eager is the underline strategy used in the state-of-the-art technology ImputeDB [34] (see Chapter 3 for the detailed description).

(Seconds)	Min	Max	Avg	Avg Speed Up VS Offline	Avg Speed Up VS QTC-Eager
WiFi	1.3	32.7	9.1	1822X	19.2X
WiFi-Large	3.5	55.8	27.5	20736X	186X

Table 6.6: ZIP in the Presence Queries.

(Seconds)	Min	Max	Avg	Avg Speed Up VS Offline	Avg Speed Up VS QTC-Eager
WiFi	3.1	49.3	19.6	273X	4.6X
WiFi-Large	6.3	82.9	41.5	2315X	42X

Table 6.7: ZIP in the Occupancy Queries.

Experimental Results.

We report the runtime of ZIP and its improvement over the offline and QTC-Eager approaches in the presence query workload in Table 6.6, and occupancy query workload in Table 6.7. In the presence query workload, we observe that ZIP has 9.1 and 27.5 seconds of run time in WiFi and WiFi-large data sets on average, and it speeds up the offline approach by 1822X and 20736X in WiFi and WiFi-large data sets. ZIP also outperforms the QTC-Eager strategy by 19.2X and 186X in WiFi and WiFi-large data sets, respectively. In the occupancy query workload, we similarly observed improvements due to ZIP in both WiFi and WiFi-large data sets. But the improvement due to ZIP in the occupancy query workload is less than the improvement in the presence query workload. This is because the occupancy queries tend to be less selective than the presence queries in our query workload. For instance, compared with a presence query that determines the trajectories of a person during time interval (st, et), an occupancy query that determines the occupancy at location loc during the same time interval needs to clean the missing locations for all devices in the time interval (st, et) if there are no other predicates to help reduce the complex.

The above observations clearly show that adding ZIP optimization considerably improves the performance of the queries in the generated query workload to support an interactive analysis in the occupancy and contact tracing applications.

(Minutes)	Min	Max	Avg
VanillaDB	1.2	13.5	4.2
PLAQUE	0.08	6.9	0.8

Table 6.8: PLAQUE VS VanillaDB. (Presence Queries, WiFi)

(Minutes)	Min	Max	Avg
VanillaDB	3.8	22.3	7.6
PLAQUE	0.6	8.6	1.9

Table 6.9: PLAQUE VS VanillaDB. (Occupancy Queries, WiFi)

(Minutes)	Min	Max	Avg
VanillaDB	18.3	287.2	128.3
PLAQUE	0.3	7.8	2.3

Table 6.10: PLAQUE VS VanillaDB. (Presence Queries, WiFi-large)

(Minutes)	Min	Max	Avg
VanillaDB	28.5	384.7	202.8
PLAQUE	0.9	17.3	3.1

Table 6.11: PLAQUE VS VanillaDB. (Occupancy Queries, WiFi-large)

6.5.2 PLAQUE Optimization

PLAQUE, as described in Chapter 4, explores the opportunities to learn predicates at query run time to accelerate query execution from various operators, such as MIN/MAX aggregate, theta-join, equi-join, GROUP BY/HAVING. We implemented PLAQUE technology in VanillaDB [20], which is an Apache database prototype project. VanillaDB contains several key components in a relational database system, such as the storage, query optimizer, and query executor where several popular join algorithms are implemented, such as hash-join, index-join, sort-based-join, nested-loop-join.

Compared Strategy.

- *VanillaDB.* VanillaDB, serves as the representative of a standard query engine without PLAQUE optimization, i.e., VanillaDB calls LOCATER UDF to compute the required location whenever the need comes during query execution.

By adding PLAQUE optimizations, the system automatically adds and refines new predicates in the query plan tree associated with the given query in the middle of a query execution to speed up the query processing. We run VanillaDB in the same datasets, WiFi and WiFi-large, using the presence and occupancy query workloads.

Experimental Results.

In the WiFi data set, as shown in Table 6.8 and Table 6.9, adding the new predicates learned in PLAQUE improves the standard query execution without PLAQUE optimizations in VanillaDB by 5.25X and 4X on average in the presence query workload and the occupancy query workload, respectively.

When we evaluate PLAQUE in a larger data set WiFi-Large in the same query workload, as shown in Table 6.10 and Table 6.11, the improvements due to PLAQUE becomes more significant. In particular, PLAQUE outperforms VanillaDB by 55.8X and 65.4X on average in the presence query workload and the occupancy query workload, respectively. It shows that PLAQUE will offer more advantages in the larger data set since it can potentially skip more expensive UDF calls by using the learned predicates in PLAQUE.

6.6 Conclusions

In this chapter, we build two real applications, occupancy and contact tracing, using LOCATER, which itself is a standalone data cleaning technology to predict the indoor location of an individual at a given time instance at an accuracy of 90% on average. We further explored the optimization opportunities by using the ZIP (described in Chapter 3) and PLAQUE (described in Chapter 4) to improve the performance of both applications. Both ZIP and PLAQUE significantly improved performance. For instance, for Presence Queries ZIP brought down the execution time from 5115 seconds of the baseline QTC-Eager to 27.5 seconds, and PLAQUE optimization brought down execution times from 7698 to 138 seconds. Likewise, for occupancy queries, both PLAQUE and ZIP brought in significant improvements. However, we note that the execution time for both Presence queries and Occupancy queries remain above 27.5 and 41.5 seconds respectively (on WiFi-Large) which,

while significantly improved over baselines, is still on the high side for interactive analytics. This prompts us to a future work that explores combining both PLAQUE and ZIP to reduce execution time further, and possibly integrating it with progressive computing, as implemented in EnrichDB [48, 47] to bring down the execution times to what would truly be an interactive rate of query execution. Combining these strategies into the same system likely introduces new opportunities for deeper co-optimization of the system, which may lead to further exciting research.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In today's data-driven analysis and applications, vast streams of data are generated at high rates. For instance, there are 20k WiFi connectivity logs per minute at UCI campus during the busy hour. We need to process the data appropriately prior to it being used for the downstream applications/analysis. The goal of this thesis is to develop mechanisms to support computationally expensive operations, such as data enrichment, imputations, information extraction, and data interpretation, in data management systems to support interactive analysis.

The data flow pipeline is similar to the traditional data flow for analytical or hybrid analytical/transactions applications with some differences, the key of which is that data must be transformed prior to being used in building applications. Such transformations are reminiscent of data cleaning, data preparation, and data enrichment tasks in modern analytical data flows. In both cases, data cleaning/preparation/enrichment is expensive and, as the thesis argued, can hardly be done completely during the data ingestion time. For instance,

as we have shown imputing one hour of missing locations in WiFi connectivity data takes more than 180 hours prior to query analysis.

As a result, we explore a different data flow pipeline motivated by the recent data lake and ELT (Extract-Load-Transform) where raw data is processed only when needed, also known as query-time approach or analysis-aware approach. The key idea of the query time approach is to only perform the data computations that are related to the given queries, minimizing the required computations to be performed and thus scaling to large volumes of data sets. However, since the query-time approach moves the computations from pre-processing to query-time, it increases the query latency. The thesis explored two strategies, a lazy approach (i.e., ZIP) and a predicate-learning approach (i.e., PLAQUE), to reduce such latency to optimize query processing for compute-intensive tasks as described in Chapter 3 and Chapter 4.

In Chapter 3, we introduced ZIP, a query-time approach to reduce the missing value imputations during query execution. Specifically, ZIP co-optimizes imputation cost and query processing cost by modifying relational operators to be imputation-aware. The modified operators use a cost-based decision function to determine whether to invoke imputation or to defer imputations to the downstream operators to resolve missing values. The modified query processing logic ensures the results with deferred imputations are identical to those produced if all missing values were imputed first. ZIP includes a novel outer-join-based approach to preserve missing values during execution, and a bloom filter-based index to optimize the space and running overhead. ZIP outperforms the offline approach by up to 19607 times in a real UCI WiFi data set.

In Chapter 4, we introduced PLAQUE, a system that is able to learn new predicates at query time from various operators, such as MIN/MAX aggregate, theta join, equi join, GROUP BY, HAVING. By placing the learned predicates in their optimal locations in the query plan tree, and supporting various implementations of the learned predicates (in-memory

predicate or index-predicate), PLAQUE is able to accelerate query execution considerably. Comprehensive evaluations on both synthetic and real datasets demonstrate that the learned predicate approach adopted by PLAQUE can significantly accelerate query execution by up to 33x, and this improvement increases to up to 100x when User-Defined Functions (UDFs) are utilized in queries.

We next explored the use of the above technologies in a real-world smart space use case in UCI by building LOCATER to find the indoor location of an individual inside a building based on WiFi connectivity data. LOCATER in itself is a novel technical contribution since it is the first principled approach that uses data cleaning to predict the indoor missing location. As described in Chapter 5, LOCATER is free, does not require any new hardware to be installed in the building, nor require the active participation of users, such as installing applications in users' phone. LOCATER has been deployed in over 40 buildings in UCI for four years since 2019 with an accuracy of 87% on average.

Finally, we concluded the thesis with a real-world case study (described in Chapter 6) where we build two applications, occupancy, and contact tracing, by using LOCATER technology. We also implemented ZIP and PLAQUE optimizations inside these applications, demonstrating that with the help of ZIP and PLAQUE, these real applications we built in UCI can scale to large data sets.

7.2 Future Work

While our work on developing the lazy and predicate learning approaches opens several new directions of research as future work.

Future work in ZIP

In Chapter 3, we described a lazy approach entitled ZIP to co-optimize query processing and missing value imputation. ZIP imputes a minimal missing number of data to answer the given query. It opens several interesting technical directions to be explored further. First, when deciding whether or not a missing value needs to be imputed or its imputation should be delayed, ZIP uses a cost-based solution decision function. This decision function is applied at the level of individual missing values. It would be interesting to explore if a decision function could be applied on the column level instead of a tuple level to further reduce the overhead of decision making in ZIP. While a column-level decision making strategy could improve the performance further, how to adaptively update the decision based on the cost estimation during query execution in such a strategy is non-trivial.

Second, ZIP is most suitable for non-blocking imputation methods such as rule-based, time-series-based, knowledge base or crowdsourcing-based imputation approaches. However, ZIP poses some limitations when the blocking imputation methods are used, such as a learning-based approach that is able to impute a missing value only after the complete table data is used for training purposes. In the current ZIP system, when using such a learning-based imputation approach to impute a column C with missing data in one relation R, ZIP needs to load the complete relation R and uses the tuples whose attribute value under column C is not missing to construct a training data set to train a model D. ZIP will then use D to impute any missing data in the column C. In this case, if the training cost dominates the inference cost, ZIP will not provide much benefit. How to design strategies inside ZIP to make it more suitable and to better support such learning-based imputation approaches remains an important future work in ZIP to enhance its adoption.

Future work in PLAQUE

In Chapter 4, we described PLAQUE which is able to learn a suite of new predicates at query runtime to accelerate query execution from various query operators, such as MIN/MAX aggregate, theta join, equi join, GROUP BY, and HAVING. It leads to several interesting

future directions of research. First, in the current system implementation of PLAQUE, PLAQUE keeps updating all the learned predicates during query execution to maximize the effectiveness of the learned predicates. However, the frequent updates of a learned predicate might not be worth it if the predicate is not effective in filtering tuples in the remaining query execution. In such a situation, the overhead incurred by PLAQUE to update the predicates might not be justified. One direction of the work is to explore a trade-off between the updating frequency and the effectiveness of the learned predicates. Specifically, can we set a frequency f to update the predicate every k time units (or every k tuples) so that PLAQUE can co-optimize the predicate learning overhead and its power to filter tuples in the system. Another challenge is how to model and estimate the effectiveness of a learned predicate during query execution to achieve the above trade-off.

Second, PLAQUE so far focuses on optimization for a single query. It would be interesting to explore the knowledge from the query workload or domain knowledge to further learn new predicates or improve the learned predicates during query execution. In particular, if there exist strong patterns in a query workload, such as determining the maximum occupancy at a given location during a certain time interval every day, the information from the previous queries might be useful to improve future queries with similar query patterns.

Future work in LOCATER

In Chapter 5 we described LOCATER, a data cleaning technology to predict the indoor location of an individual. LOCATER is a passive and free solution since it only explores the WiFi connectivity data, which offers great usability since LOCATER does not require any new hardware to be installed in the building or the installation of any applications on users' phones. However, if there is some information that is *already available* in the environment as domain knowledge, exploiting such information would potentially improve the accuracy of LOCATER. For instance, if the knowledge of the meeting schedule of a group is available (e.g., group X will have regular meetings in room L during certain time intervals weekly), such

an event schedule calendar would be useful to be explored and embedded into LOCATER system by implementing as an add-on API. The robustness and the accuracy of LOCATER can be improved by exploring such new information in the environment or even combining it with other available sensors in the building, such as cameras. It would be interesting to explore how to integrate different knowledge from various data sources into LOCATER to make it a more reliable and robust software.

Building Database System based on ZIP and PLAQUE

Lastly, we talk about future work in terms of system building at a more conceptual level.

In our current work, different pieces of technologies, ZIP and PLAQUE, have been implemented independently in different database prototypes. ZIP is implemented in the SimpleDB [4], a database prototype system developed at MIT that has been used for research purposes at several universities including MIT, University of Washington, and Northwestern University. PLAQUE is implemented in an Apache project VanillaDB [20, 112], codebase of which includes key components, such as query executor and optimizer. It would be useful to fully integrate ZIP and PLAQUE in EnrichDB or TippersDB or other database systems, such as PostgreSQL, Spark or AsterixDB.

Second, besides the lazy approach used in ZIP, and the predicate learning approach explored in PLAQUE, another useful idea, progressiveness, has been also widely explored in EnrichDB and TippersDB as we introduced in Chapter 2. A progressive approach explores the trade-off of the quality of query answers and the time overhead it takes to return the result. It aims to return query results fast in the early stage of query processing whose quality will be improved over time. All of the above approaches, lazy, predicate learning as well as progressive approaches, are all query-time strategies that aim to reduce the query latency. How to combine and implement three approaches together into one data system carefully remains an important challenge and future work.

Last but not least, the proposed technologies, such as ZIP or PLAQUE, focus on the single-node machine, and it is exciting to explore the extension of current systems to a distributed data system. Especially, an intriguing avenue of exploration would be the extension of PLAQUE, automatic predicate learning at query runtime, to distributed data systems. Consider a distributed database system comprising n nodes or machines. Each node is allocated a data partition and conducts query processing independently. If we can learn effective predicates at each node and facilitate communication of these learned predicates among all nodes, it may yield a significant improvement in overall performance. Additionally, optimizing data partitioning based on learned predicates presents another interesting direction. One unique property of the learned predicates is the order of the stored data. For example, in the case of a query requesting the maximum value in attribute A, subject to a set of predicates, if the data values in A are stored in decreasing order, the learned predicate from the max aggregate condition would prove optimal at the very early stages of query processing. This optimization occurs as soon as the aggregate operator receives its first tuple during the pipeline query processing. Such potential extensions of predicate learning could significantly enhance query processing in distributed systems, a topic we are keen to explore further.

Bibliography

[1] *Center for Disease Control. National Health and Nutrition Examination Survey.* https://www.cdc.gov/nchs/nhanes/ContinuousNhanes/Default.aspx?BeginYear=2013., 2013-2014.

[2] *Skew TPC-H Benchmark.*https://bit.ly/2wvdNVo, 2016.

[3] *Github Codebase of ImputeDB.* https://github.com/mitdbg/imputedb.git, 2017.

[4] *6.830 Lab 1: SimpleDB.* http://db.csail.mit.edu/6.830/assignments/lab1.html, 2019.

[5] https://en.wikipedia.org/wiki/Possible_world, 2020.

[6] *Histogram-Based Gradient Boosting Ensembles.* https://machinelearningmastery.com/histogram-based-gradient-boosting-ensembles/, 2021.

[7] https://en.wikipedia.org/wiki/Bloom_filter, 2021.

[8] https://www.covidwatch.org/platform, 2021.

[9] https://www.wsj.com/articles/singapore-built-a-coronavirus-app-but-it-hasnt-wor
2021.

[10] *Table Functions.*https://www.postgresql.org/docs/7.3/xfunc-tablefunctions.html, 2021.

[11] *Tabular UDFs.*https://docs.snowflake.com/en/developer-guide/udf/sql/udf-sql-tabular-functions, 2021.

[12] *Cloud4WI.* https://cloud4wi.com/proximity-messaging/, 2022.

[13] https://scikit-learn.org/stable/modules/generated/sklearn.impute.KNNImputer.html, 2022.

[14] *IMDB Data Set.*https://developer.imdb.com/non-commercial-datasets/, 2022.

[15] *IndoorAtlas.* *https://www.indooratlas.com/solutions/indooratlas-location/*, 2022.

[16] *WIACOM.* *https://wiacom.ai/*, 2022.

[17] *Event Condition Action (ECA).*https://en.wikipedia.org/wiki/Event_condition_action, 2023.

[18] *TPC-DS Benchmark.*http://www.tpc.org/tpcds, 2023.

[19] *TPC-H Benchmark.*http://www.tpc.org/tpch, 2023.

[20] *VanillaDB.*https://www.vanilladb.org/, 2023.

[21] A. Afram and F. Janabi-Sharifi. Theory and applications of hvac control systems–a review of model predictive control (mpc). *Building and Environment*, 72:343–355, 2014.

[22] Y. Agarwal, B. Balaji, R. Gupta, J. Lyles, M. Wei, and T. Weng. Occupancy-driven energy management for smart building automation. In *Proceedings of the 2nd ACM workshop on embedded sensing systems for energy-efficiency in building*, pages 1–6, 2010.

[23] T. Aljuaid and S. Sasi. Proper imputation techniques for missing values in data sets. In *2016 International Conference on Data Science and Engineering (ICDSE)*, pages 1–5. IEEE, 2016.

[24] H. Altwaijry et al. Query-driven approach to entity resolution. *PVLDB*, 6(14):1846–1857, 2013.

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

[26] M. Ambrosin, M. Conti, R. Lazzeretti, M. M. Rabbani, and S. Ranise. Collective remote attestation at the internet of things scale: State-of-the-art and future challenges. *IEEE Communications Surveys & Tutorials*, 22(4):2447–2461, 2020.

[27] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pages 214–223. PMLR, 2017.

[28] M. Athanassoulis, K. S. Bøgh, and S. Idreos. Optimal column layout for hybrid workloads. *Proceedings of the VLDB Endowment*, 12(13):2393–2407, 2019.

[29] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15, 1985.

[30] P. Bansal, P. Deshpande, and S. Sarawagi. Missing value imputation on multidimensional time series. *arXiv preprint arXiv:2103.01600*, 2021.

[31] D. Birant and A. Kut. St-dbscan: An algorithm for clustering spatial–temporal data. *Data & knowledge engineering*, 60(1):208–221, 2007.

[32] P. G. Brown and P. J. Haas. Bhunt: Automatic discovery of fuzzy algebraic constraints in relational data. In *Proceedings 2003 VLDB Conference*, pages 668–679. Elsevier, 2003.

[33] L. F. Burgette and J. P. Reiter. Multiple imputation for missing data via sequential regression trees. *American journal of epidemiology*, 172(9):1070–1076, 2010.

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

[35] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[36] Y. Chen et al. Pegasus: Data-adaptive differentially private stream processing. In *ACM SIGSAC*, pages 1375–1388, 2017.

[37] X. Chu et al. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *SIGMOD*, pages 1247–1261, 2015.

[38] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proceedings of the VLDB Endowment*, 6(13):1498–1509, 2013.

[39] N. C. Dang, M. N. Moreno-García, and F. De la Prieta. Sentiment analysis based on deep learning: A comparative study. *Electronics*, 9(3):483, 2020.

[40] G. Deak et al. A survey of active and passive indoor localisation systems. *Computer Communications*, 35(16):1939–1954, 2012.

[41] E. Elnahrawy and B. Nath. Online data cleaning in wireless sensor networks. In *Proceedings of the 1st International conference on Embedded networked sensor systems*, pages 294–295, 2003.

[42] R. Enns, M. Bjorklund, and J. Schoenwaelder. Netconf configuration protocol. Technical report, RFC 4741, December, 2006.

[43] W. Fan et al. Towards certain fixes with editing rules and master data. *The VLDB journal*, 21(2):213–238, 2012.

[44] R. Gerhards et al. The syslog protocol. Technical report, RFC 5424, March, 2009.

[45] L. Getoor and A. Machanavajjhala. Entity resolution: theory, practice & open challenges. *Proceedings of the VLDB Endowment*, 5(12):2018–2019, 2012.

[46] S. Ghayyur et al. Iot-detective: Analyzing iot data under differential privacy. In *SIGMOD*, pages 1725–1728, 2018.

[47] D. Ghosh, P. Gupta, S. Mehrotra, and S. Sharma. Supporting complex query time enrichment for analytics. In *26th International Conference on Extending Database Technology (EDBT)*, 2023.

[48] D. Ghosh, P. Gupta, S. Mehrotra, R. Yus, and Y. Altowim. Jenner: just-in-time enrichment in query processing. *Proceedings of the VLDB Endowment*, 15(11):2666–2678, 2022.

[49] S. Giannakopoulou et al. Cleaning denial constraint violations through relaxation. In *SIGMOD*, pages 805–815, 2020.

[50] A. Go, R. Bhayani, and L. Huang. Twitter sentiment classification using distant supervision. *CS224N project report, Stanford*, 1(12):2009, 2009.

[51] L. Golab, F. Korn, F. Li, B. Saha, and D. Srivastava. Size-constrained weighted set cover. In *2015 IEEE 31st International Conference on Data Engineering*, pages 879–890. IEEE, 2015.

[52] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[53] J. W. Graham. Missing data analysis: Making it work in the real world. *Annual review of psychology*, 60:549–576, 2009.

[54] P. Gupta, M. J. Carey, S. Mehrotra, and o. Yus. Smartbench: a benchmark for data management in smart spaces. *Proceedings of the VLDB Endowment*, 13(12):1807–1820, 2020.

[55] P. Gupta et al. Quest: Practical and oblivious mitigation strategies for covid-19 using wifi datasets. *arXiv preprint arXiv:2005.02510*, 2020.

[56] P. Gupta, S. Mehrotra, S. Sharma, R. Yus, and N. Venkatasubramanian. Sentaur: Sensor observable data model for smart spaces. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 3131–3140, 2022.

[57] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. *ACM SIGMOD Record*, 28(2):287–298, 1999.

[58] J. M. Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 25, 2008.

[59] M. Herschel, M. A. Hernández, and W.-C. Tan. Artemis: A system for analyzing missing answers. *Proceedings of the VLDB Endowment*, 2(2):1550–1553, 2009.

[60] I. F. Ilyas and X. Chu. *Data cleaning*. Morgan & Claypool, 2019.

[61] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: Automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658, 2004.

[62] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *2008 IEEE 24th International Conference on Data Engineering*, pages 774–783. IEEE, 2008.

[63] C. Jee. *Is a successful contact tracing app possible? These countries think so.*`https://www.technologyreview.com/2020/08/10/1006174/covid-contract-tracing-app-germany-ireland-success/`, 2020.

[64] C. S. Jensen et al. Graph model based indoor tracking. In *MDM*, pages 122–131. IEEE, 2009.

[65] R. Jia et al. Soundloc: Accurate room-level indoor localization using acoustic signatures. In *CASE*, 2015.

[66] Y. Jiang, X. Pan, K. Li, Q. Lv, R. P. Dick, M. Hannigan, and L. Shang. Ariel: Automatic wi-fi based room fingerprinting for indoor localization. In *Proceedings of the 2012 ACM conference on ubiquitous computing*, pages 441–450, 2012.

[67] S. Kandula, L. Orr, and S. Chaudhuri. Pushing data-induced predicates through joins in big-data clusters. *Proceedings of the VLDB Endowment*, 13(3):252–265, 2019.

[68] W. Kang and Y. Han. Smartpdr: Smartphone-based pedestrian dead reckoning for indoor localization. *IEEE Sensors journal*, 15(5):2906–2916, 2014.

[69] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.

[70] M. Khayati et al. Mind the gap: an experimental evaluation of imputation of missing values techniques in time series. *PVLDB*, 13(5):768–782, 2020.

[71] W. Kim, B.-J. Choi, E.-K. Hong, S.-K. Kim, and D. Lee. A taxonomy of dirty data. *Data mining and knowledge discovery*, 7(1):81–99, 2003.

[72] H. Kimura, G. Huo, A. Rasin, S. Madden, and S. B. Zdonik. Correlation maps: A compressed access method for exploiting soft functional dependencies. *Proceedings of the VLDB Endowment*, 2(1):1222–1233, 2009.

[73] M. Kotaru et al. Spotfi: Decimeter level localization using wifi. In *SIGCOMM*. ACM, 2015.

[74] M. Kurucz, A. A. Benczúr, and K. Csalogány. Methods for large scale svd with missing values. In *Proceedings of KDD cup and workshop*, volume 12, pages 31–38. Citeseer, 2007.

[75] G. Li et al. vcontact: Private wifi-based contact tracing with virus lifespan. *arXiv preprint arXiv:2009.05944*, 2020.

[76] P. Li, X. Rao, J. Blase, Y. Zhang, X. Chu, and C. Zhang. Cleanml: a study for evaluating the impact of data cleaning on ml classification tasks. In *36th IEEE International Conference on Data Engineering (ICDE 2020)(virtual)*. ETH Zurich, Institute for Computing Platforms, 2021.

[77] Z. Li et al. A passive wifi source localization system based on fine-grained power-based trilateration. In *WoWMoM*, pages 1–9. IEEE, 2015.

[78] W.-C. Lin and C.-F. Tsai. Missing value imputation: a review and analysis of the literature (2006–2017). *Artificial Intelligence Review*, 53(2):1487–1509, 2020.

[79] Y. Lin et al. Locater: cleaning wifi connectivity datasets for semantic localization. *PVLDB*, 2021.

[80] Y. Lin, P. Khargonekar, S. Mehrotra, and N. Venkatasubramanian. T-cove: an exposure tracing system based on cleaning wi-fi events on organizational premises. *Proceedings of the VLDB Endowment*, 14(12):2783–2786, 2021.

[81] H. Liu et al. Survey of wireless indoor positioning techniques and systems. *SMC*, 37(6):1067–1080, 2007.

[82] T. Liu, J. Fan, Y. Luo, N. Tang, G. Li, and X. Du. Adaptive data augmentation for supervised learning over missing data. *Proceedings of the VLDB Endowment*, 14(7):1202–1214, 2021.

[83] Z.-g. Liu, Q. Pan, J. Dezert, and A. Martin. Adaptive imputation of missing values for incomplete pattern classification. *Pattern Recognition*, 52:85–95, 2016.

[84] C. Luo et al. Pallas: Self-bootstrapping fine-grained passive indoor localization using wifi monitors. *IEEE TMC*, 16(2):466–481, 2016.

[85] S. Makridakis and M. Hibon. The m3-competition: results, conclusions and implications. *International journal of forecasting*, 16(4):451–476, 2000.

[86] S. Mehrotra et al. Tippers: A privacy cognizant iot environment. In *PerCom Workshops*, pages 1–6, 2016.

[87] X. Miao, Y. Wu, L. Chen, Y. Gao, J. Wang, and J. Yin. Efficient and effective data imputation with influence functions. *Proceedings of the VLDB Endowment*, 15(3):624–632, 2021.

[88] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. *ACM SIGMOD Record*, 23(2):103–114, 1994.

[89] A. Musa and J. Eriksson. Tracking unmodified smartphones using wi-fi monitors. In *Sensys*, pages 281–294, 2012.

[90] F. Oldewurtel, D. Sturzenegger, and M. Morari. Importance of occupancy information for building climate control. *Applied energy*, 101:521–532, 2013.

[91] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, 2008.

[92] N. Panwar et al. Iot notary: Sensor data attestation in smart environment. In *NCA*, pages 1–9. IEEE, 2019.

[93] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *Proceedings of the VLDB Endowment*, 11(6):663–676, 2018.

[94] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *MobiCom*, pages 32–43. ACM, 2000.

[95] Z. Qi, H. Wang, J. Li, and H. Gao. Frog: Inference from knowledge base for missing value imputation. *Knowledge-Based Systems*, 145:77–90, 2018.

[96] T. E. Raghunathan, J. M. Lepkowski, J. Van Hoewyk, P. Solenberger, et al. A multivariate technique for multiply imputing missing values using a sequence of regression models. *Survey methodology*, 27(1):85–96, 2001.

[97] P. Royston and I. R. White. Multiple imputation by chained equations (mice): implementation in stata. *Journal of statistical software*, 45:1–20, 2011.

[98] S. Salimi and A. Hammad. Optimizing energy consumption and occupants comfort in open-plan offices using local control based on occupancy dynamic data. *Building and Environment*, 176:106818, 2020.

[99] M. Seifeldin et al. Nuzzer: A large-scale device-free passive localization system for wireless environments. *TMC*, 12(7):1321–1334, 2012.

[100] E.-L. Silva-Ramírez, R. Pino-Mejías, M. López-Coello, and M.-D. Cubiles-de-la Vega. Missing value imputation on missing completely at random data using multilayer perceptrons. *Neural Networks*, 24(1):121–129, 2011.

[101] M. Sokolova and G. Lapalme. A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4):427–437, 2009.

[102] S. Song, A. Zhang, L. Chen, and J. Wang. Enriching data imputation with extensive similarity neighbors. *Proceedings of the VLDB Endowment*, 8(11):1286–1297, 2015.

[103] M. Staudt and M. Jarke. *Incremental maintenance of externally materialized views*. Citeseer, 1995.

[104] J. Tervonen et al. Applying and comparing two measurement approaches for the estimation of indoor wifi coverage. In *NTMS*, 2016.

[105] A. Tharwat. Classification assessment methods. *Applied Computing and Informatics*, 2020.

[106] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[107] N. Tran, A. Lamb, L. Shrinivas, S. Bodagala, and J. Dave. The vertica query optimizer: The case for specialized query optimizers. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1108–1119. IEEE, 2014.

[108] A. Trivedi et al. Digital contact tracing: technologies, shortcomings, and the path forward. *SIGCOMM*, 50(4):75–81, 2020.

[109] A. Trivedi et al. Wifitrace: Network-based contact tracing for infectious diseasesusing passive wifi sensing. *arXiv preprint arXiv:2005.12045*, 2020.

[110] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *TOIS*, 10(1):91–102, 1992.

[111] J. Weise, S. Schmidl, and T. Papenbrock. Optimized theta-join processing. *BTW 2021*, 2021.

[112] S.-H. Wu, T.-Y. Feng, M.-K. Liao, S.-K. Pi, and Y.-S. Lin. T-part: Partitioning of transactions for forward-pushing in deterministic database systems. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2012.

[113] C. Xu et al. Scpl: Indoor device-free multi-subject counting and localization using radio signal strength. In *IPSN*, pages 79–90, 2013.

[114] C. Ye, H. Wang, J. Li, H. Gao, and S. Cheng. Crowdsourcing-enhanced missing values imputation based on bayesian network. In *International Conference on Database Systems for Advanced Applications*, pages 67–81. Springer, 2016.

[115] C. Ye, H. Wang, W. Lu, and J. Li. Effective bayesian-network-based missing value imputation enhanced by crowdsourcing. *Knowledge-Based Systems*, 190:105199, 2020.

[116] W. Young, G. Weckman, and W. Holland. A survey of methodologies for the treatment of missing values within datasets: Limitations and benefits. *Theoretical Issues in Ergonomics Science*, 12(1):15–43, 2011.

[117] M. Youssef, M. Mah, and A. Agrawala. Challenges: device-free passive localization for wireless environments. In *MobiCom*, pages 222–229, 2007.

[118] F. Zafari et al. A survey of indoor localization systems and technologies. *IEEE Communications Surveys & Tutorials*, 21(3):2568–2599, 2019.

[119] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *arXiv preprint arXiv:1208.0081*, 2012.

[120] Q. Zhou, J. Arulraj, S. Navathe, W. Harris, and J. Wu. Sia: Optimizing queries using learned predicates. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2169–2181, 2021.

[121] J. Zhu, N. Potti, S. Saurabh, and J. M. Patel. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. *Proceedings of the VLDB Endowment*, 10(8):889–900, 2017.