Scalability aspects of data cleaning

by

Hemant Saxena

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science- Internship

Waterloo, Ontario, Canada, 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: **Leonidas Galanis**
Director of Engineering, Snowflake

Supervisor(s): **Ihab F. Ilyas**
Professor, Cheriton School of Computer Science,
University of Waterloo

Internal Member: **M. Tamer Özsu**
Professor, Cheriton School of Computer Science,
University of Waterloo
Grant Weddell
Professor, Cheriton School of Computer Science,
University of Waterloo

Internal-External Member: **Lukasz Golab**
Associate professor, Department of Management Sciences,
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis consists of material all of which I authored or co-authored. Chapter 2 is based on a joint work with Lukasz Golab and Ihab F. Ilyas [85, 84]. Chapter 3 is based on the joint work with Shrinu Kushagra , Ihab F. Ilyas, and Shai Ben-David [59]. Chapter 4 is a joint work with Lukasz Golab, Stratos Idreos, and Ihab F. Ilyas [83].

Abstract

Data cleaning has become one of the important pre-processing steps for many data science, data analytics, and machine learning applications. According to a survey by Gartner, more than 25% of the critical data in the world's top companies is flawed, which can result in economic losses amounting to trillions of dollars a year. Over the past few decades, several algorithms and tools have been developed to clean data. However, many of these solutions find it difficult to scale, as the amount of data has increased over time. For example, these solutions often involve a quadratic amount of tuple-pair comparisons or generation of all possible column combinations. Both these tasks can take days to finish if the dataset has millions of tuples or a few hundreds of columns, which is usually the case for real-world applications.

The data cleaning tasks often have a trade-off between the scalability and the quality of the solution. One can achieve scalability by performing fewer computations, but at the cost of a lower quality solution. Therefore, existing approaches exploit this trade-off when they need to scale to larger datasets, settling for a lower quality solution. Some approaches have considered re-thinking solutions from scratch to achieve scalability and high quality. However, re-designing these solutions from scratch is a daunting task as it would involve systematically analyzing the space of possible optimizations and then tuning the physical implementations for a specific computing framework, data size, and resources.

Another component in these solutions that becomes critical with the increasing data size is how this data is stored and fetched. As for smaller datasets, most of it can fit in-memory, so accessing it from a data store is not a bottleneck. However, for large datasets, these solutions need to constantly fetch and write the data to a data store. As observed in this dissertation, data cleaning tasks have a *lifecycle-driven* data access pattern, which are not suitable for traditional data stores, making these data stores a bottleneck when cleaning large datasets.

In this dissertation, we consider scalability as a first-class citizen for data cleaning tasks and propose that the scalable and high-quality solutions can be achieved by adopting the following three principles: 1) by having a new primitive-base re-writing of the existing algorithms that allows for efficient implementations for multiple computing frameworks, 2) by efficiently involving domain expert's knowledge to reduce computation and improve quality, and 3) by using an adaptive data store that can transform the data layout based on the access pattern. We make contributions towards each of these principles. First, we present a set of primitive operations for discovering constraints from the data. These primitives facilitate re-writing efficient distributed implementations of the existing discovery

algorithms. Next, we present a framework involving domain experts, for faster clustering selection for data de-duplication. This framework asks a bounded number of queries to a domain-expert and uses their response to select the best clustering with a high accuracy. Finally, we present an adaptive data store that can change the layout of the data based on the workload's access pattern, hence speeding-up the data cleaning tasks.

Acknowledgements

I would like to take this opportunity to express my sincere thanks to my advisor Prof. Ihab F. Ilyas. He has been a great support over the past five years, giving me the freedom and time to pursue projects across different domains such as distributed systems, theoretical machine learning, and building database systems, which eventually culminated into this dissertation. Without his broad experience and knowledge, I wouldn't have been able to pull-off these diverse projects. His advising style is unique, where he is not just an academic advisor but also a personal mentor. Our technical discussions were often accompanied by pep-talks which were not just motivating but also taught me valuable lessons such as abstracting out the ideas from the details, multi-tasking, and always pushing hard to deliver beyond what is expected. As I embark on a new journey from this point onwards, I will treasure and remember all that I have learnt from him.

Next, I would like to thank my committee members, starting with Prof. Lukasz Golab. He was like a co-supervisor to me, always available for brainstorming ideas which helped me make progress when I was stuck, and taught me how to get better at technical writing. Next, I would like to thank Prof. Grant Weddell and Prof. Tamer Özsu for serving as my internal committee members, and spending time reading and providing valuable comments on my dissertation. Finally, I would like to thank my external committee member Dr. Leonidas Galanis, for his valuable industrial experience, which helped me address the practical aspects of my work.

This journey wouldn't have been possible without the support and camaraderie of my amazing friends at Waterloo. Special shout-out to Anirudh, Sharat, Jimit, Shrinu, Abhinav, and Dhinakaran, who started their PhD journeys at around the same time as me, for many fun hangout sessions, for being my squash/bouldering buddies, and supporting each other through the ups and downs of a PhD life. Amongst my friends, Priya, also to-be life partner, has a special place for always being there as an emotional support and making me feel at home while being far away from home. Her cheerfulness, encouragement, and patience definitely made this journey a bit more fun and a bit less tiring.

Finally, without the love and support of my parents, I wouldn't have even dreamt of accomplishing everything that I have done today. Their tireless efforts, the importance they gave to a good education, and their patience and trust in me helped me each step of the way in pursuing my dreams.

Dedication

To my parents and my brother Divyansh.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the amount of data generated and consumed by modern businesses increases, enforcing and maintaining the quality of the data becomes a critical task [55]. In these businesses, as the data from multiple sources accumulates in a database over time, many errors can creep into the data. For example, many records in the data end up having missing values, typos, mixed formats, duplicate entries, and violations of business and data integrity rules. Gartner predicted [89] that more than 25% of the critical data in the world's top companies is flawed. According to a survey, referenced in [56] about the state of data science and machine learning (ML), dirty data is the most common issue faced by users dealing with data. Therefore, data cleaning has become one of the most important pre-processing steps for many data science, data analytics, and ML workflows.

Figure 1.1 shows a typical data cleaning workflow for modern business applications. The cleaning workflow consists of three main steps: *discovery*, *error detection*, and error repair [56]. The *discovery* step is responsible for extracting rules or constraints which are used for detecting errors. For example, an implicit data quality rule of an *employee* instance is the following: "*two persons with the same zip code live in the same state*", or $zip \rightarrow state$ in short. If in an *employee* instance, we identify two employees with the same zip code but different state, we are certain that at least one of the data values of zip code and state for the two employees is erroneous. Since designing data quality rules by consulting with domain experts is an expensive and time-consuming process, they are mostly mined automatically from the data.

The *error detection* step consists of detecting qualitative errors, quantitative errors and duplicate entries from the data [55]. Qualitative errors are the values that violate business rules or constraints, which are discovered in the *discovery* step and sometimes available

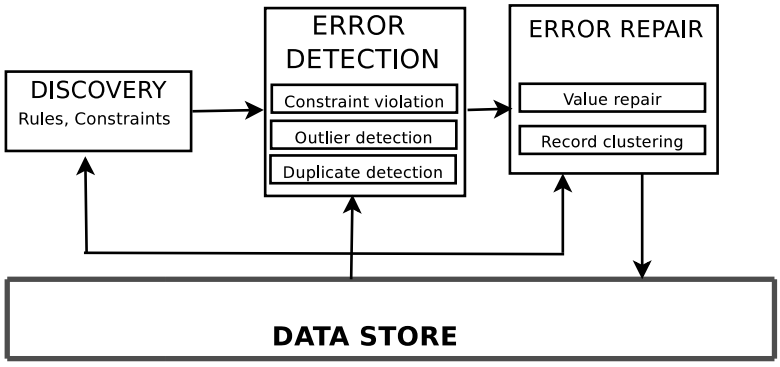


Figure 1.1: Data cleaning workflow for analytics applications.

from domain experts. For example, any violation to the constraint $zip \rightarrow state$ from the previous example would be considered a qualitative error. Quantitative errors are the values that are statistical outliers when compared to the rest of the data. For example, if in an *employee* table salaries are expected to be around $100,000, then an employee with salary of $10,000 would be considered an outlier. Lastly, duplicate entries are one of the pervasive errors in business applications and are often introduced while integrating data from different sources. Duplicate detection involves detecting entries which belong to the same real-world entity. For example, two employee entries with same name and phone number, but different addresses could point to the same physical employee. Detecting duplicate entries is also known as record linkage, record matching, or entity resolution, and has been extensively studied over the past few decades [45, 73, 49].

The *error repair* step is responsible for generating fixes for the erroneous values. Most of the error repair techniques [55] use the existing data and optionally domain experts to generate fixes, which are applied to the data. For example, to fix the error for $zip \rightarrow state$ we consult the data to find all tuples which are part of the error and use them to generate candidate fixes, which are then presented to a domain expert who finally decides the right fix. Fixing duplicate entries involves clustering entries which can represent the same real-world entity, and then either automatically or using a domain expert, fuse these entries into a single entry.

1.1 Scalability of data cleaning workflows

As the scale of data has increased manifold over time, it has become challenging to scale data cleaning solutions [87] due to their computational complexity. For example, many of

the data cleaning algorithms have a quadratic complexity from comparing pairs of tuples, as seen in data de-duplication [87]. This quadratic complexity becomes a bottleneck as it can take days to enumerate pairs of tuples in a dataset with a few million tuples. Similarly, tasks like constraint discovery become a bottleneck with an increasing number of columns, since these tasks need to consider all possible column combinations.

1.1.1 Trade-off between scalability and quality

However, the data cleaning tasks have a trade-off between scalability and quality — they can achieve scalability by having a lower quality solution, thereby creating a design space that existing solutions have explored. For example, correlation clustering is a gold-standard solution for data de-duplication [60]. Here, the goal is to find the clustering that correlates 'as much as possible' with the 'true clustering', where entries corresponding to the same entity belong in the same cluster and different entries belong to different clusters. However, correlation clustering is NP-Hard [60]. Therefore, practitioners often use an ensemble of cheap clustering algorithms such as hierarchical clustering, with the hope of selecting the best clustering from the group. Since many of these clustering algorithms have their own hyper parameters, selecting the best clustering is a complex task and often results in a poor quality clustering.

Similarly, discovering all constraints from the data has an exponential search space — a constraint can be defined on any combination of columns. To overcome the exponential search space, many discovery tasks often limit the search to constraints with fewer columns. However, having an incomplete set of constraints can then result in un-detected errors.

Re-designing data cleaning solutions from scratch to achieve scalability and quality is another alternative, but is often a complex task. Re-designing from scratch would involve efforts towards (i) systematically analyzing the space of possible optimizations, i.e., identifying the core data processing steps and optimizing them for scalability and quality, and (ii) tuning the physical implementations of these steps for a specific computing framework (e.g., centralized computing, distributed computing), data sizes, and resources (e.g., optimizing for memory, optimizing for disk I/O). This could then result in over-fitting the solution to a specific framework, making it inefficient or unusable for certain other frameworks. For example, consider Spark [94] and MapReduce, two commonly used distributed computing frameworks with different performance characteristics, especially in the case of storing intermediate results (Data Frames [94] vs. storing to HDFS [86]). An algorithm optimized for Spark might not perform well on MapReduce.

1.1.2 Importance of a Data Store

As shown in Figure 1.1, all the data cleaning tasks depend on an underlying data store where entries are stored and fetched as needed. Using data cleaning solutions for large datasets would require many entries to be fetched and written to the data store by each of these tasks. Therefore, having a suitable data store can also significantly speed-up data cleaning tasks, allowing it to scale to large datasets.

In this dissertation, we observe that many tasks such as detecting constraint violations, outlier detection, and de-duplication, access data in both row-at-a-time and column-at-a-time pattern. For example, in the case of outlier detection, we need to build histograms and probability distribution functions (PDFs) on each column [14], this requires column-at-a-time access over the data. Whereas when checking for outliers, data needs to be accessed one row-at-a-time to check each entry's column values against the histogram. Moreover, as we discuss this data access pattern in detail in Chapter 4, we show that error detection happens over newer data and statistical evidences such as histograms, and PDFs are built over older data. Therefore, the data access pattern depends on the age of the data.

Existing solutions rely either on a pure row-store, or a pure column-store, and are therefore unable to benefit from the other storage layout. Alternatively, they replicate the data across the two storage types, adding complexity to data management. Therefore, having an adaptive data store that can transform the layout of the data as it gets old can speed-up the data cleaning tasks by up to an order of magnitude, as we show in Chapter 4.

1.1.3 Dissertation's hypothesis

In this dissertation, we consider scalability as a *first-class citizen* for data cleaning workflows and propose that data cleaning tasks can achieve scalability and high-quality solutions by adopting the following three principles:

- *Using a new primitive-based re-writing of the existing algorithms that allows for efficient implementations for multiple computing frameworks.* As mentioned in Section 1.1.1, re-designing algorithms for a new computing framework, for example a distributed framework, is a daunting task. To facilitate the end-user in overcoming this challenge, the existing algorithms can be decomposed into data-operations such as *joins*, *group-by*, etc., which are well understood across many computing frameworks. These data-operations will then allow for rewrites of the existing algorithms

independent of the framework, analyze their costs, and explore the space of possible designs.

- *By efficiently involving domain expert's knowledge to reduce computation and improve quality.* Domain experts are expected to provide high-quality solutions for many data cleaning tasks, such as identifying errors, consolidating duplicates, and generating fixes for errors. However, using domain experts is costly and time consuming, making it difficult to scale when used naïvely. If we can develop techniques that involve domain experts to fetch only the most valuable information, we can reduce the computation while achieving high-quality results.

- *By using an adaptive data store that can transform the data layout based on access pattern.* As mentioned in Section 1.1.2, many of the data cleaning tasks need to fetch entries in both row-at-a-time and column-at-a-time pattern. Having a single storage system than can transform the layout of the data based on its access pattern can significantly improve the data access time, thereby speeding-up the cleaning tasks.

1.2 Contributions and outline

As part of an attempt to build scalable data cleaning workflows, we make contributions under each of the principles proposed in the previous section. Our contributions are the following:

1. **Distributed discovery of constraints:** Following the principle of using a *new primitive-based re-writing*, we consider the discovery step (see Figure 1.1) of cleaning workflows and propose a set of primitive data-operations that can be used to rewrite the distributed version of constraint discovery algorithms. Constraints are expressed using the formal language of integrity constraints (ICs), such as functional dependencies (FDs), denial constraints, etc., and discovering these rules is considered as a major part of the *data profiling* task [15], which is responsible for discovering the metadata. Many of the previous constraint discovery (also known as dependency discovery) algorithms have assumed a centralized setting where data is stored locally. Therefore, deploying them on distributed platforms naïvely results in a high communication cost, as most of these algorithms often use large intermediate results to optimize computation cost.

   To facilitate efficient distributed versions of constraint discovery algorithms, we decompose existing algorithms into six logical *primitives*, corresponding to data processing steps separated by communication barriers. These primitives allow us to

rewrite the discovery algorithms, analyze the computation and communication costs of each step, and explore the space of possible distributed designs, each with different performance characteristics.

2. **Framework for faster clustering selection for de-duplication:** Following the principle of *involving domain experts*, we propose a faster clustering selection framework with human-in-the-loop for data de-duplication. The problem of data de-duplication is also viewed as a clustering task [51]. Here, the goal is to put records corresponding to the same physical entity in the same cluster, while separating the records corresponding to different entities into different clusters. Solving the de-duplication problem then means finding the right clustering. In practice, we try out an ensemble of clustering algorithms, and further, each algorithm with different hyper-parameter values, with the goal of selecting the "best" clustering. A clustering is defined to be better than another clustering based on a certain loss function. Computing the loss function for a clustering requires knowledge of whether a pair of items belong to the same entity or not. This knowledge is often known to domain-experts. As an abstraction, we assume that an "oracle" is in possession of this knowledge. However, querying the oracle is costly. To obtain the true value of the loss function, one would need to query the oracle for every pair of items, i.e. $O(n^2)$ queries [60], where n is the number of items. For large datasets, making queries for $O(n^2)$ pairs is either infeasible or extremely costly. We propose a novel framework which allows us to select the best clustering by making a bounded number of queries to an oracle, with the bound independent of n.

3. **Storage engine to speed up data cleaning tasks:** Finally, in line with the principle of an *adaptive data store*, we propose a Log-Structured Merge (LSM) Tree based storage engine. In Section 4.1, we observe that many data cleaning tasks such as detecting constraint violations, detecting outliers, and data de-duplication access recent data row-at-a-time to detect and fix errors, whereas old data is accessed via column scans to collect evidences towards fixing errors in the recent data. This access pattern has also been observed in many Hybrid Transactional-Analytical Processing (HTAP) applications, such as content recommendation, real-time inventory/pricing, high frequency trading, and IoT [74]. Therefore, it has motivated the development of several HTAP systems such as SAP HANA [47], MemSQL [7], and IBM Wildfire [28], which employ hybrid data layouts. The data in these systems is stored in different formats throughout their lifecycle. Recent data is stored in a row-oriented format to serve On-Line Transactional Processing (OLTP) workloads and support high data rates, while older data is transformed to a column-oriented format for On-

Line Analytical Processing (OLAP) access patterns. Such systems can be described as having a *lifecycle-aware* data layout.

We observe that a Log-Structured Merge (LSM) Tree [75] is a natural fit for a lifecycle-aware storage engine due to its high write throughput and level-oriented structure, in which records propagate from one level to the next over time. To the best of our knowledge, the RDBMSs used by current data cleaning workflows are either pure row-store systems such as MySQL or Postgres, or pure column-store systems such as MonetDB. We show that the data cleaning tasks run faster on our LSM-Tree based storage engine, when compared to running them on Postgres and MonetDB. In addition to the data cleaning tasks, this storage engine can also be integrated with RDBMS systems, such as MySQL that allows for different storage plugins, to run real-time analytics applications such as content recommendation, real-time inventory/pricing, and high frequency trading.

Outline: The outline of the dissertation is as follows: In chapter 2, we show how to have efficient distributed implementations of dependency discovery algorithms. In chapter 3, we give the details of our novel framework which allows selection of the best clustering by asking bounded number of queries to an oracle. In chapter 4, we present the LSM-Tree based storage engine that can speed-up data cleaning workloads. Finally, in chapter 5, we provide concluding remarks along with a discussion of future work.

Chapter 2

Distributed Dependency Discovery

Column dependencies such as candidate keys or Unique Column Combinations (UCCs), Functional Dependencies (FDs), Order Dependencies (ODs) and Denial Constraints (DCs) are critical in the error detection step of data cleaning. In addition to that, they are deemed important in many data management tasks such as schema design, data analytics and query optimization. Despite their importance, dependencies are not always specified in practice, and even if they are, they may change over time. Furthermore, dependencies that hold on individual datasets may not hold after performing data integration. As a result, there has been a great deal of research on automated discovery of dependencies from data; see, e.g., [15, 66] for recent surveys.

Existing work on dependency discovery proposes methods for pruning the exponential search space in order to minimize computation costs. However, existing methods assume a centralized setting where the data are stored locally. In contrast to centralized settings, in modern big data infrastructure, data are naturally partitioned (e.g., on HDFS [86]) and computation is parallelized (e.g., using Spark [94]) across multiple compute nodes. In these cases, it is inefficient at best and infeasible at worst to move the data to a centralized profiling system, motivating the need for distributed profiling.

In distributed environments, ensuring good performance requires minimizing computation and communication costs. A naïve solution to minimize communication costs is to allow no data communication at all: each node locally discovers dependencies from the data it stores, and then we take the intersection of the locally-discovered dependencies. To see why this approach fails, consider a table with a schema (A, B) and assume the table is partitioned across two nodes: the first node storing tuples (a_1, b_1), (a_1, b_1), and the second node storing tuples (a_1, b_2), (a_1, b_2). The FD $A \rightarrow B$ locally holds on both nodes but it

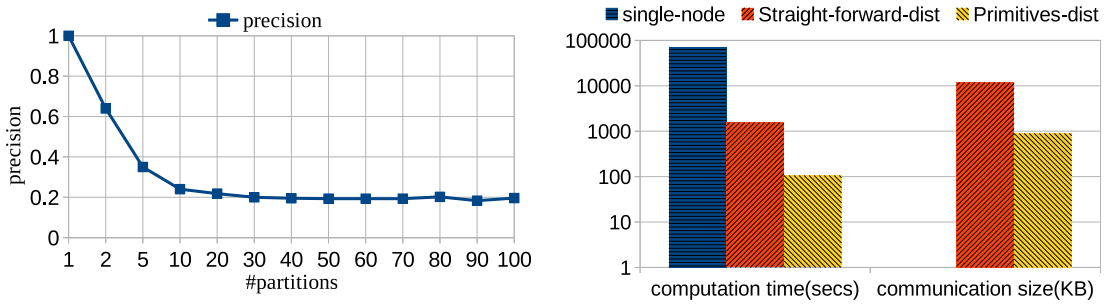
does not hold globally over the whole table. We show in Figure 2.1a that this problem gets worse quickly, even for as few as ten nodes, where the majority of the dependencies discovered locally do not hold on the entire dataset (TPC-H *lineitem* table with one million rows [11]). Notably, discovering dependencies from a sample has a similar problem.

Another possible solution is to parallelize existing non-distributed dependency discovery algorithms in a straightforward way. The problem with this approach is that existing algorithms often generate large intermediate results to minimize computation, leading to high communication overhead. Other problems include parallelizing the computation and load balancing—issues that, naturally, were not considered in centralized implementations.

In this work, we argue that to implement efficient distributed algorithms, an end-user needs to (i) systematically analyze the space of possible optimizations, i.e., identify the core data processing steps and optimize for both computation and communication when parallelizing these steps, and (ii) tune the physical implementations of these steps, i.e., design a distribution strategy for a given workload and the available computational and memory resources. To facilitate the end-user in overcoming these challenges, we decompose existing dependency discovery algorithms into six logical *primitives*, corresponding to data processing steps separated by communication barriers. The primitives allow us to rewrite the algorithms, analyze the computation and communication costs of each step, and explore the space of possible distributed designs, each with different performance characteristics.

From the point of view of an end-user, our primitive-oriented framework decouples writing distributed versions of the algorithms from tuning their physical implementations. We refer to the logical rewrites using our primitives as *logical discovery plans* or LDPs, and their physical implementations as *physical discovery plans* or PDPs. We present case studies (Sections 2.5 through 2.7), showing how our primitives allow us to explore the space of possible designs. In particular, for each algorithm, we write two LDPs using our primitives. One LDP follows the design principles of the original non-distributed implementations, and the other LDP modifies the original algorithm to make it distribution-friendly. We then demonstrate that different physical implementations of the primitives lead to different PDPs for the same LDP.

Figure 2.1b illustrates the impact of exploring different design options using our primitives. We compare the performance of three versions of the FastFDs algorithm [93] for FD discovery on a *homicide* data set with 100,000 rows and 24 columns. The first version, *single-node*, is the original FastFDs algorithm executed on a single machine. The second, *original-dist* is a distributed version of the original algorithm running on a 55-worker Spark cluster, which is faster than the single-machine solution but has a significant communication overhead. The third, *dist-friendly*, is a distribution friendly version that reduces the



(a) Drop in precision (i.e., the fraction of locally discovered FDs that hold over the entire dataset) with number of nodes

(b) Computation and communication costs of FastFDs

Figure 2.1: Precision of naive solution for distributed FD discovery and performance of three FastFDs implementations

data communication costs and is over an order of magnitude faster than *original-dist*.

Our contributions in this work, are as follows.

1. We propose a generalized framework for analyzing dependency discovery algorithms (including UCCs, FDs, ODs and DCs), which consists of six primitives that serve as building blocks of existing algorithms.

2. Using case studies, we illustrate how the primitives allow us to (i) decouple logical designs from physical implementations, and (ii) analyze the cost of individual data processing steps.

3. Using the proposed primitives, we implement and experimentally evaluate different distributed versions of seven existing dependency discovery algorithms on several real datasets.

2.1 Prior Work

There has been recent work on parallelizing dependency discovery algorithms across multiple threads, but it considers a singe-node shared-everything architecture where communication costs are not a bottleneck [48]. There is also some early work on distributed FD discovery. However, it suffers from the same issues as the naïve solution we mentioned earlier (i.e., it returns locally-discovered FDs which may not hold globally) [65], or it assumes

tid	ID	GD	AC	PH	CT	ST	ZIP	SAL	TR	STX
t1	1009	M	304	232-7667	Anthony	WV	25813	5000	3	2000
t2	2136	M	719	154-4816	Denver	CO	80290	60000	4.63	0
t3	0457	F	636	604-2692	Cyrene	MO	64739	40000	6	0
t4	1942	F	501	378-7304	West Crossett	AR	72045	85000	7.22	0
t5	2247	M	319	150-3642	Gifford	IA	52404	15000	2.48	50
t6	6160	M	970	190-3324	Denver	CO	80251	60000	4.63	0
t7	4312	F	501	154-4816	Kremlin	AR	72045	70000	7	0
t8	3339	F	304	540-4707	Kyle	WV	25813	10000	4	500

Table 2.1: Tax data records

that data are partitioned vertically and ensures efficiency by limiting the search space to FDs with single attributes [64].

2.2 Preliminaries

We begin by defining important concepts used in dependency discovery algorithms. Table 2.2 explains important symbols used in this chapter.

Definition 1. ***Relational model:*** *Let $R = \{A_1, A_2, ..., A_m\}$ be a set of attributes describing the schema of a relation R and r be a finite instance of R.*

2.2.1 Dependencies

Definition 2. ***Minimal unique column combination:*** *An attribute combination $X \subseteq R$ is a unique column combination (UCC) if X uniquely identifies tuples in r. X is a minimal UCC if no proper subset of it is a UCC.*

Definition 3. ***Minimal functional dependency:*** *Let $X \subset R$ and $A \in R$. A functional dependency (FD) $X \rightarrow A$ holds on r iff for every pair of tuples $t_i, t_j \in r$ the following is true: if $t_i[X] = t_j[X]$, then $t_i[A] = t_j[A]$. An FD $X \rightarrow A$ is minimal if A is not functionally dependent on any proper subset of X.*

Definition 4. ***Minimal order dependency:*** *Let X be a list of distinct attributes of R and $A \in R$. An order dependency (OD) $X \longmapsto A$ holds iff sorting r by X means that r is*

also sorted by A. An OD $X \longmapsto A$ is minimal if A is not order dependent on any proper subset of X.

Definition 5. ***Denial constraint:*** *A Denial constraint (DC) ψ is a statement of the form $\psi : \forall t_i, t_j \in r, \neg(P_1 \wedge ... \wedge P_k)$ where P_i is of the forms $v_1 \phi v_2$ with $v_1, v_2 \in t_x[A]$, $x \in \{i, j\}$, $A \in R$ and $\phi \in \{=, \neq, <, \leq, >, \geq\}$. The expression inside the brackets is a conjunction of predicates, each containing two attributes from R and an operator ϕ. An instance r satisfies ψ iff ψ is satisfied for any two tuples t_i, $t_j \in r$.*

Definition 6. ***Predicate space:*** *It is defined as the space of all possible expressions of the form $v_1 \phi v_2$ with $v_1, v_2 \in t_x[A]$, $x \in \{i, j\}$, $A \in R$ and $\phi \in \{=, \neq, <, \leq, >, \geq\}$.*

Example 2.2.1. *Consider the tax dataset in Table 2.1. The set $\{AC, PH\}$ is a UCC. Two persons with same zip code live in the same state, therefore the FD $ZIP \rightarrow ST$ holds. The single tax exemption decreases as salary increases, therefore the OD $SAL \longmapsto STX$ holds. If two persons live in the same state, the one earning a lower salary has a lower tax rate, therefore the following DC holds: $c : \forall t_i, t_j \in R, \neg(t_i.ST = t_j.ST \wedge t_i.SAL < t_j.SAL \wedge t_i.TR > t_j.TR)$.*

In the remainder of this chapter, discovering dependencies refers to discovering *minimal* dependencies.

2.2.2 Data structures

Definition 7. ***Equivalence classes:*** *Equivalence class of a tuple $t \in r$ with respect to an attribute set $X \subseteq R$ is denoted by $[t]_X = \{u \in r | \forall A \in X \; t[A] = u[A]\}$. The set $\pi_X = \{[t]_X | t \in r\}$ contains the equivalence classes of r under X.*

Note that π_X is a *partition* of r such that each equivalence class corresponds to a unique value of X. Let $|\pi_X|$ be the number of equivalence classes in π_X, i.e., the number of distinct values of X.

Definition 8. ***Evidence sets:*** *For any two tuples t_i and t_j, in r, their evidence set $EV(t_i, t_j)$ is the set of predicates satisfied by them, drawn from some predicate space P.*

Recall the predicate space considered by DCs from Definition 5, namely those with two attributes from R and an operator from ϕ. In Table 2.1, tuples t_2 and t_6 give $EV(t_2, t_6) = \{t_2.ID \neq t_6.ID, t_2.ID \leq t_6.ID, t_2.ID < t_6.ID, t_2.GD = t_6.GD, t_2.CT = t_6.CT, t_2.ST =$

Symbol	Meaning
R	A relation
r	An instance of R
n	Number of tuples in r
m	Number of attributes R
π_X	Equivalence classes of $X \subseteq R$
$EV(t_i, t_j)$	Evidence set due to tuple pair (t_i, t_j)
k	Number of workers/compute nodes
$\mathbf{Y}$	Maximum computation done by any worker
$\mathbf{X}$	Maximum data sent to any worker

Table 2.2: Summary of symbols

$t_6.ST$, ...}. For FDs and UCCs, it suffices to consider a restricted space of predicates that identify which attribute values are different. Here, $EV(t_2, t_6) = \{t_2.ID \neq t_6.ID$, $t_2.AC \neq t_6.AC$, $t_2.PH \neq t_6.PH$, $t_2.ZIP \neq t_6.ZIP\}$. As we will show in Section 2.3, some algorithms use evidence sets to identify dependencies that do *not* hold.

Definition 9. ***Partition refinement:*** *Partition π refines partition π' if every equivalence class in π is a subset of some equivalence class of π'.*

Distributed methods: We assume a map-reduce style of computation, where map jobs perform local computation, followed by a data communication step and a reduce step to compute the final results in parallel. Intermediate results are stored on a distributed file system such as HDFS, or maintained in memory, e.g., as Spark Resilient Distributed Datasets (RDD) [94]. Suppose we have k workers or compute nodes. Let X_i and Y_i be the amount of data sent to the i^{th} worker and the computation done by the i^{th} worker, respectively [36]. The runtime of a distributed algorithm depends on the runtime of the slowest worker. Thus, we will compute the following quantities for each tested algorithm:

$$\mathbf{X} = \max_{i \in [1,k]} X_i \qquad\qquad \mathbf{Y} = \max_{i \in [1,k]} Y_i$$

We compute these costs at the granularity of data values instead of tuples or columns. This makes our analysis independent of the data partitioning scheme (e.g., horizontal vs. vertical).

2.3 Algorithms

We classify dependency discovery algorithms into three categories: schema-driven, data-driven and hybrid.

Schema-driven algorithms: this class of algorithms traverses an *attribute lattice* in a breadth-first manner, an example of which is shown in Figure 2.2b for $R = \{A, B, C, D\}$. The nodes in the i^{th} lattice level, denoted L_i, correspond to sets of i attributes. Each node also stores the equivalence classes (Definition 7) corresponding to its attribute set. Edges between nodes are based on a set containment relationship of their attribute sets. The time complexity of schema-driven algorithms depends mainly on the size of the lattice, which is exponential with respect to the number of columns, but it is linear on the number of tuples, and therefore these algorithms work well for large datasets with few columns.

Consider the TANE [53] algorithm for discovering FDs (FastOD [90] is similar but it discovers ODs). For each lattice level, TANE performs three tasks: *generate next level*, *compute dependencies*, and *prune*. To generate the next level, TANE first creates new attribute sets by combining pairs of attribute sets from the current level; e.g., combining AB and AC gives ABC. This corresponds to a self-join of the current level's attribute combinations. Next, new equivalence classes are created by *intersecting* pairs of equivalence classes from the current level. For example, in Figure 2.2a we have $\pi_A = \{\{1, 3\}, \{2, 4\}$, $\pi_B = \{\{1\}, \{2\}, \{3, 4\}\}$, $\pi_C = \{\{1, 2, 3\}, \{4\}\}$ and $\pi_D = \{\{1, 2\}, \{3\}, \{4\}\}$. Intersecting π_C and π_D gives $\pi_{\{C,D\}} = \{\{1, 2\}, \{3\}, \{4\}\}$.

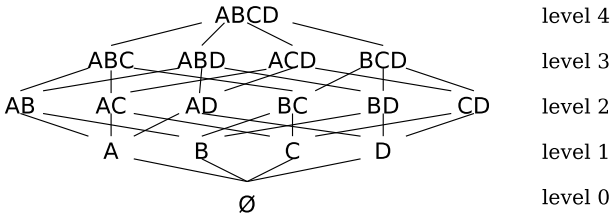
Once the attribute sets and equivalence classes for the next level L_l have been generated, the *compute dependency* task discovers FDs of the form $X \setminus A \rightarrow A$ for all $X \in L_l$, and for all $A \in X$. To determine if $X \setminus A \rightarrow A$, TANE checks if $\pi_{X \setminus A}$ *refines* (Definition 9) π_A. For example, in Figure 2.2a, $D \rightarrow C$ holds because π_D refines π_{CD}; however, $C \rightarrow D$ does not hold because π_C does not refine π_{CD}.

Compute dependencies is simpler for FD and UCC discovery but more complex for ODs. For FDs, it suffices to check if $|\pi_{X \setminus A}| = |\pi_X|$. For UCCs, X is a UCC if $|\pi_X| = r$, i.e., if all equivalence sets are singletons. On the other hand, for an OD $X \longmapsto Y$ to hold, every set in $\pi_{X \setminus A}$ must be a subset of some set in π_A and furthermore the elements must be ordered in the same way.

Finally, *prune* leverages the fact that only minimal dependencies are of interest; for example, if $A \rightarrow D$ holds then $AB \rightarrow D$ is not minimal. Depending on the algorithm, various pruning rules are applied to eliminate nodes from the lattice that cannot produce non-minimal dependencies. If a node is pruned, then any nodes connected to it can also be eliminated. For example, for FDs, a node labelled with an attribute set X can be pruned

A	B	C	D
a	a	a	a
b	b	a	a
a	c	a	c
b	c	d	e

(a) Relation instance



(b) Attribute lattice

Figure 2.2: Example relation instance and attribute lattice

if X is a key or $X \setminus A \rightarrow A$ was found to hold. Returning to our example, CD is pruned because $D \rightarrow C$ holds, and the following nodes are pruned because they correspond to keys: AB, AD, BC, and BD.

Data-driven algorithms: this class of algorithms examines pairs of tuples to identify evidence sets (Definition 8) and violated dependencies; in the end, any dependencies not found to be violated must hold. The time complexity of data-driven algorithms depends on the number of tuples, but not on the number of columns, and therefore these algorithms work well for small datasets with many columns.

Consider the FastFDs [93] algorithm for FD discovery. Returning to Figure 2.2a, we get the following evidence sets (expressed concisely as attributes whose values are different) from the six tuple pairs:

$$EV(t_1, t_2) = \{A, B\},\ EV(t_2, t_3) = \{A, B, D\},\ EV(t_1, t_4) = \{A, B, C, D\},$$
$$EV(t_1, t_3) = \{B, D\},\ EV(t_2, t_4) = \{B, C, D\},\ EV(t_3, t_4) = \{A, C, D\}$$

After FastFDs generates evidence sets, for each possible right-hand-side attribute of an FD, it finds all the left-hand-side attribute combinations that hold. Say A is the right-hand side attribute currently under consideration. The algorithm first removes A from the evidence sets, giving $\{\{B\}, \{B, C, D\}, \{B, D\}, \{C, D\}\}$. Next, FastFDs finds *minimal covers* of this set, i.e., minimal sets of attributes that intersect with every evidence set. In this example, we get BC and BD, and therefore we conclude that $BC \rightarrow A$ and $BD \rightarrow A$. FastDC [37] works similarly to discover DCs.

FastFDs avoids considering all $n(n-1)/2$ pairs of tuples when generating evidence sets. Instead, it only considers pairs of tuples that belong to the same equivalence class for at least one attribute. For example, in Figure 2.2a, tuples 1 and 4 are not in the same equivalence class for any of the four attributes. In these cases, a tuple pair has

Primitive	**TANE**	**FASTOD**	**FastFDs**	**FastDCs**	**HyFD**	**HyUCC**	**Hydra**
$genEQClass(X, I)$	✓	✓	✓		✓	✓	✓
$genEVSet(t_i, t_j, P)$			✓	✓	✓	✓	✓
$checkRefinement(X, Y, I)$	✓	✓			✓	✓	
$join(S_i, S_j, p)$	✓	✓	✓	✓	✓	✓	✓
$setCover(S)$			✓	✓	✓	✓	✓
$sort(S, Comparator)$		✓	✓	✓	✓	✓	✓

Table 2.3: Summary of primitives and their usage across algorithms

no attributes in common and therefore the corresponding evidence set is all of R, which trivially intersects with every possible cover.

Hybrid algorithms: these algorithms switch back and forth between schema-driven and data-driven phases; examples include HyFD [77], HyUCC [78] and Hydra [32]. For example, HyFD starts with a data-driven phase, but it generates evidence sets only from a sample of tuples. From these evidence sets, HyFD identifies FDs that have not been violated by the sampled tuple pairs (but may be violated by some other tuple pairs). To represent these potential FDs, HyFD uses an *FDTree* data structure, which is a prefix-tree, each of whose nodes corresponds to a set of attributes. Next, to validate the potential FDs, HyFD switches to a schema-driven phase, which traverses the FDTree level-wise, similarly to how TANE traverses the attribute lattice level-wise. At some point, HyFD may switch back to a data-driven phase and generate evidence sets from a different sample of tuples, and so on. HyUCC [78] and Hydra [32] are similar to HyFD but Hydra switches only once from the data-driven phase to the schema-driven phase.

2.4 Primitives

Our approach to design efficient distributed methods for dependency discovery is to identify the data processing steps and explore different implementations of these steps. To realize this approach, we propose a general framework consisting of six primitives listed below. We identified the primitives by decomposing existing algorithms into common data processing steps whose distributed implementations are well-understood. For example, as explained in Section 2.3, generating the next lattice level in schema-driven algorithms consists of a join (to generate new lattice nodes) and a group-by operation (to generate new equivalence classes). Similarly, generating evidence sets consists of a join.

1. **Generate equivalence classes ($genEQClass$):** $(X, r) \rightarrow \pi_X$
   Given set $X \subseteq R$ and data instance r, this primitive computes π_X. This is similar to the relational group-by operator and can be implemented by sorting or hashing the data. This primitive is used to verify if dependencies hold by schema-driven algorithms and to decide which tuple pairs to examine by data-driven algorithms.

2. **Generate evidence set ($genEVSet$):** $(t_i, t_j, P) \rightarrow EV(t_i, t_j)$
   Given a pair of tuples $t_i, t_j \in r$, this primitive generates a set of dependencies (defined under a predicate space P) that are violated by this tuple pair, i.e. $EV(t_i, t_j)$. Recall from Section 2.2 that DCs have the most general predicate space while FDs, ODs and UCCs have simpler predicate spaces. This primitive is used in data-driven and hybrid algorithms.

3. **Partition refinement ($checkRefinement$):** $(\pi_X, \pi_Y) \rightarrow \{\text{True, False}\}$
   This primitive returns true if π_X refines π_Y and false otherwise. Schema-driven and hybrid algorithms use this primitive. As discussed in Section 2.3, in UCC and FD discovery, this primitive can return true or false by comparing the counts of $|\pi_X|$ and $|\pi_Y|$, whereas in OD discovery, it needs to check the ordering of tuples within matching equivalence classes.

4. **Join ($join$):** $(S_i, S_j, p) \rightarrow \{x \cup y \mid x \in S_i \wedge y \in S_j \wedge p\}$
   This primitive joins two sets of elements, S_i and S_j, using p as the (optional) join predicate. Schema-driven (and hybrid) algorithms use the join when generating attribute sets for the next lattice level; here, it is a self-join of the previous level's attribute sets. Data-driven (and hybrid) algorithms join pairs of tuples to generate evidence sets.

5. **Cover ($setCover$):** $\{s_1, s_2, ..., s_c\} \rightarrow \{s_1^{min}, s_2^{min}, ...\}$
   Given a set of evidence sets $S = \{s_1, s_2, ..., s_c\}$ where $s_i \subseteq R$, this primitive computes all minimal covers $s_j^{min} \subseteq R$, and therefore the dependencies that hold given S. Cover is used in all data-driven and hybrid algorithms.

6. **Sort ($sort$):** $(S, Comparator) \rightarrow S$
   This primitive sorts the set S based on the provided comparator. FastOD sorts tuples within equivalence classes and checks for proper ordering during the partition refinement check. Data-driven algorithms sort evidence sets based on their cardinality to speed up the minimal cover operation. Hybrid algorithms use sorting during sampling (of tuple pairs to generate evidence sets).

Table 2.3 highlights the expressiveness of the primitives and their usage across seven popular and state-of-the-art dependency discovery algorithms. As mentioned earlier in the chapter, we refer to logical rewrites of the algorithms using the primitives as *logical discovery plans* (LDPs) and their physical implementations as *physical discovery plans* (PDPs).

Design space: There are many possible physical implementations of our primitives. As in DBMSs, one important factor to consider is the size of the input compared to the available memory, e.g., to determine when to use a *hash-join* or a *sort-merge-join*. Similar choices exist in distributed frameworks such as Spark and Map-reduce. In particular, different distribution strategies have different memory footprints, suggesting a space of possible optimizations. In Sections 2.5-2.7, we explore this optimization space with the help of our primitives. We consider TANE as a schema-driven example, FastFDs as a data-driven example and HyFD as a hybrid example (however, our conclusions apply to other algorithms within these three categories). For each case study, we show that different LDPs exist and we show two (of the many possible) distributed PDPs for each LDP. One PDP, which we call *large-memory plan* or lmPDP, assumes that each worker's memory is large enough for all computations; the other, which we call *small-memory plan* or smPDP, assumes that the data may spill to external storage. For the PDPs, we assume Spark to be the data processing framework. We show that different PDPs have different performance characteristics in terms of communication and computation cost, but we defer the issue of automatically selecting the best PDP for a given workload and system configuration to future work.

2.5 Case study 1: TANE

We start by studying TANE [53]. As explained in Section 2.3, for each lattice level, schema-driven algorithms such as TANE compute dependencies holding in this level, prune the search space based on the discovered dependencies, and generate the next level. The execution plans presented in this section can be easily extended to discover order dependencies [90] as follows. Discovering ODs is more expensive than FDs because, as discussed in Section 2.3, verifying ODs requires a refinement check and an ordering check. Thus, to implement the $checkRefinement$ primitive, complete equivalence classes must be examined (not just the number of equivalence classes, i.e., the number of distinct values).

```
1  TANE (Relation r, Schema R)
2  for A ∈ R do
3      L_1 = L_1 ∪
         {(A, genEQClass(A, r))}
4  l = 1
5  while L_l ≠ φ do
6      computeDependencies(L_l)
7      prune(L_l)
8      generateNextLevel(L_l)
9      l++
10 computeDependencies (Level L_l)
11 for X ∈ L_l do
12     for A ∈ X do
13         checkRefinement(X/A, X,
            (|π_{X/A}|, |π_X|))
14 generateNextLevel (Level L_l)
15 for (X, π_X, Y, π_Y) ∈ join(L_l, L_l) do
16     π_{X∪Y} = genEQClass(X ∪ Y,
        (π_X, π_Y))
17     L_{l+1} = L_{l+1} ∪ {(X ∪Y, π_{X∪Y})}
```

(a) Logical discovery plan 1

```
1  TANE (Relation r, Schema R)
2  for A ∈ R do
3      L_1 = L_1 ∪
         {(A, |genEQClass(A, r)|)}
4  l = 1
5  while L_l ≠ φ do
6      computeDependencies(L_l)
7      prune(L_l)
8      generateNextLevel(L_l)
9      l++
10 computeDependencies (Level L_l)
11 for X ∈ L_l do
12     for A ∈ X do
13         checkRefinement(X/A, X,
            (|π_{X/A}|, |π_X|))
14 generateNextLevel (Level L_l)
15 for (X, π_X, Y, π_Y) ∈ join(L_l, L_l)
    do
16     π_{X∪Y} = genEQClass(X ∪ Y, r)
17     L_{l+1} = L_{l+1} ∪ {(X ∪Y,
        |π_{X∪Y}|)}
```

(b) Logical discovery plan 2

Figure 2.3: TANE algorithm

2.5.1 LDP 1: Original TANE

Figure 2.3a shows the first LDP written using our primitives, which follows the design principles of the original TANE algorithm: compute new equivalence classes by intersecting the previous level's equivalence classes. To implement this, the input to the *genEQClass* primitive in line 16 consists of a new attribute combination $X \cup Y$ and the equivalence classes π_X and π_Y from the previous level. Furthermore, the lattice stores attribute combinations and their associated equivalence classes (lines 3 and 17).

Large-memory PDP

Generating first level: In lines 2-3, we generate equivalence classes for the first lattice level, i.e., for single attributes. This is implemented by distributing the columns in R across the k workers in a round-robin fashion. Each worker scans the tuples in r and hashes their values to compute equivalence classes for the columns assigned to it.

Computing dependencies: As discussed in Section 2.3, to check if an FD $X \setminus A \rightarrow A$ holds, it suffices to verify that $|\pi_{X \setminus A}| = |\pi_X|$. Thus, we distribute the counts of the equivalence classes (i.e. $|\pi_X|$) in the current lattice level L_i (i.e., the dependencies to check) across the k workers in a round-robin fashion and we broadcast the counts for attribute combinations in L_{i-1} to each worker.

Pruning: The driver machine then receives the discovered dependencies from the workers and applies pruning rules to the current lattice level. Lattice nodes that have not been pruned are used to generate the next level.

Generating next level: This requires a (self-) join to produce new attribute combinations and their equivalence classes. We implement the join as a map-reduce job, in which each worker creates a subset of nodes in the next lattice level. We use a distributed self-join strategy called the *triangle distribution strategy* [36], which was shown to be optimal in terms of communication and computation costs. Next, a map job generates new equivalence classes, in which each worker creates equivalence classes for the nodes it has generated during the join. New equivalence classes are created by intersecting pairs of equivalence classes from the previous level. For example, if a worker receives equivalence classes for AB and AC during the join, it can create equivalence classes for ABC. The new equivalence classes are written to the distributed filesystem.

Cost analysis: To generate equivalence classes in the first lattice level, the computation is linear in terms of the number of tuples (single pass to hash the tuple values), and

the columns are equally shared across all the workers. The cost of computing dependencies is negligible since we only need to compare counts of equivalence classes (line 13). To generate the next level of equivalence classes, the cost of the triangle distribution strategy is given by [36]: $X_i \leq |I| * \sqrt{2/k}$, and $Y_i \leq |I|(|I|-1)/2k$, where $|I|$ is size of the input to the join, which is a lattice level (L_l) in our case. Due to pruning, we compute equivalence classes for $|L_{l+1}|$ attribute combinations and not for all pairs of attribute combinations resulting from the self-join. This approximation gives: $Y_i \leq 2n|L_{l+1}|/k$. Aggregating the costs for up to m levels in the lattice, we get $\mathbf{X} \leq 2^m n\sqrt{2/k}$, and $\mathbf{Y} \leq 2^m 2n/k$. The factor of $2n$ in $\mathbf{Y}$ is because the intersection of two equivalence classes requires a scan over each one, which is $2n$ in the worst case.

Small-memory PDP

Note that generating equivalence classes is memory-intensive due to the size of the attribute lattice. In the small-memory PDP, we focus on an alternative implementation of this task.

Generating first level: Our strategy in the lmPDP was to use each worker to generate equivalence classes for multiple columns. Here, our strategy is to use multiple workers to generate one column's equivalence classes. We do this by using Spark's distributed *groupBy* operation. This has two advantages: it reduces the memory load per worker and allows Spark to take care of spilling the computation to disk.

Generating next level: The lmPDP required $O(n2^m/\sqrt{k})$ memory for the triangle distribution strategy (which is the memory footprint given in [36]), and the equivalence classes assigned to a worker also must fit in its memory. In smPDP, we consider two regimes of memory capacity at a worker: (1) not enough memory to use the triangle distribution strategy, and (2) even less memory such that even the equivalence classes do not fit. For regime (1), we implement the self-join using Spark's *cartesian* operation and let Spark do the memory management. For regime (2), each worker reads the required equivalence classes π_X and π_Y from external storage in chunks (small enough to fit in memory) to create $\pi_{X\cup Y}$. Note that while doing this, a worker needs to make multiple passes over the input equivalence classes.

Cost analysis: To generate the first level's equivalence classes, the input to each *groupBy* call is a column from R of size n. In the worst case, the data can be skewed such that all n tuples belong to the same group and are shuffled to a single worker, and this can happen for each column. Since each worker initially does roughly the same amount of computation in the map stage, we get $\mathbf{X} \leq nm$ and $\mathbf{Y} \leq nm/k$. The communication cost is greater than that for lmPDP for generating the first level. To generate the next

level, the cost for both memory regimes is higher than the cost in lmPDP. For regime (1), Spark's *cartesian* operation does more data shuffling than the triangle strategy. For regime (2), the cost is even greater due to the need to make multiple passes over the equivalence classes.

2.5.2 LDP2: Modified TANE

LDP1 computes new equivalence classes by intersecting pairs of equivalence classes from the previous lattice level. This requires materializing and communicating equivalence classes to workers, which is expensive. In fact, the equivalence classes for a given lattice level may be larger than the input dataset. We now suggest an alternative LDP that *computes new equivalence classes directly from the data instead of computing them using the previous level's equivalence classes.*

Figure 2.3b shows LDP2, with changes highlighted in blue. The primitive $genEQClass$ now takes as input a column combination and the tuples in r (Line 3 and 16). Also, note the difference in line 3 and 17: a lattice level now includes attribute combinations and the number of the corresponding equivalence classes, not the equivalence classes themselves.

Large-memory PDP

The implementation to generate equivalence classes for the first level and to compute dependencies is the same as in LDP1 from Section 2.5.1. To generate the next level, we again use the triangle strategy to implement the self-join, which divides new attribute combinations among workers. Equivalence classes are not materialized with the corresponding attribute combinations, but we do need to store the number of equivalence classes for each attribute combination, which is needed to $checkRefinement$. Workers then compute the new equivalence classes assigned to them directly from the data (using hashing).

Cost analysis: The cost to generate the first lattice level and compute dependencies is the same as in Section 2.5.1. The cost of the self-join is negligible because it only involves attribute combinations and not the equivalence classes. Therefore it does not depend on the number of tuples n. To generate equivalence classes for a new lattice level, each worker is responsible for roughly the same number of attribute combinations (i.e. $|L_{l+1}|/k$), and using hashing it requires a single pass over the data (i.e., nm). Doing this for up to m lattice levels gives: $\mathbf{Y} \leq nm2^m/k$, and $\mathbf{X} \leq nm * m$. When compared to lmPDP in Section 2.5.1, the communication cost of this plan is significantly lower because it avoids communicating the previous level's equivalence classes to the workers.

Furthermore, LDP2 creates opportunities for further reduction of communication cost for the Spark framework. The *Broadcast* mechanism in Spark allows data to be cached at the workers for the lifetime of a job. Thus, if each worker's memory is large enough to store the input dataset, it only needs to be sent once and can be reused for each lattice level. In our experimental evaluation (Section 2.8), we exploit this optimization whenever possible.

Small-memory PDP

We borrow the strategy from Section 2.5.1: we use multiple workers to generate equivalence classes for a particular attribute combination using Spark's *groupBy*. The difference is that we only save the number of equivalence classes, not the equivalence classes themselves, since new equivalence classes are always computed from the input dataset, not from previous level's equivalence classes.

Cost analysis: The cost of generating the first level is the same as in Section 2.5.1, and the cost of the self-join and computing dependencies is the same as in Section 2.5.2. To generate equivalence classes using *groupBy* for a particular lattice level, the number of calls made to *groupBy* is the same as the number of nodes in the lattice. In the worst case, each call will re-partition the data (i.e. n tuples) to compute *groupBy*. Hence, the cost of this smPDP is higher than the cost of the lmPDP (Section 2.5.2) because it requires much more data shuffling. However, this smPDP still has a lower cost than the smPDP in Section 2.5.1 because it avoids the expensive *cartesian* operation to create new equivalence classes by intersecting the previous level's equivalence classes (which, in the worst case, means that each worker may need to be sent the entire previous lattice level).

2.6 Case study 2: FastFDs

We now study FastFDs [93]. As explained in Section 2.3, the general strategy of data-driven algorithms is to generate evidence sets and then find minimal covers of the evidence sets. We note that the plans shown in this section can also be applied to the FastDCs algorithm for discovering DCs. The difference is that a richer predicate space will be used by $genEVSet$.

```
1  fastFD (Relation r, Schema R)
2  EV_I = {}
3  generateEvidence(r, R)
4  EV_I = sort(EV_I)
5  FDs = setCover(EV_I)
6  generateEvidence (Relation r, Schema R)
7  EQ = {}
8  for A ∈ R do
9      EQ = EQ ∪ genEQClass(A, r)
10 for π ∈ EQ do
11     for (t_i, t_j) ∈ join(π,π) do
12         EV_I = EV_I ∪ genEVSet(t_i,t_j, {≠})
```

(a) Logical discovery plan 1

```
1  fastFD (Relation r, Predicate P = {≠})
2  EV_I = {}
3  generateEvidence(r)
4  EV_I = sort(EV_I)
5  FDs = setCover(EV_I)
6  generateEvidence (Relation r)
7  for (t_i, t_j) ∈ join(r,r) do
8      EV_I = EV_I ∪ genEVSet(t_i,t_j, P)
```

(b) Logical discovery plan 2

Figure 2.4: FastFDs algorithm

2.6.1 LDP1: Original FastFDs

Figure 2.4a shows the first LDP that follows the main idea of original FastFDs algorithm [93], which is to compare only those tuple pairs which belong to the same equivalence class for at least one attribute. Lines 8-9 compute the equivalence classes for all attributes in R. Then, lines 10-12 perform a join operation on each equivalence class to compare tuples and generate evidence sets. Note that the predicate space used by $genEVSet$ consists of just inequalities (line 12), which is sufficient for FDs. Finally, we sort the evidence sets by their cardinality and compute their minimal covers (lines 4-5).

Large-memory PDP

Implementing lines 8-9 is similar to generating the first level of equivalence classes in TANE (Section 2.5.1): by distributing the columns among workers in a round-robin fashion, with each worker generating equivalence classes for multiple columns using hashing.

Next, generating evidence sets (lines 10-12) requires two jobs. First, a map-reduce job implements a self-join that joins pairs of tuples within the same equivalence class. For example, returning to Figure 2.2a, equivalence classes for A generate tuple pairs (1,3) and (2,4); equivalence classes for B generate (3,4), and so on. To implement this type of self-join in a distributed fashion, we use the *Dis-Dedup*$^+$ algorithm from [36]. This algorithm was originally proposed for data deduplication, where a dataset is partitioned into blocks, potentially by multiple partitioning functions, and tuple pairs from the same block are checked for similarity. Observe that our scenario is similar, in which a dataset is partitioned into blocks via equivalence classes and FastFDs only needs to compare tuple pairs from the same equivalence class (block). Afterward, a map job generates evidence sets, in which each worker computes evidence sets for the tuple pairs it created during the self-join.

Finally, we sort the equivalence classes and compute minimal covers. We do these steps locally at the driver node because FastFDs uses a depth-first-search strategy to compute all minimal covers, which is inherently sequential [70, 81].

Cost analysis: The cost analysis for generating equivalence classes is the same as for TANE in Section 2.5.1. Next, we examine the cost of generating evidence sets. If the size of an equivalence class j is B_j, then the number of comparisons done to generate evidence sets for all tuple pairs from this equivalence class is $B_j(B_j - 1)/2 \approx B_j^2/2$. Assuming c is the total number of equivalence classes, the total number of comparisons when generating evidence sets is $W = \sum_{j=1}^{c} B_j^2/2$. Each tuple pair comparison takes m amount of work,

therefore the total work done is $m * W$. With this, we can directly use the cost analysis of *Dis-Dedup*$^+$ from [36], which gives us $\mathbf{X} \leq 5m^2 max(n/k, \sqrt{2W/k})$, and $\mathbf{Y} \leq 5mW/k$. Note that we have m "blocking functions" and m amount of work is required to compare (all m attributes of) each tuple pair.

Small-memory PDP

To generate equivalence classes (lines 8-9 in Figure 2.4a), we again use multiple workers to generate equivalence classes for one attribute using Spark's *groupBy*, as in Section 2.5.1 for TANE. To generate evidence sets, (lines 10-12) we can off-load the (self) join operation (line 11) to Spark using the *cartesian* operation over RDDs (but we need to filter out redundant pairs). In this case, Spark internally does the memory management of the RDD, spilling to disk if required. Note that the *cartesian* operation will be called once for each equivalence class.

Cost analysis: The cost analysis of generating equivalence classes is same as in Section 2.5.1. Next, each call to the *cartesian* operation shuffles tuples from the input equivalence class across multiple workers such that some tuples might be sent to multiple (or all) workers. When combining the data shuffle across all calls to the *cartesian* operation, each worker might end up seeing close to all the input tuples multiple times. This gives a much higher communication cost compared to the lmPDP (Section 2.6.1).

2.6.2 LDP2: Modified FastFDs

The *Dis-Dedup*$^+$ algorithm is the current state-of-the-art, but it still incurs a non-trivial communication and computation cost. One problem is the redundant pair-wise tuple comparisons. Consider the equivalence classes $\pi_A = \{\{1,3\},\{2,4\}\}$ and $\pi_C = \{\{1,2,3\},\{4\}\}$ from the example in Section 2.3. According to LDP1, tuple 1 and tuple 3 are compared twice because they co-occur in two partially overlapping equivalence classes. Also, increasing the number of attributes increases the overlap of equivalence classes, thereby increasing the number of redundant pair-wise tuple comparisons. This is also evident from the m^2 factor in the cost analysis of *Dis-Dedup*$^+$ in Section 2.6.1. It is possible to eliminate this problem, but it would require an expensive comparison of all pairs of tuples in order to eliminate duplicate tuple pairs. In LDP2, we explore this idea, which trades off communication for computation. Figure 2.4b shows the pseudocode for LDP2, with changes highlighted in blue. In particular, in line 7, we perform a self *join* on the complete relation r.

Large-memory PDP

We again use the triangle join strategy from [36] to implement the *join* in line 7. This requires one map-reduce job to compute a full self-join of r and then a map job to generate evidence sets from all tuple pairs.

Cost analysis: Applying the cost analysis for triangle join strategy, we get: $\mathbf{X} \leq nm\sqrt{2/k}$ and $\mathbf{Y} \leq mn^2/2k$. This is an improvement over the cost of lmPDP in Section 2.6.1, specially when m is large, which is a typical use case for FastFDs.

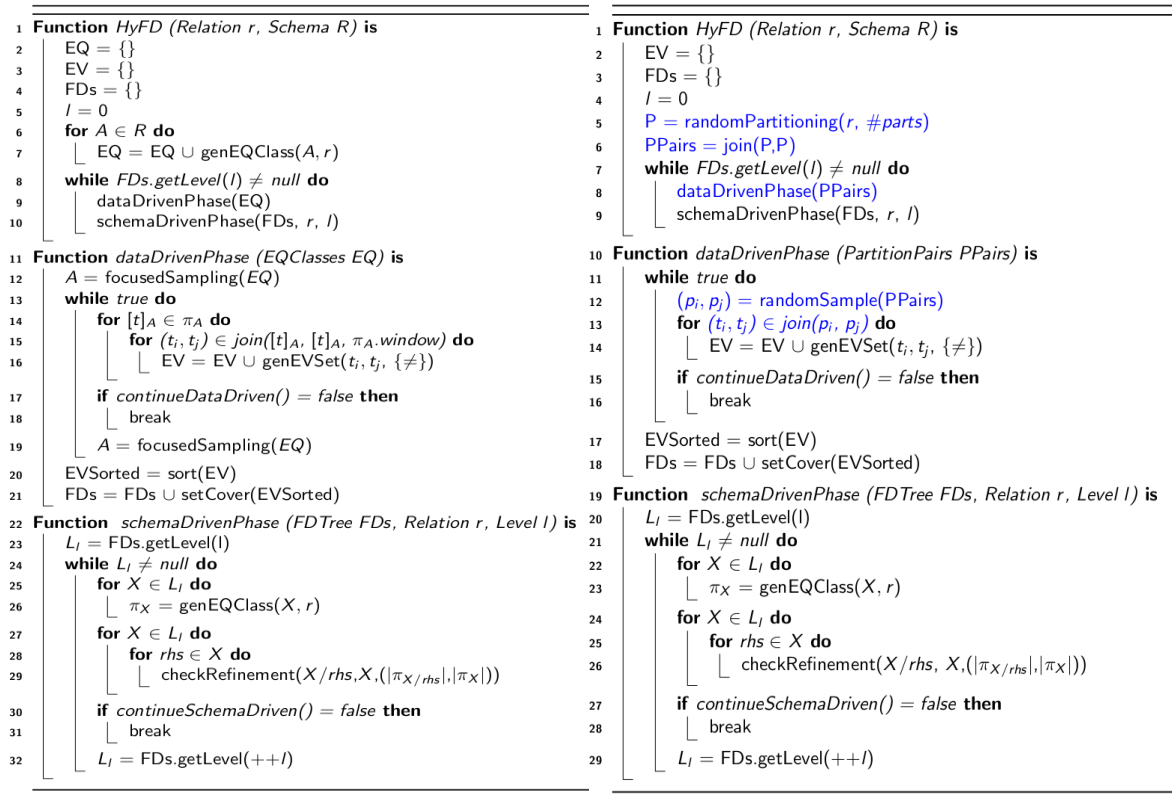
Small-memory PDP

The triangle join strategy in the lmPDP has a memory footprint of $O(nm/\sqrt{k})$ per worker. When each worker's memory is smaller than that, we off-load the *join* implementation to Spark's *cartesian* operation (we filter out redundant tuple pairs), and let Spark do the memory management.

Cost analysis: [36] showed that triangle strategy is optimal in terms of communication cost when implementing the self-join. Therefore, the cost of implementing the self-join using the *cartesian* operation cannot be lower. However, compared to the smPDP in Section 2.6.1, this implementation can still perform better when m is large because each tuple is compared exactly once.

2.7 Case study 3: HyFD

In this section, we analyze the HyFD algorithm [77]. As outlined in Section 2.3, HyFD alternates between data-driven and schema-driven phases. We note that HyUCC [78] and Hydra [32] can also be implemented using the plans described in this section, with some modifications. HyUCC [78] works similarly to HyFD, with pruning rules designed for UCC discovery. Hydra discovers DCs and hence uses a richer predicate space than FD and UCC discovery. Unlike HyFD, Hydra switches from the data-driven phase to the schema-driven phase only once, after the rate of generating DC violations drops below a user-supplied threshold.



(a) Logical discovery plan 1

(b) Logical discovery plan 2

Figure 2.5: HyFD algorithm

2.7.1 LDP1: Original HyFD

Figure 2.5a shows the LDP of HyFD based on the original algorithm [77]. As in FastFDs, it begins by generating equivalence classes for all the attributes in R (lines 6-7). Then in the data-driven phase, similar to FastFDs, it generates tuple pairs and the corresponding evidence sets. Compared to FastFDs, the difference is that not all tuple pairs are generated. Instead, HyFD picks one attribute A at a time and uses its equivalence classes to decide which tuple pairs should generate new evidence sets. To decide which attribute to use, the algorithm maintains a ranking of the attributes based on how many FDs have been violated according to their evidence sets. This process is referred to in [77] as *focused sampling* (line 12).

Next, two tuples, t_i and t_j, within the same equivalence class are compared only if

$j - i = window$, where j and i are their positions in the equivalence class, and $window$ is a threshold, with different attributes having different values of $window$. This corresponds to a join with a $window$ predicate in line 15. Whenever an attribute is selected in the data-driven phase, its $window$ value is incremented, which leads to new tuple pairs being generated.

The data-driven phase stops when newly generated evidence sets fail to identify new FD violations (encapsulated in the $continueDataDriven$ check in line 17). The evidence sets collected so far (line 16) are then used to generate FDs that have not yet been violated via set cover (lines 20-21) and inserts them into the FDTree.

The schema-driven phase traverses the FDTree level-wise; the $getLevel$ function in lines 23 and 32 retrieves all attribute sets from a given level. For each attribute set, HyFD checks which FDs hold via $checkRefinement$ (Line 29). The original HyFD implementation computes equivalence classes directly from the data (Line 26), and not by intersecting the previous level's equivalence classes. This corresponds to our LDP2 for TANE (Section 2.5.2). HyFD returns to the data-driven phase if the schema-driven phase spends too much time on a particular FDTree level (encapsulated in the $continueSchemaDriven$ function in line 30).

Large-memory PDP

We implement lines 6-7 in the same way we generated first-level equivalence classes in TANE in Section 2.5.1. Next, we use the following strategy to generate evidence sets in lines 14-16. If the selected attribute A (in line 12) has c equivalence classes, then we need to equally distribute these c equivalence classes across k workers. As in [36], we use a load balancing heuristic that arranges the equivalence classes in increasing order of their sizes, and divides them into $g = c/k$ groups, each group with k equivalence classes. Next, one equivalence class from each group is sent to a worker in round-robin fashion, such that each worker receives g equivalence classes. Each worker then uses the $window$ parameter to decide which tuple pairs to generate. Finally, a map job generates evidence sets from the tuple pairs. The implementation of generating new equivalence classes and checking refinement is same as in TANE, described in Section 2.5.2.

Cost analysis: The cost of generating equivalence classes is the same as in Section 2.5.1. The cost of generating evidence sets in one iteration of the loop in lines 13-19 in the data-driven phase is $\mathbf{X} \leq cm|B_{max}|/k$ and $\mathbf{Y} \leq cm|B_{max}|/k$ where B_{max} is size of the largest equivalence class (each worker receives c/k equivalence classes and each of them could be of size at most B_{max}).

In the worst case, if HyFD discovers all the FDs using the data-driven phase, then the data-drive phase can be performed up to n times (the size of the largest equivalence class for a given attribute can be n, and therefore the *window* threshold can be incremented up to n times). The cost of generating equivalence classes and checking refinement in the schema-driven phase is the same as in TANE in Section 2.5.2.

Small-memory PDP

In the lmPDP, we assigned multiple equivalence classes (c/k of them) to each worker. In the smPDP, we reduce the memory footprint of each worker by assigning each equivalence class to multiple workers. Essentially, we use multiple workers to perform the *join* in line 15 and then equally distribute all the generated pairs across workers to generate evidence sets. We implement the *join* using Spark's *join* operation and the *window* parameter controls which keys to join. We borrow the implementation of generating equivalence classes (lines 6-7 and 25-26) from TANE, as described in Section 2.5.2

Cost analysis: The communication cost of this implementation is higher than that of lmPDP, simply because multiple rounds of data shuffle (one for each equivalence class) are needed to generate tuple pairs. Additionally, the cost of generating equivalence classes in this PDP is higher than the cost in lmPDP in Section 2.7.1. We know this from the cost analysis of TANE in Section 2.5.2.

2.7.2 LDP2: Modified HyFD

A drawback of LDP1 is its high communication cost during the data-driven phase, which is amplified if the data-driven phase is repeated multiple times. Also, as in FastFDs, there could be redundant evidence sets due to overlapping equivalence classes.

In LDP2 (Figure 2.5b), we use the learnings from FastFDs: *instead of computing evidence sets from tuple pairs that belong to the same equivalence class, we generate tuple pairs directly from the data*. This means that focused sampling no longer applies as we are sampling tuples directly from the data and not from the equivalence classes over specific attributes. We use random sampling without replacement, as explained below, which is easier to parallelize. As before, changes are highlighted in blue. In line 5, we randomly partition the dataset into k groups, and in line 6, we generate all possible pairs of groups, including pairs of the same group. Then, the data-driven phase uses pairs of groups instead of equivalence classes. In particular, line 12 samples k pairs of groups *without replacement*,

and lines 13-14 join each pair of groups to generate tuples pairs and the corresponding evidence sets.

Large-memory PDP

Random partitioning of the data in line 5 is implemented using a simple map-reduce job: mappers assign partition IDs to tuples and reducers group tuples belonging to the same partition ID. Then, another map-reduce job implements the *join* in Line 6 which generates pairs of groups.

In line 12, we sample k pairs of groups *without replacement* which are distributed across k workers. Therefore, each worker is responsible for generating evidence sets from one pair of groups. The implementation of equivalence class generation and checking refinement is the same as in LDP1 in Section 2.7.1.

Cost analysis: With k workers, the cost of generating evidence sets in each iteration of the data-driven phase is: $\mathbf{X} \leq 2nm/k$ and $\mathbf{Y} \leq mn^2/k^2$. This is because two groups of size mn/k each are sent to every worker and every worker generates all the tuple pairs, i.e., mn^2/k^2. The data-driven phase can run up to $(k+1)/2$ times. To see this, note that every group must be joined with every other group, including itself, which amounts to $k(k+1)/2$ group-wise comparisons. With k workers in parallel, each working on one group-pair, the $k(k+1)/2$ comparisons can be packed into $(k+1)/2$ jobs. This gives $\mathbf{X} \leq (2nm/k)*(k+1)/2 \approx nm$, which is the size of the data. Compared to the lmPDP in Section 2.7.1, this implementation performs more computation but much less data communication. The cost of the schema-driven phase is the same as in TANE in Section 2.5.2. The random partitioning step in line 5 and the join in line 6 use tuple identifiers and do not involve significant computation (only a linear scan of tuple IDs), and hence their cost is negligible compared to the other operations.

In this LDP (and its lmPDP and smPDP), we use a cost-based approach to decide when to switch between the phases. That is, we switch to the other phase if the communication cost plus the computation cost of proceeding with the current phase exceeds the cost of operating in the other phase. The switching conditions are encapsulated in the *continueDataDriven* (line 15) and the *continueSchemaDriven* (line 27).

Small-memory PDP

The data-driven phase of lmPDP assumes that two groups of size nm/k each fit in each worker's memory to generate evidence sets. However, we can reduce the size of each group

by creating more groups in line 5. The drawback is that we will perform fewer comparisons in each round of the data-driven phase, hence possibly generating fewer evidence sets. The implementation of generating equivalence classes (lines 6-7 and 25-26) is borrowed from TANE, as described in Section 2.5.2.

Cost analysis: The cost of generating evidence sets in each iteration of the data-driven phase will reduce in proportion to the number of groups generated in line 5. However, the data-driven phase can run more times as compared to the lmPDP (Section 2.7.2). For example, if we generate $2k$ groups instead of k groups (as in Section 2.7.2), then the data-driven phase can run up to $(2k+1)$ times. With k workers in parallel, each working on one group-pair, the $2k(2k+1)/2$ comparisons can be packed into $(2k+1)$ jobs. This gives $\mathbf{X} \leq (2nm/2k) * (2k+1) \approx 2nm$, which is twice the communication cost in Section 2.7.2. Additionally, the cost of generating equivalence classes in this PDP is higher than the cost in Section 2.7.2. We know this from the cost analysis for TANE in Section 2.5.2.

2.8 Experiments

We now present the evaluation of different algorithms and their different implementations. In Section 2.8.2 we show that the large memory PDPs of the modified logical plans (LDP2s) are more computation and communication efficient than the large memory PDPs of LPD1s. In Section 2.8.3, we show that our smPDPs can discover dependencies when worker memory is small compared to the data size, but their runtimes are significantly higher. We then demonstrate that our implementations scale well with the number of worker machines, rows, and columns (Section 2.8.4). In Section 2.8.5 we evaluate large and small memory plans under different cluster settings. In Section 2.8.6 we evaluate the distributed implementations against single-node implementations. Finally, we examine the relative performance of various algorithms on different datasets (Section 2.8.7).

2.8.1 Experimental Setup

We performed the experiments on a 6-node Spark 2.1.0 cluster. Five machines run Spark workers and one machine runs the Spark driver. Each worker machine has 64GB of RAM and 12 CPU cores and runs Ubuntu 14.04.3 LTS. On each worker machine, we spawn 11 Spark workers, each with 1 core and 5GB of memory. The driver machine has 256GB of RAM and 64 CPU cores, and also runs Ubuntu 14.04.3 LTS. The Spark driver uses one core and 50GB of memory. We run Spark jobs in standalone mode with a total of 55 executors (11 workers times 5 worker nodes).

		LDP1	LDP2
TANE	lmPDP	- Materializes equivalence classes (EQCs) in both PDPs - Uses memory intensive triangle distribution strategy for *join*	- Computes EQCs directly from data - Computes multiple EQCs at each worker, hence memory intensive
	smPDP	- Uses *cartesian* operation for *join*	- Computes each EQC using multiple workers
FastFDs	lmPDP	- Can perform redundant tuple comparisons - Each worker works on multiple EQCs, hence memory intensive	- Uses self-join to compare tuples exactly once - Uses triangle distribution [36]
	smPDP	- Multiple workers work on each EQC	- Implements self-join via *cartesian* operation
HyFD	lmPDP	- Data-driven (DD) phase same as LDP1 lmPDP in FastFDs - Schema-driven (SD) phase same as LDP2 lmPDP in TANE	- Performs self-join split across multiple runs of this phase - SD phase same as LDP2 lmPDP in TANE
	smPDP	- DD phase same as LDP1 smPDP in FastFDs - SD phase same as LDP2 smPDP in TANE	- Compares fewer tuples in each DD phase to reduce memory - SD phase same as LDP2 smPDP in TANE

Table 2.4: Comparison of logical and physical plans

All algorithms are implemented in Java. We obtained the source code for TANE, FastFDs, HyFD, Hydra, and HyUCC from the Metanome GitHub page [8]. We obtained the source code for FastDCs and FastODs from the respective authors. We use similar datasets as those used in recent work on dependency discovery [76]: *adult*, *TPC-H lineitem*, *homicide*, *fd-reduced*, *ncvoter* and *flight*. Their properties (number of rows, number of columns, number of functional dependencies) are summarized in Table 2.8.

In Table 2.4, we summarize our physical and logical plans to help understand the results in this section. As a general guideline: (1) smPDPs are suitable when workers have small memory, but these plans reduce memory footprint at the cost of communication (**X**); (2) LDP2s are faster than LDP1s.

2.8.2 Comparison of LPD1s and LPD2s

We use two datasets to compare the two LDPs studied in Sections 2.5-2.7: one with a large number of rows (*lineitem*) and one with a large number of columns (*homicide*). To ensure that the LDP1 implementations finish within a reasonable time, we delete a fraction of rows from these datasets. For *lineitem*, we use 0.5 million rows, and for *homicide*, we use 100,000 rows. We focus on large memory PDPs for this comparison because analysis has

shown that the runtime in the small memory regime is significantly higher (also shown empirically in Section 2.8.3). Therefore, for both LDPs, we use the lmPDP as described in Sections 2.5-2.7. For each tested algorithm, we measure data shuffle amount in MBs, and we instrument the code to separately report computation time and time spent during communication (which we assume to be the total time minus the computation time).

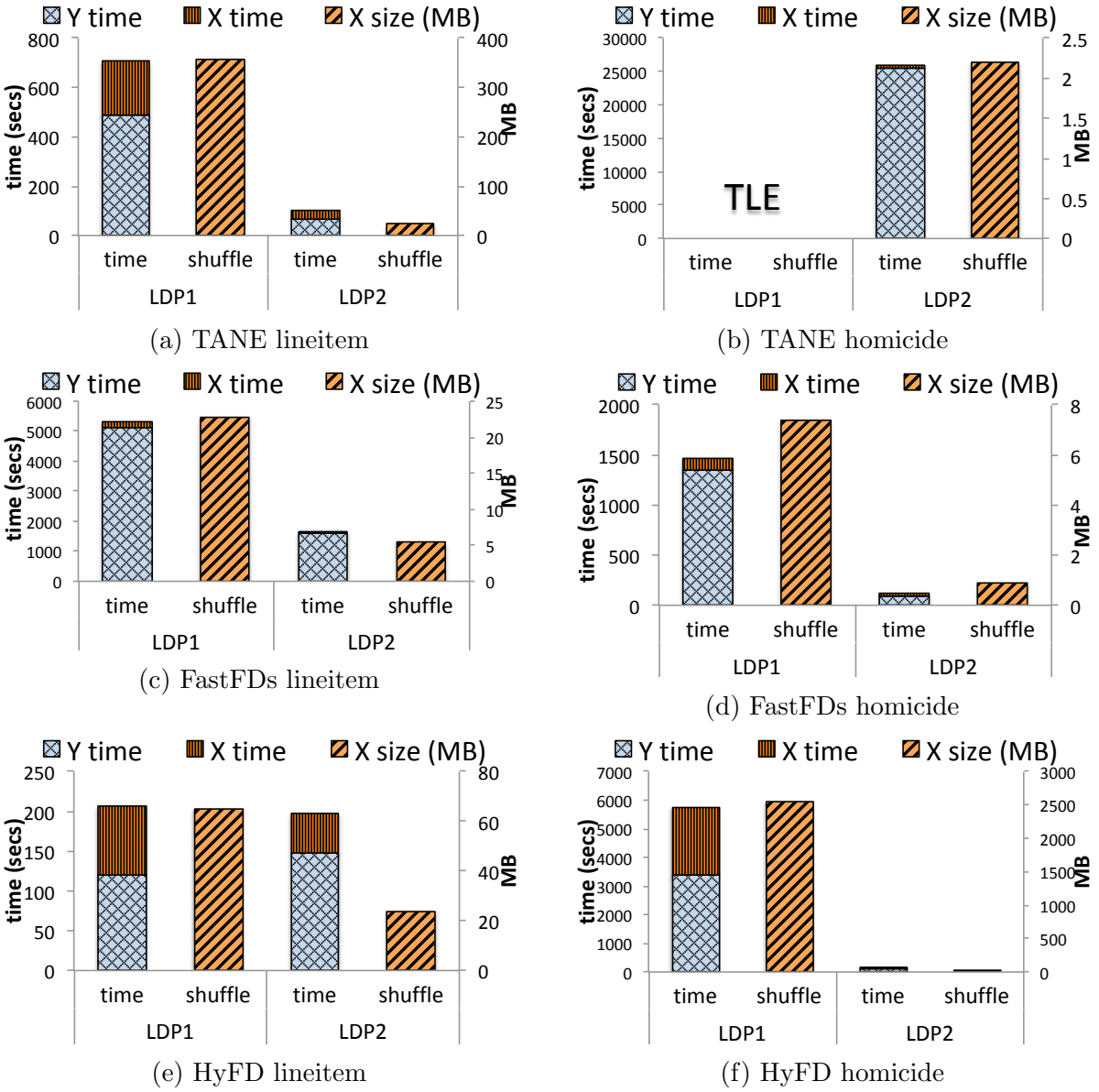
TANE: Figures 2.6(a-b) show the communication runtime ("X time"), the computation runtime ("Y time"), and the maximum data sent to any worker ("X size") for TANE LDP1 (Section 2.5.1) and LDP2 (Section 2.5.2). LDP1 exceeded the time limit of 24 hours (denoted "TLE") on the *homicide* dataset and was nearly an order of magnitude slower on the *lineitem* dataset. These speedups are in agreement with the cost analysis in Section 2.5.1 and 2.5.2. For LDP2, we cache the dataset at the workers (using Spark's *Broadcast* mechanism) to avoid re-reading it when computing equivalence classes for the next level. This explains why the communication size and time are lower in LDP2.

FastFDs: Figures 2.6(c-d) show the runtime and maximum data sent to any worker for FastFDs LDP1 (Section 2.6.1) and LDP2 (Section 2.6.2). Again, LDP2 is significantly more time and communication-efficient. In the cost analysis in Section 2.6, we pointed out that the computation and communication cost of the LDP1 becomes worse as the number of attributes increases. This is evident from Figure 2.6(c-d), where the improvement on *homicide* is 14x and on *lineitem* it is 2.5x.

HyFD: Figures 2.6(e-f) show the runtime and maximum data sent to any worker for HyFD LDP1 (Section 2.7.1) and LDP2 (Section 2.7.2). As discussed in Section 2.7, the data-driven phase can lead to a high volume of data shuffle if the dataset has many columns. This explains why LDP1 performs poorly on *homicide* which has 24 columns. We observed that due to the large schema of *homicide*, LDP2 spent most of the time in the data-driven phase. On *lineitem*, the algorithm spent more time in the schema-driven phase because of the smaller schema. On the other hand, LDP1 did a few rounds of sampling, and due to the focused sampling strategy, it was able to discover a significant number of non-FDs. This significantly pruned the search space of the schema-driven phase, allowing LDP1 to be as fast as LDP2. However, LDP2 still performs less data shuffling than LDP1 because the data are cached in the workers' memory, and therefore we only need to send it once at the beginning of the data-driven phase.

2.8.3 Comparison of smPDPs

In this experiment, we reduce each worker's memory from 5GB to 1GB. This is small enough that for our largest dataset, *lineitem*, its equivalence classes do not fit in a worker's



(a) TANE lineitem

(b) TANE homicide

(c) FastFDs lineitem

(d) FastFDs homicide

(e) HyFD lineitem

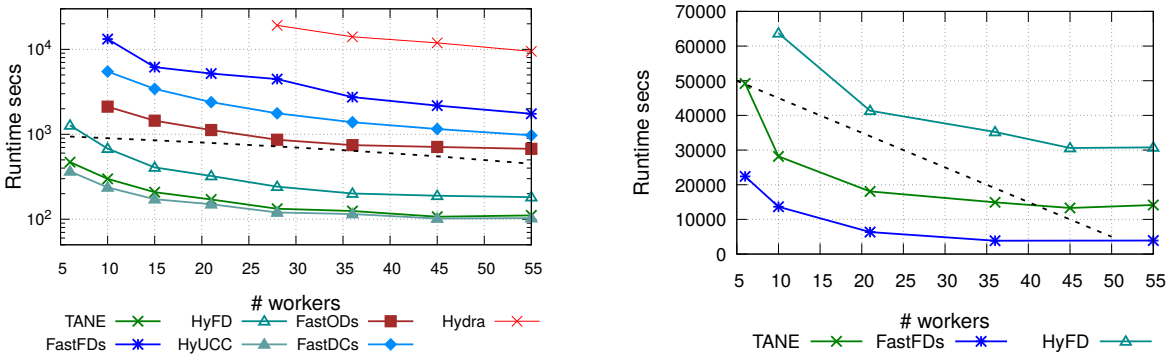
(f) HyFD homicide

Figure 2.6: Comparison of communication and computation cost of LDP1 and LDP2 of TANE, FastFDs, and HyFD. We only consider lmPDPs here.

memory. We use the smPDPs, which are designed for this scenario. We use LDP2 for each algorithm because in the previous experiment (Section 2.8.2) we saw that *LDP1* has

lineitem 6Mx16	TANE	FastFDs	HyFD
lmPDP	1.9 hrs	≈3 days	3.9 hrs
smPDP	3.9 hrs	≈106 days	8.5 hrs

Table 2.5: Runtimes of smPDPs compared to lmPDPs (for LDP2)



(a) large-memory plans (lmPDPs)

(b) small-memory plans (smPDPs)

Figure 2.7: Scalability with the number of workers of LDP2 plans

a higher runtime, which gets worse in the small memory regime.

Table 2.5 shows the runtimes for TANE, FastFDs and HyFD. The runtime of the smPDPs is almost twice as high as the corresponding lmPDPs for TANE and HyFD. For FastFDs, we extrapolate the runtimes by running it on datasets with 100K, 300K, 500K, and 700K rows because FastFDs has quadratic complexity in number of rows, and *lineitem* has 6 million rows. The smPDP of FastFDs is an order of magnitude slower than the lmPDP.

2.8.4 Scalability

We now show scalability in terms of the number of workers, the number of rows, and the number of columns. In addition to TANE, FastFDs and HyFD, we test FastODs, FastDCs, Hydra and HyUCC. We use lmPDP and LDP2 to make sure all algorithms terminate in reasonable time. For DC discovery, it has been reported in previous work that FastDCs is significantly slower than Hydra [32], so we restrict FastDCs to discover DCs with at most 5 predicates. Note that our distributed implementations perform DC discovery from

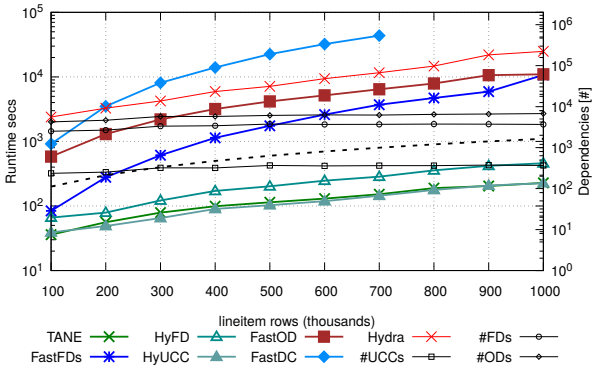


Figure 2.8: Scalability (of lmPDP of LDP2) with the number of rows (in thousands) for lineitem

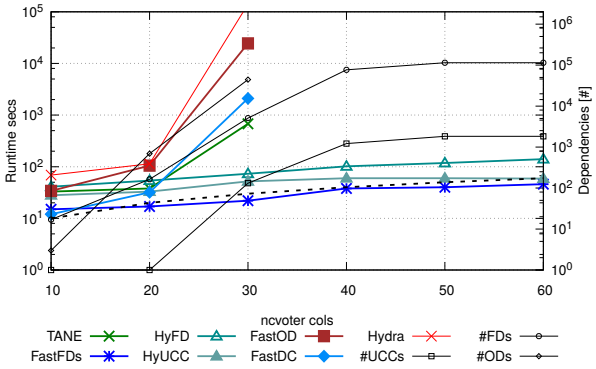


Figure 2.9: Scalability (of lmPDP of LDP2) with the number of columns for ncvoter

evidence sets centrally at the driver (by finding a minimal set cover). Therefore, this restriction has the same impact on the distributed and non-distributed implementations.

Worker Scalability

We first demonstrate nearly linear scalability with the number of workers. We vary the number of Spark workers from 6 to 55. Figure 2.7a shows the results for large memory plans of the LDP2s. We use the *lineitem* dataset with 0.5 million rows for TANE, FastFDs, HyFD, HyUCC, and Hydra and with only 100k rows for FastOD, and FastDCs (because of their high sensitivity to the number of tuples, as will be shown in Section 2.8.4). Note that the y axis is logarithmic and the dashed line shows linear scaling for reference. FastFDs

and FastDCs are impacted more by the number of workers because their complexity is quadratic in the size of the input. TANE outperforms FastFDs because the dataset has a small schema, and HyFD closely follows TANE because HyFD spent most of the time in the schema-driven phase. Scale-out of FastODs is similar to TANE, and scale-out of HyUCC and Hydra is similar to HyFD. Recall that FastODs and FastDCs are running on a smaller dataset, so their runtimes are relatively low.

Figure 2.7b shows that the small memory plans also show nearly linear scalability with the number of workers. For this experiment, we again reduced the memory of each worker to 1GB and tested the scalability on the largest dataset, i.e. *lineitem* (we ran FastFDs only on 100K rows because of its sensitivity to number of rows). We do not report the smPDP runtimes of OD and DC discovery algorithms because they exceeded the time limit and did not scale well to large data sets.

Row Scalability

Next, we test scalability with the number of rows. We use the *lineitem* dataset and test all seven algorithms: TANE, FastFDs, HyFD, HyUCC, FastOD, FastDCs, and Hydra. Results are shown in Figure 2.8 (with logarithmic y-axes), including algorithm runtimes and the number of dependencies that were discovered.

TANE and FastOD behave similarly and their runtime grows almost linearly with the number of rows. FastOD is similar to TANE but partition refinement for order dependency discovery is more expensive, resulting in much longer runtimes for FastOD compared to TANE. HyFD and HyUCC behave similarly and they closely follow the scalability of TANE; they both spend most of the time in the schema-driven phase due to smaller schema of *lineitem*. HyUCC is similar to HyFD, as described in Section 2.7.2.

FastFDs and FastDCs perform similarly and their runtime grows almost quadratically with the number of rows. However, for *lineitem*, there are 64 predicates that define the space of DCs. Thus, the minimal set cover operation in FastDCs [37] is significantly more expensive. As expected, the performance of Hydra is significantly better than FastDCs, even when we restrict FastDCs to DCs with at most 5 predicates. We also tested row scalability using *homicide* and observed similar trends.

Column Scalability

We now evaluate scalability with the number of columns. We use the *ncvoter* dataset, which has a sufficient number of columns and 10,000 rows. We restrict FastDCs, Hydra and

FastOD to fewer columns because of their high sensitivity to the schema size. Results are shown in Figure 2.9, including algorithm runtimes and the numbers of various dependencies that were discovered. Again, the y-axis is logarithmic. As expected, TANE and FastODs runtimes increase exponentially with the number of columns because these algorithms are schema-driven. FastDCs and Hydra runtimes increase exponentially because the predicates space of DCs increases significantly with the number of columns. The runtime of FastFDs stays almost linear with the number of columns, and it performs best among the FD discovery algorithms. HyFD performs similar to FastFDs due to the low cost of the data-driven phase. However, HyFD still needs to switch to the schema-driven phase and hence it does not perform as well as FastFDs. The behaviour of HyUCC is similar to HyFD. Recall that we restrict FastDCs to discover DCs only with up to 5 predicates, so its runtimes are lower than those of Hydra. We also tested column scalability using the *flight* dataset and observed similar trends.

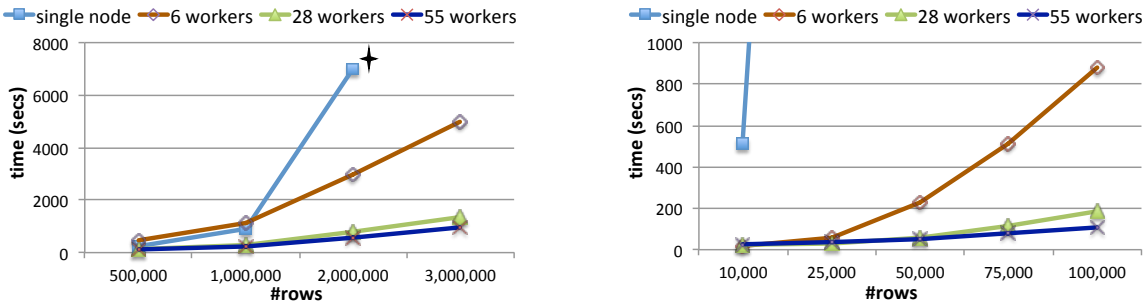
2.8.5 Experiments with different cluster settings

We now test the smPDPs and lmPDPs (of LDP2) under different cluster settings. For a fixed cluster memory of 55GB, we consider different numbers of workers and worker memory as shown in Table 2.6. With different cluster settings we try to represent different real-world cluster scenarios and the goal is to suggest implementations that are suitable for each scenario. We observe that for memory-intensive algorithms such as TANE and HyFD, the lmPDPs suffer when worker memory is low (due to thrashing) and eventually run out of memory when more workers are used with smaller memory. Therefore, running smPDP is advisable when cluster memory is small. On the other hand, FastFDs is more computation-intensive, and therefore it is always advisable to use more workers and the lmPDP (the smPDP is significantly slower because of the *cartesian* operation in Spark).

	Cluster setting		**lineitem 6Mx16**		**lineitem 0.5Mx16**
Plans	# workers	worker-memory	TANE	HyFD	FastFDs
smPDP	55	1GB	3.9 hrs	8.5 hrs	17.4 hrs
lmPDP	28	1.9GB	OOM	OOM	1.1 hrs
lmPDP	15	3.6GB	5.7 hrs	10.6 hrs	2.0 hrs
lmPDP	10	5.5GB	4.2 hrs	9.6 hrs	2.9 hrs

Table 2.6: Runtimes under different cluster settings

We also test the smPDP and lmPDP plans under different worker memory settings (1GB, 2GB, and 4GB), keeping the number of workers fixed at 55. The goal of this



(a) TANE on lineitem

(b) FastFDs on homicide

Figure 2.10: Single-node vs distributed performance

experiment is to determine if more memory helps. We observe that as long as there is enough memory for the dataset and for the intermediate results, increasing memory does not impact the runtime. In fact, when there are many small jobs (as in the smPDPs), over-provisioning can be harmful because Spark's garbage collection runs more frequently.

2.8.6 Distributed vs. Single-Node Runtimes

We now compare the performance of single-node (or non-distributed) implementations against the distributed implementations (lmPDP LDP2). We run the single machine implementations on one machine from our cluster (12 CPU cores and 64GB RAM). We run the distributed implementations on 55 workers with 5GB memory each. We use *lineitem* and *homicide* datasets.

Table 2.7 shows that TANE single-node ran out of memory on *lineitem* with 6M rows whereas the distributed version finished in about 2 hours. Both single-node and distributed versions exceeded the time limit of 24 hours for *homicide* because of the large schema. Figure 2.10a further shows the runtimes for different sizes of the *lineitem* dataset, where the single-node implementation ran out of memory at about 3M rows, and the distributed implementation took less time than the single-node implementation, even with only six workers (having 5GB of memory each). We also tested our LDP2 (which is more computation intensive) of TANE on a single node, and found that it performed about 5x slower than the original implementation using the *lineitem* dataset with 500K rows.

FastFDs benefits from the parallelism of the distributed implementations. The single-node implementation exceeded the time limit of 24 hours for all datasets except *homicide* 100K rows (Table 2.7), whereas the distributed version finished in much less time for all

but the *lineitem* 6M rows dataset. Figure 2.10b further shows the runtimes for different sizes of the *homicide* dataset and different numbers of workers. Even with only six workers, the distributed algorithm's runtime is significantly lower than the single-node runtime. We also tested our LDP2 on a single node and found that it performed 5x faster than the original single-node implementation using the *homicide* dataset with 100K rows.

HyFD[1] is the most memory and computation efficient single-node algorithm, and outperforms the distributed version as long as the dataset fits on a single machine (Table 2.7). When we reduced the single machine memory to 8GB, the *lineitem* dataset did not fit in memory; however, smPDP terminated in about 8 hours when executed on a Spark cluster with five machines (with 55 workers) restricted to 8 GB RAM each (40GB total, 0.7GB per worker).

	Single node				**Distributed LDP2 lmPDP**		
Dataset (#rows)	TANE	FastFDs	HyFD	HyFD 12 threads	TANE	FastFDs	HyFD
lineitem (500K)	413	TLE	124	43	100	1617	197
homicide (100K)	OOM	71581	115	74	25832	114	179
lineitem (6M)	OOM	TLE	3396	1124	6854	TLE	14311
homicide (0.6M)	TLE	TLE	745	683	TLE	3152	3113

Table 2.7: Runtimes (in seconds) of single-node and distributed implementations

2.8.7 Experiments on Different Datasets

Finally, we evaluate (lmPDP implementations of LDP2s of) TANE, FastFDs and HyFD on several datasets with at least 14 and up to 109 columns. We omit FastDCs, Hydra and FastOD because these algorithms do not perform well on datasets with a large number of columns. Results are shown in Table 2.8. *Adult* is the smallest dataset, and all three algorithms finished in a reasonable time. *lineitem* has a large number of rows (6 million), meaning that FastFDs struggles but TANE and HyFD perform better. However, HyFD takes longer than TANE because, as mentioned before, HyFD incurs the overhead of creating partitions and it does not prune keys. *homicide* and *ncvoter* are examples where HyFD switches between the two phases and discovers FDs the fastest. For *ncvoter*, FastFD ran out of memory at the driver because the search space for minimal set covers grew large. For *fd-reduced*, TANE performs best because almost all of the discovered FDs are present in the third level of the lattice; this is observed in HyFD [77] as well. For the *flight* dataset,

[1]HyFD is the only existing algorithm that has a multi-threaded implementation in Metanome. We run it on 12 threads, which equals the number of physical cores on our machine.

HyFD spent most of the time in the data-driven phase, but it still had to validate millions of FDs in the schema-driven phase, and hence it could not beat FastFDs.

Recent work [76, 77] has compared FD discovery algorithms on similar datasets and concluded that schema-driven algorithms are suitable for datasets with many rows and data-driven algorithms are suitable for the datasets with many columns. Hybrid algorithms perform best by spending most of their time in either the data-driven phase or the schema-driven phase, depending on their relative cost. We observed similar trends in the distributed versions of these algorithms (and additionally explored scalability with the number of workers).

Dataset	# Columns	# Rows	# FDs	TANE	FastFDs	HyFD
adult	14	32,560	78	50 secs	**23 secs**	101 secs
lineitem	16	6,000,000	4,145	**1.9 hrs**	>48 hrs	3.9 hrs
homicide	24	600,000	637	38.1 hrs	53 mins	**51 mins**
fd-reduced	39	250,000	89,571	**86 secs**	648 secs	228 secs
ncvoter	60	1,000,000	2,638,634	>48 hrs	MLE	**43.2 hrs**
flight	109	1,000	1,150,815	>24 hrs	**99 secs**	351 secs

Table 2.8: Runtimes on different datasets

Chapter 3

A semi-supervised framework of clustering selection for de-duplication

As the data accumulates from multiple sources over time, many errors creep into the data. For example, many records end up having duplicate entries. Data de-duplication is a central task in managing large scale databases. The goal is to detect records in a database that correspond to the same real word entity.

The problem of data de-duplication can be viewed as a clustering task. Here, the goal is to put records corresponding to the same physical entity in the same cluster while separating the records corresponding to different entities into different clusters. Clustering for de-duplication has many characteristics which are different from standard clustering problems. Many popular clustering algorithms like k-means or k-median receive as input the value k, that is the number of clusters to output. This information is unknown in de-duplication applications. In any dataset, the number of different-cluster pairs (i.e. different entity pairs) is order of magnitude greater than the number of positive or same-cluster pairs (i.e. same entity pairs). Hence, common machine learning tools of classification prediction (learning a binary classifier over the set of pairs of instances) do not automatically transfer to this domain as the dataset is heavily skewed towards the negative pairs.

The framework of correlation clustering is very natural for modelling the problem of data de-duplication [26]. Here, de-duplication is viewed as an optimization problem over graphs. More formally, given a dataset X and a complete graph G over the set. The edges of the graph are labelled 0 or 1. An edge label of zero indicates that the corresponding vertices have been deemed to be in different cluster while an edge label of one indicates that the corresponding vertices should be in the same cluster. The motivation for edge labelling

is the following. Often the practitioner can design a pairwise similarity function over the pairs of points. The pairs whose similarity is above a certain threshold are deemed as positive (or same-cluster) and the remaining pairs are deemed to be negative (or different-cluster). Sometimes, the similarity metric is also learned from training data.

Given the graph, the goal of correlation clustering is to find a clustering of the dataset which correlates 'as much as possible' with the given edges. In other words, find a clustering which minimizes the correlation loss w.r.t the given edges. Correlation loss is defined as the sum of edges labeled zero within a cluster plus the number of edges labeled one across different clusters. However, solving this optimization problem is NP-Hard [26].

Recently, [60] introduced the framework of restricted correlation clustering (RCC) to model de-duplication problems. This framework has two important differences from the standard correlation clustering formulation. Firstly, the goal is to find a clustering from a given class $\mathcal{F}$ of clusterings. In correlation clustering, the optimization problem was to find a clustering over the set of all possible clusterings of the dataset. In the framework of RCC, the optimization problem is restricted over a given class $\mathcal{F}$ of clusterings. Secondly, the goal of correlation clustering was to find a clustering which correlates as much as possible with the given edges. In this framework, the goal is to find a clustering which correlates as much as possible with the unknown target clustering. Formally, given a set X, an unknown target clustering C^* of X and a class $\mathcal{F}$ of clusterings of X, the goal is to find a clustering C from the set $\mathcal{F}$ which minimizes the correlation loss w.r.t the unknown target clustering; the sum of different-cluster edges (w.r.t C^*) within a C-cluster plus the sum of same-cluster edges across different C-clusters.

This framework is highly suitable for de-duplication applications. The target clustering C^* should be understood as the ground truth clustering. That is, C^* is the clustering where only records corresponding to same entity are in the same cluster. The more interesting aspect of the framework is introduction of the class $\mathcal{F}$. For many real-world applications, the optimal solution to the correlation clustering is the desired solution. However, the negative NP-Hardness results make it infeasible to find that solution. In such scenarios, a reasonable objective could be the following. Run a variety of efficient clustering algorithms available and obtain different clusterings of the dataset. Then choose the clustering which is 'closest to' the ground truth clustering. For example, one can have $\mathcal{F} = \{T_1, \ldots, T_s\}$ where each T_i is a hierarchical clustering of X. Then the goal of RCC is to find a pruning in the hierarchical clustering which is closest to the ground truth clustering.

The framework of RCC tries to find a clustering which is closest to the ground truth clustering. In the absence of any information about the ground truth, one can not hope to solve the problem. The framework provides 'indirect' information about the ground truth

in the following two ways. Firstly, we allow the algorithm to make *same-cluster* queries to an oracle. The oracle should be understood as a human expert who has knowledge about the ground truth clustering. Given two records from a dataset the expert answers yes or no depending upon whether the two records refer to the same entity or not. Secondly, the user designed or learned similarity (or distance) function provides indirect information about the ground truth. Pairs of records whose distance is below a certain threshold are likely to belong to the same cluster (correspond to the same entity).

To solve the RCC problem, we adopt the following strategy. We get a small set of labelled samples of pairs of points with the help of our oracle. Our sampling procedure uses two sub-procedures. One for sampling negative or different-cluster pairs and one for positive or same-cluster pairs. We then choose the clustering which makes the smallest number of 'mistakes' on the sample. In our work [59], we show that this strategy is theoretically sound. Our method only finds a clustering which *performs best* on the sampled pairs of points. But we prove that this sampling based approach is guaranteed to find a clustering which is 'close' to the best clustering in $\mathcal{F}$. Informally, the best performer on the sample is guaranteed to be close to the best true performer or the optimal solution of the RCC problem. A more formal description of the results is deferred to Section 3.3.

Another important contribution of this work is the sampling sub-procedure for positive pairs. In many datasets, the similarity function is such that it supports locality sensitive hashing (LSH). We use this fact to obtain a procedure which requires only linear pre-processing time and can sample according to the distribution P^+ (the uniform distribution over the same-cluster pairs). We also prove that the number of queries to the same-cluster oracle (to sample one positive pair) is upper bounded by a small constant and is independent of the size of the dataset. We carry extensive experimental evaluation of our framework on a diverse class of clustering algorithms and across multiple real world datasets.

3.1 Preliminaries

Given X, a clustering C of the set X partitions it into k disjoint subsets or clusters. The clustering C can also be viewed as a $\{0,1\}$-function over the domain $X^{[2]} := \{(x_1, x_2) : x_1 \neq x_2\}$. Here, $C(x_1, x_2) = 1$ iff x_1, x_2 belong to the same cluster according to C.

We allow a clustering algorithm to make queries to a human oracle in the following way.

Definition 10 (Same-cluster oracle [23]). *Given a set X and an unknown target clustering C^*. A same-cluster C^*-oracle receives a pair $(x_1, x_2) \in X^{[2]}$ as input and outputs 1 if and only if x_1, x_2 belong to the same cluster according to C^*.*

From the perspective of de-duplication, a same-cluster oracle receives two records x_1 and x_2. The oracle returns 1 if x_1 and x_2 correspond to the same real-world entity. Otherwise, the oracle responds 0.

Definition 11 (Correlation loss[26]). *Given graph $G = (X, E)$ where X is the set of vertices (the given dataset to be clustered) and E is the set of edges. The correlation loss of a clustering C w.r.t the edges E is defined as*

$$\begin{aligned} corrL_E(C) &= corrN_E(C) + corrP_E(C), \ \textit{where} \\ corrN_E(C) &= |\{(x,y) : C(x,y) = 1 \ \textit{and} \ E(x,y) = 0\}|, \\ corrP_E(C) &= |\{(x,y) : C(x,y) = 0 \ \textit{and} \ E(x,y) = 1\}| \end{aligned} \tag{3.1}$$

A weighted version of the loss function places weights of w_1 and w_2 on the two terms and is defined as

$$corrL_E^{w_1,w_2}(C) = w_1\, corrN_E(C) + w_2\, corrP_E(C) \tag{3.2}$$

The goal of correlation clustering is to find a clustering which minimizes the (weighted) correlation loss. For our purposes, it is more relevant to consider a loss function which takes values in the range $[0, 1]$. Also, we consider the correlation loss w.r.t a target clustering C^* rather than the edges E.

Definition 12 (Normalized correlation loss[60]). *Given domain X and a target clustering C^*. The loss of a clustering C w.r.t the target C^* is defined as*

$$\begin{aligned} L_{C^*}(C) = \ \mu \ L_{P^+}(C) \ + \ (1-\mu) \ L_{P^-}(C), \ \textit{where} \\ L_{P^+}(C) = \mathop{\mathbf{P}}_{(x,y)\sim P^+}\big[C(x,y) = 0\big], \\ L_{P^-}(C) = \mathop{\mathbf{P}}_{(x,y)\sim P^-}\big[C(x,y) = 1\big] \end{aligned} \tag{3.3}$$

where P^+ is the uniform distribution over $X_+^{[2]} = \{(x,y) : C^*(x,y) = 1\}$ and P^- is the uniform distribution over $X_-^{[2]} = \{(x,y) : C^*(x,y) = 0\}$.

The normalized correlation loss measures two quantities for the clustering C. The first is the fraction of the true positive pairs that C gets wrong (or loss over the positive pairs). The second is the fraction of true negative pairs that C gets wrong (or the loss over the negative pairs). It then obtains a weighted sum of the two losses.

Lets observe the relation between Defns. 11 and 12. Define $\gamma_0 := \mathbf{P}[C^*(x,y) = 1]$, that is the probability of true positive (or same-cluster pairs) in the dataset. Using the notation of Defn. 11, we see that $corrP_{C^*}(C) = \gamma_0 |X^{[2]}| L_{P^+}(C)$ and $corrN_{C^*}(C) = (1-\gamma_0)|X^{[2]}|L_{P^-}(C)$. Normalising by $|X^{[2]}|$ and choosing $\mu = \frac{w_2\gamma_0}{w_1(1-\gamma_0)+w_2\gamma_0}$ gives us the normalized version of the loss function.

Definition 13 (Informative metric [60])**.** *Given a metric space (X,d), a target clustering C^* and a parameter λ. We say that the metric d is (α,β)-informative w.r.t C^* and λ if*

$$\mathop{\mathbf{P}}_{(x,y)\sim U^2} \left[d(x,y) > \lambda \mid C^*(x,y) = 1 \right] \leq \alpha \tag{3.4}$$

$$\mathop{\mathbf{P}}_{(x,y)\sim U^2} \left[C^*(x,y) = 1 \mid d(x,y) \leq \lambda \right] \geq \beta \tag{3.5}$$

Here U^2 is the uniform distribution over $X^{[2]}$.

In deduplication applications, often the distance function is such that pairs with distance within a certain threshold are likely to be in the same cluster. The definition of an informative metric formalizes this intuition. It says that most of the true positive pairs have a distance of atmost λ between them. Also, amongst all pairs with distance $\leq \lambda$, at least a β fraction of them belong to the same cluster.

Definition 14 (γ-skewed)**.** *Given X and a target clustering C^*. We say that X is γ-skewed w.r.t C^* if*

$$\mathop{\mathbf{P}}_{(x,y)\sim U^2} \left[C^*(x,y) = 1 \right] \leq \gamma$$

In de-duplication applications, most of the pairs are negative (or belong to different clusters). The above definition states this property formally. We are now ready to introduce the framework of *restricted correlation clustering*.

3.2 Restricted Correlation Clustering (RCC)

We know that finding the clustering which minimizes the correlation loss is NP-Hard. Moreover, it is NP-Hard even when we are allowed access to a same-cluster oracle [17].

Observe that the requirement of correlation clustering is very demanding. The algorithm is required to find a clustering over the set of all possible clusterings of the domain X. In the restricted framework, we change the goalpost slightly. The algorithm is now required to find a clustering C from a finite class $\mathcal{F}$ (of clusterings of X).

Definition 15 (Restricted correlation clustering (RCC))**.** *Given a clustering instance (X, d), an unknown target clustering C^* and weighting parameter μ. Given a finite class $\mathcal{F}$ of clusterings of the set X. Find $C \in \mathcal{F}$ such that*

$$\hat{C} = \arg\min_{C \in \mathcal{F}} \; L_{C^*}(C) \tag{3.6}$$

3.2.1 Relation to practical applications

Consider the following scenario from the practitioner's point of view. The practitioner wants to implement correlation clustering. However, he/she knows that the problem is NP-Hard. The practitioner has prior knowledge that one of the many hierarchical clustering algorithms (like single-linkage or max-linkage or average-linkage or complete-linkage) is suitable for his/her dataset[1]. A hierarchical clustering algorithm outputs a clustering tree. Every pruning of the tree is a clustering of the original dataset. He/she would now like to know which amongst these clustering algorithms is suitable for his task. After having fixed the algorithm, the practitioner would then like to know which amongst these many prunings he/she should chose.

The framework of restricted correlation clustering is applicable in such scenarios. When $\mathcal{F} = \{T\}$ where T is a hierarchical clustering of X, the goal of RCC is to find the pruning from the tree T which has minimum normalized correlation loss. When $\mathcal{F} = \{T_1, \ldots, T_s\}$ where each T_i is a hierarchical clustering of X. Then the goal of RCC is to find a pruning with minimum loss amongst the prunings of all the s trees. Note that finding the pruning of the tree is the same as choosing the stopping point criteria when running linkage-based algorithms. Hence, the framework can help us choose the right stopping point for a particular hierarchical clustering algorithm.

If $\mathcal{F} = \{C_1, \ldots, C_s\}$ where each C_i is a clustering of the set X then the goal is to find a clustering with minimum loss. Note that $\mathcal{F}$ can be any of the examples as defined above or a union of these or some other finite class.

[1] A nice overview of hierarchical clustering techniques can be found in [69]

3.2.2 Solution strategy

In the RCC framework, we wish to minimize the loss which depends on the unknown target clustering C^*. However, in the absence of any information about C^*, there is no hope to find a clustering that minimizes L_{C^*}. Hence, to solve the RCC problem we allow the clustering (or learning) algorithm to make queries to a C^*-oracle.

Our goal is to calculate quantities $L_{P^+}(C)$ and $L_{P^-}(C)$ (Defn. 12) for each of the clusterings $C \in \mathcal{F}$ and then choose the clustering with minimum loss. To calculate both these quantities exactly, for each pair of points in our dataset, we would need to know whether they belong to the same-cluster or different-cluster. In other words, we would need access to the complete ground truth clustering C^*. Thus, instead of calculating these two quantities exactly we want to estimate them from a small sample, sampled according to the distributions P^+ and P^-.

One strategy to estimate $L_{P^+}(C)$ (and L_{P^-}) could be the following. Sample a set S_+ (and S_-) of pairs using the distribution P^+ (and P^-). Compute the fraction of mistakes made by each clustering C on S_+ (and S_-). Using the standard results from vc-dimension theory, it is known that using this procedure we can estimate L_{P^+} for each of the clusterings $C \in \mathcal{F}$. Similarly, we could also estimate L_{P^-}. Using the two estimates, we could then estimate the loss L_{C^*} for each of the clusterings in our class and choose the clustering which has the smallest loss.

The main problem in this approach is that the distributions P^+ and P^- are unknown (as the target clustering C^* is not known). In Section 3.3, we discuss two approaches which (approximately) sample according to these distributions. Then, we show how these sampling procedures can be used to estimate L_{C^*} for all the clusterings in our class $\mathcal{F}$.

3.3 Sampling for Restricted Correlation Clustering

We first describe the procedure $\mathcal{P}_0$ which samples according to P^-. Then we describe the procedure $\mathcal{P}_1$ which samples approximately according to the distribution P^+. The procedure $\mathcal{P}_0$ is the same as described in [60]. However, we state the algorithm here for completeness.

The procedure samples a pair uniformly at random. Then using the oracle it checks if the sampled pair is negative and terminates if such a pair is found. If not then the process is repeated again. From the algorithm description it is clear that to sample one negative pair we might need to make more than one query to the C^*-oracle. However, since the

Algorithm 1: Procedure $\mathcal{P}_0$ for negative pairs

Input: A set X and a C^*-oracle.
Output: (x, y) such that $C^*(x, y) = 0$

```
while TRUE do
    Sample (x, y) using U^2
    if C*(x, y) = 0 then
        Output (x, y)
    end
end
```

number of negative pairs is much greater than the number of positive pairs (γ-skewed) the number of 'wasted' queries to the oracle is small. The proof[2] of this result is given in Lemma 17 of [59].

3.3.1 Sampling positive pairs

We now discuss our procedure $\mathcal{P}_1$ which approximates the distribution P^+. We show that the procedure samples according to a distribution T which has the following property. The loss $L_{\mathcal{T}}$ (Defn. 12) and the loss L_{P^+} for any clustering are close to one another. Hence, estimating the loss of a clustering w.r.t the distribution $\mathcal{T}$ also gives an estimate of the loss of that clustering w.r.t P^+. Now, we discuss the details of the sampling procedure.

Our metric d is (α, β)-informative w.r.t the target clustering C^*. That is, amongst all pairs with distance λ at least β-fraction are positive. The sampling strategy of [60] was the following. Construct a set $K = \{(x, y) : d(x, y) \leq \lambda\}$ and then sample uniformly from the set K till a positive sample is found. Since most of the positive pairs have distance $\leq \lambda$, this sampling procedure approximates P^+ (the uniform distribution over the set of true positives). However, constructing the set K requires $\Theta(|X|^2)$ time. This makes the sampling procedure impractical for many situations. In this section, we will use techniques from locality sensitive hashing (LSH) combined with rejective sampling to develop a sampling procedure $\mathcal{P}_1$. We will show that $\mathcal{P}_1$ needs only linear pre-processing time (to build the hash maps) and outputs a positive pair sampled approximately according to P^+.

[2]Some of the Theorems in this Section are contributions of Shrinu Kushagra, hence omitted from this dissertation. Instead, we provide references to the Theorems and Lemmas in [59], where they were originally presented as a joint work.

Locality Sensitive Hashing (LSH)

Before we describe our technique, we introduce some relevant notation. A hash function $h : X \to \mathbf{N}$ maps the set X onto the set of natural numbers. Thus, a hashing function partitions the input of size n into $m \leq n$ different buckets (or blocks) $B_1, \ldots, B_m$ where each $B_i = \{x : h(x) = b_i\}$ for some b_i. Given (X, d), a Locality Sensitive Hashing (LSH) scheme w.r.t the distance metric d (or a similarity metric) aims to partition X into buckets such that 'similar' items map to the same bucket with high probability and 'dissimilar' items end up in different buckets with high probability. For example, MinHash scheme w.r.t Jaccard similarity measure [34, 33] is a common LSH-based hashing scheme. Another example is SimHash scheme w.r.t hamming similarity measure [35].

Definition 16 (LSH-based hashing algorithm)**.** *Given a set (X, d) and parameter s. An LSH-based hashing algorithm (or scheme) $\mathcal{A}$ outputs s different partitions $P_1, \ldots, P_s$ of X. Denote $P_i = \{B_{i1}, \ldots, B_{in_i}\}$. We say that $\mathcal{A}$ is (ϵ, ϵ')-tight w.r.t d and λ, λ' if*

- *If $d(x, y) \leq \lambda$ then $\mathbf{P}[b(x, y) = 1] > 1 - \epsilon$*
- *If $d(x, y) > \lambda'$ then $\mathbf{P}[b(x, y) = 1] < \epsilon'$*

where $b(x, y) = 1$ if and only if x, y are together in atleast one of the blocks B_{ij}.

Infact, in [59] it is shown that by choosing s (and other parameters) appropriately, we can construct LSH schemes which are $(\epsilon, \epsilon' = s\ln(1 + \epsilon))$-tight w.r.t λ and $\lambda' = 2\lambda\ln(1 + 1/\epsilon)$. Thus, for simplicity of notation, we say that $\mathcal{A}$ is ϵ-tight w.r.t λ to mean that it is (ϵ, ϵ')-tight w.r.t λ, λ' as chosen above.

The ϵ-tightness assumption of the hashing scheme means that any (x, y) pair which is farther in the distance metric has low probability of being together in any of the blocks created by the hashing function. Similarly, any pair which is close in the distance metric have a high probability of being together in at least one of the blocks. Throughout the remainder of this section, we will assume that the hashing scheme satisfies ϵ-tightness.

We now describe our sampling procedure. Let $\mathcal{B} := \{P_1, \ldots, P_s\} = \{B_{ij} : 1 \leq i \leq s, 1 \leq j \leq |P_i|\}$ be the set of blocks outputted by the hashing scheme and let $Q := \{(x, y) \in B_{ij}\}$. We first choose a block $B \in \mathcal{B}$ with probability proportional to $|B|^2$ (the number of pairs in the block). Then we sample a pair uniformly at random from this block B. Note that this strategy doesn't give us a uniform sample from Q. This is because a pair (x, y) may be present in multiple blocks. To get the uniform sample, we reject the pair with probability

inversely proportional to $a(x, y)$ (the number of blocks in which x, y are together). This approach based on rejection sampling ensures that we have a uniform sample from Q.

Next, we check if the pair satisfies $d(x, y) \leq \lambda$. Note that the LSH-based scheme tries to put similar points in the same bucket, hence the probability of success at this step is 'high'. Finally, we check if $C^*(x, y) = 1$. Our sampling procedure $\mathcal{P}_1$ is described in Alg. 2.

Algorithm 2: Sampling procedure $\mathcal{P}_1$ for positive pairs

Input: A set X, a hashing algorithm $\mathcal{A}$, a C^*-oracle and parameter λ.
Output: (x, y) such that $C^*(x, y) = 1$

Pre-compute:

Use an LSH-based hashing scheme $\mathcal{A}$ to obtain partitions $\{P_1, \ldots, P_s\}$.
$\mathcal{B} := \{P_1, \ldots, P_s\} = \{B_{ij} : 1 \leq i \leq s, 1 \leq j \leq |P_i|\}$.

Sampling:

while *TRUE* **do**
    Sample a block B from $\mathcal{B}$ with probability $\propto |B|^2$.
    Sample (x, y) uniformly at random from B^2.
    Let $a(x, y) = \{(x, y) \in B^2 : B \in \mathcal{B}\}$.
    Sample u uniformly at random from $[0, 1]$.
    if $u > \frac{1}{|a(x,y)|}$ **then**
        continue.
    end
    if $d(x, y) \leq \lambda$ *and* $C^*(x, y) = 1$ **then**
        Output (x, y).
    end
end

Theorem 8 in [59] shows that with high probability the procedure $\mathcal{P}_1$ samples a pair according to a distribution $\mathcal{T}$ which approximates P^+. To sample one same-cluster pair, we might need to make more than one same-cluster query to the C^*-oracle. Lemma 9 in [59] shows that with high probability, the number of queries made by $\mathcal{P}_1$ to sample one positive pair is upper bounded by a small constant.

The pre-compute stage uses a hashing algorithm to obtain s different partitions, which runs in $O(n)$ time ($n = |X|$). Theorem 10 in [59] shows that under reasonable assumptions, the time taken to sample one same-cluster pair is upper bounded by a constant with high probability.

So far we have seen how to sample (approximately) according to the distributions P^+ and P^-. We sample a 'small' set of true positive (or same-cluster) and true negative (or different-cluster) pairs using our distributions. We then choose the clustering $\hat{C} \in \mathcal{F}$ with the minimum number of mistakes on the sampled pairs. We describe this procedure in Alg. 3.

Algorithm 3: Empirical Risk Minimization

Input: (X, d), a set of clusterings $\mathcal{F}$, a C^*-oracle, parameter λ and sizes m_+ and m_-.

Output: $C \in \mathcal{F}$

1 Sample a sets S_+ and S_- of sizes m_+ and m_- using procedures $\mathcal{P}_1$ and $\mathcal{P}_0$ respectively.

2 For every $C \in \mathcal{F}$, compute

$$\hat{P}(C) = \frac{|\{(x,y) \in S_+ : C(x,y) = 0\}|}{|S_+|}$$
$$\hat{N}(C) = \frac{|\{(x,y) \in S_- : C(x,y) = 0\}|}{|S_-|}$$

3 Define $\hat{L}_{C^*}(C) = \mu\hat{P}(C) + (1-\mu)\hat{N}(C)$.

4 Output $\arg\min_{C \in \mathcal{F}} \hat{L}_{C^*}(C)$

Theorem 11 in [59] shows that as long as the number of labelled positive (m_+) and negative (m_-) pairs are in $O(\frac{\text{VC-Dim}(\mathcal{F})}{\epsilon^2})$ then our algorithm finds a clustering $\hat{C}$ which is close to the best clustering in $\mathcal{F}$. Here, VC-Dim is a combinatorial property which measures how 'complex' or rich the class of clusterings is. Note that the number of samples needed is independent of the size of the dataset X.

For common classes, like $\mathcal{F} = \{T_1, \ldots, T_s\}$ where each T_i is a hierarchical clustering of X, [60] showed that the VC-Dim($\mathcal{F}$) is in $o(\log s)$. Thus for such classes a small number of samples suffice to find a clustering which is close to the best clustering in $\mathcal{F}$.

3.4 Evaluation

We now present the evaluation of our framework on a simulated and four real world datasets. In Section 3.4.2 we show that our framework is generic and can be used to

	simulated		publications		products I		products II		restaurants	
Clustering	true loss	estimated loss	true loss	estimated loss	true loss	estimated loss	true loss	estimated loss	true loss	estimated loss
ArtPt	0.091	0.105	0.023	0.005	0.206	0.170	0.153	0.160	0.094	0.110
Star	0.052	0.060	0.100	0.050	0.207	0.190	0.231	0.170	0.041	0.045
ApproxCorr	NA[a]	NA	0.180	0.145	0.380	0.310	0.373	0.340	0.094	0.065
Markov	0.011	**0.000**	0.017	0.010	0.159	0.130	0.125	0.085	0.045	0.030
NaïveDedup	0.397	0.365	0.497	0.495	0.413	0.405	0.394	0.380	0.094	0.080
C1 (single)	0.019	0.025	0.016	0.018	0.150	0.110	0.131	0.120	0.022	**0.015**
C2 (complete)	0.005	0.005	0.009	0.009	0.150	0.130	0.135	0.065	0.034	0.040
C3 (weighted)	0.002	**0.000**	**0.005**	**0.006**	**0.110**	0.110	0.107	0.070	**0.019**	0.020
C4 (average)	**0.001**	**0.000**	0.007	0.017	0.120	**0.100**	**0.099**	**0.060**	**0.019**	0.020
Mean loss difference		0.016		0.014		0.027		0.035		0.010

Table 3.1: True loss and the loss estimated by our framework.

[a]Algorithm did not finish within a reasonable time limit because of the high computational cost.

		25, 25 samples		100, 100 samples		500, 500 samples	
Clustering	true loss	# queries	estimated loss	# queries	estimated loss	# queries	estimated loss
C1 (single)	**0.06107**	51	0.06	204	0.025	1023	0.024
C2 (complete)	**0.04177**	50	0.02	210	0.005	1024	0.016
C3 (weighted)	**0.03831**	50	0.02	203	0.015	1027	0.016
C4 (average)	**0.03489**	52	0.02	207	0.020	1043	0.013

Table 3.2: Simulated dataset: Impact of number of samples on the loss of the clustering

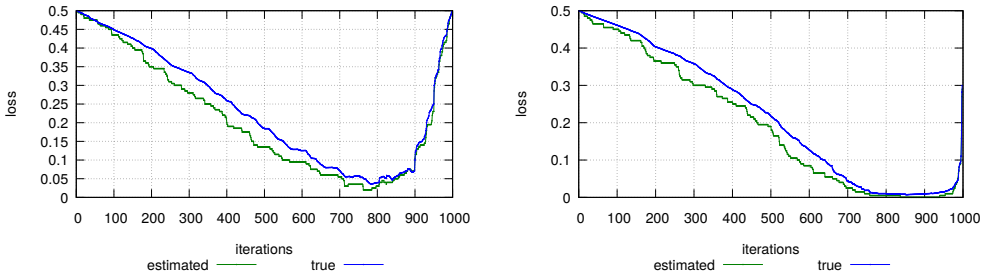
choose amongst many of the classes of algorithms for de-duplication. We also show that our framework can always choose a clustering which is close to the best clustering (algorithm) from a given class of clustering (algorithms) and our estimated loss for each of the clustering is very close to the true loss of these clustering algorithms. In Section 3.4.3 we show that our framework is robust to upto 10% of oracle mistakes, which far exceeds the intended settings dealing with human experts. Finally, in Section 3.4.4 we show that in our framework a relatively small number of samples are enough to accurately estimate the loss of a clustering.

3.4.1 Evaluation setup

Algorithms In our evaluation we use four graph based clustering algorithms: (1) Articulation point clustering (*ArtPt*) [39], (2) Star clustering (*Star*) [24], (3) Approximate correlation clustering (*ApproxCorr*) [26], (4) Markov clustering (*Markov*) [91]. These graph based algorithms have been used for de-duplication problems as shown in previous work [51]. Hierarchical clustering algorithms are very effective and have been widely used to perform de-duplication. We consider 4 different linkage methods for hierarchical clustering: single linkage (C1), complete linkage (C2), weighted linkage (C3), and average linkage (C4). In addition to this we also implemented a heuristic based de-duplication algorithm

		25, 25 samples		100, 100 samples		500, 500 samples	
Clustering	true loss	# queries	estimated loss	# queries	estimated loss	# queries	estimated loss
C1 (single)	**0.11075**	51	0.08	208	0.055	1031	0.041
C2 (complete)	**0.37172**	50	0.34	204	0.315	1035	0.334
C3 (weighted)	**0.29622**	51	0.14	203	0.260	1037	0.239
C4 (average)	**0.26877**	50	0.20	204	0.195	1027	0.202

Table 3.3: Publications dataset: Impact of number of samples on the loss of the clustering

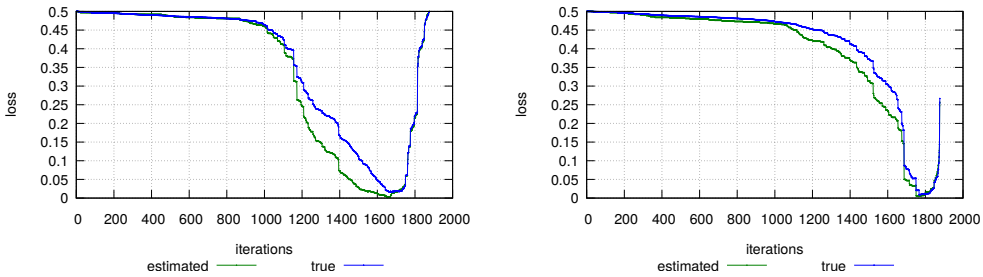


(a) Single linkage (b) Complete linkage

Figure 3.1: Simulated dataset: Loss reported for every iteration of hierarchical clustering

(*NaïveDedup*) where any two data points are considered similar if their distance is below a certain threshold. The output of this algorithm is pairs of data points which are marked similar.

Datasets For our evaluation we use five datasets. First dataset is a simulated dataset of ten thousand strings of length 20 where we simulate a clustering over the set of strings and use it as our ground truth. We use Jaro distance [57] as the distance metric for strings. To simulate a clustering we generate some seed strings and then for each seed string we generate multiple secondary strings by slightly editing the seed string. Each cluster of strings resembles a single entity. Second dataset is a real-world bibliographical information of scientific publications [1]. The dataset has 1879 publication records with duplicates. The ground truth of duplicates is available. To perform clustering on this dataset we first tokenized each publication record and extracted 3-grams from them. Then, on 3-grams we used Jaccard distance to define distance between two records. Next two datasets are lists of E-commerce products: First dataset contains 1,363 products from Amazon, and 3,226 products from Google, and the ground truth has 1,300 matching products. Second dataset contains 1,082 products from Abt, and 1,093 products from Buy, and the ground truth has 1,098 matching products. Both these products datasets are publicly available at [3]. The fifth dataset is a list of 864 restaurants from the Fodor's and Zagat's restaurant guides that contains 112 duplicates. This dataset is also publicly available at [4]. To perform clustering



(a) Single linkage (b) Complete linkage

Figure 3.2: Publications dataset: Loss reported for every iteration of hierarchical clustering

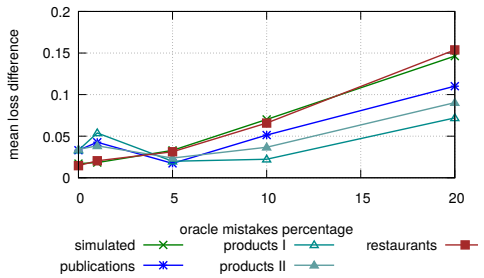


Figure 3.3: Impact of oracle mistakes

on the products and restaurants datasets we normalized the records (product or restaurant description) using standard techniques from natural language processing, namely; denoising text, word tokenization, normalization, and stemming and lemmatization. Given a record, this process gives us a list of word tokens. For each token, we first obtained a vector representation of the word using *Global Vectors* for word representations (GloVe [79]). We averaged this representation across word tokens to obtain the representation of a single record. We use cosine similarity to define the distance between two records. For the simulated and publications datasets, our distance metric was Jaccard and hence we use the MinHash [34] as the hashing scheme. For the rest of the datasets, we used SimHash [35] as the hashing scheme. For all the datasets we use ground truth as the oracle that can answer *same-cluster queries*. To calculate the true loss of a clustering (i.e. $L_{C^*}(C)$) we access all of the ground truth. Our framework uses only a sample of the ground truth to estimate the loss of a clustering. To judge the performance of our framework we compare the estimated loss $\hat{L}_{C^*}(C)$ against the true loss.

3.4.2 Clustering selection

In this experiment we demonstrate that our framework is generic and can be used to choose the best clustering algorithm amongst any of the classes of algorithms for de-duplication. We used our framework on all the algorithms mentioned in Section 3.4.1. The results on five datasets are summarized in Table 3.1. For each dataset we report the loss of the true-best clustering ($L_{C^*}(C)$) and the estimated loss of the best clustering selected by our framework $\hat{L}_{C^*}(C)$. This experiment highlights two main features of our framework: (i) our framework can always choose a clustering close to the best clustering algorithm from a given class of clustering algorithms using only a small number of samples, which is 200 (100 positive samples, and 100 negative samples) for all datasets and all the algorithms in Table 3.1. (ii) Our estimated loss for each clustering is very close to the true loss of these clustering algorithms. At the bottom of the table we report the mean loss difference between estimated loss and true loss computed over all the algorithms.

We would also like to emphasize that in our framework we sample only once for each dataset and use that sample to estimate the loss of all the clusterings. In Figure 3.1 and 3.2 we show that our sample can very closely estimate the loss of every clustering generated at each iteration of the hierarchical clustering. Similarly, for Table 3.1 we sampled only once for each dataset and evaluated all the clusterings generated by each algorithm. Note that, each of the graph based algorithms have a hyper-parameter, i.e. the threshold on the edge weights. Edges with weights above this threshold represent dissimilar items and are pruned from the graph. For each of the graph based clustering algorithm we applied our framework on multiple values of the hyper-parameter and report only the ones with least true loss. However, for every choice of the hyper-parameter we observed that the estimated loss was very close to the true loss.

3.4.3 Effect of oracle mistakes

In this experiment we show that our framework is effective in real-world scenarios where the oracle may not be perfect and can make mistakes. Whenever the oracle classifies a similar pair as dissimilar or a dissimilar pair as similar we count it as a mistake. In our datasets we artificially introduce such mistakes and vary their ratio from 0%, to 20%. In Figure 3.3, we show that our framework can closely estimate the clustering loss up to 10% of oracle mistakes, which, in real-world far exceeds the intended settings dealing with human experts. The Y-axis in Figure 3.3 reports the mean difference between true loss and estimated loss over all the clusterings selected in Table 3.1.

3.4.4 Impact of sample size

As mentioned in Section 3.3.1, the number of labelled positive and negative samples required by the framework are $O(\frac{\text{VC-Dim}(\mathcal{F})}{\epsilon^2})$, which is a small constant, independent of the size of the data. In this experiment we show that a small number of samples are enough to estimate the true loss ($L_{C^*}(C)$). We consider four different clusterings, each one picked at random from the four hierarchical clustering methods (C1 - C4). Table 3.2 and 3.3 reports the loss for simulated and publications dataset, respectively. For each dataset we increased the number of positive and negative samples and measured the loss. The table also shows the true loss of the clustering. It can be seen that the estimated loss calculated by our framework is close to the true loss even with 25 positive samples and 25 negative samples. In addition to this, the loss does not change much by increasing the number of samples. Which means that there is no incentive to sample more. We also show that the number of queries performed by our framework are close to the sample size, which are orders of magnitude less than $O(|X|^2)$. For example, in the simulated dataset and single linkage clustering (C1) with 25 positive and 25 negative samples our framework performed 51 queries, that means only one query was wasted. Similarly, 4 queries were wasted for 100 positive and 100 negative samples, and so on.

3.5 Related Work

The problem of evaluating clustering algorithms for the de-duplication problem was considered by [51]. They carried out extensive experimental evaluation of different graph-based clustering algorithms on a simulated dataset of strings. They showed that these algorithms perform "extremely well in terms of both accuracy and scalability." Our framework differs from theirs in many crucial ways. To evaluate a clustering algorithm, we do not need access to the complete ground truth clustering. We can find clustering closest to the best clustering even when given access to a small number of oracle answers. Our framework is generic and can work for any class of clustering algorithms, be it graph-based (as was considered by [51]) or hierarchical clustering or any other de-duplication heuristic.

Another class of work related to ours is related to the problem of correlation clustering[26] and its many variants. For example, [43] considered the problem of weighted correlation clustering. In this framework, the edge labels are allowed to be any real number in $[0, 1]$ instead of just zero or one. They showed that this problem is NP-Hard and gave a $O(\log |X|)$ approximation algorithm for the same. Besides correlation clustering, some application oriented works have also modelled de-duplication as a clustering problem. For example, [52]

assumed that the set of duplicate records are transitive. Finding the clustering of the given graph G is now equivalent to computing the connected components of G.

Many application oriented works have also tried to address the problem of data de-duplication. Many works have focused on designing the right metric (or similarity measure) for the given dataset. Once this measure is defined, pairs of points whose distance are below a certain threshold (or whose similarity is above a certain threhhold) are deemed to be duplicates (or belonging to the same cluster). For example, to capture duplicates in the data generated due to typographical errors (like spelling mistakes), edit distance is used [63]. Jaro distance [57] is another measure which tries to capture typographical errors. Phonetic-based similarity measures tries to capture words which are similar sounding. Other measures try to capture the similarity in numerical data. A nice overview of such techniques can be found in [46]. While hand-designing a similarity measure for the given domain is quite popular. Some works also try to learn this function from labelled examples. For example, [38] and [31] use supervised learning techniques (like SVM) to try to learn the weights in the distance function.

Another theme in our work is the notion of human supervision for the clustering task. Many works have tried to introduce the supervision into the clustering problem. For example, [58], [30] and [29] introduced the concept of *link/don't link* constraints. Here, besides the usual input the clustering algorithm also receives a set of pairs of points which should belong to the same cluster and a set of pairs of points which should not belong to the same cluster. The algorithm then finds a clustering which respects these constraints. Continuing this line of work, [23] introduced an interactive version of these constraints called *same-cluster queries*. Here, the clustering algorithm interacts with an oracle by asking whether two points belong to the same or different cluster. The oracle responds by answering 'yes' or 'no' depending on whether the two points belonged to the same clustering according to some ground truth clustering.

Chapter 4

LSM-Tree based storage engine for data cleaning workloads

The need for real-time analytics or Hybrid Transactional-Analytical Processing (HTAP) is ubiquitous in present-day applications such as content recommendation, real-time inventory/pricing, high-frequency trading, blockchain data analytics, and IoT [74]. Many of these applications typically have multiple data sources, such as multiple sensors, multiple inventory warehouses, and so on, which continuously generate a high volume of data that needs to be pre-processed, cleaned, and stored for analytics. Data cleaning in these applications is mostly performed on the newer data, whereas the older data, which is relatively cleaner, is used for analytics and for collecting evidences towards detecting and fixing errors in the new data. For example, column histograms for outlier detection are constructed over the older data. Similarly in data de-duplication, hash functions are applied on columns of old and new data to generate blocks of similar values. We observe that the data cleaning tasks in these applications access newer data in an On-Line Transactional Processing (OLTP) style, i.e., they fetch rows of data to check for errors and possibly update them with the correct values, whereas older data is accessed in an On-Line Analytical Processing (OLAP) style, i.e., they scan columns to build histograms, generate blocks, etc. A typical cleaning workflow for these applications is shown in Figure 4.1 where the cleaning tasks are same as in Figure 1.1 of Chapter 1 but they access new data in OLTP style (shown in bold grey lines) and old data in OLAP style (shown in bold black lines) This access pattern is also in-line with the access pattern of the HTAP applications [74], which fetch new data in OLTP style (e.g., for alerting) and old data in OLAP style (e.g., for weekly or monthly business reports).

Recent systems, such as SAP HANA [47], MemSQL [7], and IBM Wildfire [28], support

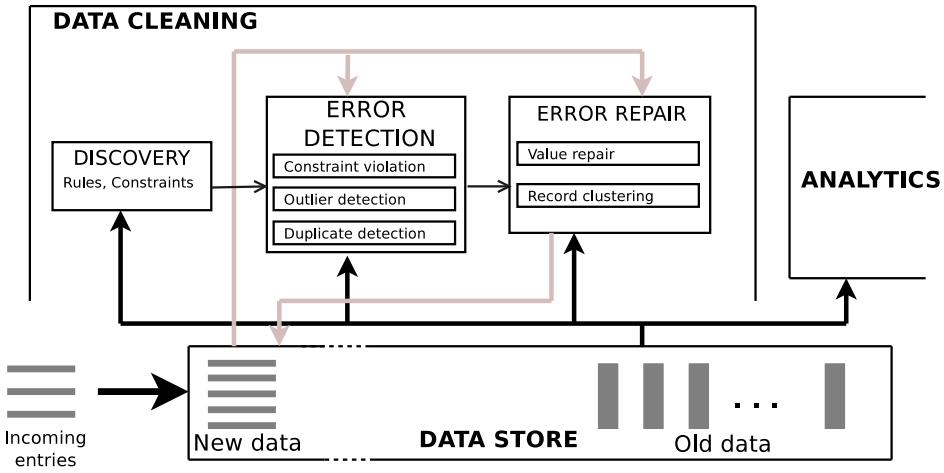


Figure 4.1: Data access pattern of cleaning workflows.

this access pattern by employing hybrid data layouts, in which data is stored in different formats throughout their lifecycle. Recent data is stored in a row-oriented format to serve OLTP workloads and support high data rates, while older data is transformed to a column-oriented format for OLAP access patterns. Such systems can be described as having a *lifecycle-aware* data layout.

We observe that a Log-Structured Merge (LSM) Tree [75] is a natural fit for a lifecycle-aware storage engine. LSM-Trees are widely used in key-value stores (e.g., Google's BigTable and LevelDB, Cassandra, Facebook's RocksDB), RDBMSs (e.g., Facebook's MyRocks [71]), blockchains (e.g., Hyperledger uses LevelDB), and data stream and time-series databases (e.g., InfluxDB). While Cassandra and RocksDB can simulate columnar storage via *column families*, we are not aware of any lifecycle-aware LSM-Trees in which the storage layout can change throughout the lifetime of the data. We fill this gap in our work, by extending the capabilities of LSM-based systems to efficiently serve data cleaning workloads and real-time analytics workloads.

An LSM-Tree is a multi-level data structure with a main-memory buffer and a number of levels of exponentially-increasing size (discussed in detail in Section 4.2). Periodically, or when full, the buffer is flushed to Level-0. When Level-0, which stores multiple flushed buffers, is nearly full, its data are merged into the sorted runs residing in level one (via a *compaction* process), and so on. We observe that LSM-Trees provide a natural framework for a lifecycle-aware storage engine due to the following reasons.
① **LSM-Trees are write optimized:** All writes and data transfers between levels are batched, allowing high write throughput.

② **LSM-Trees naturally propagate data through the levels over time:** At any point in time, the buffer stores the most recent data that have not yet been flushed, and lower levels contain data from subsequent time intervals. For example, if a workload has a steady rate of new inserts, then the buffers may store the entries inserted in the last hour, Level-0 may contain data between one hour and 24 hours old, and levels one and beyond store even older data.
③ **Different levels can store data in different layouts:** Data may be stored in row format in the buffer and in some of the levels, and in column format in other levels. This suggests a flexible and configurable storage engine that can be adapted to the workload.
④ **Compaction can be used to change data layout:** Transforming the data from a row to a column format can be done seamlessly during compaction, when a level is merged into the next level.

We make the following **contributions** in this work.

- We show that the data cleaning tasks in real-time analytics applications have a lifecycle-aware data access pattern. The recent data is accessed in OLTP style, whereas old data is accessed in OLAP style.
- We propose the *Real-Time LSM-Tree*, which extends the design of a traditional LSM-Tree with the ability to store data in a row-oriented or a column-oriented format in each level.
- We characterize the design space of possible Real-Time LSM-Trees, where different designs are suitable for different workloads. To navigate this design space, we provide a cost model to select good designs for a given workload.
- We develop and empirically evaluate LASER, a Lifecycle-Aware Storage Engine for Real-time analytics based on Real-Time LSM-Trees. We implement LASER using RocksDB, which is a popular open-source key-value store based on LSM-Trees. We show LASER can speedup data cleaning workloads, by upto 2x compared to Postgres (a row-store RDBMS) and an order of magnitude compared to MonetDB (a column-store RDBMS).

4.1 Lifecycle-aware access pattern in data cleaning

As shown in Figure 4.1, data cleaning consists of three steps: discovery, error detection, and error repair. Out of these, discovery is an offline task performed occasionally. Whereas,

error detection and error repair are done online as new data enters the system, therefore interact with the data store more frequently than the discovery step. The error detection phase consists of detecting three types of errors: outliers, duplicates, and constraint violations [14]; and the data repair phase involves techniques that update the data with fixes [55]. In this section, we analyze the data access patterns of these individual tasks and show that they access recent data in row-oriented style and old data in column-oriented style.

- **Outlier detection:** This involves detecting column values that deviate from the distribution of the values in a column of a table. Common algorithms to identify the distribution of column values involve building models like histograms, or Gaussian and multivariate Gaussian mixtures [14]. Building these models require column scans over the historical data. However, to detect outliers in the recently inserted entries, we fetch these entries row-at-a-time and compare them against these models. Therefore, for detecting outliers, the new data is accessed in row-oriented style and old data via column scans.

- **Duplicate detection:** As mentioned in Chapter 3, duplicate detection involves selecting pairs of entries that belong to the same real-world entity. A popular technique in duplicate detection is to apply a hash function, such as minHash or SimHash, that puts similar entries in the same block, then test the pairs of entries within a block for duplicates. The hash functions are typically applied only over a small set of columns that can help in grouping similar entries. For example, in an employee table, columns such as *firstname* and *lastname* are good candidates for applying MinHash scheme w.r.t. the Jaccard similarity measure. Therefore, applying a hash function involves scanning a small set of columns. In the next step of duplicate detection, each of the recently inserted entries need to be tested if it is a duplicate of any older entry or not. This requires fetching recent entries one-by-one and fetching the old entries that co-occur with this entry in a block and testing them for duplicates. Therefore, the second step accesses all of the recent data and a small portion of the old entries (which co-occur in a block) in a row-oriented style.

- **Constraint violations:** This refers to values that violate integrity constraints such as Functional Dependencies (FDs) and Denial Constraints (DCs). The data access pattern for detecting constraint violations is similar to outlier detection. Integrity constraints are defined over a set of columns and the entries are considered erroneous if their column values do not conform to a given integrity constraint. To validate the recently inserted data against these constraints we again fetch new data row-at-a-time and scan the columns of the old data to find the entries that match the

constraint condition. For example, consider FD $\{zip \rightarrow city, state\}$, to check if a recently inserted entry conforms with the FD we need to scan old data for columns *zip*, *city*, and *state* and select tuples where *zip* value matches the *zip* value of the recent entry, and check if the *city* and *state* values also match.

- **Error repair:** The workload pattern of repairing errors is simpler than the error detection tasks. Since most of the repairing techniques fix data in-place [55], the workload consists of only update operations. Moreover, we expect most of the repairing to happen in the recent data, since old data is assumed to be cleaner. Therefore, the workload consists of update operations mostly updating the recent data.

We summarize these tasks and their data access pattern in Table 4.1.

Task	**Access pattern**
Outlier detection: detect column values of the recent data that deviate from the distribution of the values in a column	**Row-access:** detect outliers in the recent data one entry-at-a-time. **Columnar-access:** scan columns to generate histograms
Duplicate detection: detect if recently inserted entries are duplicate to some of the older entries. Use hash functions to put similar entries in a block. Then, check entries within a block for duplicates.	**Row-access:** most of the recent entries and any old entries that co-occur with the recent entries in a block. **Columnar-access:** scan a subset of columns which are used by the hash function
Constraint violations: detect any recent entries that do not conform to integrity constraints	**Row-access:** fetch recent entries row-at-a-time to check for constraint violations **Columnar-access:** scan columns, which are part of the constraints, to fetch entries which match the recent data
Error repair: update the entries with correct column values	**Row-access:** most of the updates to the recent data, with few occasional updates to old data.

Table 4.1: Data access pattern of the data cleaning tasks

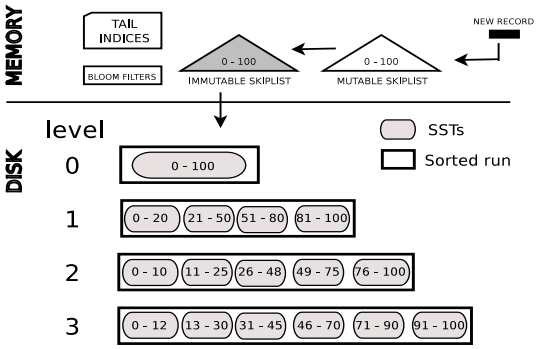


Figure 4.2: LSM-Tree with leveling merge strategy

4.2 Overview of LSM-Trees

We now describe the LSM-Tree, on which the storage engine is based.

4.2.1 Design

Compared to traditional read-optimized data structures such as B-trees or B$^+$-trees, LSM-Trees focus on high write throughput while allowing indexed access to data [68]. LSM-Trees have two components: an in-memory piece that buffers inserts and a secondary-storage (SSD or disk) piece. The in-memory piece consists of trees or skiplists, whereas the disk piece consists of sorted runs.

Figure 4.2 shows the architecture of an LSM-Tree, with the memory piece at the top, followed by multiple levels of sorted runs on disk (four levels, numbered zero to three, are shown in the figure). The memory piece contains two or more skiplists of user-configured size (two are shown in the figure). New records are inserted into the most recent (mutable) skiplist and into a write-ahead-log for durability. Once inserted, a record cannot be modified or deleted directly. Instead, a new version of it must be inserted and marked with a tombstone flag in case of deletions.

Once a skiplist is full, it becomes immutable and can be *flushed* to disk via a sequential write. Flushing is executed by a background thread (or can be called explicitly) and does not block new data from being inserted into the mutable skiplist. During flushing, each skiplist is sorted and serialized to a sorted run. Sorted runs are typically range-partitioned into smaller chunks called Sorted Sequence Tables (SSTs), which consist of fixed-size blocks. In Figure 4.2, we show sorted runs being range-partitioned by key into multiple SSTs. For

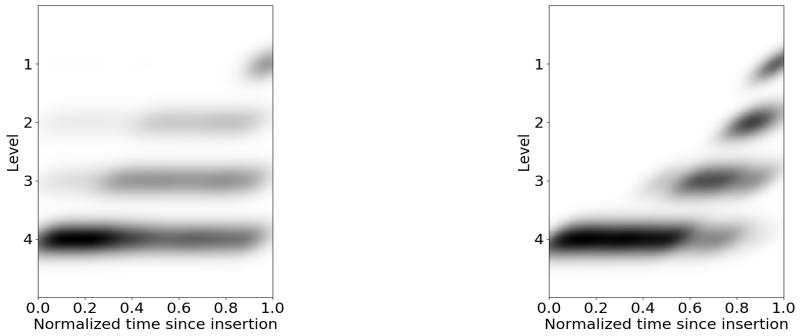
example, the sorted run in Level-1 has four SSTs; the first SST contains values for the keys in the range 0-20, the second in the range 21-50, and so on. Each SST contains a list of data blocks and an index block. A data block stores key-value pairs ordered by key, and an index block stores the key ranges of the data blocks.

As sorted runs accumulate over time, query performance tends to degrade since multiple sorted runs may be accessed to find a record with a given key. To address this, sorted runs are gradually merged by a background process called *compaction*. The merging process organizes the disk piece into L logical levels of increasing sizes with a size ratio of T. For example, a size ratio of two means that every level is twice the size of the previous one. In Figure 4.2, we show four levels with increasing sizes. The parameters L and T are user-configurable in most LSM-Tree implementations, and their value depends on the expected number of entries in the database.

Two common merging strategies are *leveling* and *tiering* [41, 68]. Their trade-offs are well understood: leveling has higher write amplification but it is more read-optimized than tiering. Furthermore, the "wacky continuum" [42] provides tunable read/write performance by adjusting the merging strategy and size ratios. Our Real-Time LSM-Tree is independent of the merging strategy, but we will use the leveling strategy in LASER since this is also used by RocksDB.

In leveling, each level consists of one sorted run, so the run at level i is T times larger than the run at level $i-1$. As a result, the run at level i will be merged up to T times with runs from level $i-1$ until it fills up. If multiple versions of the same key exist, then only the most recent version is kept, and any key with a tombstone flag is deleted. In practice, merging is done at SST granularity, i.e., some SSTs from level $i-1$ are merged with overlapping SSTs in level i. This divides the merging process into smaller tasks, bounding the processing time and allowing parallelism. Sorted runs in Level-0 are not partitioned into SSTs (or have exactly one SST) because they are directly flushed from memory. Some implementations, such as RocksDB, make an exception for Level-0 and allow multiple sorted runs to absorb write bursts.

The merging process moves data from one level to the next over time. This puts recent data in the upper levels and older data in the lower levels, providing a natural framework for a lifecycle-aware storage engine proposed in this chapter. In Figure 4.3, we present the results of an experiment using RocksDB with an LSM-Tree having five levels (zero through four), with Level-0 starting at 64MB and $T = 2$. We inserted data at a steady rate until all the levels were full, with background compaction enabled. We show the distribution of keys in terms of their time-since-insertion for two compaction policies commonly used in RocksDB: *kByCompensatedSize* (Figure 4.3a) prioritizes the



(a) Compaction prioritized by size (*kByCompensatedSize*)

(b) Compaction prioritized by time (*kOldestSmallestSeqFirst*)

Figure 4.3: Distribution of keys across levels based on time

largest SST, and *kOldestSmallestSeqFirst* (Figure 4.3b) prioritizes SSTs whose key range has not been compacted for the longest time. For both compaction priorities, each level has a high density of keys within a certain time range. We will use time-based compaction priority because it is better at distributing keys based on time since insertion. Dong et. al. [44] provide aggregate statistics such as the number of files, amount of data, etc., at each level of an LSM-Tree for three applications. Our observation of time based distribution of data across levels, as reported in Figure 4.3, complements their statistics as we provide a more fine grained trace of how data spreads across levels.

A *point query* starts from the most recent data and stops as soon as the search key is found (there may be older versions of this key deeper in the LSM-Tree, but the query only returns the latest version). First, the in-memory skiplists are probed. If the search key has not been found, then the sorted runs on disk are searched starting from Level-0. Within a sorted run, binary search is used to find the SST whose key range includes the key requested by the query. Then, the index block of this SST is binary-searched to identify the data block that may contain the key. Many LSM-Tree implementations include a bloom filter with each SST, and an SST is searched only if the bloom filter reports that the key may exist. We assume that the ranges of SSTs, the index blocks of SSTs, and bloom filters fit in main memory and are cached, as illustrated in Figure 4.2. For *range queries*, all the skiplists and the sorted runs are scanned to find keys within the desired range. In many implementations (including RocksDB), range queries are implemented using multiple iterators, which are opened in parallel over each sorted run and the skiplists. Then, similar to a *k-way merge*, keys are emitted in sorted order while discarding old versions.

N	number of entries
L	total number of levels
T	size ratio between adjacent levels
B	# of row style entries in a block
B_{ji}	#entries in blocks at CG j at level i
pg	number of blocks in Level-0
c	number of columns
s	range query selectivity (i.e., # entries selected)
Π	set of projected columns
g_i	#column groups at level i, $1 \leq g_i \leq c$
cg_size_{ji}	size of j^{th} CG at level i
$\mathbf{CG_i}$	CGs at level i
E_i^g	estimated number of CGs required by a projection
E_i^G	estimated sum of sizes of CGs required by a projection

Table 4.2: Summary of terms used in this chapter

4.2.2 Cost Analysis

We now summarize the cost of LSM-Trees in terms of writes, point queries, range queries, and space amplification [41, 40, 68]. We assume that leveling is used for compaction, that sorted runs are not partitioned into SSTs, and that the LSM-Tree is in a steady state, with all levels full and the volume of inserts equal to the volume of deletes.

Table 4.2 summarizes the symbols used in the analysis. Let N be the total number of records, T be the size ratio between consecutive levels, and L be the number of levels. Let B denote the number of records in each data page, and let pg denote the number of pages in Level-0. For example, with a 4kB page and each record of size 100 bytes, $B = 40$; with Level-0 of size 64MB, $pg = 16,000$. Level-0 contains at most $B.pg$ entries, and level i ($i \geq 0$) contains at most $T^i.B.pg$ entries. Furthermore, the largest level contains approximately $N.\frac{T-1}{T}$ ($\approx T^L.B.pg$) entries. The total number of levels is given by Equation 4.1.

$$L = \left\lceil \log_T \left(\frac{N}{B.pg} . \frac{T-1}{T} \right) \right\rceil \tag{4.1}$$

Write amplification: Inserted or updated keys are merged multiple times across different levels over time, therefore the insert or update I/O cost is measured in terms of write amplification. The worst-case write amplification corresponds to the I/O required to merge an entry all the way to the last level. An entry in level i is copied and merged every

time level $i-1$ fills up and is merged with level i. This can happen up to T times. Adding this up over L levels, each entry is merged $L.T$ times. Since each disk page contains B entries, the write cost for each entry across all the levels is $O(\frac{T.L}{B})$.

Point queries: The worst-case lookup cost for an existing key is $O(L)$ without bloom filters because the entry may exist in the last level, requiring access to one block (whose range overlaps with the search key) in each level along the way. With bloom filters, the average cost of fetching a block from the first $L-1$ levels is $(L-1).fpr$, plus one I/O to fetch the entry from last level, where fpr is the false positive rate of the bloom filter. In practice, fpr is roughly 1%, giving an I/O cost of $O(1)$.

Range queries: Let s be the selectivity, which is the number of unique entries across all the sorted runs that fall within the target key range. If keys are uniformly spread across the levels, then in each level i, s/T^{L-i} entries will be scanned. With B entries per block, the total number of I/Os is $O(\frac{s}{B}\sum_{i=0}^{L}\frac{1}{T^{L-i}})$. Since the largest level contributes most of the I/O, the cost simplifies to $O(\frac{s}{B})$.

Space amplification: This is defined as $amp = \frac{N}{unq} - 1$, where unq is the number of unique entries (keys). The worst-case space amplification occurs when all the entries in the first $L-1$ levels correspond to updates to the entries in the largest level. The first $L-1$ levels contain $\frac{1}{T}$ of the data. Therefore, $\frac{1}{T}$ of the data in the last level are obsolete, giving a space amplification of $O(\frac{1}{T})$.

4.3 Real-Time LSM-Tree Design

4.3.1 Definitions

Lifecycle-driven hybrid workloads: We target data cleaning workloads and the real-time analytics workloads with high data ingest rates and access patterns that change with the lifecycle of the data. These workloads include a mix of writes and reads, with recent data accessed by OLTP-style queries (point queries, inserts, updates), and older data by OLAP-style queries (range queries) [12]. From a storage engine's viewpoint, we represent these workloads as combinations of inserts, updates, deletes, point reads, and scans. With *key* as the row identifier, *row* as the tuple with all the column values, and Π as the set of projected columns (e.g., $\Pi = \{A, C\}$ means that the query requires values for columns A and C only), we consider the following operations:

- *insert(key, row)*: inserts a new entry.

- *read(key,* Π*)*: for the given *key*, reads the values of columns in Π.
- *scan(*key_{low}*,* key_{high}*,* Π*)*: reads the values of the columns in Π where the key is in the range key_{low}, and key_{high}. Range queries based on non-key column values also use this operator by simply scanning all the entries and filtering out the entries that are not within the range.
- *update(key,* $value_{\Pi}$*)*: updates the values of the columns in Π for the given *key*. $value_{\Pi}$ contains the column identifiers and their new values. For example, $value_{\Pi} = \{(A, nv_a), (B, nv_b)\}$ indicates new values for columns A and B for the given key.
- *delete(key)*: deletes the entry identified by *key*.

We assume that *read* and *update* access recently inserted keys with a wide Π (almost all the columns), while *scan* accesses a range of keys spanning historical and recent data with a narrow Π (one column or a few columns depending on the age of the data).

Column groups (CGs): Hybrid storage layouts support HTAP workloads defined above [74]. A hybrid storage layout is defined by column groups (CGs) that are stored together as rows [27]. Suppose we have a table with four columns: A, B, C, and D. In a row-oriented layout, there is a single CG corresponding to all the columns. In a column-oriented layout, each column corresponds to a separate CG. Other hybrid layouts are possible, e.g., two CGs of $< A, B, C >$ and $< D >$, where the projection over columns A, B, and C is stored in row format, and the projection over D is stored separately. Column groups are advantageous when certain column values are co-accessed often in the workload.

4.3.2 Design Overview

The key insight that makes the Real-Time LSM-Tree a natural fit for a lifecycle-aware storage engine is that *different levels may store data in different layouts*. This creates a design space.

Design space: The design space for Real-Time LSM-Trees can be characterized by the column groups used in each level. In Figure 4.4, we show three examples. On the left, we show an extreme design point corresponding to a row-oriented format, which is used by existing LSM-Tree storage engines, and is suitable for OLTP. On the right, we show the other extreme, corresponding to a pure columnar layout, which is suitable for OLAP. In the middle, we show a hybrid design, in which Level-0 is row-oriented, levels 1 and 2

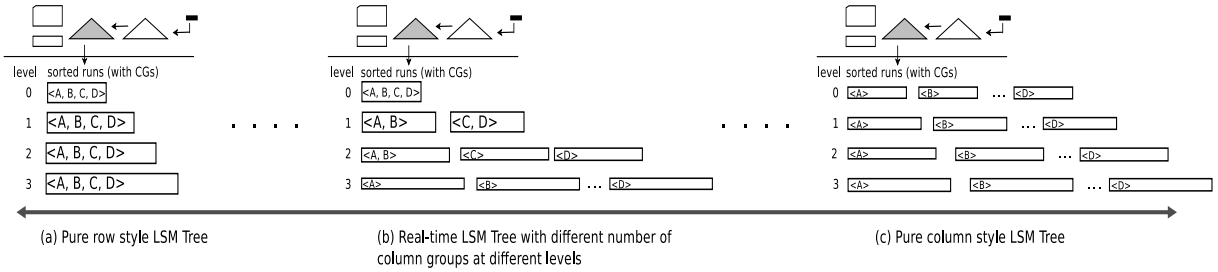


Figure 4.4: Design space of Real-Time LSM-Trees, with example column group (CG) configurations.

use different combinations of CGs, and Level-3 switches to a pure columnar layout. This design may be suitable for mixed or HTAP workloads, with a column group configuration depending on the access patterns during the data lifecycle.

In the Real-Time LSM-Tree design, we keep the in-memory component and Level-0 the same as in the original LSM-Tree, as described in Section 4.2, to maintain high write throughput. However, the on-disk levels beyond Level-0 are split into CGs, where each CG stores its own sorted runs. As we will see in Section 4.4, each such sorted run is associated with tail indices and bloom filters to answer queries that access columns within the CG.

Since different levels may have different CG configurations, a Real-Time LSM-Tree must be able to change the data layout as data move from one level to another. As we will explain in Section 4.4.4, this can naturally be done during the compaction process.

CG containment assumption: In principle, the space of Real-Time LSM-Trees consists of all possible combinations of CGs in each level. However, we make a simplifying assumption since access patterns throughout the data lifecycle tend to change from row-friendly OLTP to column-friendly OLAP. In particular, we assume that any CG in level i must be a subset of (i.e., contained in) a single CG in level $i-1$, for $i \geq 1$. Returning to Figure 4.4, the design in the middle has two column groups in Level-1: $< A, B >$ and $< C, D >$. This means that, for example, a CG of $< A, B, C >$, or a CG of $< B, C >$ is not a valid choice in Level-2. This assumption also simplifies layout changes during compaction, as we will see in Section 4.4.4.

No replication assumption: For some workloads, data in a given level may be touched by both OLTP and OLAP style queries, meaning that no single CG layout is suitable for that level. This may be true especially in the last level, which stores the oldest and the majority of the data. For these workloads, a level can be replicated to maintain two layouts, similar to the "fractured mirrors" approach [80], at the cost of storage and

write amplification. However, we expect such situations to be rare in practice because OLTP patterns tend to be limited to recent data in real-time analytics workloads [74, 12], which are expected to fit in the first few levels.

4.4 LASER Storage Engine

We now describe the design of *LASER* – our HTAP storage engine based on Real-Time LSM-Trees. LASER borrows several concepts from column-store systems [13]: a data model for storing column groups (Section 4.4.1), column updates (Section 4.4.2), and "stitching" individual column values to reconstruct tuples (Section 4.4.3). LASER also requires a mechanism to change the data layout from one level to the next in the Real-Time LSM-Tree (Section 4.4.4).

4.4.1 Column Group Representation

Since entries in an LSM-Tree are stored across multiple sorted runs and levels, column scans do not access the data contiguously. To fetch data in sorted order from different levels, we need to locate entries by their keys. Therefore, we store the keys along with the column group values, as shown in Figure 4.5. This is known as simulated columnar storage [13], and incurs read and storage overhead compared to storing only the column values in a contiguous data block. However, in LSM-Trees, this overhead is reduced due to the leveling merge strategy, and can be further reduced by compressing the data blocks and delta-encoding the keys within each data block. For example, in our evaluation, we observed that naïvely storing keys along with column group values took 86GB of disk space, using Snappy compression took 51GB, and delta-encoding the keys further reduced the space usage to 48GB. Storing the same amount of data in a pure column-store (MonetDB [54]), which stores only the column values, requires 43GB.

4.4.2 Write Operations

Inserts are *performed in the same way as in original LSM-Trees*, where an entry is inserted in the in-memory skiplist, and is eventually moved to lower levels via flush and compaction jobs. Insertion of an existing key (and a corresponding value, containing the values of the remaining attributes) acts as an update, whereas insertion of an existing key with a tombstone flag acts as a deletion.

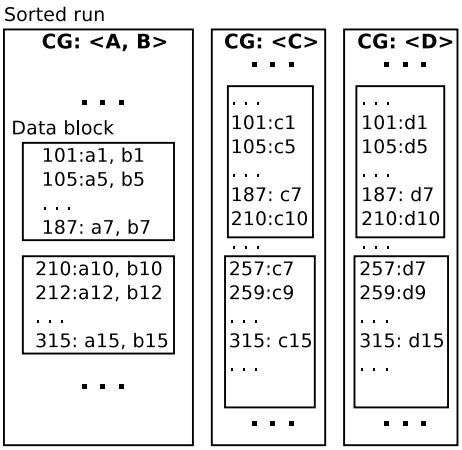


Figure 4.5: Simulated column-group representation

Updates of individual columns may be implemented in two ways. A straightforward way is to fetch the entire tuple that is to be updated, modify the column that is being updated, and re-insert the entire tuple. This is the standard approach in a row-oriented storage engine. Column-oriented storage engines [62, 88] and some HTAP storage engines [22] allow updates of individual columns. Similarly, in LASER, we allow insertion of partial rows that contain only the updated column values. Partial rows are eventually merged with complete rows, or other partial rows, at the time of compaction, and any older column values are discarded. For example, suppose we have four columns, $< A, B, C, D >$, and suppose we update columns B and C of the tuple with key 100. Here, we insert the following key-value pair: $100 : -, b', c', -$ where b', c' are the updated values, and $-$ denotes an unchanged value. If, during compaction, we find another entry for the same key, $100 : a, b, c, d$, then the two entries are merged to give $100 : a, b', c', d$.

4.4.3 Read Operations

Point queries (with projections) are handled by searching for the given key in the skiplist, and then down the levels until the latest value is found. To support projections efficiently, in each level, we only probe the CGs that overlap with the projected columns, and the query result is returned as soon as the values for all of the projected columns are found. Since we allow updates of individual columns, (the latest version of) a given tuple may exist partially in one level and partially in another. For example, in Figure 4.6, the latest values of A and B for tuple 108 exist in Level-0, but the values of C and D exist in Level-2.

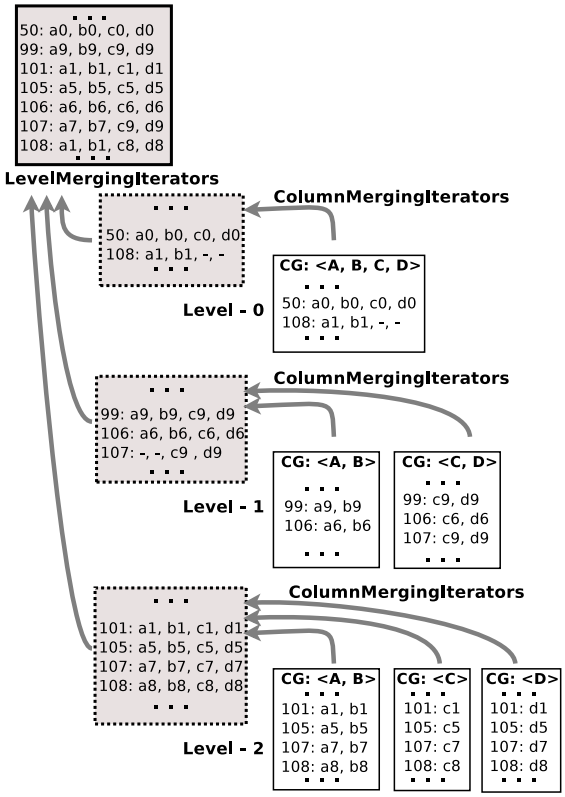


Figure 4.6: *ColumnMergingIterators* and *LevelMergingIterators*

Range queries (with projections) are also handled by opening iterators for each level and returning values in a sorted order, while discarding older versions of the entries. We optimize range queries with projections by opening iterators only for the overlapping column-groups in each level. As was the case for point queries, subsets of column values may be found across different levels. We use *LevelMergingIterators* to merge values across levels, and to stitch column values within a level we use *ColumnMergingIterators*. We provide the details of these iterators in Section 4.4.4.

4.4.4 Real-Time LSM-Tree Compaction

In Section 4.2, we described the compaction process used by LSM-Trees to improve query performance. In LASER, we also utilize compaction to change the data layout. A compaction job selects a level that overflows the most, and merges all of its entries with the next

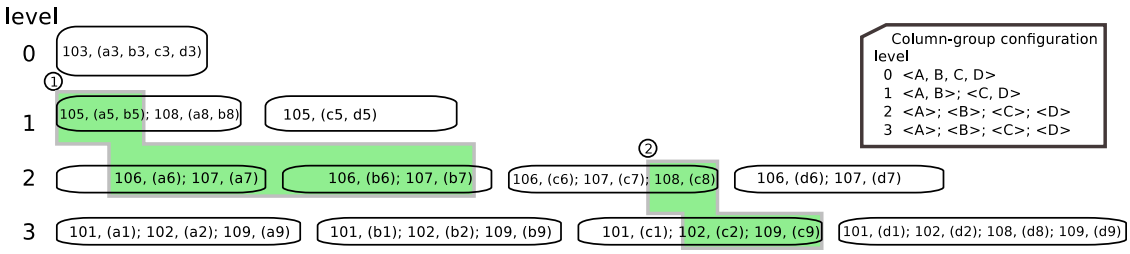


Figure 4.7: Sorted runs of a Real-Time LSM-Tree with two highlighted compaction jobs.

level. Using this approach in LASER would require merging entries from all the CGs of an overflowing level with the next level. However, since we allow individual column updates, as mentioned in Section 4.4.2, different CGs can fill up at different rates. For example, certain column groups (bank balance, inventory) may be updated more frequently than others (contact information, item description). Therefore, treating all the CGs in the same way when scheduling a compaction job might push certain CG values to deeper levels even when top levels are not full, and therefore disrupt the distribution of entries across levels based on time. We modify the compaction strategy to select the most overflowing CG in the most overflowing level. To determine if a CG is overflowing, we define the capacity of a CG within a level by proportionally dividing the level capacity across all the CGs, and any CG that exceeds its capacity is identified as an overflowing CG.

We call this strategy a *CG local compaction* strategy, in which the span of a compaction job is limited to only one CG from level i and the overlapping CGs at level $i+1$. We show two example compaction jobs in Figure 4.7. Compaction job ① merges entries from CG $< A, B >$ in level-1 to overlapping CGs (i.e., $< A >$; $< B >$) in level-2. Similarly, compaction job ② is limited to only CG $< C >$ in level-2 and level-3. To perform *CG local compaction*, we require two types of merging iterators: *LevelMergingIterators* that merge entries from different levels, and *ColumnMergingIterators* that combine column values from different CGs within the same level.

LevelMergingIterators support range queries and compaction jobs by fetching and merging qualifying tuples from each level, and discarding old attribute values when multiple versions are found for the same key. Figure 4.6 shows LevelMergingIterators collecting tuples from three levels to answer a range query for keys between 50 and 108. Only the latest versions of keys 107 and 108 are returned.

ColumnMergingIterators combine values from different column groups within the same level. For each LevelMergingIterator, multiple ColumnMergingIterators are opened. Since there can exist only one version for each key and column value within a level, these

iterators do not have to discard old versions. Instead, they fetch all the required column values for each key, some of which may be empty due to column updates. In Figure 4.6, we show ColumnMergingIterators for each level. In level-0, the iterators return partial values for 108 because the corresponding entry corresponds to an update of columns A and B. Similarly, in level-1, key 107 has a partial value.

The above iterators are used by *CG local compaction* in the following way: we first identify the most overflowing level, and the most overflowing CG at that level. Then, we identify the overlapping CGs in the next level, open LevelMergingIterators for both levels, and open the required ColumnMergingIterators for the respective LevelMergingIterators. Once the iterators are opened, entries are emitted in sorted order and are written to the new sorted run belonging to the next level.

4.5 Cost Analysis of LASER

In this section, we analyze the cost of each operation supported by LASER, and compare it with the cost of a purely row-oriented LSM-Tree (Section 4.2) and a purely column-oriented LSM-Tree (a special case of a Real-Time LSM-Tree with as many CGs as columns). Table 4.3 summarizes the operations and their costs.

We use the variables listed in Table 4.2. Let $1 \leq g_i \leq c$ be the number of CGs at level $0 \leq i \leq L$, where c is the total number of columns. The size of the j^{th} $(1 \leq j \leq g_i)$ CG at level i is defined as the number of columns in the CG and is represented by cg_size_{ji}. cg_size_{ji} is c for all column-groups at all levels for a row-style LSM-Tree and 1 for all column-groups at all levels for a column-style LSM-Tree. For each level i, we have the following relation between c, g_i, and cg_size_{ji}:

$$c = \sum_{j=1}^{g_i} cg\_size_{ji} \tag{4.2}$$

We define B_{ji} to be the number of entries in a data block of a j^{th} CG at level i. From Section 4.2, we know that a row-style LSM-Tree contains B entries in a block. The block size, in bytes, is fixed for an LSM-Tree; for example in RocksDB, it is 4kB by default. If D is the block size in bytes, then we have $D = B.(key\text{-}size + value\text{-}size) = B.(1.dt\_size + c.dt\_size)$, where dt_size is the average datatype size of the columns, which includes the column value and the key. This can be generalized for a Real-Time LSM-Tree, in which a block contains B_{ji} entries: $D = B_{ji}.(1 + cg\_size_{ji}).dt\_size$. For example, in

Figure 4.5, the relationship between the number of entries in a block of CG $< A, B >$, and CG $< C >$ is $D = B_{<A,B>}.(1+2).dt\_size = B_{<C>}.(1+1).dt\_size$, or $B_{<A,B>} = 2.B_{<C>}/3$. The relationship between B and B_{ji} is as follows.

$$B_{ji} = B.\frac{(1+c)}{(1+cg\_size_{ji})} \tag{4.3}$$

This gives $B_{ji} = B.(1+c)/2$ for all column-groups at all levels for a column-style LSM-Tree. As cg_size_{ji} reduces, B_{ji} increases because we can pack more entries of smaller CG size in a block.

Write amplification: We start with the cost of write amplification for *insert(key, row)* operations. For a row-style LSM-Tree, the write amplification is the same as described in Section 4.2, i.e., $O(T.\frac{L}{B})$. For a column-style LSM-Tree, each level has c column-groups (each with one column). Therefore, the write amplification is $O(c.T.\frac{L}{B_{ji}})$, where $B_{ji} = B.(1+c)/2$ for all CGs. For a Real-Time LSM-Tree, the write amplification is summed across all the CGs and all the levels. For example, in level-2 of the Real-Time LSM-Tree shown in Figure 4.7, entries will be merged for CGs $< A, B >; < C >; < D >$ where $B_{02} = B(1+4)/(1+2) = 5B/3$ (i.e., for CG $< A, B >$) and $B_{12} = B_{22} = 5B/2$. For each CG, the merge cost is given by T/B_{ji} (because entries are merged T times, as explained in Section 4.2). The total write amplification cost is: $O(\sum_{i=0}^{L}\sum_{j=1}^{g_i} T/B_{ji})$. Using Equations 4.2 and 4.3, this simplifies to $O(\frac{T.L}{B} + \frac{T}{B.c}\sum_{i=0}^{L} g_i)$. The second term (i.e. $\frac{T}{B.c}\sum_{i=0}^{L} g_i$) represents the overhead of storing keys along with CG values due to the simulated column group representation. This overhead is at most TL/B (because $1 \leq g_i \leq c$) in a column-style LSM-Tree.

$$W := O\left(\frac{T.L}{B} + \frac{T}{B.c}.\sum_{i=0}^{L} g_i\right) \tag{4.4}$$

Point lookups: The cost for a row-style LSM-Tree is the same as in Section 4.2, i.e. $O(1)$ (assuming the false positive rate of bloom filters is much smaller than 1). For a column-style LSM-Tree, the cost is equal to the number of column groups containing the columns projected by the query. For a Real-Time LSM-Tree, this cost is similarly equal to the number column-groups containing the projected columns, summed over all the levels. We use E_i^g $(1 \leq E_i^g \leq g_i)$ to define the number of column-groups required at level i. For example, if there are two CGs, $< A, B >; < C, D >$, in level i, then $E_i^g = 2$ when the projection is $\Pi = \{A, C\}$ and $E_i^g = 1$ when $\Pi = \{A, B\}$. The total I/O cost is bounded

by $O(\sum_{i=0}^{L} E_i^g)$.

$$P := O(\sum_{i=0}^{L} E_i^g) \tag{4.5}$$

Range queries: The I/O cost for a row-style LSM-Tree is the same as in Section 4.2, i.e., $O(\frac{s}{B})$. For a column-style LSM-Tree, this depends on the number of CGs containing the projected columns. Therefore, the I/O cost is $O(|\Pi|.\frac{s}{B.c})$ (here, $B_{ji} = B.(1+c)/2$). For a Real-Time LSM-Tree, different levels contribute different costs depending on the CG configuration. Anytime a CG contains one or more columns projected by the query, the entire block of that CG must be fetched. Therefore, for each level, we have $O(\sum_{j \in G_i} s_i/B_{ji})$, where s_i is the selectivity at level i, and G_i is the set of CGs containing the projected columns. In Section 4.2, we defined s to be the selectivity of a range query for all the levels; selectivity s_i for an individual level i can be estimated by dividing s by the capacity of that level. Using Equation 4.3, we obtain the following cost for each level: $O(\frac{s_i}{c.B} \sum_{j \in G_i} (1 + cg\_size_{ji}))$. We define $E_i^G := \sum_{j \in G_i} (1 + cg\_size_{ji})$, i.e., the sum of the sizes of all the required CGs and corresponding keys. For example, if there are CGs $< A, B >; < C, D >$ in level i, then $E_i^G = 6$ when the projected columns are $\Pi = \{A, C\}$ and $E_i^G = 3$ when $\Pi = \{A, B\}$. The overall cost of a range query is:

$$Q := O(\sum_{i=0}^{L} s_i.E_i^G/c.B) \tag{4.6}$$

Update amplification: The update amplification for a row-style LSM-Tree is the same as the insert amplification: $O(\frac{L.T}{B})$. For a column-style LSM-Tree, the cost depends on the number of column values that are updated due to our *CG local compaction* strategy (Section 4.4.4). The amplification is given by $O(\frac{L.T.|\Pi|}{B.c})$, where Π is the set of updated columns. For a Real-Time LSM-Tree, update amplification depends on the sum of the sizes of the required CGs. This is estimated by E_i^G (see range query cost above). Therefore, the amplification of an update operation is

$$U := O(\sum_{i=0}^{L} T.E_i^G/c.B) \tag{4.7}$$

Space amplification: As explained in Section 4.2, similar to a row-oriented LSM-Tree, the worst-case space amplification in a Real-Time LSM-Tree happens when the first

$L-1$ levels contain updates of entries in the last level. The fraction of entries in the first $L-1$ levels is still $\frac{1}{T}$. Therefore, the space amplification is still $O(\frac{1}{T})$.

Operation	Row-style LSM-Tree	Real-Time LSM-Tree	Column-style LSM-Tree
Insert amplification (W)	$O(\frac{T.L}{B})$	$O(\frac{T.L}{B} + \frac{T.\sum_{i=0}^{L} g_i}{B.c})$	$O(\frac{T.L}{B})$
Existing key lookup (P)	$O(1)$	$O(\sum_{i=0}^{L} E_i^g)$	$O(\lvert\Pi\rvert)$
Range query (Q)	$O(\frac{s}{B})$	$O(\sum_{i=0}^{L} \frac{s_i.E_i^G}{c.B})$	$O(\frac{\lvert\Pi\rvert.s}{c.B})$
Update amplification (U)	$O(\frac{T.L}{B})$	$O(\sum_{i=0}^{L} \frac{T.E_i^G}{c.B})$	$O(\frac{T.L.\lvert\Pi\rvert}{c.B})$

Table 4.3: Summary of operations and their costs.

4.6 Design Selection

In this section, we describe how to select a suitable Real-Time LSM-Tree design for a given workload using the cost analysis from Section 4.5. Our goal is to find an optimal CG configuration for each level to minimize the total I/O cost for a given workload. Finding an optimal CG layout for a given workload has been studied in the context of hybrid database systems. Solutions span from heuristics [82] to complex modeling techniques that allow ranking the candidate CGs [16, 50].

In the context of LSM-trees, the problem is critically different due to the flexibility of assigning a different layout for each level of the tree. That is, we are not searching for a single CG layout across the whole data and tree, but rather we are searching for an optimal layout for each level of the tree in a way that holistically optimizes the overall performance. A critical invariant is the CG containment constraint described in Section 4.3.2, (a CG at level i must be a subset of some CG at level $i-1$). LASER makes a decision per level regardless of whether the LSM-tree is based on leveling (one run per level), tiering, or lazy leveling (where there may be several runs per level). This is because in the latter case, runs overlap in terms of the range of values stored, and so all runs will see the same access patterns given a workload at this level.

In the remainder of this section, we define the optimization problem in the context of Real-Time LSM-trees, and we describe our search algorithm, which is inspired by Hyrise [50] and brings novel design elements to allow different layouts in every level of the tree and to ensure CG containment.

4.6.1 Input

Parameters: To find an optimal CG layout for a given workload, LASER requires: 1) parameters defining the Real-Time LSM-Tree structure, and 2) parameters defining the workload. As explained in Section 4.5, the costs of the operations depend on the Real-Time LSM-Tree structure, which is defined by the parameters T, L, and B (Section 4.2), and on the CG configuration **CG**. We represent the workload by **wl**, which is a set of operations. Let w be the number of *insert* operations, p be the number of *read* operations for existing keys, q be the number of *scan* operations, and u be the number of *update* operations in **wl**. Since we are searching for an optimal CG layout for each level independently, we additionally define level i's workload by $\mathbf{wl_i}$, and similarly, p_i, q_i, and u_i represent the number of *read*, *scan*, and *update* operations, respectively, served at level i.

Obtaining parameter values: We assume that the values of the LSM-Tree parameters (T, L, B) are fixed based on the data size (N) and the Operating System configuration (e.g., page size). Past research has shown how to tune T and L in an LSM-tree [42, 40, 41]. Furthermore, B is usually fixed based on a 4kB block size (as in RocksDB). Overall, these parameter choices are orthogonal to LASER: they govern the high-level LSM-tree architecture while LASER optimizes the architecture within each run. As for the workload, we assume that, at the logical level, it consists of SQL statements. For the LASER storage engine, we convert the workload to the operations defined in Section 4.3.1. Profiling the workload $\mathbf{wl_i}$ at each level allows us to determine the values for w, p_i, q_i, u_i, and s_i. The workload profile can be collected either from the current instance of the storage engine, or from a secondary copy of the LSM-Tree data. As long as the secondary copy and the primary copy have the same distribution of the data with respect to time, the profile will look similar. The layout of the secondary LSM-Tree could either be the same layout of the primary LSM-Tree, or if this is the very first profile, the secondary copy can have all the levels with pure row layout. Finally, the values for E_i^g and E_i^G are determined by the workload trace and the CG configuration under consideration, as discussed in Section 4.5.

4.6.2 Optimization Problem

Cost function: Let W_k be the cost of the k^{th} write operation in the workload, obtained using Equation 4.4; we define P_k, Q_k and U_k similarly based on Equations 4.5 through 4.7. Following previous work on LSM-Tree design [41], we compute the cost of a workload for a given CG configuration **CG** by adding up the costs of each operation, as shown in Equation 4.8.

$$cost(\mathbf{CG}) = \sum_{k=1}^{w} W_k + \sum_{k=1}^{p} P_k + \sum_{k=1}^{q} Q_k + \sum_{k=1}^{u} U_k \tag{4.8}$$

Since we need to find an optimal CG configuration at each level using per-level workload statistics, the cost function in Equation 4.8 can be split into per-level cost, given by the following equation:

$$cost(\mathbf{CG_i}) := \tag{4.9}$$
$$\frac{w.T.g_i}{B.c} + \sum_{k=1}^{p_i} E^g_{ik} + \sum_{k=1}^{q_i} \frac{s_{ik}.E^G_{ik}}{c.B} + \sum_{k=1}^{u_i} \frac{T.E^G_{ik}}{c.B}$$

Here, $\mathbf{CG_i} = \{cg_{i1}, cg_{i2}, ..., cg_{ig}\}$ is the partitioning of columns into g groups at level i that satisfies the CG containment constraint.

Optimization problem: For each level i, we want to find an optimal $\mathbf{CG_i}$ such that $cost(\mathbf{CG_i})$ is minimized for the workload $\mathbf{wl_i}$ and the CG containment constraint is satisfied. This leads to the following optimization problem:

$$\forall i : 1 \leq i \leq L \tag{4.10}$$
$$\mathbf{CG_i^*} = \underset{\mathbf{CG_i}}{\operatorname{argmin}}\ cost(\mathbf{CG_i})$$
$$s.t. :$$
$$\forall cg_{ij} \in \mathbf{CG_i}\ \exists\ cg_{(i-1)k} \in \mathbf{CG_{(i-1)}} \mid cg_{ij} \subseteq cg_{(i-1)k}$$

Recall that we keep level-0 row-oriented, so the CG containment constraint is trivially satisfied for level-1.

4.6.3 Solution

Previous work [50] takes the following three-step approach: 1) pruning the space of candidate CGs, 2) merging candidate CGs to avoid overfitting, and 3) selecting an optimal CG

layout from the candidate CGs. The *CG containment* constraint can be added to the first step, further pruning the space of candidate CGs.

Let $\{a_1, a_2, ..., a_c\}$ be the attributes in relation $\mathbf{R}$, and let Π_j be the projection of the j^{th} operation (point lookup, range query, or update operation) at level i. In the first step, we generate a CG partitioning with the smallest subsets, where every subset contains columns that are co-accessed by at least one operation. This is done by recursively splitting the attributes of $\mathbf{R}$ using the projections Π_j. For example, suppose $\mathbf{R} = \{a_1, a_2, a_3, a_4\}$, and let $\Pi_1 = \{a_2, a_3, a_4\}$, $\Pi_2 = \{a_1, a_2\}$, and $\Pi_3 = \{a_1, a_2, a_3, a_4\}$. Then, splitting using Π_1 gives subsets: $\{a_1\}, \{a_2, a_3, a_4\}$, and further splitting using Π_2 gives subsets: $\{a_1\}, \{a_2\}, \{a_3, a_4\}$ (Π_3 does not split any subsets).

The next step is to merge the subsets from the previous step. This is beneficial for point queries, which typically have wider projections, while smaller subsets are beneficial for range scan operations, which typically have narrow projections. This tension between the access patterns of point queries and scan operations is used to decide which subsets should be merged. We merge smaller subsets only if the cost of running the workload with the larger subsets is lower. To systematically evaluate all merging possibilities, we start with the smallest subsets from the previous step, and consider all possible permutations of them for merging.

Finally, in the third step, we generate all possible CG partitions (covering all attributes of $\mathbf{R}$) from the subsets generated in the previous step, and output the least-cost solution (the cost is given by Equation 4.9).

To satisfy the CG containment constraint, when considering level i, we change the initial set of attributes $\mathbf{R}$ to be the set of attributes from one CG at level $i-1$, and we separately execute our solution for each CG at level $i-1$. For example, if level-2 has CGs: $< a_1, ..., a_4 >$; $< a_5, ..., a_8 >$, then we solve two CG selection problems for level-3, one with $\mathbf{R} = \{a_1, ..., a_4\}$ and one with $\mathbf{R} = \{a_5, ..., a_8\}$. The design selection algorithm starts with level-1, where the complete schema $\mathbf{R}$ is split into CGs using the three steps described above. Then, this process is repeated for level-2 onwards, where each CG at level $i-1$ is split into optimal CGs for level i. The worst case time complexity of finding an optimal CG configuration at a single level is given by [50], which is exponential in the number of partitions generated in the first step. The overall worst case time complexity for all levels equals the number of levels times the worst case complexity of each level. Since the number of partitions is small in practice [50] and the CGs get smaller from one level to the next, the actual time taken by the design selection algorithm is expected to be small. For example, in our evaluation (Section 4.7), design selection took only 3 seconds for 100 columns and 8 LSM-Tree levels.

4.7 Evaluation of LASER

In the experimental evaluation of LASER, we show that:

- The empirical behaviour of LASER matches the cost model from Section 4.5,
- LASER can outperform pure row-store, pure column-store, and other column-group hybrid designs
- Finally, LASER can speed up data cleaning workloads.

Experimental setup: We deployed LASER on a Linux machine running 64-bit Ubuntu 14.04.3 LTS. The machine has 12 CPUs in total across two NUMA nodes, running at 1.2GHz, with 15MB of L3 cache, 16GB of RAM, and a 4TB rotating Seagate Constellation ES.3 disk.

LASER implementation: We implemented LASER on top of RocksDB 5.14. We added the components described in Section 4.4: simulated CG layout, CG updates, support for projections in queries, *LevelMergingIterators* and *ColumnMergingIterators*, and the CG local compaction strategy. We reused other necessary but orthogonal components provided by RocksDB, such as in-memory skiplists, index blocks for SSTs, bloom filters, snapshots, and concurrency. To collect workload traces for design selection, we modified the RocksDB profiling tools to collect per-level statistics about operations and their projections. We implemented the design selection algorithm as an offline process that takes in the workload trace and the LSM-Tree parameters as input.

LASER configuration: Unless specified otherwise, we use the leveling compaction strategy with *kOldestLargestSeqFirst* compaction priority, with a maximum of 6 compaction threads. We use the RocksDB default values of other parameters such as Level-0 size, SST size, and compression.

Alternative compaction strategies: While we use leveling in our experiments, the results are orthogonal to the compaction strategy: regardless of the strategy, the number of entries in every level remains constant given a fixed size ratio (T). For example, tiering, the write optimized merging strategy, or lazy leveling and the wacky continuum [42], which balance read and write costs, only affect the number of runs within a level but do not affect the number of entries in a level (since runs will simply be smaller with those strategies compared to leveling). In our experiments, we vary the size ratio (T), which affects how entries spread across the levels and the number of levels. This is critical as it affects the number of column-group layouts a Real-Time LSM-tree can hold simultaneously.

4.7.1 Workloads

We evaluate the performance of LASER over two types of workloads: 1) generic HTAP workloads: they are used to analyze the behaviour of LASER empirically and compare its performance against other storage layouts and systems. 2) data cleaning workloads: these workloads mimic various data cleaning tasks (described in Section 4.1) and are used to demonstrate the speedup offered by LASER when compared to other systems.

Generic HTAP workloads: We generate HTAP workloads using the benchmark proposed by previous works [22, 25]. The benchmark consists of the following queries that are common in HTAP workloads: (Q_1) *inserts* new tuples, (Q_2) is a *point query* that selects a specific row, (Q_3) is an *aggregate query* that computes the maximum values of selected attributes over selected tuples, (Q_4) is an *arithmetic query* that sums a subset of attributes over the selected tuples, and (Q_5) is an *update query* that updates a subset of attributes of a specific row. These queries are written in SQL as follows:

Q_1: **INSERT INTO** R **VALUES** $(a_0, a_1, ..., a_c)$
Q_2: **SELECT** $a_1, a_2, ..., a_k$ **FROM** R **WHERE** $a_0 = v$
Q_3: **SELECT** $MAX(a_1), ..., MAX(a_k)$ **FROM** R **WHERE** $a_0 \in [v_s, v_e)$
Q_4: **SELECT** $a_1 + a_2 + ... + a_k$ **FROM** R **WHERE** $a_0 \in [v_s, v_e)$
Q_5: **UPDATE** R **SET** $a_1 = v_1, ..., a_k = v_k$ **WHERE** $a_0 = v$

The parameters k, v, v_s, and v_e, control projectivity, selectivity, overlap between queries, and access patterns throughout the data lifecycle. The benchmark includes two types of tables: a narrow table (with 30 columns) and a wide table (with 100 columns). Each table contains tuples with a 8-byte integer primary key a_0 and a payload of c 4-byte integer columns $(a_1, a_2, ..., a_c)$. Unless otherwise noted, the experiments use the table with 30 columns, with uniformly distributed integer values as keys. In all experiments, we run an initial data load phase, followed by a steady workload phase in which we record measurements such as total workload time and latency.

Data cleaning workloads: These workloads represent the data cleaning tasks mentioned in Table 4.1. We consider data cleaning scenarios for real-time analytics applications where new data is continuously ingested and data cleaning tasks are performed at regular intervals to detect and fix errors in the recently inserted data. We consider a table as described for HTAP workloads above, but with 60 columns. We found that the previous work on error detection [14], which considers real-world datasets, has a maximum of 60 columns in their tables. We use query Q_1 to continuously insert data. Query Q_2 is used to select recently inserted entries, whereas a modified form of query Q_4, labeled Q_{4c}, is used to scan columns.

Q_{4c}: **SELECT** $a_1, a_2, ...a_k$ **FROM** R **WHERE** $a_0 \in [v_s, v_e)$

Query Q_5 is used to repair the errors. Using these sets of queries we represent four types of cleaning workloads:

1. *Outlier* represents the tasks of outlier detection. We consider four columns for this task, and scan them individually to build histograms. Rows of recent entries are fetched to detect if they are outliers or not.

2. *Constraint* represents the task of detecting constraint violations. We consider four Functional Dependency (FD) constraints, each with either 2 or 3 columns. These columns are scanned to check if any of the old entries match with the column values of the new entries, and whether they conform with the FD or not.

3. *Dedup* represents the task of detecting duplicates. We consider four columns for applying the hash function. Then, fetch rows of recent entries and any matching older entries.

4. *Complete* represents a complete data cleaning workload suite, including all the three error detection techniques and update operations for fixing the errors. This workload represents many real-world scenarios where a collection of data cleaning tasks are performed simultaneously [14, 87, 55].

4.7.2 Validation of Cost Model

Goal: We begin by validating the cost of point reads, range scans, and write amplification presented in Section 4.5.

Methodology: For a fixed schema and system environment (i.e., fixed c and B) the cost of these operations depends on the query projection size and the CG configuration. We validate the cost model using the narrow table and $T = 2$, as well as the wide table with $T = 10$. For the narrow table, we consider six Real-Time LSM-Tree designs, in which the CG sizes vary from 1 to 30, covering the extreme pure row and pure column layouts, and other designs in between. For each design, we use $g = 30/cg\_size$ equi-width column groups in each level, and we set cg_size to a value in $\{1, 2, 3, 6, 15, 30\}$. In each design, the LSM-Tree has 8 levels with Level-0 in row-format. For the wide table, we consider 4 Real-Time LSM-Tree designs, with cg_size values in $\{1, 4, 10, 100\}$, and 5 LSM-Tree levels. To generate *read* and *scan* operations, we use Q_2 and Q_3, respectively, and we vary k from

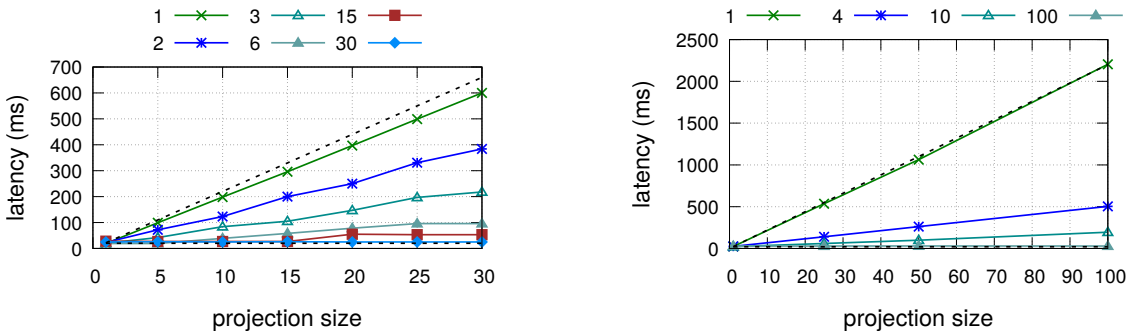
1 to 30 to control the projection size. The queries are executed after the load phase (400 million entries loaded into the narrow table, and 200 million entries into the wide table) with the OS cache cleared, and we measure the average latency. The write amplification cost of the LSM-Tree is reflected in the background compaction process. To measure the compaction time, we first loaded all the entries in Level-0, with compaction disabled, and then scheduled compaction manually and measured its runtime. Compaction is completed when no level exceeds its capacity.

Results: Figures 4.8 and 4.9 show the latency of *read* operations with respect to the projection size and the number of CGs, respectively. The left figures correspond to the narrow table and the right figures correspond to the wide table. In Figure 4.8, when the CGs are small (i.e., similar to a column-oriented layout), latency increases linearly with the projection size because more CGs need to be fetched from disk. However, when the CGs are large (i.e., similar to a row-oriented layout), latency stays unchanged with the projection size because for any projection size, all the columns are fetched. This is also implied by the point query cost given by Equation 4.5, which is plotted in black dotted lines for cg_size=1 (top line) and for cg_size=30/100 (bottom line). The empirical data in Figure 4.8 are in-line with the cost equation.

In Figure 4.9, we vary the number of CGs while keeping the projection sizes fixed. For wide projections (i.e., fetching complete rows), the cost increases linearly with the number of CGs, because each CG is fetched in a separate disk I/O. However, for narrow projections (i.e., fetching a single column value), the I/O cost stays unchanged because a single disk I/O is enough to fetch the required column value. This is consistent with the point query cost given by Equation 4.5, which is plotted in black dotted lines for projection size 30/100 (top line) and 1 (bottom line) in Figure 4.9.

In Figures 4.10 and 4.11, we measure the latency of *scan* operations with respect to the projection size and CG size, respectively. Again, the left figures correspond to the narrow table and the right figures correspond to the wide table. Similar to Figure 4.8, we vary the projection sizes in Figure 4.10. For small CGs (i.e., similar to a column-oriented layout), latency increases linearly with the projection size because more disk I/O is required to fetch more CGs. However, for large CGs (i.e., similar to a row-oriented layout), latency stays almost constant with projection size, because many columns are fetched in a single disk I/O. This is consistent with the range query cost given by Equation 4.6, which is plotted in black dotted lines for CG size 1 (top line) and 30/100 (bottom line) in Figure 4.10.

In Figure 4.11, we vary the CG size while keeping the projection sizes fixed. For wider projection sizes, latency should stay constant with CG size, because almost all the columns



(a) T=2 (b) T=10

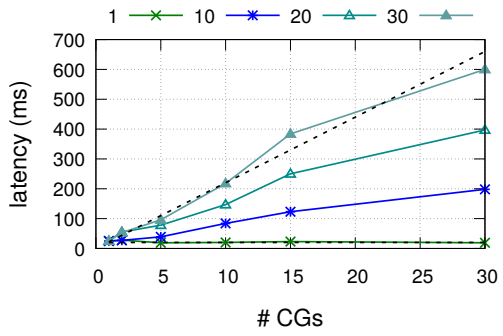
Figure 4.8: *Read* operation: average latency w.r.t. projection size for different CG sizes

need to be fetched irrespective of the CG layout, hence keeping the disk I/O cost constant. However, we see a latency decrease as we increase the CG size; this is because of the simulated CG layout used in LASER. For large CGs, we fetch the key only once, whereas for smaller CGs, the key is fetched along with each CG, hence increasing the latency. The change in latency for wider projection sizes is proportional to $const_1/cg\_size + const_2$ (derived from Equation 4.6), as shown by the top black dotted line in the Figure 4.11. For smaller projections, we expect latency to increase with CG size because of the overhead of fetching unnecessary columns. This is empirically observed in Figure 4.11 and matches the cost given by Equation 4.6. Similar observations were made in prior work on HTAP systems that allow configurable column groups [18, 22].

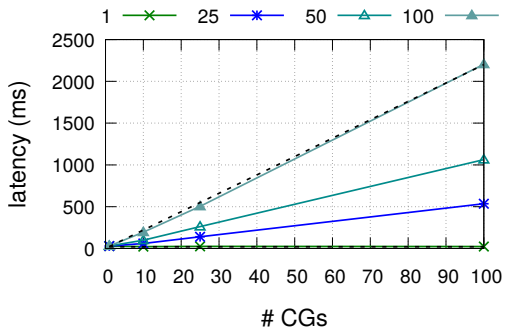
In Figure 4.12, we show that compaction time (which reflects write amplification) for different CG sizes matches our write amplification cost. The cost given by Equation 4.4 is shown using a black dotted line for reference. This completes the empirical validation of the cost model described in Section 4.5, which is an important component of our design selection algorithm in Section 4.6.

4.7.3 Performance of LASER

Goal: We show that LASER can speed up mixed workloads that change with the data lifecycle. We compare the performance of LASER's Real-Time LSM-Tree storage layout against pure row-oriented, pure column-oriented, and certain fixed column-group layouts.

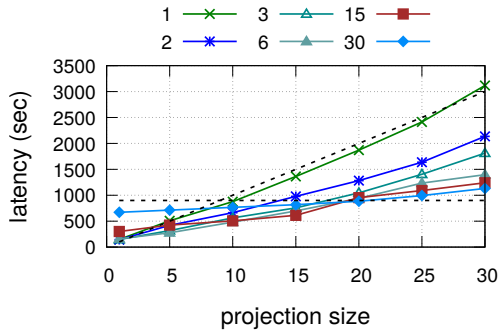


(a) T=2

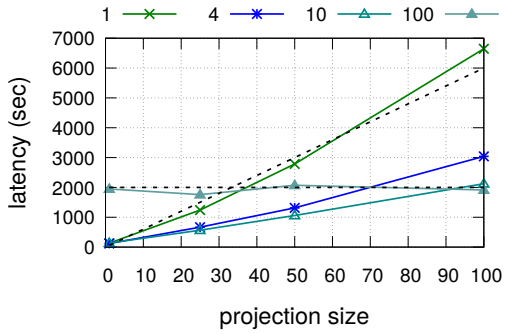


(b) T=10

Figure 4.9: *Read* operation: average latency w.r.t. #CGs for different projection sizes



(a) T=2

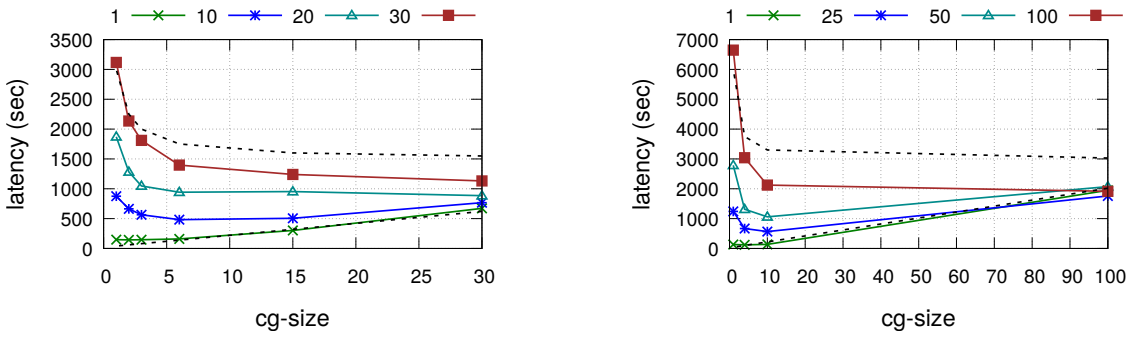


(b) T=10

Figure 4.10: *Scan* operation: average latency w.r.t. projection size for different CG sizes

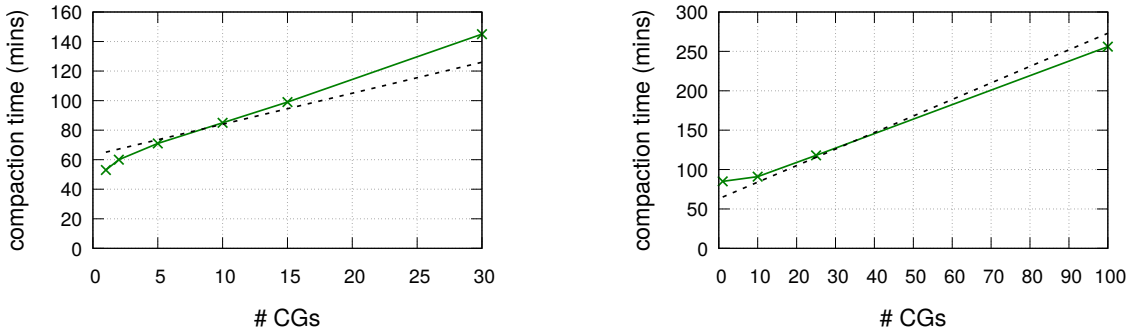
We also compare LASER with a row-store DBMS: Postgres, and a column-store DBMS: MonetDB [54].

Methodology: We generate an HTAP workload (HW) using queries $Q_1 - Q_5$. To emulate a data lifecycle, we continuously insert new data (Q_1) at a steady rate of 10,000 insert operations per second. This ensures that entries continuously move from one level to the next. Along with the inserts, we issue 100 updates per second, i.e., one percent of the insert rate, via Q_5, where a randomly chosen column value is updated for a recently inserted key. This update pattern mimics updates and corrections frequently taking place in mixed analytical and transactional processing [25]. Furthermore, we control the access patterns throughout the data lifecycle by selecting k, v, v_s, and v_e for queries $Q_2 - Q_4$ such that the upper levels of the LSM-Tree are mostly accessed by point read operations



(a) T=2 (b) T=10

Figure 4.11: *Scan* operation: average latency w.r.t. CG sizes for different projection sizes



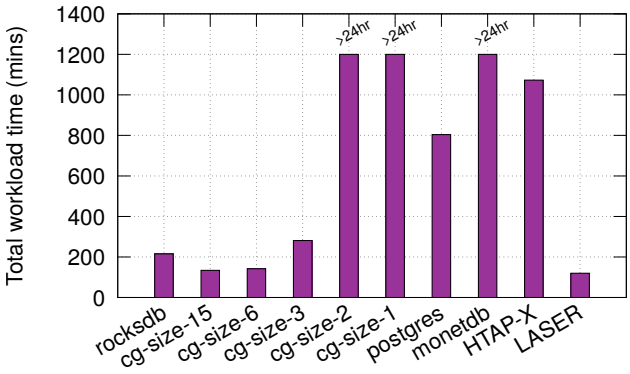
(a) T=2 (b) T=10

Figure 4.12: *Write amplification:* compaction time w.r.t. # CGs

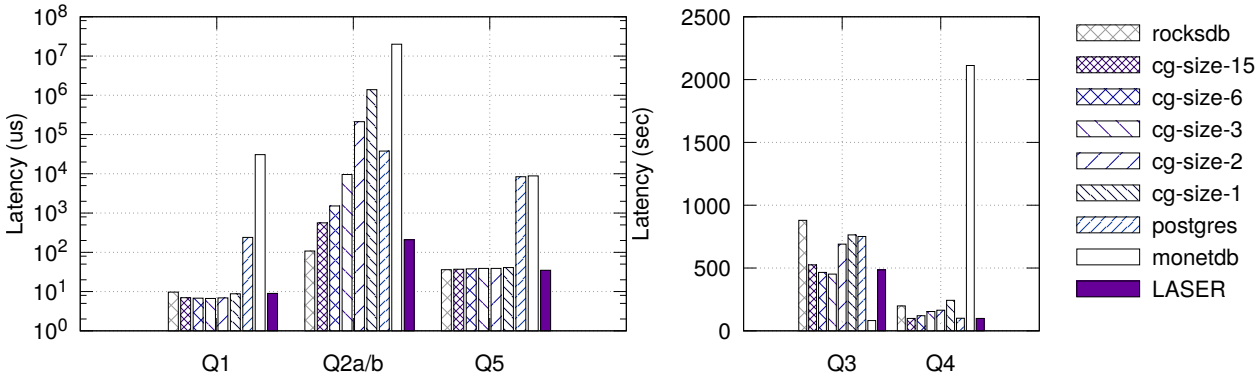
and wider projections, whereas lower levels are accessed by scan operations and narrower projections. This allows us to generate a lifecycle-driven hybrid workload, as described in Section 4.3.1, to stress test LASER's Real-Time LSM-Tree.

We use two variants of Q_2 for point access of recent data: HW-Q_{2a} and HW-Q_{2b}. The v value in each variant is determined by a normal distribution over the time-since-insertion values of the keys. In Figure 4.14a, we show the two distributions from which v is selected. The mean of the first distribution is 0.98 (typically accessing data from in-memory skiplists, Level-0, or Level-1), and 0.85 for the second distribution (typically accessing data from Level-2 or Level-3); each distribution has a standard deviation of 0.02. Q_{2a} queries fetch all 30 attributes, whereas Q_{2b} fetches columns 16-30.

For analytical operations, we use Q_4, which accesses columns 21-30 for 5% of the keys,



(a) Workload runtime of different designs



(b) Average latency of inserts (Q1), point queries (Q2a, Q2b), updates (Q5).

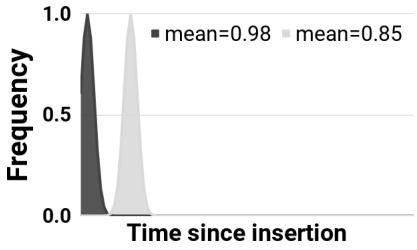
(c) Average latency of range queries (Q3, Q4).

Figure 4.13: LASER performs the best on the HTAP workload (HW).

and Q_3, which accesses columns 28-30 for 50% of the keys. Since our keys are uniformly distributed integer values, these queries access data from all the levels, but the amount of data scanned at level $i+1$ is a factor $T = 2$ more than that scanned at level i. Table 4.4 summarizes the properties of these operations.

We first load 400 million entries, and then execute the workload until another 20 million entries are inserted. Queries HW-Q_{2a} and HW-Q_{2b} are spread uniformly, whereas Q_3 and Q_4 are executed towards the end. Queries $Q_2 - Q_4$ are issued using four concurrent client threads, whereas a separate client thread is responsible for write operations (Q_1 and Q_5).

The CG configuration used by LASER for this workload is labelled *D-opt* (see Figure



(a) Q_{2a}: mean=0.98, Q_{2b}: mean=0.85

$L0$:< 1-30 >
$L1$:< 1-30 >
$L2$:< 1-15 >< 16-30 >
$L3$:< 1-15 >< 16-30 >
$L4$:< 1-15 >< 16-20 >< 21-30 >
$L5$:< 1-15 >< 16-20 >< 21-30 >
$L6$:< 1-15 >< 16-20 >< 21-27 >< 28-30 >
$L7$:< 1-15 >< 16-20 >< 21-27 >< 28-30 >

(b) Design *D-opt* used by LASER

Figure 4.14: *Read* operation patterns and optimal design used in Section 4.7.3

Query	**Projection (k)**	**Key (v) distribution**	**Count**
Q_1	1-30	uniform	10,000/sec
Q_{2a}	1-30	normal,0.98,0.02	500,000
Q_{2b}	16-30	normal, 0.85,0.02	500,000
Q_3	28-30	uniform, 50% of data	12
Q_4	21-30	uniform, 5% of data	12
Q_5	any 1 of 30	uniform, 1% of data	100/sec

Table 4.4: Summary of the HTAP workload HW

4.14b) and is obtained using the approach described in Section 4.6.3. For comparison, we select five other designs with varying CG sizes. The design with cg_size=30 corresponds to a pure row-oriented layout, which is default RocksDB in our setting, and the design with cg_size=1 corresponds to a pure column-oriented layout. The remaining three designs correspond to CG sizes that match the projections of the operations in the workload *HW*. The design with cg_size=15 matches the projection of Q_{2b}, the design with cg_size=3 matches the projection of Q_3, and the design with cg_size=6 is partly suitable for projections of Q_3 and Q_4. To test various CG layouts within reasonable time, we opted to have deeper LSM-Trees, therefore, we set the level size ratio (T) to 2. For all the designs, the LSM-Trees have 8 levels with Level-0 in row-format. To isolate the impact of the storage layout, we simulate these five designs within LASER. Additionally, we executed this workload in Postgres-9.3 (a row-store) and MonetDB 5 server v11.33.3 (a column-store) [54]. Since Postgres and MonetDB are complete RDBMS systems, and LASER is a storage engine with no overhead of SQL parser, and query execution, this comparison is just to give an estimate of the performance difference due to the layouts used by LASER, Postgres, and MonetDB.

Results: Figure 4.13 shows that LASER's optimal design outperforms the other storage layouts when executing the mixed workload described in Table 4.4. Figure 4.13a shows that LASER took the least total time to execute queries Q_{2a}, Q_{2b}, Q_3, and Q_4. Designs with cg_size=1 and 2, and MonetDB did not finish within our time-limit-exceeded (TLE) window of 24 hours. Therefore, we instead report their average latencies in Figure 4.13b and 4.13c. In Figure 4.13b, LASER's design has the latency close to the lowest latency for inserts (Q_1), point queries (Q_{2a}, Q_{2b}), and updates (Q_5). The lowest latency for (Q_{2a}, Q_{2b}), and (Q_5) is observed for *rocksdb*, which is expected, as it is a pure row-store layout. MonetDB has the highest latency, orders of magnitude slower than LASER. Insert and update latencies across all the LSM-Tree designs (including LASER's) are the same because they all append the data to an in-memory skiplist, which is not impacted by the layout of the disk levels. In Figure 4.13c, LASER's latency is close to the latency of the design best suitable for the query. For example, cg_size=3 is suitable for Q_3 and cg_size=15 is suitable for Q_4, and LASER's latency is close to the best latency for both of these queries. For Q_3, MonetDB performs 5x better than LASER because it stores all the data in contiguous columns, making it suitable for aggregation queries. However, MonetDB performs 20x worse than LASER for Q_4. Postgres performs as well as LASER for Q_4 but it performs 1.5x worse on Q_3. Since Postgres is based on B+-trees, which has very contrasting performance characteristics from an LSM-Tree (i.e., less write-optimized, more read-optimized than an LSM-Tree), we also compared LASER against MySQL using MyRocks storage engine [71], which is based on LSM-Tree. We observed that MyRocks performed similar to LASER for Q_2 but performed an order of magnitude worse on queries Q_3 and Q_4; which is attributed to MySQL's row-oriented layout.

Comparison to a simulated HTAP system: Traditional HTAP systems, such as SAP HANA [47], typically consist of a write-optimized component for inserts/updates and point queries against recent data, and a read-optimized component for analytical queries. To demonstrate the benefits of LASER against an ideal traditional HTAP system, we utilize Postgres and MonetDB. That is, we extrapolate the performance of a hypothetical HTAP system that consists of Postgres as a write-optimized component and MonetDB as a read-optimized component. We call this system *HTAP-X*, and we report numbers that represent an ideal scenario, which does not include the cost of moving data from the row-store to the column-store component. This cost can be high if we need to persist columnar data on disk, or relatively low if we are only keeping columnar data temporarily in memory and then discarding the data. Even if this move happens in the "background", i.e., query processing happens on the rest of the data in parallel, those background threads could have been used for query processing, especially by the read-optimized part of the HTAP system that will typically do large scans. In Figure 4.13a, we show results where

the Postgres part of HTAP-X handles Q_1, Q_2, and Q_5, and the MonetDB part handles Q_3 and Q_4. The results show that LASER would perform almost 9x better than this ideal HTAP system. The reason for the improved performance is that LASER leverages the fundamental LSM-architecture, which is able to absorb data much faster than Postgres, while at the same time being able to do analytical queries as fast as MonetDB due to its hybrid layouts.

4.7.4 Speedups for data cleaning workloads

Goal: We show that LASER can speedup the data cleaning workloads described in Section 4.7.1.

Methodology: We use the four data cleaning workloads described in Section 4.7.1. For each of the workloads we have an initial data loading phase where we load 100 million entries into the table. After the data loading phase, we have a continuous stream of inserts at a steady rate of 10,000 inserts per second. We execute the queries of the workloads at every 1 million entries, i.e., at every 1 million new inserts, depending on the workload, we generate new histograms, re-apply hash functions to construct blocks, fetch column values for constraints and so on, and finally, read the recent 1 million entries one at a time to detect errors. In the case of a *complete* workload, we also perform updates (to repair errors) after the error detection completes. We repeat these tasks for 5 iterations, i.e., we stop the workload after inserting 5 million entries. Similar to Section 4.7.3, we compare the performance of LASER against Postgres and MonetDB. LASER's design is selected using the algorithm from Section 4.6.3. We use the LSM-Tree with T= 10 and 5 levels.

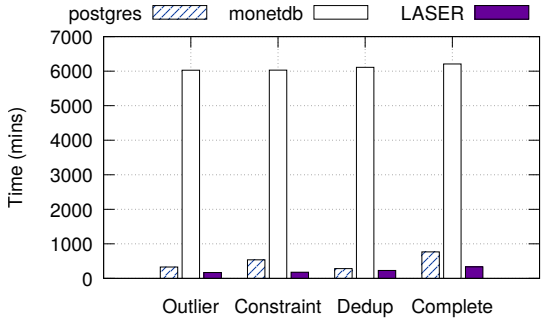


Figure 4.15: LASER performs better than row-store DBMS and column-store DBMS for data cleaning workloads

Results: In Figure 4.15, we show that LASER takes the least time to complete the queries in all the workloads. Similar to Section 4.7.3 results, we observe an order of magnitude speedup when compared to MonetDB (up to 10x speedup), and around 2x speedup when compared to Postgres.

4.8 Other envisioned usage of LASER

In addition to speeding-up data cleaning tasks, LASER storage engine is suitable for any workloads where the access pattern changes from row-oriented access to column-oriented access as the data gets old. Recent HTAP survey [74] indicates that this data access pattern is prevalent in real-time analytics applications such as content recommendation, real-time inventory/pricing, high-frequency trading, blockchain data analytics, and IoT.

LASER in its current form is just an embedded storage engine like RocksDB. Although its API is richer than RocksDB's simple *get* and *put* API (LASER supports column projections), it is not capable of executing SQL queries on its own and therefore requires a SQL parser, and query optimizer on top to handle SQL workloads. Systems such as MyRocks [71] and CockroachDB [2] already use RocksDB as an underneath storage engine, enabling SQL workloads on LSM-Trees, therefore we believe that LASER, which is also built on RocksDB, can be integrated as a storage engine for lifecycle-driven SQL workloads, and we consider this as a future work.

Limitations of LASER: As mentioned in Section 4.3.2, LASER is capable of serving row-oriented workloads and column-oriented workloads simultaneously due to the leveled structure of the LSM-Trees. However, this leveled structure makes read and scan operations slightly expensive because the results have to be merged from different levels. Therefore, for pure row-oriented or OLTP workloads, or pure column-oriented workloads performing large scans, LASER would not be an ideal choice, when compared to a pure row-store or a pure column-store system. We observed this performance gap when we compared the latency of column scan operations on LASER vs MonetDB (a pure column-store system).

4.9 Related Work

The proposed idea of a Real-Time LSM-Tree lies at the intersection of LSM-Trees and HTAP storage engines. In this section, we discuss related work in these domains.

LSM-Trees for key-value stores and NoSQL systems: LSM-Trees are used in key-value stores and NoSQL systems such as LevelDB [6], RocksDB [9], Cassandra [61], HBase [5], and AsterixDB [20]. LevelDB [6] is a simple key-value embedded storage engine open-sourced by Google. LevelDB pioneered the partitioned leveling merge policy used in RocksDB [9] and in our system, LASER. RocksDB [9] was developed by Facebook. It is based on LevelDB with new features added. For example, RocksDB supports additional merging policies, including the *kOldestSmallestSeqFirst* policy we use in LASER. RocksDB also supports the merge filter API that allows users to provide custom logic to garbage-collect obsolete entries. We use this API in LASER to support the merge of column updates. In addition to these embedded storage engines, LSM-Trees have been used in full database management systems: HBase [5] and Cassandra [61] are distributed NoSQL data storage systems, AsterixDB [20] is a Big Data Management System for managing semi-structured (e.g., JSON) data, and MyRocks [71] is one of the largest sharded RDBMS deployed by Facebook to manage the social graph data and Facebook Messenger messages. A recent survey [68] describes how the original LSM-Tree idea [75] has been adopted by various industry and open-source DBMSs.

Other use of LSM-Trees: The log-structured history access method (LHAM) [72] supports temporal workloads by attaching timestamp ranges to each sorted run and pruning irrelevant sorted runs at query time. Furthermore, LSM-trie [92] is an LSM-Tree based hash index for managing a large number of key-value pairs where the metadata, such as index pages, cannot be fully cached. However, LSM-trie only supports point lookups since its optimizations heavily depend on hashing. Finally, the LSM-based tuple compaction framework in AsterixDB [19] leverages LSM lifecycle events (flushing and compaction) to extract and infer schemas for semi-structured data. Similarly, we exploited LSM-Tree properties, such as data propagation through the levels over time and compaction, in LASER.

Improvements of LSM-Trees: Recent works have optimized various components of LSM-Trees such as allocating space for Bloom filters [40] and tuning the compaction strategy [41]. LSM-Trees have been criticized for their write stalls and large performance variance due to the mismatch between in-memory writes and slow background operations. To mitigate this, Luo et. al. [67] proposed a scheduler for concurrent compaction jobs. Later versions of RocksDB [10] also support prefix bloom filters to optimize range queries for certain workloads, such as scans over primary and secondary index ranges. Many of these recent improvements are orthogonal to the design of our Real-Time LSM-Tree. In future work, we will incorporate these improvements in LASER to further improve performance.

HTAP systems and workload-driven storage: One of the first approaches, frac-

tured mirrors, simultaneously maintained one copy of the data in row-major layout for OLTP workloads and another copy in column-major layout for OLAP workloads [80]. This approach has been adopted by Oracle and IBM to support columnar layout as an add-on. Although these systems achieve better ad-hoc OLAP performance than a pure row-store, the cost of synchronizing the two replicas is high. HYRISE [50] automatically partitions tables into column groups based on how columns are co-accessed by queries. Systems such as SAP HANA [47], MemSQL [7], and IBM Wildfire [28] split the storage into OLTP friendly and OLAP friendly components. Data are ingested by the OLTP friendly component, which is write-optimized and uses a row-major layout, and are eventually moved to the OLAP friendly component, which is read-optimized and uses a column-major layout. Peloton [22] generalizes this idea by partitioning the data into multiple components called tiles, with different column group layouts. In this work, we described these systems as having a *lifecycle-aware* data layout, and we showed that LSM-Trees are a natural fit for a lifecycle-aware storage engine. The general idea of workload-driven storage layout has been adopted by several systems and across different domains such as relational data [50, 16] and Resource Description Framework (RDF) data [21]. In the relational data setting the best layout is decided by the optimal horizontal and vertical partitioning of the data for a given workload, whereas in the RDF data the best layout is decided by physically clustering the frequently co-accessed triplets [21].

Chapter 5

Conclusion and Future Work

In this chapter, we conclude the dissertation and discuss some promising future works.

5.1 Conclusion

As maintaining data quality has become an important task for many real-world applications, existing data cleaning solutions find it challenging to scale to large datasets without sacrificing the quality of the output. In this dissertation, we considered scalability as a *first-class citizen* for data cleaning solutions and proposed the following three principles that can help achieve scalability without sacrificing quality: 1) a new primitive-based re-writing of the existing algorithms that allows for efficient implementations for multiple computing frameworks, 2) efficiently involving domain expert's knowledge to reduce computation and improve quality, and 3) using an adaptive data store that can transform the data layout based on the access pattern.

Following these principles, we made three contributions in this dissertation. First, we proposed six primitives that correspond to the data processing steps of existing discovery algorithms for UCCs, FDs, ODs and DCs. These primitives allowed us to analyze the algorithms in terms of their communication and computation costs, and enabled an exploration of the space of possible optimizations. We demonstrated this exploration via case studies and an empirical evaluation. In particular, our experimental results showed that the execution plans which revisit the design decisions made in the original non-distributed algorithms outperform the straightforward distributed plans.

Next, we considered the framework of restricted correlation clustering (RCC) to model the problem of data de-duplication. The goal of RCC is to choose the clustering from a given class of clusterings which minimizes the correlation loss. The clustering algorithm is allowed to interact with a domain expert by asking whether two points belong to the same or different cluster. We showed that our novel hashing based sampling procedure requires only a bounded number of labelled samples from the domain expert to find a clustering which is close to the best clustering in the finite class. Moreover, to attain one labelled sample, it only makes a constant number of queries to the domain expert (with high probability). We complement our theoretical results with an extensive empirical evaluation on a diverse class of clustering algorithms applied on multiple real-world datasets.

Finally, we presented a Log-Structured Merge (LSM) Trees based storage engine that can speed-up data cleaning tasks. The storage engine is built on our novel idea of Real-Time LSM-Tree, in which different levels may be configured to store the data in different formats, ranging from purely row-oriented to purely column-oriented. We presented a design advisor to select an appropriate Real-Time LSM-Tree design given a representative workload, and we implemented a proof-of-concept prototype, called LASER, on top of the RocksDB key-value store. Our experimental results showed that for data cleaning workloads, LASER is almost 2x faster than Postgres (a pure row-store) and an order of magnitude faster than MonetDB (a pure column-store).

5.2 Future work

In this dissertation, we considered only some of the scalability aspects of data cleaning workflows. However, building scalable workflows would require efforts all through the data cleaning stack shown in Figure 1.1. In the future, we plan to investigate some new ideas and improve upon some of our existing contributions. In particular, we consider the following tasks to be promising directions for future work.

First, designing a cost-model driven algorithm that can automatically select the best algorithm and the best physical execution plan for a given dataset and resources. In Chapter 2, we showed how primitives can help in exploring the design space of different implementations of distributed discovery algorithms. This exploration can be performed by an algorithm that can compute the cost of different versions of the distributed discovery algorithms for a given dataset and resources, and then select the version with least cost.

Second, employing machine learning (ML) techniques in a semi-supervised fashion to automate data cleaning tasks. For example, we may consider using ML techniques for

data de-duplication: to learn a classifier, or to learn the similarity function from training samples. However, to reduce the number of training samples required, domain experts must be involved to either provide high quality training data, or to provide certain rules that can speed up the training process.

Third, further improving the LSM-Tree based storage engine by incorporating various recent improvements proposed for LSM-Trees, such as optimal bloom filter space allocation, optimal compaction strategy, and optimal scheduling of compaction jobs.

References

[1] Bibliography of scientific publications. https://www13.hpi.uni-potsdam.de/fileadmin/user_upload/fachgebiete/naumann/projekte/dude/CORA.xml.

[2] CockroachDB. https://www.cockroachlabs.com/docs/stable/.

[3] E-commerce. https://dbs.uni-leipzig.de/en/research/projects/object_matching/fever/benchmark_datasets_for_entity_resolution.

[4] Fodor's and zagat's restaurant guides. http://www.cs.utexas.edu/users/ml/riddle/data.html.

[5] HBase. https://hbase.apache.org/.

[6] LevelDB. https://github.com/google/leveldb.

[7] MemSQL. http://www.memsql.com/.

[8] Metanome. https://github.com/HPI-Information-Systems/metanome-algorithms.

[9] RocksDB. https://github.com/facebook/rocksdb.

[10] RocksDB - prefix seek. https://github.com/facebook/rocksdb/wiki/Prefix-Seek.

[11] TPCH. http://www.tpc.org/tpch/.

[12] Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. https://www.gartner.com/en/documents/2657815, 2014.

[13] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., Hanover, MA, USA, 2013.

[14] Ziawasch Abedjan, Xu Chu, Dong Deng, Raul Castro Fernandez, Ihab F. Ilyas, Mourad Ouzzani, Paolo Papotti, Michael Stonebraker, and Nan Tang. Detecting data errors: Where are we and what needs to be done? *Proc. VLDB Endow.*, 9(12):993–1004, August 2016.

[15] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: a survey. *The VLDB Journal*, 24(4):557–581, Aug 2015.

[16] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 359–370, New York, NY, USA, 2004. Association for Computing Machinery.

[17] Nir Ailon, Anup Bhattacharya, and Ragesh Jaiswal. Approximate correlation clustering using same-cluster queries. In *Latin American Symposium on Theoretical Informatics*, pages 14–27. Springer, 2018.

[18] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2o: A hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1103–1114, New York, NY, USA, 2014. Association for Computing Machinery.

[19] Wail Y. Alkowaileet, Sattam Alsubaiee, and Michael J. Carey. An lsm-based tuple compaction framework for apache asterixdb. *Proc. VLDB Endow.*, 13(9):1388–1400, 2020.

[20] Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Inci Cetindil, Madhusudan Cheelangi, Khurram Faraaz, Eugenia Gabrielova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis J. Tsotras, Rares Vernica, Jian Wen, and Till Westmann. Asterixdb: A scalable, open source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, 2014.

[21] Güneş Aluç, M. Tamer Özsu, and Khuzaima Daudjee. Building self-clustering rdf databases using tunable-lsh. *The VLDB Journal*, 28(2):173–195, April 2019.

[22] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 583–598. ACM, 2016.

[23] Hassan Ashtiani, Shrinu Kushagra, and Shai Ben-David. Clustering with same-cluster queries. In *Advances in neural information processing systems*, pages 3216–3224, 2016.

[24] J.A. Aslam, E. Pelekhov, and D. Rus. *The Star Clustering Algorithm for Information Organization*, pages 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[25] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. Optimal column layout for hybrid workloads. *Proc. VLDB Endow.*, 12(13):2393–2407, September 2019.

[26] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56(1-3):89–113, 2004.

[27] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[28] Ronald Barber, Christian Garcia-Arellano, Ronen Grosman, René Müller, Vijayshankar Raman, Richard Sidle, Matt Spilchen, Adam J. Storm, Yuanyuan Tian, Pinar Tözün, Daniel C. Zilio, Matt Huras, Guy M. Lohman, Chandrasekaran Mohan, Fatma Özcan, and Hamid Pirahesh. Evolving databases for new-gen big data applications. In *CIDR*, 2017.

[29] Sugato Basu, Arindam Banerjee, and Raymond Mooney. Semi-supervised clustering by seeding. In *In Proceedings of 19th International Conference on Machine Learning (ICML-2002*. Citeseer, 2002.

[30] Sugato Basu, Mikhail Bilenko, and Raymond J Mooney. A probabilistic framework for semi-supervised clustering. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 59–68. ACM, 2004.

[31] Mikhail Bilenko, Raymond Mooney, William Cohen, Pradeep Ravikumar, and Stephen Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems*, 18(5):16–23, 2003.

[32] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. Efficient denial constraint discovery with hydra. *Proc. VLDB Endow.*, 11(3):311–323, November 2017.

[33] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[34] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[35] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[36] Xu Chu, Ihab F. Ilyas, and Paraschos Koutris. Distributed data deduplication. *Proc. VLDB Endow.*, 9(11):864–875, July 2016.

[37] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraints. *Proc. VLDB Endow.*, 6(13):1498–1509, August 2013.

[38] Munir Cochinwala, Verghese Kurien, Gail Lalk, and Dennis Shasha. Efficient data reconciliation. *Information Sciences*, 137(1-4):1–15, 2001.

[39] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[40] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 505–520, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In *ACM SIGMOD International Conference on Management of Data*, 2019.

[43] Erik D Demaine, Dotan Emanuel, Amos Fiat, and Nicole Immorlica. Correlation clustering in general weighted graphs. *Theoretical Computer Science*, 361(2-3):172–187, 2006.

[44] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017.

[45] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, Jan 2007.

[46] Ahmed K Elmagarmid, Panagiotis G Ipeirotis, and Vassilios S Verykios. Duplicate record detection: A survey. *IEEE Transactions on knowledge and data engineering*, 19(1):1–16, 2007.

[47] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The sap hana database - an architecture overview. *IEEE Data Eng. Bull.*, 35:28–33, 03 2012.

[48] E. Garnaud, N. Hanusse, S. Maabout, and N. Novelli. Parallel mining of dependencies. In *HPCS*, pages 491–498, 2014.

[49] Lise Getoor and Ashwin Machanavajjhala. Entity resolution: Theory, practice open challenges. *Proc. VLDB Endow.*, 5(12):2018–2019, August 2012.

[50] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. Hyrise: A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, November 2010.

[51] Oktie Hassanzadeh, Fei Chiang, Hyun Chul Lee, and Renée J. Miller. Framework for evaluating clustering algorithms in duplicate detection. *Proc. VLDB Endow.*, 2(1):1282–1293, August 2009.

[52] Mauricio A Hernández and Salvatore J Stolfo. The merge/purge problem for large databases. In *ACM Sigmod Record*, volume 24, pages 127–138. ACM, 1995.

[53] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: an efficient algorithm for discovering functional and approximate dependencies. *Comput. J.*, 42(2):100–111, 1999.

[54] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[55] I. F. Ilyas and X. Chu. *Trends in Cleaning Relational Data: Consistency and Deduplication.* now, 2015.

[56] Ihab F. Ilyas and Xu Chu. *Data Cleaning.* Association for Computing Machinery, New York, NY, USA, 2019.

[57] Matthew A Jaro. *UNIMATCH, a Record Linkage System: Users Manual.* Bureau of the Census, 1980.

[58] Brian Kulis, Sugato Basu, Inderjit Dhillon, and Raymond Mooney. Semi-supervised graph clustering: a kernel approach. *Machine learning*, 74(1):1–22, 2009.

[59] S. Kushagra, H. Saxena, I. F. Ilyas, and S. Ben-David. A semi-supervised framework of clustering selection for de-duplication. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 208–219, April 2019.

[60] Shrinu Kushagra, Shai Ben-David, and Ihab Ilyas. Semi-supervised clustering for deduplication. In *AISTATS*, 2019.

[61] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[62] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801, August 2012.

[63] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[64] Weibang Li, Zhanhuai Li, Qun Chen, Tao Jiang, and Hailong Liu. Discovering functional dependencies in vertically distributed big data. In *WISE 2015*, pages 199–207.

[65] Weibang Li, Zhanhuai Li, Qun Chen, Tao Jiang, and Zhilei Yin. Discovering approximate functional dependencies from distributed big data. In Feifei Li, Kyuseok Shim, Kai Zheng, and Guanfeng Liu, editors, *Web Technologies and Applications*, pages 289–301, Cham, 2016. Springer International Publishing.

[66] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Discover dependencies from data—a review. *IEEE Trans. on Knowl. and Data Eng.*, 24(2):251–264, February 2012.

[67] Chen Luo and Michael J. Carey. On performance stability in lsm-based storage systems. *Proc. VLDB Endow.*, 13(4):449–462, 2019.

[68] Chen Luo and Michael J. Carey. Lsm-based storage techniques: a survey. *VLDB J.*, 29(1):393–418, 2020.

[69] Oded Maimon and Abel Browarnik. Nhecd-nano health and environmental commented database. In *Data mining and knowledge discovery handbook*, pages 1221–1241. Springer, 2009.

[70] S.A.M. Makki and George Havas. Distributed algorithms for depth-first search. *Information Processing Letters*, 60(1):7 – 12, 1996.

[71] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proc. VLDB Endow.*, 13(12):3217–3230, August 2020.

[72] Peter Muth, Patrick Neil, Achim Pick, and Gerhard Weikum. The lham log-structured history data access method. *The VLDB Journal, v.8, 199-221 (2000)*, 8, 02 2000.

[73] Felix Naumann and Melanie Herschel. An introduction to duplicate detection. *Synthesis Lectures on Data Management*, 2(1):1–87, 2010.

[74] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1771–1775, New York, NY, USA, 2017. Association for Computing Machinery.

[75] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[76] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proc. VLDB Endow.*, 8(10):1082–1093, June 2015.

[77] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *SIGMOD*, pages 821–833, 2016.

[78] Thorsten Papenbrock and Felix Naumann. A hybrid approach for efficient unique column combination discovery. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, pages 195–204. Gesellschaft für Informatik, Bonn, 2017.

[79] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *In EMNLP*, 2014.

[80] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 430–441. VLDB Endowment, 2002.

[81] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229 – 234, 1985.

[82] Philipp Rösch, Lars Dannecker, Franz Färber, and Gregor Hackenbroich. A storage advisor for hybrid-store databases. *Proc. VLDB Endow.*, 5(12):1748–1758, August 2012.

[83] Hemant Saxena, Lukasz Golab, Stratos Idreos, and Ihab F. Ilyas. Real-time LSM-trees for HTAP workloads, 2021.

[84] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. Distributed discovery of functional dependencies. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 1590–1593. IEEE, 2019.

[85] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. Distributed implementations of dependency discovery algorithms. *PVLDB*, 12(11):1624–1636, 2019.

[86] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *MSST'10*.

[87] Michael Stonebraker and Ihab F. Ilyas. Data integration: The current status and the way forward. *IEEE Data Eng. Bull.*, 41(2):3–9, 2018.

[88] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the*

31st International Conference on Very Large Data Bases, VLDB '05, page 553–564. VLDB Endowment, 2005.

[89] Nikki Swartz. Gartner warns firms of 'dirty data'. *Information Management*, 41(3):6, 2007.

[90] Jaroslaw Szlichta, Parke Godfrey, Lukasz Golab, Mehdi Kargar, and Divesh Srivastava. Effective and complete discovery of order dependencies via set-based axiomatization. *Proc. VLDB Endow.*, 10(7):721–732, March 2017.

[91] Stijn van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000.

[92] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 71–82, USA, 2015. USENIX Association.

[93] Catharine M. Wyss, Chris Giannella, and Edward L. Robertson. Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In *DaWaK 2001*, pages 101–110.

[94] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *HotCloud'10*.