UNIVERSITY OF CALIFORNIA,
IRVINE

Frameworks for Semantically Meaningful Data Deletion: Inference-Resilient Erasure in Databases

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Vishal Chakraborty

Dissertation Committee:
Professor Sharad Mehrotra, Chair
Professor Felix Naumann
Professor Faisal Nawab

2026

DEDICATION

To Mā & Bābā, who made my dream theirs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

This dissertation studies deletion, but this section is devoted to the opposite problem: deciding what must never be erased. The work in this thesis is mine only in the narrowest sense. It was made possible by the generosity, patience, guidance, collaboration, and friendship of many people who shaped both the research and the person engaged in it. Any list here is necessarily a finite approximation to a much larger set; limited space, not limited gratitude, determines its length.

First, I am deeply grateful to my advisor, *Professor Sharad Mehrotra*. Sharad gave me the freedom to pursue ideas that were often uncertain, the discipline to turn those ideas into research, and the patience to sit through the many stages in between. From 1 a.m. meetings to lively discussions to conversations that began with a simple question and ended with a new research agenda, Sharad taught me how to think like a researcher. Much of this thesis exists because he believed there was something important in the problem of data deletion even before there was clarity about how the problem could be precisely stated.

I am also very grateful to *Professor Faisal Nawab*. Faisal introduced me to Sharad and, in doing so, changed the direction of my academic life. Since then, he has been a constant source of support, advice, and perspective. What I value most is the ease and honesty with which he has guided me: serious when the situation called for it, direct when I needed clarity, and generous with encouragement when the path ahead felt uncertain.

I owe a special debt of gratitude to *Professor Felix Naumann*. My time at the Hasso Plattner Institut, as a guest researcher, was one of the most important periods of my Ph.D. Felix supported my research with great generosity and helped me understand the privilege and responsibility of doing scientific work. I remain amazed by his ability to read text on a screen and detect errors ranging from a missing assumption in a theorem to an extra space between two words. I do not know whether this is training, talent, or magic, but my writing has benefited enormously from it.

I thank *Professor Nalini Venkatasubramanian* for her guidance. Her advice to think carefully about driving use cases has shaped the way I think about systems research. Whenever I was in danger of building an abstraction that only its author could love, her voice reminded me to ask what real problem it served.

I am thankful for the support and guidance I received from *Professor Sarvesh Pandey* during his visit to UCI as a Fulbright scholar.

I was fortunate to have amaizing collaborators, especially *Professors Sujaya Mayya, Primal Pappachan, and Mohammad Sadoghi*. They have been kind mentors as I navigated graduate school, and I am thankful for the many conversations that helped me think more clearly about research, career choices, and the broader academic journey.

I acknowledge the funding from the UC Noyce Initiative's AI, Law, and Society community and the HPI Research Center in Machine Learning and Data Science at UC Irvine. Their

support gave me the rare luxury of being able to focus deeply on research without having to teach beyond the two required quarters. It also made research visits, conference travel, and collaborations possible, enriching my academic career.

My path to this dissertation began long before my time at UCI. I thank *Professors Aldo Antonelli* and *Elaine Landry*, whose undergraduate classes at UC Davis shaped my interests in logic and philosophy. Those interests ultimately grew into the technical and mathematical habits of thought that continue to help me in my daily research. The possibility of doing a Ph.D. was first planted in my mind by Aldo, and I remain grateful for that early encouragement.

I am also deeply thankful to *Professor Phokion Kolaitis*, by whom I had the privilege of being mentored for three years. Phokion taught me the rigor of technical writing, the art of writing proofs, and the importance of patience with oneself. His mentorship continues to influence how I read, write, and think.

I am eternally indebted to my parents, *Susmita Chakraborty* and *Ratan Chakraborty*. The sacrifices they have made cannot be fully expressed in words. They made my dream of earning a Ph.D. their own, and they supported it with love, faith, and strength even when the path was difficult for them to understand. Although they did not do a Ph.D. themselves, that never stopped them from helping me figure things out, from standing by me, and from believing that this endeavor was possible. Everything I have done rests on what they have given me.

I am thankful to my twin sister, *Dr. Riti Chakraborty*. After all, she is the real doctor in the family. Her love, humor, and perspective have been a steady source of comfort, and I am grateful to have had her support throughout this journey.

I have been sustained throughout this journey by the friendship of many wonderful people. I am particularly grateful to *Bubu, Brent, Paul, Mohor, Avinash, Zehui, Rahul, Pratyoy, Jared, Manjima, Libby, and Brett*. They gave me motivation in difficult times, perspective when everything felt too important, and the ability to laugh things off when the only other reasonable option was to open Overleaf again. A Ph.D. can often feel solitary, but because of them, it never felt lonely for too long. I am also thankful to my therapist, *Kimberly Prohaska*, who, with immense patience, helped me navigate the personal and professional challenges I faced during my Ph.D.

Finally, I acknowledge, with sympathy and admiration, every Ph.D. student who has ever stared at a paragraph long enough to forget what it was supposed to mean, and then returned the next morning to make it better. A dissertation records research, but it also quietly records endurance. I have drawn strength from being part of that community.

To all of you: thank you for what you helped me build and for what you helped me preserve.

VITA

Vishal Chakraborty

EDUCATION

Doctor of Philosophy in Computer Science — **2021—2026**
University of California — *Irvine, USA*

Master of Philosophy in Advance Computer Science — **2017—2018**
Cambridge University — *Cambridgeshire, England*

Bachelor of Science & Arts in Computer Science & Philosophy — **2013—2017**
University of California, Davis — *Davis, USA*

RESEARCH EXPERIENCE

Research Intern — **July—August, 2026**
US Air Force Research Laboratory — *Rome, USA*

PhD Research Intern — **January—June, 2026**
Cisco — *San Jose, USA*

Visiting PhD Scholar — **March—May, 2024**
Hasso Plattner Insitut, University of Potsdam — *Potsdam, Germany*

Graduate Research Assistant — **January, 2021—December, 2025**
University of California — *Irvine, USA*

Graduate Research Assistant — **September, 2018—August, 2021**
University of California — *Santa Cruz, USA*

TEACHING EXPERIENCE

Teaching Assistant — **September—December, 2021**
University of California — *Irvine, USA*

Associate Instructor — **April—June, 2020**
University of California — *Santa Cruz, USA*

Teaching Assistant — **January—December, 2019**
University of California — *Santa Cruz, USA*

PUBLICATIONS

Towards Inference-Aware Privacy Guidance for Data Preparation **2026**
Vishal Chakraborty, Felix Naumann
International Workshop on Quality in Databases (QDB 2026), co-located with VLDB 2026

Post-Quantum Cryptographic Analysis of Message Transformations Across the Network Stack **2026**
Ashish Kundu, Vishal Chakraborty, Ramana Rao Kompella
Preprint, arXiv:2604.08480

WorkLoadWeaver: Realistic Synthetic Query Workloads for Adaptive Database Systems **2026**
Keming Li, Nada Lahjouji, Ashwin Colaco, Eric Zhou, Pratyoy Das, GuangXue Zhang, Sahil Makijani, Vishal Chakraborty, Sarvesh Pandey, Sharad Mehrotra
Workshop on Synthetic Data for AI (SynthAI 2026), co-located with ACM SIGMOD 2026

Inference-Aware & Privacy-Preserving Deletions in Databases **2026**
Vishal Chakraborty, Youri Kaminsky, Arnav Abhijit Dhariya, Sharad Mehrotra, Felix Naumann, Sarvesh Pandey
In Proceedings of the *1st ACM SIGMOD Workshop on Secure and Private Data Management (SeQureDB '26)*, co-located with ACM SIGMOD 2026

Meaningful Data Erasure in The Presence of Dependencies **2024**
Vishal Chakraborty, Youri Kaminsky, Sharad Mehrotra, Felix Naumann, Faisal Nawab, Primal Pappachan, Mohammad Sadoghi, Nalini Venkatasubramanian
International Conference on Very Large Data Bases (VLDB 2025)

Data-CASE: Grounding Data Regulations for Compliant Data Processing Systems **2024**
Vishal Chakraborty, Stacy Ann-Elvy, Sharad Mehrotra, Faisal Nawab, Mohammad Sadoghi, Shantanu Sharma, Nalini Venkatasubramanian, Farhan Saeed
International Conference on Extending Database Technology (EDBT 2024)

Hiding Access-pattern is Not Enough! Veil: A Storage and Communication Efficient Volume-Hiding Algorithm **2023**
Shanshan Han, Vishal Chakraborty, Michael T. Goodrich, Sharad Mehrotra, Shantanu Sharma
Proceedings of the ACM on Management of Data, 1(4), Article 265; presented at ACM SIGMOD 2024

Croesus: Multi-Stage Processing and Transactions for Video-Analytics in Edge-Cloud Systems **2022**
Samaa Gazzaz, Vishal Chakraborty, Faisal Nawab
International Conference on Data Engineering (ICDE 2022)

Algorithmic Techniques for Necessary and Possible Winners **2021**
Vishal Chakraborty, Théo Delemazure, Benny Kimelfeld, Phokion G. Kolaitis, Kunal Relia, Julia Stoyanovich
ACM/IMS Transactions on Data Science, 2(3), Article 22

Classifying the Complexity of the Possible Winner Problem on Partial Chains **2021**
Vishal Chakraborty, Phokion G. Kolaitis
International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021)

GreenFLY: Adding Carbon to the Equation in Online Flight Searches **2017**
Angela Sanguinetti, Andrew Kwon, Yitong Li, Vishal Chakraborty, Suhaila Sikand, Otavio Tarelho, Ying Chen, Nina Amenta
Design, User Experience, and Usability (DUXU 2017)

MANUSCRIPT UNDER SUBMISSION

Differential-SEAL: Data Deletion with Bounded Reinference **2026**
Vishal Chakraborty, Youri Kaminsky, Sharad Mehrotra, Felix Naumann, Sarvesh Pandey, Arnav Dhariya
Under review

Don't Be a Pot Stirrer! Authorized Vector Data Retrieval via Access-Aware Indexing **2026**
Shanshan Han, Vishal Chakraborty, Sharad Mehrotra
Under review

SOFTWARE

P2E2-Erasure `https://github.com/HPI-Information-Systems/P2E2-Erasure`
Research artifact for meaningful data erasure in databases; includes source code, datasets, and experiment configuration for the VLDB 2025 paper.

DiffDel `https://github.com/vchakrab/DiffDel`
Research artifact for differential deletion experiments; includes data generation, leakage analysis, optimization scripts, and reproducibility artifacts.

ABSTRACT OF THE DISSERTATION

Frameworks for Semantically Meaningful Data Deletion:
Inference-Resilient Erasure in Databases

By

Vishal Chakraborty

Doctor of Philosophy in Computer Science

University of California, Irvine, 2026

Professor Sharad Mehrotra, Chair

Databases organize, derive, summarize, and expose data through queries, views, logs, caches, and downstream applications. As a result, deleting a value from a database does not necessarily remove what the system makes knowable about that value. For example, an employee's salary may be reconstructed from title and work hours; a user's location may be recovered from posts, and derived statistics. The deleted value of a target cell may remain inferable from other dependent data after the cell has been set to null. Meaningful deletion may therefore require deleting more than just the value of the target cell. The need to define meaningful deletion is reinforced by data regulations, which grant users the right to have their data deleted but do not provide a precise system-level interpretation of deletion. This dissertation studies a strong inference-resilient interpretation of deletion: how can a system prevent re-inference of a deleted value while minimizing additional deletions?

To model inference, the dissertation introduces relational dependency rules (RDRs), an SQL-grounded language for describing data dependencies. Such dependencies are frequently present in semantically rich domains such as medical, social network applications, etc. Within this framework, information regarding a deleted value can leak through two channels: the post-deletion visible state of the database and the deletion pattern itself, namely the observable set of cells deleted in addition to the target cell.

The first notion of deletion developed in the dissertation, *Semantic Erasure Against Leakage* (SEAL), is a deterministic framework for inference-resilient deletion under binary dependencies. A dependency is treated as either supporting inference of the target or not. A deletion mechanism must ensure that the retained database and deletion pattern disallow the deleted value from being re-inferred beyond the reference allowed by the deletion semantics. We formulate a variant of SEAL named Pre-insertion Post-Erasure Equivalence (P2E2). P2E2 requires that, after erasure, the dependencies available for inferring the target be no stronger than those that existed when the target value was inserted. This offers a semantic rollback guarantee while avoiding the excessive cost of cascade-style approaches that delete all dependent data regardless of when the dependency arose.

Semantic dependencies are often discovered from data or learned statistically, and hold approximately. To accommodate this, the dissertation develops Differential-SEAL (Diff-SEAL), a probabilistic framework for inference-resilient deletion under weighted dependencies. Diff-SEAL bounds the adversary's ability to re-infer the deleted value by a tunable threshold. It accounts separately for leakage through the post-deletion visible state and leakage through the deletion pattern, and combines the two channels on the log-odds scale to obtain an end-to-end bound on posterior recovery.

We design mechanisms for P2E2 and Diff-SEAL and evaluate them on real-world datasets. The results show that meaningful deletion need not require the removal of all dependent data. By reasoning about when dependencies arise, how strongly they support inference, and what the deletion pattern reveals, the mechanisms can enforce inference-resilient deletion with reasonable overheads. Finally, the dissertation connects these deletion guarantees to regulatory compliance through Data-CASE, a framework for grounding ambiguous regulatory concepts into precise system actions. The dissertation does not seek, nor presume, to settle the legally authoritative meaning of erasure; instead, it shows how one strong technical interpretation can be formalized, implemented, and related to regulatory requirements. Together, these

contributions frame deletion in databases from an operation on stored representations into a guarantee about what can be inferred after deletion.

Chapter 1

Introduction and Motivation

1.1 The Problem of Deletion in Databases

Deletion appears to be one of the simplest operations in a database system: remove a tuple, erase a value, or make a record inaccessible. This apparent simplicity is misleading. In practice, deletion is not a single operation, but a family of system behaviors spread across logical visibility, physical storage, maintenance processes, derived artifacts, and application semantics. A value may disappear from query results while its old version remains on disk; it may be marked by a tombstone before compaction removes it; it may be excluded from an interface while replicas, logs, caches, aggregates, or correlated attributes continue to carry evidence about it. The target value is no longer directly retrievable, but the system may still retain enough information to reconstruct it. Thus, even before asking how to implement deletion, we must ask what operation the word *deletion* is meant to denote.

Different data systems attach the word *delete* to different layers of the stack. In MVCC engines such as PostgreSQL and MySQL, deletion may first make a tuple or row version invisible to later transactions, while physical reclamation is deferred to mechanisms such

EID	Team	Title	Level	Manager	Location	SalaryBand
e_1	Ads	SWE	L4	m_{10}	NYC	B4
e_2	Ads	SWE	L4	m_{10}	NYC	B4
e_3	Ads	SWE	L5	m_{10}	NYC	B5
e_4	Infra	SRE	L4	m_{12}	SEA	B4
e_5	Ads	SWE	L4	m_{10}	NYC	***B4***

Table 1.1: Employee records for Example 1.1.

as `VACUUM` or purge [98, 100, 88]. In LSM-based systems, deletion is commonly represented by a tombstone that invalidates older entries and is physically removed only later through compaction [71, 110]. Document stores may implement retention-driven deletion through time-to-live policies [81]. Log systems such as Kafka use tombstone records in compacted topics, where a delete marker removes earlier records for a key but is itself retained for some period before cleanup [4]. Embedded engines such as SQLite expose a separate secure-deletion mode that overwrites deleted content while leaving it disabled by default for performance [115]. Thus, deletion may mean query invisibility, deferred garbage collection, tombstone insertion, retention expiration, log compaction, or physical overwriting. These are all reasonable system behaviors, but they answer different questions. They specify how data is hidden, propagated, scheduled, reclaimed, or overwritten; they do not, by themselves, specify what information remains inferable about the deleted data from the post-deletion system.

The gap becomes clear in the following example.

Example 1.1. A company maintains employee records (Table 1.1) with attributes EID, Team, Title, Level, Manager, Location, and SalaryBand.

Suppose that an observer knows the following semantic dependencies. Here, $\rightsquigarrow$ denotes a semantic dependency (rather than constraints such as function dependency, etc.), and the attributes on the left provide evidence about the attribute on the right but need not determine it uniquely. We will formalize this notion later in 4.

- D1: $(\mathsf{Team}, \mathsf{Title}) \rightsquigarrow \mathsf{SalaryBand}$, which expresses that the team and job titles provide evidence about the salary band
- D2: $\mathsf{Level} \rightsquigarrow \mathsf{SalaryBand}$, which captures the fact that levelling ladders constrain compensation bands;
- D3: $\mathsf{Manager} \rightsquigarrow \mathsf{Level}$, which captures the fact that managers often supervise employees within limited level ranges.

Suppose we wish to delete the target cell $c^* = e_5[\mathsf{SalaryBand}]$.

The request in Example 1.1 has several plausible implementations with varying implications. In a relational system, the target value could be removed by updating SalaryBand to `NULL`, or by deleting and reinserting the tuple without the target cell. If the deletion is tuple-level, a `DELETE` removes e_5 from the query-visible state; if foreign-key relationships are declared, referential actions such as `CASCADE` may propagate the deletion to linked tuples in other tables through foreign keys (child tuples) [99]. These actions preserve retrieval semantics and referential integrity, but they do not reason about dependencies D1–D3, which are semantic rather than referential dependencies. From the perspective of the DBMS, the deletion may be complete: the target value is no longer returned by the relevant query. From the perspective of information, however, the situation remains unresolved.

This distinction is all the more important because data regulations have elevated deletion from a storage-management operation to a privacy and compliance obligation [6, 25, 112]. Regulations such as the GDPR [87], CCPA [24], HIPAA [122], LGPD [21], PDPA [113], and PIPEDA [96] impose obligations around erasure, deletion, retention limitation, data minimization, or restricted use of personal information. Yet, these regulations generally do not specify which database action implements those obligations [112, 25]. They motivate the need for deletion, but they do not by themselves settle what deletion should mean inside a database system.

Consider Example 1.1 again. If erasure means logical inaccessibility, setting SalaryBand to `NULL` may suffice. If erasure means byte-level disappearance, the system must reason about physical persistence and reclamation. If erasure means that the deleted salary band cannot be reconstructed from what remains, neither logical deletion nor physical reclamation is enough since retained attributes create several inference paths. When SalaryBand is set to `NULL`, the deleted value may still be inferred through D1 from Team and Title, or through D2 from Level. It may also be inferred transitively through D3 composed with D2: Manager $\rightsquigarrow$ Level $\rightsquigarrow$ SalaryBand. The value is deleted as a stored representation, but not necessarily deleted as a fact that may be re-inferred.

The problem is not only that residual data can support inference but also that the act of deletion may itself reveal information. For instance, suppose D1 is value-dependent and predictive only for salary bands B3 and above. If the deletion mechanism removes Team and Title along with SalaryBand, the deletion footprint itself reveals that the hidden salary band likely lies in the range where D1 applies. Thus, deletion creates two distinct inference questions: what can be inferred from the data left behind and what can be inferred from the deletion pattern itself?

This thesis studies deletion at the level of inferences that are possible from the remaining data and the deletion pattern. Its starting point is that meaningful deletion in databases cannot be defined only by what bytes remain in the database. Nor can it be defined by what values are directly retrievable through the query interface. Deletion must also account for what the post-deletion system allows an observer to know. In the following section, we abstract from specific database mechanisms, turn to the conceptual foundations of deletion, and ask what it should mean for data to be deleted. With the insights gained, Section 1.3 returns to database systems and develops the inference-centric view of deletion adopted in this thesis. Finally, Section 1.5 outlines the organization of the thesis.

1.2 What Should Deletion Mean?

The technical ambiguity around deletion, both in data regulations and in database systems, reflects a deeper conceptual ambiguity: what does it mean for information to be gone? As Viktor Mayer-Schönberger observes in his work on digital forgetting, "Since the beginning of time, for us humans, forgetting has been the norm and remembering the exception" [78]. Digital systems have inverted this default. Storage is cheap, replication is routine, and derived artifacts are common. As a result, forgetting is no longer a passive default but an operation that must be deliberately engineered. To motivate the inference-resilient view developed in this thesis, we briefly turn to philosophical accounts of forgetting and privacy jurisprudence, where courts have confronted how technology can make information knowable even without direct possession of the original artifact.

Deletion as forgetting in philosophy. Forgetting has long been understood as being an active capacity rather than a mere failure of memory. In *On the Genealogy of Morals* (1887), Friedrich Nietzsche describes forgetfulness as "an active ability to suppress, positive in the strongest sense of the word [...] a little peace, a little tabula rasa of consciousness to make room for something new" [85, II.1]. In *Untimely Meditations* (1874), he similarly argues that it is generally impossible to live without forgetting [86]. Although these are not technical claims, they offer useful framing: forgetting is not simply the absence of retention. It is an active operation that changes what remains available.

A second useful distinction is whether forgetting means *loss* or *inaccessibility*. Paul Ricoeur distinguishes *forgetting-as-erasure*, where traces are destroyed and information becomes irrecoverable, from *forgetting-as-reserve*, where traces persist but are not presently accessible [105]. Database systems often implement something closer to the second form. A value disappears from query results, but the information it carried may remain reconstructible through dependencies, derived data, logs, or retained versions. Plato's *Meno*, a classical

philosophical account of recollection, offers a related test: knowledge that can be reconstructed through questioning was, in that account, never truly lost [97]. In databases, the corresponding method of reconstruction is structural. If a deleted value can be reconstructed from retained data, then the system has not eliminated the information content of that value.

Deletion, inference, and privacy jurisprudence. The legal literature predating recent data regulations reaches a compatible conclusion from a different direction. In their foundational article on the right to privacy, Samuel Warren and Louis Brandeis argued for legal protection against technologies that made private life easier to capture, reproduce, and disseminate [124]. Later, dissenting in *Olmstead v. United States*, Justice Brandeis warned that technology could allow the government to reproduce private information "without removing papers from secret drawers." [117] This warning is directly relevant to database deletion: information can be reproduced without possessing the data that was deleted.

The same concern appears in more recent digital privacy cases. In *Riley v. California*, Chief Justice Roberts recognized that digital data on a cell phone is not analogous to the contents of an ordinary physical container: modern phones contain, and can reveal, extensive records of private life [118]. More recently, in *Carpenter v. United States*, the Court extended this reasoning to cell-site location information held by wireless carriers [119]. The significance of those records was *inferential*. Each record, in isolation, is a technical carrier log. But a sequence of such records could reconstruct a person's movements over time and reveal intimate associations, including familial, political, professional, religious, and other private patterns of everyday life.

These philosophical and legal perspectives converge on one claim: *deletion should be evaluated by what remains knowable, not only by what remains stored.* In database systems, this means that deletion should not be defined only by whether the target representation is physically present or whether the target value can be directly retrieved. It should also be defined by

whether the deleted value remains inferable from the system state produced by deletion. Defining semantically meaningful deletion is therefore an epistemic problem: it turns on what an observer can still come to know using what the system still stores. The next section develops this inference-resilient interpretation as the technical view of deletion studied in this thesis.

1.3 An Inference-Resilient View of Deletion

The earlier discussions lead to three levels at which deletion can be understood in a database system. Each level answers a different question about the state produced by a deletion request.

1. **Physical deletion** removes the representation of the data. Examples include overwriting bytes on disk, reclaiming storage through mechanisms such as `VACUUM` in PostgreSQL [100], compaction in LSM-based engines [110], and tombstone removal. This level asks whether the artifact still exists in the physical storage state.

2. **Logical deletion** removes the data from accessibility. The data may still persist physically, but it is no longer reachable through the normal query interface. Examples include executing `DELETE` statements, applying referential actions such as `CASCADE` [99], and relying on MVCC visibility rules [98] to exclude old versions from query results. This level asks whether the data can still be retrieved through the database interface.

3. **Epistemic deletion** removes the knowability of the data. Even when the target data has been removed from storage and from query results, it may still be re-inferred from what remains in the database through semantic dependencies, derived artifacts, or other inference mechanisms. This level asks whether the deleted fact can still be known, which is the focus of this thesis.

Physical and logical deletions concern representations and interfaces: they specify what happens to bytes, versions, tuples, and the query-visible state. Epistemic deletion, on the other hand, concerns the information that remains after those operations have been performed. It asks whether the post-deletion system still gives an observer enough evidence to recover the target value. A DBMS can therefore implement physical and logical deletion correctly, while still failing to delete in the epistemic sense.

Example 1.1 illustrates this distinction. Setting e_5[SalaryBand] to `NULL` changes the query-visible state. Reclaiming the old tuple version later changes the physical state. Neither action, by itself, addresses the semantic dependencies that relate SalaryBand to Team, Title, Level, and Manager. If these retained values still determine, or strongly predict, the deleted salary band, then the target cell has disappeared as a stored value but remains present as an inferable fact.

Adverserial model. Observe that in our example, inference is facilitated through the data dependencies. In considering inference-resilient data deletion, this thesis adopts a single snapshot adversary, i.e., the adversary can view the entire database one state at a time. Furthermore, the set of data dependencies is known to them. Therefore, post-deletion, the adversary observes the database state and attempts to re-infer a deleted value using the declared set of data dependencies. Two leakage channels arise in this setting which we examine in detail in the following.

Figure 1.1 presents the life cycle of data as an inference timeline, adapted from our earlier work [26]. Before insertion, some information about a value may already be inferable from an external context or prior database state. Inserting the value can create new inference paths by interacting with later data and dependencies. Logical deletion removes direct visibility of the value and physical deletion removes the stored representation. Neither step necessarily returns the system to the inferential state that existed before insertion.

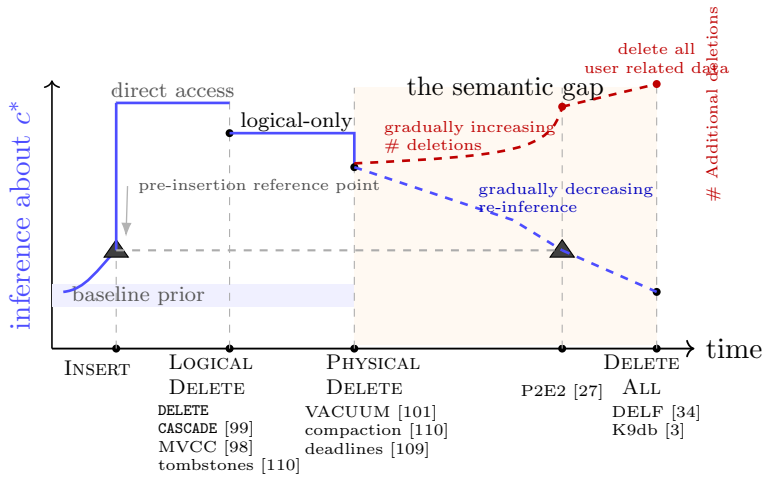


Figure 1.1: Inference timeline and the semantic gap.

The semantic gap. We call the region between storage-level deletion and inference-level deletion the *semantic gap*. Closing this gap requires semantic rollback: removing, masking, perturbing, or otherwise controlling the auxiliary information that supports inference about the target value. Semantic rollback is not the same as pretending the data never existed. An adversary may have had prior knowledge before the data was inserted, and some correlations may exist independently of the database. The relevant question is narrower and more operational: whether storing the value and later deleting it leaves additional inferential support in the system.

Leakage channels. After a deletion mechanism executes, an adversary may re-infer a deleted value though two leakage channels (Figure 1.2).

1. **Residual state (V).** The data that remains visible after deletion, including base tables, views, caches, and externally visible derived artifacts. Dependencies in the residual state may enable re-inference of the deleted value.

2. **Deletion pattern (P).** The set of auxiliary cells that were deleted, or the observable fact that they were deleted. Different inference paths may require different auxiliary deletion sets. An adversary observing which cells were deleted can infer which path

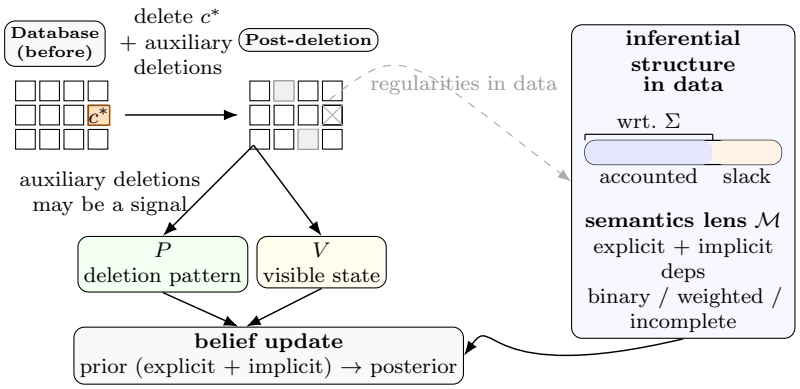


Figure 1.2: Inference inside the semantic gap.

was active, thereby learning about the deleted value.

A deletion mechanism must therefore contend with both channels. Channel V captures what is traditionally understood as inference through dependencies: the retained state itself may still support recovery of the deleted value. Channel P is more subtle: it captures the information leaked by the act of deletion. In our view, a meaningful inference-resilient deletion guarantee must bound the adversary's advantage through both channels jointly.

1.4 Contributions

The preceding sections motivate an inference-resilient view of deletion: deletion should be evaluated not only by what is removed from storage or hidden from queries, but also by what remains inferable from the post-deletion system. This thesis develops the formal and system foundations needed to make this view precise. Its main contributions are as follows.

A three-layer framework to reason about deletion, and a taxonomy of the design space. We observe that a deletion request originates as a statement about a *fact*, must be compiled into operations over query-addressable *data items*, and is finally mapped onto byte ranges in *storage*. This gives three layers, each carrying its own invariant, and the three are separately satisfiable: a value may be unreachable through the query interface while remaining legible

on the medium, and absent from both while remaining inferable from data that was never a representation of it. It follows that a mechanism claiming a privacy guarantee must state three things that are left implicit: the invariant it establishes at each layer, the observer that invariant binds, and the mappings against which it is stated. We use this view to organize the design space of deletion mechanisms along eight mechanism-intrinsic axes and two invocation-context overlays, to place existing deletion mechanisms within it. The unoccupied regions motivate the technical contributions that follow.

An inference framework for semantic dependencies. We introduce relational dependency rules (RDRs), a SQL-grounded language for describing how cells in a database may carry information about other cells. RDRs provide a convenient way to capture inference through traditional data dependencies, such as functional dependencies and integrity constraints, as well as derived values, aggregate relationships, domain rules, and cell-level semantic dependencies. Instantiating RDRs on a database state yields a dependency hypergraph over cells. This hypergraph provides the common structure used throughout the thesis to reason about inference, blocking, and deletion.

SEAL and P2E2. We define *Semantic Erasure Against Leakage* (Seal), or SEAL, as a deterministic framework for inference-resilient deletion under binary dependencies. In SEAL, a dependency is treated as either supporting inference of the target or not, and a deletion mechanism must ensure that the retained database and the deletion pattern do not make the deleted value recoverable beyond the reference allowed by the deletion semantics.

We then develop Pre-insertion Post-Erasure Equivalence (P2E2), a temporal instance of SEAL. A conservative way to satisfy an inference-resilient guarantee is to delete all data that depends on the target value. P2E2 avoids this overly strong approach by observing that some dependencies may already have existed when the target value was inserted. P2E2 therefore requires the post-erasure dependencies available for inferring the target to be no stronger

than those that existed at insertion time. This gives a semantic rollback guarantee while avoiding the cost of cascade-style approaches that delete all dependent data regardless of when the corresponding dependency arose. We give exact and approximate mechanisms for enforcing P2E2 and evaluate their deletion overhead on real-world datasets.

Differential-SEAL. The binary dependency model underlying SEAL is a useful first abstraction, but semantic inference is often not binary. Dependencies may be approximate, discovered from data, learned statistically, supplied by domain knowledge, or weighted by their inferential strength. Treating each such dependency as simply present or absent can be too coarse: weak dependencies may not justify the same deletion burden as deterministic rules, but many weak dependencies may still combine to produce meaningful leakage.

We therefore formulate *Differential Seal*, denoted τ-Seal, as a probabilistic framework for inference-resilient deletion under weighted dependencies. τ-Seal bounds the adversary's ability to re-infer the deleted value by a tunable threshold. It accounts separately for leakage through the residual-state channel V and leakage through the deletion-pattern channel P, and combines the two channels to obtain an end-to-end bound on posterior recovery. We design randomized mechanisms for enforcing this guarantee and evaluate the resulting leakage–deletion tradeoff.

Data-CASE. Finally, we connect these deletion guarantees to the broader problem of regulatory compliance. Data regulations have made deletion a central data-management obligation, but they generally do not specify which system action counts as erasure in a particular database system. Nor does this thesis seek, or presume, to settle the legally authoritative meaning of deletion. Instead, it treats the move from regulation to implementation as a grounding problem: given an ambiguous regulatory concept, identify its possible technical readings, select the reading appropriate for the application and risk setting, formalize it as a system guarantee, and map that guarantee to concrete system actions. We develop Data-

CASE as a framework for carrying out this process for erasure and other compliance-related data-management requirements.

1.5 Thesis Outline

The remainder of this thesis is organized as follows.

Chapter 2: Related Work. We survey deletion mechanisms and adjacent areas, including storage-level deletion, deletion propagation, database repair, inference control, differential privacy, compliance systems, and machine unlearning.

Chapter 3: A Taxonomy of Deletion. We present a three-layered framework for reasoning about deletion from a privacy perspective. Then we organize related work into a taxonomy that clarifies the gap this thesis addresses: while existing systems can remove data from storage, interfaces, or learned models, they generally provide no database-level guarantee that deleted values cannot be inferred from the retained state.

Chapter 4: Background and Preliminaries. We introduce the shared data model used throughout the technical chapters. This includes cell-level deletion targets, relational dependency rules (RDRs), and dependency hypergraphs used for reasoning about inference and erasure under semantic dependencies. These preliminaries provide the common language for Chapters 5 and 6.

Chapter 5: Deterministic Erasure Under Dependencies. We define Pre-insertion Post-Erasure Equivalence (P2E2), a deterministic notion of semantic deletion under declared dependencies. The chapter develops algorithms for demand-driven and retention-driven erasure and establishes the deterministic foundation for making deleted values no more inferable after erasure than they were before insertion.

Chapter 6: Probabilistic Erasure with Weighted Dependencies. We extend the

deterministic setting to weighted dependencies and bounded leakage. The chapter develops mechanisms that trade deletion cost against a user-specified inference-risk threshold, while accounting for both residual-state leakage and deletion-pattern leakage.

Chapter 7: Grounding Erasure. We present Data-CASE, a framework for grounding ambiguous regulatory concepts into concrete system actions, and apply it to erasure. This grounding shows how different interpretations of erasure correspond to different system guarantees, with inference-resilient erasure as the strongest interpretation.

Chapter 8: Conclusion and Future Work. We summarize the contributions of the thesis and discuss open directions.

Chapter 2

Related Work

As argued in Chapter 1, we look at data deletion as an epistemic problem: the question is not only whether data has been removed from the database, but whether the deleted data can be re-inferred. This framing connects deletion to several bodies of work. Storage systems study how data is removed from physical media and the states visible to queries. Database repair, view-update, provenance, and dependency frameworks study how changes propagate through constraints and derived data. Inference control and differential privacy study what an observer can learn from released information. Compliance systems study how regulatory requirements are mapped into operational mechanisms. Machine unlearning studies how the influence of deleted data persists in learned artifacts. Figure 2.1 summarizes how the chapter is organized.

These areas address different facets of the same underlying question: when a deletion request is executed, what has actually been removed and what remains? We organize this chapter around this progression. Section 2.1 surveys deletion as artifact removal. Section 2.2 considers deletion as a consistency and propagation problem under constraints. Section 2.3 connects deletion to inference control and privacy. Section 2.4 discusses deletion as a legal and

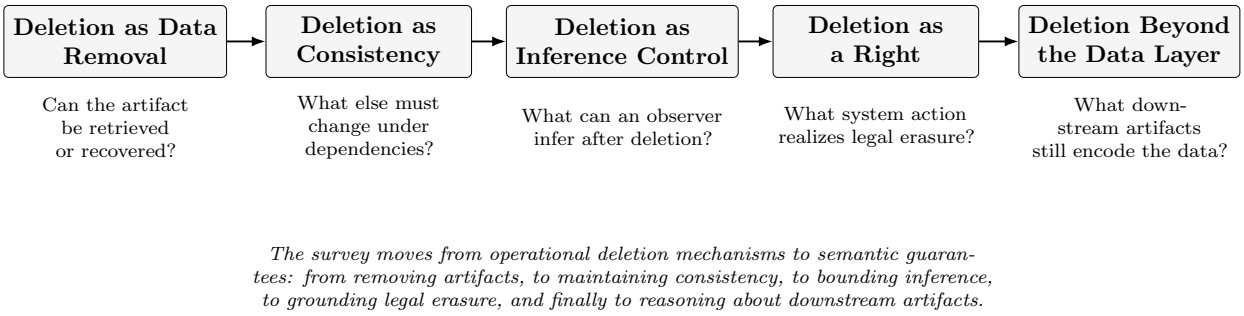


Figure 2.1: Five views of deletion surveyed in this chapter.

compliance obligation. Section 2.5 surveys deletion beyond databases, especially in learned artifacts. We then synthesize the literature into a taxonomy of deletion mechanisms along six axes, identifying the gaps addressed by this thesis (Section **??**). Section 3.9 summarizes these gaps and maps them to the corresponding chapters of the thesis.

2.1 Deletion as Data Removal

The simplest understanding of deletion is artifact removal: making data disappear from storage or from query results. Most database systems implement deletion primarily at this level. They provide mechanisms for removing tuples from the query-visible state, reclaiming physical storage, expiring records, or propagating deletion through application-defined dependencies. These mechanisms are essential, but they answer a limited question: whether the artifact remains accessible or physically present.

Logical deletion in relational systems. In relational databases, the `DELETE` keyword removes tuples from query-visible results, and referential actions such as `CASCADE` propagate the deletion to dependent tuples in child tables [99]. In MVCC engines such as PostgreSQL, deleted tuples are not immediately reclaimed; they remain as dead versions until maintenance processes such as `VACUUM` execute [98, 100]. Application-specific deletion logic can also be expressed through triggers [82]. These mechanisms prevent direct retrieval of the deleted value through the ordinary database interface, but they make no claim about what can be

inferred from the data that remains.

Physical deletion and timeliness. Physical deletion targets byte-level disappearance from storage. In LSM-based engines, deletions are recorded as *tombstones* and become persistent only after compaction. Lethe [110] introduces deadline-aware compaction policies that provide user-facing timeliness guarantees for persistent deletion, achieving this with modest impact on read throughput. Cassandra [71] uses tombstone garbage collection with configurable grace periods. In MVCC/heap engines, `VACUUM` and `VACUUM FULL` reclaim dead tuple storage in PostgreSQL. Sarkar and Athanassoulis [109] observe that current query languages lack the expressiveness to capture deletion intent. They propose SQL extensions, covering both schema definition and data manipulation operations, for expressing retention-driven and on-demand persistent deletion requirements. Recent work [6, 109] argues that timely deletion should be treated as a first-class system guarantee rather than a best-effort maintenance byproduct. These systems strengthen the operational meaning of deletion, but even perfect byte reclamation does not answer whether the deleted value can still be inferred from what remains.

Secure deletion. Reardon, Basin, and Capkun [102] provide a systematization of knowledge (SoK) on secure data deletion, formalizing adversary models such as snapshot versus multi-snapshot and local versus remote adversaries, and cataloging storage-level defenses that include overwriting, encryption-based sanitization, and media-level techniques. Their taxonomy remains a central reference for storage-level deletion threats. Crypto-shredding, where data is encrypted and deletion is achieved by destroying the key, has been explored for relational databases with policy-based key life-cycle management [111]. These approaches address the physical persistence of data on storage media, but they do not reason about the inferential content that may remain in other, non-deleted data.

Application-layer propagation. Several systems propagate deletion across multiple stores and derived artifacts at the application layer. DELF [34] safeguards deletion correctness in social networks by maintaining dependency graphs across multiple backing stores and supporting temporarily reversible deletions. K9db [3] achieves compliance by construction through schema-level ownership annotations and automatic deletion procedures, organizing storage around per-data-subject micro-databases. Edna [121] uses cryptographic disguises with user-controlled "reveal" operations, supporting a reversible deletion window. MongoDB's TTL [81] provides native time-to-live indices for automatic document expiry. Deletion in blockchain contexts has been explored in [69]. These systems recognize that deletion often must propagate beyond one record or one store. However, their guarantees remain operational: they ensure that designated artifacts are hidden, expired, disguised, or removed, rather than formally bounding what information remains inferable after deletion.

History-independent data structures. A complementary line of work studies whether the memory representation of a data structure reveals information about past operations, including deletions. Naor and Teague [83] introduced *history-independent* data structures whose memory layout is distributed identically regardless of the sequence of insertions and deletions that produced the current state. Bender et al. [12] recently achieved operation-order privacy for B-tree-style partitioning and LSM-trees. These approaches address *deletion-pattern privacy* at the storage-representation level. They complement, but do not replace, the database-level inference guarantees pursued in this thesis.

The work surveyed in this section treats deletion primarily as artifact removal. It addresses the question *does the artifact exist?* in the case of physical deletion, or *can the data be retrieved?* in the case of logical deletion. These are necessary questions, but they are not sufficient for semantically meaningful erasure. The question central to this thesis is different: *can the deleted value be known from what remains?*

2.2 Deletion as Consistency

A more sophisticated understanding of deletion emerges once constraints and dependencies are taken into account. Deleting a tuple may violate referential integrity; deleting a source tuple may invalidate a view; and deleting a premise may leave a derived fact unsupported. Work in this area asks what else must change when data is deleted. The objective of the question, however, is usually consistency or propagation rather than inference control.

View-update and deletion propagation. The classical view-update problem [23] asks how changes to a view should be translated back to the base tables. Deletion propagation [63] studies a related but different problem: given a deletion request on the output, find a minimal set of source tuples whose removal achieves the desired effect. Makhija and Gatterbauer [76] recently unified known deletion-propagation variants into a single ILP-based framework that is coarse-grained instance-optimal, meaning that it runs in polynomial time for known tractable cases and gives an optimal formulation for hard cases. Their earlier work [75] provides a unified ILP formulation for resilience and causal responsibility, handling conjunctive queries under set and bag semantics. Freire et al. [43] establish a complexity dichotomy for resilience of binary conjunctive queries with self-joins, identifying structural properties such as chains, confluences, and permutations that characterize NP-hardness. Hu and Sintos [54] show that the head-domination property from deletion propagation precisely captures approximation hardness for the smallest witness problem. These works provide complexity-theoretic foundations for understanding when minimal deletion is tractable. Their objective, however, is to preserve, remove, or explain query answers, not to prevent inference of deleted values from the retained database state.

Database repair and consistent query answering. When a database violates its constraints, the repair problem seeks minimal modifications that restore consistency. The foundational work of Arenas, Bertossi, and Chomicki [5] introduced Consistent Query Answering (CQA) and the

notion of database repair as constructing a consistent database that differs minimally from the original. Subsequent work has studied minimal repair under denial constraints [29, 73, 116]; the relationship between repairs, counterfactual causality, and attribution scores [14]; and practical systems for cleaning constraint violations. Giannakopoulou et al. [46] propose Daisy, a system that repairs denial-constraint violations on demand by relaxing query results and weaving cleaning operators into query plans. Ni et al. [84] provide a comprehensive evaluation of twelve state-of-the-art repair algorithms, revealing practical trade-offs between repair quality and computational cost.

A key distinction between repair and deletion, as noted in our earlier work [27], is the direction of the problem. Repair begins with an *inconsistent* database and seeks a nearby consistent state. Inference-aware deletion begins with a *consistent* database and must produce a post-deletion state that remains consistent while also satisfying an erasure goal. Thus, although repair techniques are relevant, their objective is not the same: restoring consistency is not equivalent to making a deleted value uninferable.

Provenance, attribution, and causality. Provenance tracks how output data derives from input data. Semiring provenance [28, 48] annotates query results with algebraic expressions over input tuple identifiers, enabling precise reasoning about which tuples contributed to which outputs. Provenance is relevant to deletion because it identifies what may need to be changed when a source tuple is removed. Recent work on Shapley-based attribution [15] quantifies the contribution of individual tuples to query answers, providing a principled measure of deletion impact. Bertossi [14] establishes deep connections between database repairs, counterfactual causality, and attribution scores, showing how repair-based reasoning can inform deletion decisions. These frameworks are valuable because they identify *what depends on what*. They do not, however, answer the inference question addressed in this thesis.

Dependency frameworks and discovery. Dependencies are central to the deletion problem because they encode the inferential structure of the data. Functional dependencies, denial constraints [29, 73], and conditional functional dependencies [42] have been studied extensively in data cleaning [103] and consistent query answering [5]. Delta rules [47] generalize reasoning across a broad class of dependencies for deletion-based repair. Dependency discovery has also been studied in several forms, including functional dependencies [18], approximate functional dependencies [91, 95], denial constraints [93, 92], and conditional functional dependencies [19]. Parciak et al. [91] provide a formal comparison of approximate functional-dependency measures using Shannon and logical entropy, explicitly connecting information-theoretic measures to dependency quality. Kaminsky et al. [60] introduce an incremental approach for detecting denial-constraint violations as data changes, avoiding an expensive full recomputation.

An important caveat is that discovered dependencies may be unreliable. Martin et al. [77] show that over 95% of denial constraints discovered by state-of-the-art algorithms may be false. From a deletion perspective, this risk is asymmetric. Over-specified dependencies, including false ones, can lead to over-deletion, which is conservative with respect to inference but costly in utility. The more dangerous case is *under-specification*: missing dependencies that allow the deleted value to remain inferable from what the system retains.

The relational dependency rules (RDRs) introduced in this thesis (Chapter 4) build on these dependency frameworks, but adapt them to the needs of fine-grained erasure. They support cell-level dependencies and aggregate dependencies that are essential for reasoning about inference after deletion, while remaining grounded in SQL for expressiveness and ease of specification.

2.3 Deletion as Inference Control

The preceding sections frame deletion as a storage operation or a consistency problem. This thesis argues that deletion is also an inference-control problem: the goal is not only to remove an artifact or maintain a constraint but also to limit what can be *known* about the deleted value from what the system retains and exposes. This section surveys work on classical inference control, differential privacy, semantic generalizations of privacy, and deletion-specific privacy formalizations.

Classical inference control in statistical databases. The observation that query answers can reveal information about individual records predates the modern deletion literature by several decades. Denning, Denning, and Schwartz [37] introduced the "tracker" attack, demonstrating that carefully chosen aggregate queries can compromise individual records in a statistical database. Adam and Wortmann [1] provide a comparative study of restriction-based and perturbation-based defenses, while Needham [38] offers a broader view of inference control. Classical inference control asks: *can a query sequence reveal an individual record?* On the other hand, deletion privacy asks: *can the post-deletion database state reveal the deleted value?* Both questions pertain to inference problems. Deletion, however, introduces an additional leakage channel: the deletion pattern itself.

Differential privacy. Differential privacy (DP) [39] provides a mathematical framework for bounding the information that an observation reveals about any individual record. A mechanism satisfies ε-DP if its output distribution changes by at most a factor of e^{ε} on neighboring databases, typically databases that differ in one individual's record. The exponential mechanism [79] enables a differentially private selection from discrete options via utility-weighted sampling. Standard DP is powerful because it provides a robust syntactic guarantee about the effect of changing one record. However, when data contains correlations or constraints, the protected secret may not be captured by changing one tuple independently.

Kifer and Machanavajjhala [61] show that privacy and utility guarantees require explicit assumptions about the data-generating setting, motivating semantic generalizations of DP.

Semantic generalizations of DP. Pufferfish [62] generalizes DP by allowing explicit specification of secrets, discriminative pairs, and data evolution scenarios, thus enabling privacy definitions that are tailored to correlated data and application-specific secrets. Song et al. [114] develop practical mechanisms for Pufferfish, including the Markov Quilt Mechanism, which exploits Bayesian-network structure to improve tractability for correlated data. Blowfish [53] introduces policy graphs to specify which secrets to protect and which constraints may be known. Bayesian Differential Privacy [126] evaluates privacy under correlated data and incomplete prior knowledge, while Dependent Differential Privacy [72] introduces dependence coefficients to quantify privacy loss under correlated tuples. These frameworks are highly relevant because they go beyond independent tuple-level adjacency. Our work in Chapter 6 differs from the frameworks mentioned earlier in the way that we construct the adjacency relation. In our work, dependencies are built directly into a cell-level dependency hypergraph, so changing one cell value may require cascading changes across multiple tuples in order to retain consistency with the constraints.

DP in database systems. Applications of DP in databases include SQL engines with constraint handling [66, 59], federated querying [11], streaming [104], update-pattern hiding [123], and inconsistency estimation under DP [80]. These works add noise to *outputs* of queries, obfuscate *access patterns*, or protect intermediate query-processing information. In contrast, DiffSEAL (Chapter 6) uses DP to make selections between *deletion strategies*: the mechanism's choice of which cells to delete is itself the private output, and this choice must be insensitive to the true value of the deleted cell.

Deletion-specific privacy formalizations. Several recent works formalize what it means for deletion to be private. Garg, Goldwasser, and Vasudevan [45] provide a cryptographic

formalization of deletion: *deletion compliance* requires that a system after deletion be computationally indistinguishable from one that never observed the deleted data. This is closely related to the epistemic deletion claim of this thesis, but is developed in a cryptographic setting. Gao et al. [44] formalize deletion inference and deletion reconstruction attacks, where an adversary exploits before-and-after model pairs to identify or reconstruct deleted data, and show that deletion compliance prevents these attacks. Chourasia and Shah [30] demonstrate that prior unlearning guarantees can fail to protect deleted records and propose a sound deletion guarantee, proving that the privacy of existing records is necessary for the privacy of deleted records. These works establish that deletion is not merely an operational requirement but is also a privacy problem with formal structure. Our work builds on this understanding at the database level, where semantic dependencies create structured inference paths that must be explicitly addressed.

2.4 Deletion as a Right

Data regulations have elevated deletion from a storage-management feature to a compliance obligation. This section surveys work on formalizing data regulations, building compliance-oriented systems, and managing deletion and retention obligations in deployed systems.

Formal specifications of data regulations. Logic-based formalizations of the GDPR have been explored using deontic logic [107, 106] and temporal model checking [31, 7]. These specifications operate at the level of broad data-processing concepts and support compliance verification through formal reasoning. Basin et al. [10] present a language-based approach to Privacy by Design, using Metric First-Order Temporal Logic (MFOTL) with purpose tracking. This tracking maintains a stack of active purposes during execution and instrumentation, intercepting personal-data actions to ensure that each action matches a declared consent. Hublet et al. [56] study proactive, real-time enforcement of privacy policies. Hublet et al. [55] identify inaccuracies in prior GDPR formalizations and propose a methodology to create

enforceable specifications. These works provide formal reasoning about regulatory obligations, but they do not by themselves determine how concepts such as "erasure" should be *interpreted* or *implemented* in a database system. The same legal article can correspond to different system-level actions depending on the interpretation chosen. This is the gap that Data-CASE (Chapter 7) addresses.

Compliance-by-construction systems. K9db [3] achieves compliance by construction through schema-level ownership annotations and automatic deletion procedures, organizing storage around per-data-subject micro-databases. Agarwal et al. [2] introduce GDPRizer, a tool that uses foreign keys, query logs, data-driven relationship discovery, and DBA annotations to identify a data subject's personal data across legacy schemas. Klein et al. [64] present Fontus, a JVM-based taint-tracking framework that transparently labels personal data in existing Java applications and enforces data-protection policies without requiring changes in the application. SchengenDB [67] proposes compliance-oriented data placement, including storing facts about a consumer in a way that supports deletion and purpose-based use. These systems provide operational compliance mechanisms, but they do not give formal guarantees about what information remains inferable after deletion.

Deletion at industry scale. Jacques-Silva et al. [58] describe Meta's Unified Lineage System, which produces daily provenance graphs with billions of nodes and edges. Applied to deletion compliance, the system uses lineage and optimization to select approximately 1,700 core tables for deleted-data filtering and approximately 2,400 core tables for reduced retention, resolving roughly 21,000 downstream dependencies. This provides a rare published account of deletion infrastructure at a large industrial scale. An empirical study of 90 online services [108, 111] finds a 27% non-compliance rate with GDPR Article 17 erasure requests, underscoring the gap between regulatory requirements and deployed practice. The impact of GDPR compliance on database performance has been benchmarked in [112], revealing a metadata explosion phenomenon where compliance metadata can exceed personal data in size by an order of

magnitude.

Policy management and auditing. Sieve [89] provides middleware for scalable fine-grained access control in database systems. TattleTale [90] addresses a closely related inference problem: preventing sensitive data protected by access-control policies from being leaked through data dependencies. AuditGuard [74] studies database auditing under retention restrictions, where retention policies remove portions of history and audit queries must be answered over incomplete records. Sarkar et al. [109] propose query-language extensions for expressing retention policies. Basin et al. [9] explore purpose-based data processing. Contextual integrity [8] frames privacy as appropriate information flow. This is related but orthogonal to our contributions, as our goal is not to define what privacy should mean but to reason about whether a system's deletion mechanisms are adequate given a chosen interpretation.

These developments, taken collectively, reveal a gap. Formal specifications reason about regulations, but not about how ambiguous regulatory concepts map to concrete database actions. Compliance systems implement operational deletion, access control, lineage tracking, retention, and auditing, but generally lack formal guarantees about inference after deletion. Data-CASE (Chapter 7) fills this gap by grounding regulatory language into system-level actions through formal semantics that can support multiple interpretations of erasure.

2.5 Deletion Beyond the Data Layer

Deletion cannot stop at the database. Views, caches, learned models, and downstream artifacts may all encode information about deleted data. This section surveys work on machine unlearning and related directions, which address deletion once data has influenced artifacts beyond the base database state.

Machine unlearning. Machine unlearning aims to remove the influence of specific training data from a learned model. Bourtoule et al. [20] introduce SISA training, which partitions data into shards and slices so that deletion requires retraining only the affected portion of the ensemble, achieving significant speedups over full retraining. Guo et al. [49] define *certified removal* for linear classifiers using second-order update mechanisms, where the model after removal is indistinguishable from one that was never trained on the removed data. Gupta et al. [51] study adaptive deletion sequences, where removal requests may depend on previously published models. They provide a reduction from adaptive deletion guarantees to non-adaptive ones using differential privacy, and they demonstrate practical attacks against SISA under adaptive deletion sequences. Eisenhofer et al. [40] develop a cryptographic framework for verifiable unlearning, using SNARKs and hash-chain-based commitments to let users audit whether an unlearning procedure was executed correctly.

Unlearning in data structures. A directly relevant line of work examines unlearning in *learned database components*. Kurmanji et al. [70] provide a systematic experimental study of unlearning effects on neural-network-based database components, including selectivity estimation, approximate query processing, and data generation. They show that unlearning in learned database systems differs from standard ML unlearning: accuracy requirements are tied to downstream query-processing tasks; data distributions and update patterns are database-specific; and the effects of unlearning must be evaluated through both model-internal and task-level metrics. This work highlights that deletion must eventually be addressed not only in the base data but also in the learned components built on top of it.

Attacks on unlearning. The guarantees of unlearning are not always as strong as they appear. Bertran et al. [16] demonstrate near-perfect reconstruction of deleted data from linear regression models, showing that the act of requesting deletion can paradoxically enable data recovery when the model update is not protected by appropriate privacy mechanisms. Gao et al. [44] formalize deletion inference and deletion reconstruction attacks, where adversaries

exploit before-and-after model pairs to identify or reconstruct deleted data. They establish formal conditions under which deletion compliance prevents such attacks.

Cryptographic certified deletion. In the cryptographic setting, Broadbent and Islam [22] introduce quantum encryption with certified deletion: a receiver possessing a quantum ciphertext can generate a classical certificate proving that the encrypted message has been deleted. Their work is motivated by a fundamental limitation of classical information: classical ciphertexts can be copied, so this form of certified deletion cannot be achieved using classical information alone. Although not directly applicable to database systems today, this work clarifies theoretical limits on what deletion guarantees can mean in cryptographic settings.

Data-layer mechanisms, such as those developed in this thesis, and model-layer mechanisms, such as machine unlearning, address complementary leakage surfaces. The former prevent inference through the visible database state; the latter prevent inference through trained model parameters. Neither suffices alone for systems where databases feed into machine learning pipelines. Extending dependency frameworks to cover dependencies between base data and learned models remains an open direction, discussed further in Chapter 8.

Chapter 3

A Taxonomy of Deletion

Having surveyed existing work related to data deletion, we now focus on deletion mechanisms from the perspective of privacy. Deletion is not a single operation but a family of mechanisms that differ in what they guarantee, whom they guarantee it against, and what they leave behind. This chapter develops a three-layer view of a deletion request, states what a privacy guarantee must specify, and organizes the design space accordingly.

3.1 Deletion as a Privacy Preserving Operation

The database community has long been careful about inserting data into databases. A system that accepts arbitrary data becomes unusable for querying. To maintain meaningfulness of stored data, a layer of declarative statements sits above the stored data. For example, domain and `CHECK` constraints reject values that are meaningless under their intended interpretation; key constraints ensure that tuples can be identified; foreign keys prevent references to non-existent data. Assertions and denial constraints, on the other hand, rule out states that are individually well-formed but jointly contradict a regularity of the modelled semantics.

Data constraints, while themselves not data, describe what stored data is permitted to *mean*.

The system compiles them into checks over tuples, indexes, and pages. Insertion therefore has three parts: semantics, a data representation, and physical storage, along with mappings between these parts.

Contrastingly, data deletion received a different treatment. It has traditionally been viewed as a means for *storage reclamation*: a record is no longer needed, its space should be returned, and the question is when and at what cost. The two-way distinction thus follows naturally — *physical deletion* manipulates the bits that hold a value and *logical deletion* manipulates visibility, so the value remains in storage while queries no longer return it. Reclamation asks only whether the space has come back and whether anyone can still see what is in it, and these two classifications answer those two questions.

Data regulations changed the status of data deletion [6, 25]. Deletion has become a privacy and security obligation rather than a housekeeping feature, and the existing notions of deletion say nothing about what remains *knowable* post-deletion. Removing a value from query results does not remove the information it contributed, and neither does reclaiming its bytes.

Prescriptive and descriptive dependencies. Insertion and deletion use the same formalism in opposite ways. Insertion uses dependencies *prescriptively*: a declared functional dependency is enforced, so no state violating it is admitted. Whereas deletion must use them *descriptively*, because a dependency describes what data comprise a fact in the semantic layer.

For example, if $\mathsf{level} \rightarrow \mathsf{salary}$ is a statistical regularity, an adversary who knows an employee's level guesses the band. If it is an enforced constraint, the adversary derives it, because the system has guaranteed that no counterexample exists. The integrity machinery that makes a database trustworthy and meaningful also facilitates inference for an adversary.

If deletion is to become a first-class privacy operation, it must be specified declaratively, compiled into a plan, and given a guarantee that can be checked. We argue that this requires

the three-part structure insertion already has: a semantic layer, a data layer, and a storage layer, with explicit mappings between them.

In the next section, we analyze a deletion request, often motivated by a user exercising their right to deletion granted by data regulations.

3.2 The Anatomy of a Deletion Request

We begin eith an example. An employer maintains four relations: Employees(*e_id*, dept, title, level, salary), Pay(*e_id*, wk_hrs, hourly), Expense(*dept_id*, avg), and Tax_Rate(*e_id*, rate). Compensation rules induce dependencies among these attributes. Title constrains level, level constrains the salary band, and the band constrains the hourly rate, the departmental average, and the tax rate. Sam exercises a right to erasure over his salary, so the target cell is $c^* = \mathsf{Employees}[\text{Sam}, \text{salary}]$.

We distinguish two parties/observers throughout. A *querier* interacts with the system through its query interface, i.e., the data layer. A *storage owner* holds the medium and reads bytes directly, so they see everything the querier sees and more. Section 3.6 makes this precise.

Sam asks that the system cease to convey what his salary band is. No system offers an operation over facts. A fact must first be mapped to data the system holds, and the gap between what was asked and what can be issued by the system is the first thing a taxonomy must account for.

Deleting the tuple. SQL offers no operation that removes a single cell, so the administrator issues `UPDATE` or appropriate `DELETE` queries. Under multiversion concurrency control, no byte is overwritten. The tuple is marked invisible to later snapshots, and the band remains on the page until reclamation runs. It also remains in the log record that captured the change, in the buffer pool, on each replica, and in the most recent base backup. A querier sees nothing.

A storage owner reads the value directly off the page.

Reclaiming the space. Running `VACUUM` afterwards removes the dead version of the tuple and returns the slot to the free-space map. What holds for the querier is unchanged. What changes is that the bytes are no longer reachable through the engine's own addressing, which is weaker than their absence from every medium: the write-ahead log, the archive, and the replicas were never in scope. Whether the bytes are overwritten depends on subsequent write transactions.

Deleting in a log-structured engine. In Cassandra the same request appends a tombstone that shadows the record on reads and merges [71]. The effect for the querier is immediate, as with `DELETE`. The record persists in its table until a compaction merges the two, and when that happens is governed by the compaction policy. The tombstone must also be retained for a grace period, so the act of deleting stores a readable assertion that a deletion occurred.

What remains inferable. In all three cases, the salary band is still re-inferable, and the exemplified operations never considered it. Sam's tuple in Pay survives, and the hourly rate it records is determined by the band. His tuple in Tax_Rate is not deleted and is determined the same way. The departmental average in Expense was computed from the band and retains information about it. The deletion operations exemplified thus fail to prevent inference, and no amount of further reclamation at the storage level would change this.

The three examples above to implement Sam's request produce three distinct outcomes at each layer, presented in Table 3.1. The value became unreachable through the query interface in all cases. However, it remained inferable from remaining deep-rooted data.

Deletion creates data. The dead tuple version, missing data, and the log record are objects created by the act of deleting data, and persist. The tombstone is the clearest case, since it is retained deliberately and states that a deletion happened. Auxiliary deletions are observable

Operation	**Semantic** *c^* not inferable (either observer)*	**Data** *no query returns c^* (querier)*	**Storage** *no legible bytes remain (storage owner)*
`DELETE/UPDATE`	fails; Pay and Tax_Rate rows still determine the band	holds	fails; dead version, log record, replica, backup
followed by `VACUUM`	fails, unchanged	holds, unchanged	partial; slot reusable but not overwritten, archive out of scope
tombstone (LSM)	fails, unchanged	holds immediately	deferred to compaction; the tombstone itself is retained

Table 3.1: Invariants of deletion requests.

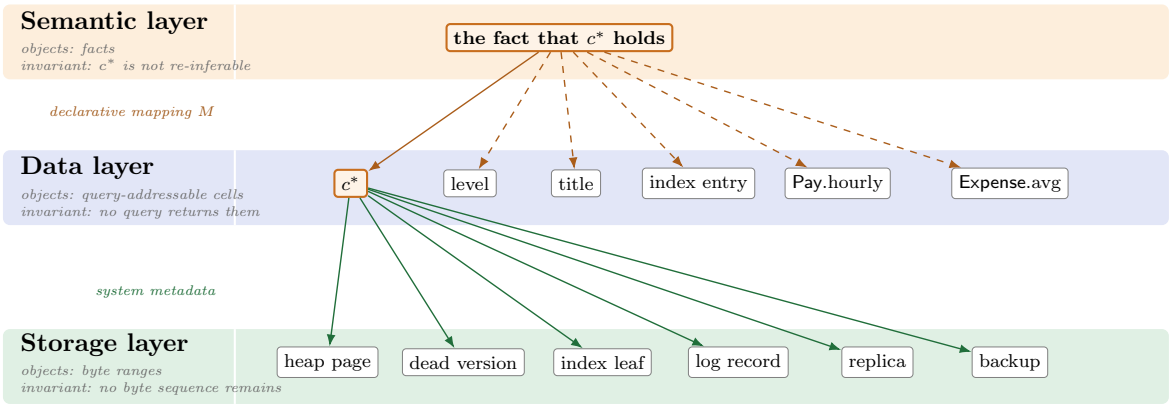


Figure 3.1: The three layers. A deletion request names a fact. A mapping M determines which data items represent it, and system metadata determines which byte ranges realize each item. The upper mapping is a modelling choice; the lower is recorded by the system.

in the same way: what was removed alongside the target is a signal about the target. We return to this as the deletion pattern P in Section 3.7.

What a privacy guarantee must specify. The anatomy shows that "the data has been deleted" is not a well-formed meaningful guarantee. A deletion mechanism must state three things. First, the *invariant it establishes at each layer*, since the three are separately satisfiable. Second, the *observer* that invariant binds, since a property established at one layer constrains only parties confined to it. Third, the *mappings* between layers the guarantee is stated with respect to, since which data represents a fact is a modelling choice and which bytes represent a data item is a fact the system records. Sections 3.3 through 3.6 develop each in turn.

3.3 The Three Layers In Which Deletion Exists

Now we are ready to describe the three layers.

The semantic layer. Its objects are facts, and its edges are the dependencies that make one fact inferable from others. These may be declared, discovered, or learned. A deletion request originates here and its guarantee is ultimately stated here. The layer has no native operation, because facts are not stored. It is a *specification* layer: one states which fact must cease to be inferable, and that statement is then compiled into operations at the layers below.[1]

The data layer. Its objects are the query-addressable items in which the fact is represented. In the running example, Sam's level and title in teh database, the corresponding row of Pay, the departmental average in Expense, an index entry over the salary attribute, and any materialized view that touches it exist in the data layer.

The storage layer. Objects in this layer are the byte ranges that realize each data item. The relation is one-to-many. For example, a single cell is carried by a heap page, by dead versions retained for concurrency control, by index leaves, by log records, by replicas, and by backups. The recovery subsystem exists to make effects survive, and every mechanism through which it does so creates another object a deletion must reach.

We draw the boundary by *addressability.* An object the system exposes through its query interface is a data item, and therefore participates in the dependencies of the semantic layer; an index entry, a materialized view row, and a history row in a system-versioned table are examples. An object the system does not expose is a storage representation, reached through layout metadata rather than through a dependency; a log record, a replica page, and a backup block are examples.

[1]We use *semantic* in the sense of what stored data lets one know. This is unrelated to its use in layered transaction models, where it refers to the commutativity of high-level operations.

Observe that the boundary is system-dependent. In a store that exposes history as a queryable relation, prior versions are data items carrying dependencies. In an engine whose log is not queryable, the same information is a storage representation.

3.4 The Mappings Between Layers

The two mappings between the three layers of Figure 3.1 are different in kind. We elaborate on these in teh following.

Semantic to data: a modelling choice. Which data items represent a fact is not something the system records. It is supplied by a mapping M, and different systems supply very different ones. Consider what each answers on the running example, where the request is to delete Sam's salary band.

Referential integrity is the mapping used in `CASCADE` [99]. The items representing a parent's existence are the rows that refer to it. If Pay.e_id and Tax_Rate.e_id are foreign keys into Employees, deleting Sam's tuple cascades to his Pay and Tax_Rate rows. No foreign key relates level to salary, so for the request as stated the mapping proposes nothing at all.

Ownership is the mapping used in K9db [3]. The schema declares how each table's rows are owned by users. Here an Employees row is owned by the employee it describes, and Pay and Tax_Rate rows are owned through e_id. "Sam's data" is then those three rows. The departmental average in Expense is owned by no individual and is left alone, and so are the rows of Sam's colleagues.

Reachability is used as the mapping in DELF [34]. Each edge type of a graph store is annotated deep or shallow. Modelling the example as a graph, the edge from an employee to their pay record would be deep, so deleting Sam removes it, while the edge from an employee to their department would be shallow, so traversal stops before the department and

its aggregate.

Each is a legitimate answer to what a deletion request might mean. What distinguishes them is which semantics they encode. Ownership and reachability are proxies for dependence.

Inference as the mapping. This thesis takes inference as M. A data item represents c^* when the remaining state, interpreted under a stated dependency model, supports re-inference of c^* through it. On the running example, that set contains Sam's Pay and Tax_Rate rows, since both are determined by the band, and the rows of colleagues at the same level, which the other three mappings do not reach.

Not all dependencies of such a model are equally strong. Mined dependencies hold with a confidence, whereas enforced constraints hold by construction. The closure over enforced edges is therefore the part of Σ on which a guarantee is least negotiable.

Choosing base-relation deletions that realize a requested change to a derived object is not a new problem. Deletion propagation asks: given a request to remove a tuple from a view, which base tuples must go, minimizing side effects. The line of work has produced complexity characterizations and practical solvers [63, 76]. That is a semantic-to-data compiler with a cost objective, which is structurally the component this chapter argues is missing. What differs is the invariant it compiles against. *View correctness is not non-inferability*, so its side-effect measure counts tuples the view should not have lost, not evidence an adversary should not retain.

Declared, discovered, and learned mappings. Calling M declarative describes how the mappings above are supplied. A *declared* mapping is written by a schema designer: a foreign key, an ownership annotation, an edge label. It is auditable and stable, and incomplete in the way any hand-written specification is.

A *discovered* mapping is mined from an instance [92, 18, 57]. It is estimated is a property of

the current instance. A dependency that holds today may not hold after a batch of inserts, so a guarantee established at deletion time can lapse without anyone touching the deleted data.

Provenance is orthogonal to how the mapping is quantified. It governs whether a guarantee can be audited, whether it decays as the instance evolves, and whether a closure can be computed at all.

Data to storage: recorded, not declared. The lower mapping is of a different character. Which byte ranges realize a data item is a fact the system already maintains, distributed across metadata that storage engines and operating systems keep for their own purposes: free-space maps and page directories, transaction-visibility fields and version chains, index structures, write-ahead and append-only logs, snapshot and archive catalogues, replication topologies, and, below the database, and file-system tables. It is therefore available in principle and fragmented in practice. No single component holds it, several may lie outside the database, and some record relocations the database can neither observe nor undo. For the purposes of a guarantee, a mapping that is recorded but unreadable is no better than one that was never stated.

Guarantees that do not depend on the mapping. Both mappings are stated, so every guarantee expressed through them inherits their incompleteness. It holds with respect to a declared Σ and a known set of storage representations. A different shape of guarantee would not carry this dependence, asserting instead that some property of the deletion holds whatever the adversary knows. The appeal of differential privacy rests substantially on being of this kind, bounding an observer's belief change without quantifying over the observer's prior. Whether an analogous statement can be formulated for deletion is an open question that this thesis does not settle.

3.5 Invariants at Each Layer

We now state the three properties of Table 3.1 as invariants relative to an observer O. Table 3.2 records what establishing each one requires, what it costs, and how it may fail.

The three differ in what it takes to *check* them. The data-layer invariant is decidable given the query language. The storage-layer invariant is decidable in principle and unverifiable in practice, since the set of media within an observer's reach is open-ended and partly outside the database. The semantic-layer invariant is not checkable by the system at all, because it quantifies over an adversary's belief; it can only be bounded relative to a declared model. This is why only the semantic layer requires a model and a reference point, and it caps what any attestation can assert.

The data-layer invariant. No query expressible through the interface returns c^* to O. It is established by manipulating visibility, not by removal. The `DELETE` of Section 3.2 records the deleting transaction in the tuple header so later snapshots skip the version. The tombstone appends a marker that shadows the record on reads and merges [71]. Neither rewrites the value, so both cost constant work per item.

The storage-layer invariant. No byte sequence from which c^* can be reconstructed remains on any medium within O's reach. Reclamation is not erasure. Vacuuming returns the slot to the free-space map [100] and compaction drops the shadowed record [110, 71], but neither overwrites.

The semantic-layer invariant. After observing what remains, O's belief about c^* is no stronger than at a stated reference point. It cannot be stated absolutely, since some information about c^* was available before it was ever stored. The natural reference point is the state of knowledge before insertion, so what deletion undoes is the incremental inferability that storing the value introduced [27].

Layer	Established by	Cost	Typical failure
Data	visibility metadata: version headers, tombstones, keyspace removal, TTL sweeps	constant per item; nothing rewritten	partial coverage of the interface: indexes, views, caches, history, statistics
Storage	overwriting, or rendering bytes unreadable; reclamation alone is insufficient	write amplification, proportional to volume rewritten	unbounded latency; media outside the database's control
Semantic	deleting an auxiliary set $\mathcal{T}$ that blocks the inference paths	utility loss, borne partly by the subject and partly by other users	model incompleteness; no stated reference point

Table 3.2: What establishing each invariant requires, what it costs, and how it typically fails.

Establishing it requires deleting an auxiliary set $\mathcal{T}$ that blocks the inference paths. Mechanisms differ chiefly in how $\mathcal{T}$ is chosen, which by Section 3.4 follows from the mapping they adopt. None of the three operations of Section 3.2 states this invariant, and none establishes it.

How the invariants interact. The semantic invariant is stated over everything observer O sees, so against a given O it subsumes the layer invariants at every layer O reaches. Were the cell still returned or the bytes still legible, O would read the value rather than infer it. This is a consequence of how the observer is defined, not a theorem.

The three layers are separately achievable, by disjoint machinery, at costs that differ by orders of magnitude and with different degrees of checkability. No deployed system implements semantic deletion as a primitive. Every system implements pieces of it, and the pieces line up with the layers.

Read contrapositively, the subsumption has operational consequence. A failure at any layer the observer reaches defeats the semantic guarantee for that observer. In particular, semantic erasure against a storage owner requires the storage invariant, since no amount of auxiliary deletion substitutes for legible bytes. The subsumption holds because legible bytes disclose their contents, so where the representation is not interpretable, as when each subject's data is encrypted under a key that is then destroyed [102], the bytes may remain while conveying

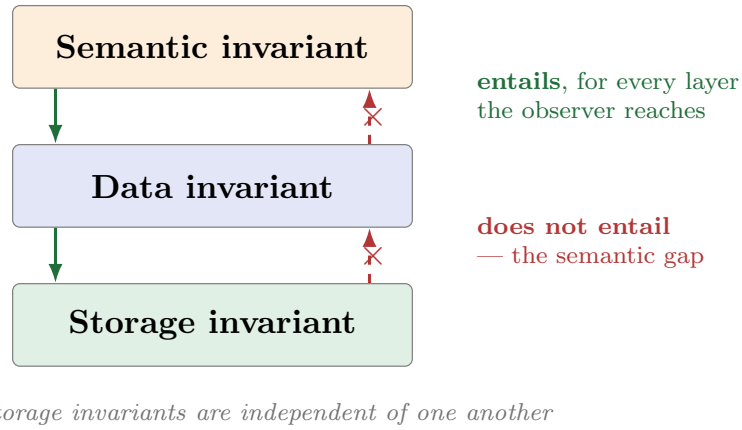


Figure 3.2: Relations among the layer invariants. A semantic invariant entails the layer invariants at every layer the observer reaches. The converse fails, and the data and storage invariants are independent of one another.

nothing.

The converse fails. We refer to this as the *semantic gap*. In the running example, even if a mechanism erases every byte of c^* and returns it from no query, the band is still recovered from Sam's surviving Pay and Tax_Rate rows. Figure 3.2 summarizes the three relations.

3.6 Observers

The invariants above are stated relative to an observer O. We characterize O by two independent properties: how far it *reaches* into the layers, and how far it *reasons* beyond what it reads. Figure 3.3 shows the resulting space.

Reach is nested. A querier's direct observations are data items, namely whatever the system chooses to return to the query interface. A storage owner reads byte ranges directly, and observes everything the querier observes as well. Cloud providers, administrators, whoever acquires a decommissioned disk, and an adversary with a filesystem foothold are all storage owners in this sense. A guarantee established at the data layer therefore binds the querier and leaves the storage owner unconstrained, which is what the example of Section 3.2 illustrates.

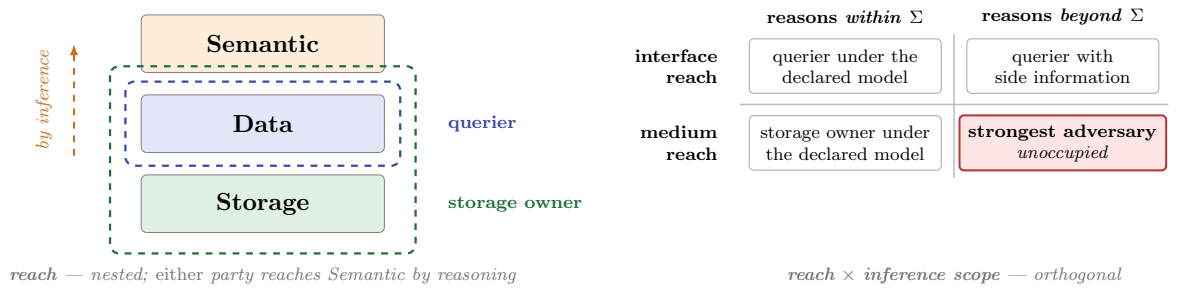


Figure 3.3: An observer is a reach and an inference scope. Reach is nested; inference scope is orthogonal to it.

Inference scope is orthogonal. Either observer may reason about what it reads. The natural first assumption is that it reasons under the declared model, but that model is an approximation. Dependency discovery is incomplete by construction, correlations may be strong without being declared, and an adversary may hold side information the system has never seen. An observer reasoning beyond Σ is the realistic case.

3.7 Mechanism-Intrinsic Axes (T1–T8)

We now factor the design space. Beyond the layer structure, deletion mechanisms differ in what they observe, how they model dependencies, what they cost, and how they are realized. We separate *mechanism-intrinsic* axes, which capture what a system can guarantee regardless of who invoked the deletion or why, from *invocation-context* overlays, which matter in policy discussions but are not properties of a deletion primitive. Figure 3.4 places representative systems along the intrinsic axes.

A mechanism produces a post-deletion state, possibly performing auxiliary actions such as additional deletions, perturbations, or derived-data refreshes. We call the footprint of these actions the *auxiliary set* $\mathcal{T}$. An adversary can then observe two channels: the *residual state* V, the data that remains visible, and the *deletion pattern* P, what the deletion operation itself reveals, including the structure and timing of $\mathcal{T}$. Most existing mechanisms address only V.

T1: Layers addressed, and the semantic mapping. The most basic distinction is at which layers a mechanism establishes an invariant. Since a mechanism may address any subset of the three, this axis is a subset. `DELETE` [99] and TTL-based removal in document stores [81] address the data layer alone. Lethe, timely deletion, and tombstone-based deletion in Cassandra address the data and storage layers [110, 109, 71]. `CASCADE`, DELF, K9db, Edna, and the mechanism of Chapter 5 address the semantic and data layers [99, 34, 3, 121, 27] but under varying semantics.

Mechanisms reaching the semantic layer are further distinguished by the mapping M they use: referential integrity for `CASCADE`, ownership for K9db, reachability for DELF, a disguise-and-reveal correspondence between stored and displayed values for Edna [121], interpretation for key destruction, and inference for the mechanisms of this thesis. This second distinction matters because it, and not the layer alone, determines whether a semantic invariant entails anything about re-inference of the deleted value.

T2: Observables controlled (V only, P only, $V{+}P$). A mechanism may aim to control leakage through V, through P, or through both. Most systems control only V and treat the deletion footprint as an implementation byproduct. Secure deletion [102] addresses storage-level threats but not semantic inference. Inference control and differential privacy provide frameworks for probabilistic guarantees, but they have not been applied to the choice of auxiliary cells deleted by a database erasure mechanism. The P-only and $V{+}P$ cells are unoccupied by prior database deletion mechanisms.

The maintenance a deletion triggers belongs to P as well, and can leak on its own. In the running example, Expense.avg was computed from Sam's band, so recomputing it after deletion changes a published value by an amount that discloses the removed contribution. This is a differencing attack, familiar from the statistical-database literature: the leak is caused not by what the system retains but by the action the deletion triggered.

T3: Semantics model (none, binary, weighted, slack, model-independent). Where T1 records which mapping a mechanism uses, T3 records what its guarantee is stated with respect to, and therefore how far an adversary's reasoning may extend.

Binary models treat each declared dependency as present or absent, so deletion must block every active dependency. This includes schema rules such as `CASCADE` [99], annotation and ownership graphs [34, 3, 121], and dependency-scoped semantic rollback as in P2E2 [27].

Weighted models assign strengths to dependencies, reflecting how reliably each enables inference in practice. As Section 2.2 showed, dependency discovery routinely produces approximate constraints with confidence scores [91, 95, 92], yet existing mechanisms do not exploit these weights as part of the erasure guarantee.

Slack captures the residual inferential support lying outside the specified model: correlations, background knowledge, statistical signals, or dependencies no extraction procedure captures. A slack-aware guarantee binds an observer reasoning beyond Σ. A *model-independent* guarantee would make no reference to a declared model at all, and Section 3.4 states this as open. Mechanisms establishing no semantic invariant have no semantics model and appear here only for completeness.

T4: Recoverability (irreversible, reversible window). Some mechanisms preserve a recovery window by design. DELF supports temporarily reversible deletion operations [34]. Others are irreversible in the operational sense that no per-item undo contract exists after the operation commits. Recoverability is a per-layer property and does not transfer between layers. A mechanism may offer no undo contract at the data layer while every byte it targeted remains recoverable from a backup.

T5: Expiry-awareness (external, native). The distinction is whether the mechanism natively represents expiry or deadline metadata, or whether scheduling is handled externally by the

caller. Mongo TTL [81], Lethe [110], timely deletion [109], and Cassandra [71] treat time as a first-class entity. `DELETE`-based systems can be wrapped in an external scheduler, but the engine remains oblivious to expiry metadata. This axis affects performance, batching opportunities, and the visibility of deletion footprints.

T6: Storage substrate (LSM-tree, MVCC/heap, app-layer). Physical semantics are constrained by the substrate over which the data-to-storage mapping is recorded. LSM-based engines rely on tombstones and compaction [110, 71]. MVCC and heap engines retain dead versions until vacuum or purge [100, 109]. Application-layer systems orchestrate deletion across multiple stores and derived artifacts [34, 3, 121]. This axis determines which portions of the inference timeline (Figure 1.1) are reachable in a given deployment.

T7: Deletion cost (target-only, closure-bounded, subject-wide, budgeted). A semantic guarantee is purchased with auxiliary deletions, and how many is a property of the mechanism rather than of the request. *Target-only* mechanisms delete nothing beyond c^*, which includes `DELETE` [99], TTL removal [81], and the timeliness mechanisms [110, 109, 71]. This is unsurprising, since none of them makes a semantic claim. *Closure-bounded* mechanisms delete the closure of the target under their declared mapping: the referential closure for `CASCADE`, the reachability closure for DELF [34], and the inference closure for P2E2 [27]. *Subject-wide* mechanisms delete everything attributed to the requester, as in K9db's ownership closure [3], bounding cost by the subject's footprint rather than by the target's dependencies.

A fourth position, *budgeted*, would choose $\mathcal{T}$ to minimize cost subject to a stated leakage bound rather than to eliminate leakage outright. This is the position Chapter 6 occupies, and it is otherwise unoccupied.

Note that the cost at this axis is also utility loss. Similarly, systems overhead is a product of T7 and T6 instead of a separate axis. The same closure amplifies into compaction work on a log-structured engine, into index maintenance and vacuum pressure on a heap engine, and

into cross-store orchestration at the application layer.

T8: Verifiability (asserted, logged, attested). A guarantee is useful to a data subject or a regulator only to the extent that it can be communicated and checked. *Asserted* guarantees are reported without an accompanying statement of what holds; this describes most mechanisms surveyed here. *Logged* guarantees leave an operational record that an operation ran, which supports auditing that a request was serviced but not that any invariant was established.

An *attested* guarantee would produce a machine-checkable certificate naming the layers addressed, the observer bound, and the semantics model assumed, so that claims made by a database, its storage engine, and its downstream artifacts could be composed.

This axis is where the taxonomy touches what a deletion operation informs its initiator. Under T1 through T6, two mechanisms may be equally strong and yet differ entirely in what a subject can be told. For example, "the row is gone" is compatible with the value remaining legible on a backup, and a subject has no way to distinguish the cases from the system's response.

3.8 Invocation-Context Overlays (I1–I2)

Two further axes appear frequently in compliance discussions but are not intrinsic deletion mechanism properties.

I1: Initiator. Deletion may be initiated by a data subject, an administrator, or an automated policy engine. This is a calling-convention property: any mechanism can be invoked by any initiator, depending on deployment. The initiator is easily confused with the observer. The initiator is who asks; the observer is who the resulting guarantee binds. They are related only in that a data subject has little reason to accept a guarantee binding the querier alone.

I2: Trigger mode. Deletions may be demand-driven, as in deletion on request, or retention-driven, as in expiry or lifecycle policies. This is a deployment property. The taxonomy captures the structural consequence of time through T5 rather than conflating it with trigger mode.

The axes are not independent. We list five constraints below.

L1. Deleting beyond the target requires a semantic mapping. A mechanism can remove data other than c^*, the target cell, only under a rule that says what else to remove, and such a rule is a semantic-to-data mapping. Hence T7 beyond target-only requires the semantic layer in T1. The target-only column is `DELETE`, TTL, Lethe, timely deletion, and Cassandra, precisely the mechanisms whose T1 excludes the semantic layer. The converse holds as capability, since a semantic mechanism may compute an empty closure, as `CASCADE` does on the running example.

L2. Controlling the deletion pattern presupposes having one. The pattern P has two components. Its *content*, which cells were deleted and therefore which inference paths were blocked, exists only when the set of additional deletions, $\mathcal{T}$, is non-empty, and by L1 that requires a semantic mechanism.

L3. A reversible window contradicts the storage invariant. A reversible window requires the data to be reconstructible for its duration, which requires that it remain legible somewhere in the system. A reversible-window mechanism therefore violates the storage invariant while the reconstructible window is open, by construction. DELF's restoration interval [34] and Edna's reveal [121] are not weak erasure. They are a different operation, whose purpose is served by exactly the retention that erasure forbids. The cell (reversible, storage invariant established) is thus not feasible.

L4. Systems overhead is the product of T6 and T7. How much a mechanism deletes and the substrate it deletes on jointly determine its operational footprint. A mechanism's cost is therefore read off the pair.

L5. Attestation cannot exceed checkability. The three invariants differ in what it takes to verify them. A data-layer claim can in principle be attested outright. A storage-layer claim can be attested only over an enumerated set of media. A semantic claim cannot be attested absolutely; the strongest guarantee is relative to a stated Σ and a stated reference point.

Limitations L1–L5 partition the empty cells in the taxonomy. The cell (reversible, storage erased) is foreclosed by L3, and content-level P control for target-only mechanisms is foreclosed by L2. Everything else marked open in Figure 3.4 presents direction of interesting future work: joint (V, P) control for semantic mechanisms, weighted and slack-aware models, invariants at all three layers with a stated observer, cost-budgeted auxiliary selection, and composable attestation.

3.9 Discussion of Open Areas

Figure 3.4 highlights the mechanism-level gaps that motivate the technical contributions of this thesis.

Prior to this thesis, the semantic layer in T1 was reached only through mappings that are proxies for dependence. `CASCADE`, K9db, DELF, and Edna each supply a declarative semantics, but referential integrity, ownership, reachability, and disguise say nothing about whether the deleted value remains inferable. The inference mapping is what makes a semantic invariant a statement about knowledge rather than about structure.

The P-only and $V{+}P$ columns in T2 were likewise unoccupied. Secure deletion addresses storage-level threats, and differential privacy provides frameworks for probabilistic guarantees,

but neither has been used to protect the choice of which cells to delete as a private output of the deletion mechanism.

The weighted and slack-aware portions of T3 remain largely unoccupied for deletion guarantees. Dependency discovery [92, 18, 57] routinely produces approximate constraints with confidence scores, and leaves the possibility of unmodeled dependencies open. We are not aware of a formulation of deletion whose guarantee makes no reference to a declared dependency model, and remains an interesting future direction.

No mechanism establishes invariants at all three layers. Reading T1 and T3 together against the four observers of Section 3.6, every guarantee surveyed here binds a querier reasoning within the declared model. None binds a storage owner, and none binds an observer reasoning beyond Σ. The strongest adversary the framework expresses remains unaddressed.

Two further regions follow from the cost and verifiability axes. Under T7, no mechanism chooses its auxiliary set to minimize cost subject to a stated leakage bound. Under T8, no mechanism produces a statement a subject or regulator could check.

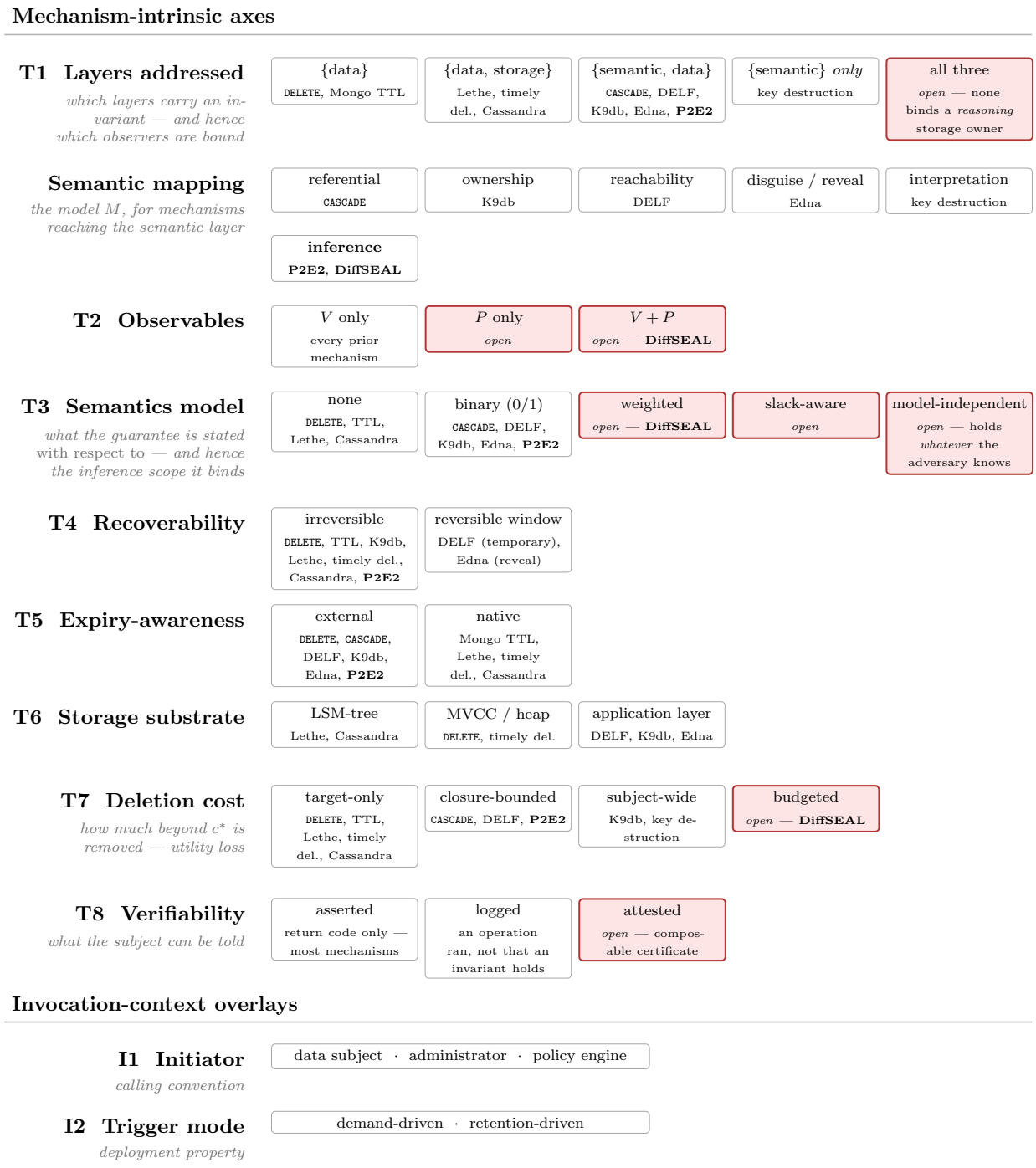


Figure 3.4: Taxonomy of deletion mechanisms across axes T1–T8 and overlays I1–I2. Shaded cells are unoccupied by prior database deletion mechanisms.

Chapter 4

Background and Preliminaries

This chapter establishes the formal framework used in the rest of this thesis. We begin with a cell-level relational data model, since the deletion requests studied here target individual values rather than entire tuples (Section 4.1). We then introduce relational dependency rules (RDRs), a SQL-grounded language for specifying which cells carry information about which other cells (Section 4.2). Instantiating these rules on a database instance yields a dependency hypergraph whose vertices are cells and whose hyperedges are local inference channels (Section 4.3). We then define inference chains over this hypergraph, including direct derivation, closure, inferability, and blocking (Section 4.4).

4.1 Data Model

We adopt a relational data model. The unit of deletion is a cell rather than a tuple. This choice is important for the applications that motivate this thesis: a deletion request may target a salary band, a location, or a manager assignment, while other attributes of the same record must remain available for legitimate operational use. We use Example 1.1 from Chapter 1 as the running example throughout this chapter.

Schema and instance. A *relational schema* is a pair $\boldsymbol{S} = (\mathcal{R}, \mathcal{A})$, where $\mathcal{R}$ is a finite set of relation symbols and $\mathcal{A}$ is a finite set of attributes. Each relation $R \in \mathcal{R}$ has a fixed signature $\mathrm{sig}(R) \subseteq \mathcal{A}$. Each attribute $\mathsf{A} \in \mathcal{A}$ has an associated domain $\mathsf{Dom}(\mathsf{A})$. Following the convention of our prior work [27], we treat each attribute as a partial function from record identifiers to domain values.

A *database instance* over $\boldsymbol{S}$ at time t is a state $\mathcal{D}_t$ that assigns, to each relation $R \in \mathcal{R}$, a finite set of records identified by unique *record identifiers* (rids). When the timestamp is unimportant, we just write $\mathcal{D}$. We denote by $\mathrm{Rids}(R, \mathcal{D})$ the set of rids of records in relation R in instance $\mathcal{D}$.

Cells and values. A *cell* is a pair (t, A), where $t \in \mathrm{Rids}(R, \mathcal{D})$ and $\mathsf{A} \in \mathrm{sig}(R)$. We write the cell as $\mathsf{A}(t)$ when emphasizing the attribute-function view and as $t[\mathsf{A}]$ when emphasizing the tuple-projection view. The two notations are synonymous. The set of all cells of $\mathcal{D}$ is denoted $\mathit{Cells}(\mathcal{D})$.

The *value* of a cell in instance $\mathcal{D}$ is given by the function $\mathit{Val}(\cdot, \mathcal{D}) : \mathit{Cells}(\mathcal{D}) \to \bigcup_{\mathsf{A} \in \mathcal{A}} \mathsf{Dom}(\mathsf{A}) \cup$ `NULL`. In the inference framework, a `NULL`-valued cell is unavailable as a value premise for deriving other cells. The position of a `NULL` cell may still be observable, however, and later chapters account for the information carried by the deletion pattern itself. We write $\mathit{Val}(c)$ when the instance is clear from context.

Target cell and auxiliary deletions. A *deletion request* identifies a *target cell* $c^* \in \mathit{Cells}(\mathcal{D})$ whose value is to be erased. In the $\mathsf{Employee}$ instance of Example 1.1, a request to delete record e_5 would remove not only the sensitive salary band but also the team, manager, and location, some of which may need to be retained for organizational purposes. The natural target of the request is therefore the single cell $c^* = \mathsf{SalaryBand}(e_5) = e_5[\mathsf{SalaryBand}]$.

Erasing a cell means setting its value to `NULL`. For a set of cells $\mathcal{U} \subseteq \mathit{Cells}(\mathcal{D})$, we write

$\mathcal{D}[\mathcal{U} \leftarrow \texttt{NULL}]$ for the instance that is identical to $\mathcal{D}$ except that every cell in $\mathcal{U}$ has value `NULL`. This operation can be realized by a SQL `UPDATE` that assigns `NULL` to the relevant attributes, or it can be realized in a column-store by removing the stored values of the selected cells.

A *deletion mechanism* takes as input a request $(\mathcal{D}, c^*)$ and produces:

- a *residual instance* $\mathcal{D}'$ over the same schema $\boldsymbol{S}$; and
- a set $\mathcal{T} \subseteq \mathit{Cells}(\mathcal{D}) \setminus \{c^*\}$ of *extra deletions*, consisting of additional cells whose values are erased in order to prevent the re-inference of c^*.

The set $\mathcal{T}$ is also called the *deletion footprint* of the mechanism. The residual instance is $\mathcal{D}' = \mathcal{D}[\mathcal{T} \cup \{c^*\} \leftarrow \texttt{NULL}]$. Mechanisms in this thesis differ primarily in how they choose $\mathcal{T}$.

Example 4.1 (Running Employee instance)**.** In Example 1.1 from Chapter 1, the schema $\boldsymbol{S}$ consists of a single relation Employee with the set of attributes $\mathcal{A} = \{\mathsf{Team}, \mathsf{Title}, \mathsf{Level}, \mathsf{Manager}, \mathsf{Location}, \mathsf{SalaryBand}\}$. The instance $\mathcal{D}$ has five records with rids e_1, e_2, e_3, e_4, e_5, recapitulated in Table 4.1. The target cell throughout this chapter is $c^* = \mathsf{SalaryBand}(e_5) = e_5[\mathsf{SalaryBand}]$, whose value in $\mathcal{D}$ is `B4`.

rid	Team	Title	Level	Manager	Location	SalaryBand
e_1	Ads	SWE	L4	$\mathtt{m}_{10}$	NYC	B4
e_2	Ads	SWE	L4	$\mathtt{m}_{10}$	NYC	B4
e_3	Ads	SWE	L5	$\mathtt{m}_{10}$	NYC	B5
e_4	Infra	SRE	L4	$\mathtt{m}_{12}$	SEA	B4
e_5	Ads	SWE	L4	$\mathtt{m}_{10}$	NYC	B4

Table 4.1: The Employee instance $\mathcal{D}$ (running example), with target cell c^* highlighted.

4.2 Relational Dependency Rules

The cell-level data model of Section 4.1 fixes the granularity at which deletion requests are issued. We now need a language to specify *which* cells carry information about *which* other cells, i.e., a *dependency model*. We adopt *relational dependency rules* (RDRs), introduced in

our earlier work [27], as the dependency model used throughout this thesis. RDRs express, in a SQL-grounded syntax, the claim that the value of one cell is not independent of the values of certain other cells.

An RDR has two parts: a *dependence statement* that names the attribute functions whose values are mutually dependent, and a *condition* that identifies, by a SQL query, the records to which the dependence applies.

Definition 4.1 (Relational Dependency Rule)**.** *Let $\boldsymbol{S} = (\mathcal{R}, \mathcal{A})$ be a relational schema with attributes $\mathsf{A}, \mathsf{A}_1, \ldots, \mathsf{A}_p \in \mathcal{A}$, and let Q be a SQL query over $\boldsymbol{S}$ whose output is a relation of rid tuples $(\boldsymbol{X}, \boldsymbol{X}_1, \ldots, \boldsymbol{X}_p)$. A* relational dependency rule (RDR) *r is given by*

$$\begin{aligned} &\textit{Dependence:} \quad \mathsf{A}(\boldsymbol{X}) \;\not\perp\!\!\!\perp\; \mathsf{A}_1(\boldsymbol{X}_1),\ \ldots\ , \mathsf{A}_p(\boldsymbol{X}_p) \\ &\textit{Condition:} \quad Q \end{aligned} \tag{4.1}$$

The dependence statement asserts that, on every tuple of rids returned by Q, the value of the cell $\mathsf{A}(\boldsymbol{X})$ is not independent of the values of the cells $\mathsf{A}_1(\boldsymbol{X}_1), \ldots, \mathsf{A}_p(\boldsymbol{X}_p)$. We refer to $\mathsf{A}(\boldsymbol{X})$ as the head *of r and to $\mathsf{A}_1(\boldsymbol{X}_1), \ldots, \mathsf{A}_p(\boldsymbol{X}_p)$ as the* tail *of r. When an attribute function appears in both the dependence and the condition, it is dropped from the condition to avoid duplication.*

The symbol $\not\perp\!\!\!\perp$ denotes "is not independent of." Informally, the head's value can be inferred from the tail's values when the condition holds. The RDR does not specify the functional form of the dependency: that is the responsibility of the inference mechanism that consumes it. What the RDR declares is the existence of an inference channel between named cells.

Example 4.2 (The salary-band FD as an RDR)**.** Recall dependency D1 from Example 1.1: $(\mathsf{Team}, \mathsf{Title}) \Rightarrow \mathsf{SalaryBand}$. Two employees with the same team and title are paid in the

same salary band. We express D1 as an RDR:

Dependence: $\mathsf{SalaryBand}(\boldsymbol{X}) \not\perp\!\!\!\perp \mathsf{SalaryBand}(\boldsymbol{Y})$

Condition:
```
SELECT X.rid, Y.rid FROM Employee X, Employee Y
WHERE X.Team = Y.Team AND X.Title = Y.Title
AND X.rid <> Y.rid
```

The dependence states that the salary band of any record $\boldsymbol{X}$ is not independent of the salary band of any record $\boldsymbol{Y}$ that the condition pairs with $\boldsymbol{X}$. The condition pairs records by matching team and title. Knowing $\boldsymbol{Y}$'s salary band, together with the fact that the condition is satisfied, enables inference of $\boldsymbol{X}$'s salary band.

Three further RDRs capture the remaining dependencies in our running example.

Example 4.3 (D2 and D3 as RDRs)**.** Dependency D2 from Example 1.1, $\mathsf{Level} \Rightarrow \mathsf{SalaryBand}$, is expressed as

Dependence: $\mathsf{SalaryBand}(\boldsymbol{X}) \not\perp\!\!\!\perp \mathsf{SalaryBand}(\boldsymbol{Y})$

Condition:
```
SELECT X.rid, Y.rid FROM Employee X, Employee Y
WHERE X.Level = Y.Level AND X.rid <> Y.rid
```

Dependency D3, $\mathsf{Manager} \Rightarrow \mathsf{Level}$, is expressed as

Dependence: $\mathsf{Level}(\boldsymbol{X}) \not\perp\!\!\!\perp \mathsf{Level}(\boldsymbol{Y})$

Condition:
```
SELECT X.rid, Y.rid FROM Employee X, Employee Y
WHERE X.Manager = Y.Manager AND X.rid <> Y.rid
```

We refer to these three rules collectively as $\{r_1, r_2, r_3\}$, corresponding to D1, D2, and D3

respectively.

Example 4.4 (A per-row strengthening of D2 as an RDR)**.** In many organizations, the level-to-band mapping is supplied as a fixed lookup table: knowing an employee's level alone determines that employee's band. This may be expressed as follows:

Dependence: $\mathsf{SalaryBand}(\boldsymbol{X}) \not\perp\!\!\!\perp \mathsf{Level}(\boldsymbol{X})$

Condition: `SELECT X.rid FROM Employee X`

Each instantiation of r_4 pairs an employee's salary band with that same employee's level: $\delta_{4,i} : \mathsf{SalaryBand}(e_i) \not\perp\!\!\!\perp \mathsf{Level}(e_i)$, with cells $\{\mathsf{SalaryBand}(e_i), \mathsf{Level}(e_i)\}$. Rule r_4 implies D2 (two employees at the same level necessarily share a salary band), but is strictly stronger: it relates an employee's level directly to that employee's band without invoking a second employee. We include r_4 in Δ so that the inference framework of Section 4.4 can exhibit a transitive chain through the bridge cell $\mathsf{Level}(e_5)$.

The running example uses the set $\Delta = \{r_1, r_2, r_3, r_4\}$ of dependencies.

4.2.1 Subsumption of Standard Dependency Formalisms

RDRs can express data dependencies arising from the main schema-level dependency formalisms used in the database literature and extend them with two shapes that those formalisms cannot express.

Functional dependencies. A functional dependency $\mathsf{X}_1, \ldots, \mathsf{X}_k \to \mathsf{A}$ states that, in any instance of the schema, any two records that agree on the attributes $\mathsf{X}_1, \ldots, \mathsf{X}_k$ also agree on A. The RDR encoding mirrors Example 4.2: the dependence is $\mathsf{A}(\boldsymbol{X}) \not\perp\!\!\!\perp \mathsf{A}(\boldsymbol{Y})$, and the condition is an equi-join of the relation with itself on the determinant attributes. The Employee FDs D1, D2, and D3 are instances of this pattern.

Denial constraints. A denial constraint forbids the simultaneous truth of a conjunction of cell predicates. For example, the constraint "no two employees with the same manager differ by more than one level" is a denial constraint. RDRs encode such constraints by making the forbidden conjunction the condition Q of the rule and the affected cell the head. When the condition activates on a tuple of rids, the head's value is constrained to a subset of its domain that avoids the forbidden conjunction. The RDR encoding of denial constraints follows the framework of [29, 73].

Conditional functional dependencies. A conditional functional dependency (CFD) [42] is a functional dependency that holds only on the subset of records that satisfy a condition. We consider a conditional version of D2: the FD $\mathsf{Level} \to \mathsf{SalaryBand}$ holds only on records with $\mathsf{Title} = \texttt{SWE}$. This captures a situation in which the leveling ladder for software engineers maps cleanly to compensation bands, but other engineering tracks (such as site reliability) follow different mappings. Expressed as an RDR:

Dependence: $\mathsf{SalaryBand}(\boldsymbol{X}) \not\perp\!\!\!\perp \mathsf{SalaryBand}(\boldsymbol{Y})$

Condition:
```
SELECT X.rid, Y.rid FROM Employee X, Employee Y
WHERE X.Level = Y.Level AND X.rid <> Y.rid
AND X.Title = 'SWE' AND Y.Title = 'SWE'
```

Here, the title predicate narrows the rule's scope: on the Employee instance, this CFD pairs records among $\{e_1, e_2, e_3, e_5\}$ but excludes e_4, whose title SRE falls outside the scope. The unconditional D2, by contrast, would pair e_4 with the L4 SWE records and assert that all four share a salary band.

Aggregate dependencies. Aggregate dependencies are constraints on aggregations of cell values, such as sums, averages, or counts. They are common in organizational databases. For example, suppose the Ads team has a fixed salary-band budget Σ_{Ads} recorded in a separate

TeamBudget table. The constraint is that the sum of the amounts for each band for Ads employees equals Σ_{Ads}. Knowing all but one of the Ads salary bands and the budget allows the inference of the missing band. As an RDR:

Dependence: $\mathsf{SalaryBand}(\boldsymbol{X}) \not\perp\!\!\!\perp \mathsf{SalaryBand}(\boldsymbol{Y}_1), \ldots, \mathsf{SalaryBand}(\boldsymbol{Y}_{k-1}), \mathsf{Budget}(\boldsymbol{B})$

Condition:

```
SELECT X.rid, Y1.rid, ..., Yk1.rid, B.rid
FROM Employee X, Employee Y1, ..., Yk1,
    TeamBudget B
WHERE X.Team = Yi.Team = B.Team = 'Ads'
AND X.rid, Y1.rid, ..., Yk1.rid pairwise distinct
```

Aggregate dependencies cannot be expressed within the FD or DC formalisms because both quantify over pairs of records, not over groups of arbitrary cardinality.

Cell-level dependencies. A cell-level dependency pins the value of a single cell from a one-cell premise, typically supplied by an external source such as an audit log. Suppose an HR log entry records that e_3 was promoted to level L5 at time t. The log entry constrains the value of $\mathsf{Level}(e_3)$ regardless of any other database content. As an RDR:

Dependence: $\mathsf{Level}(\boldsymbol{X}) \not\perp\!\!\!\perp \mathsf{NewValue}(\boldsymbol{L})$

Condition:

```
SELECT X.rid, L.rid FROM Employee X, HRLog L
WHERE L.target = X.rid AND L.field = 'Level'
```

The condition matches an HR log entry to the employee record it targets. Cell-level dependencies are not expressible in the FD or DC formalisms because their scope is a single named cell rather than a class of records.

SQL Grounding. Every RDR is grounded by its condition Q, which is an executable SQL query over the schema $\boldsymbol{S}$. Table 4.2 summarizes the SQL shape of the condition for each of the dependency formalisms above. The grounding makes RDRs declarable in any system with a SQL interface and verifiable by running the condition against the current instance.

Shape	**Condition Q form**	**Example fragment**
Functional dep. ($\mathsf{X} \to \mathsf{A}$)	Self-join on X	`X.Team=Y.Team AND X.Title=Y.Title`
Denial constraint	Self-join with a forbidden predicate	`X.Manager=Y.Manager AND |X.Level-Y.Level|>1`
Conditional FD	FD self-join plus a side filter	FD condition `AND X.Location='NYC'`
Aggregate dep.	Multi-way join over a group plus an external aggregate carrier	Join all Ads employees with TeamBudget
Cell-level dep.	Equi-join with an external record (e.g., log)	`L.target=X.rid AND L.field='Level'`

Table 4.2: SQL shape of the condition Q for each dependency expressible as an RDR.

4.2.2 Instantiated RDRs

An RDR as defined in Definition 4.1 is a schema-level object: it names attributes and contains a query over rids. To reason about specific cells, we need its instantiation on a database state.

Definition 4.2 (Instantiated RDR)**.** *Let r be an RDR given by Equation (4.1), and let I be an instance of $\boldsymbol{S}$. An* instantiated RDR *of r on I is an assignment of the rid variables $\boldsymbol{X}, \boldsymbol{X}_1, \ldots, \boldsymbol{X}_p$ to rids $\boldsymbol{x}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_p$ such that the tuple $(\boldsymbol{x}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_p)$ is returned by $Q(I)$. We denote an instantiated RDR by*

$$\delta : \; A(\boldsymbol{x}) \;\not\perp\!\!\!\perp\; A_1(\boldsymbol{x}_1), \; \ldots \; , A_p(\boldsymbol{x}_p). \tag{4.2}$$

We write $\mathit{Head}(\delta) = \{A(\boldsymbol{x})\}$ for the head of an instantiated RDR, $\mathit{Tail}(\delta) = \{A_1(\boldsymbol{x}_1), \ldots, A_p(\boldsymbol{x}_p)\}$ for its tail, and $\mathit{Cells}(\delta) = \mathit{Head}(\delta) \cup \mathit{Tail}(\delta)$ for the cells it names.

Example 4.5 (Instantiations of D1)**.** The RDR r_1 from Example 4.2 encodes dependency

D1. Its condition selects pairs of records that agree on team and title. On the Employee instance, records e_1, e_2, e_3, e_5 all share $\mathsf{Team} = \texttt{Ads}$ and $\mathsf{Title} = \texttt{SWE}$, so each ordered pair drawn from this group yields an instantiation. Two instantiations include the target cell $c^* = \mathsf{SalaryBand}(e_5)$. They are:

$$\delta_{1,a} : \mathsf{SalaryBand}(e_5) \not\perp\!\!\!\perp \mathsf{SalaryBand}(e_1),$$

$$\delta_{1,b} : \mathsf{SalaryBand}(e_5) \not\perp\!\!\!\perp \mathsf{SalaryBand}(e_2).$$

Both have as their head $\{c^*\}$ and a tail comprising one cell. Record e_4 does not satisfy the condition with e_5, since e_4 belongs to a different team, and therefore contributes no instantiation involving c^*.

4.2.3 The Semantic Model

We collect the RDRs governing a schema in a set that defines the database's semantic model.

Definition 4.3 (RDR set and instantiations)**.** *Given a database instance $\mathcal{D}$, the set Δ of RDRs, the set of all instantiations of RDRs in Δ on $\mathcal{D}$ is denoted as $\Delta(\mathcal{D})$.*

The set Δ is the *semantic model* of the database: it specifies which cells carry information about which other cells and through which conditions. In the chapters that follow, Δ plays the same role as $\mathcal{M}$ in the conceptual sketch of Chapter 1.

A deletion mechanism may change which RDRs are instantiated. All inference after deletion is therefore evaluated on the residual instance $\mathcal{D}'$, using $\Delta(\mathcal{D}')$ rather than the set $\Delta(\mathcal{D})$ of instantiations. Since RDR conditions are SQL queries, if a cell used by a condition is set to `NULL`, the corresponding SQL predicate may no longer match, and the instantiated RDR may not be in $\Delta(\mathcal{D}')$. Cells appearing in a condition determine whether an instantiation exists; cells appearing in the dependence statement determine which values participate in the inference channel.

For the running example, the set of RDRs is $\Delta = \{r_1, r_2, r_3, r_4\}$, where r_1, r_2, r_3 encode D1, D2, D3 from Chapter 1, and r_4 is the per-row strengthening introduced in Example 4.4.

Integrity constraints, consistency, and RDRs. We distinguish two related notions in the dependency literature.

Integrity constraints are schema-level rules declared on the database: primary keys, foreign keys, functional dependencies, denial constraints, `CHECK` predicates, and so on. The DBMS enforces them at commit time, and any committed instance *satisfies* them. We assume throughout this thesis that the instance the deletion mechanism receives is consistent in this sense, and that the residual produced by the mechanism is consistent as well. Because we delete by assigning `NULL` and most integrity constraints either tolerate `NULL` or are vacuously satisfied on `NULL`-valued cells, this consistency is preserved naturally.

RDRs in our framework are different: they express our inference channels, not constraints. Thus, there is no notion of the database "satisfying" an RDR. An RDR is a declaration of the form "if the cells named in the tail are known and the condition matches, the head can be inferred," and it serves only to provide inputs for the inference framework of Section 4.4.

The two are, however, not unrelated. An integrity constraint that the DB satisfies provides an obvious source of an RDR: its dependence statement and condition transcribe directly into RDR form, and the resulting RDR's instantiations are guaranteed to agree on any derived value, because the underlying constraint holds. RDRs may also be derived from regulatory annotations, ownership rules, statistical analyses, or domain knowledge that has no corresponding integrity constraint and for which the instance is under no obligation to be consistent. Our framework does not distinguish between RDRs based on how they were derived.

Example 4.6 (Integrity constraints and RDRs in the Employee instance)**.** In our running

example, dependencies D1, $(\mathsf{Team}, \mathsf{Title}) \to \mathsf{SalaryBand}$, and D2, $\mathsf{Level} \to \mathsf{SalaryBand}$, are integrity constraints: the instance satisfies them, and the residual should too. They are also declared as RDRs (r_1, r_2) so that the inference framework reasons about the channels they expose. Dependency D3, $\mathsf{Manager} \to \mathsf{Level}$, is, however, not an integrity constraint: the instance is permitted to deviate from it, and indeed does so for e_3 (manager $\mathtt{m}_{10}$, level `L5`, while e_1, e_2, e_5 are at `L4`). D3 is declared as RDR (r_3), using which an adversary may exploit the approximate regularity it describes.

RDRs may be declared manually, derived from regulatory annotations, or discovered automatically. Recent work [77, 91] shows that automatically discovered constraints can be unreliable: in particular, denial constraint discovery on real datasets can produce a substantial fraction of false positives. When the declared Δ over-specifies the true dependency structure, inference-aware deletion is conservative (it over-protects, at the cost of additional auxiliary deletions). When Δ under-specifies, inference-aware deletion is unsound (an undeclared rule may permit recovery of the deleted cell).

4.3 Dependency Hypergraphs

The set Δ of RDRs specifies dependencies at the schema level; the instantiated set $\Delta(\mathcal{D})$ grounds them to cells from a given instance. To reason about how inference propagates among cells, we represent $\Delta(\mathcal{D})$ as a *dependency hypergraph*. This section defines the hypergraph and identifies the operations on it that the rest of the thesis uses.

Definition 4.4 (Dependency hypergraph)**.** *Let $\mathcal{D}$ be an instance over $\boldsymbol{S}$ and let Δ be an RDR set over $\boldsymbol{S}$. The* dependency hypergraph *induced by $(\mathcal{D}, \Delta)$ is the undirected hypergraph $H(\mathcal{D}, \Delta) = (N, E)$ where $N = \bigcup_{\delta \in \Delta(\mathcal{D})} \mathit{Cells}(\delta)$, and $E = \big\{ \mathit{Cells}(\delta) \ : \ \delta \in \Delta(\mathcal{D}) \big\}$. Each* hyperedge *$e \in E$ is a set of cells, namely $\mathit{Cells}(\delta) = \mathit{Head}(\delta) \cup \mathit{Tail}(\delta)$ for the instantiation δ that induces it. We identify hyperedges with their inducing instantiations and write e_δ for the*

hyperedge induced by δ when the distinction matters.

Example 4.7 (Employee hypergraph)**.** Figure 4.1 depicts the fragment of $H(\mathcal{D}, \Delta)$ relevant for the target cell $c^* = \mathsf{SalaryBand}(e_5)$ for the set $\Delta = \{r_1, r_2, r_3, r_4\}$ of RDRs. The figure shows the cells and hyperedges around the target. There are seven edges that are incident on c^*: three from r_1 (matching team and title), three from r_2 (matching level), and one edge $\{c^*, \mathsf{Level}(e_5)\}$ from r_4 that bridges salary and level for the same employee. Three edges are incident on $\mathsf{Level}(e_5)$ from r_3 (matching manager). Each hyperedge is a set of cells, drawn as an undirected line between them; rules are color-coded.

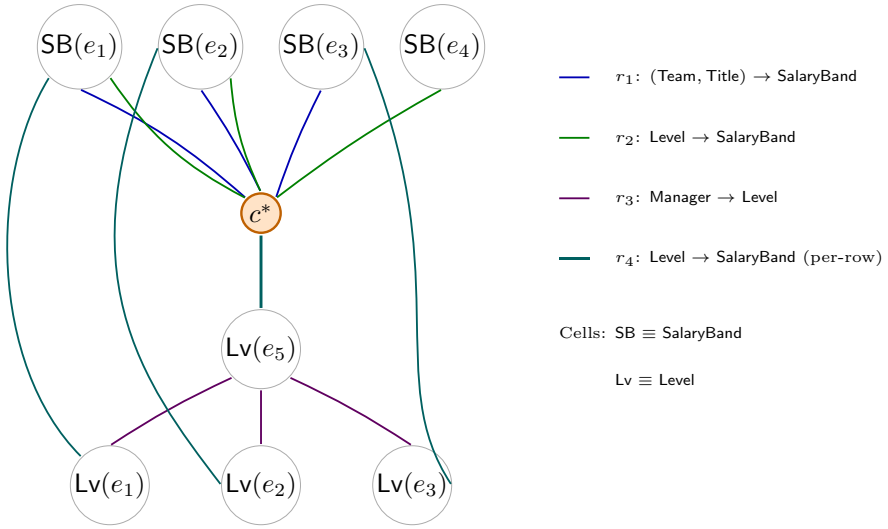


Figure 4.1: Fragment of the dependency hypergraph $H(\mathcal{D}, \Delta)$ on the Employee instance, centered on c^*.

The head and tail of an instantiated RDR are specification-level distinctions: they reflect how the dependence statement was written. From the perspective of inference, we assume that dependencies are *symmetric*, i.e., the head may lead to the inference of all the attributes in the tail and the attributes in the tail together infer the head. Operationally, an instantiated RDR with n cells says that, given any $n-1$ of those values, the nth is determined (or, in a probabilistic generalization, predicted with elevated confidence). We therefore drop the head-tail distinction at the hypergraph level and treat each hyperedge as a set. The head-tail labels remain attached to each δ for specification convenience.

The hypergraph $H(\mathcal{D}, \Delta)$ captures, in a single picture, every direct dependency that the semantics model declares to hold on the current instance. Cells that do not appear in an instantiation of any rule are not in the set N of vertices; they are inferentially isolated under the set Δ of dependencies.

For a cell $c \in N$, we associate with c the set of hyperedges that contain it. Let $c \in N$. The set of *incident edges* of c is given by $E(c) = \{e \in E : c \in e\}$.

In Figure 4.1, the incident set $E(c^*)$ contains the seven edges incident into c^*, and the incident set $E(\mathsf{Level}(e_5))$ contains the three edges drawn into $\mathsf{Level}(e_5)$. Each incident edge of a cell supports inference of that cell when the other $|e| - 1$ cells of e are known. The incident-edge set is the natural unit of analysis for *blocking*: a mechanism that wants to prevent the re-inference of c must invalidate every $e \in E(c)$ by removing at least one cell in each e.

Blocking and residual re-instantiation. A hyperedge e supports inference of a cell $c \in e$ only when every other cell of e is not `NULL`. Setting any one of those other cells to `NULL` *blocks* e as an inference channel. Under the convention above, setting a condition cell to `NULL` may entirely remove the channel since the condition that produced the instantiation may no longer match on the residual instance.

Definition 4.5 (Blocked edge)**.** *Let $\mathcal{T} \subseteq Cells(\mathcal{D})$ be a set of cells whose values are set to* `NULL`*, and let $\mathcal{D}' = \mathcal{D}[\mathcal{T} \leftarrow$ `NULL`$]$. A hyperedge $e \in E(H(\mathcal{D}, \Delta))$ is* blocked *by $\mathcal{T}$ with respect to a cell $c \in e$ if at least one other cell of e is in $\mathcal{T}$, that is, if $(e \setminus \{c\}) \cap \mathcal{T} \neq \emptyset$.*

The *residual hypergraph* after erasing $\mathcal{T}$ is $H(\mathcal{D}', \Delta)$. It is obtained by re-instantiating the RDR set on the residual instance. Thus, an edge may remain present but be blocked because one of its premise cells is `NULL`; it may also be entirely absent when a `NULL` value in the condition prevents the SQL query from returning the corresponding rid tuple.

Example 4.8 (Value blocking and condition blocking)**.** In Figure 4.1, erasing $\mathcal{T} = \{\mathsf{SalaryBand}(e_1)\}$

value-blocks the r_1 edge $\{c^*, \mathsf{SalaryBand}(e_1)\}$ and the r_2 edge $\{c^*, \mathsf{SalaryBand}(e_1)\}$ with respect to c^*. By contrast, altogether erasing $\mathsf{Team}(e_5)$ may remove the r_1 instantiations involving e_5 because the condition `X.Team = Y.Team` is no longer satisfied under SQL's treatment of `NULL`. The residual hypergraph $H(\mathcal{D}', \Delta)$ captures both.

Example 4.9 (Employee hypergraph instantiations)**.** We now illustrate the instantiations of $\Delta = \{r_1, r_2, r_3, r_4\}$ that produce the hypergraph of Figure 4.1.

Instantiations of r_1 (D1, (Team, Title) → SalaryBand). The condition of r_1 selects ordered pairs of distinct records that agree on team and title. On the instance, four records carry $(\mathsf{Team}, \mathsf{Title}) = (\texttt{Ads}, \texttt{SWE})$: e_1, e_2, e_3, e_5. Among the instantiations whose cells include c^*, three are relevant:

$$\delta_{1,a} : c^* \not\perp\!\!\!\perp \mathsf{SalaryBand}(e_1), \quad \delta_{1,b} : c^* \not\perp\!\!\!\perp \mathsf{SalaryBand}(e_2), \quad \delta_{1,c} : c^* \not\perp\!\!\!\perp \mathsf{SalaryBand}(e_3).$$

Each contributes one two-cell hyperedge to $E(c^*)$. The hyperedges are undirected: each connects c^* and another SalaryBand cell, and either cell can be inferred from the other given, the rule.

Instantiations of r_2 (D2, Level → SalaryBand). The condition of r_2 selects pairs of distinct records at the same level. Four records sit at $\mathsf{Level} = \texttt{L4}$: e_1, e_2, e_4, e_5. Three instantiations involving c^* are

$$\delta_{2,a} : c^* \not\perp\!\!\!\perp \mathsf{SalaryBand}(e_1), \quad \delta_{2,b} : c^* \not\perp\!\!\!\perp \mathsf{SalaryBand}(e_2), \quad \delta_{2,c} : c^* \not\perp\!\!\!\perp \mathsf{SalaryBand}(e_4).$$

Instantiations of r_3 (D3, Manager → Level). The condition of r_3 selects pairs of distinct records sharing a manager. Four records report to m_{10}: e_1, e_2, e_3, e_5. The cells named by r_3's dependence statement are levels, so instantiations of r_3 have $\mathsf{Level}(e_5)$ as one cell of the

hyperedge whenever the other rid is one of the other m_{10} reports. We list three instantiations:

$$\delta_{3,a} : \mathsf{Level}(e_5) \not\perp\!\!\!\perp \mathsf{Level}(e_1), \quad \delta_{3,b} : \mathsf{Level}(e_5) \not\perp\!\!\!\perp \mathsf{Level}(e_2), \quad \delta_{3,c} : \mathsf{Level}(e_5) \not\perp\!\!\!\perp \mathsf{Level}(e_3).$$

As noted in Section 4.2, the records e_1, e_2, e_5 at `L4` and e_3 at `L5` are all under the same manager. The hypergraph contains all three edges by Definition 4.4: the hypergraph is determined by the rules' conditions. Two of the three edges, when used for inference, would derive $\mathsf{Level}(e_5) = \texttt{L4}$; the third would derive `L5`.

Instantiations of r_4 (per-row $\mathsf{Level} \to \mathsf{SalaryBand}$). The condition of r_4 is a trivial selection: for each rid e_i, the query returns e_i and the instantiation $\delta_{4,i}$ has cells $\{\mathsf{SalaryBand}(e_i), \mathsf{Level}(e_i)\}$. The instantiation most relevant to the target is $\delta_{4,5} : c^* \not\perp\!\!\!\perp \mathsf{Level}(e_5)$, the r_4 edge between the target salary band and the level of the same employee. This edge is the bridge that the chase of Section 4.4.2 traverses, after a chain of r_3 edges has derived $\mathsf{Level}(e_5)$. The other instantiations $\delta_{4,1}, \delta_{4,2}, \delta_{4,3}, \delta_{4,4}$ link the salary band and level of each remaining employee.

The transitive path. The cell $\mathsf{Level}(e_5)$ does not appear in $\mathit{Cells}(\delta_{2,a})$, $\mathit{Cells}(\delta_{2,b})$, or $\mathit{Cells}(\delta_{2,c})$. The condition of r_2 uses Level values to identify which record pairs satisfy the rule, but only attributes that appear in the dependence statement contribute to $\mathit{Cells}(\delta)$, and the dependence statement of r_2 names only $\mathsf{SalaryBand}$. The cells appearing in a condition help *determine* which instantiations exist; they are not part of the instantiated RDR's cells.

Consequently, the transitive path $\mathsf{Level}(e_5) \rightsquigarrow c^*$ is not directly visible inside any single hyperedge of H. It emerges from the closure procedure of Section 4.4, which composes hyperedges across rules through shared cells. The hypergraph itself records only the direct dependencies named by each rule.

Size of the Hypergraph. The size of $H(\mathcal{D}, \Delta)$ depends on the shape of the rules in Δ and on the cardinality of the instance.

- **FD-shaped rules.** An FD-shaped RDR whose condition is a self-join on the determinant attributes produces, in the worst case, $O(n^2)$ instantiations on an instance with n records, since every ordered pair satisfying the condition contributes one hyperedge.

- **CFD-shaped rules.** A CFD restricts the FD's self-join to a subset of records and so produces at most as many instantiations as the unconditional FD.

- **Aggregate rules.** An aggregate rule whose tail names every cell in a group of size k produces $O(n^k)$ instantiations in the worst case, with k scaling with group cardinality. In practice, aggregate rules are typically instantiated once per group, with the group fixed by the condition.

- **Cell-level rules.** A cell-level rule whose condition matches a single external record (such as a log entry) produces $O(n)$ instantiations across the instance: one per record matched by the log.

The polynomial bounds above are worst-case. For the running example, with $|\mathcal{D}| = 5$ and three FD-shaped rules, the hypergraph contains only a handful of edges incident on c^*.

4.4 Inference Framework

The dependency hypergraph $H(I, \Delta)$ records, for any instance I, the direct inference channels declared by the RDR set Δ on that instance. To reason about whether the value of a specific cell can be re-inferred from the values of others, we need an inference procedure that operates on the hypergraph. This section develops that procedure: we begin with a single-edge derivation (Section 4.4.1), iterate it to obtain a closure that captures multi-step inference (Section 4.4.2), and classify cells by the outcome of closure as either inferable or blocked (Section 4.4.3). Section 4.4.4 states the inverse problem of choosing an auxiliary set unless this should be "that blocks a given target cell.

rid	Te	Ti	Lv	Mg	Lo	SB
e_1	Ads	SWE	L4	m_{10}	NYC	B4
e_2	Ads	SWE	L4	m_{10}	NYC	B4
e_3	Ads	SWE	L5	m_{10}	NYC	B5
e_4	Infra	SRE	L4	m_{12}	SEA	B4
e_5	Ads	SWE	L4	m_{10}	NYC	c^*

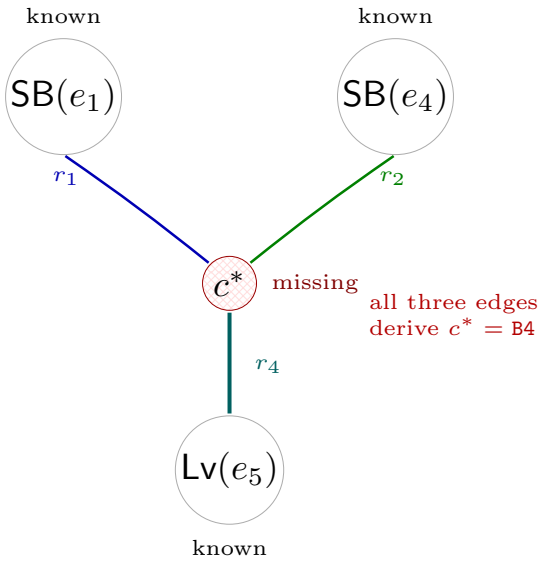


Figure 4.2: Direct inference into c^* on the Employee instance.

We emphasize that the inference framework reasons about derivations of cell values from other cell values via the RDRs in Δ. It is independent of the question of whether the instance is consistent with the schema's integrity constraints, which we have assumed throughout (Section 4.2).

4.4.1 Direct Inference

Consider the Employee instance with $c^* = \mathsf{SalaryBand}(e_5)$ missing and every other cell visible. The hyperedge $\delta_{1,a} : c^* \not\perp\!\!\!\perp \mathsf{SalaryBand}(e_1)$ in the hypergraph contains two cells: only one of them ($\mathsf{SalaryBand}(e_1)$) is known. Rule r_1, the FD $(\mathsf{Team}, \mathsf{Title}) \to \mathsf{SalaryBand}$, states that two employees with the same team and title share a salary band. Knowing the value $\mathit{Val}(\mathsf{SalaryBand}(e_1), \mathcal{D}) = \mathtt{B4}$, together with the fact that the pair (e_5, e_1) satisfies the condition of r_1, lets us recover $c^* = \mathtt{B4}$. Figure 4.2 visualizes this derivation.

We generalize this observation. An instantiated RDR with n cells states that any $n-1$ of those values determine the nth (the operational reading of the undirected interpretation in Section 4.3). If δ has all of its cells in the known set except for one, that remaining cell can be inferred. We formalize this below.

Definition 4.6 (Direct inference)**.** *Let I be an instance, let $K \subseteq \mathit{Cells}(I)$ be a set of known cells, let $\delta \in \Delta(I)$, and let $c \in \mathit{Cells}(\delta)$. We say that δ* supports *a direct inference of c from K if $\mathit{Cells}(\delta) \setminus \{c\} \subseteq K$.*

When direct inference holds, δ yields a *candidate value* for c, $v_{\delta,c} = f_{\delta,c}\big(\widehat{\mathit{Val}}_K(c') : c' \in \mathit{Cells}(\delta) \setminus \{c\}\big)$, where $\widehat{\mathit{Val}}_K(c')$ denotes the visible or previously derived value associated with the known cell c'. The function $f_{\delta,c}$ is determined by the shape of the originating RDR and by the role of c within δ. We write $\delta : K \vdash c = v_{\delta,c}$ for the derivation step.

The inference function. For an FD-shaped RDR whose dependence is $\mathsf{A}(\boldsymbol{x}) \not\perp\!\!\!\perp \mathsf{A}(\boldsymbol{y})$, $f_{\delta,c}$ is the equality $f_{\delta,c} = \widehat{\mathit{Val}}_K(c')$, where c' is the unique cell of δ other than c; the inference is symmetric, so either cell may be the inferred one. For an aggregate RDR whose dependence relates one cell to the sum of others, $f_{\delta,c}$ is the residual of the aggregate after subtracting the values of the other cells. The framework does not fix $f_{\delta,c}$ beyond what the dependence statement implies; the deletion mechanisms in later chapters reason about reachability through the hypergraph rather than about specific derived values.

Conjunctive within an edge, disjunctive across edges. The condition $\mathit{Cells}(\delta) \setminus \{c\} \subseteq K$ is conjunctive: *every* other cell of δ must be in K for δ to derive c. If $\delta = \{c, a, b\}$, both a and b must be known; knowing only a does not suffice. When a cell c has several incident edges, by contrast, the relationship across edges is disjunctive: c is derivable as long as *any one* incident edge has all of its other cells in K. Different edges may use different sets of premise cells, and the framework treats them as independent alternatives. Blocking a cell therefore requires invalidating *every* incident edge, the dual of the disjunctive recovery condition.

Example 4.10 (The Employee instance, multiple edges agree)**.** Continuing the opening observation, let $K = \mathit{Cells}(\mathcal{D}) \setminus \{c^*\}$. The edge $\delta_{1,a}$ supports a direct derivation, as shown. So do $\delta_{1,b}$ and $\delta_{1,c}$ from r_1, and the three r_2 edges containing c^*. Every one of these edges, when applied as a direct derivation under K, yields `B4`. The agreement is not a coincidence:

rid	Te	Ti	Lv	Mg	Lo	SB
e_1	Ads	SWE	L4	m_{10}	NYC	NULL
e_2	Ads	SWE	L4	m_{10}	NYC	NULL
e_3	Ads	SWE	L5	m_{10}	NYC	NULL
e_4	Infra	SRE	L4	m_{12}	SEA	NULL
e_5	Ads	SWE	NULL	m_{10}	NYC	c^*

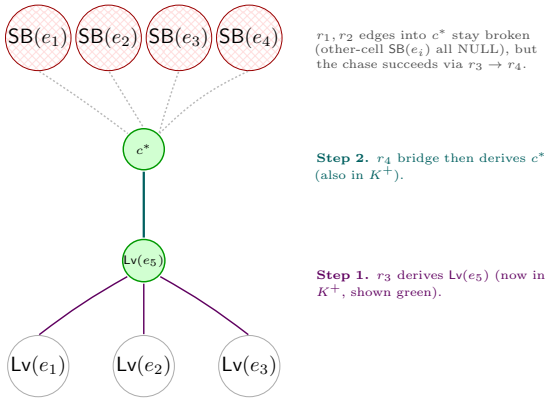


Figure 4.3: Closure iteration (the chase) recovering c^* on the Employee instance.

D1 and D2 are integrity constraints satisfied by the instance, so the inferences they enable on a consistent instance must agree.

4.4.2 Inference Closure

Direct inference uses a single edge. But a cell derived by one edge could become an input to another, enabling a *chain*. Consider the Employee instance with the target cell c^*, the four salary bands $\mathsf{SB}(e_1), \ldots, \mathsf{SB}(e_4)$, and the level $\mathsf{Lv}(e_5)$ all set to NULL. The direct r_1 and r_4 paths into c^* are value-blocked, and the r_2 paths may also disappear because $\mathsf{Lv}(e_5)$ appears in the SQL condition for matching levels. Yet inference on c^* is not blocked. The cell $\mathsf{Lv}(e_5)$ is itself in the cells of three r_3 edges whose other cells $\mathsf{Lv}(e_1), \mathsf{Lv}(e_2), \mathsf{Lv}(e_3)$ are visible: any one of these edges supports a derivation of $\mathsf{Lv}(e_5)$. Once $\mathsf{Lv}(e_5)$ is known, the edge r_4 given by $\{c^*, \mathsf{Lv}(e_5)\}$ has its other cell known, and a second derivation gives c^*. Two hyperedges in sequence, joined at the cell $\mathsf{Lv}(e_5)$ gives us the chase depicted in Figure 4.3.

The closure procedure formalizes the iteration. We start from the known cells and repeatedly grow the set by any cell derivable from cells already in it.

Definition 4.7 (Inference closure)**.** *Let I be an instance and let $K \subseteq Cells(I)$. The* inference closure *of K under Δ on I, written $K^+_\Delta(I)$ or simply K^+, is the smallest set $K^+ \subseteq Cells(I)$ satisfying:*

Algorithm 1 Inference closure K^+ of a known set K under Δ on an instance I.

Input: Instance I, RDR set Δ, known set $K \subseteq \mathit{Cells}(I)$
Output: Closure K^+

$K^+ \leftarrow K$
repeat
  $\mathit{changed} \leftarrow$ FALSE
  for all $\delta \in \Delta(I)$ and $c \in \mathit{Cells}(\delta)$ with $c \notin K^+$ and $\mathit{Cells}(\delta) \setminus \{c\} \subseteq K^+$ **do**
    $K^+ \leftarrow K^+ \cup \{c\}$
    $\mathit{changed} \leftarrow$ TRUE
until $\mathit{changed} =$ FALSE
return K^+

1. $K \subseteq K^+$, *and*

2. *if* $\delta \in \Delta(I)$ *and* $c \in \mathit{Cells}(\delta)$ *are such that* $\mathit{Cells}(\delta) \setminus \{c\} \subseteq K^+$, *then* $c \in K^+$.

The closure is well-defined and unique: $\mathit{Cells}(I)$ is finite, the rule set is finite, and adding a cell to K^+ never removes one already present. Algorithm 1 computes K^+ by the obvious fixed-point procedure and terminates in at most $|\mathit{Cells}(I)|$ iterations of the outer loop. Blocking a cell amounts to removing it from K by setting its value to `NULL`: the edges that contained it can no longer be used, and the chase cannot reach the cells whose only inference paths run through it.

Direct and indirect inference. The closure unifies the direct and indirect inference of our work in [27]. A direct inference of c is captured by a single application of the inner step. An indirect inference is captured by a chain of applications: an earlier derivation adds a cell c_1 to K^+, which then enables another instantiation δ_2 to derive c_2, and so on. Two instantiations participate in such a chain whenever they share a cell, because the shared cell is the bridge through which inference propagates from one edge to the next.

The closure tracks *which* cells become derivable, not *which value* each derived cell takes. Values are computed alongside, by the corresponding $f_{\delta,c}$ functions, when needed. When two supported instantiations derive different candidate values for the same cell, the cell is

still inferable in the reachability sense, but the value-level output is a set or distribution of candidates rather than a single value. Later chapters account for this distinction through leakage criteria. The deletion mechanisms in later chapters operate primarily on reachability and use closure as their primary tool.

Example 4.11 (The chase on the Employee instance)**.** We complete the calculation begun in the opening observation. Take the auxiliary set

$$\mathcal{T} \;=\; \{\mathsf{SB}(e_1),\, \mathsf{SB}(e_2),\, \mathsf{SB}(e_3),\, \mathsf{SB}(e_4),\, \mathsf{Level}(e_5)\}$$

and the residual $\mathcal{D}' = \mathcal{D}[\mathcal{T} \cup \{c^*\} \leftarrow \texttt{NULL}]$. The starting known set is $K = \mathit{Cells}(\mathcal{D}') \setminus (\mathcal{T} \cup \{c^*\})$.

Step 0. Initial $K^+ = K$. The cells c^*, $\mathsf{SB}(e_i)$ for $i \in \{1, 2, 3, 4\}$, and $\mathsf{Level}(e_5)$ are missing; everything else, including $\mathsf{Manager}(e_5)$, $\mathsf{Title}(e_5)$, and the levels of e_1, e_2, e_3, is in K^+.

Step 1. We look for any instantiation δ and cell $c \notin K^+$ such that $\mathit{Cells}(\delta) \setminus \{c\} \subseteq K^+$. The r_3 edge $\delta_{3,a} : \mathsf{Level}(e_5) \not\perp\!\!\!\perp \mathsf{Level}(e_1)$ has cells $\{\mathsf{Level}(e_5), \mathsf{Level}(e_1)\}$. The other cell, $\mathsf{Level}(e_1)$, is in K^+, so $\delta_{3,a}$ supports a derivation of $\mathsf{Level}(e_5)$. Closure adds $\mathsf{Level}(e_5)$ to K^+. ($\delta_{3,b}$ would equally work; $\delta_{3,c}$ may yield a different candidate value because e_3 has level `L5`.)

Step 2. With $\mathsf{Level}(e_5)$ now in K^+, the r_4 bridge edge $\delta_{4,5} : c^* \not\perp\!\!\!\perp \mathsf{Level}(e_5)$ has its other cell in K^+. So $\delta_{4,5}$ supports a derivation of c^*. Closure adds c^* to K^+.

Step 3. The remaining missing cells $\mathsf{SB}(e_1), \ldots, \mathsf{SB}(e_4)$ are also derivable now: for each e_i, the r_4 edge $\delta_{4,i} : \mathsf{SB}(e_i) \not\perp\!\!\!\perp \mathsf{Level}(e_i)$ has its other cell in K^+. Closure adds them and terminates.

Conclusion. $c^* \in K^+$. Two distinct hyperedges, $\delta_{3,a}$ then $\delta_{4,5}$, neither of which would have derived c^* on its own from K, jointly recovered the target through their shared cell $\mathsf{Level}(e_5)$. This is the chase: the closure procedure traversed the residual hypergraph $H(\mathcal{D}', \Delta)$ along

rid	Te	Ti	Lv	Mg	Lo	SB
e_1	Ads	SWE	NULL	m_{10}	NYC	NULL
e_2	Ads	SWE	NULL	m_{10}	NYC	NULL
e_3	Ads	SWE	NULL	m_{10}	NYC	NULL
e_4	Infra	SRE	L4	m_{12}	SEA	NULL
e_5	Ads	SWE	NULL	m_{10}	NYC	c^*

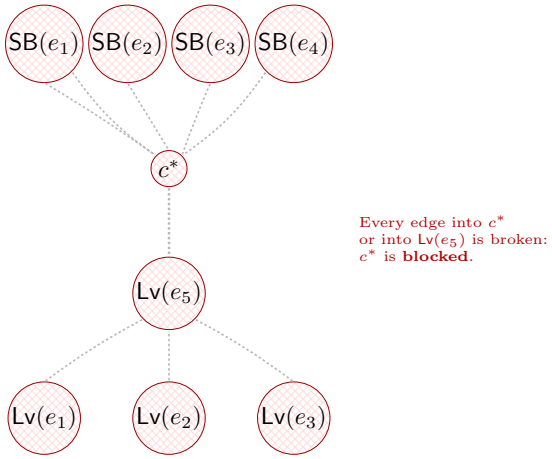


Figure 4.4: A blocked scenario for c^* on the Employee instance.

the path Level(e_1) - Level(e_5) - c^*, using r_3 then r_4.

4.4.3 Inferable and Blocked Cells

Closure has two possible outcomes for any missing cell. Either the chase reaches it ($c \in K^+$), or the chase exhausts every available step without ever covering c ($c \notin K^+$).

Definition 4.8 (Inferable and blocked cells)**.** *Let I be an instance, let $K \subseteq Cells(I)$, and let K^+ be the closure of K under Δ on I. A cell $c \in Cells(I) \setminus K$ is* inferable *if $c \in K^+$ and* blocked *if $c \notin K^+$.*

The two categories partition the missing cells. Blocked is the only category that prevents the value of c^* from being re-inferred. A cell is blocked exactly when, for every $\delta \in \Delta(I)$ with $c \in Cells(\delta)$, at least one other cell of δ is missing from K^+.

Example 4.12 (Why blocking c^* requires a larger auxiliary set.)**.** The blocked scenario in Figure 4.4 masks eight auxiliary cells. The reason is structural: with r_4 in Δ, the bridge edge $\{c^*, \text{Level}(e_5)\}$ gives c^* a one-step recovery whenever Level(e_5) is reachable. To block c^*, we must therefore block Level(e_5) too, which in turn requires breaking the r_3 edges that name it. Together with breaking the direct r_1, r_2, and r_4 routes into c^*, the auxiliary set grows. The

minimum-cost choice of $\mathcal{T}$ is the optimization problem we state next.

4.4.4 The Blocking Problem

Closure determines, for a given residual database, whether the target cell is inferable or blocked. The inverse question is: given the original database instance $\mathcal{D}$, the target cell c^*, and the set Δ of RDRs, which auxiliary sets $\mathcal{T}$ produce a residual instance $\mathcal{D}'$ whose dependency hypergraph blocks the value of c^* from being re-inferred?

Definition 4.9 (Blocking auxiliary set)**.** *A set $\mathcal{T} \subseteq \mathit{Cells}(\mathcal{D}) \setminus \{c^*\}$ is a* blocking auxiliary set *for c^* if, after both c^* and the cells in $\mathcal{T}$ are set to* `NULL`*, the cell c^* is blocked in the resulting residual under Δ. Formally, let $\mathcal{D}' = \mathcal{D}[\mathcal{T} \cup \{c^*\} \leftarrow \texttt{NULL}]$ and $K = \mathit{Cells}(\mathcal{D}') \setminus (\mathcal{T} \cup \{c^*\})$. Then $\mathcal{T}$ is blocking if $c^* \notin K^+$ where K^+ denotes the closure of the starting set K under Δ on $\mathcal{D}'$.*

Many auxiliary sets may be blocking. In our running example, the set used in Figure 4.4 is an example. A trivially blocking choice is to set every cell of every employee other than e_5's rid to `NULL`: under the residual re-instantiation semantics, this leaves c^* with no path through the hypergraph. The interesting question is which *smaller* sets suffice, and at what cost.

Definition 4.10 (Minimum blocking problem)**.** *Let cost : $\mathit{Cells}(\mathcal{D}) \to \mathbb{R}_{\geq 0}$ be a non-negative cost function over cells. The* minimum blocking problem *asks for a blocking auxiliary set $\mathcal{T}^*$ such that $\sum_{c \in \mathcal{T}^*} \mathit{cost}(c)$ is minimized.*

We state the problem abstractly. Later chapters of the thesis consider this question under different settings and develop corresponding exact and approximate algorithms.

4.4.5 The Adversary

After a cell c^* and a set $\mathcal{T}$ of cells has been deleted from a database, an *adversary* $\mathcal{A}$ observes a single snapshot of the post-deletion database instance. The adversary is also aware of the

schema, the set of RDRs. It attempts to reinfer the value of the target cell c^* using the set of RDRs. We assume that the adversary does not remember (or has a copy) the database states prior to deletion, as this would enable it to trivially infer the target cell c^*.

Definition 4.11 (Adversary)**.** *For a deletion mechanism that produces a database instance D', an* adversary $\mathcal{A}$ *is a procedure that takes as input*

- *the residual instance $V = \mathcal{D}'$, including the values of all non-`NULL` cells;*
- *the deletion pattern $P = \mathcal{T} \cup \{c^*\}$, the set of cells whose values were set to `NULL` by the deletion mechanism;*
- *the semantics model $\mathcal{M} = \Delta$, i.e., the set of RDRs under which inference is reasoned;*

and outputs a probability distribution over the domain of c^*.

The guarantees of deletion formalized in this thesis are subject to a pre-declared set of dependencies through which an adversary reasons.

Chapter 5

Erasure under Deterministic Dependencies

5.1 Motivation: Deletion Across Time

Chapter 1 introduced the *semantic gap*: the region between storage-level deletion (the cell is no longer retrievable) and inference-level deletion (the value of the cell is no longer inferable). Closing the semantic gap requires *semantic rollback*: removing, masking, perturbing, or otherwise controlling the auxiliary information that supports inference about the target value. This chapter develops the first reference point in the thesis for closing the gap deterministically: pre-insertion post-erasure equivalence, or **P2E2**, originally introduced in our earlier work [27].

Informally, P2E2 states the following requirement: *after erasure, the inferences available about c^* should be no more than the inferences that were available before c^* was ever inserted.* Rolling the set of dependencies back to its pre-insertion shape is the deterministic counterpart of bounding belief on V, the residual visible state, in the conceptual framework of Section 1.3. This chapter formalizes the principle (Definitions 5.1 and 5.2), characterizes the inferences

the mechanism must address (Theorem 5.2), and develops three algorithms for executing the deletion at minimum cost.

Before developing the formal framework, we present an example that makes the intuition clear. Chapter 4 developed an inference framework over a single database state: the dependency hypergraph $H(\mathcal{D}, \Delta)$ records every inference channel, and the closure K^+ tells us which cells the channels reach. Now we introduce another dimension: *time*. A database is not a static snapshot. By the time an erasure request arrives for c^*, the instance may contain values that were not present when c^* was inserted, and those new values may have created new inference channels that did not exist before. Naively setting c^* to `NULL` leaves the new channels intact. An adversary recovers c^* through them, and the storage-level deletion fails to close the semantic gap. We illustrate the situation, and the rollback that P2E2 requires, on the Employee instance of Chapter 4 interpreted as a temporal trace.

Example 5.1 (A Temporal Walk through the Employee Instance.)**.** Suppose that the Employee table is observed over time. Records are inserted one at a time. Each record carries the cells the schema requires (a salary band, a level, a team, a title, a manager, and a location). The target cell is $c^* = \mathsf{SalaryBand}(\mathsf{e}_5)$.

Figure 5.1 traces the relevant sequence of events. Initially, only e_4 exists, i.e.,`(Infra, SRE, L4, B4)`; this is the state at time t_0. At time $t_b = \kappa(c^*)$, the record e_5 is inserted `(Ads, SWE, L4, manager` m_{10}`, NYC,` $c^* =$ `B4)`. At this moment, the dependency set on c^* has exactly two channels: the one corresponding to the edge $r_2 = \{c^*, \mathsf{SB}(e_4)\}$ (because e_4 and e_5 both have $\mathsf{SalaryBand} = \mathtt{L4}$, so the FD $\mathsf{Level} \to \mathsf{SalaryBand}$ links them), and the one corresponding to edge r_4, which creates the dependency $\{c^*, \mathsf{Lv}(e_5)\}$ (the per-row level-to-band lookup). Observe that records e_4 and e_5 do not share team, title, or manager, so no r_1 or r_3 edge exists.

Between time t_b and time t_e, three more records are inserted. At time t_1, employee e_1 joins

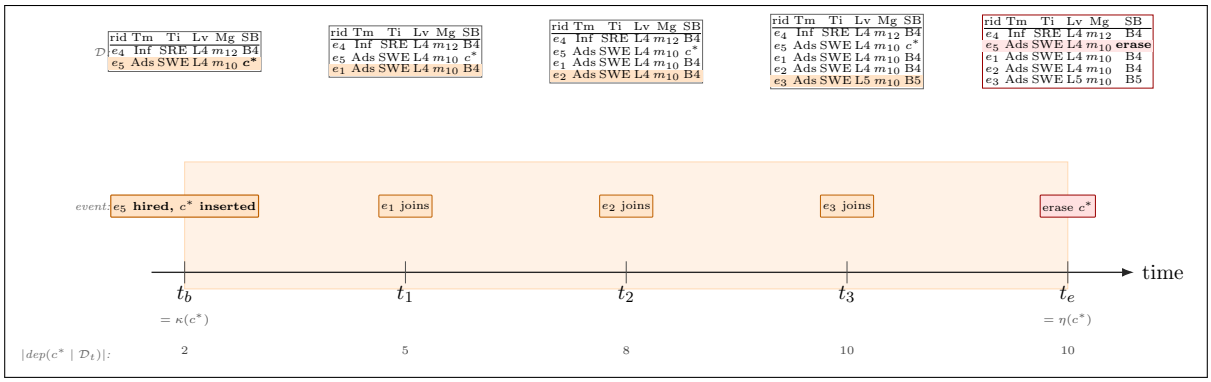


Figure 5.1: Evolution of the Employee instance from t_b to t_e.

(Ads, SWE, L4, m_{10}, B4). The instance now contains a second Ads/SWE employee at L4 under the same manager as e_5. This single record adds three new channels into c^*: an r_1 edge $\{c^*, \mathsf{SB}(e_1)\}$ via shared team (Tm in the figure) and title (Ti in the figure), an additional r_2 edge $\{c^*, \mathsf{SB}(e_1)\}$ via shared level, and an r_3 edge $\{\mathsf{Lv}(e_5), \mathsf{Lv}(e_1)\}$ via shared manager. The dependency set jumps from two channels to five. At t_2, e_2 joins with the same profile and brings three more channels. At time t_3, employee e_3 joins (also Ads/SWE/m_{10}, but at L5/B5) and contributes another r_1 and r_3 edge, respectively. By the time the erasure request arrives at t_e, the dependency set has ten channels: eight more than at time t_b.

In the example, a mechanism that simply sets c^* to NULL at time t_e leaves all ten inference channels intact. The closure algorithm, discussed in Chapter 4, then reaches c^* from many of them. The adversary recovers B4 through, say, the r_1 edge to $\mathsf{SB}(e_1)$ in one step. The naive deletion has not erased what the database now knows about c^*.

Pre-Insertion Post-Erasure Equivalence. The orange-shaded window in Figure 5.1 contains every cell that arrived after c^* was inserted. These are the cells that created new inference channels. **P2E2**, the principle this chapter presents, asks the mechanism to "roll-back" the inference structure on c^* back to its pre-insertion state. Informally: *After erasure, the inferences available about c^* should be no more than the inferences available before c^* was inserted.*

To make the comparison meaningful, the definition of P2E2 restores c^*'s original value in the post-erasure state, asks which inference channels include c^* in that restored state, and requires those channels to be a subset of what existed at time t_b. On our temporal walk in Figure 5.1, at time t_b, the inferences available that include c^* comprise just $\{r_2 : \{c^*, \mathsf{SB}(e_4)\},\ r_4 : \{c^*, \mathsf{Lv}(e_5)\}\}$. P2E2 requires the post-erasure inferences on c^* be a subset of this set. The mechanism must therefore eliminate the eight other channels visible at time t_e. Recall that here, eliminating a channel means setting at least one of its cells to `NULL`.

Which New Inferences Must the Mechanism Address? The eight new channels on the timeline (in Figure 5.1) share a structural feature: each of them involves at least one cell of a record that was inserted *after* time t_b (the time at which e_5 was inserted). The edge $r_1 = \{c^*, \mathsf{SB}(e_1)\}$ has $\mathsf{SB}(e_1)$, whose $\kappa = t_1 > t_b$. The edge $r_3 = \{\mathsf{Lv}(e_5), \mathsf{Lv}(e_2)\}$ uses $\mathsf{Lv}(e_2)$, whose $\kappa = t_2 > t_b$. And so on. None of these channels could have existed at time t_b because the cells that complete them did not yet exist.

The observation has computational consequences, of which the mechanisms presented in this chapter take advantage. Rather than computing two dependency sets (one at time t_b, another at time t_e) and subtracting, we identify the inferences that the mechanism must address directly: a channel is a new inference for c^* if and only if it uses at least one cell whose $\kappa > t_b$. The set of such channels is $\Delta(\text{P2E2}, c^*)$, and restoring P2E2 reduces to blocking every channel in it by setting at least one of its cells to `NULL`. The optimization is to do so at minimum cost. The problem (Opt-P2E2) is NP-hard. We give an exact polynomial-time algorithm under an acyclicity assumption and a constant-factor approximation otherwise.

Contributions. The chapter develops P2E2 as the thesis's first reference point for closing the semantic gap. The main contributions are as follows:

Formalization (Section 5.3). A precise definition of P2E2 and its associated optimization problem Opt-P2E2, with a proof that the problem is NP-hard.

Analytic characterization (Section 5.4). The structural theorem that ties the inferences that the mechanism must address to a single local condition on cell insertion times. The theorem reduces the comparison of two dependency sets at two times to a single-pass walk through the current instance, filtered by their insertion times.

Algorithms (Sections 5.5–5.7). A breadth-first procedure to materialize the set $\Delta(\text{P2E2}, c^*)$ of relevant dependencies, three approaches for Opt-P2E2 (ILP encoding, exact hypergraph algorithm, bounded-ratio greedy approximation), and the extensions to batched on-demand erasure and retention-driven scheduling of derived cells.

Empirical validation (Section 5.8). A measurement of the framework's cost across five real datasets, comparing against cascade-style baselines.

Section 5.2 sets up the temporal vocabulary and Section 5.9 closes with limitations.

5.2 The Temporal Layer

We continue with the formal model introduced in Chapter 4. We will recall the relevant concepts as and when required. To talk about cells inserted before or after the target cell c^*, we associate to each cell in a database a timestamp.

Insertion and Expiration Times. Given a database $\mathcal{D}$, for each cell $c \in \mathit{Cells}(\mathcal{D})$, the database records two timestamps:

- $\kappa(c)$, the *creation time* of c, defined as the moment the cell received its first non-*NULL* value.
- $\eta(c)$, the *expiration time* of c, defined as the moment the cell is to be erased.

We treat an update as a deletion followed by an insertion: if a cell's value changes at time τ, the old cell expires ($\eta = \tau$) and a new cell is inserted with $\kappa = \tau$. The instance at time τ,

denoted $\mathcal{D}_\tau$, contains every cell with $\kappa \leq \tau < \eta$.

For the target cell c^* we write $t_b = \kappa(c^*)$ and $t_e = \eta(c^*)$. At time t_e, the erasure mechanism executes: the value of the cell c^* becomes `NULL`, and the mechanism may also set to `NULL` a set $\mathcal{T} \subseteq \mathit{Cells}(\mathcal{D}) \setminus \{c^*\}$ of auxiliary cells to remove inference channels. We write t_e^+ for the time just after the mechanism has finished. The data remaining post-erasure is denoted as $\mathcal{D}_{t_e^+}$.

Base and Derived Relations. We distinguish two kinds of relations in the instance.

Base relations contain cells whose values are inserted, updated, or deleted directly by users, applications, or external sources. The Employee relation of our running example is a base relation, and its cells carry the time stamps κ and η.

Derived relations contain cells whose values are computed from base cells. Examples include materialized views, summary statistics, and cached aggregates. A derived cell c has a recomputation period $\mathit{freq}(c)$ that specifies how often it must be refreshed against current base data. We treat each recomputation as a deletion followed by an insertion: the previous derived value expires and a fresh one is created with time κ equal to the recomputation time. Derived cells inherit dependencies from the base cells that produced them. The chapter mostly addresses base-cell deletion in Sections 5.3–5.6. Derived cells are addressed in Section 5.7, when we discuss retention-driven erasure.

Hypergraph and Closure: A Brief Recall. We will operate on the dependency hypergraph and the inference closure of Chapter 4. Additionally, we will index them by the instance at which they are evaluated.

Recall (Definition 4.4) that for an instance $\mathcal{D}$ and RDR set Δ, the dependency hypergraph $H(\mathcal{D}, \Delta) = (N, E)$ has cells as nodes and instantiated RDRs as undirected hyperedges, with $E = \{\mathit{Cells}(\delta) : \delta \in \Delta(\mathcal{D})\}$. Recall (Definition 4.7) that the inference closure $K_\Delta^+(\mathcal{D})$ of a

known cell set K contains every cell reachable from K by repeated application of the rule "if all other cells of some δ are already known, add $\mathit{Head}(\delta) \cup \mathit{Tail}(\delta) \setminus K$ to K." We use $K^+(c \mid \mathcal{D}_\tau)$ to denote the closure on the instance at time τ, starting from the cells visible at that time. In Chapter 4, time was implicit; here, we make it explicit because the comparison between $K^+(c^* \mid \mathcal{D}_{t_b})$ and $K^+(c^* \mid \mathcal{D}_{t_e^+})$ is central to P2E2.

We also reuse the dependency-set view of P2E2 [27]: $\mathit{dep}(c \mid \mathcal{D}_\tau)$ denotes the set of instantiated RDRs that directly or indirectly involve c at time τ, all of whose cells do not contain `NULL` in $\mathcal{D}_\tau$. The two views are interchangeable: a cell is in $K^+(c \mid \mathcal{D}_\tau)$ iff some instantiation in $\mathit{dep}(c \mid \mathcal{D}_\tau)$ uses it on a path to c. We will mostly write $\mathit{dep}(c \mid \mathcal{D}_\tau)$ when reasoning about specific inference channels and the set K^+ when reasoning about whether cells can be inferred.

5.3 Pre-insertion Post-Erasure Equivalence (P2E2)

In this section, we formalize P2E2. Intuitively, the definition states a containment between two dependency sets at two different points in time, computed with the value of the target cell c^* restored to its original value. The optimization problem that follows asks for the minimum-cost set of auxiliary cells that achieves the containment.

The P2E2 Guarantee. Let v denote the value of the cell c^* at the time of erasure, i.e., $v = \mathit{Val}(c^*, \mathcal{D}_{t_e})$. Let $\mathcal{D}_{t_e^+} \cup \{\, \mathit{Val}(c^*) \leftarrow v \}$ denote the post-erasure instance with c^*'s value temporarily restored to v. This restoration allows us to compute the dependency set on c^* in a state where the value of the cell c^* itself is not `NULL`, so that channels containing c^* are not trivially excluded from the set of dependencies.

Definition 5.1 (Pre-Insertion Post-Erasure Equivalence)**.** *Given a database $\mathcal{D}$ and a set of Δ of dependencies, an erasure mechanism satisfies* ***pre-insertion post-erasure equivalence***

(P2E2) *for the target cell c^*, with $\kappa(c^*) = t_b$ and $\eta(c^*) = t_e$, if*

$$dep\big(c^* \mid \mathcal{D}_{t_e^+} \cup \{Val(c^*) \leftarrow v\}\big) \subseteq dep\big(c^* \mid \mathcal{D}_{t_b}\big),$$

where $v = Val(c^*, \mathcal{D}_{t_e})$.

In other words, we have the set of inference channels that have c^* in the post-erasure state, with c^* restored. These must be a subset of the set that existed when c^* was first inserted. The mechanism may eliminate inference channels that were present at time t_b (this is not unsafe); it may not preserve channels that did not exist at time t_b.

We make the principle as concrete as possible in the temporal walkthrough of Section 5.1. At t_b, $dep(c^* \mid \mathcal{D}_{t_b}) = \{\delta_{2,e_4}, \delta_{4,e_5}\}$, where δ_{2,e_4} is the r_2 edge $\{c^*, \mathsf{SB}(e_4)\}$ and δ_{4,e_5} is the r_4 edge $\{c^*, \mathsf{Lv}(e_5)\}$. P2E2 requires that, after erasure, the dependency set on c^* contains *at most* these two channels. The eight channels added between t_b and t_e must be eliminated.

Why "equivalence" and not "equality." The definition uses the notion of a subset rather than equality because a mechanism may eliminate channels that existed at t_b as a side effect of eliminating later ones. Equality would unnecessarily forbid such collateral elimination. Besides, the privacy guarantee does not require it: information already available before c^* existed is, by assumption, already available to the adversary, so its loss does not weaken privacy.

Owner assumption. We assume that the database owner has full access to the current state and the dependency model at all times. The owner is not adversarial: the threat we defend against is downstream observers of the database, not an attacker who maintains hidden copies of the cells in $\mathcal{D}_{t_b}$. Incorporating malicious owners goes beyond the scope of P2E2.

The Opt-P2E2 *Optimization Problem.* A P2E2-satisfying mechanism may set arbitrarily many auxiliary cells to `NULL`. Each cell has a deletion cost $\mathit{Cost}(c) \in \mathbb{R}_{\geq 0}$ that captures the value lost to the application by erasing cell c. For example, an attribute frequently queried for downstream reports may have a high cost; a cell whose erasure would invalidate a derived view has an additional cost for reconstructing the view.

Definition 5.2 (Opt-P2E2)**.** *Given a database instance $\mathcal{D}_{t_e}$, a set Δ of RDRs, a target cell c^*, and a cost function $\mathit{Cost} : \mathit{Cells}(\mathcal{D}) \to \mathbb{R}_{\geq 0}$, the* Opt-P2E2 *problem asks for a set*

$$\mathcal{T} = \{c^*\} \cup \{c_1, c_2, \ldots, c_k\} \;\subseteq\; \mathit{Cells}(\mathcal{D}_{t_e})$$

such that setting the values of every cell in $\mathcal{T}$ produces the state $\mathcal{D}_{t_e^+}$ and P2E2 holds for the cell c^*, and the total cost $\sum_{c \in \mathcal{T}} \mathit{Cost}(c)$ is minimized.

The set $\mathcal{T}$ is the *deletion set*: it always contains the target cell c^* and may contain additional auxiliary cells. The mechanism's task is to find such a set of minimum cost.

Theorem 5.1 (Hardness [27])**.** *The* Opt-P2E2 *is* NP*-hard.*

Proof. We reduce from Set Cover. Let $(U, \mathcal{S} = \{S_1, \ldots, S_m\}, k)$ be an instance of Set Cover with universe U and collection of sets $\mathcal{S}$. We build an Opt-P2E2 instance as follows. Create one target cell c^* with $\kappa(c^*) = t_b$; for each set S_i, create a cell s_i with $\kappa(s_i) > t_b$; for each element $u \in U$, create a cell c_u with $\kappa(c_u) > t_b$ and a single instantiated RDR δ_u with $\mathit{Head}(\delta_u) = \{c^*\}$ and $\mathit{Tail}(\delta_u) = \{c_u\} \cup \{s_i \mid u \in S_i\}$. Assign uniform cost 1 to every s_i and mark every c_u as undeletable. Set the deletion budget $B = k$.

By construction, each δ_u involves the newly-inserted cell c_u and so lies in $\Delta(\text{P2E2}, c^*)$, and the deletable cells of $\mathit{Tail}(\delta_u)$ are exactly $\{s_i \mid u \in S_i\}$. A deletion set $\mathcal{T}$ guarantees P2E2 iff every δ_u has a cell in $\mathcal{T}$, which (under the cost weighting) is equivalent to requiring that $\mathcal{T}' = \{S_i \mid s_i \in \mathcal{T}\}$ covers every $u \in U$. Hence $\mathcal{T}$ is a P2E2 deletion set of cost $\leq k$ iff $\mathcal{T}'$ is a

set cover of size $\leq k$. Since Set Cover is NP-hard and the reduction is polynomial-time, Opt-P2E2 is NP-Hard. □

In the following sections, we discuss algorithmic approaches to find the deletion set. But first, we turn to an efficient characterization of the set of dependencies of the target cell c^* that needs to be addressed for P2E2 to hold.

5.4 The Set of New Dependencies $\Delta(\textbf{P2E2}, c^*)$

To find a deletion set, we first identify which inference channels P2E2 actually constrains. P2E2 (Definition 5.1) only pertains to inference channels that exist at time t_e^+ but not at time t_b. We collect those channels into a single parametric object defined as follows.

Definition 5.3 (Set of new dependencies for P2E2)**.** *For a target cell c^* with $\kappa(c^*) = t_b$ and $\eta(c^*) = t_e$, the* set of new inferences *on c^* is*

$$\Delta(\text{P2E2}, c^*) \;=\; dep\big(c^* \;\big|\; \mathcal{D}_{t_e}\big) \;\setminus\; dep\big(c^* \;\big|\; \mathcal{D}_{t_b}\big).$$

An instantiated RDR is in $\Delta(\text{P2E2}, c^*)$ exactly when it infers c^* at time t_e but did not do so at time t_b. These are precisely the channels that a mechanism satisfying P2E2 for the cell c^* must address. By definition, a deletion set $\mathcal{T}$ guarantees P2E2 iff for every $\delta \in \Delta(\text{P2E2}, c^*)$, at least one cell of δ is in $\mathcal{T}$. Restoring P2E2 reduces to hitting every δ in $\Delta(\text{P2E2}, c^*)$.

However, what is not yet obvious is how to compute $\Delta(\text{P2E2}, c^*)$ without explicitly constructing two dependency sets at two different times as in Definition 5.3. Next, we will provide the structural characterization that makes computation efficient.

We first introduce some notation. Between t_b and t_e, the database may have inserted cells (giving them κ in $(t_b, t_e]$) and erased cells (giving them η in $(t_b, t_e]$). Let N denote the set of

cells with $\kappa(c) > t_b$ (cells newer than c^*), and let E denote the set of cells with $\eta(c) > t_b$ and $\eta(c) \leq t_e$ (cells erased between t_b and t_e). The current instance at t_e is $\mathcal{D}_{t_e} = (\mathcal{D}_{t_b} \setminus E) \cup N$.

Theorem 5.2 (Deletion-set characterization, from [27])**.** *Given a target cell c^* with $\kappa(c^*) = t_b$ and $\eta(c^*) = t_e$, a set Δ of RDRs, and the set E of cells such that for each $c \in E$, we have $\eta(c) > t_b$ and $\eta(c) < t_e$. The following hold:*

1. $dep(c^* \mid \mathcal{D}_{t_b} \setminus E) \subseteq dep(c^* \mid \mathcal{D}_{t_b})$.
2. $dep(c^* \mid \mathcal{D}_{t_e}) \setminus dep(c^* \mid \mathcal{D}_{t_b} \setminus E) \;=\; \big\{\delta \in dep(c^* \mid \mathcal{D}_{t_e}) \;:\; \exists\, c \in Cells(\delta) \text{ with } \kappa(c) > t_b\big\}$.

Proof. Part 1. Let $\delta \in dep(c^* \mid \mathcal{D}_{t_b} \setminus E)$. By definition, every cell of δ lies in $\mathcal{D}_{t_b} \setminus E \subseteq \mathcal{D}_{t_b}$, so every cell of δ exists at t_b. Restricting attention to a superset of cells can only add inference channels, not remove them: the rule that produced δ is still applicable in $\mathcal{D}_{t_b}$. Hence $\delta \in dep(c^* \mid \mathcal{D}_{t_b})$.

Part 2, $\subseteq$. Let $\delta \in dep(c^* \mid \mathcal{D}_{t_e}) \setminus dep(c^* \mid \mathcal{D}_{t_b} \setminus E)$. Since δ is not in the dependency set under $\mathcal{D}_{t_b} \setminus E$, at least one cell $c \in Cells(\delta)$ does not belong to $\mathcal{D}_{t_b} \setminus E$. There are two cases. If $c \in E$, then c was erased between t_b and t_e, so $c \notin \mathcal{D}_{t_e}$, contradicting $\delta \in dep(c^* \mid \mathcal{D}_{t_e})$. Hence $c \notin \mathcal{D}_{t_b}$ at all, which means that c was inserted after t_b: $\kappa(c) > t_b$.

Part 2, $\supseteq$. Let $\delta \in dep(c^* \mid \mathcal{D}_{t_e})$ with some $c \in Cells(\delta)$ satisfying $\kappa(c) > t_b$. Then $c \notin \mathcal{D}_{t_b}$, hence $c \notin \mathcal{D}_{t_b} \setminus E$. The rule that instantiated δ requires c to be present, so $\delta \notin dep(c^* \mid \mathcal{D}_{t_b} \setminus E)$, placing δ in the left-hand-side difference. □

The first part states that erasing other cells between time t_b and time t_e never introduces new inference channels on the target cell c^*. The second part states the local characterization: every dependency in $\Delta(\text{P2E2}, c^*)$ contains at least one cell that was inserted after the cell c^* was.

Theorem 5.2 lets us identify $\Delta(\text{P2E2}, c^*)$ by traversing the dependency set at time t_e and

filtering on the insertion time κ, rather than by constructing two dependency sets and subtracting them.

Example 5.2. We illustrate the characterization of the theorem in the timeline of Figure 5.1. Recall $t_b = \kappa(c^*)$, and the cells inserted after t_b are the cells of e_1, e_2, e_3.

At t_e, the dependency set on c^* contains ten channels. Two of them are δ_{2,e_4} and δ_{4,e_5}, which existed at t_b. The remaining eight are:

- Three r_1 edges: $\{c^*, \mathsf{SB}(e_1)\}, \{c^*, \mathsf{SB}(e_2)\}, \{c^*, \mathsf{SB}(e_3)\}$.
- Three r_2 edges: $\{c^*, \mathsf{SB}(e_1)\}, \{c^*, \mathsf{SB}(e_2)\}$. (Note: these are the same sets of cells as the r_1 edges, but instantiated from a different rule, so they form a distinct δ in the dependency set.)
- Three indirect r_3 edges: $\{\mathsf{Lv}(e_5), \mathsf{Lv}(e_1)\}, \{\mathsf{Lv}(e_5), \mathsf{Lv}(e_2)\}, \{\mathsf{Lv}(e_5), \mathsf{Lv}(e_3)\}$.

Each of these eight dependencies contains at least one cell of employee records e_1, e_2, or e_3, all of which have insertion time $\kappa > t_b$. By Theorem 5.2, all eight are in $\Delta(\text{P2E2}, c^*)$. Equally, neither δ_{2,e_4} nor δ_{4,e_5} contains a cell with $\kappa > t_b$ (their cells are either c^* itself, with $\kappa = t_b$, or they are cells of e_4 or e_5 inserted at or before t_b), so neither is in $\Delta(\text{P2E2}, c^*)$. Figure 5.2 visualizes the containment.

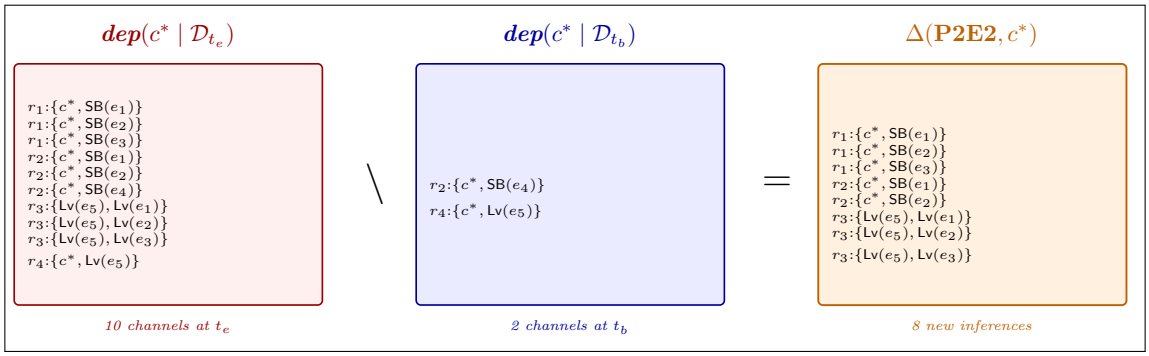


Figure 5.2: The set of new inferences as a set difference on the Employee instance.

Resolving the Violation Set Is a Hitting Set. P2E2 is satisfied iff every $\delta \in \Delta(\text{P2E2}, c^*)$ has at least one cell in $\mathcal{T}$. Solving Opt-P2E2 amounts to choosing $\mathcal{T}$ as a minimum-cost hitting set for the family $\Delta(\text{P2E2}, c^*)$. Each cell of cost $\mathit{Cost}(c)$ in some δ contributes its cost if selected, and we minimize the sum.

This is a useful reframing because hitting-set has a well-understood structure: it is NP-hard in general (consistent with Theorem 5.1), it admits an exact LP-based encoding (Section 5.6.1), and it admits a greedy approximation whose ratio depends on the maximum number of δs in which a single cell appears (Section 5.6.3). The chapter's algorithms exploit an additional structure: the instantiated RDRs in $\Delta(\text{P2E2}, c^*)$ naturally form a hypergraph whose paths admit a polynomial-time exact algorithm under acyclicity (Section 5.6.2).

5.5 Computing $\Delta(\textbf{P2E2}, c^*)$

Theorem 5.2 gives the local condition that an instantiated RDR satisfies iff it is in $\Delta(\text{P2E2}, c^*)$: at least one cell of the RDR has insertion time $\kappa > t_b$. We materialize $\Delta(\text{P2E2}, c^*)$ by a breadth-first walk over the dependency hypergraph that, at each step, only retains instantiations whose cells include at least one with $\kappa > t_b$ in Algorithm 2 (DepInst).

The procedure eval on line 8 evaluates the SQL condition of r to find rid tuples that include c, instantiate the rule, and return the resulting δ (or $\perp$ if no instantiation exists). The check on line 9 enforces Theorem 5.2: a δ is admitted to $\mathcal{I}$ only when some cell of δ is newer than c^*. Cells of δ that are not yet in $\mathcal{V}$ are queued to capture indirect inference: an RDR whose tail contains a cell c' that itself can be inferred from yet other cells contributes a longer derivation path, and we must follow c' to find those RDRs as well.

BFS and not full closure. We do not need to implement the closure discussed in Chapter 4 on $\mathcal{D}_{t_e}$ to find $\Delta(\text{P2E2}, c^*)$. The closure tracks which cells become known; DepInst tracks which instantiated RDRs are new inferences for P2E2. The latter is captured by a topological

Algorithm 2 Materializing the set of new inferences $\Delta(\text{P2E2}, c^*)$.

Input: Instance $\mathcal{D}_{t_e}$, RDR set Δ, target cell c^* with $\kappa(c^*) = t_b$.
Output: Set $\mathcal{V}$ of cells reachable from c^* and set $\mathcal{I}$ of instantiated RDRs in $\Delta(\text{P2E2}, c^*)$.

```
1: Q ← {c*}                                              ▷ BFS queue of cells
2: V ← ∅; I ← ∅
3: while Q ≠ ∅ do
4:     c ← Q.pop()
5:     if c ∉ V then
6:         V ← V ∪ {c}
7:         for all rules r ∈ Δ that name c in their dependence do
8:             δ ← EVAL(r, c, D_{t_e})                   ▷ instantiate r at c
9:             if δ ≠ ⊥ and ∃ c' ∈ Cells(δ) with κ(c') > t_b then
10:                I ← I ∪ {δ}
11:                for all c' ∈ Tail(δ) do
12:                    Q.push(c')
13: return (V, I)
```

walk over the RDR graph rooted at c^*, and the check on the insertion time κ. Since every δ added to $\mathcal{I}$ has at least one cell with $\kappa > t_b$, and since such cells did not exist at t_b, none of the admitted instantiations could have been in $dep(c^* \mid \mathcal{D}_{t_b})$. Therefore, by Theorem 5.2, the set $\mathcal{I}$ that DepInst returns equals $\Delta(\text{P2E2}, c^*)$.

Asymptotic Complexity. Let $u = |\mathcal{V}|$ be the number of distinct cells the BFS visits, $\rho = |\Delta|$ the number of RDR templates, and a the maximum cell arity of any rule (the largest number of cells δ can name). Each cell is processed once; for each cell, up to ρ rules are evaluated; each evaluation produces an instantiation of size at most a, each of whose tail cells is pushed onto the queue. The total work is $O(u\rho a)$, and the queue and instantiation set together use $O(u + u\rho)$ space.

In practice, the dependency hypergraph is sparse: most rules involve only a small number of cells, and most cells are touched by only a few rules. Empirically (Section 5.8), DepInst runtimes scale near-linearly with instance size on all our datasets.

Example 5.3. We run DepInst on the scenario discussed in Figure 5.1. The starting cell is $c^* = \mathsf{SalaryBand}(e_5)$. The BFS proceeds as follows.

Step 1. Pop c^*. The rules naming c^* in their dependence are r_1, r_2, r_4. Instantiations:

- $\delta_{1,e_1} : c^* \not\perp\!\!\!\perp \mathsf{SB}(e_1)$. Cells $\{c^*, \mathsf{SB}(e_1)\}$. $\kappa(\mathsf{SB}(e_1)) = t_1 > t_b$. **Admit.**
- $\delta_{1,e_2} : c^* \not\perp\!\!\!\perp \mathsf{SB}(e_2)$. Cell $\mathsf{SB}(e_2)$ has $\kappa = t_2 > t_b$. **Admit.**
- $\delta_{1,e_3} : c^* \not\perp\!\!\!\perp \mathsf{SB}(e_3)$. Cell $\mathsf{SB}(e_3)$ has $\kappa = t_3 > t_b$. **Admit.**
- $\delta_{2,e_4} : c^* \not\perp\!\!\!\perp \mathsf{SB}(e_4)$. Cell $\mathsf{SB}(e_4)$ has $\kappa < t_b$ (inserted at t_0). **Reject.**
- $\delta_{2,e_1}, \delta_{2,e_2} : c^* \not\perp\!\!\!\perp \mathsf{SB}(e_1), \mathsf{SB}(e_2)$. Same arguments as r_1 counterparts: cells $\mathsf{SB}(e_i)$ with $\kappa > t_b$. **Admit.**
- $\delta_{4,e_5} : c^* \not\perp\!\!\!\perp \mathsf{Lv}(e_5)$. Cell $\mathsf{Lv}(e_5)$ has $\kappa = t_b$, not greater than t_b. **Reject.**

Push the tail cells of admitted instantiations: $\mathsf{SB}(e_1), \mathsf{SB}(e_2), \mathsf{SB}(e_3)$.

Step 2 (and beyond). Pop each $\mathsf{SB}(e_i)$. For each, evaluate the rules. Most resulting instantiations recurse to c^* (already visited) or to peers ($\mathsf{SB}(e_j)$); these are rejected if their cells are all pre-t_b, or admitted otherwise. The BFS also reaches $\mathsf{Lv}(e_5)$ through r_4 edges from $\mathsf{SB}(e_i)$, and from $\mathsf{Lv}(e_5)$ continues to $\mathsf{Lv}(e_1), \mathsf{Lv}(e_2), \mathsf{Lv}(e_3)$ through r_3. Each r_3 instantiation $\delta_{3,e_j} : \mathsf{Lv}(e_5) \not\perp\!\!\!\perp \mathsf{Lv}(e_j)$ has its tail cell with $\kappa > t_b$ (since e_j was inserted after t_b), so all three are admitted.

Result. The set $\mathcal{I}$ at termination contains the eight new inferences. The two pre-existing instantiations δ_{2,e_4} and δ_{4,e_5} are correctly filtered out by the κ check on line 9 of Algorithm 2. The cell set $\mathcal{V}$ contains c^* and the cells of e_1, e_2, e_3 that appear as tails in admitted instantiations, together with $\mathsf{Lv}(e_5)$. Figure 5.3 traces the BFS step by step and highlights the admit/reject decisions made by the κ filter.

Rule	Instantiation δ (popping c^*)	$\exists c \in \textbf{\textit{Cells}}(\delta) : \kappa(c) > t_b$?	Decision
r_1	$\delta_{1,e_1} : c^* \not\perp \mathsf{SB}(e_1)$	yes ($\kappa(\mathsf{SB}(e_1)) = t_1$)	**admit**
r_1	$\delta_{1,e_2} : c^* \not\perp \mathsf{SB}(e_2)$	yes ($\kappa(\mathsf{SB}(e_2)) = t_2$)	**admit**
r_1	$\delta_{1,e_3} : c^* \not\perp \mathsf{SB}(e_3)$	yes ($\kappa(\mathsf{SB}(e_3)) = t_3$)	**admit**
r_2	$\delta_{2,e_1} : c^* \not\perp \mathsf{SB}(e_1)$	yes	**admit**
r_2	$\delta_{2,e_2} : c^* \not\perp \mathsf{SB}(e_2)$	yes	**admit**
r_2	$\delta_{2,e_4} : c^* \not\perp \mathsf{SB}(e_4)$	no ($\kappa(\mathsf{SB}(e_4)) < t_b$)	**reject**
r_4	$\delta_{4,e_5} : c^* \not\perp \mathsf{Lv}(e_5)$	no ($\kappa(\mathsf{Lv}(e_5)) = t_b$)	**reject**

Subsequent BFS steps pop $\mathsf{SB}(e_1), \mathsf{SB}(e_2), \mathsf{SB}(e_3)$ and admit their r_1, r_2, r_4 instantiations; then pop $\mathsf{Lv}(e_1), \mathsf{Lv}(e_2), \mathsf{Lv}(e_3)$ and admit the r_3 edges into $\mathsf{Lv}(e_5)$. BFS terminates with eight admitted instantiations: $\Delta(\text{P2E2}, c^*) = \mathcal{I}$.

Figure 5.3: DepInst BFS trace on the Employee instance.

5.6 Solving Opt-P2E2: Three Approaches

With $\Delta(\text{P2E2}, c^*)$ in hand, the remaining task is to choose a minimum-cost set of cells that hits every instantiation in it. We present three approaches with complementary strengths. The first encodes Opt-P2E2 as an integer linear program (ILP), suitable as a baseline and for small instances where an off-the-shelf solver is applicable. The second is an exact polynomial-time algorithm over a structural object that we call the *dependence hypergraph*; it requires the hypergraph to be acyclic, an assumption that holds in all our datasets. The third is a top-down greedy approximation with a bounded approximation ratio in terms of the maximum cell-arity of rules; it scales to the largest workloads.

5.6.1 ILP Encoding

The simplest encoding represents the set of new inferences as an *induced bipartite graph* and writes the hitting condition as linear inequalities.

Definition 5.4 (Induced bipartite graph)**.** *For a target cell c^* and set of new inferences $\Delta(\text{P2E2}, c^*) = \{\delta_1, \ldots, \delta_n\}$, the* induced bipartite graph *is $\mathcal{B} = (V_L \cup V_R, E_H \cup E_T)$ where $V_L = \{\delta_i : 1 \le i \le n\}$, $V_R = \bigcup_i \mathit{Cells}(\delta_i)$, $E_H = \{(\delta_i, c) : c \in \mathit{Head}(\delta_i)\}$, and $E_T = \{(\delta_i, c) :$*

$c \in \mathit{Tail}(\delta_i)\}$.

Variables. Following [27], we introduce four families of binary variables::

- $a_j \in \{0,1\}$ for each cell $c_j \in V_R$: indicates whether c_j is in the deletion set $\mathcal{T}$.
- $b_i \in \{0,1\}$ for each instantiation $\delta_i \in V_L$: indicates whether the head cell of δ_i is in $\mathcal{T}$.
- $h_i^j \in \{0,1\}$ for each head edge $(\delta_i, c_j) \in E_H$: tracks whether the cell at the head of δ_i is in $\mathcal{T}$.
- $t_i^j \in \{0,1\}$ for each tail edge $(\delta_i, c_j) \in E_T$: tracks whether a cell in the tail of δ_i is in $\mathcal{T}$.

Constraints. The constraints are:

1. $a_{c^*} = 1$ (the target is always deleted).
2. $a_j = h_i^j$ for every $(\delta_i, c_j) \in E_H$ (the head-edge variable mirrors cell deletion).
3. $b_i = h_i^j$ for the single head edge of each δ_i (the head indicator of δ_i equals its head-edge variable).
4. $\sum_{c_j \in \mathit{Tail}(\delta_i)} t_i^j \geq b_i$ for every $\delta_i \in V_L$ (if the head of δ_i is deleted, at least one tail cell of δ_i must also be deleted).
5. $a_j = t_i^j$ for every $(\delta_i, c_j) \in E_T$ (the tail-edge variable mirrors cell deletion).

The objective is

$$\min \sum_{c_j \in V_R} a_j \cdot \mathit{Cost}(c_j).$$

The system $\mathcal{O}$ has $O(|V_R|)$ cell variables and $O(|V_L|)$ instantiation variables, plus $O(|E_H| + |E_T|)$ edge variables. It has $O(|V_L| \cdot |V_R|)$ constraints.

How the cascade ensures P2E2. The encoding works through a cascade rooted at c^*. Constraint (1) forces $a_{c^*} = 1$. For every δ_i with c^* as head, constraint (2) gives $h_i^{c^*} = 1$, constraint (3) gives $b_i = 1$, and constraint (4) forces at least one tail cell of δ_i to be deleted. By constraint (5), deleting that tail cell sets $t_i^{c'} = 1$ and consequently $a_{c'} = 1$ for the chosen tail cell c'. If c' is itself the head of a downstream $\delta_{i'}$, constraints (2)–(4) repeat at $\delta_{i'}$. The cascade propagates through every chain rooted at c^* in the bipartite graph until every $\delta \in \Delta(\text{P2E2}, c^*)$ has been hit. The encoding therefore does *not* require the dependence hypergraph to be acyclic: the constraints express the hitting condition through edge variables that cascade, regardless of whether the underlying graph is a tree or contains cycles. This is the principal advantage of the ILP over the hypergraph algorithm of Section 5.6.2, whose optimality assumes acyclicity.

Theorem 5.3 (Equivalence of Opt-P2E2 and the ILP, from [27])**.** *For any $w \in \mathbb{N}$, the minimum-cost deletion set $\mathcal{T}^*$ guaranteeing P2E2 for c^* has cost w iff the ILP $\mathcal{O}$ admits a 0–1 solution with objective $W = w$.*

Proof. We first establish a structural property of the bipartite graph $\mathcal{B}$ obtained from Algorithm 2, then use it to prove both directions of the equivalence.

Chain connectivity. The BFS of Algorithm 2 places c^* in the queue at depth 0 and admits an instantiation δ_i into $\mathcal{I}$ only after a cell $c \in \mathit{Cells}(\delta_i)$ has been popped. The popping order partitions the cells of V_R into BFS depths $d(c) \in \mathbb{N}$ with $d(c^*) = 0$. By construction, for every admitted $\delta_i \in \Delta(\text{P2E2}, c^*)$, either (i) $c^* \in \mathit{Head}(\delta_i)$ (depth-1 instantiation) or (ii) there exists a sequence $c^* = c_{(0)}, \delta_{(1)}, c_{(1)}, \delta_{(2)}, c_{(2)}, \ldots, c_{(k-1)}, \delta_{(k)} = \delta_i$ in $\mathcal{B}$ such that, for each ℓ, $c_{(\ell-1)} \in \mathit{Tail}(\delta_{(\ell)})$, $c_{(\ell)} \in \mathit{Head}(\delta_{(\ell)})$, and $c_{(\ell)} \in \mathit{Head}(\delta_{(\ell+1)})$. We call this a *head-tail chain* from c^* to δ_i. The chain exists because BFS reached δ_i via a cell-edge alternation rooted at c^*.

ILP solution $\Rightarrow$ P2E2 deletion set. Let (a, b, h, t) be a feasible 0–1 assignment, and set $\mathcal{T} = \{c_j \in V_R : a_j = 1\}$. Constraint (1) gives $c^* \in \mathcal{T}$. We prove by induction on the length

k of the head-tail chain from c^* to δ_i that some cell of $\mathit{Cells}(\delta_i)$ is in $\mathcal{T}$. *Base ($k = 1$):* $c^* \in \mathit{Head}(\delta_i)$. Constraint (2) with $j = c^*$ gives $h_i^{c^*} = a_{c^*} = 1$; constraint (3) gives $b_i = 1$; constraint (4) gives $\sum_{c_j \in \mathit{Tail}(\delta_i)} t_i^j \geq 1$, so some tail cell c_j has $t_i^j = 1$; and constraint (5) gives $a_j = 1$, so $c_j \in \mathcal{T} \cap \mathit{Cells}(\delta_i)$. *Step:* suppose the claim holds for chains of length $< k$, and let $c^* = c_{(0)}, \delta_{(1)}, \ldots, \delta_{(k)} = \delta_i$ be a chain of length k. At step $\ell = k - 1$, the induction gives a tail cell $c_{(\ell)} \in \mathit{Tail}(\delta_{(\ell)}) \cap \mathcal{T}$. Since the chain alternation has $c_{(\ell)} \in \mathit{Head}(\delta_{(\ell+1)}) = \mathit{Head}(\delta_i)$, constraint (2) at $(\delta_i, c_{(\ell)})$ gives $h_i^{c_{(\ell)}} = 1$, constraint (3) gives $b_i = 1$, and constraint (4) forces some tail cell of δ_i into $\mathcal{T}$ exactly as in the base. Hence $\mathcal{T}$ hits every $\delta \in \Delta(\text{P2E2}, c^*)$, and by Theorem 5.2, $\mathcal{T}$ guarantees P2E2 for c^*. The ILP objective $\sum_j a_j \cdot \mathit{Cost}(c_j)$ equals $\sum_{c \in \mathcal{T}} \mathit{Cost}(c)$.

P2E2 deletion set $\Rightarrow$ ILP solution. Let $\mathcal{T}$ be any P2E2 deletion set with $c^* \in \mathcal{T}$, and assume w.l.o.g. that $\mathcal{T}$ is the closure of its head-tail propagation. In other words, whenever $\mathit{Head}(\delta_i) \cap \mathcal{T} \neq \emptyset$, the propagation requires some cell of $\mathit{Tail}(\delta_i)$ to be in $\mathcal{T}$. If $\mathcal{T}$ does not contain such a tail cell, replace it with one — this can only reduce or preserve cost, since a smaller P2E2 deletion set always exists under such normalization (or, more formally, the optimum lies in this normalized family). Define $a_j = \mathbf{1}[c_j \in \mathcal{T}]$; $b_i = \mathbf{1}[\mathit{Head}(\delta_i) \cap \mathcal{T} \neq \emptyset]$; $h_i^j = a_j \cdot \mathbf{1}[c_j \in \mathit{Head}(\delta_i)]$; $t_i^j = a_j \cdot \mathbf{1}[c_j \in \mathit{Tail}(\delta_i)]$. Constraint (1) holds by $c^* \in \mathcal{T}$. Constraints (2), (3), and (5) hold by definition. For constraint (4), if $b_i = 1$ then $\mathit{Head}(\delta_i)$ has a cell in $\mathcal{T}$; by normalization, $\mathit{Tail}(\delta_i) \cap \mathcal{T} \neq \emptyset$, so $\sum_{c_j \in \mathit{Tail}(\delta_i)} t_i^j \geq 1$. The ILP objective evaluates to $\sum_{c \in \mathcal{T}} \mathit{Cost}(c)$.

Equivalence of optima. Both directions are cost-preserving, so the minimum-cost feasible ILP solution and the minimum-cost P2E2 deletion set have the same cost. In particular, $\min \sum_j a_j \cdot \mathit{Cost}(c_j) = w$ iff the minimum-cost deletion set has cost w. □

The ILP baseline is appropriate when the set of new inferences is small or when an exact optimum is required without structural assumptions. For larger workloads, the hypergraph

algorithm in Section 5.6.2 is generally faster and optimal when the dependence graph is acyclic.

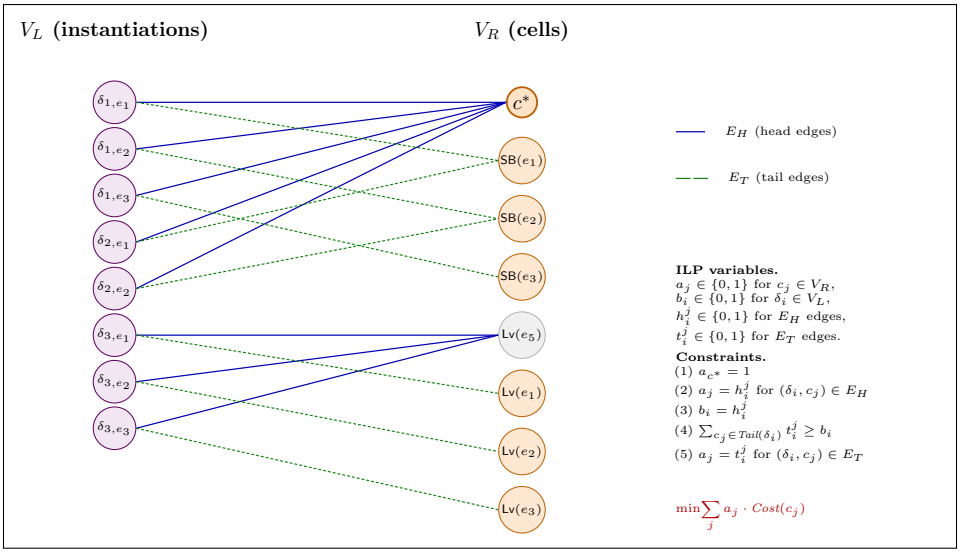


Figure 5.4: Induced bipartite graph for the ILP encoding.

5.6.2 Exact Algorithm via the Dependence Hypergraph

The dependence hypergraph organizes the set of new inferences so that the hitting condition becomes a path-cover condition, which admits a polynomial-time exact solution by two passes over the structure.

Definition 5.5 (Dependence hypergraph for the set of new inferences)**.** *The* dependence hypergraph *for* $\Delta(\text{P2E2}, c^*)$ *is* $\mathcal{H} = (V, E)$*, where* $V = \bigcup_{\delta \in \Delta(\text{P2E2}, c^*)} \mathit{Cells}(\delta)$ *and* $E = \{(\mathit{Head}(\delta), \mathit{Tail}(\delta)) : \delta \in \Delta(\text{P2E2}, c^*)\}$.

This is a special case of a hypergraph built on the set $\Delta(\text{P2E2}, c^*)$ of dependencies.

Definition 5.6 (Root, leaf, path, complete path)**.** *Let* $\mathcal{H} = (V, E)$ *be a dependence hypergraph. A vertex* $v \in V$ *is a* root *if* v *appears in no Tail, that is, for every* $(\mathit{Head}(\delta), \mathit{Tail}(\delta)) \in E$*,* $v \notin \mathit{Tail}(\delta)$*. A vertex* v *is a* leaf *if* v *appears in no Head, that is,* v *is not in* $\mathit{Head}(\delta)$ *for any edge. A sequence* $P : v_1, v_2, \ldots, v_n$ *of vertices is a* path *if (i) some edge has* v_1 *in its*

Algorithm 3 Exact Opt-P2E2 via dependence hypergraph (OptPath).

Input: Output $(\mathcal{V}, \mathcal{I})$ from Algorithm 2.
Output: Deletion set $\mathcal{T}$ guaranteeing P2E2 with minimum total cost.

Phase 1 (bottom-up): compute minimum cost to break inference from each cell.

$\mathcal{Q} \leftarrow \text{leaves}(\mathcal{V})$; $\mathcal{S} \leftarrow \emptyset$
while $\mathcal{Q} \neq \emptyset$ **do**
  $c \leftarrow \mathcal{Q}.\text{pop}()$
  if $c \notin \mathcal{S}$ **then**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{c\}$; $\widehat{\text{Cost}}(c) \leftarrow \mathit{Cost}(c)$
    for all $\delta \in \mathcal{I}(c)$ **do**
      if $c \in \mathit{Head}(\delta)$ **then**
        $\widehat{\text{Cost}}(c) \leftarrow \widehat{\text{Cost}}(c) + \min_{c' \in \mathit{Tail}(\delta)} \widehat{\text{Cost}}(c')$
      else
        $\mathcal{Q}.\text{push}(\mathit{Head}(\delta))$

Phase 2 (top-down): extract the deletion set.

$\mathcal{T} \leftarrow \{c^*\}$; $\mathcal{Q} \leftarrow \{c^*\}$; $\mathcal{S} \leftarrow \emptyset$
while $\mathcal{Q} \neq \emptyset$ **do**
  $c \leftarrow \mathcal{Q}.\text{pop}()$
  if $c \notin \mathcal{S}$ **then**
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{c\}$
    for all $\delta \in \mathcal{I}(c)$ with $c \in \mathit{Head}(\delta)$ **do**
      $c_{\text{child}} \leftarrow \arg\min_{c' \in \mathit{Tail}(\delta)} \widehat{\text{Cost}}(c')$
      $\mathcal{T} \leftarrow \mathcal{T} \cup \{c_{\text{child}}\}$; $\mathcal{Q}.\text{push}(c_{\text{child}})$
return $\mathcal{T}$

head, and (ii) for every $1 \leq i < n$*, there exists an edge whose head contains* v_i *and whose tail contains* v_{i+1}*. The sequence is a* subpath *of* P *if it is a path consisting of consecutive vertices of* P*. The path* P *is* complete *if* v_n *is a leaf.*

A deletion set $\mathcal{T}$ guarantees P2E2 iff every complete path from c^* in $\mathcal{H}$ contains at least one vertex in $\mathcal{T}$. This is the path-cover view of the hitting-set condition: each complete path corresponds to a chain of inference that, if entirely visible, would recover c^*. Deleting any one vertex on the path blocks the chain.

Algorithm 3 (OptPath) computes the optimum in two passes.

Phase 1: bottom-up cost propagation. Leaves of $\mathcal{H}$ are assigned their own cost $\widehat{\text{Cost}}(c) = \mathit{Cost}(c)$. An inner vertex's cost is its own cost plus, for each instantiation in which it appears

as the head, the minimum cost over the cells in that instantiation's tail. Intuitively, to block inference into c via this δ, we either delete c itself or delete the cheapest tail cell of δ (whose recursive cost has already been computed). Summing across head-incident instantiations expresses the disjunctive option per edge.

Phase 2: top-down extraction. We start from c^* and, at each vertex, descend into each head-incident instantiation by the tail cell of minimum $\widehat{\text{Cost}}$. The chosen tail cell joins $\mathcal{T}$. The recursion gives a deletion set that hits every complete path emanating from c^* at exactly one vertex per path, by construction.

Theorem 5.4 (Correctness and optimality of OptPath, from [27])**.** *If $\mathcal{H}$ is acyclic, the set $\mathcal{T}$ returned by Algorithm 3 guarantees P2E2 for c^* and minimizes $\sum_{c\in\mathcal{T}} \mathit{Cost}(c)$.*

Proof. We use the *chain-blocking* characterization of P2E2: $\mathcal{T}$ guarantees P2E2 for c^* iff, for every $\delta \in \Delta(\text{P2E2}, c^*)$, $\mathcal{T} \cap \mathit{Cells}(\delta) \neq \emptyset$. Since deleting any one cell of δ prevents that rule from firing, blocking δ blocks every complete path through δ at the inference level (Theorem 5.2).

Correctness. We show by induction on the depth of c in $\mathcal{H}$ that, when Phase 2 visits c, every $\delta \in \mathcal{I}$ with $c \in \mathit{Head}(\delta)$ has some cell in $\mathcal{T}$. *Base (c a leaf):* no δ has c as head, so the claim is vacuous; if $c = c^*$ is a leaf (extremal case), $\mathcal{T} \ni c^*$ directly. *Step:* suppose the claim holds for all cells of greater depth. When Phase 2 visits c, it iterates over every $\delta \in \mathcal{I}(c)$ with $c \in \mathit{Head}(\delta)$, picks one $c_{\text{child}} \in \mathit{Tail}(\delta)$, and adds c_{child} to $\mathcal{T}$. Hence $\mathcal{T} \cap \mathit{Cells}(\delta) \supseteq \{c_{\text{child}}\} \neq \emptyset$ for every such δ. Starting from $c^* \in \mathcal{T}$ and applying this argument to every cell reached by the recursion, every $\delta \in \Delta(\text{P2E2}, c^*)$ whose head is reachable in $\mathcal{H}$ from c^* has its cell set hit. By the construction of $\mathcal{H}$ from $\Delta(\text{P2E2}, c^*)$ via Algorithm 2, every such δ is reachable from c^* along head-tail chains. Hence $\mathcal{T}$ blocks every δ and guarantees P2E2.

Optimality. For each cell c, define

$$OPT(c) \;=\; \min_{\mathcal{T}'}\Big\{ \sum_{c' \in \mathcal{T}'} Cost(c') : c \in \mathcal{T}',\ \mathcal{T}' \text{ blocks every } \delta \text{ reachable in } \mathcal{H} \text{ from } c \Big\}.$$

On a leaf, no δ has c in its head, so $OPT(c) = Cost(c)$. On an inner cell c with head-incident edges $\delta_1, \ldots, \delta_k$, any valid $\mathcal{T}'$ must, for each ℓ, contain at least one cell of $Cells(\delta_\ell)$; the cheapest such commitment per edge is $\min_{c' \in Tail(\delta_\ell)} OPT(c')$. (Picking c itself in lieu of a tail cell does not avoid the per-edge tail commitment, since c being in $Head(\delta_\ell)$ does not block δ_ℓ. The rule is applicable whenever all tail cells are present, regardless of head deletion.) The acyclicity of $\mathcal{H}$ ensures the tail-subtree subproblems are disjoint, so the cost decomposes additively and

$$OPT(c) \;=\; Cost(c) + \sum_{\ell=1}^{k} \min_{c' \in Tail(\delta_\ell)} OPT(c').$$

Phase 1 computes $\widehat{\mathrm{Cost}}(c) = Cost(c) + \sum_\ell \min_{c' \in Tail(\delta_\ell)} \widehat{\mathrm{Cost}}(c')$ bottom-up, so by induction on depth, $\widehat{\mathrm{Cost}}(c) = OPT(c)$ for every c. Phase 2 selects the arg min at each edge and accumulates them; the total cost of $\mathcal{T}$ equals $\widehat{\mathrm{Cost}}(c^*) = OPT(c^*)$. Any P2E2 deletion set containing c^* has cost $\geq OPT(c^*)$, so $\mathcal{T}$ is optimal. □

The acyclicity assumption requires that no two instantiations δ_1, δ_2 in the set of new inferences share a non-trivial tail. When cycles occur, a simple normalization restores acyclicity at no cost to correctness: among any two tail-sharing instantiations, discard the one with the larger tail. The discarded δ is dominated in the path-cover analysis by the one that is kept. In our evaluation (Section 5.8), all five datasets satisfy acyclicity natively or after this normalization.

Complexity. Let $n = |\mathcal{I}|$, a be the maximum rule arity, and $C_{\max}$ be the maximum cell cost. The bottom-up phase runs in $O(n \cdot a \cdot \log C_{\max})$ (each instantiation contributes a tail comparisons; the log comes from the priority structure used to maintain a minimum of tails).

The top-down phase adds $O(C_{\max} \cdot a \cdot \log C_{\max})$. The total is $O((n + C_{\max}) \cdot a \cdot \log C_{\max})$, with $O(n \cdot a)$ space.

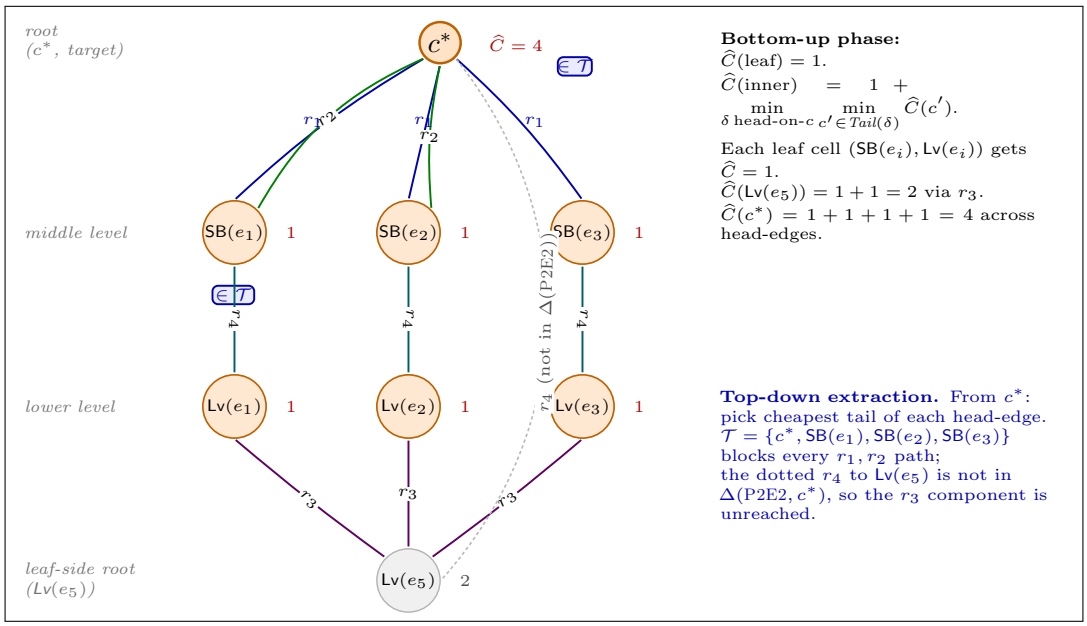


Figure 5.5: Dependence hypergraph for $\Delta(\text{P2E2}, c^*)$ with OptPath trace.

5.6.3 Greedy Approximation

When the set of inferences is very large or acyclicity does not hold, we use a top-down greedy variant of OptPath that avoids the bottom-up cost propagation by committing to a locally cheap choice at each step.

Algorithm sketch. Starting from c^*, the greedy algorithm walks down each head-incident instantiation, picking the tail cell of minimum *Cost* (not $\widehat{\text{Cost}}$). Each picked cell is added to $\mathcal{T}$ and is recursed into. The procedure visits each instantiated cell at most once and terminates when all reachable instantiations have been hit.

Complexity. With n instantiations and maximum arity a, the greedy algorithm runs in $O(an)$ time and $O(an)$ space. It is essentially a single top-down pass over the partial hypergraph, instantiating rules lazily.

By construction, the greedy procedure terminates with a deletion set that hits every complete path emanating from c^*: at every visited cell, it picks one cell of every head-incident instantiation, which by Definition 5.6 is on every complete path through that cell.

Theorem 5.5 (Correctness of the greedy approximation, from [27]). *The set $\mathcal{T}$ returned by the greedy approximation procedure of Section 5.6.3 guarantees P2E2 for c^*.*

Proof. We argue via the chain-blocking characterization: $\mathcal{T}$ guarantees P2E2 for c^* iff $\mathcal{T} \cap \mathit{Cells}(\delta) \neq \emptyset$ for every $\delta \in \Delta(\mathrm{P2E2}, c^*)$. The greedy procedure starts with $\mathcal{T} = \{c^*\}$ and, at every cell c it visits, processes every $\delta \in \mathcal{I}(c)$ with $c \in \mathit{Head}(\delta)$ by selecting some $c' \in \mathit{Tail}(\delta)$ (the cell with minimum *Cost*) and adding c' to $\mathcal{T}$. By construction, $\mathcal{T} \cap \mathit{Cells}(\delta) \ni c'$, so δ is blocked.

By induction on the depth of c in $\mathcal{H}$ (with c^* at depth 0), every δ reachable from c^* through head-tail chains in $\mathcal{B}$ has its head cell visited and hence has its cell set hit. By the construction of $\Delta(\mathrm{P2E2}, c^*)$ from BFS rooted at c^* (Algorithm 2), every $\delta \in \Delta(\mathrm{P2E2}, c^*)$ is reachable from c^* along such a chain. Therefore $\mathcal{T} \cap \mathit{Cells}(\delta) \neq \emptyset$ for every $\delta \in \Delta(\mathrm{P2E2}, c^*)$, and P2E2 holds.

The procedure may choose tail cells of strictly higher cost than the recursive optimum of Phase 1 in OptPath, so $\sum_{c \in \mathcal{T}} \mathit{Cost}(c)$ may exceed the optimum. □

Greedy can be arbitrarily worse than optimal if costs vary widely: forcing a cheap choice early may later require an expensive deletion to block a previously cheap-looking path. When the objective is cardinality (or equivalently, when costs are uniform), we obtain a bound in terms of the structural parameters of $\mathcal{H}$.

Theorem 5.6 (Approximation ratio of greedy, from [27]). *Let $\mathcal{H} = (V, E)$ be the (acyclic) dependence hypergraph for the set of new inferences, with $|V| = n$ and $|\Delta(\mathrm{P2E2}, c^*)| = r$. Let a be the maximum cells-per-instantiation, and let d be the maximum number of instantiations*

a single cell appears in. Let $\mathcal{T}$ be the greedy output and $\mathcal{T}^*$ be the optimal. Then

$$\frac{|\mathcal{T}|}{|\mathcal{T}^*|} \;\le\; \min\left\{ \frac{d}{r}\left(1+\log_2\frac{a\cdot r}{d}\right),\; \frac{a\cdot d}{n-1}\left(1+\log_2(a\cdot n)\right)\right\}.$$

Proof. The chain-blocking characterization (Theorem 5.2) recasts the deletion task as the hitting-set instance $(V, \Delta(\mathrm{P2E2}, c^*))$, where each $\delta \in \Delta(\mathrm{P2E2}, c^*)$ is a set of cells of size $|\mathit{Cells}(\delta)| \le a$, and each cell lies in at most d such sets. Weighted Set Cover is the dual hypergraph problem, and the greedy procedure of Section 5.6.3 (with arity at most a per visited edge) is a one-pass approximation of the greedy procedure. Its analysis follows the Chvátal–Johnson framework [33]: the greedy ratio for set cover with maximum set size s is $H(s) = 1 + \frac{1}{2} + \cdots + \frac{1}{s} \le 1 + \ln s$.

For the cardinality case, the lower bound $|\mathcal{T}^*| \ge \lceil r/d \rceil$ holds because each cell hits at most d instantiations and the optimum must hit all r. The Chvátal–Johnson bound applied to the bipartite incidence with sets of effective size ar/d (an upper bound on the per-cell coverage of distinct chains) gives

$$|\mathcal{T}| \le |\mathcal{T}^*| \cdot \big(1 + \log_2(ar/d)\big),$$

which combined with $|\mathcal{T}^*| \ge \lceil r/d \rceil$ yields the first bound.

The second bound is obtained by applying the same greedy analysis to a coarser incidence (paths through $\mathcal{H}$ rather than instantiations): a cell lies on $\le ad$ complete paths in $\mathcal{H}$, the total number of non-root cells participating in paths is $\le n - 1$, and the per-cell coverage on this dual instance is $\le an$. The constants follow the same harmonic-sum analysis. The precise constant chasing is carried out in [27]; we report the stated closed form here.

Taking the minimum of the two bounds gives the inequality. □

The first bound is tighter when $d \ll r$ (cells appear in few rules); the second is tighter when

$a \cdot d$ is small relative to n. The bound implies a constant factor of at most a in general and $O(\log d)$ when each cell participates in $O(\log d)$ instantiations. In our evaluation, the typical ratio observed on real workloads is well within these bounds: $1.1\times$ to $1.5\times$ optimal in cardinality, with corresponding-or-better cost ratios.

5.6.4 Comparison

Table 5.1 summarizes the three approaches.

Approach	**Optimality**	**Complexity**	**Best for**
ILP encoding (§5.6.1)	Exact	Worst-case exponential; tractable for small set of new inferences	Baseline; small instances; when no structural assumption is acceptable
OptPath (§5.6.2)	Exact (under acyclicity)	$O((n + C_{\max}) \cdot a \cdot \log C_{\max})$	Typical workloads (all evaluated datasets)
Greedy (§5.6.3)	Bounded ratio (Thm. 5.6)	$O(an)$	Largest workloads; cyclic hypergraphs; cardinality objective

Table 5.1: Three solution approaches to Opt-P2E2.

In the evaluation, we use OptPath as the default exact solver and the greedy as the scalable approximation. The ILP is used primarily as a baseline.

5.7 Demand-Driven and Retention-Driven Erasure

The algorithms in Section 5.6 take a target cell c^* and a current database instance, and produce a P2E2-satisfying deletion set. Real erasure workloads are of two types: *demand-driven* erasures (a user requests deletion of c^* before $\eta(c^*)$), and *retention-driven* erasures (the system erases c^* when $\eta(c^*)$ arrives, on a known schedule). The two types admit different optimizations.

5.7.1 Demand-Driven Erasure with Batching

Data regulations often allow a delay Γ between an erasure request and its physical execution (the *grace period*). On Twitter, for example, GDPR allows a 30-day window in which the system may continue to serve the data while preparing the erasure. The grace period provides us with an opportunity to batch.

If a set of cells $\mathcal{S} = \{c_1^*, c_2^*, \ldots, c_k^*\}$ has all its elements queued for erasure within a window of length Γ, we can compute its set of new inferences jointly. This results in two structural savings.

First, the set of new inferences often shares instantiations. Two cells in the same record, say, may participate in a common r_1 edge; deleting that edge counts toward satisfying P2E2 for both. Only one cell needs to be deleted for two shared instantiations.

Second, the dependence hypergraphs share vertices. When c_i^* and c_j^* both have inference paths through a common cell c', instantiating c' in $\mathcal{V}$ for one suffices for the other.

We exploit both savings by extending DepInst to a multi-target version that pools the queues, marks any cell in $\mathcal{S}$ as already-to-be-deleted (skipping further instantiation of those cells), and feeds a single combined hypergraph to OptPath. The size of the combined hypergraph is typically less than the sum of the per-cell hypergraphs by a factor of two to four on our workloads.

Time-based vs. cardinality-based batching. The grace period defines a maximum window. A practical implementation can trigger batched erasure either when the window expires (time-based) or when the queue reaches a target size K (cardinality-based). The choice is a system-level decision; both are evaluated in Section 5.8 and offer similar amortized savings.

5.7.2 Retention-Driven Erasure of Derived Cells

Demand-driven erasure assumes that the erasure time is determined by an external request. Retention-driven erasure handles the converse: when a cell's η is known in advance and the system is responsible for executing it at the right time. For base cells, retention-driven erasure reduces to a scheduled instance of the demand-driven mechanism, batchable as in Section 5.7.1. The interesting case is *derived cells*, which require reconstruction whenever their base dependencies are erased.

Reconstruction of derived data. A derived cell c with recomputation period $\mathit{freq}(c)$ depends on a set of base cells $\{c_1, \ldots, c_n\}$ with erasure intervals (η_i^b, η_i^e). The system must guarantee two invariants:

1. c is reconstructed at least once every $\mathit{freq}(c)$ (functional invariant).

2. P2E2 holds for every c_i at its erasure time (privacy invariant).

A naive policy reconstructs c at every η_i^b for $i = 1, \ldots, n$, and again periodically to satisfy the functional invariant. The number of reconstructions is then $O(n)$ per period, even when many η_i^b are tightly clustered in time.

Scheduling by maximum interval overlap. Algorithm 4 (SCHEDULER) computes a reconstruction schedule that satisfies both invariants while minimizing the number of reconstructions. The key insight is that reconstructing c at a time ρ that lies inside the erasure interval (η_i^b, η_i^e) simultaneously discharges P2E2 for every c_i whose interval contains ρ. We therefore choose each reconstruction time to maximize the number of erasure intervals within which it falls, subject to the spacing constraint $\rho_{k+1} - \rho_k \leq \mathit{freq}(c)$.

CREATESCHEDULE repeatedly picks the next reconstruction time as the point of maximum overlap with currently pending erasure intervals, subject to being within $\mathit{freq}(c)$ of the previous

Algorithm 4 Reconstruction SCHEDULER for a derived cell c.

```
1: procedure createSchedule(c, freq(c), depSet)
2:     κ(c) ← time.now()
3:     Sch[0] ← maxOverlap(depSet, κ(c) + freq(c))
4:     depSet ← depSet \ {c_i : Sch[0] ∈ (η_i^b, η_i^e)}
5:     i ← 1
6:     while depSet ≠ ∅ do
7:         Sch[i] ← maxOverlap(depSet, Sch[i − 1] + freq(c))
8:         depSet ← depSet \ {c_i : Sch[i] ∈ (η_i^b, η_i^e)}
9:         i ← i + 1
10: procedure updateSchedule(c, freq(c), Sch, depSet)
11:     for all (c_i, η_i^b, η_i^e) ∈ depSet do
12:         if η_i^b > Sch[0] and η_i^e < Sch[1] then
13:             createSchedule(c, freq(c), depSet)
```

reconstruction. Each pick discharges every c_i whose interval covers the chosen point; those c_i are removed from the pending set. The procedure terminates when all dependencies are discharged. updateSchedule is called when a new dependent cell is inserted with an erasure interval that falls between two existing reconstructions; if so, the schedule is recomputed.

The maxOverlap subroutine is a standard interval-overlap maximization: it sorts the interval endpoints and computes the overlap. It runs in $O(n \log n)$. The full algorithm runs in $O(n \log n)$ time and $O(n)$ space.

In our evaluation, Scheduler reduces the number of derived-cell reconstructions by 4% to 55% relative to the naive reconstruction with every deletion, depending on how clustered the dependency erasure intervals are. The savings are largest when many dependencies share an erasure window and smallest when intervals are uniformly spread.

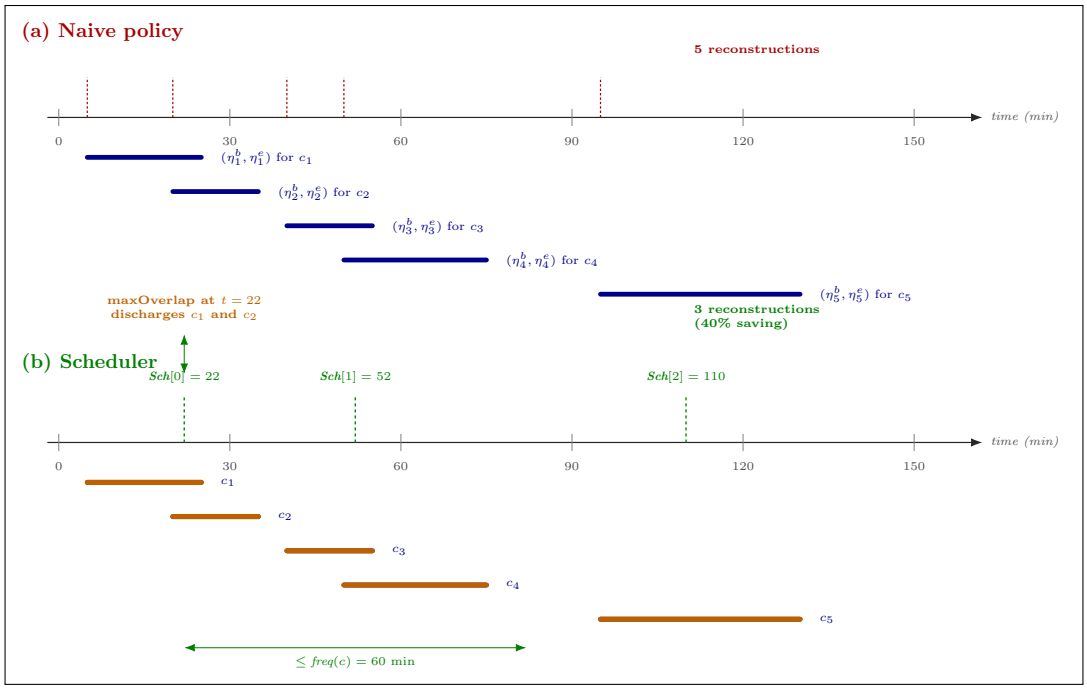


Figure 5.6: SCHEDULER versus a naive reconstruction policy.

5.8 Evaluating P2E2

We evaluate our framework on five datasets. The empirical questions we explore are:

- how much do P2E2 mechanisms cost relative to cascade baselines (Experiment 1)?
- How does cost scale with the degree of dependency in the data (Experiment 2)?
- What savings does batching provide (Experiment 3)?
- What savings does retention-driven scheduling of derived cells provide (Experiment 4)?
- And how do the two regimes interact in a mixed workload (Experiment 5)?

5.8.1 Experimental Setup

Setup. We implemented our algorithms on a single-threaded Java 11 on PostgreSQL 12, ILP encoding solved by Gurobi 11. An Intel Xeon E5-2650, 256 GB RAM was used. All

RDR sets are cycle-free; rules in Twitter and SmartBench join on non-key columns, which we index separately.

Datasets. We use the following five datasets in our evaluations.

(1) *Twitter* [41]. This dataset contains a subset of tweets posted on 𝕏 (formerly Twitter) that represents a real-world instance of our running example. The RDRs were designed manually and express dependencies between individuals and their content.
(2) *Tax* [19]. This dataset is a synthetic dataset created using the real-world distribution of values of American tax records. We discovered all present DCs using [92]. The top-10 DCs that are not entirely comprised of equality predicates are transformed into RDRs. The RDRs include the conditional functional dependencies used in the original publication for data cleaning.
(3) *SmartBench* [50]. This dataset is based on real data collected from sensors deployed throughout the campus at the University of California, Irvine. RDRs capture dependencies between multiple physical sensors that are used to compute derived metrics, like occupancy from Wi-Fi AP locations.
(4) *HotCRP* [65]. This is a dataset containing a sample of real-world conference data. It stores authors, papers, conferences, and their relationships. The RDRs are generated by the method in [2].
(5) *TPC-H* [120]. This well-known benchmark dataset stores transactions of commercial actors: customers and their orders, as well as the suppliers that fulfill those orders. The RDRs capture the links between the tables, i.e., foreign keys. In TPC-H, both customers and suppliers may delete their data. We create two separate scenarios and combine them proportionally to the number of customers and suppliers.
The set of RDRs for each dataset is cycle free. Some RDRs in the Twitter and SmartBench dataset join on non-key columns. To speed up the instantiations, we index those columns separately.

Baselines. Three cascade-style baselines:

- INST: delete every cell in the instantiated dependence set, mimicking traditional foreign-key cascading deletes.
- OPR: delete one cell per instantiated RDR, regardless of cell sharing.
- MINSET: minimum hitting set over instantiated RDRs, computed exactly, but *ignoring* the κ filter of Theorem 5.2.

MINSET is the strongest baseline: it correctly solves the hitting-set problem but processes all dependencies, including those present at t_b that need not be addressed. The gap between MINSET and HGR (the OPTPATH algorithm of Section 5.6.2) isolates the contribution of the temporal filter.

Metrics. We measure (i) the number of additional cells deleted beyond c^*; (ii) total runtime decomposed into RDR instantiation, model construction, model optimization, and DB update; (iii) model memory footprint; and (iv) the number of derived-cell reconstructions for retention-driven scenarios.

5.8.2 Experiments

Experiment 1: Single demand-driven cost. We sample 100 random target cells per base attribute and measure the average per-erasure cost. Table 5.2 reports instantiated and deleted cells across the three P2E2 mechanisms (ILP, HGR, APX) and the three baselines.

On every dataset other than Tax, the baselines delete one to four orders of magnitude more cells than the P2E2 mechanisms. On high-volume social and sensor data (Twitter, SmartBench, and HotCRP), where each user or sensor generates many related entries, the overhead of cascade-style deletion is large. On Tax, every user record is inserted simultaneously, so the temporal component of the deletion set is empty: MINSET matches HGR exactly because no

Dataset	Instantiated			Deleted (P2E2)			Deleted (baselines)		
	ILP	HGr	Apx	ILP	HGr	Apx	Inst	OpR	MinSet
Twitter	0.29	0.29	0.29	0.29	0.29	0.29	837.2	517.8	321.1
Tax	6.93	6.93	5.14	4.07	4.07	4.20	6.93	6.6	4.07
SmartBench	5.43	5.43	5.43	5.43	5.43	5.43	1021.8	1021.8	1021.8
HotCRP	134.4	134.4	4.81	3.78	3.78	3.78	815.0	721.7	74.5
TPC-H	17.63	17.63	17.63	17.63	17.63	17.63	722.2	722.2	722.2

Table 5.2: Average instantiated and deleted cells per erasure request across five datasets [27].

cells were inserted after c^*. The result on Tax demonstrates that the gap between MinSet and HGr on the other datasets is entirely attributable to the κ filter from Theorem 5.2. Among the P2E2 mechanisms, ILP and HGr are exact and always delete the same number of cells. Apx is greedy and within 1.05× of optimal in every dataset except Tax (4.20 vs. 4.07). The runtime gap is wider: on HotCRP, where the RDR chain instantiates a long sequence of cells (most of which are irrelevant to the optimal cost). Apx avoids these instantiations and runs an order of magnitude faster than ILP and HGr. On other datasets, runtimes are dominated by instantiation, not optimization. Figure 5.7 reports the average runtime and space overhead per request across all five datasets. Total per-request runtime decomposes into RDR instantiation, model construction (graph construction for HGr/Apx and ILP setup for ILP), model optimization (the hypergraph traversal or Gurobi solve), and the DB update that writes the *NULL*s. Across all datasets, instantiation dominates (60–90% of total). The optimization phase is nonexistent for HGr. ILP's Gurobi solve adds 0.5–8 seconds on the hardest instances, two orders of magnitude greater than HGr. Memory footprints follow the same pattern. On Twitter (22M cells), total per-erasure runtime is around 50 ms for HGr.

Experiment 2: Sensitivity to dependency degree. The cost of a P2E2 erasure should scale with the number of instantiated cells reached from c^*, not with the size of the database or the number of RDRs in the schema. We test this by running HGr on 100 random erasures per dataset, varying the subset of RDRs that is active for each erasure. Figure 5.8 plots the average number of instantiated cells against the average number of deleted cells; the size of

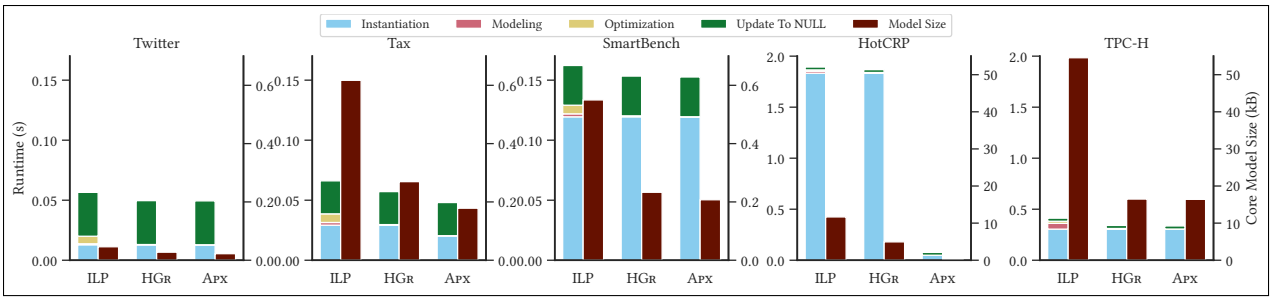


Figure 5.7: Average runtime and space overhead per demand-driven erasure across the five datasets.

each point is the number of RDRs active in that subset. Intuitively, the more dependent cells a target has, the more must be deleted, and the data follow a sublinear trend. The number of rules has no direct effect: large and small points are mixed along the general trend line, confirming that it is the dependency *footprint* for a target cell and not the size of the dependency *schema* that governs cost. HotCRP is a visible outlier: a long chain of RDR instantiations produces many cells, but few of them are on a cheap deletion path. That is why the deletion count remains low while the instantiation count is high.

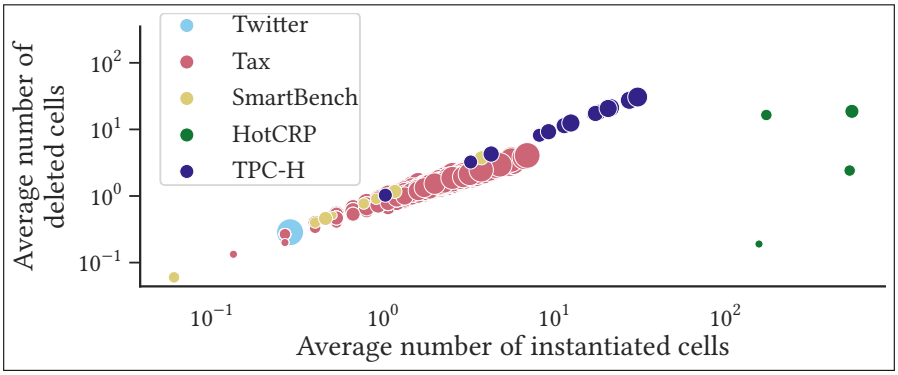


Figure 5.8: Impact of dependencies on P2E2 overhead (log scale); point size denotes the number of RDRs in the subset.

Experiment 3: Batching. Batching exploits the grace period Γ to share cost across erasure requests. We measure both time-based (collect requests for Γ, then process) and cardinality-based (collect K requests, then process) batching. For time-based batching, 1000 erasures are sampled from a 12-hour window on Twitter, Tax, and SmartBench, and processed with grace periods of one, three, and six hours. HotCRP and TPC-H use longer time-scales (one,

seven, and fourteen days for HotCRP; one, three, and six months for TPC-H). Figure 5.9 reports the cumulated runtime and number of deletions for each batch size. Larger batches reduce total runtime and number of deletions; the marginal benefit decreases as the batch grows because the first few merged erasures already discover most of the shared instantiations. On Twitter and Tax, additional deletions plateau quickly; on HotCRP and TPC-H, they continue to decrease in proportion to runtime improvements. For cardinality-based batching, 1000 erasures sampled uniformly, grouped into batches of 10, 50, 100, 500, 1000. The same pattern recurs: larger batches dominate smaller ones, but the bulk of the savings come from the first batching step. On SmartBench, the largest improvement occurs from batch size 100 to 500; on HotCRP, HGR and ILP match APX when batched, because pre-existing cells in the batch obviate the long RDR chain. Across datasets, batching reduces total erasure cost by 16–32% at modest grace periods and recovers a further 5–15% as the grace period grows.

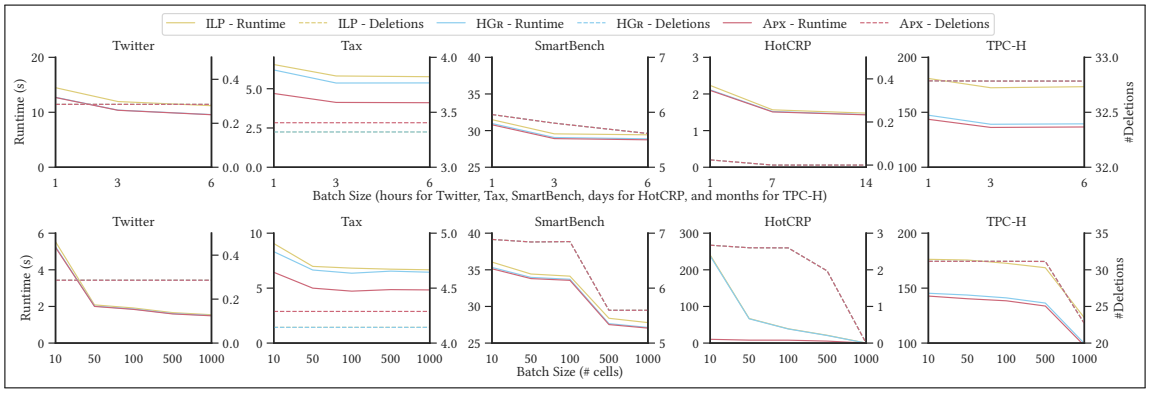


Figure 5.9: Cumulated runtime and number of deletions under time-based and cardinality-based batching.

Experiment 4: Retention scheduling. For retention-driven erasure, the relevant cost is the number of derived-cell reconstructions. We add aggregate derived cells (e.g., total likes per profile on Twitter, or total sales per supplier on TPC-H) to each dataset that did not already contain them, and then ran SCHEDULER on 100 random derived-cell instances per attribute. $freq(c)$ and Γ are dataset-specific (one day for Twitter and Tax with $\Gamma \in [0, 23]$ hours; one hour for SmartBench with $\Gamma \in [0, 55]$ minutes; and one week and one month

for HotCRP and TPC-H with proportional grace periods). Figure 5.10 reports the average reconstructions saved compared to the baseline that reconstructs the derived cell at every base-cell erasure plus periodic refresh. Savings depend on each dataset's update characteristic, which determines how often base-cell expiration intervals overlap. HotCRP and SmartBench have synchronously aggregated base data: maximum saving is reached immediately at $\Gamma = 0$ (55.8% and 4.4% respectively), and the grace period adds nothing further. Twitter and TPC-H exhibit bursty updates: initial increases in Γ reduce reconstructions sharply (Twitter: 38% at $\Gamma = 1$ hour, 48% at $\Gamma = 12$ hours) and then plateau. Tax has continuous updates with every record at a unique time: with $\Gamma = 0$ there is no overlap and no saving, but the saving grows steadily with Γ to 35% at $\Gamma = 23$ hours, and the large number of base cells per derived cell makes the eventual saving the largest of any dataset.

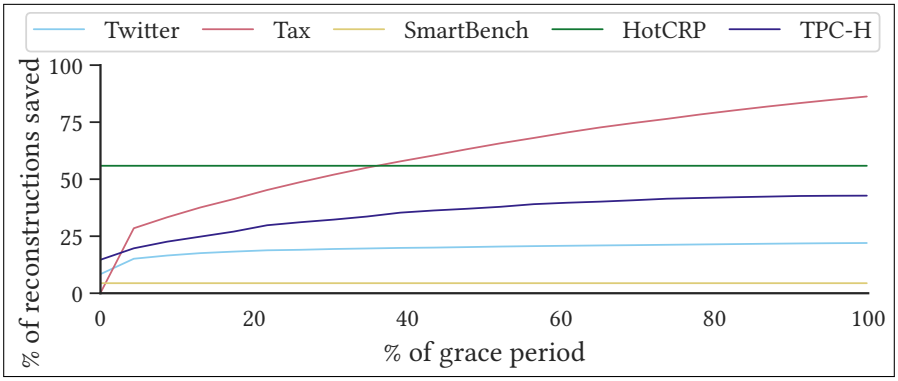


Figure 5.10: Reconstructions saved by SCHEDULER as a function of the grace period.

Experiment 5: Mixed workload. Real systems handle a mixture of on-demand erasure requests and retention-driven expirations. We vary the fraction of erasures handled by SCHEDULER versus on-demand, holding a fixed grace period for both ($\Gamma = 1$ hour for Twitter and Tax, 1 minute for SmartBench, 1 day for HotCRP, and 1 week for TPC-H), with time-based batching on both regimes. Figure 5.11 reports the average number of deleted cells and the necessary reconstructions as the retention-driven share varies. As the share grows, both deletion counts and reconstruction counts decrease. The reduction is largest on datasets with aggregations that span many base cells (HotCRP: 90% deletion saving and 50% reconstruction saving,

moving from 0% to 100% retention-driven). It is smallest on datasets where base data is inserted simultaneously (Tax: no deletion saving, since the base set is the same regardless of timing). Between the extremes, Twitter sees 25–40% deletion savings and 20% reconstruction savings; the savings of SmartBench and TPC-H land in the middle.

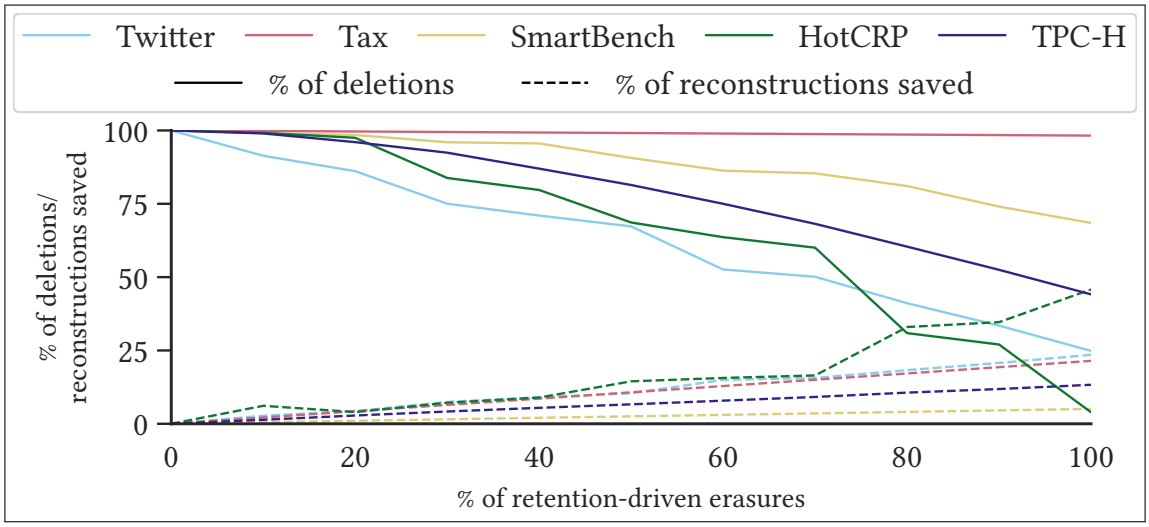


Figure 5.11: P2E2 overhead under varying shares of demand-driven and retention-driven erasures.

Summary Across the five datasets and the experiments, P2E2 erases one to four orders of magnitude fewer auxiliary cells than the cascade baselines on workloads where temporal structure is non-trivial. It also matches the strongest baseline (MinSet) on the one workload (Tax) where the temporal structure is trivial. The gap between MinSet and HGr on Twitter, SmartBench, HotCRP, and TPC-H is the cost of ignoring the κ filter from Theorem 5.2; the equality between them on Tax confirms that the filter is what does the work. Runtime is dominated by RDR instantiation in every configuration; HGr is the practical default, with Apx being preferred for long chains and large batches, and ILP being kept as a sanity check. Batching and retention scheduling compound: a realistic mix reduces deletions by 25–90% and reconstructions by 20–50% on top of the per-request savings, and the per-request overhead remains under 200 milliseconds on the largest workload and slowest mechanism.

5.9 Discussion

In this chapter, we formalized inference-awareness as Pre-insertion Post-erasure Equivalence. We defined the corresponding optimization problem (Opt-P2E2), characterized the new inferences that a P2E2-satisfying mechanism must address (Theorem 5.2), and solved the optimization using three complementary algorithms. The evaluation showed that the auxiliary-deletion cost is one to three orders of magnitude lower than the cascade-style alternatives, with most of the difference being attributable to the temporal filter.

Three assumptions pave the way for future work.

Completeness of dependencies. The framework assumes that the RDR set Δ supplied to the mechanism captures all inference channels of concern. If the analyst omits an RDR that an adversary in practice exploits, P2E2 holds with respect to the declared Δ but does not protect against the missing channel. Dependency discovery is often imperfect, and over-specification produces conservative, excessive deletion while under-specification leaves the deleted data vulnerable.

Acyclicity for exact optimality. OptPath's optimality requires the dependence hypergraph to be acyclic. The normalization of Section 5.6.2 recovers acyclicity in practice (and did so on all evaluated datasets), but on a workload with genuinely cyclic dependencies the normalization may be overly conservative, and the greedy approximation becomes the appropriate fallback with its bounded-ratio guarantee. The complexity-theoretic status of Opt-P2E2 on cyclic hypergraphs remains open.

All-or-nothing dependencies. The third limitation is structural: P2E2 treats every dependency as a binary inference channel. An adversary who observes a new δ either recovers c^* exactly through it (if the inference is deterministic) or recovers nothing (if the channel is blocked). Real-world dependencies are often probabilistic. For example, employees under the

same manager tend to occupy a narrow band of levels. Such dependencies leak information approximately. The deterministic framework either treats them as full-strength dependencies (and over-deletes) or omits them (and under-protects).

A probabilistic relaxation lifts this restriction. Allowing RDRs to carry confidence weights and relaxing the guarantee from a set-containment equivalence to a bound on the adversary's posterior accommodates dependencies that leak partial information without determining the value exactly. The thesis develops such a relaxation in the next chapter.

Chapter 6

Probabilistic Erasure with Weighted Dependencies

6.1 From Deterministic to Probabilistic Framework

In Chapter 5 we presented **P2E2**, an instance of the class of dependency-based inference-aware deletion introduced in Chapter 1, which we call *Semantic Erasure Against Leakage* (SEAL). P2E2 enforces dependencies in a *binary* framework: once a dependency is admitted, it must be blocked completely; if it is pruned, it is ignored. This is limited for dependencies that hold only approximately, and it has two consequences.

The first is *over-deletion under approximate dependencies.* A binary framework must implicitly fix a single threshold on dependency strength. Set the threshold high, and weak but genuine inference channels are dropped, leaking the value the user asked to delete; worse, many weak channels together may reveal the value with high probability, an aggregate that a binary framework cannot measure. Set the threshold low and many marginal dependencies are enforced as if certain, deleting far more data than the risk warrants. No single threshold is at

once safe as well as economical. A deletion that instead reasons about *how much* the retained data raises the probability of re-inference, rather than whether it does so at all, avoids both extremes.

The second is *mask-pattern leakage under value-dependent dependencies*. When a dependency is conditional on the value of the target cell, the cells that must be deleted depend on that value. In the medical example below, an infectious diagnosis is exposed through one set of cells and a metabolic one through another. A mechanism that deletes exactly the cells implicated by the actual value then leaks the value through its own choice of what to delete: the deletion pattern itself becomes a channel for leakage. Deterministic deletion mechanisms, such as those guaranteeing P2E2, sidestep this only by deleting the union of all cells implicated under *any* value, which inflates the number of extra deletions. τ-Seal instead randomizes the deletion pattern so that it reveals little about the deleted value.

τ-Seal addresses both issues. It exploits dependency weights to leave weak channels in place at controlled probabilistic cost, and it bounds the deletion-pattern channel with a differential-privacy-style guarantee. The chapter's two central claims are that the two channels are formally separable and that they compose, on the log-odds scale, into a single re-inference probability threshold τ.

Example 6.1 (Running Example: A Medical Records Instance.)**.** We will use the following database instance that contains medical records as a running example in this chapter.

ID	Zip	Symptom	Treatment	Diagnosis	Age	Gender	BMI	Test
t_1	94022	Fever	Tamiflu	?	45	M	28	72367-6
t_2	94022	Fever	Tamiflu	Flu	32	F	22	72367-6
t_3	92617	Fatigue	Metformin	Diabetes	45	M	28	4548-4
t_4	94301	Cough	Rest	Cold	55	F	30	72367-6
t_5	92617	Nausea	Metformin	Diabetes	45	M	28	4548-4

Figure 6.1: The medical-records instance D; target $c^* = t_1[\mathsf{Diagnosis}]$.

The instance D in Figure 6.1 records medical observations over attributes Zip, Symptom, Treatment, Diagnosis, Age, Gender, BMI, and Test. Here `Flu` and `Cold` are infectious diseases

and `Diabetes` is a metabolic condition.

RDR (σ)	**Specification**	σ
σ_1	Dep (Δ): Diagnosis($\mathbf{X}$) $\not\perp\!\!\!\perp$ Treatment($\mathbf{X}$) Cond (Q): `SELECT rid AS X FROM D`	1
σ_2	Dep (Δ): Diagnosis($\mathbf{X}$) $\not\perp\!\!\!\perp$ Age($\mathbf{X}$), BMI($\mathbf{X}$) Cond (Q): `SELECT rid AS X FROM D` `WHERE Diagnosis IN (Metabolic Conditions)`	0.85

σ_3	Dep (Δ): Diagnosis($\mathbf{X}_1$) $\not\perp\!\!\!\perp$ Diagnosis($\mathbf{X}_2$)
	Cond (Q): `SELECT` t_1`.rid AS` $\mathbf{X}_1$`,` t_2`.rid AS` $\mathbf{X}_2$ `FROM D` t_1`, D` t_2
	`WHERE` t_1`.Symptom =` t_2`.Symptom AND` t_1`.Zip =` t_2`.Zip`
	`AND` t_1`.Diagnosis IN (Infectious Diseases)`
	`AND` t_1`.rid` $\neq$ t_2`.rid`
		0.20

Figure 6.2: Weighted RDRs for the running example.

Three weighted dependencies, written as weighted relational dependency rules (RDRs introduced in Section 6.2), model the inference channels (Figure 6.2). The first, σ_1, is exact and universal: Treatment determines Diagnosis with certainty (weight 1), so t_1[Treatment] = `Tamiflu` reveals the diagnosis outright. The second, σ_2, is conditional on a metabolic diagnosis: there, Age and BMI jointly predict Diagnosis with weight 0.85. The third, σ_3, is conditional on an infectious diagnosis: two tuples sharing Symptom and Zip share Diagnosis with weight 0.20.

Suppose now that we want to erase $c^* = t_1$[Diagnosis] in the above example, whose value is `Flu`. Setting c^* alone to `NULL` does not erase the diagnosis. Through σ_1, the visible Treatment `Tamiflu` still determines it; through σ_3, tuple t_2 shares Symptom and Zip with t_1 and points to the same diagnosis. The dependencies differ in strength (σ_1 at 1, σ_3 at 0.20, nearly fivefold), yet P2E2 treats both of them in the same way. And the channels are value-dependent. If t_1[Diagnosis] was the metabolic value `Diabetes` instead of the value Flu, σ_3 would not be applicable and σ_2 would activate. This would expose the diagnosis through Age and BMI instead of Symptom and Zip. A mechanism that masks the channels of the actual value reveals, by its choice of mask, whether the diagnosis was infectious or metabolic.

Contributions. We make the following contributions in this chapter:

Formal guarantee (Section 6.2). A τ-Seal guarantee parameterized by a single re-inference threshold τ, extending the RDR formalism of Chapter 4 with weights, and a log-odds

decomposition of the post-deletion observation into two leakage channels.

Dependency leakage (Section 6.3). A model of re-inference under a mask that captures transitive chains and aggregates their evidence without overcounting chains that share a masked cell.

Mask-pattern leakage (Section 6.4). A differential-privacy-style guarantee over the mask distribution across neighboring databases, where adjacency follows the value of c^* and the consistency cascades it forces, together with a composition theorem that ties the two channels to τ.

Mechanisms (Section 6.5). Two non-deterministic mechanisms that realize τ-Seal, namely ExpoMech which uses the exponential-mechanism to sample a mask from the set of all masks (mask-space) and Gum which constructs a mask progressively without enumerating the entire mask-space.

Empirical validation (Section 6.6). An extensive evaluation of how accepting bounded re-inference leakage impacts mask-size, leakage, and runtime on five datasets against the strongest deterministic baseline (ILP-based approach in Chapter ILP-based 5). Furthermore, we analyze how a fixed threshold τ should be split between the two leakage channels.

6.2 Weighted Dependencies and the τ-Seal Guarantee

We extend the data model and dependency framework of Chapter 4. We define weighted RDRs, then state the τ guarantee and decompose it into two channels.

Definition 6.1 (Weighted RDR)**.** *A* weighted RDR *is a triple* $\sigma = (\Delta, Q, w_\sigma)$ *over schema* $\mathcal{S}$*, where* $\Delta : A(\mathbf{X}) \not\perp\!\!\!\perp A_1(\mathbf{X}_1), \ldots, A_p(\mathbf{X}_p)$ *is a dependence with Head* $A(\mathbf{X})$ *and Tail* $\{A_1(\mathbf{X}_1), \ldots, A_p(\mathbf{X}_p)\}$*,* Q *is a condition query, and* $w_\sigma \in (0, 1]$ *is the rule weight. The*

semantics are: on a tuple binding returned by Q, an adversary that knows every cell of the instantiation except one recovers the remaining cell's value with probability w_σ.

Setting $w_\sigma = 1$ recovers the unweighted RDR; a hard constraint, key, or schema-level FD has weight 1. Setting $w_\sigma < 1$ encodes an approximate rule: an approximate FD of confidence w, a denial constraint with violation rate $1 - w$, or a similarity rule with empirical re-inference rate w. Instantiation, the indexed set $\hat{\Sigma}(D)$, and the *Head*/*Tail*/*Cells* notation follow Chapter 4, and each instantiation $\hat{\sigma}$ inherits its rule's weight w_σ.

Example 6.2. In Figure 6.2, each tuple instantiates σ_1: on t_1 it gives $\hat{\sigma}_1^{(1)} : t_1[\mathsf{Diagnosis}] \perp \not\perp t_1[\mathsf{Treatment}]$ with weight 1, and likewise for $t_2, \ldots, t_5$. For σ_3, tuples t_1 and t_2 share Symptom and Zip; since $t_1[\mathsf{Diagnosis}] = \texttt{Flu}$ is infectious, the condition holds and yields $\hat{\sigma}_3^{(1,2)} : t_1[\mathsf{Diagnosis}] \perp\!\!\!\not\perp t_2[\mathsf{Diagnosis}]$ with weight 0.20. Were $t_1[\mathsf{Diagnosis}]$ metabolic, σ_3's condition would fail and these instantiations would not exist, while σ_2 would instead instantiate on t_1.

A deletion request for a target cell, denoted c^*, produces an auxiliary set of cells to delete along with it. We refer to this as a deletion mask.

Definition 6.2 (Deletion mask)**.** *Given a target cell c^* in D, a* deletion mask *is a set $M \subseteq D \setminus \{c^*\}$ of auxiliary cells set to* `NULL` *in addition to c^*. A* deletion mechanism *$\mathbb{M} : (D, c^*) \to M$ maps a request to a mask; the post-deletion instance $D \setminus (M \cup \{c^*\})$ sets c^* and every cell of M to* `NULL` *and leaves the rest unchanged. We write $D \setminus M$ for $D \setminus (M \cup \{c^*\})$ when c^* is clear from context.*

In Chapter 5, $\mathbb{M}$ is deterministic and produces a mask M_{det} that deletes all dependent data of c^*, so $D \setminus M_{\mathsf{det}}$ reveals nothing about c^* through the modeled dependencies. In this chapter $\mathbb{M}$ is randomized and may produce a smaller mask.

Formalizing the guarantee. The reference point against which we measure re-inference is the probability of re-inferring the value under deterministic SEAL. After a deterministic mask M_{det}, every dependency channel into c^* is blocked, so the adversary's posterior is restored to background knowledge alone. Let the *baseline re-inference probability*

$$P \;=\; \Pr\big[D[c^*] = v \;\big|\; D \setminus M_{\mathsf{det}}\big]. \tag{6.1}$$

A mechanism that produces a smaller mask leaves some dependent data visible, so the adversary's posterior may exceed P. We define τ-SEAL to bound the excess.

Definition 6.3 (τ-SEAL)**.** *Given a target c^* with value $v = D[c^*]$ and a threshold $\tau \in (0, 1)$, a deletion mechanism $\mathbb{M}$ satisfies τ-SEAL if, for the mask M it produces,*

$$\Pr\big[D[c^*] = v \;\big|\; D \setminus M\big] \;\leq\; \tau. \tag{6.2}$$

The threshold is the maximum probability with which the adversary may correctly re-infer c^* after deletion. Smaller τ gives stronger privacy. For $\tau \leq P$ the inequality forces the adversary back to the baseline and produces deterministic SEAL (with the mask M_{det}); for $\tau > P$ the mechanism may produce smaller masks in exchange for a bounded increase in re-inference leakage.

Problem. Given D, a set Σ of weighted RDRs, a target c^*, and a threshold τ, we seek a randomized mechanism $\mathbb{M} : (D, c^*) \to M$ that satisfies τ-SEAL while minimizing the expected mask size $\mathbb{E}[|M|]$.

Bounding the two leakage channels. An adversary observing $D \setminus M$ learns from two distinct sources (Figure 6.3). The *remaining dependent data* may still support re-inference of c^* through the dependencies; we call this the *dependency leakage* channel. And the *mask pattern*,

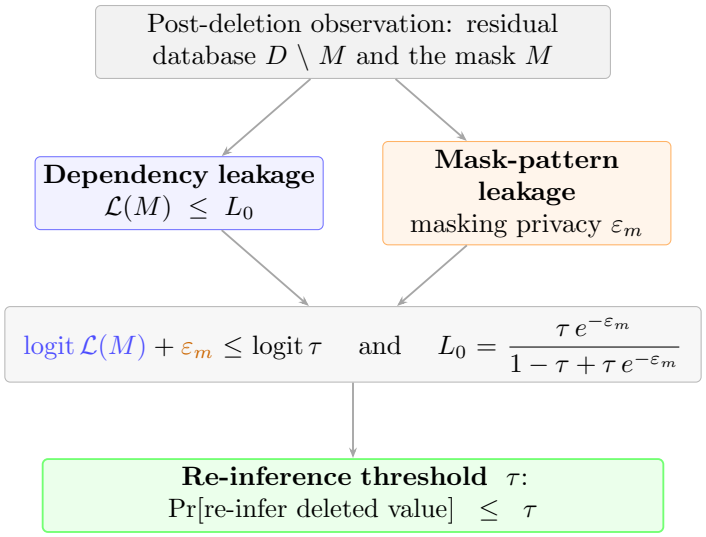


Figure 6.3: The two leakage channels and their composition into τ.

the set of cells now set to `NULL`, may itself reveal c^* when the choice of mask depended on its value; we call this the *mask-pattern leakage* channel. The guarantee τ must bound both.

We work with the *odds* of a correct guess. For event E with $\Pr[E] = p$, write $\mathsf{odds}(E) = p/(1-p)$. By Bayes' theorem, the posterior odds given the observation $D \setminus M$ factor through the baseline:

$$\frac{\Pr[c^* = v \mid D \setminus M]}{\Pr[c^* \neq v \mid D \setminus M]} = \frac{P}{1-P} \times \frac{\Pr[D \setminus M \mid c^* = v]}{\Pr[D \setminus M \mid c^* \neq v]}. \tag{6.3}$$

The mechanism first selects which cells to mask, producing M; the visible values $D_{\mathsf{vis}} = D \setminus (M \cup \{c^*\})$ then follow from M and the underlying data. The chain rule splits the likelihood ratio along exactly the two channels:

$$\frac{\Pr[D \setminus M \mid c^* = v]}{\Pr[D \setminus M \mid c^* \neq v]} = \underbrace{\frac{\Pr[M \mid c^* = v]}{\Pr[M \mid c^* \neq v]}}_{\textit{mask-pattern}} \times \underbrace{\frac{\Pr[D_{\mathsf{vis}} \mid M, c^* = v]}{\Pr[D_{\mathsf{vis}} \mid M, c^* \neq v]}}_{\textit{dependency}}. \tag{6.4}$$

The first factor is the likelihood ratio of *which mask was selected*; we bound it by an ε_m-differential-privacy-style guarantee on the mask distribution (Section 6.4). The second is the likelihood ratio of *the visible values under the selected mask*; we bound it by the dependency

leakage $\mathcal{L}(M)$ developed in Section 6.3.

Taking logarithms turns the product into an additive budget. Writing ε_m for the log-odds gain of the mask-pattern factor and noting that the dependency channel shifts the adversary from the baseline P to $\mathcal{L}(M)$, with log-odds contribution $\mathrm{logit}(\mathcal{L}(M)) - \mathrm{logit}(P)$, Equations (6.3)–(6.4) give

$$\mathrm{logit}(\tau) \;\geq\; \mathrm{logit}(P) + \varepsilon_m + \big(\mathrm{logit}(\mathcal{L}(M)) - \mathrm{logit}(P)\big) \;=\; \varepsilon_m + \mathrm{logit}(\mathcal{L}(M)). \tag{6.5}$$

The baseline log-odds cancel, so the guarantee depends only on the mask-privacy budget ε_m and the dependency leakage $\mathcal{L}(M)$. Inverting, if the mechanism guarantees ε_m-mask privacy and the system fixes a dependency-leakage threshold L_0, the largest threshold it may admit is governed by

$$L_0 \;=\; \frac{\tau\, e^{-\varepsilon_m}}{1 - \tau + \tau\, e^{-\varepsilon_m}}, \qquad \text{equivalently} \qquad \mathrm{logit}(\tau) \;=\; \varepsilon_m + \mathrm{logit}(L_0). \tag{6.6}$$

The user specifies a single threshold τ; given a system-level ε_m, the dependency-leakage threshold L_0 follows from Equation (6.6). Setting $L_0 = 0$ tolerates no dependency leakage and forces the deterministic mask M_{det} to recover SEAL regardless of ε_m. As τ grows, L_0 grows, permitting smaller masks at controlled risks. The mechanism then favors masks with $\mathcal{L}(M) \leq L_0$. Sections 6.3 and 6.4 formalize $\mathcal{L}(M)$ and ε_m in turn, and Section 6.4.4 proves that $\mathcal{L}(M) \leq L_0$ together with ε_m-mask privacy implies τ-SEAL.

6.3 Dependency Leakage

We now formalize the first channel: the probability that the deleted value is re-inferred from the retained database through the weighted dependencies. We represent instantiated RDRs as a weighted dependency hypergraph, define inference chains under a mask, aggregate the chains into a leakage value, and characterize the hardness of computing it exactly.

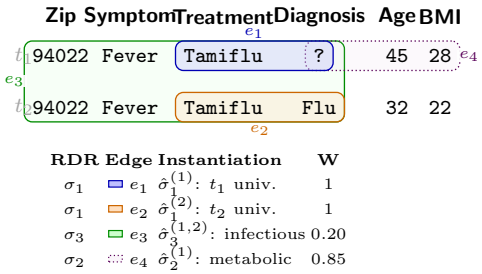


Figure 6.4: Dependency hypergraph $\mathcal{H}_D^{c^*}$ for $c^* = t_1[\mathsf{Diagnosis}]$ partially depicted on $\{t_1, t_2\}$.

6.3.1 Inference Over Weighted Dependency Hypergraph

An instantiated RDR induces an inference relationship among its cells: knowing all but one cell of the instantiation determines the remaining cell with probability equal to the rule's weight. We collect these relationships into a weighted hypergraph.

Definition 6.4 (Dependency hypergraph)**.** *Given weighted RDRs Σ, database D, and target cell c^*, the* dependency hypergraph $\mathcal{H}_D^{c^*} = (V, E)$ *contains one weighted hyperedge $e = Cells(\hat{\sigma})$ with weight w_σ for each instantiated RDR $\hat{\sigma} \in \hat{\Sigma}(D)$ connected to c^*, where $\hat{\sigma}$ is* connected *to c^* if a sequence of instantiations, consecutive ones sharing a cell, links an instantiation containing c^* to $\hat{\sigma}$. The vertex set is $V = \bigcup_{e \in E} e$.*

The hypergraph is undirected, exactly as in Chapter 4: an edge e supports inference of any one of its cells $x \in e$ once all cells in $e \setminus \{x\}$ are known. Inference may be direct, using a single edge to reach c^*, or transitive, using cells inferred earlier to activate later edges.

Figure 6.4 shows the dependency hypergraph for $c^* = t_1[\mathsf{Diagnosis}]$ on tuples t_1 and t_2. The exact rule σ_1 contributes $e_1 = \{t_1[\mathsf{Treatment}], c^*\}$ and e_2 on t_2, both of weight 1. The conditional rules contribute value-dependent edges: e_3 from σ_3 (weight 0.20) appears only when the diagnosis is infectious, and e_4 from σ_2 (weight 0.85, dotted) only when it is metabolic. Which of e_3, e_4 is present therefore depends on the value of c^*.

To illustrate transitive inference, which a two-cell hypergraph cannot exhibit, we use the richer hypergraph $\mathcal{H}_{ex}$ in Figure 6.5, drawn at the attribute level for one tuple. It augments the direct

edges $e_1 = \{\mathsf{Treatment}, \mathsf{Diagnosis}\}$ (weight 1) and $e_2 = \{\mathsf{Age}, \mathsf{BMI}, \mathsf{Diagnosis}\}$ (weight 0.85) with three auxiliary edges $e_a = \{\mathsf{Gender}, \mathsf{Age}\}$, $e_b = \{\mathsf{Gender}, \mathsf{BMI}\}$, and $e_c = \{\mathsf{Age}, \mathsf{Treatment}\}$ among the predictor attributes.

Inference chains under a mask. A mask sets some cells to `NULL`, which changes which edges the adversary can use. An edge e is *active* for inferring $x \in e$ with respect to a knowledge set K if $e \setminus \{x\} \subseteq K$. Starting from the visible cells and repeatedly applying active edges, the adversary recovers masked cells, which may activate further edges. The cells reachable this way form a closure.

Definition 6.5 (Derived visibility)**.** *Given a mask M, let $K_0(M) = V \setminus (M \cup \{c^*\})$ be the initially visible cells, and let*

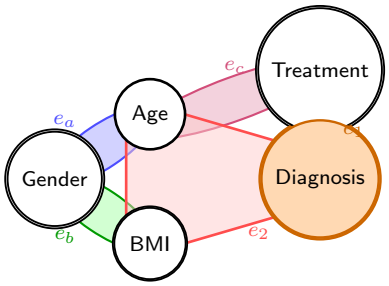
$$K_{t+1}(M) = K_t(M) \cup \big\{x \in V : \exists\, e \in E,\ x \in e,\ e \setminus \{x\} \subseteq K_t(M)\big\}. \tag{6.7}$$

A cell is transitively inferable *under M if it lies in the least fixpoint $K^*(M)$.*

The closure decides *whether* c^* is inferable. An inference chain records *how*: the ordered sequence of edges that one line of reasoning follows to reach c^*.

Definition 6.6 (Inference chain)**.** *An* inference chain *to c^* under mask M is a sequence of distinct hyperedges $p = e_1\text{-}e_2\text{-}\cdots\text{-}e_k$ that can be applied in order so that each e_i infers one new cell x_i, with $x_k = c^*$, each inference enabled by the initially visible cells together with cells inferred earlier in the chain: there exist $K_0(M) \subseteq K_1 \subseteq \cdots \subseteq K_{k-1} \subseteq K^*(M)$ with $K_i = K_{i-1} \cup \{x_i\}$ and $e_i \setminus \{x_i\} \subseteq K_{i-1}$ for every i. Its weight is $w(p) = \prod_{i=1}^{k} w_{e_i}$, and its* masked intermediates *are Masked$(p) = \{x_i : i < k,\ x_i \in M\}$.*

We take chains to be *prefix-minimal*: if some e_i is already active with respect to $K_0(M)$, the shorter suffix $e_i\text{-}\cdots\text{-}e_k$ is used instead. This prevents counting the same re-inference at different lengths. Intuitively, *Masked*(p) records the hidden cells that must be recovered



e_a: {Gender, Age}
e_b: {Gender, BMI}
e_c: {Age, Treatment}
e_1: {Treatment, Diagnosis}
e_2: {Age, BMI, Diagnosis}

Edges	e_a	e_b	e_c	e_1 e_2
Weights	.90	.92	.88	1.0 .85

Mask M	**Cells in** M	**Inf. Chains**	$\mathcal{A}(M)$	$\mathcal{L}(M)$
M_1	$\emptyset$	e_1, e_2	e_1, e_2	1.000
M_2	{Treatment}	e_c–e_1, e_2	e_2, e_c–e_1	.982
M_3	{Treatment, Age}	e_a–e_c–e_1, e_a–e_2	e_a–e_c–e_1	.792
M_4	all four cells	none	$\emptyset$	0

Figure 6.5: Masks on the hypergraph $\mathcal{H}_{ex}$ for c^* = Diagnosis.

before the chain can reach c^*.

Example 6.3. Take the mask M_3 = {Treatment, Age} in Figure 6.5, so $K_0(M_3)$ = {Gender, BMI}. Neither direct edge into Diagnosis is active: e_1 needs the masked Treatment, and e_2 needs the masked Age. Edge e_a needs only Gender, so it infers Age (with probability 0.90); then e_c needs Age, so it infers Treatment (0.88); then e_1 reaches Diagnosis (1.0). The chain e_a-e_c-e_1 has weight $0.90 \cdot 0.88 \cdot 1.0 = 0.792$ and masked intermediates {Age, Treatment}. A second chain e_a-e_2 infers Age and then reaches Diagnosis through e_2 (since BMI is visible), with weight $0.90 \cdot 0.85 = 0.765$ and masked intermediates {Age}.

Let $\mathcal{P}(M)$ be the set of inference chains to c^* under M. Each chain p succeeds when all its edge inferences succeed, with probability $w(p)$, and provides one line of evidence for c^*. If the chains were independent, the aggregate re-inference probability under noisy-OR would be $1 - \prod_{p \in \mathcal{P}(M)}(1 - w(p))$. But chains that share a masked intermediate are not independent: if the adversary fails to recover that hidden cell, every chain through it fails together. Treating them as independent overcounts.

In Example 6.3, both chains for M_3 must first recover Age, so $\mathit{Masked}(e_a\text{-}e_c\text{-}e_1) \cap \mathit{Masked}(e_a\text{-}e_2) =$

$\{\mathsf{Age}\} \neq \emptyset$. They share a single bottleneck and should not be counted as two independent attacks. We therefore aggregate over chains whose masked intermediates do not overlap. Two chains p, p' are *mask-disjoint* if $\mathit{Masked}(p) \cap \mathit{Masked}(p') = \emptyset$.

We build an independent-evidence set greedily, ordering chains by decreasing weight and keeping a chain only if its masked intermediates avoid those of every chain already kept.

Definition 6.7 (Dependency leakage)**.** *Order $\mathcal{P}(M)$ as $w(p_1) \geq w(p_2) \geq \cdots$, and let*

$$\mathcal{A}(M) = \Big\{ p_i : \ \mathit{Masked}(p_i) \cap \bigcup_{\substack{p_j \in \mathcal{A}(M) \\ j<i}} \mathit{Masked}(p_j) = \emptyset \Big\}. \tag{6.8}$$

The dependency leakage *is the noisy-OR over the retained chains,*

$$\mathcal{L}(M, c^*, D) = 1 - \prod_{p \in \mathcal{A}(M)} \big(1 - w(p)\big). \tag{6.9}$$

We write $\mathcal{L}(M)$ when D and c^* are clear. If all cells are masked, no chain remains and $\mathcal{L}(M) = 0$; if $M = \emptyset$, every available chain is active.

Example 6.4 (Aggregation on the running masks)**.** For $M_3 = \{\mathsf{Treatment}, \mathsf{Age}\}$, the two chains of Example 6.3 share Age; the greedy rule keeps only the stronger, e_a-e_c-e_1 (weight 0.792), so $\mathcal{A}(M_3) = \{e_a\text{-}e_c\text{-}e_1\}$ and $\mathcal{L}(M_3) = 0.792$. The naive noisy-OR over both chains would report $1-(1-0.792)(1-0.765) = 0.951$, overstating re-inference by treating the shared Age bottleneck as two independent attacks. By contrast, under $M_2 = \{\mathsf{Treatment}\}$ the chains e_c-e_1 (masked intermediates $\{\mathsf{Treatment}\}$, weight 0.88) and e_2 (no masked intermediates, weight 0.85) are mask-disjoint, so both are kept and $\mathcal{L}(M_2) = 1-(1-0.88)(1-0.85) = 0.982$. Here the noisy-OR is tight, because the two attacks are genuinely independent.

The aggregator sits between the two natural extremes one might use in its place.

Lemma 6.1 (Leakage bounds)**.** *For any D, target c^*, and mask M,*

$$\max_{p \in \mathcal{P}(M)} w(p) \;\le\; \mathcal{L}(M) \;\le\; 1 - \prod_{p \in \mathcal{P}(M)} \big(1 - w(p)\big). \tag{6.10}$$

Proof. Upper bound. $\mathcal{A}(M) \subseteq \mathcal{P}(M)$, and the noisy-OR aggregator $1 - \prod_p (1 - w(p))$ is non-decreasing as chains are added, since each factor $(1 - w(p)) \in [0, 1)$ only shrinks the product. Hence $\mathcal{L}(M) \le 1 - \prod_{p \in \mathcal{P}(M)} (1 - w(p))$.

Lower bound. Let $p^\star \in \arg\max_{p \in \mathcal{P}(M)} w(p)$. The greedy ordering considers $p^\star$ first and admits it, so $p^\star \in \mathcal{A}(M)$ and $\mathcal{L}(M) \ge 1 - (1 - w(p^\star)) = w(p^\star)$. □

The single-best-chain score (the lower bound) is optimistic: it ignores genuinely independent alternative attacks. The noisy-OR over all chains (the upper bound) is pessimistic: it overcounts paths through shared masked bottlenecks. The mask-disjoint aggregator $\mathcal{L}(M)$ captures multiple independent attacks while suppressing the dominant overcounting.

6.3.2 Hardness of the Optimal Aggregation

The greedy rule of Definition 6.7 is a proxy for the ideal aggregator that would retain the *weight-maximizing* mask-disjoint collection. However, finding it is intractable.

Definition 6.8 (Maximum mask-disjoint collection)**.** *Given chains $\mathcal{P}$, each with weight $w(p) \in (0, 1]$ and masked intermediates Masked(p), the* MMDC *problem is to find a pairwise mask-disjoint subset $\mathcal{D} \subseteq \mathcal{P}$ maximizing $1 - \prod_{p \in \mathcal{D}} (1 - w(p))$.*

Theorem 6.2 (Hardness of MMDC)**.** MMDC *is* NP*-hard.*

Proof. We reduce from Independent Set. Let $(G = (V_G, E_G), k)$ be an instance. Construct an MMDC instance with one chain per vertex: for each $v \in V_G$, create p_v with weight $w(p_v) = \frac{1}{2}$ and masked intermediates $\mathit{Masked}(p_v) = \{e \in E_G : v \in e\}$, the edges incident on v.

Two chains p_u, p_v are mask-disjoint iff u and v share no incident edge, that is, iff $\{u, v\} \notin E_G$. A mask-disjoint collection therefore corresponds exactly to an independent set of G. Since all weights are equal, $1 - \prod_{p \in \mathcal{D}}(1 - w(p)) = 1 - 2^{-|\mathcal{D}|}$ is strictly increasing in $|\mathcal{D}|$, so maximizing it is equivalent to maximizing $|\mathcal{D}|$. Hence the constructed instance admits a mask-disjoint collection of size $\geq k$ iff G has an independent set of size $\geq k$. As Independent Set is NP-hard, so is MMDC. □

Computing $\mathcal{L}(M)$ exactly is therefore intractable in general, which is why Definition 6.7 adopts the decreasing-weight greedy collection $\mathcal{A}(M)$ as the operational leakage. It is efficient, admits the strongest single chain (Lemma 6.1), and, as the evaluation in Section 6.6 shows, tracks the pessimistic noisy-OR closely while avoiding its overcounting on the datasets we study.

6.4 Mask-Pattern Leakage

The second channel leaks through the mechanism's *choice* of mask. When the cells implicated by c^* depend on its value, a mechanism that masks exactly those cells reveals the value through the pattern. Bounding this channel requires a value-independent candidate space to sample from (Section 6.4.1), a notion of which databases the mechanism must output similar masks on (Section 6.4.2), and a privacy guarantee with bounded-sensitivity utility (Section 6.4.3). Section 6.4.4 then composes the two channels into τ.

6.4.1 The Maximum Hypergraph and the Inference Zone

The dependency hypergraph $\mathcal{H}_D^{c^*}$ depends on the value of c^*. In Figure 6.4, an infectious value activates e_3 while a metabolic value activates e_4 instead. A mechanism that drew its candidate masks from the value-specific hypergraph would draw from different supports for different values, and the support boundary would leak the value. We avoid this by sampling

from a value-independent space: the union of every edge that could arise under any value of c^*.

Definition 6.9 (Maximum hypergraph)**.** *Given weighted RDRs Σ, database D, and target c^*, the* maximum hypergraph *is*

$$\mathcal{H}^{c^*}_{\max} = \bigcup_{v \in \mathit{dom}(c^*)} \mathcal{H}^{c^*}_{D[c^* \leftarrow v]}, \tag{6.11}$$

where $D[c^* \leftarrow v]$ is D with c^* reset to v, and the union of hypergraphs unions their vertices and edges. The inference zone *is $\mathcal{I}(c^*) = V(\mathcal{H}^{c^*}_{\max}) \setminus \{c^*\}$.*

By construction $\mathcal{H}^{c^*}_{D} \subseteq \mathcal{H}^{c^*}_{\max}$ for every value of c^*, so $\mathcal{I}(c^*)$ depends only on the schema, the RDRs, the tuple identifiers, and the cells outside the zone, never on the value of c^* itself. The inference zone serves three roles. It fixes the candidate mask space $2^{\mathcal{I}(c^*)}$, identical across the possible values of c^*. It delimits the cells that may change when we compare alternative values the cell c^* can possibly take (Section 6.4.2). And it lets an application declare cells that may not be masked, such as data belonging to another user or retained by policy, by excluding them from the zone; the resulting guarantee is then relative to the chosen zone.

Example 6.5 (Building $\mathcal{H}^{c^*}_{\max}$)**.** For $c^* = t_1[\mathsf{Diagnosis}]$ on tuples t_1, t_2 (Figure 6.4), the exact edges e_1, e_2 from σ_1 are present for every value. An infectious value adds e_3 from σ_3; a metabolic value adds e_4 from σ_2. The maximum hypergraph collects all of them, $\{e_1, e_2, e_3, e_4\}$, so the inference zone contains the cells of every edge that could ever be active: $t_1[\mathsf{Treatment}]$, $t_2[\mathsf{Treatment}]$, $t_2[\mathsf{Diagnosis}]$, $t_1[\mathsf{Symptom}]$, $t_1[\mathsf{Zip}]$, $t_2[\mathsf{Symptom}]$, $t_2[\mathsf{Zip}]$, $t_1[\mathsf{Age}]$, and $t_1[\mathsf{BMI}]$. The candidate mask space $2^{\mathcal{I}(c^*)}$ is then the same whether the diagnosis is `Flu` or `Diabetes`, so a mechanism sampling from it has the same support under both.

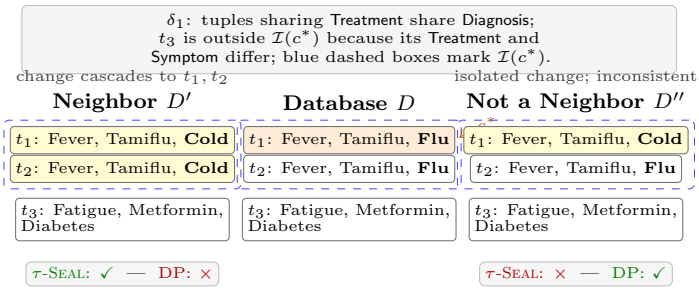


Figure 6.6: Neighboring databases under τ-Seal.

6.4.2 Neighboring Databases

Mask-pattern leakage is measured by comparing the masks a mechanism would produce under the different feasible values of c^*. We require the comparison to range over genuinely feasible alternatives, not arbitrary values. Feasibility of values is determined by the *exact* constraints [13], which are hard consistency constraints that the database must adhere to. Let $\Gamma = \{\delta_1, \ldots, \delta_n\}$ be the set of such constraints, and let $\mathcal{F}_\Gamma$ be the set of database instances satisfying Γ. We define neighboring databases as follows.

Definition 6.10 (Neighboring database)**.** *Given a feasible $D \in \mathcal{F}_\Gamma$, target c^*, and inference zone $\mathcal{I}(c^*)$, a database D' is a* neighbor *of D, written $D \sim_{c^*} D'$, if*

1. $D' \in \mathcal{F}_\Gamma$
2. $D[c^*] \neq D'[c^*]$
3. *for every cell $c \notin \mathcal{I}(c^*) \cup \{c^*\}$, we have $D[c] = D'[c]$*

A neighbor is thus a feasible alternative world in which the target holds a different value. Cells inside $\mathcal{I}(c^*)$ may differ between D and D', but only as part of a consistent completion of the change to c^*. This departs from the standard notion of adjacency in differential-privacy [39], where an arbitrary single-tuple change is a neighbor: here a lone change to c^* can be *invalid* if it breaks consistency with respect to Γ, while a valid neighbor may differ in several cells when changing c^* forces a feasible cascade.

Example 6.6 (Neighbors)**.** In Figure 6.6, $t_1[\mathsf{Diagnosis}] = \texttt{Flu}$ and $\delta_1 \in \Gamma$ requires tuples with the same Treatment to share Diagnosis. The state D' with $t_1[\mathsf{Diagnosis}] = \texttt{Cold}$ is a neighbor only if it stays feasible: since t_2 shares Treatment with t_1 and $t_2[\mathsf{Diagnosis}] \in \mathcal{I}(c^*)$, the change cascades to $t_2[\mathsf{Diagnosis}] = \texttt{Cold}$. The state D'' that changes $t_1[\mathsf{Diagnosis}]$ to Cold while leaving $t_2[\mathsf{Diagnosis}] = \texttt{Flu}$ violates σ_1 and is *not* a neighbor, even though it differs from D only inside the zone. The mechanism must hide c^* across feasible cascades, but need not account for inconsistent states that cannot arise under the declared semantics. Tuple t_3, with a different Treatment and Symptom, lies outside the zone and is identical across all neighbors.

6.4.3 Masking Privacy and Utility

A randomized mechanism controls the mask-pattern channel when its output distribution is stable across neighbors.

Definition 6.11 (ε_m-masking privacy)**.** *A randomized deletion mechanism $\mathbb{M}$ satisfies ε_m-*masking privacy *if, for all neighbors $D \sim_{c^*} D'$ and all measurable sets of masks $\mathcal{O}$,*

$$\Pr[\mathbb{M}(D, c^*) \in \mathcal{O}] \;\le\; e^{\varepsilon_m}\, \Pr[\mathbb{M}(D', c^*) \in \mathcal{O}]. \tag{6.12}$$

To realize this guarantee through the exponential mechanism, we score candidate masks with a utility that rewards small masks and penalizes masks whose dependency leakage exceeds the budget.

Definition 6.12 (Mask utility)**.** *For a mask $M \subseteq \mathcal{I}(c^*)$, dependency-leakage threshold $L_0 \in (P, 1)$, and penalty $\lambda > 0$,*

$$U(M, c^*, D) \;=\; -|M| \;-\; \lambda \cdot \mathbb{1}\big[\mathcal{L}(M, c^*, D) > L_0\big]. \tag{6.13}$$

The first term penalizes auxiliary deletions; the second imposes a flat penalty whenever the

leakage exceeds L_0. Higher utility thus means fewer deletions while staying within the leakage budget, and λ controls how strictly the budget is enforced. The exponential mechanism needs a bound on how much the utility can change across neighbors.

Lemma 6.3 (Utility sensitivity)**.** *For neighbors $D \sim_{c^*} D'$, the following holds:*

$$\Delta U \;=\; \max_{M \subseteq \mathcal{I}(c^*)} \big|U(M, c^*, D) - U(M, c^*, D')\big| \;\le\; \lambda. \tag{6.14}$$

Proof. Fix a mask M. The term $-|M|$ depends only on M and the inference zone, both fixed across neighbors (the zone is determined by $\mathcal{H}^{c^*}_{\max}$, which is value-independent), so it is identical under D and D'. The only data-dependent term is $-\lambda \cdot \mathbb{1}[\mathcal{L}(M, c^*, D) > L_0] \in \{0, -\lambda\}$, whose value can differ across neighbors by at most λ when the indicator flips. Hence $|U(M, c^*, D) - U(M, c^*, D')| \le \lambda$ for every M, and the maximum is at most λ. □

6.4.4 Composing the Two Channels

We combine the two bounds using the following composition theorem.

Theorem 6.4 (Composition)**.** *If a randomized deletion mechanism $\mathbb{M}$ satisfies ε_m-masking privacy and produces a mask M with $\mathcal{L}(M, c^*, D) \le L_0$, then $\mathbb{M}$ satisfies τ-*Seal *with*

$$\tau \;=\; \frac{e^{\varepsilon_m} \cdot L_0}{1 - L_0 + e^{\varepsilon_m} \cdot L_0}, \qquad \textit{equivalently} \qquad \operatorname{logit}(\tau) \;=\; \varepsilon_m + \operatorname{logit}(L_0). \tag{6.15}$$

Proof. By Bayes' theorem and the chain-rule decomposition of Equations (6.3)–(6.4), the posterior odds factor as

$$\frac{\Pr[c^* = v \mid D \setminus M]}{\Pr[c^* \neq v \mid D \setminus M]} = \frac{P}{1-P} \cdot \underbrace{\frac{\Pr[M \mid c^* = v]}{\Pr[M \mid c^* \neq v]}}_{\text{mask-pattern}} \cdot \underbrace{\frac{\Pr[D_{\mathsf{vis}} \mid M, c^* = v]}{\Pr[D_{\mathsf{vis}} \mid M, c^* \neq v]}}_{\text{dependency}}.$$

Mask-pattern factor. Fix any $v' \neq v$ and let D' be a neighbor of D with $D'[c^*] = v'$

(Definition 6.10). By ε_m-masking privacy applied to the singleton $\{M\}$, $\Pr[\mathbb{M}(D, c^*) = M] \leq e^{\varepsilon_m} \Pr[\mathbb{M}(D', c^*) = M]$. Identifying $\Pr[M \mid c^* = v]$ with $\Pr[\mathbb{M}(D, c^*) = M]$ and taking the worst case over $v' \neq v$ bounds the mask-pattern factor by e^{ε_m}.

Dependency factor. By Definition 6.7, the dependency leakage $\mathcal{L}(M, c^*, D)$ upper-bounds the probability that the adversary recovers c^* from the visible cells D_{vis} through the modeled dependencies, so $\Pr[c^* = v \mid D_{\mathsf{vis}}] \leq \mathcal{L}(M) \leq L_0$. In terms of odds, the dependency factor shifts the posterior from the baseline $P/(1-P)$ to at most $L_0/(1-L_0)$, contributing a log-odds term of at most $\operatorname{logit}(L_0) - \operatorname{logit}(P)$.

Combining. Taking logarithms and substituting both bounds,

$$\operatorname{logit}\big(\Pr[c^* = v \mid D \setminus M]\big) \leq \operatorname{logit}(P) + \varepsilon_m + \big(\operatorname{logit}(L_0) - \operatorname{logit}(P)\big) = \varepsilon_m + \operatorname{logit}(L_0).$$

The right-hand side is $\operatorname{logit}(\tau)$ for τ as in Equation (6.15). Since logit is monotone, $\Pr[c^* = v \mid D \setminus M] \leq \tau$, which is τ-Seal (Definition 6.3). □

The theorem gives the user-facing interface to τ-Seal: the user fixes τ; the system picks an (ε_m, L_0) pair on the iso-τ curve of Equation (6.15); the mechanism then guarantees ε_m-masking privacy by design and certifies $\mathcal{L}(M) \leq L_0$ for the mask it outputs. Masking privacy is a property of the mechanism, independent of the realized output, whereas $\mathcal{L}(M) \leq L_0$ is a property of the sampled mask, certified after the fact. Section 6.5 builds two mechanisms that satisfy the first and certify the second.

6.5 Realizing τ-Seal: Two Mechanisms

We present two mechanisms that satisfy τ-Seal. Both share the inference framework of Section 6.3: they construct a dependency hypergraph, enumerate inference chains under a candidate mask, and compute leakage. They differ in how they explore the candidate

space. ExpoMech samples directly from the entire power set $2^{\mathcal{I}(c^*)}$, which suits a moderate inference zone or a workload that can amortize a candidate template. Gum samples from a nested prefix family by using the Gumbel-max trick, avoiding the materialization of $2^{|\mathcal{I}(c^*)|}$ candidates.

6.5.1 Supporting Algorithms

Hypergraph construction. We build a hypergraph for c^* by frontier expansion. Starting from $\{c^*\}$, we repeatedly instantiate the RDRs incident on newly discovered cells and add the resulting weighted hyperedges until no new cell is reached. RDRs are indexed by attribute, so each lookup is constant time for each incident rule, and the construction is linear in the size of the local hypergraph. The construction takes a mode: in Actual mode it evaluates each condition against the current values of D, yielding $\mathcal{H}_D^{c^*}$; in Max mode it admits a condition whenever some assignment of values inside the inference zone could satisfy it, yielding $\mathcal{H}_{\max}^{c^*}$ and hence the inference zone $\mathcal{I}(c^*) = V(\mathcal{H}_{\max}^{c^*}) \setminus \{c^*\}$.

Chain enumeration and leakage. Given a mask M, chain enumeration performs an output-sensitive breadth-first search over partial derivations. A search state pairs an ordered hyperedge sequence with the cells that the adversary currently knows; the search starts from the visible cells $V \setminus (M \cup \{c^*\})$, extends a chain only when a new edge infers a unique missing cell, and records a chain when the inferred cell is c^*. Duplicate states are pruned with hash keys over edge sequences and knowledge sets. Algorithm 5 then evaluates the dependency leakage of Definition 6.7: it weights each chain, greedily keeps a mask-disjoint collection in decreasing-weight order, and applies the noisy-OR aggregation. The number of valid chains can be exponential in the worst case, which is unavoidable for explicit chain-based aggregation, but the process remains a practical one on the datasets of Section 6.6.

Writing $P_M = |\mathcal{P}(M)|$ for the number of chains, ℓ for the maximum chain length, and $T_{\mathcal{L}}$ for the cost of one leakage computation, Table 6.1 depicts the costs of the supporting computations.

Algorithm 5 Dependency leakage by mask-disjoint greedy aggregation (ComputeLeakage).

Input: Mask M, target c^*, hypergraph $\mathcal{H}_D^{c^*} = (V, E)$
Output: Dependency leakage $\mathcal{L}(M, c^*, D)$

$\mathcal{P} \leftarrow \textsc{EnumerateChains}(M, c^*, \mathcal{H}_D^{c^*})$
for all $p \in \mathcal{P}$ **do**
  $w(p) \leftarrow \prod_{e \in p} w_e$; $\mathit{Masked}(p) \leftarrow \{x_i : i < |p|,\ x_i \in M\}$
sort $\mathcal{P}$ by decreasing $w(p)$
$\mathcal{A} \leftarrow \emptyset$; $\mathit{used} \leftarrow \emptyset$
for all $p \in \mathcal{P}$ **do**
  if $\mathit{Masked}(p) \cap \mathit{used} = \emptyset$ **then**
    $\mathcal{A} \leftarrow \mathcal{A} \cup \{p\}$; $\mathit{used} \leftarrow \mathit{used} \cup \mathit{Masked}(p)$
return $1 - \prod_{p \in \mathcal{A}} (1 - w(p))$

Algorithm 6 Exact τ-Seal via the exponential mechanism (ExpoMech).

Input: Database D, target c^*, RDRs Σ, parameters $\varepsilon_m, L_0, \lambda$
Output: Mask M

$\mathcal{H}_{\max}^{c^*} \leftarrow \textsc{ConstructHypergraph}(D, \Sigma, c^*, \textsc{Max})$; $\mathcal{I}(c^*) \leftarrow V(\mathcal{H}_{\max}^{c^*}) \setminus \{c^*\}$
$\mathcal{H}_D^{c^*} \leftarrow \textsc{ConstructHypergraph}(D, \Sigma, c^*, \textsc{Actual})$; $\mathcal{R} \leftarrow 2^{\mathcal{I}(c^*)}$
for all $M \in \mathcal{R}$ **do**
  $\mathcal{L}(M) \leftarrow \textsc{ComputeLeakage}(M, c^*, \mathcal{H}_D^{c^*})$; $U(M) \leftarrow -|M| - \lambda \cdot \mathbb{1}[\mathcal{L}(M) > L_0]$
sample M with $\Pr[M] \propto \exp\big(\varepsilon_m U(M)/(2\lambda)\big)$
return M

Chain enumeration dominates; sorting and greedy selection add $O(P_M \log P_M + P_M|M|)$.

Table 6.1: Asymptotic costs for a target cell c^* ($z = |\mathcal{I}(c^*)|$, $h = |E|$).

Procedure	Time	Space
Hypergraph construction	$O(h)$	$O(z + h)$
Chain enumeration	$O(P_M\, h)$	$O(P_M\, \ell)$
Leakage computation $T_{\mathcal{L}}$	$O(P_M h + P_M \log P_M + P_M\|M\|)$	$O(P_M \ell + \|M\|)$

6.5.2 ExpoMech: Direct Exponential Mechanism

ExpoMech is the direct instantiation of τ-Seal through the exponential mechanism. It uses $\mathcal{H}_{\max}^{c^*}$ to fix the candidate space $\mathcal{R} = 2^{\mathcal{I}(c^*)}$, computes for each candidate $M \in \mathcal{R}$ its leakage on $\mathcal{H}_D^{c^*}$ and utility $U(M)$ from Equation (6.13), and samples M with a probability proportional to $\exp\big(\varepsilon_m U(M)/(2\lambda)\big)$, the exponential mechanism that has sensitivity λ (Algorithm 6).

Theorem 6.5 (Correctness of ExpoMech)**.** ExpoMech *satisfies ε_m-masking privacy. If the sampled mask M satisfies $\mathcal{L}(M, c^*, D) \leq L_0$, then the released $D \setminus M$ satisfies τ-*Seal *with τ as in Equation* (6.15)*.*

Proof. The candidate space $\mathcal{R} = 2^{\mathcal{I}(c^*)}$ is determined by $\mathcal{H}^{c^*}_{\max}$, which by Definition 6.9 is value-independent and hence is identical across neighbors $D \sim_{c^*} D'$. The sampling distribution is therefore supported on the same set under D and D'. By Lemma 6.3 the utility sensitivity is at most λ, so sampling with probability proportional to $\exp(\varepsilon_m U(M)/(2\lambda))$ is the exponential mechanism with sensitivity λ and parameter ε_m, which gives ε_m-masking privacy (Definition 6.11). When the sampled mask further satisfies $\mathcal{L}(M) \leq L_0$, both hypotheses of Theorem 6.4 hold, so the release satisfies τ-Seal. □

Two-phase deployment. ExpoMech enumerates 2^z masks ($z = |\mathcal{I}(c^*)|$), spending $O(2^z T_{\mathcal{L}})$ time and $O(2^z + z + h)$ space to store the candidate utilities. In retention-driven workloads, where the same target attribute is deleted repeatedly, this cost is paid once. An offline phase constructs $\mathcal{H}^{c^*}_{\max}$ and materializes the candidate template; the online phase, on each request, constructs $\mathcal{H}^{c^*}_{D}$, evaluates the candidate utilities, and samples. This moves the combinatorial work off the request path at the cost of candidate storage.

6.5.3 Gum: Progressive Mask Generation

Gum avoids enumerating 2^z masks. It builds a nested candidate family from a value-independent ordering of the inference zone and samples one prefix mask from it. Let $\mathcal{Z} = \mathcal{I}(c^*) \setminus \{c^*\}$. We order $\mathcal{Z}$ by a score that prioritizes cells appearing in many high-weight edges of $\mathcal{H}^{c^*}_{\max}$,

$$\mathsf{score}(c) \;=\; \sum_{e \in E(\mathcal{H}^{c^*}_{\max}):\, c \in e} -\ln\bigl(1 - w_e\bigr), \tag{6.16}$$

Algorithm 7 Progressive nested-mask τ-SEAL via Gumbel-max (GUM).

Input: Database D, target c^*, max hypergraph $\mathcal{H}^{c^*}_{\max}$, parameters $\varepsilon_m, L_0, \lambda$
Output: Mask $M \subseteq \mathcal{I}(c^*)$

$\mathcal{Z} \leftarrow \mathcal{I}(c^*) \setminus \{c^*\}$; $M_0 \leftarrow \emptyset$
compute $\pi = (c_{\pi_1}, \ldots, c_{\pi_z})$ by sorting $\mathcal{Z}$ via Equation (6.16)
for $k = 1, \ldots, z$ **do** $M_k \leftarrow M_{k-1} \cup \{c_{\pi_k}\}$
$\mathcal{H}^{c^*}_D \leftarrow$ CONSTRUCTHYPERGRAPH$(D, \Sigma, c^*,$ ACTUAL$)$
for $k = 0, \ldots, z$ **do**
  $\overline{\mathcal{L}}_k \leftarrow$ COMPUTELEAKAGE$(M_k, c^*, \mathcal{H}^{c^*}_D)$; $U_k \leftarrow -|M_k| - \lambda \cdot \mathbb{1}[\overline{\mathcal{L}}_k > L_0]$
sample $\hat{k}$ with $\Pr[\hat{k} = k] \propto \exp(\varepsilon_m U_k/(2\lambda))$
return $M_{\hat{k}}$

giving $\pi = (c_{\pi_1}, \ldots, c_{\pi_z})$ in decreasing score, and form the nested family $M_0 = \emptyset$, $M_k = M_{k-1} \cup \{c_{\pi_k}\}$. Since π is computed from $\mathcal{H}^{c^*}_{\max}$, the family is identical across neighbors. For each M_k the mechanism computes leakage on $\mathcal{H}^{c^*}_D$ and a utility U_k, then samples k by the exponential mechanism, implemented through the Gumbel-max trick: drawing $g_k \sim \text{Gumbel}(0, b)$ with $b = 2\lambda/\varepsilon_m$ and returning $\arg\max_k(U_k + g_k)$ is equivalent to sampling $\Pr[k] \propto \exp(\varepsilon_m U_k/(2\lambda))$ without computing a normalizer (Algorithm 7).

Theorem 6.6 (Correctness of GUM)**.** GUM *satisfies ε_m-masking privacy. If the sampled mask $M_{\hat{k}}$ satisfies $\overline{\mathcal{L}}(M_{\hat{k}}, c^*, D) \le L_0$, then $D \setminus M_{\hat{k}}$ satisfies τ-*SEAL *with τ as in Equation* (6.15).

Proof. The ordering π is computed from $\mathcal{H}^{c^*}_{\max}$, which is value-independent (Definition 6.9), so the nested family $\{M_k\}_{k=0}^{z}$ is identical across neighbors $D \sim_{c^*} D'$. For each fixed k, $U_k = -|M_k| - \lambda \cdot \mathbb{1}[\overline{\mathcal{L}}(M_k, c^*, D) > L_0]$ has its first term data-independent ($|M_k| = k$) and its second term changing by at most λ across neighbors, so $\Delta U_k \le \lambda$. Gumbel-max sampling with scale $b = 2\lambda/\varepsilon_m$ is the exponential mechanism with sensitivity λ and parameter ε_m over a fixed candidate family, giving ε_m-masking privacy. When the sampled $M_{\hat{k}}$ satisfies the leakage threshold, Theorem 6.4 yields τ-SEAL. □

GUM evaluates only $z + 1$ nested masks, one leakage computation each, for $O(z \log z + z\, T_{\mathcal{L}})$ time and $O(z + h)$ space, against EXPOMECH's $O(2^z\, T_{\mathcal{L}})$ time and $O(2^z + z + h)$ space.

It trades the breadth of the candidate space for tractability: it considers $z+1$ of the 2^z masks but evaluates every one on the actual hypergraph, and its value-independent ordering preserves masking privacy.

Sampling from value-independent mask space. A natural temptation is to restrict the candidate space to masks satisfying $\mathcal{L}(M) \leq L_0$, guaranteeing the leakage threshold by construction. The set of masks meeting the threshold depends on the realized database through $\mathcal{H}_D^{c^*}$, so the feasible set itself varies across neighbors, and the mask pattern could leak through the choice of feasible set. Instead, both mechanisms sample from a value-independent family, the full power set for ExpoMech and the nested prefix family for Gum, biased by utility toward low-leakage, low-deletion masks. The realized leakage is computed afterward and the per-output τ-Seal certification follows when the realization satisfies the threshold. This is expressed in Theorem 6.4: masking privacy is a property of the mechanism, while $\mathcal{L}(M) \leq L_0$ is certified for the sampled mask. When the certification fails, the application falls back to the deterministic mask M_{det}. As we will show, in our evaluations, all masks meet the threshold.

6.5.4 Deploying τ-Seal

We now discuss how an application would deploy τ-Seal. A τ-Seal guarantee is a system-level contract: once the application owner specifies the deletion scope, retention requirements, dependency model, and privacy parameters, the mechanism computes a mask with respect to that specification. This is a similar deployment interface used by existing compliance-driven deletion frameworks [34, 6, 27, 3].

Retention requirements. The application owner declares which attributes are sensitive, which deletion requests are supported, which auxiliary cells may be masked, and which cells must be retained. Retention requirements may arise from legal obligations, auditing needs, or application-level ownership rules [25, 112]. For example, a hospital may allow masking cells

belonging to the requesting patient while disallowing masks over another patient's record or over attributes retained for billing or audit. These choices determine the inference zone used by our mechanisms.

Dependency catalog. The application owner maintains a dependency catalog Σ describing the channels through which deleted values may be re-inferred as weighted RDRs. Domain experts may specify application-specific dependencies, such as medical, financial, or organizational rules. Data processors or administrators may specify ETL, provenance, or data-processing rules. Dependencies may be discovered from data using compliance-motivated frameworks such as [2], or standard dependency-discovery and data-profiling tools [93, 32, 68, 17, 125, 91, 92]. Each dependency is represented as a weighted RDR: the condition query Q specifies where the dependency applies, including value-specific, tuple-specific, and cross-tuple conditions, while the weight specifies its strength as an inference channel.

Weights. Exact dependencies take weight 1; approximate dependencies take a weight that estimates their strength as an inference channel. We estimate the weight of a rule from its *predictive lift beyond a base rate.* For a rule with predicates over at most two tuple variables, we score each inference direction j separately: over sampled tuple pairs with non-null referenced attributes, we estimate the base rate $p_0(e,j) = \Pr[\neg\varphi_j \mid V_e]$ of predicting the head without the premise and the conditional accuracy $p_1(e,j) = \Pr[\neg\varphi_j \mid E_{e,j}]$ with it. The directional weight is the confidence-bounded normalized gain

$$w(e,j) = \max\Big\{0,\ \frac{\mathrm{LCB}(p_1) - \mathrm{UCB}(p_0) - \gamma(e,j)}{1 - \mathrm{UCB}(p_0)}\Big\}, \qquad \gamma(e,j) = \gamma_{\mathrm{frac}}\big(1 - \mathrm{UCB}(p_0)\big), \tag{6.17}$$

which measures the fraction of the residual uncertainty above the base rate that the premise removes, using a lower confidence bound on p_1 and an upper bound on p_0 for robustness. The rule weight is $w_e = \max_j w(e,j)$, and rules with $w_e = 0$ are pruned. The parameter γ_{frac} sets

the stringency: a direction earns positive weight only if it closes at least a γ_{frac} fraction of the gap to perfect prediction, and we use $\gamma_{\text{frac}} = 0.25$ throughout. Minimum-support thresholds on the validity set and the premise guard against weights estimated from rare events.

Leakage budget allocation. The user-facing privacy parameter is the re-inference threshold τ. The application owner allocates this target between dependency leakage L_0 and mask-pattern leakage ε_m. A smaller L_0 forces the mechanism to remove more dependent data; a smaller ε_m makes the mask distribution more stable across feasible alternative values of the target cell, reducing what the deletion pattern itself can reveal.

In practice, the application can choose this allocation by tracing the Pareto curve between auxiliary deletion cost and leakage. The natural operating point is the elbow of the curve: before the elbow, a small relaxation often saves many auxiliary deletions; after it, further relaxation provides little additional utility. In Sec. 6.6, we study this tradeoff empirically and derive a heuristic based on structural properties of the dependency graph, including its density and the number of inference paths. We also verify that the masks prodeced respect the required leakage bound. If no feasible mask exists at a chosen threshold, the application may fall back to the mask that guarantees Seal, which is always guaranteed to exist.

6.6 Evaluating τ-Seal

We evaluate τ-Seal along four questions.

- **Q1:** how much auxiliary deletion does bounded dependency leakage save against the strongest deterministic baseline?
- **Q2:** How do ExpoMech and Gum compare?
- **Q3:** How do L_0 and ε_m affect mask size and realized leakage?

- **Q4:** Given a fixed τ, how should it be split between the two leakage channels?

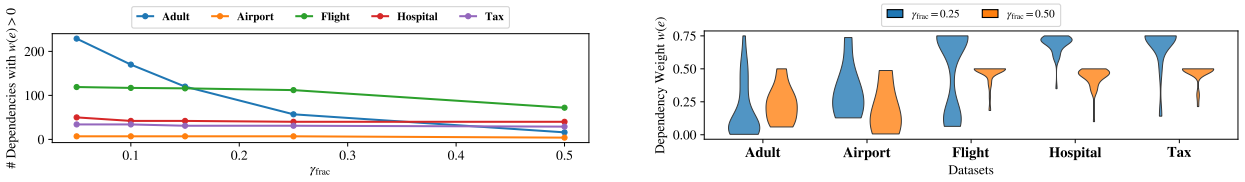
6.6.1 Experimental Setup

Implementation. All experiments run single-threaded on an Apple M4 Pro with 24 GB of RAM under macOS Sequoia 15.1. The mechanisms are implemented in Python 3.13 over data stored in MySQL 9.4.0. The deterministic baseline solves its ILP with Gurobi 13.0 [52]. Each reported point aggregates 100 deletion requests per dataset.

Datasets. We use five datasets that are common in constraint-discovery and data-quality studies, spanning a range of dependency profiles (Table 6.2). Adult [68] has U.S. census demographics, whose constraints capture approximate socio-economic correlations and yield moderate-to-strong inference paths. Airport [94] has airport records whose dependencies are largely structural, induced by geographic hierarchies and identifiers. Hospital [17, 32] has facility descriptors and quality measures, where strong facility and geographic dependencies coexist with weaker ones on derived aggregates. Flight [68] has U.S. domestic flight records, whose identifier, route, and scheduling constraints produce the largest inference neighborhoods. Tax [17, 32] is a synthetic taxpayer dataset widely used to benchmark constraint-based methods.

Baseline. The baseline is MinAlg, the mask-minimizing algorithm of Chapter 5, which produces the minimum-cost deterministic mask M_{det} satisfying Seal by solving an ILP [27]. The mask-size reduction over M_{det} measures how much auxiliary deletion a τ-Seal mechanism saves in exchange for bounded leakage.

Weights. We assign weights by the predictive-lift estimator of Section 6.5.4. Figure 6.7 characterizes the result. As γ_{frac} rises, fewer rules retain positive weight (left), and the reduction is dataset-dependent: Adult prunes substantially from $\gamma_{\text{frac}} = 0.25$ to 0.50, while Hospital barely changes, its surviving rules already clearing a stricter margin. At



(a) Rules retained as γ_{frac} varies.

(b) Weight distribution at $\gamma_{\text{frac}} = 0.25$.

Figure 6.7: Weight estimation across the five datasets.

$\gamma_{\text{frac}} = 0.25$, most retained rules carry modest positive weights, statistically reliable lift over the base rate without approaching certainty, with FLIGHT showing the broadest tail of strong rules (right).

Metrics. We report, per dataset and mechanism, the following metrics:

- *mask-size reduction* M.R. $= 100\,(\mathbb{E}|M_{\mathsf{det}}| - \mathbb{E}|M|)/\mathbb{E}|M_{\mathsf{det}}|$ (where higher values preserve more data);
- realized dependency leakage $\mathcal{L}(M)$ as a percentage, and the mask-pattern budget ε_m (for both, lower is better);
- deletion efficiency $\eta = (1 - \mathcal{L}(M))/(|M| + 1)$, which folds leakage and deletion into one scalar (where higher is better);
- online latency T, offline build time T_{build} for EXPOMECH, and persisted model memory `Mem`;
- two structural quantities, for a hypergraph $\mathcal{H} = (E, V)$, namely, its density $\rho = |E|/|V|$ and the path count $|\Pi|$ that remains after masking.

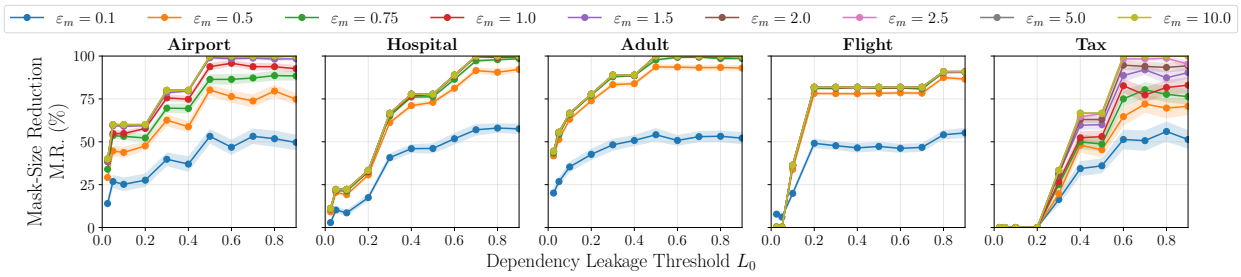
6.6.2 The Leakage–Deletion Tradeoff

Sensitivity to the two parameters. Figures 6.8 and 6.9 sweep $L_0 \in [0.1, 0.9]$ and $\varepsilon_m \in \{0.1, 0.5, \ldots, 10\}$ for EXPOMECH and GUM, reporting mask-size reduction (top) and realized

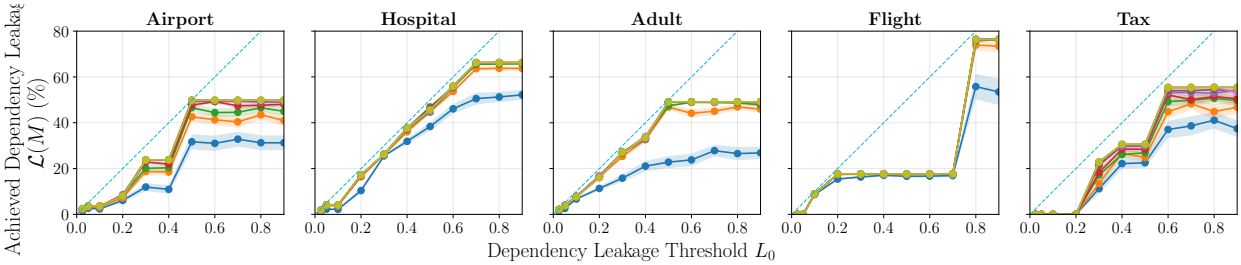
leakage (bottom); the diagonal in the leakage panels marks $\mathcal{L}(M) = L_0$. Increasing L_0 monotonically reduces the mask on every dataset, since a looser leakage budget admits more candidates, but the curve shape is dataset-dependent. AIRPORT saturates by $L_0 = 0.3$ at roughly 40% reduction, its few inference paths already tolerable at modest leakage. HOSPITAL climbs steadily from 8% to 43% through $L_0 = 0.6$; its paths are expensive to suppress, and so they carry leakage as they remain. FLIGHT exceeds 15% leakage even at moderate L_0, but achieves more than 70% reduction with GUM, whose dense structure leaves many active re-inference paths.

The effect of ε_m differs by mechanism. For EXPOMECH, a larger ε_m relaxes mask-pattern privacy and lets the exponential mechanism concentrate probability on high-utility masks, shifting the reduction curves upward and shrinking their variance; on ADULT, mean reduction rises from 35% at $\varepsilon_m = 0.1$ to 89% at $\varepsilon_m = 10$. For GUM the relation is non-monotone because ε_m enters the Gumbel scale at each sequential decision and so amplifies randomness rather than concentrating mass; on several datasets reduction falls as ε_m grows. The useful range for both is $\varepsilon_m \in [0.1, 1.0]$; beyond $\varepsilon_m \approx 2$, EXPOMECH saturates and GUM becomes erratic. The leakage panels confirm that realized $\mathcal{L}(M)$ stays at or below L_0 across the sweep.

Default operating point. We anchor the comparison at the conservative point $\varepsilon_m = 0.1$, $L_0 = 0.2$, $\lambda = 1000$, giving $\tau \approx 0.22$. Table 6.2 reports every metric there. Both mechanisms reduce mask size on every dataset whose inference zone is not already minimal. On FLIGHT ($|\mathcal{I}(c^*)| = 13$), GUM reduces the mask size by 72% at 17% leakage and EXPOMECH by 49% at 15%; on ADULT ($|\mathcal{I}(c^*)| = 12$), GUM reaches 55% at 8% and EXPOMECH 43% at 11%. On the sparse AIRPORT dataset both reach 28–31% at 5–6% leakage. On HOSPITAL, despite a moderate zone, the reductions are smaller (5–17%) because many high-weight dependencies make it difficult to reduce mask-size. On TAX, the deterministic mask already equals the three-cell inference zone, so neither mechanism can improve on it.



(a) Mask-size reduction over MinAlg.



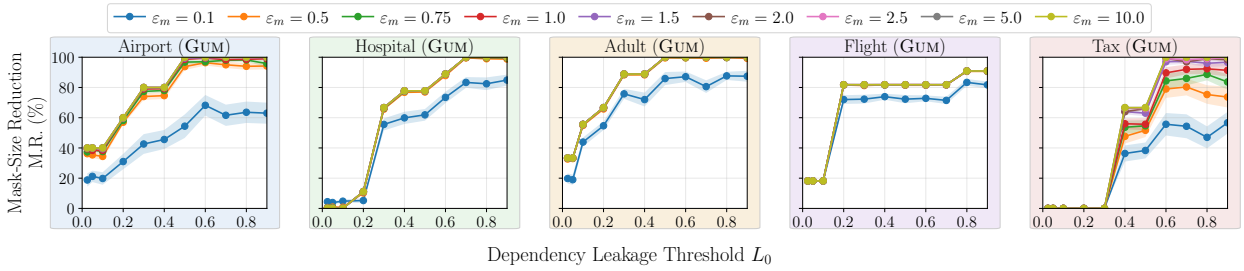
(b) Realized dependency leakage.

Figure 6.8: Sensitivity of ExpoMech to L_0 and ε_m across the five datasets.

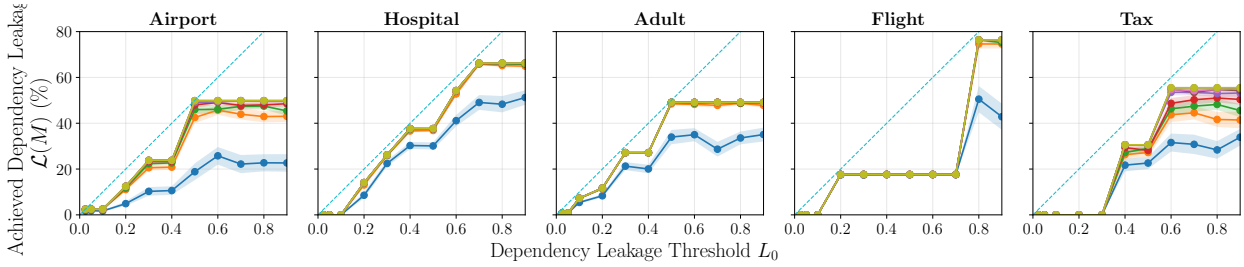
Table 6.2: Per-dataset evaluation at the default operating point ($\varepsilon_m = 0.1$, $L_0 = 0.2$), $\tau \approx 0.22$, over 100 requests. T and T_{build} in ms, `Mem` in KB.

	Dataset			MinAlg				ExpoMech						Gum						ExpoMech
	cells	$\lvert\Sigma\rvert$	ρ	T	`Mem`	η	$\mathcal{L}$	T	`Mem`	η	$\mathcal{L}$	M.R.	$\lvert\Pi\rvert$	T	`Mem`	η	$\mathcal{L}$	M.R.	$\lvert\Pi\rvert$	T_{build}
Airport	495K	7	1.57	10.9	1.1	0.20	0.00	2.4	3.1	0.21	0.06	27.6%	3.1	3.4	0.5	0.23	0.05	31.0%	2.7	5.6
Hospital	1.7M	40	2.46	21.0	2.9	0.10	0.00	2.8	54.7	0.11	0.10	17.4%	43.8	14.4	1.8	0.10	0.09	5.2%	46.4	597.6
Adult	488K	57	4.23	36.0	5.6	0.10	0.00	2.5	231.9	0.15	0.11	42.6%	180.0	57.9	2.9	0.19	0.08	54.7%	214.2	8431.7
Flight	10M	112	3.53	61.0	6.9	0.08	0.00	4.7	473.7	0.14	0.15	49.1%	71.8	423.1	5.9	0.22	0.17	72.0%	172.7	55680.9
Tax	1.4M	31	1.13	7.5	0.6	0.25	0.00	2.6	0.6	0.25	0.00	0.0%	0.7	3.1	0.2	0.25	0.00	0.0%	0.7	4.6

The leakage–deletion frontier. Figure 6.10 plots mean mask size against mean realized leakage, sweeping L_0 at $\varepsilon_m \in \{0.1, 1.0\}$, with MinAlg becoming a single point at $\mathcal{L} = 0$" or "with MinAlg as a single point at $\mathcal{L} = 0$. Both mechanisms trace descending, concave frontiers: a small leakage tolerance buys the largest reduction, and returns diminish thereafter. The knee of the curves is dependent on the structure of the dependency hypergraph. Airport saturates by $L_0 = 0.3$, Hospital and Adult keep benefiting to about $L_0 = 0.5$, and Tax is nearly vertical. On the dense Adult and Flight, the two ε_m curves separate, with more mask-pattern budgets unlocking smaller masks at the same leakage; on the sparse Airport and Tax, they overlap, mask-pattern privacy being the binding constraint there.



(a) Mask-size reduction over MinAlg.



(b) Realized dependency leakage.

Figure 6.9: Sensitivity of Gum to L_0 and ε_m across the five datasets.

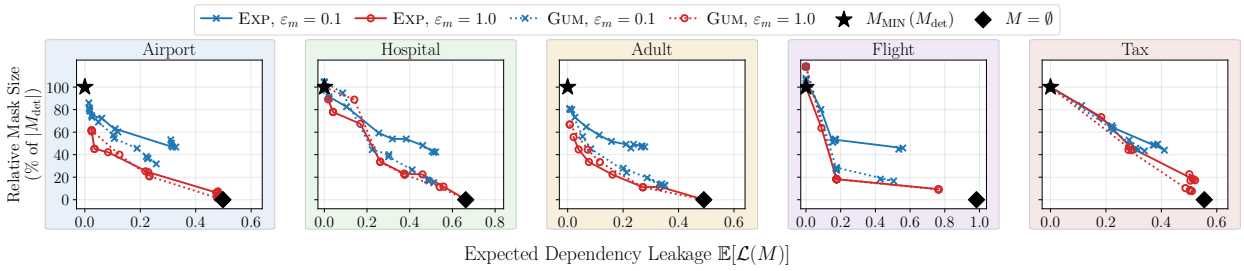


Figure 6.10: Pareto curves for all datasets.

6.6.3 Comparing the Two Mechanisms

Mask quality. The reductions in mask size (listed in Table 6.2) depend on the structure of the inference zone. On datasets with large inference zones, Gum produces smaller masks: 72% versus 49% on Flight, 55% versus 43% on Adult. Its progressive construction adds cells in priority order, which pays off when many cells contribute marginally to the leakage budget. The pattern reverses on Hospital (17% for ExpoMech versus 5% for Gum): the zone is moderate but densely coupled by high-weight edges, and Gum's value-independent ordering does not adapt to the actual hypergraph, missing masks that ExpoMech's global sampling finds. Deletion efficiency η summarizes the joint behavior: Gum leads on Airport,

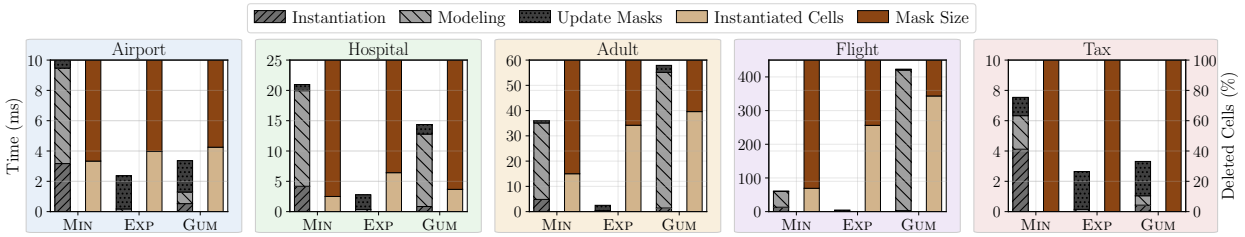


Figure 6.11: Runtime decomposition by dataset and mechanism, mean over 100 requests.

Adult, and Flight (e.g., 0.22 on Flight, nearly triple MinAlg's 0.08), and ExpoMech leads on Hospital. ExpoMech's realized-leakage variance is lower on every dataset, making it the more predictable mechanism when the worst-case behavior matters.

Runtime and build cost. Figure 6.11 decomposes deletion time into instantiation, modeling, and update. The mechanisms embody a classic offline–online tradeoff. ExpoMech completes each online request in 2–5 ms regardless of dataset, because enumerating the $2^{|\mathcal{I}(c^*)|}$ candidates and their utilities happens once in an offline build. That build, T_{build}, scales with the zone, from under 6 ms on Airport and Tax, to 0.6 s on Hospital, 8.4 s on Adult, and 55.7 s on Flight (whose 13-cell inference zone yields 2^{13} candidate masks), and is amortized across repeated deletions of the same attribute. Gum has no offline phase but costs higher per request: 3.4 ms on Airport, 58 ms on Adult, and 423 ms on Flight, the modeling phase dominating on dense zones where every nested candidate needs a fresh leakage evaluation. Against MinAlg's 11–61 ms ILP, ExpoMech's online phase is 4–15× faster and Gum is faster on four of five datasets. Flight has the largest number of dependencies, and computing the leakage takes a long time.

Memory. ExpoMech constructs the candidate set with precomputed utilities ($\sim$ 474 KB on Flight, 232 KB on Adult, and under 5 KB on Airport and Tax); Gum constructs only the static cell ordering ($\sim$ 5.9 KB on Flight and 2.9 KB on Adult), roughly an 80× reduction on the larger datasets. Therefore, ExpoMech trades storage for fast online deletions and suits retention-driven workloads that reuse one target attribute; Gum trades

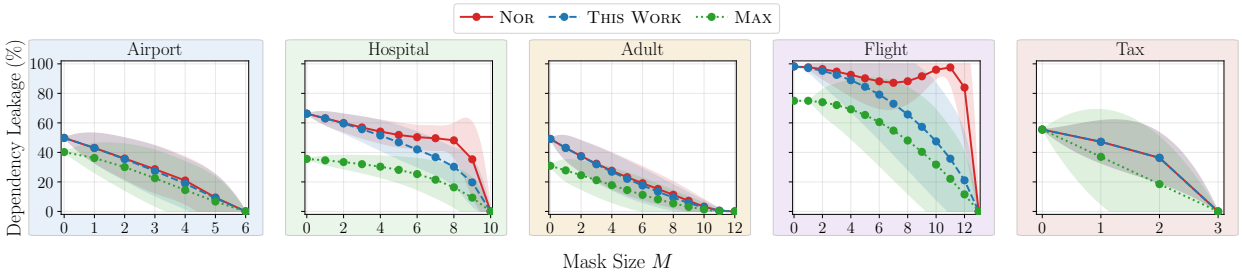


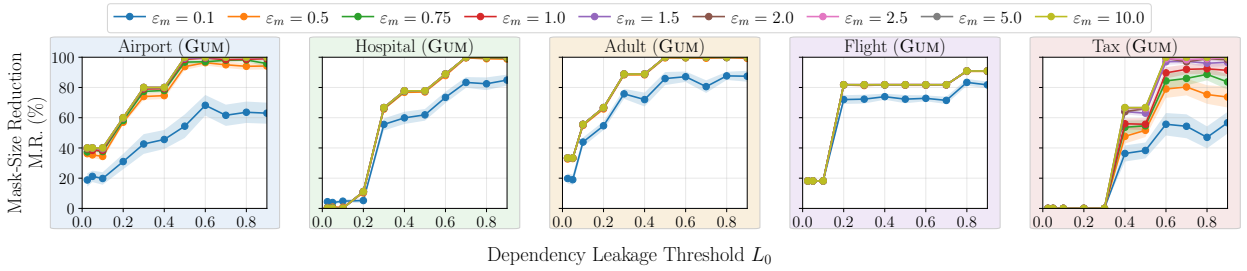
Figure 6.12: The three leakage estimators on $2^{\mathcal{I}(c^*)}$; bands are $\pm 1\sigma$.

computation for a small footprint and suits ad-hoc deletions that are spread across many attributes.

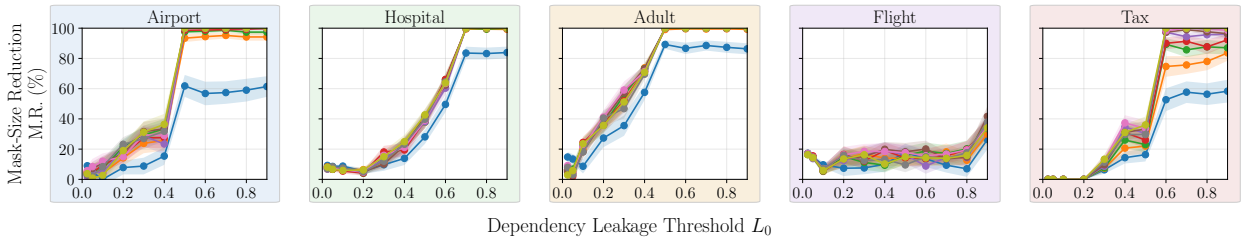
Structural sensitivity. The residual-path count $|\Pi|$ mediates both leakage and runtime. Flight carries the highest realized leakage across all L_0, while Airport and Tax stay low: dense zones keep many channels simultaneously active under partial masking, so each unmasked cell re-exposes several paths. The same density explains Gum's runtime, a near-threshold removal triggering an expensive leakage re-evaluation at each step, which is why Gum is a runtime outlier on Flight (423 ms versus ExpoMech's 4.7 ms) but comparable on sparse Airport (3.4 versus 2.4 ms).

6.6.4 Validating the Leakage Model and the Gum Score

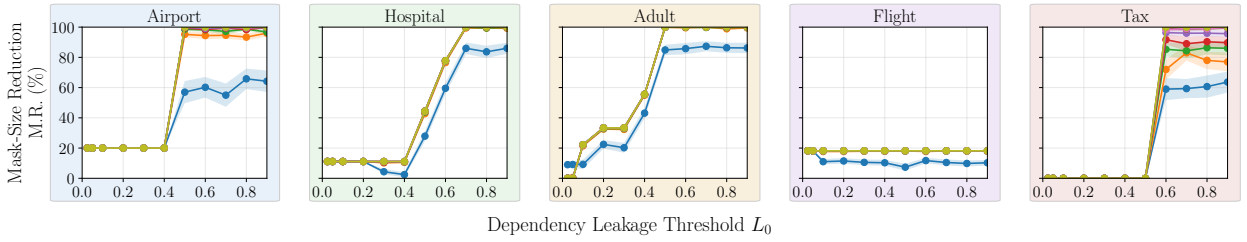
Leakage estimators. Lemma 6.1 bounds $\mathcal{L}(M)$ between the optimistic single-best-chain score and the pessimistic noisy-OR. Figure 6.12 plots all three leakage functions over the candidate space $2^{\mathcal{I}(c^*)}$ for a representative target on each dataset. The bound holds for every mask. On Airport, Adult, and Tax, the noisy-OR nearly coincides with $\mathcal{L}(M)$: chains rarely share masked intermediates there, so the overcounting that noisy-OR is prone to does not arise. On Hospital and Flight, the gap widens, because chains pass through a few shared masked overlapping cells (treatments, geographic hubs) that noisy-OR double-counts. The single-best-chain score is optimistic by a wide margin everywhere, confirming it to be inadequate once more than one independent chain exists.



(a) Proposed weighted score.



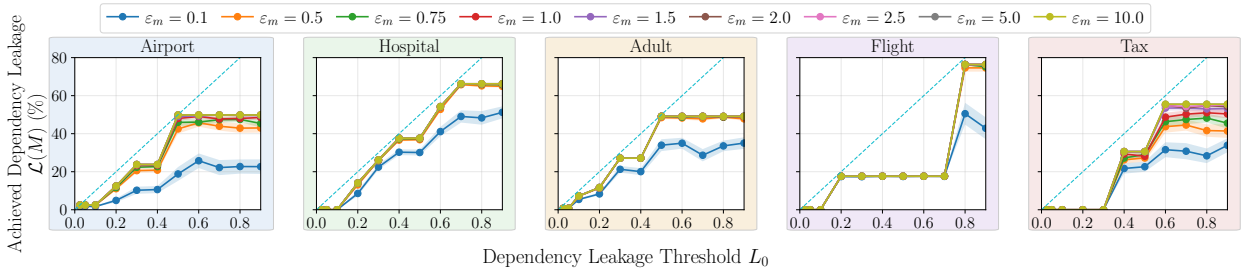
(b) Random ordering.



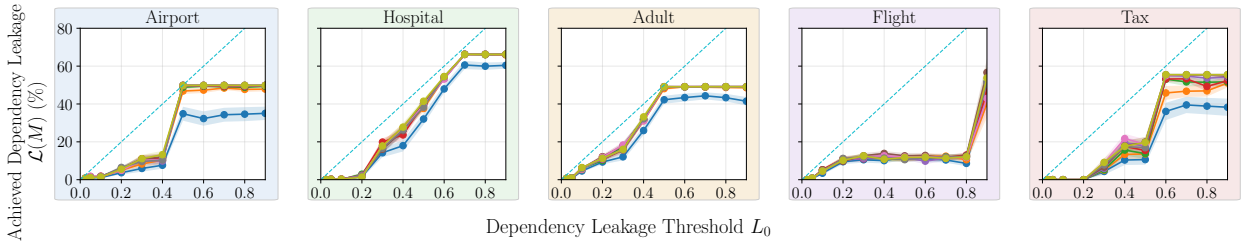
(c) Zero ordering.

Figure 6.13: Ablation of the Gum score: mask-size reduction over MinAlg.

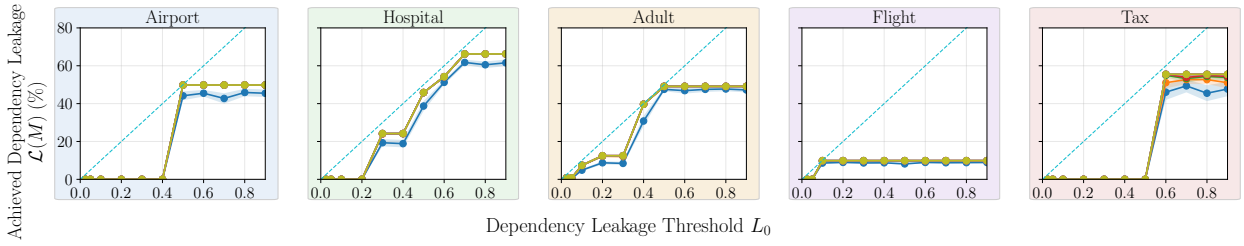
The scoring function in Gum. Gum's cell ordering determines the quality of masks it produces: a poor ordering forces the nested family of masks to become larger through low-utility prefixes. We ablate the score of Equation (6.16) against a uniform random ordering and a zero ordering that ignores the hypergraph. Figures 6.13 and 6.14 report mask-size reduction and realized leakage for the three orderings across all datasets and budgets. The proposed score calibrates leakage closest to L_0 while delivering the largest reduction in size. The zero ordering is entirely insensitive to ε_m, every curve collapsing regardless of budget; the random ordering keeps some sensitivity but with higher variance and weaker calibration. The ordering determined using the maximum-hypergraph is therefore the empirical driver of both leakage calibration and mask efficiency in Gum.



(a) Proposed weighted score.



(b) Random ordering.



(c) Zero ordering.

Figure 6.14: Ablation of the GUM score: realized leakage.

6.6.5 Allocating the Privacy Budget

A fixed re-inference threshold τ admits many (ε_m, L_0) pairs on its iso-τ curve (they add up to the same value), and the previous subsections showed that mask size is sensitive to the budget each leakage channel is assigned. Figure 6.15 shows mask-size reduction over the (ε_m, L_0) plane with iso-τ contours overlaid. The landscape is not aligned with the contours: on FLIGHT under EXPOMECH, the $\tau \approx 0.32$ contour spans reductions from 31% to 81%; on ADULT, the same contour spans 44% to 78%. Two deployments offering the same bound τ can therefore differ by up to fifty points in preserved data, purely through the split.

Figure 6.16 quantifies the best-versus-worst gap at four τ levels. On FLIGHT at $\tau \approx 0.32$, EXPOMECH's best split ($\varepsilon_m = 0.75$, $L_0 \approx 0.20$) reaches 81% while the worst ($\varepsilon_m = 0.1$,

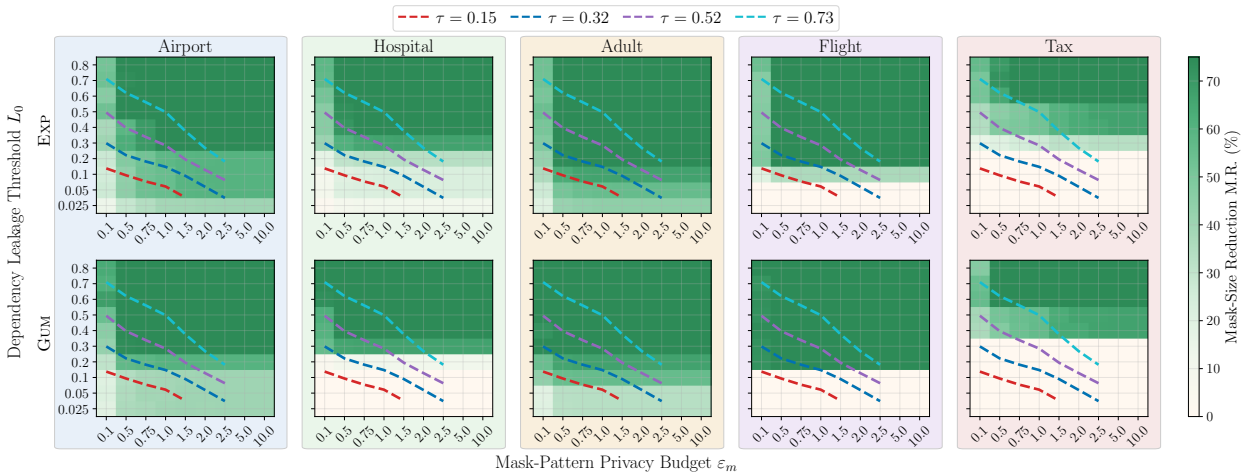


Figure 6.15: Mask-size reduction (%) over MinAlg in the (ε_m, L_0) plane.

Figure 6.16: Best- versus worst-split mask-size reduction over MinAlg at four τ levels.

$L_0 = 0.30$) reaches 49%; for Gum the worst split recovers essentially nothing while the best exceeds 70%. The gap grows with τ, exceeding forty points on Hospital at $\tau \approx 0.73$ for both mechanisms.

A structure-aware heuristic. We observe that the best split of τ between ε_m and L_0 tracks the target cell's dependency structure, which gives a simple heuristic based on the inference hypergraph. When the hypergraph is sparse (low density ρ) and the deterministic mask M_{det} is small, blocking dependency leakage is cheap: a small L_0 costs few deletions, so allocate a

larger masking-privacy ε_m to hide the mask pattern. When ρ is high and M_{det} is large, the opposite holds: blocking every path is costly, so a larger L_0 tolerates bounded dependency leakage and saves deletions, but at the cost of a smaller ε_m. The path count $|\Pi|$ corroborates the choice: many remaining paths mean that the savings came from relaxing the dependency leakage threshold, while few remaining paths suggest that raising L_0 further will not help.

6.6.6 Summary

The evidence is consistent across the five datasets. Bounded dependency leakage saves auxiliary deletion against deterministic SEAL on every dataset, the savings being determined primarily by L_0. The two mechanisms are complementary: EXPOMECH for retention-driven workloads with stable target attributes and tight latency budgets, where its offline template recoups itself; GUM for ad-hoc deletion requests or memory-constrained settings, where its small footprint and zero build cost dominate. The mask-disjoint aggregator stays tight under the shared bottlenecks where noisy-OR overcounts, the scoring function in GUM drives its efficiency, and the composition theorem exposes two interpretable knobs whose best allocation is guided by the density metric ρ and $|\Pi|$

6.7 Discussion

We introduced τ-SEAL, a probabilistic framework that bounds two composable sources of leakage: weighted dependencies and retained data, as well as the mask pattern itself. We developed two mechanisms that explore the tradeoff between re-inference leakage and auxiliary deletion.

Four limitations bound the present framework. First, *weight quality bounds guarantee quality*: a principled calibration methodology for weighted RDRs in deployment remains open. Second, *chain enumeration can be exponential*: it is manageable on the datasets we study, but a worst-case bound would need structural assumptions on the dependency hypergraph. Third, the

(ε_m, L_0) *split is heuristic*: it is derived on the basis of the hypergraph density ρ. A principled optimization using runtime information from the inference zone, perhaps reallocating budget across a batch of requests, would be more robust. Fourth, the *adversary is restricted to the modeled dependencies*; composing τ-SEAL with explicit auxiliary-knowledge models would broaden its reach. These directions notwithstanding, τ-SEAL shows that meaningful deletion need not be all-or-nothing: it quantifies residual leakage and trades it off against deletion cost, without requiring a full joint probabilistic model of the database.

Chapter 7

Grounding Erasure: From Regulation to System Actions

7.1 Motivation: Regulations Mandate Erasure Without Defining It

Recent data regulations, such as the GDPR [87], CCPA [24], and PIPEDA [96], have strengthened individual privacy rights, including the right to erasure. But these regulations are written for legal reasoning and litigation, not for direct implementation in database systems. As a result, they articulate rights and obligations at a high level while leaving the system-level meaning of terms such as deletion, erasure, retention, anonymization, and reconstruction open. They do not specify which database action implements them. The same legal requirement may admit several technically valid readings, and those readings can differ substantially in both cost and guarantee.

We adopted an inference-resilient deletion in this thesis as one such interpretation. Under that interpretation, deleting a value is not merely removing its stored representation or

suppressing it from query results; it also requires controlling what can be inferred about the value from the retained database and from the deletion pattern. P2E2 and Differential-SEAL instantiate this inference-resilient view with different technical guarantees. Whether this is the interpretation that a system should adopt for a particular regulation, application, or risk setting, however, is a separate question. A regulation may also be grounded as physical removal, logical suppression, propagation to derived artifacts, retention enforcement, anonymization, or some combination of these actions.

This chapter studies the gap between regulatory language and system-level actions. We do not claim that a regulation uniquely determines a single deletion semantics. Instead, we ask how a system designer can move from an ambiguous regulatory requirement to an explicit, auditable technical interpretation. The key idea is to make the interpretation step first-class: identify the possible system-level readings of a regulatory concept, choose the reading appropriate for the application and risk setting, formalize it as a precise guarantee, and map that guarantee to concrete system actions. Data-**C**ollection **A**ccess **S**haring **E**rasure (Data-CASE) provides a framework for carrying out this process. It treats compliance not as a direct translation from law to code, but as a structured grounding pipeline from legal concepts to technical semantics to enforceable database mechanisms.

Example 7.1 (The ambiguity of an erasure request)**.** A company, MetaSpace, stores personal data, including user locations for smart-space applications, in PostgreSQL. A user exercises the right of Article 17 and asks that their data be deleted within a reasonable time. The request has several plausible implementations. The system could set the relevant values to *NULL*; it could issue a `DELETE` and reclaim the storage with `VACUUM` [100]; it could write a tombstone, as a log-structured engine does [71, 110], leaving the bytes physically present until compaction; or it could attempt to remove every copy across replicas and caches. Each is a defensible reading of "delete." They differ by orders of magnitude in cost, and they differ in whether the deleted value can still be read, reconstructed, or recovered from disk. Absent

a precise specification, MetaSpace cannot demonstrate that any of them complies, and the ambiguity itself is a source of legal exposure.

The ambiguity in Example 7.1 is the regulatory face of the three-level distinction of Chapter 1. Setting the value to `NULL` achieves logical deletion; reclaiming the bytes achieves physical deletion; neither, by itself, achieves the epistemic deletion that prevents the value from being inferred again. A regulation that says "erase" without choosing among these levels leaves the system to choose, and a system that chooses the weakest level can claim compliance while leaving the information intact.

We resolve this by *grounding*: mapping an ambiguous regulatory concept to a single, unambiguous, system-level interpretation, and then to the concrete actions that implement it. This chapter develops grounding through the example of erasure, drawing on our earlier work [25]. Figure 7.1 shows the pipeline schematically: a regulatory concept admits several interpretations, the system developer selects and formalizes one (the *grounded* concept), and the grounded concept is mapped to the system actions that implement it. We introduce a framework (Section 7.2) in which regulatory requirements can be stated as invariants over data-processing concepts, we ground the erasure concept by enumerating its candidate interpretations and mapping each to system actions (Section 7.3), and we show that the strongest interpretation, the one that forbids reconstruction by inference, is exactly the interpretation that P2E2 and τ-Seal realize. We then illustrate how organizations use the framework to make and defend compliance decisions (Section 7.4), measure the cost of competing interpretations (Section 7.5), and evaluate them in (Section 7.5).

Scope. Erasure is one concept among several that a regulation governs. Our earlier work organizes them as the *Data-CASE* concepts: **C**ollection, **A**ccess, **S**haring, and **E**rasure, together with supporting concepts such as policies and histories. This thesis grounds erasure in depth because erasure is the concept to which it contributes inference-aware semantics.

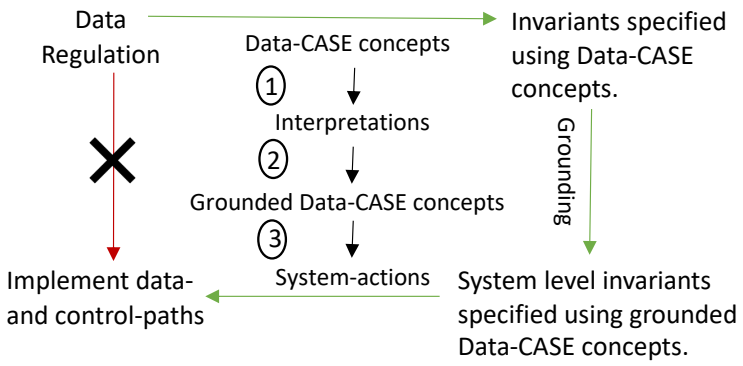


Figure 7.1: Schematic of the Data-CASE grounding pipeline.

The remaining concepts are part of the same program but are not developed here; we return to them as future work in Chapter 8.

7.2 The Data-CASE Model

To state a regulatory requirement precisely enough to check it, we need a vocabulary for the data a system holds, the policies attached to it, the actions performed on it, and the histories those actions leave behind. We introduce that vocabulary here. It is sufficient enough to express requirements as invariants and to make the erasure grounding of Section 7.3 precise. The model is the one we proposed in [25]; we present it in full in this chapter.

Data-CASE groups a regulation's requirements into eight categories: five that track the data life cycle (disclosure, storage, pre-processing, sharing and processing, and erasure) and three that capture system properties (record keeping, obligations, and accountability). Figure 7.2 shows these categories with the GDPR articles that populate each, stated as informal invariants. The model below introduces only the concepts that these requirements quantify over.

Entities, Data Units, and Their Provenance. As data flows through its life cycle, it is collected from a *data-subject* by a *controller*, who may share it with *processors*; and compliance is verified by *auditors*. Data-CASE calls these roles *entities* and denotes one by e. We use a

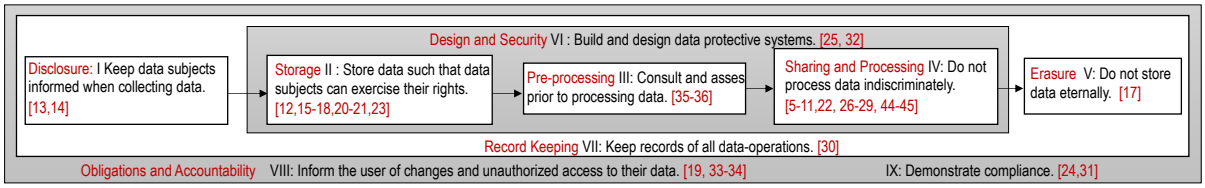


Figure 7.2: GDPR data-governance requirements grouped by Data-CASE category.

running example. Suppose Netflix collects the credit-card information of its subscribers and stores it on a cloud provider.

The finest granularity at which the model refers to data is a *data unit*. What counts as a unit depends on the system, application, and regulation: for a click-stream service, it might be a user's session; for a camera, it might be all the data it captured in an interval; and for Netflix, it is a subscriber's credit-card record. We write a data unit as a tuple

$$X = (S, O, V, P),$$

where S is the data-subject the unit identifies, O is the origin from which the data was collected, $V = \{(v_1, t_1), (v_2, t_2), \ldots\}$ is a set of time-stamped values (v_i is the value at time t_i), and P is the set of policies attached to the unit. A collection of data units is written $\vec{X}$.

Data-CASE classifies data units into three kinds. *Base data* is collected directly or indirectly from data-subjects. *Derived data* is computed from base data; a derived unit has the same four aspects except that its subject and origin are the aggregated (and possibly multiple) subjects and origins of the base units from which it was computed. *Metadata* includes subjects, policies, and the bookkeeping that the model itself needs.

Relation to the cell model. The data unit is a generalization of the cell model presented in Chapter 4 of this thesis. A cell is a data unit at the granularity of a single attribute value: its subject and origin are recorded by the surrounding tuple, its time-stamped values V are exactly the creation and expiration times κ and η that Chapter 5 associates to a cell. The

base/derived classifaction of data units in Data-CASE is the base/derived relation split that the retention-driven mechanism of Chapter 5.7.2 operates on.

Purposes, Policies, and State. Collected data is used for *purposes*: tasks or services that justify processing. A base unit can serve several purposes; Netflix collects a credit-card number for billing and view history for recommendation. A purpose is denoted by p.

The flow of a data unit through a system is governed by *policies*. A policy on a unit X is a tuple $\langle p, e, t_b, t_f \rangle$, a constraint stating that entity e may access X for purpose p from time t_b to time t_f.

Example 7.2 (Policies on a data unit)**.** For the unit $X = \mathsf{credit\_card}$ of subscriber 1234, the policy $\pi_1 = \langle \texttt{billing}, \mathsf{Netflix}, \texttt{010123}, \texttt{010124} \rangle$ states that Netflix may access X for billing from 01/01/23 through 01/01/24. The policy $\pi_2 = \langle \texttt{retention}, \mathsf{Cloud}, \texttt{010123}, \texttt{010124} \rangle$ states that the cloud provider may retain X over the same window.

For a unit X, we write $V(t)$ for its value at time t and $P(t) = \{\langle p, e, t_b, t_f \rangle \in P : t_b \leq t \leq t_f\}$ for the policies in force at t. The *state* of X at time t is $X(t) = (S(t), O(t), V(t), P(t))$, and the state of a database is the collection of the states of its units. An *action* is any operation that changes a unit's state: creation, deletion, a value change, or a read or write of any aspect. We write $a(X)$ for the state of X after action a. Actions on a database D produce a sequence of states $\mathcal{D}_1, \mathcal{D}_2, \ldots$, the temporal sequence of instances that Chapter 5 reasons over.

Histories and Lawful Processing. Regulations require systems to record how data is processed. Each action is logged as an *action-history tuple* $(X, p, e, a(X), t)$, recording that entity e performed action a on X for purpose p at time t. The *action-history* of X, written $\mathcal{H}(X)$, is the set of all such tuples for X.

Example 7.3 (Action history)**.** The tuple (1234, `contract`, Netflix, `collect`, `010223`) records

that on 01/02/23 Netflix contracted to collect the data of subscriber 1234, obtaining consent to set policies π_1 and π_2. The tuple $(X, \texttt{billing}, \mathsf{Netflix}, \texttt{read}(X), \texttt{022623})$ records a later billing read.

Data regulations specify what makes processing lawful. Data-CASE abstracts this as *policy-consistent* processing. An action-history tuple $(X, p, e, a(X), t)$ is *policy-consistent* if there is a policy $\langle p, e, t_b, t_f \rangle \in P(t)$ that authorizes it or if the action is one that a regulation itself requires. Processing of X is policy-consistent if every tuple in $\mathcal{H}(X)$ is also policy consistent.

Stating Regulations as Invariants. With this vocabulary, a regulatory requirement becomes an invariant: a predicate that must hold in every state the system passes through. We illustrate two regulatory requirements, writing $\mathcal{G}i$ for Article i of the GDPR.

Lawfulness ($\mathcal{G}6$). $\mathcal{G}6$ defines when processing personal data is lawful. In Data-CASE, it is exactly policy-consistency: for every data unit X and every action a on X, the action is policy-consistent. The legal grounds and the data-subject's consent are encoded as the purposes carried by the policies in P.

Erasure ($\mathcal{G}17$). $\mathcal{G}17$ requires that personal data be kept no longer than necessary for the purpose for which it was collected, and that it be erased without undue delay. In Data-CASE, for every data unit $X = (S, O, V, P)$ there is a policy $\pi = \langle \texttt{compliance-erase}, e, t_b, t_f \rangle \in P$, and the last action-history tuple on X is $(X, \texttt{compliance-erase}, e, \texttt{erase}(X), t)$ with $t \leq t_f$. The invariant says that every unit carries a compliance deadline and is in fact erased before it.

These invariants capture $\mathcal{G}6$ and $\mathcal{G}17$ formally, but they are not yet executable: the concepts they invoke, *erasure* and *policy*, still admit several valid interpretations. The invariant for $\mathcal{G}17$ mentions $\texttt{erase}(X)$ without saying what erasing X does to the bytes, the query interface, or the inferential trace it leaves. Making the invariant checkable requires fixing that

interpretation.

7.3 Grounding Erasure

Grounding a concept means choosing one interpretation for it, formalizing that interpretation in the model, and mapping it to the system actions that implement it. Once a concept is grounded, an invariant that mentions it becomes a checkable property of a specific system. We ground erasure here because erasure is the concept that this thesis develops and because the grounding exposes precisely where the inference-aware mechanisms of Chapters 5 and 6 fit.

7.3.1 Four Interpretations of Erasure

Example 7.4 (Erasing in PostgreSQL, four ways)**.** Return to MetaSpace and its PostgreSQL store (Example 7.1). The system could (i) add a flag that hides the unit from data-subjects while the controller retains it, (ii) `DELETE` the tuple and `VACUUM` to reclaim its storage, (iii) additionally erase the dependent data and logs from which the unit could be reconstructed, or (iv) additionally apply a drive-level sanitization pass. Each option leaves the system in an observably different state, and each one costs differently.

The four readings in Example 7.4 are the four interpretations that Data-CASE distinguishes:

- *Reversibly inaccessible:* the unit cannot be read by data-subjects but remains accessible to the controller or processor, often recoverable after a specific action.
- *Deleted:* the unit and all its copies are physically erased.
- *Strongly deleted:* the unit is deleted, and so is all dependent data from which the subject is identifiable.
- *Permanently deleted:* the unit is strongly deleted, and an advanced physical sanitization

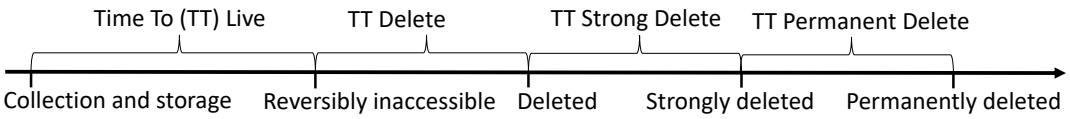


Figure 7.3: The four interpretations of erasure, ordered by strictness.

technique has been applied.

These interpretations are ordered by strictness: permanent delete implies strong delete implies delete. The order induces a notion of how strong a compliance claim is. Some readings protect the subject better than others, but they carry higher overheads and may be infeasible on a given system. Figure 7.3 depicts their temporal relationship.

7.3.2 Three Distinguishing Properties

To ground the interpretations rather than merely name them, we characterize each by three properties of the post-erasure state.

- *Erasure-inconsistent read* (**Illegal Read**, IR). There is an illegal read on X if some tuple $(X, p, e, \texttt{read}(X), t) \in \mathcal{H}(X)$ occurs while $P(t) = \emptyset$: the unit was read with no policy authorizing it.

- *Erasure-inconsistent inference* (**Illegal Inference**, II). There is an illegal inference on X if X has been erased yet $X = f(Y)$ for some other units Y and some dependency f that reconstructs X from Y: the value survives through dependent data, provenance, or correlated units.

- *Transformation invertibility* (**Invertibility**, Inv). To prevent illegal reads and inferences, a unit X is typically transformed to $f(X)$ (an overwrite, an encryption). The transformation may be invertible (recoverable) or not.

Table 7.1 grounds the four interpretations in these properties and maps each to PostgreSQL system actions. The check and cross marks read as "the property is prevented" and "the

Table 7.1: Grounding the four erasure interpretations.

Interpretation	IR	II	Inv	PostgreSQL system action(s)
reversibly inaccessible	✓	×	×	add visibility attribute
delete	✓	×	✓	`DELETE` + `VACUUM`
strong delete	✓	✓	✓	`DELETE` + `VACUUM FULL`; erase dependent logs
permanently delete	✓	✓	✓	not supported (requires drive sanitization)

property is not prevented," respectively. Strong delete and permanent delete share the same property profile; they differ only in that permanent delete adds physical sanitization.

Only the strong and permanent interpretations prevent illegal inference (the II column). The weaker interpretations, including the ordinary `DELETE` + `VACUUM` that a database engineer would reach for first, leave II unaddressed: the deleted value can still be reconstructed from dependent data. This is the regulatory statement of the semantic gap.

7.4 Using the Framework

Grounding turns an ambiguous statement in a data regulation into a decision an organization can make: which interpretation of erasure to support, and which system actions implement it. We illustrate it in the following.

7.4.1 Service Providers and Application Developers

Service providers and application developers have their own system requirements, which constrains the database engine they use. Data-CASE gives them a principled way to identify the data-governance properties they need and to check whether their engine supports them.

Example 7.5 (Choosing an erasure interpretation)**.** MetaSpace (Example 7.1) wishes to offer strong erasure semantics to satisfy $\mathcal{G}17$. It grounds erase in Data-CASE (Section 7.3), reads off the PostgreSQL system actions that implement each interpretation (Table 7.1), and benchmarks them on the customer workload of GDPRBench [112] to learn what each costs

(Section 7.5). The result is sometimes counter-intuitive: `DELETE` + `VACUUM` is no slower than `DELETE` alone on a read-heavy workload, while the strict `DELETE` + `VACUUM FULL` grounding costs far more because it locks the table. The interpretation the organization can afford is therefore a function of its workload mix, and Data-CASE turns the choice into an informed decision rather than a default.

7.4.2 Database Providers

Database providers face the converse problem: how to design or retrofit an engine so that it is compliant. In Data-CASE, a provider expresses the concepts and actions its engine supports in terms of the effects they have on personal data, which fixes the interpretations that the engine can offer. The resulting invariants tell the provider which requirements are already met by the engine and which ones call for retrofitting.

Example 7.6 (Three implementations of compliance)**.** RelDB runs on PostgreSQL and wants to make its service compliant at minimum cost while offering interpretations that are useful to its clients. It implements three groundings of GDPR compliance, which grow increasingly restrictive:

- P_Base: role-based access control through roles and memberships; histories through native `csv` logging with a row-level policy recording query responses; encryption with AES-256; erasure as `DELETE` + `VACUUM`. The least restrictive, it is expected to cost the least.

- P_GBench: policies and metadata are stored in a separate table, so queries join to enforce them; histories by logging all queries and responses; encryption with LUKS (SHA-256); erasure as `DELETE`.

- P_SYS: fine-grained access control, which PostgreSQL does not natively support and which is therefore retrofitted with the Sieve middleware [89]; data and logs encrypted

with AES-128; erasure as `DELETE` + `VACUUM FULL` together with deletion of the data unit's logs.

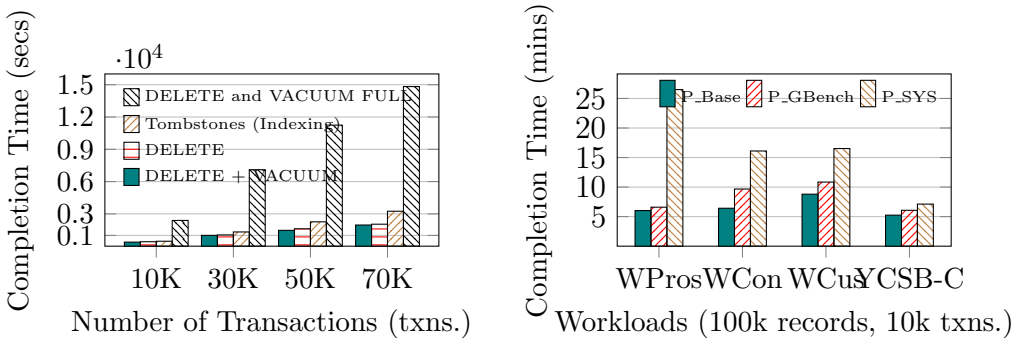
The three differ in code and design because they ground policy, history, and erasure differently. We empirically examine how different groundings affect performance metrics in Section 7.5.

7.4.3 Other Uses

Multinational organizations must reconcile conflicting regulations across jurisdictions; Data-CASE makes the mapping from each regulation's requirements to system actions explicit, which supports decisions about data location, choice of processors, and the features offered to clients. *Regulators and privacy-impact assessment* ($\mathcal{G}35$) need to identify, before deployment, which groundings a system must support to be compliant; and they need to certify, after deployment, that it does. Data-CASE supplies the mapping between grounded concepts and the system actions that realize them, which is the object that such an assessment reasons over.

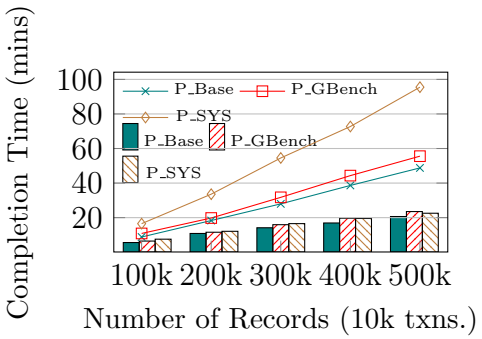
7.5 Evaluating Data-CASE

Experimental setup. We benchmark the three implementations of Example 7.6 using GDPRBench [112] and the Yahoo Cloud Serving Benchmark (YCSB, Workload-C) [35]. GDPRBench supplies three workloads: Controller (WCon: 25% create, 25% delete, 50% metadata update), Processor (WPro: 80% key reads, 20% metadata reads), and Customer (WCus: 20% each of reads, updates, and deletes of data, plus reads and updates of metadata). We enrich the records with the Mall dataset [89], generated from personal devices in a shopping complex with the SmartBench simulator [50]. We report *completion time*, the total time required to finish a workload, and the *space factor*, the ratio of total database size to the size of the personal data it holds, which captures the metadata explosion each grounding induces. All measurements ran on a virtual machine with a six-core AMD Ryzen 5 5600X



(a) Interpretations of Data Erasure in PSQL on WCus

(b) Completion time for workloads



(c) WCus (lines) & YCSB-C(bars) workloads

Figure 7.4: (a) Erasure-grounding cost under WCus. (b) Completion time across the four workloads. (c) Scalability under WCus.

processor, 16 GB of RAM, and PostgreSQL.

Erasure groundings. Figure 7.4(a) isolates the cost of the erasure grounding alone, under WCus. `DELETE + VACUUM` is the cheapest, and no slower than `DELETE` alone: the cost `VACUUM` pays to reclaim storage on the fraction of the workload that is deletion is repaid by faster reads on the rest. The indexed-tombstone grounding that realizes reversible inaccessibility is intermediate, carrying the overhead of maintaining its index. The strict `DELETE + VACUUM FULL` grounding is the most expensive by a wide margin because it locks the table during erasure. Cost broadly rises with strictness, and the strongest interpretation, strong delete, is the costliest to realize through native actions.

Table 7.2: Storage overhead for the three implementations (totals include indices).

System	Personal data (MB)	Metadata (MB)	Total DB (MB)	Space factor
P_Base	7	14	21	3×
P_GBench	7	10	26	3.7×
P_SYS	7	111	120	17.1×

End-to-end overhead. Figure 7.4(b) reports completion time across the four workloads. P_SYS costs more than P_GBench, which costs more than P_Base, as the increasing restrictiveness predicts. The gap is largest on read-heavy WPro, where P_SYS's fine-grained policy checks are exercised most, and on WCon, where create, delete, and update operations each trigger metadata access and logging. The YCSB-C workload, which carries no compliance metadata, completes fastest under every implementation, which shows that the overhead of the compliance machinery is small on operations that do not invoke it. Figure 7.4(c) confirms that the ordering persists as data volume grows: P_SYS is affected most by scale, P_Base is affected the least.

Storage overhead. The groundings also differ in storage. Table 7.2 reports the space factor. The personal data is 7 MB in every case; the metadata is what varies. P_SYS reaches a 17.1× space factor because the Sieve middleware maintains additional metadata and indices, against 3× space factor for P_Base.

Observe that none of the PostgreSQL groundings in Table 7.1 prevents illegal inference without deleting more than the target cell. The only one that achieves II = ✓, the `VACUUM FULL` cascade with log deletion, does so by erasing all dependent data, which both costs the most (Figure 7.4) and deletes more than what the inference channels into the target require.

7.6 Discussion

This chapter makes the case that designing, deploying, and reasoning about the compliance of systems to data regulations like GDPR and others requires a formal framework to express

data processing concepts. The chapter proposes such a formal framework titled Data-CASE and shows how data governing principles of data regulations can be formalized in the form of grounding concepts defined into concrete system-actions and by defining invariants using these concepts. A full realization of our vision of a formal framework to support the development and deployment of compliant systems opens up several challenges:

• **Completeness and correctness:** Demonstrating the correctness of a formal framework like Data-CASE and to what degree it can capture the system requirements of data regulation remain unexplored. This is an important step towards compliance.

• **Grounding concepts:** We focused on erasure and its possible interpretations in systems. Other concepts which capture the requirements of data regulations need to be defined. Once defined, special attention needs to be given to the interactions between and compatibility of different possible interpretations of the concepts.

• **From a formal framework of compliance to a system to support compliance:** Data-CASE supports compliance-related decision-making. Automating this process will enable us to build a comprehensive tool that can be retrofitted on any non-compliant system to make it compliant with a given data regulation. Traditionally, retrofitting for compliance requires multiple tools and often complicates data flow [36].

Chapter 8

Conclusion

Deletion is deceptively simple. At the storage layer, it appears to mean removing bytes, reclaiming pages, or inserting tombstones that will later be compacted away. At the query layer, it appears to mean making a value no longer retrievable through the ordinary database interface. This dissertation has argued that neither view is sufficient for modern semantic databases. Database systems organize data through dependencies, derive new artifacts from it, and expose it through views, caches, logs, statistics, and downstream applications. A value can therefore disappear as a stored representation while remaining present as an inferable fact.

The central claim of this dissertation is that meaningful deletion should be understood as a guarantee about inference. A deletion mechanism should not only remove the target value; it should also control what the retained database and the deletion pattern reveal about that value. This does not mean that a system can entirely "forget" that the value ever existed. Prior knowledge, external correlations, and domain regularities may remain. The operational question is narrower: after the system stores and later deletes a value, does the post-deletion system leave behind additional inferential support for recovering it? The thesis developed

formal frameworks and mechanisms for answering this question.

8.1 Summary of Contributions

The first contribution of the thesis is a formal vocabulary for reasoning about inference-resilient deletion. Chapter 4 introduced a cell-level data model, because the deletion requests studied here often target individual values rather than whole tuples. It then introduced relational dependency rules (RDRs), a SQL-grounded language for describing how cells may carry information about other cells. RDRs provide a common way to represent traditional data dependencies, derived values, aggregate relationships, domain rules, and cell-level semantic dependencies. Instantiating RDRs on a database state yields a dependency hypergraph, and inference over that hypergraph is captured through closure. This offers a verifier for deletion: after a target cell and auxiliary cells are set to `NULL`, is the target still reachable through the residual dependency structure?

The second contribution is a deterministic semantics for deletion under binary dependencies. Chapter 5 developed Pre-insertion Post-Erasure Equivalence (P2E2), an instance of *Semantic Erasure Against Leakage* (Seal). P2E2 uses time as the reference point for deletion. Rather than requiring the system to delete every cell that is dependent on the target, it asks whether the post-erasure dependencies available for inferring the target are no stronger than those that existed when the target value was inserted. This distinction is important. Some dependencies may have existed before the target was inserted, and deleting all dependent data regardless of when the dependency arose can be excessive. P2E2 formalizes semantic rollback: the inference structure around the target is restored to its pre-insertion state.

The chapter also developed mechanisms for realizing P2E2. It characterized the new dependencies that must be addressed by a local condition on insertion time, reducing the comparison between two temporal dependency sets to a traversal of the current instance. It

then gave three approaches for the resulting optimization problem: an ILP encoding, an exact hypergraph algorithm under acyclicity, and a greedy approximation. The evaluation showed that reasoning about when dependencies arise can substantially reduce auxiliary deletion cost relative to cascade-style baselines. Batching demand-driven erasures and scheduling retention-driven reconstruction further reduce the operational overhead.

The third contribution is a probabilistic semantics for deletion under weighted dependencies. Chapter 6 addressed a limitation of the binary framework: inference through semantic dependencies is often not all-or-nothing. Dependencies may be discovered from data, learned statistically, supplied by domain experts, or valid only approximately. Treating such dependencies as either present or absent forces a brittle choice. If they are included, the system may over-delete; if they are ignored, the system may under-protect. Differential-SEAL provides the corresponding probabilistic framework. It assigns to dependencies inferential strength and uses a tunable threshold to bound the adversary's ability to re-infer the deleted value.

A key feature of Differential-SEAL is that it accounts for two leakage channels. The first is the post-deletion visible state: the data that remains may still provide evidence about the target. The second is the deletion pattern: the set of auxiliary cells selected by the mechanism may itself reveal which dependencies were active. Chapter 6 separated these channels, bounded them separately, and composed them on the log-odds scale to obtain an end-to-end recovery bound. It then developed two randomized masking mechanisms, one based on the exponential mechanism and one based on progressive Gumbel-max selection. The evaluation showed that the mechanisms expose an explicit leakage–utility tradeoff and can enforce probabilistic deletion guarantees with a reasonable overhead.

The fourth contribution is a framework for connecting deletion guarantees to regulatory compliance. Chapter 7 developed Data-CASE as a way to ground ambiguous regulatory concepts into precise system actions. This chapter made the interpretation step explicit in operationalizing data regulations. A regulatory concept such as erasure may be grounded

as physical deletion, logical deletion, propagation to derived artifacts, or inference-resilient deletion. Each interpretation has different costs and different guarantees. Data-CASE provides a pipeline from legal concepts to technical semantics to enforceable mechanisms, and positions the inference-resilient guarantees of this thesis as the strongest interpretation of erasure.

8.2 Future Directions

The dissertation opens several directions for future work.

The first is dependency completeness. All guarantees in this thesis are stated relative to a semantic model: an RDR set in the deterministic framework, or a weighted dependency model in the probabilistic framework. If important dependencies are missing, the guarantee may hold formally but fail against an adversary who knows the omitted dependencies. If the model contains many false-positive dependencies or dependencies that do not strongly support re-inference, the mechanism may delete more than what is necessary, leading to a loss of utility. Future work can explore methods for discovering, validating, maintaining, and auditing dependency models for deletion, or explore broadly from the perspective of re-inference. This includes incorporating uncertainty about the dependency model itself and providing meaningful and quantifiable measures.

The second direction is slack outside the declared model. Even a carefully constructed dependency set may not capture every correlation in the data. Weak signals may accumulate, and external information may interact with the database in ways the system does not explicitly model. Differential-SEAL begins to address this issue by moving from binary dependencies to weighted leakage, but further work is needed to reason about unknown or partially specified dependencies. A complete theory of deletion will need to distinguish declared dependencies, learned dependencies, and residual inferential slack that remains outside the system's explicit

model.

The third direction is deletion across data processing pipelines. The thesis focuses on relational databases and derived database artifacts, but modern systems increasingly connect databases to feature stores, vector indexes, machine learning models, dashboards, caches, and agentic workflows. A value deleted from the source database may persist as an embedding, a model update, a cached summary, or an intermediate workflow artifact. Extending RDR-style reasoning to these settings requires richer provenance, node-specific mitigation actions, and planning algorithms that decide whether to mask, regenerate, invalidate, or recompute downstream artifacts.

The fourth direction is deeper system integration. The mechanisms in this thesis can be implemented as middleware or side-car components, but inference resilient deletion would benefit from first-class database support. Query optimizers could maintain dependency indices, storage engines could expose deletion-pattern controls, and transaction managers could coordinate auxiliary deletions with regular operational updates. Such integration would also raise new questions about concurrency, recovery, indexing, and performance isolation: inference-resilient deletion should become a database primitive, not merely an offline cleanup procedure.

The fifth direction is richer policy expression. This thesis studied deletion of target cells under specific semantic references and leakage thresholds. Some deployments may require heterogeneous guarantees: strict deletion for some attributes, probabilistic bounds for others, retention exceptions for legal obligations, and role-dependent visibility for different observers. Future systems should allow application owners to state such policies in a declarative language, compile them into formal guarantees, and then synthesize mechanisms that enforce them with a measurable cost.

8.3 Closing Remarks

The broader message of this dissertation is that deletion is not simply the absence of data. It is a state of the system, and that state must be judged by what it still allows an observer to know about the deleted value. Physical deletion and logical deletion remain essential; without them, no stronger guarantee can be trusted. But in dependency-rich databases, they are not enough. The information carried by a value can survive in dependent cells, derived artifacts, and even in the pattern of deletion itself.

By introducing RDRs, P2E2, Differential-SEAL, and Data-CASE, this dissertation provides a path from the informal request from a user to "delete" their personal data to formal and enforceable system guarantees. It shows that meaningful deletion can be made precise, that it can be implemented with reasonable overhead, and that its relationship to data regulations can be made explicit rather than assumed. In doing so, the thesis reframes deletion as an inference-resilient database guarantee: not as just a question of what has been removed, but as a question of what can still be recovered from what remains.

Bibliography

[1] N. R. Adam and J. C. Worthmann. Security-control methods for statistical databases: a comparative study. *ACM Comput. Surv.*, 21(4):515–556, Dec. 1989.

[2] A. Agarwal, M. George, A. Jeyaraj, and M. Schwarzkopf. Retrofitting gdpr compliance onto legacy databases. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 15(4):958–970, Dec. 2021.

[3] K. D. Albab, I. Sharma, J. Adam, B. Kilimnik, A. Jeyaraj, R. Paul, A. Agvanian, L. Spiegelberg, and M. Schwarzkopf. K9db: Privacy-compliant storage for web applications by construction. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 99–116, Boston, MA, USA, 2023. USENIX.

[4] Apache Software Foundation. Apache Kafka Documentation: Topic Configuration. `https://kafka.apache.org/30/generated/topic_config.html`. Accessed: 2026-06-29.

[5] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '99, page 68–79, New York, NY, USA, 1999. Association for Computing Machinery.

[6] M. Athanassoulis, S. Sarkar, Z. Zhu, and D. Staratzis. Building deletion-compliant data systems. *IEEE Data Engineering Bulletin*, 45(1):21–36, 2022.

[7] M. Barati, O. Rana, I. Petri, and G. Theodorakopoulos. Gdpr compliance verification in internet of things. *IEEE Access*, 8:119697–119709, 2020.

[8] A. Barth, A. Datta, J. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: framework and applications. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15 pp.–198, 2006.

[9] D. Basin, S. Debois, and T. Hildebrandt. On purpose and by necessity: Compliance under the gdpr. In *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 – March 2, 2018, Revised Selected Papers*, page 20–37, Berlin, Heidelberg, 2018. Springer-Verlag.

[10] D. Basin, F. Hublet, S. Krstić, and H. Nguyen. Mechanizing privacy by design. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications*

Security, CCS '25, page 2–5, New York, NY, USA, 2025. Association for Computing Machinery.

[11] J. Bater, X. He, W. Ehrich, A. Machanavajjhala, and J. Rogers. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 12(3):307–320, Nov. 2018.

[12] M. A. Bender, M. Farach-Colton, M. T. Goodrich, and H. Komlós. History-independent dynamic partitioning: Operation-order privacy in ordered data structures. In *Proceedings of the ACM on Management of Data*, volume 2, New York, NY, USA, May 2024. Association for Computing Machinery.

[13] L. Bertossi. *Database Repairing and Consistent Query Answering*. Morgan & Claypool Publishers, 2011.

[14] L. Bertossi. From database repairs to causality in databases and beyond. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems LIV*, volume 14160 of *LNCS*. Springer, 2023.

[15] L. Bertossi, B. Kimelfeld, E. Livshits, and M. Monet. The shapley value in database management. *SIGMOD Rec.*, 52(2):6–17, Aug. 2023.

[16] M. Bertran, S. Tang, M. Kearns, J. Morgenstern, A. Roth, and Z. S. Wu. Reconstruction attacks on machine unlearning: simple models are vulnerable. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, 2024. Curran Associates Inc.

[17] T. Bleifuß, S. Kruse, and F. Naumann. Efficient denial constraint discovery with hydra. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 11(3):311–323, Nov. 2017.

[18] T. Bleifuß, T. Papenbrock, T. Bläsius, M. Schirneck, and F. Naumann. Discovering functional dependencies through hitting set enumeration. *Proc. ACM Manag. Data*, 2(1), Mar. 2024.

[19] P. Bohannon, W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for data cleaning. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 746–755, 2007.

[20] L. Bourtoule, V. Chandrasekaran, C. A. Choquette-Choo, H. Jia, A. Travers, B. Zhang, D. Lie, and N. Papernot. Machine unlearning. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 141–159, 2021.

[21] Brazil. Lei geral de proteção de dados (lgpd) - article 18(iv), 2018. Brazilian Federal Law No. 13,709/2018. Last accessed on 2025-01-10.

[22] A. Broadbent and R. Islam. Quantum encryption with certified deletion. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part III*, page 92–122, Berlin, Heidelberg, 2020. Springer-Verlag.

[23] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, page 150–158, New York, NY, USA, 2002. Association for Computing Machinery.

[24] CCPA. Title 1.81.5. california consumer privacy act of 2018 [1798.100 - 1798.199.100]. `https://leginfo.legislature.ca.gov/faces/codes_displayText.xhtml?division=3.&part=4.&lawCode=CIV&title=1.81.5`, last accessed on 2022-02-01.

[25] V. Chakraborty, S. Ann-Elvy, S. Mehrotra, F. Nawab, M. Sadoghi, S. Sharma, N. Venkatasubramanian, and F. Saeed. Data-case: Grounding data regulations for compliant data processing systems. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 108–115, Paestum, Italy, 2024. OpenProceedings.

[26] V. Chakraborty, Y. Kaminsky, A. A. Dhariya, S. Mehrotra, F. Naumann, and S. Pandey. Inference-aware and privacy-preserving deletion in databases. In *Workshop in Privacy and Security in Databases, SIGMOD*. Association for Computing Machinery, 2026.

[27] V. Chakraborty, Y. Kaminsky, S. Mehrotra, F. Naumann, F. Nawab, P. Pappachan, M. Sadoghi, and N. Venkatasubramanian. Meaningful data erasure in the presence of dependencies. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 18(10):3435–3448, June 2025.

[28] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Found. Trends Databases*, 1(4):379–474, Apr. 2009.

[29] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Information and Computation*, 197(1):90–121, 2005.

[30] R. Chourasia and N. Shah. Forget unlearning: towards true data-deletion in machine learning. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[31] O. Chowdhury, L. Jia, D. Garg, and A. Datta. Temporal mode-checking for runtime monitoring of privacy policies. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, pages 131–149, Cham, 2014. Springer International Publishing.

[32] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 6(13):1498–1509, Aug. 2013.

[33] V. Chvatal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4(3):233–235, Aug. 1979.

[34] K. Cohn-Gordon, G. Damaskinos, D. Neto, J. Cordova, B. Reitz, B. Strahs, D. Obenshain, P. Pearce, and I. Papagiannis. Delf: safeguarding deletion correctness in online social networks. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[36] M. Davari and E. Bertino. Access control model extensions to support data privacy protection based on gdpr. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4017–4024, 2019.

[37] D. E. Denning and P. J. Denning. The tracker: a threat to statistical database security. *ACM Trans. Database Syst.*, 4(1):76–96, Mar. 1979.

[38] J. Domingo-Ferrer, editor. *Inference Control in Statistical Databases: From Theory to Practice*, volume 2316 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2002.

[39] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3–4):211–407, Aug. 2014.

[40] T. Eisenhofer, D. Riepel, V. Chandrasekaran, E. Ghosh, O. Ohrimenko, and N. Papernot. Verifiable and provably secure machine unlearning. In *2025 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*, pages 479–496, 2025.

[41] enryu43. Twitter 100 million tweets dataset, 2023. Accessed: 2025-01-25.

[42] W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2), June 2008.

[43] C. Freire, W. Gatterbauer, N. Immerman, and A. Meliou. New results for the complexity of resilience for binary conjunctive queries with self-joins. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS'20, page 271–284, New York, NY, USA, 2020. Association for Computing Machinery.

[44] J. Gao, S. Garg, M. Mahmoody, and P. N. Vasudevan. Deletion inference, reconstruction, and compliance in machine (un)learning. In *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2022.

[45] S. Garg, S. Goldwasser, and P. N. Vasudevan. Formalizing data deletion in the context of the right to be forgotten. In *Advances in Cryptology – EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II*, page 373–402, Berlin, Heidelberg, 2020. Springer-Verlag.

[46] S. Giannakopoulou, M. Karpathiotakis, and A. Ailamaki. Cleaning denial constraint violations through relaxation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 805–815, New York, NY, USA, 2020. Association for Computing Machinery.

[47] A. Gilad, D. Deutch, and S. Roy. On multiple semantics for declarative database repairs. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management*

of Data, SIGMOD '20, page 817–831, New York, NY, USA, 2020. Association for Computing Machinery.

[48] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '07, page 31–40, New York, NY, USA, 2007. Association for Computing Machinery.

[49] C. Guo, T. Goldstein, A. Hannun, and L. Van Der Maaten. Certified data removal from machine learning models. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[50] P. Gupta, M. J. Carey, S. Mehrotra, and R. Yus. Smartbench: a benchmark for data management in smart spaces. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 13(12):1807–1820, July 2020.

[51] V. Gupta, C. Jung, S. Neel, A. Roth, S. Sharifi-Malvajerdi, and C. Waites. Adaptive machine unlearning. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc.

[52] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2024.

[53] X. He, A. Machanavajjhala, and B. Ding. Blowfish privacy: tuning privacy-utility trade-offs using policies. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1447–1458, New York, NY, USA, 2014. Association for Computing Machinery.

[54] X. Hu and S. Sintos. Finding smallest witnesses for conjunctive queries. In *ACM Transactions on Database Systems*, volume 51, New York, NY, USA, June 2026. Association for Computing Machinery.

[55] F. Hublet, A. Kvamme, and S. Krstić. Towards an enforceable GDPR specification, 2024. arXiv:2402.17350.

[56] F. Hublet, L. Lima, D. Basin, S. Krstić, and D. Traytel. Proactive real-time first-order enforcement. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part II*, page 156–181, Berlin, Heidelberg, 2024. Springer-Verlag.

[57] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. Cords: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 647–658, New York, NY, USA, 2004. Association for Computing Machinery.

[58] G. Jacques-Silva, E. Kalyvianaki, K. Cohn-Gordon, A. Meguid, H. Nguyen, D. Ben-David, C. Nayak, V. Saravagi, G. Stasa, I. Papagiannis, D. Taieb, K. Tamirat, H. Wu, B. Xi, T. Zhang, and Q. Zhou. Unified lineage system: Tracking data provenance at scale. In *Companion of the 2025 International Conference on Management of*

Data, SIGMOD/PODS '25, page 457–470, New York, NY, USA, 2025. Association for Computing Machinery.

[59] N. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for sql queries. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 11(5):526–539, Oct. 2018.

[60] Y. Kaminsky, E. H. M. Pena, and F. Naumann. Incremental detection of denial constraint violations. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 18(4):1000–1012, Dec. 2024.

[61] D. Kifer and A. Machanavajjhala. No free lunch in data privacy. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 193–204, New York, NY, USA, 2011. Association for Computing Machinery.

[62] D. Kifer and A. Machanavajjhala. Pufferfish: A framework for mathematical privacy definitions. *ACM Trans. Database Syst.*, 39(1), Jan. 2014.

[63] B. Kimelfeld. A dichotomy in the complexity of deletion propagation with functional dependencies. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, page 191–202, New York, NY, USA, 2012. Association for Computing Machinery.

[64] D. Klein, B. Rolle, T. Barber, M. Karl, and M. Johns. General data protection runtime: Enforcing transparent gdpr compliance for existing applications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 3343–3357, New York, NY, USA, 2023. Association for Computing Machinery.

[65] E. Kohler. Hotcrp: Conference review system, 2024. Accessed: 2025-01-25.

[66] I. Kotsogiannis, Y. Tao, X. He, M. Fanaeepour, A. Machanavajjhala, M. Hay, and G. Miklau. Privatesql: a differentially private sql query engine. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 12(11):1371–1384, July 2019.

[67] T. Kraska, M. Stonebraker, B. L. Michael, S. Sacha, and W. J. Daniel. Schengendb: A data protection database proposal. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare - VLDB 2019 Workshops, Poly and DMAH, Los Angeles, CA, USA, August 30, 2019, Revised Selected Papers*, volume 11721 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2019.

[68] S. Kruse and F. Naumann. Efficient discovery of approximate dependencies. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 11(7):759–772, Mar. 2018.

[69] M. Kuperberg. Towards enabling deletion in append-only blockchains to support data growth management and gdpr compliance. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 393–400, 2020.

[70] M. Kurmanji, E. Triantafillou, and P. Triantafillou. Machine unlearning in learned databases: An experimental analysis. *Proc. ACM Manag. Data*, 2(1), Mar. 2024.

[71] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[72] C. Liu, S. Chakraborty, and P. Mittal. Dependence makes you vulnerable: Differential privacy under dependent tuples. In *NDSS*, volume 16, pages 21–24, San Diego, CA, 2016. The Internet Society.

[73] A. Lopatenko and L. Bertossi. Complexity of consistent query answering in databases under cardinality-based and incremental repair semantics. In *Proceedings of the 11th International Conference on Database Theory*, ICDT'07, page 179–193, Berlin, Heidelberg, 2007. Springer-Verlag.

[74] W. Lu and G. Miklau. Auditguard: A system for database auditing under retention restrictions. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1(2):1484–1487, aug 2008.

[75] N. Makhija and W. Gatterbauer. A unified approach for resilience and causal responsibility with integer linear programming (ilp) and lp relaxations. *Proc. ACM Manag. Data*, 1(4), Dec. 2023.

[76] N. Makhija and W. Gatterbauer. Is integer linear programming all you need for deletion propagation? a unified and practical approach for generalized deletion propagation. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 18(8):2667–2680, Apr. 2025.

[77] A. Martin, E. C. de Almeida, O. Romero, and A. Queralt. How and why false denial constraints are discovered. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 18(10):3477–3489, June 2025.

[78] V. Mayer-Schönberger. *Delete: The Virtue of Forgetting in the Digital Age*. Princeton University Press, 2009.

[79] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '07, page 94–103, USA, 2007. IEEE Computer Society.

[80] S. Mohapatra, A. Gilad, X. He, and B. Kimelfeld. Computing inconsistency measures under differential privacy. *Proc. ACM Manag. Data*, 3(3), June 2025.

[81] MongoDB. Ttl indexes — mongodb manual, 2026. `https://www.mongodb.com/docs/current/core/index-ttl/`, accessed 2026-03-05.

[82] MySQL. Mysql triggers. `https://dev.mysql.com/doc/refman/9.0/en/trigger-syntax.html`, 2019. Accessed:2025-01-10.

[83] M. Naor and V. Teague. Anti-persistence: history independent data structures. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, STOC '01, page 492–501, New York, NY, USA, 2001. Association for Computing Machinery.

[84] W. Ni, X. Miao, X. Zhao, Y. Wu, S. Liang, and J. Yin. Automatic data repair: Are we ready to deploy? *Proceedings of the International Conference on Very Large Databases (VLDB)*, 17(10):2617–2630, June 2024.

[85] F. Nietzsche. *On the Genealogy of Morals*. Vintage Books.

[86] F. Nietzsche. On the uses and disadvantages of history for life. In D. Breazeale, editor, *Untimely Meditations*, Cambridge Texts in the History of Philosophy, pages 57–124. Cambridge University Press, Cambridge, 1997. Translated by R. J. Hollingdale. Originally published 1874.

[87] P. Office. Regulation (eu) 2016/679, 2019. `https://eur-lex.europa.eu/eli/reg/2016/679/oj`, last accessed on 2021-10-11.

[88] Oracle. MySQL 8.4 Reference Manual: Purge Configuration. `https://dev.mysql.com/doc/refman/8.4/en/innodb-purge-configuration.html`. Accessed: 2026-06-29.

[89] P. Pappachan, R. Yus, S. Mehrotra, and J.-C. Freytag. Sieve: a middleware approach to scalable access control for database management systems. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 13(12):2424–2437, July 2020.

[90] P. Pappachan, S. Zhang, X. He, and S. Mehrotra. Don't be a tattle-tale: Preventing leakages through data dependencies on access control protected data. *Proc. VLDB Endow.*, 15(11):2437–2449, 2022.

[91] M. Parciak, S. Weytjens, N. Hens, F. Neven, L. M. Peeters, and S. Vansummeren. Measuring approximate functional dependencies: A comparative study. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 3505–3518. IEEE Computer Society, 2024.

[92] E. H. M. Pena, E. C. de Almeida, and F. Naumann. Discovery of approximate (and exact) denial constraints. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 13(3):266–278, Nov. 2019.

[93] E. H. M. Pena, F. Porto, and F. Naumann. Fast algorithms for denial constraint discovery. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 16(4):684–696, Dec. 2022.

[94] E. H. M. Pena, F. Porto, and F. Naumann. Discovering denial constraints in dynamic datasets. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 3546–3558, 2024.

[95] F. Pennerath, P. Mandros, and J. Vreeken. Discovering approximate functional dependencies using smoothed mutual information. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 1254–1264, New York, NY, USA, 2020. Association for Computing Machinery.

[96] PIPEDA. Personal information protection and electronic documents act (s.c. 2000, c. 5). `https://laws-lois.justice.gc.ca/ENG/ACTS/P-8.6/index.html`, last accessed on 2022-02-01.

[97] Plato. Meno. In J. M. Cooper and D. S. Hutchinson, editors, *Plato: Complete Works*, pages 870–897. Hackett Publishing, Indianapolis, IN, 1997. Translated by G. M. A. Grube.

[98] PostgreSQL Global Development Group. Postgresql documentation: Concurrency control — introduction. `https://www.postgresql.org/docs/current/mvcc-intro.html`, 2026. Section 13.1, current documentation; accessed March 5, 2026.

[99] PostgreSQL Global Development Group. Postgresql documentation: Constraints — foreign keys. `https://www.postgresql.org/docs/current/ddl-constraints.html`, 2026. Section 5.5.5, current documentation; accessed March 5, 2026.

[100] PostgreSQL Global Development Group. Postgresql documentation: Routine vacuuming. `https://www.postgresql.org/docs/current/routine-vacuuming.html`, 2026. Section 24.1, current documentation; accessed March 5, 2026.

[101] PostgreSQL Global Development Group. Postgresql documentation: Vacuum. `https://www.postgresql.org/docs/current/sql-vacuum.html`, 2026. SQL command reference, current documentation; accessed March 5, 2026.

[102] J. Reardon, D. Basin, and S. Capkun. Sok: Secure data deletion. In *2013 IEEE Symposium on Security and Privacy*, pages 301–315, 2013.

[103] T. Rekatsinas, X. Chu, I. F. Ilyas, and C. Ré. Holoclean: holistic data repairs with probabilistic inference. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 10(11):1190–1201, Aug. 2017.

[104] X. Ren, L. Shi, W. Yu, S. Yang, C. Zhao, and Z. Xu. Ldp-ids: Local differential privacy for infinite data streams. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1064–1077, New York, NY, USA, 2022. Association for Computing Machinery.

[105] P. Ricœur. *Memory, History, Forgetting*. University of Chicago Press, 2004. Translated by Kathleen Blamey and David Pellauer. French original: *La Mémoire, l'histoire, l'oubli*, Éditions du Seuil, 2000.

[106] L. Robaldo, C. Bartolini, and G. Lenzini. The DAPRECO knowledge base: Representing the GDPR in legalruleml. In *Proceedings of The 12th Language Resources and Evaluation Conference, LREC 2020, Marseille, France, May 11-16, 2020*, pages 5688–5697. European Language Resources Association, 2020.

[107] L. Robaldo, C. Bartolini, M. Palmirani, A. Rossi, M. Martoni, and G. Lenzini. Formalizing gdpr provisions in reified i/o logic: The dapreco knowledge base. *J. of Logic, Lang. and Inf.*, 29(4):401–449, Dec. 2020.

[108] E. Rupp and E. Syrmoudis. Leave no data behind–empirical insights into data erasure from online services. *Proceedings on Privacy Enhancing Technologies*, 2022(4):445–464, 2022.

[109] S. Sarkar and M. Athanassoulis. Query language support for timely data deletion. In

Proceedings of the International Conference on Extending Database Technology (EDBT), pages 418–429, 2022.

[110] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis. Lethe: A tunable delete-aware lsm engine. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 893–908, New York, NY, USA, 2020. Association for Computing Machinery.

[111] N. Scope, A. Rasin, B. Lenard, J. Wagner, and K. Heart. Purging compliance from database backups by encryption. *J. Data Intell.*, 3(1):149–168, 2022.

[112] S. Shastri, V. Banakar, M. Wasserman, A. Kumar, and V. Chidambaram. Understanding and benchmarking the impact of gdpr on database systems. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 13(7):1064–1077, Mar. 2020.

[113] Singapore. Personal data protection act 2012 (pdpa) - part iv: Retention limitation obligation, 2012. Singapore Statutes Online. Last accessed on 2025-01-10.

[114] S. Song, Y. Wang, and K. Chaudhuri. Pufferfish privacy mechanisms for correlated data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, SIGMOD '17, page 1291–1306, New York, NY, USA, 2017. Association for Computing Machinery.

[115] SQLite Consortium. SQLite Documentation: PRAGMA Statements. `https://sqlite.org/pragma.html#pragma_secure_delete`. Accessed: 2026-06-29.

[116] S. Staworko. *Declarative Inconsistency Handling in Relational and Semi-Structured Databases*. Phd thesis, State University of New York at Buffalo, 2007.

[117] Supreme Court of the United States. Olmstead v. united states, 277 U.S. 438, 1928. Dissenting opinion of Justice Louis D. Brandeis.

[118] Supreme Court of the United States. Riley v. california, 573 U.S. 373, 2014. Opinion of the Court by Chief Justice John G. Roberts Jr.

[119] Supreme Court of the United States. Carpenter v. united states, 585 U.S. 296, 2018. Opinion of the Court by Chief Justice John G. Roberts Jr.

[120] Transaction Processing Performance Council. TPC-H Benchmark Specification, Version 2.17.3. Technical report, Transaction Processing Performance Council, 2021.

[121] L. Tsai, H. Gross, E. Kohler, F. Kaashoek, and M. Schwarzkopf. Edna: Disguising and revealing user data in web applications. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 434–450, New York, NY, USA, 2023. Association for Computing Machinery.

[122] United States. Hipaa security rule - 45 cfr §164.310(d)(2)(i), 2003. U.S. Code of Federal Regulations. Last accessed on 2025-01-10.

[123] C. Wang, J. Bater, K. Nayak, and A. Machanavajjhala. Dp-sync: Hiding update patterns in secure outsourced databases with differential privacy. In *Proceedings of the*

2021 International Conference on Management of Data, SIGMOD '21, page 1892–1905, New York, NY, USA, 2021. Association for Computing Machinery.

[124] S. D. Warren and L. D. Brandeis. The right to privacy. *Harvard Law Review*, 4(5):193–220, 1890.

[125] R. Xiao, Z. Tan, H. Wang, and S. Ma. Fast approximate denial constraint discovery. *Proceedings of the International Conference on Very Large Databases (VLDB)*, 16(2):269–281, Oct. 2022.

[126] B. Yang, I. Sato, and H. Nakagawa. Bayesian differential privacy on correlated data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 747–762, New York, NY, USA, 2015. Association for Computing Machinery.