Achieving Regulatory Compliance in Data Management

A Dissertation Presented by

Sumeet Vijay Bajaj

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2014

Stony Brook University
The Graduate School

Sumeet Vijay Bajaj

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

Radu Sion – Dissertation Advisor
Associate Professor, Computer Science

Erez Zadok – Chairperson of Defense
Associate Professor, Computer Science

Donald Porter
Assistant Professor, Computer Science

Johannes Gehrke
Professor, Computer Science, Cornell University

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Achieving Regulatory Compliance in Data Management

by

Sumeet Vijay Bajaj

Doctor of Philosophy

in

Computer Science

Stony Brook University

2014

Regulations mandate consistent procedures for information access, processing, and storage. In the United States alone, over 10,000 data management regulations exist in the financial, life sciences, health care and government sectors. A recurrent theme in data management regulations is the need for regulatory compliant storage to ensure data confidentiality, data integrity, audit trails maintenance, data retention, and guaranteed deletion.

This thesis describes the design and implementation of several regulatory compliant relational databases and file systems. The systems increase efficiency and lower costs of regulatory compliance through the use of novel cryptographic and system security constructs.

The first system described in this thesis is TrustedDB. TrustedDB is a relational database that ensures data confidentiality. TrustedDB enables SQL query execution over an encrypted database hosted with a remote, untrusted service provider. TrustedDB is the first DBMS with data confidentiality that does not limit query expressiveness. Moreover, the per query execution costs in TrustedDB are orders of magnitude lower than current cryptography-based mechanisms. To significantly lower query execution costs, TrustedDB leverages server-hosted, tamper-proof trusted hardware in critical query processing stages.

The second system described in this thesis is CorrectDB. CorrectDB is a relational database that provides efficient, low-cost Query Authentication (QA). QA requires strict guarantees for both the correctness and completeness of the query results returned by potentially compromised providers. Similar to TrustedDB, CorrectDB leverages server-hosted trusted hardware. CorrectDB achieves economy and efficiency by minimizing server-side authentication data and by reducing the client-server communication overheads.

The third system described in this thesis is ConcurDB. ConcurDB provides concurrent query authentication in a multi-client scenario wherein many clients simultaneously perform update operations. ConcurDB achieves high concurrency by decoupling transaction execution and verification – permitting transactions to execute concurrently and performing verifications in parallel.

The fourth system described in this thesis is the history independent file system (HIFS). HIFS guarantees secure data deletion by providing full history independence across both file system and disk layers of the storage stack. HIFS overcomes the challenge of simultaneously preserving history independence and data locality. Moreover, HIFS is customizable to suit several data locality scenarios, such as block-group locality and sequential file storage.

This thesis also builds the theoretical foundations of history independence. The thesis explores the concepts of abstract data types, data structures, machine models, memory representations and history independence itself. The thesis then proposes Δ history independence (ΔHI), a generic game-based framework that is malleable enough to define a broad spectrum of new history independence notions. To bridge the gap between theory and practice, the thesis outlines a general process for building history independent systems. HIFS itself is designed using the suggested process.

Finally, this thesis describes Ficklebase. Ficklebase is a relational database that provides irrecoverable data erasure. In Ficklebase, once a tuple is deleted all side effects of the delete tuple are removed. Removal of all side effects of a deleted tuple achieves the same effect as if the deleted tuple was never inserted in the database. Ficklebase thus eliminates all traces of deleted data rendering data irrecoverable and also guaranteeing that the deletion itself is undetectable.

To Mansi
मिठी दिल्

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

First of all, I would like to thank my advisor Radu Sion for all his guidance and support during my PhD. Radu gave me the freedom to work on research topics of my choice. More importantly, he gave me the leeway to enjoy life outside of work which meant research was accompanied by memorable adventures that I will cherish for a long time. To put it briefly, Radu made PhD fun!

I also thank my committee members for all their valuable comments and suggestions from the prelim to defense.

I would also like to thank Abhishek Kumar, Sumeet Dash, Nilesh Mahajan, Vasudev Bhat, and Gaurav Menghani for their help with system implementations.

I am grateful to my parents for all their love and encouragement throughout my life. I would also like to thank my lovely wife Mansi, who has always been by my side for the better half of my PhD.

Last but not least, I would like to thank all my friends and colleagues at the NSAC lab for acknowledging several small and big requests over the last 5 years.

Chapter 1

Introduction

1.1 Regulatory Compliance

Today, regulations govern all areas of commerce. Important regulatory functions include mandating quality of manufactured goods, controlling prices of services, maintaining wage standards, ensuring suitable working conditions, limiting effects of production on the environment, preventing discrimination, and increasing corporate transparency. In short, regulations can be viewed as rules set forth to protect the interests of society at large. Adherence to regulations is *Regulatory Compliance*.

1.2 Regulations in Data Management

In recent times, the increasing collection and processing of data has raised several concerns regarding data confidentiality, access, and retention.

In some areas, such as health care and banking, data confidentiality is a prime concern since data confidentiality is directly linked to individual privacy. Illegitimate access to confidential data can have financial consequences for individuals and businesses. For example, impersonation via theft of personal information costs consumers and business billions of dollars annually [194,243]. A data breach involving sensitive health care records costs companies up to $359 per compromised record [163].

Accurate data recording, maintaining integrity of recorded data, and authorized access to data concern both regulators and regulated entities. For regulators, data serves as evidence in the event a regulation is violated. For regulated entities, data serves as proof of compliance.

For many applications, the duration between data generation or data collection and actual use of data can be significant. For example, a financial regulatory body may wish to audit a business years after the suspicious transactions occurred. Such applications are possible only if data is retained with integrity for the desired period. In certain other scenarios, such as defense records management [192], the retention of data for excessive periods increases the risk of data leaks. Hence, timely and irrecoverable disposal of data is also a concern.

Driven by the above concerns, regulators have enacted laws that govern all facets of data management. In the United States alone, there are over 10,000 regulations outlining how data records should be created, stored, accessed and retained [119].

1.2.1 Classification of Data Management Regulations

Regulations are enacted in the form of a legal document referred to as an *Act*, a *Directive*, a *Program*, or an *Agreement*. For consistency, we refer to any such legal document as a *Regulation*[1].

Although each Regulation governs multiple facets of data management, a Regulation has a primary area of focus. For example, the EU Data Protection Directive [252] is primarily concerned with data retention. However, retention of data without ensuring data integrity can be fruitless. Hence, the directive also lists secondary clauses for data protection.

To give an overview of current Regulations, Tables 1.1 and 1.2 list some of the US and foreign Regulations pertaining to data management along with selected important clauses from each Regulation. For each listed Regulation, we identify the Regulation's primary area of focus, which is one of the following – *privacy*, *audit* or *retention*.

We note that Regulations with a focus other that privacy, audit, and retention may exist. We choose to focus on privacy, audit, and retention Regulations because majority of compliance efforts are geared towards these Regulations. For instance, 90% of total compliance costs incurred by companies are towards ensuring compliance with privacy, audit, and retention Regulations [161]. Also, surveys of multinational organizations [162] have shown that privacy, audit, and retention Regulations are perceived to be most important.

1.2.2 Privacy Regulations

Privacy Regulations are primarily concerned with data confidentiality and with protecting individual privacy. The goal of data confidentiality is to ensure legitimate access to data. Data confidentiality comes into picture after data has been collected from individuals. Individual privacy refers to an individual's right to choose what personal information can be shared and with whom.

Many entities collect personal information from individuals in order to provide services [138]. If used only for the intended purpose of delivering services, the collection of personal information is beneficial to both individuals and service providers. However, since personal information can be linked to specific individuals, illegitimate access to personal information violates individuals' privacy. Hence, privacy Regulations demand service providers to ensure that personal information is not accessed for any purpose other than the intended provision of services. Specifically, privacy Regulations require service providers to employ data confidentiality, deidentify personal information before distribution, seek explicit permission from individuals before disclosing personal information, and to notify individuals in

[1] We differentiate between a regulation, with a lowercase letter 'r', and Regulation, with an uppercase letter 'R'. Regulation refers to a legal document while regulation as its dictionary meaning suggests is a rule.

Regulation		Primary Focus	Important Clauses
Health Insurance Portability and Accountability Act	[77]	Privacy	Sec 1177(a) – Penalties for wrongful disclosure of individually identifiable health information.
Patient Safety and Quality Improvement Act	[142]	Privacy	Sec 922 – Privilege and Confidentiality protections. Sec 923 – Computation of national and regional statistics preserving individual privacy.
Federal Information Security Management Act	[14]	Privacy	Title III Sec 301 § 3547 – Information security protections for national security systems.
Family Educational Rights and Privacy Act	[49]	Privacy	34 CFR § 99.31 – Prevent unauthorized disclosure of education records.
Massachusetts Identity Theft Protection Regulation	[18]	Privacy	Sec 16.04 to 16.06 – Mandates reporting agencies to comply with authorized placement, lifting and removal of security freeze on consumer data.
The Children's Online Privacy Protection Act	[75]	Privacy	Sec 1303(b)(D) – Require the service operator to establish and maintain reasonable procedures to protect the confidentiality, security, and integrity of personal information collected from children.
California Security Breach Information Act	[207]	Privacy	Sec 4 – Prompt notification of breach to any resident whose unencrypted personal information was, or is reasonably believed to have been, acquired by an unauthorized entity.
Sarbanes-Oxley Act	[232]	Audit	Title XI Sec 1102 – Penalize tampering or concealment of a record, document etc, or attempts to do so, with the intent to impair the objects integrity or availability for use in an official proceeding.
Code of Federal Regulations	[83]	Retention	17 CFR 240.17a-4 – Records to be preserved by certain exchange members, brokers and dealers. (ii)(A) – Preserve the records exclusively in a non-rewritable, non-erasable format.
Department of Defense Records Management Program	[192]	Retention	C2.2.7.6.3 – Delete electronic records approved for destruction in a manner that prevents their physical reconstruction using commonly available file restoration utilities.

Table 1.1: Partial list of US Regulations pertaining to data management.

Regulation		Primary Focus	Important Clauses
EU Data Protection Directive	[251]	Privacy	Section VIII Articles 16,17 – Confidentiality and Security of Processing.
Personal Information Protection and Electronic Documents Act (Canada)	[191]	Privacy	Sec 4,5,6 – Collection, retention and disposal of personal information. Schedule 1 Sec 5 – Model code for the protection of personal information.
Regulation of Investigatory Powers Act (UK)	[197]	Privacy	Part III – Investigation of electronic data protected by encryption (power to require disclosure).
Data Protection Act (UK)	[13]	Privacy	Principle 3 – Personal data shall be adequate, relevant and not excessive in relation to the purpose or purposes for which they are processed. Principle 5 – Personal data processed for any purpose or purposes shall not be kept for longer than is necessary for that purpose or those purposes.
Personal Data Protection Act (Singapore)	[196]	Privacy	Part IV – Collection, use and disclosure of personal data. Part V – Access to and correction of personal data. Part VI – Care of personal data (protection, retention and transfer).
Information Technology Amendment Act (India)	[193]	Privacy	66E – Prohibits publishing or transmission of image of private parts of an individual without consent.
German Corporate Governance Code	[76]	Audit	Sec 7 – Reporting and Audit of the Annual Financial Statements.
Listing Agreement to the Indian stock exchange	[237]	Audit	(D)(4)(e) – Compliance with listing and other legal requirements relating to financial statements.
Corporate Law Economic Reform Program Act (Australia)	[15]	Audit	RG 34.1,2 – Auditor Obligations. RG 34.3 – Auditor reporting requirements. RG 34.22,23 – Penalties for audit failure.
EU Data Retention Directive	[252]	Retention	(11) – Retention of traffic and location data. Article 5 – Categories of data to be retained. Article 7,8 – Protection, security and storage requirements for retained data.

Table 1.2: Partial list of foreign Regulations pertaining to data management.

case of unauthorized access to individuals' personal information.

In certain scenarios, the benefits of exchange and publication of personal information are significant. For example, the results of analytical research on health data can be extremely valuable for the well being of the populace. Regulators do recognize the benefits of sharing personal information and adapt accordingly. The Health Insurance Portability and Accountability Act (HIPAA), for instance, permits release of health information to authorized entities as long as appropriate statistical principles are applied to render information not individually identifiable.

Certain privacy Regulations are focussed to protect specific segments of the population. For example, the Children's Online Privacy Protection Act (COPPA) [75] applies to online collection of information from children under 13 years of age. COPPA requires website operators to provide suitable mechanisms for parents to review and control the personal information collected from children. Similarly, the Family Educational Rights and Privacy Act (FERPA) [49] regulates the collection and dissemination of students' educational records.

1.2.3 Audit Regulations

Audit Regulations have two broad requirements. First requirement is the collection and maintenance of data that can be used to verify compliance in the future. Second requirement is periodic audits using the collected data to detect noncompliance.

Regulations use penalties as a deterrent from noncompliance. For instance, Schedule 3 of the Corporations Act [190] lists specific penalties for over 300 offenses. An offense is a violation of the Act's clauses. The penalties are stated in form of monetary fines and criminal prosecutions leading to imprisonment.

Enforcing penalties requires evidence that can prove noncompliance in court. Evidence for noncompliance can be gathered via audits. Typically, regulations require periodic audits to verify operating practices of regulated entities. For example, the Corporate Law Economic Reform Program Act (CLERP9) [15] in Australia, requires audit of semi-yearly and yearly financial statements of companies by certified auditors. CLERP9 mandates auditors to report within 28 days any circumstances that give the auditor reasonable grounds to suspect noncompliance. Similar laws exist in other countries, such as the German Corporate Governance Code [76], the Financial Instruments and Exchange Act (J-SOX) [16] in Japan, and Keeping the Promise for a Strong Economy Act in Canada [90].

Provenance for digital records [129] also plays an important role in audits and is required by several Regulations [131]. The Gramm-Leach-Bliley Act [20] and the Securities and Exchange Commission rule 17-a [236] require audit trails for financial records.

1.2.4 Retention Regulations

Retention Regulations govern the periods for which various types of data should be retained.

To ensure the availability of data for future audits, retention Regulations mandate minimum retention periods for certain data. For example, the EU Data Retention Directive [252] requires certain communications data to be retained for a minimum period of six months.

Additionally, the directive also lists storage and protection requirements for the retained data. Section 103 of the Sarbanes-Oxley Act (SOX) [232] requires corporations to maintain audit reports for a period of seven years. SOX also mandates auditors to retain all audit data for a period of five years after an audit is concluded. To deter noncompliance, SOX imposes heavy penalties on tampering or concealment of data relevant to audits.

Retention Regulations also stipulate the maximum retention periods for certain data. Once maximum retention period ends, thorough and safe disposal of data is required. The goal of limiting retention periods is to prevent data retention past the intended use, thereby reducing the risks of data misuse. Examples of retention Regulations that mandate maximum retention periods include the EU Data Retention Directive [252], the UK Data Protection Act [13], and the Code of Federal Regulations [84]. The EU Data Retention Directive limits retention of communications data to two years. The UK Data Protection Act [13], in principle five, more generally bars the retention of personal data for any longer than its intended purpose. The Code of Federal Regulations (CFR) [84] requires physical destruction of electronic media if other means are insufficient in ensuring data irrecoverability after the maximum retention period.

1.3 Challenges in Achieving Regulatory Compliance

Many challenges lie on the path towards regulatory compliant data management. The challenges are faced by both regulators and the entities being regulated. Challenges are in the form of scale and complexity of modern systems; advent of new computing environments, such as cloud services; complexity of regulatory framework; and high costs of compliance.

1.3.1 Systems' Scale and Complexity

Today, the increasing use of digital systems for data management places immense pressure on enforcing regulatory compliance. A large organization can have hundreds of data management systems executing a host of enterprise applications, such as logistics, finance, supply chain, customer service, and human resources. Moreover, systems are increasing in both scale and complexity. Ensuring continuous regulatory compliance in such an environment is a complex undertaking for organizations and for regulators.

1.3.2 Cloud Services

Both private and government organizations are increasingly considering the use of cloud services to reap the benefits of scale, flexibility, and low costs. As a result, Regulations are expanding their scope to include cloud services. Regulations, such as the Gramm-Leach-Bailey Act [20], the Massachusetts Data Security Regulations (201 CMR 17.00) [18], and the EU E-Privacy Directive [205] have already included cloud services within their regulatory boundaries. For example, the EU Data Protection Directive [251] limits the transfer of cloud-hosted personal data to certain countries.

Regulations place the onus of compliance on cloud users and not necessarily on service providers [78]. However, cloud users have limited guarantees of where their data is stored; and how data is protected and shared. Instead, users have to rely on and trust service providers for compliance. Therefore, in the event of noncompliance by service providers, cloud users may be penalized. As a result, many users avoid cloud services and the full potential of cloud services is not realized.

1.3.3 Complexity of Regulatory Framework

The complexity of regulatory framework has been increasing. Since 1997, federal agencies have published between 2,500 and 4,500 regulations each year [54]. The number of regulations pertaining to data management have increased significantly as a result of recent infringements [7] and financial crisis [220].

A large and complex regulatory framework puts greater responsibility on system designers and application developers. Keeping track of regulatory requirements and aligning system policies accordingly is a complex task and warrants a high degree of automation.

1.3.4 Costs of Compliance

Costs for regulatory compliance are incurred by both regulators and regulated entities. Regulators incur costs to formulate regulations, to oversee enforcement, to investigate transgressions, and to prosecute. Estimated annual costs of US federal regulations range between $57 billion and $84 billion [195]. In order to be compliant, regulated entities need to invest in technology and personnel. Studies indicate that multinational organizations on an average spend $3.5 million annually to stay compliant [162]. For the hedge fund industry, seven percent of total operating costs are towards compliance related activities [140]. Moreover, per-capita compliance cost is higher for smaller organizations [140, 162].

1.4 Current State of Regulatory Compliant Data Management Systems Research

Over the last two decades a large body of research has focused on regulatory compliance in data management. Also, with increasing awareness of and emphasis on regulatory compliance, research areas that were not intended to serve regulatory compliance at inception have now been identified as a solution towards regulatory compliance. Figure 1.1 lists the areas of research that address privacy, audit and retention Regulations.

This section gives a broad overview of research areas targeting regulatory compliance in data management. This section does not serve as an in-depth review of current literature. Chapter 9 discusses related work in detail.

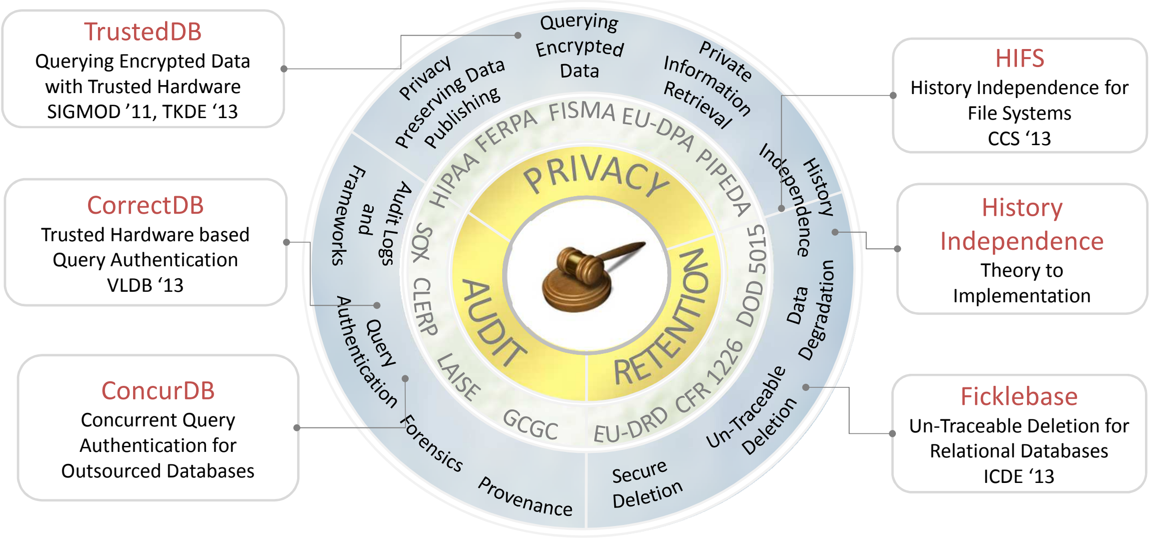

TrustedDB
Querying Encrypted Data with Trusted Hardware
SIGMOD ’11, TKDE ‘13
CorrectDB
Trusted Hardware based Query Authentication
VLDB ‘13
ConcurDB
Concurrent Query Authentication for Outsourced Databases
HIFS
History Independence for File Systems
CCS ‘13
History Independence
Theory to Implementation
Ficklebase
Un-Traceable Deletion for Relational Databases
ICDE ‘13
PRIVACY
AUDIT
RETENTION
HIPAA FERPA FISMA EU-DPA PIPEDA
DOD 5015
CFR 1226
EU-DRD
GCGC
LAISE
CLERP
SOX
Querying Encrypted Data
Private Information Retrieval
History Independence
Data Degradation
Un-Traceable Deletion
Secure Deletion
Provenance
Forensics
Query Authentication
Audit Logs and Frameworks
Privacy Preserving Data Publishing

Figure 1.1: Current research areas serving regulatory compliance in data management. Contributions of this thesis are highlighted with references.

1.4.1 Research Areas Addressing Privacy Regulations

Privacy-preserving data publishing

The sharing of personal information is permitted by certain Regulations as long as appropriate measures are taken to render data not individually identifiable. To be practical, data usefulness must be preserved when data is de-identified. Research in the area of Privacy Preserving Data Publishing (PPDP) [96] has strived to achieve deidentification while preserving data usefulness. The objective under PPDP is transformation of the original data to render inferences about individually identifiable information unlikely. Based on the privacy model [96], PPDP can be further classified into the subareas of k-anonymity [144, 228, 229], MultiR k-Anonymity [188], l-Diversity [167], Confidence Bounding [258, 259], (α, k)-Anonymity [264], (X, Y)-Privacy [257], (k, e)-Anonymity [274], Personalized Privacy [267], t-Closeness [158], δ-Presence [187], (c, t)-Isolation [59, 60], ϵ-Differential Privacy [80, 87, 135, 176], (d, γ)-Privacy [218], Distributional Privacy [40], Data Randomization [23, 24], and Information Downgrading [21, 180, 254].

Data Confidentiality

Encryption is commonly used to protect data residing with untrusted cloud services. The hope is that encryption will help data owners to stay compliant and benefit from the use of cloud services. However, encryption limits the type of operations that can be performed on data reducing the functionality that cloud services can offer.

Both theoretical and systems research have focussed to overcome the limitations imposed by encryption on computation. On the theoretical front, new mathematical constructs, such as homomorphic encryption [42, 104–106, 223, 253] have been devised to enable computation over encrypted data without the need for decryption. However, implementations of these constructs are not yet practical. For example, processing SQL queries using fully homomorphic encryption requires days of processing to recover a single database record [272]. Even for primitive operations, such as addition of two integers, the cost associated with homomorphic encryption are orders of magnitude higher that processing of plaintext data (Section 2.4).

In order to be efficient, systems research has focussed on specific scenarios, such as range and aggregation query processing over encrypted data [92,122,123,168,212]. However, limiting functionality to a small subset of query operations reduces practicality.

Private Information Retrieval (PIR)

In certain scenarios, providing data confidentiality alone is insufficient to ensure privacy. In addition, the manner in which data is accessed also needs to be hidden. Techniques to hide data access patterns are being explored under Private Information Retrieval (PIR) [102, 116, 244, 261–263, 279].

1.4.2 Research Areas Addressing Audit Regulations

Audit Logs

Regulations require maintenance of audit logs in various applications, such as drug approval data, medical information disclosure, financial records, and electronic voting [77, 222, 232]. The collection and logging of system activity is not a particularly difficult task. The real challenges lie in protecting the integrity of recorded data and in analyzing the recorded data to determine compliance. To address the challenges of data integrity and analysis in audit logs, researches have designed tamper-proof audit logs [222, 234], audit frameworks [57, 100], and forensic tools [148, 206, 248]. Audit frameworks enable application designers to specify policies in high-level languages, which can then be auto-enforced in data management systems. Forensic tools aid in analysis of audit logs.

Provenance

Digital provenance mechanisms [129–131, 164] support the collection and persistence of information about the creation, access, and transfer of data. Provenance can therefore play an important role in audits for regulatory compliance. However, the challenges of provenance information collection and migration of provenance records across organizational boundaries are yet to be overcome [129].

Verification of Computation

Outsourcing data to potentially untrusted environments, such as a third-party cloud, raises concern over the correct operation of the remote services. The concern is especially serious when the liability for compliance is on cloud users. Using techniques for verifiable computation [102,111], a remote client can verify the correct execution of an outsourced computation.

However, current techniques for verifiable computation are impractical, consuming several orders of magnitude more resources as compared to unverified computation [255]. To lower the costs of verifiable computation, research has focussed on solutions for specific scenarios, such as range query verification [85, 182, 203, 204, 240] in outsourced relational databases. Although the overheads of verification are relatively lower in specific targeted scenarios, the overheads are high enough to be a significant deterrent for widespread use (Section 3.5).

1.4.3 Research Areas Addressing Retention Regulations

Retention regulations stipulate both minimum and maximum data retention periods for the purpose of audit and privacy, respectively. Minimum retention period indicates the duration for which data must be retained. Maximum retention period mandates the time when data must be disposed and made irrecoverable.

Addressing Minimum Retention

Compliance storage [29, 132, 159] and trustworthy indexes [177, 208, 277] have been designed for secure retention of data records. Compliance storage facilitates the storage and protection of audit-related data. Data stored with compliance storage is protected from both tampering and deletion. In addition to data storage and protection, trustworthy indexes also permit verified searches on stored data. Currently, trustworthy indexes have been designed for key-based lookups and range queries.

Addressing Maximum Retention

Secure deletion [86] mechanisms are proposed to render data irrecoverable on deletion. Secure deletion is achieved by either overwriting data to be deleted [34, 68, 121, 209], or by using encryption [151, 153]. Data degradation [26, 28] on the other hand gradually degrades data precision. The final step in data degradation is secure deletion.

Overwriting deleted data or using encryption as in secure deletion does not ensure irrecoverability of deleted data. Since the past existence of delete data affects the current system state implicitly at all layers, even after secure deletion, evidence of past existence of deleted data can be recovered by analyzing data side effects and current system state [173].

1.5 Thesis Goals

This thesis is driven by the motivation for low-cost, efficient, and increasingly automated regulatory compliance in data management. Following are the thesis objectives targeting specific requirements from privacy, audit and retention Regulations.

- To increase functionality and lower costs of query processing over encrypted data in outsourced databases, thereby satisfying privacy Regulations and facilitating the adoption of cloud services.

- To serve audit Regulations by efficient and low-cost verification of SQL query results in database outsourcing.
- To comply with retention Regulations by ensuring secure, irrecoverable data erasure in relational databases and file systems.

1.6 Achieving Regulatory Compliance in Data Management

Figure 1.1 highlights our contributions towards practical, cost-efficient systems for regulatory compliance in data management. We briefly discuss our contributions here and treat each one in depth in subsequent chapters.

1.6.1 Querying Encrypted Data with TrustedDB

To address data confidentiality in an outsourced setting we have designed TrustedDB (Chapter 2). TrustedDB is a relational database that enables SQL query execution with privacy and under regulatory compliance constraints over a database hosted with an untrusted service provider. TrustedDB is the first DBMS with full privacy that does not limit query expressiveness.

To achieve full SQL execution over encrypted data, TrustedDB leverages server-hosted, tamper-proof trusted hardware in critical query processing stages. In TrustedDB, each data column is categorized as either sensitive or nonsensitive. Sensitive columns are encrypted and nonsensitive columns are stored unencrypted. A client query is then split into sub-queries. A subquery that accesses sensitive columns is processed entirely within the trusted hardware ensuring data confidentiality. A subquery that does not access any sensitive columns is processed on the host server. TrustedDB thus balances query execution load between the resource-constrained trusted hardware and the host server.

Within trusted hardware, data is decrypted before processing. The processing of data in plaintext significantly reduces the computation required as compared to current cryptographic techniques [198, 199, 223] for direct processing over encrypted data. As a result, the per-query-execution costs in TrustedDB are orders of magnitude lower than existing software-only, cryptography-based mechanisms.

For efficiency, TrustedDB is equipped with query optimization techniques designed specifically for a trusted hardware model. Query optimization in TrustedDB ensures that client queries are split in a manner that minimizes query execution time.

1.6.2 Low-cost, Efficient Query Authentication with CorrectDB

Query authentication (QA) requires strict guarantees for both the correctness and completeness of the query results returned by potentially compromised providers. For users of cloud-based databases, QA offers the ability to prove noncompliance by service providers.

Existing solutions provide QA assurances for limited query types using server-side, authenticated data structures. The client-server QA protocols in existing solutions incur high data transfer and processing costs. We show that to achieve QA, it is significantly cheaper and more practical to use server-hosted, tamper-proof trusted hardware.

To identify the benefits of trusted hardware for QA, we extensively surveyed existing QA work analyzing limitations and cost points. Our results indicate that despite the higher acquisition costs of trusted hardware, the overall costs of trusted hardware-based QA solutions is significantly lower. Further, the use of trusted hardware for QA provides additional benefits, such as the ability to handle arbitrary queries, server-side updates, replay attack detection, and client synchronization.

To demonstrate the benefits of trusted hardware for QA, we designed CorrectDB (Chapter 3). CorrectDB is a relational database that provides full QA assurances. CorrectDB leverages server-hosted trusted hardware to meet QA requirements cheaply and efficiently. In CorrectDB, query verification is performed by server-side, trusted hardware. Minimal data is sent to the client for query verification, thereby significantly reducing the client-server data transfer costs.

1.6.3 Concurrent Query Authentication with ConcurDB

Most existing authenticated data structures designed for Query Authentication (QA) assume read-only or infrequently updated databases. For dynamic datasets, the data owner is required to perform all updates on behalf of clients. Hence, for concurrent updates by multiple clients, such as for OLTP workloads, existing QA solutions are inefficient.

To overcome concurrency limitations in QA, we designed ConcurDB, a concurrent QA scheme that enables simultaneous updates by multiple clients. To realize concurrent QA, we have designed several new mechanisms. Firstly, we identify and use an important relationship between QA and memory checking to decouple query execution and verification. We allow clients to execute transactions concurrently and perform verifications in parallel. Then, to extend QA to a multi-client scenario, we design new protocols that enable clients to securely exchange a small set of authentication data even when using the untrusted provider as a communication hub. Finally, we overcome provider-side replay attacks.

Using ConcurDB, we provide and evaluate concurrent QA for the full TPC-C benchmark. For updates, ConcurDB shows a 4x performance increase over existing solutions.

1.6.4 History Independence for Regulatory Compliance

The way data structures organize data is often a function of the sequence of past operations. The organization of data is referred to as the data structure's *state*, and the sequence of past operations constitutes the data structure's *history*. A data structure state can therefore be used as an oracle to derive information about its history. For example, the current organization of data blocks on disk is a function of the sequence of previous writes to file system, or to database search indexes. Questions such as "was John's record ever in the HIV patients' dataset" can then be answered much more accurately than guessing by simply looking at the search index organization on disk since the organization could be different depending on whether John has previously been in the data set or not. The inference of past existence of deleted data is in direct violation of data retention Regulations. As a result, for compliance with retention Regulations, it is imperative to conceal historical information contained within data structure states.

Data structure history can be hidden by making data structures *history independent*. We explore how to achieve history independence in both theory and practice.

On the theoretical side we postulate the need for a broad, encompassing notion of history independence, which can capture existing history independence notions and can be used to define a broad spectrum of new history independence notions. We then introduce Δ history independence (ΔHI), a generic game-based framework that is malleable enough to accommodate existing and new history independence notions. Additionally, ΔHI helps to reason about the history preserved or hidden by existing data structures including ones that were designed without history independence in mind. In the process of formalizing ΔHI we explore the concepts of abstract data types, data structures, machine models, memory representations and history independence itself.

To bridge the gap between theory and practice, we outline a general process for building end-to-end, history independent systems. We demonstrate the use of this process in designing two file systems – a history independent file system (HIFS) and delete-agnostic file system (DAFS).

HIFS (Chapter 6) guarantees secure deletion by providing full history independence across both file system and disk layers of the storage stack. HIFS also preserves data locality and provides tunable efficiency knobs to suit different application data locality scenarios. DAFS (Chapter 6) optimizes on the HIFS design to effectively comply with maximum retention requirements in Regulations.

1.6.5 Untraceable Deletion and Ficklebase

Proper disposal of data records after the maximum retention period is mandated by certain data retention Regulations. In prior work, disposal of data is performed by *secure deletion*. Under secure deletion, data is typically deleted by overwriting. However, overwriting does not prevent recovery of deleted data. Past existence of deleted data can instead be inferred from current data organization.

We identified history independence as a solution to eliminate inferences about deleted data via data organization. Under history independence, data organization depends on current data only. Deleted data leaves no effect on current data organization that can be used for recovery. However, side effects of deleted data may persist within current system data. The current data itself can be used to derive information about deleted data in direct violation of retention Regulations.

For truly irrecoverable data erasure, secure deletion and history independence are insufficient. Along with secure deletion and history independence, complete removal of post-deletion data residues and processing side effects is required. We refer to the removal of all data residues and side effects as *untraceable deletion*. We formalize untraceable deletion for relational databases and provide insights into the new functional aspects of untraceable deletion.

Ficklebase (Chapter 8) is our relational database design that achieves untraceable deletion. In Ficklebase, once a tuple is deleted, all side effects of the deleted tuple are removed. Removal of all side effects of a deleted tuple achieves the same effect as if the deleted tuple was never inserted in the database. Ficklebase thus eliminates all traces of deleted data, rendering deleted data irrecoverable and also guaranteeing that the deletion itself is undetectable.

1.7 Summary of Contributions

This dissertation advances regulatory compliant data management systems research.

- For query processing with data confidentiality, we make the following contributions.
  - The introduction of new cost models and insights that quantify the advantages of using trusted hardware for data processing.
  - The design, development, and evaluation of TrustedDB, the first trusted hardware-based relational database with full data confidentiality and with no limits on query expressiveness.
  - New query processing techniques that overcome the processing and storage limitations of trusted hardware.
  - New query optimization techniques for a trusted hardware-based query execution model.
- Regarding Query Authentication (QA) we make contributions as follows.
  - A comparative survey of existing QA research that explores both theoretical and empirical dimensions based on published results.
  - A cost analysis and associated insights showing that using trusted hardware for QA is significantly more efficient both in cost and performance.
  - CorrectDB, a new trusted hardware-based database with full QA assurances.
  - Identification of the equivalence between memory checking and query authentication.
  - ConcurDB, a concurrent query authentication solution that supports simultaneous updates by multiple clients.
- For regulatory compliant data retention, we contribute the following.
  - A formalization of history independence via the exploration of essential concepts, such as abstract data types, data structures, machine models, and memory representations.

– New game-based definitions of weak and strong history independence that are more appropriate for the security community.

– A new notion of history independence, termed Δ history independence (ΔHI). ΔHI centers around a generic game-based definition of history independence and is malleable enough to accommodate existing and new history independence notions.

– A general recipe for designing history independent systems.

– The design, implementation and evaluation of the first history independent file system (HIFS).

– The design, implementation and evaluation of a delete-agnostic file system (DAFS).

– Introduction of untraceable deletion as a mechanism to achieve truly irrecoverable data erasure.

– Ficklebase, a new relational database that achieves untraceable deletion.

1.8 Thesis Outline

In this dissertation we make several contributions towards regulatory compliance in data management. Our contributions address requirements of privacy, audit and retention Regulations. Contributions are in the form of theoretical results and practical system designs.

Chapters 2 to 8 focus on one contribution each. Chapter 2 presents the cost-tradeoffs, design, implementation and evaluation of TrustedDB. Chapters 3 and 4 focus on CorrectDB and ConcurDB, respectively. The theoretical foundations of history independence are built and discussed in Chapter 5. Then, Chapter 6 uses the theoretical concepts and results from Chapter 5 to design the first history independent file system (HIFS). Chapter 7 presents the delete agnostic file system (DAFS). Chapter 8 introduces Un-Traceable deletion and Ficklebase. In Chapter 9 we discuss related work in regulatory compliant data management systems research. Chapter 10 concludes the thesis.

Chapter 2

Querying Encrypted Data with TrustedDB

2.1 Chapter Overview

2.1.1 Background and Motivation

Encryption is commonly used to comply with privacy Regulations. Use of encryption for in-house systems is straight-forward. Data is stored encrypted. For processing, data is first decrypted.

For cloud-hosted services, decryption of data for processing raises concerns about data confidentiality. Several instances of illicit insider behavior and data leaks have resulted in the treatment of cloud providers as untrusted, third parties. Moreover, data confidentiality guarantees of cloud services are at best declarative and subject customers to unreasonable fine print clauses. For example, permitting the service provider to use customer behavior and content for commercial, profiling, or governmental surveillance purposes [71, 72]. Users are therefore hesitant to place sensitive data under the control of a remote, third-party provider without practical assurances of data confidentiality especially in business, healthcare and government frameworks.

To ensure data confidentiality for cloud services, existing research has focussed on new mechanisms that enable processing over encrypted data without the need for decryption. For example, homomorphic encryption [42, 104–106, 223, 253]. However, instances of such mechanisms are impractical [103]. The impracticality of homomorphic schemes arises not from implementation inefficiencies but is rooted in fundamental cryptographic hardness assumptions. The simplest of homomorphic schemes permits the addition of two encrypted numbers and requires at least one modular multiplication [214] operation. Even a single modular multiplication costs upwards of 30,000 picocents[1] (Section 2.4). In comparison, performing the equivalent operation on plaintext data costs less than one picocent (Section 2.4).

[1] 1 US picocent = 10^{-14} USD.

By focussing on specific operations, processing over encrypted data can be made relatively more efficient. For example, range query execution over outsourced encrypted data [92, 122, 123, 168, 212]. However, solutions designed for specific operations limit functionality and are not suitable in practice.

For regulatory compliant cloud services, novel mechanisms are needed that significantly lower the costs of data confidentiality in the cloud without limiting functionality.

2.1.2 Our Contribution: Low-cost, full-SQL Query Execution Over Encrypted Data

We posit that using cloud-hosted trusted hardware, data confidentiality can be provided at a fraction of the cost of current cryptography-based mechanisms. To validate our hypothesis, we conducted a detailed analysis of query processing costs for homomorphic encryption and trusted hardware. Our results (Figure 2.4) indicate that for linear and join query processing, use of trusted hardware costs 1-2 orders of magnitude less than software-only, cryptography-based mechanisms.

We realize the cost benefits of trusted hardware in TrustedDB. TrustedDB is a relational database that leverages server-hosted, tamper-proof, trusted hardware for full SQL execution over encrypted data. TrustedDB decrypts data before processing, thereby avoiding the high computation costs incurred by homomorphic schemes. However, data is decrypted only within the trusted hardware ensuring confidentiality.

Since TrustedDB processes plaintext data, there are no limitations of the type of query operations that can be performed. The current TrustedDB implementation supports range, aggregation, joins and nested queries (Section 2.5).

To realize TrustedDB, we designed new mechanisms that overcome the storage and processing limitations of trusted hardware. For example, we enable the server-hosted trusted hardware to transparently access external storage while preserving data confidentiality with on-the-fly encryption. Utilization of external storage eliminates the limitations on the size of databases that can otherwise be supported via trusted hardware alone. Further, in TrustedDB, client queries are preprocessed to identify sensitive and nonsensitive operations. Sensitive operations are processed inside the trusted hardware. Nonsensitive operations are offloaded to the untrusted host server. The splitting of query execution between trusted hardware and untrusted host server greatly improves performance. We also propose new query optimization techniques (Section 2.6) to ensure efficient workload balance between trusted hardware and host server.

2.1.3 Chapter Outline

Deployment model and adversarial assumptions are discussed in Section 2.2. Section 2.3 describes trusted hardware. Section 2.4 derives and analyzes the query processing costs of trusted hardware in comparison to homomorphic encryption. Section 2.5 details the TrustedDB architecture. Query optimization techniques for trusted hardware are described

Function	**Context**	**IBM 4764**	**P4**	**AMD A8-5500**
RSA sig.	1024 bits 2048 bits	848/s 316-470/s	261/s 43/s	1373/s 265/s
RSA verif.	1024 bits 2048 bits	1157-1242/s 976-1087/s	5324/s 1613/s	28806/s 10385/s
SHA-1	1KB blk. 64 KB blk.	1.42 MB/s 18.6 MB/s	80 MB/s 120 MB/s	269 MB/s 489.6 MB/s
3DES	1KB blk. 64 KB blk.	1.08 MB/s 7.73 MB/s	18 MB/s 17 MB/s	21.8 MB/s 22.8 MB/s
AES-128	1KB blk.	14 MB/s	100 MB/s	144 MB/s
DMA xfer	end-to-end	75-90 MB/s	1+ GB/s	2+ GB/s
CPU freq		233MHz	3400MHz	3200MHz
RAM		64MB	2 GB	4+ GB

Table 2.1: Sample Performance Data. IBM 4764-001 PCI-X is slower for general purpose computation than modern systems. Benchmarked using OpenSSL 0.9.7f.

in Section 2.6. Section 2.8 presents experimental evaluation of TrustedDB. TrustedDB demo application is introduced in Section 2.10. Finally, Section 2.11 concludes the chapter.

2.2 Model

Data is uploaded by a client to a relational database hosted with a remote, untrusted service provider. For confidentiality, sensitive attributes are encrypted before uploading to the service provider. Later, the client or an authorized third party queries the outsourced dataset through a SQL interface exposed by the provider.

We assume the provider to be honest-but-curious [51]. An honest-but-curious adversary performs all communication protocols and algorithms correctly. However, the adversary may attempt to compromise data confidentiality.

2.3 Trusted Hardware

TrustedDB implementation uses the IBM 4764 PCI-X [17] cryptographic coprocessor (SCPU). The 4764 is a PowerPC 405 based board with 64 MB memory and a 233 MHz processor. The operating system on SCPU board is embedded Linux. The SCPU offers several cryptographic operations, such as AES and DES symmetric key encryptions; RSA encryption on key lengths up to 4096 bits; pseudo random number generation; and hash functions. Crypto operations are implemented in SCPU hardware.

2.3.1 SCPU Security

Since the SCPU is designed for deployment in a remote, untrusted environment, client applications need guarantees that the remote SCPU is uncompromised. Specifically, client applications at all times need to be assured of the following.

- The remote SCPU has not been tampered with.
- The remote SCPU runs the correct software stack including the user application, operating system, and SCPU firmware.
- The client-SCPU communication channel is secure.

To prevent against physical tampering, the SCPU features a tamper-resistant design. The SCPU design has FIPS 140-2 level 4 [6] certification. The tamper proof SCPU enclosure detects tampering and as a response erases sensitive memory areas containing secrets keys. Once secret keys are erased the SCPU powers down.

To assure clients that the SCPU is running a trusted code stack, the SCPU produces an on-demand certificate. Using the certificate, a client can verify that the SCPU runs the correct code stack. The process of verifying SCPU state using the certificate is referred to as outbound authentication (OA) [235].

Outbound Authentication (OA)

The goal of outbound authentication (OA) is to prove to a client that the SCPU state is uncompromised. The proof is in the form of a certificate chain referred to as an OA certificate. The chain of trust is rooted in the manufacturer's private key.

To clarify, we describe the SCPU software stack and OA certification.

SCPU software stack: The SCPU-software architecture consists of four software layers running at differing levels of trust. The layers are numbered from 0 to 3. Layer 0 is referred to as Miniboot 0. layer 1 is referred to as Miniboot 1. Layers 2 and 3 are the operating system and user application, respectively.

OA Certification: Each SCPU software layer is issued a public-private key pair. The public key certificate of layer n is generated and certified by layer $n-1$. The Layer 0 public key certificate is signed using the manufacturer's private key. The Layer 1 public key certificate is signed using the layer 0 private key. The Layer 2 public key certificate is signed using the layer 1 private key. Finally, the Layer 3 public key certificate is signed using the layer 2 private key. The layer-to-layer certification process builds a chain of trust rooted in the manufacturer's private key. The chain of public key certificates constitutes the OA certificate.

The certificate for each layer also contains a signed crypto hash of the software installed in that layer. By verifying the hashes against previously published values, a client can ensure validity of software executing in all SCPU layers. If the software in an SCPU layer n is updated as part of maintenance, then the key pairs for all layers $\geq n$ are re-generated. The next time a client requests authentication, the OA certificate sent to the client will reflect the updated certificate chain.

Client-SCPU communication channel is secured as follows. An application running inside the SCPU is granted a public-private key pair. The application public key is certified by the operating system. Clients obtain the application public key using outbound authentication. Sharing the application public key with clients enables the setup of secure client-SCPU communication channels similar to SSL communication between a web-browser and server.

2.3.2 SCPU-based Application Development Challenges

The IBM 4764 SCPU presents significant challenges in designing and deploying custom code to be run within its enclosure. We discuss the challenges here along with our approaches to overcome them.

Large-scale applications such a databases require significant storage space. However, the SCPUs have no persistent storage other than a 128KB battery backed memory. Hence, we designed a custom I/O library that enables applications executing within the SCPU to transparently access external storage.

The SCPU communicates with the host server synchronously through fixed sized messages exchanged over the PCI-X bus. Interfacing such a synchronous channel with the communication model of a database engine required the development of a custom paging module. The paging module performs message translation and enables SCPU-based applications to use the SCPU-server communication channel via normal function calls.

The SCPU's cryptographic hardware engine features a set of latencies that effectively cripple the ability to encrypt and decrypt small amounts of data, less than 1KB. To reduce the hardware latencies, we ported several cryptographic primitives, such as encryption and hashing to run on the SCPU's main processor instead. For data sizes $>$ 1KB the SCPU hardware crypto engine is used.

2.4 The Costs of Data Confidentiality

Encryption is typically used to maintain data confidentiality. In an outsourced setting, data is encrypted before uploading to a remote, untrusted service provider. Once uploaded, three solutions can be envisioned for processing over the encrypted data.

- Solution A: Client downloads data from the provider, decrypts the data, and processes the decrypted data.

- Solution B: Deploy cryptographic constructs to process encrypted data server-side without the need for decryption.

- Solution C: Process the encrypted data server-side inside tamper-proof enclosures of trusted hardware, which the clients' trust.

In this section, we compare the per query costs of solutions A, B, and C. We show that per query execution cost of solution C is 1-3 orders of magnitude lower than solutions A and B

	H, S	**M**	**L**
monthly	$44.90	$95	$13
bandwidth (d/u)	15/5 Mbps	per 1Mbps	per 1Mbps
dedicated	No	Yes	Yes
picocent/bit	115/345	3665	500

Figure 2.1: Network service costs [61–63]. H = home, S = small enterprise, M = medium enterprise, L = large cloud provider.

(Figure 2.4). That is, computation inside secure hardware processors is orders of magnitude cheaper than any equivalent cryptographic operation performed on the provider's unsecured hardware. Also, due to the extremely high costs of data transfer, the overhead of transferring data back to the client is significantly more expensive than using cryptography.

We derive per query execution costs as follows:

- Step 1: Using prior work [61–63], we compute client-side and server-side CPU cycle costs (Section 2.4.1). In addition, we also derive the per unit, client-server data transfer costs.

- Step 2: We compute per query CPU cycles consumed and the total data transferred for each of the solutions A, B, and C (Section 2.4.2).

- Step 3: Using the results from steps 1 and 2, we calculate the per query execution costs for each solution (Section 2.4.3).

The key insights for lower costs of trusted hardware as compared to solutions B and C are the following. When data is outsourced, the extremely expensive network traffic often dominates. Transferring a single bit of data between cloud and clients costs upwards of 3500 picocents [62]. The high data transfer costs make solution A very expensive.

Due to economies of scale, provider-side CPU cycles are inexpensive costing less than 0.5 picocents per cycle. In comparison, cost of a CPU cycle in trusted hardware is 56 picocents. However, despite the higher cost of trusted hardware CPU cycle, overall query processing costs for trusted hardware are much lower.

The lower costs of trusted hardware-based query processing result from the significantly less CPU cycles consumed as compared to cryptography that allows processing over encrypted data without decryption. Cryptography that allows processing on encrypted data demands extremely large numbers of cycles even for very simple operations such as addition. The simplest of cryptographic schemes permits the addition of two encrypted numbers and requires at least one modular multiplication [214] operation. Even a single modular multiplication costs upwards of 30,000 picocents (Section 2.4). On the other hand, in trusted hardware, data is decrypted before processing. The decryption of data can be achieved at much lower costs (Section 2.4.3).

Parameters	**H**	**S**	**M**	**L**
Scale	<10	<1000	<10k	>10k
CPU utilization	5-8%	10-12%	15-20%	40-56%
server:admin ratio	N.A.	100-140	140-200	800-1000
Space ($/sqft/month)	N.A.	0.5	0.5	0.25
PUE	N.A.	2-2.5	1.6-2	1.2-1.5
Hardware ($/CPU)	750	500	500	350
Electricity ($/KW)	0.09	0.07	0.07	0.06
CPU Cycle (picocent)	5	14-27	2	<0.5

Figure 2.2: Costs and key parameters for different computing environments [61–63]. H = home, S = small enterprise, M = medium enterprise, L = large cloud provider.

2.4.1 Cost of Primitives

Compute Cycles and Networks

Chen et al. [61–63] derived the cost of CPU cycles for a set of environments ranging from individual homes with a few PCs to clouds running tens of thousands of CPUs. Their cost analysis takes into account several cost factors such as hardware procurement, floor space leasing, energy consumption, personnel, and administration. Figures 2.1 and 2.2 summarize the cost analysis, which concludes that due to economies of scale CPU cycle costs decrease significantly in large cloud environments.

To validate the results of Chen et al., we compare the costs from Figures 2.1 and 2.2 with today's cloud offerings. We find that the costs derived by Chen et al. are very close to pricing offered by cloud providers today. Amazon for example, charges 1-2.5 picocents per cycle. The pricing includes Amazon's markup. Rackspace.com's CPU cycles range from 0.3-2.4 picocents. Chen et al. estimate cloud-hosted CPU cycles to cost <0.5 picocents per cycle.

Amazon's network service ranges from 800 to 1500 picocents per bit depending on source and destination region. Chen et al. estimate client to cloud per bit transfer cost to be 500 picocents without the markup.

Trusted Hardware

To evaluate the cost of SCPU cycles, we use the cost equation designed by Chen et al. [62]. The cost equation requires as input the CPU frequency, energy consumption, service, and procurement costs. For simplicity we consider only one SCPU per main CPU. To lower costs, multiple SCPUs can be installed per server[2].

The IBM 4764 SCPU runs a Motorola PowerPC 405 RISC CPU at 233 MHz. For consistency, we normalize SCPU cycles to CISC x86 cycles. We performed extensive benchmarking (Table 2.1) and identified that branching, integer ops and memory access performance of

[2] Up to 4 SCPUs can be installed on a single server.

SCPU are almost identical to an x86 AMD K6 model 6 at 200 MHz with 512 KB L2-cache, which in turn performs equivalently to a Pentium II core at 166 MHz [10].

SCPU energy consumption peaks by design at only 25W. We conservatively estimate that the deployment of SCPUs effectively doubles energy and service costs. The conservative estimate helps to defend against any critique claiming that in effect the main CPU will need to be dedicated as a communication conduit for the SCPU.

SCPUs are expensive. At the time of writing the IBM 4764 could be purchased for around $8,000 at retail, which is the number we deploy in our estimation. However, unofficial bulk pricing is around $5,000 excluding support services.

Substituting the SCPU frequency, energy consumption, and purchase cost in the cost equation of [61–63], we derive the cost of CPU cycles inside cloud-hosted SCPUs to be 56 picocents. We note that 56 picocents is indeed much higher than the < 0.5 picocent cost of a cycle on cloud commodity hardware. However, the SCPU cycle cost is comparable to CPU cycle costs for small sized enterprises, which is 14-27 picocents (Table 2.2).

2.4.2 Cost Comparison

Equipped with CPU cycle and data transfer costs, we proceed to compute the per query execution costs of solutions A, B, and C. We consider the following simple scenario: a client outsources an encrypted, integer dataset to a service provider. The encrypted data is then subjected to a simple aggregation query, which requires the server to add all integers and return the result to the client. We chose the aggregation scenario not only for its illustrative simplicity but also because sum aggregation is one of the very few types of queries for which cryptography-based solutions have been proposed in existing research. Using the aggregation scenario allows us to directly compare with existing work. In Section 2.4.3, we compare the costs for general select and join query processing.

Querying un-encrypted data: No confidentiality

As a baseline consider the most prevalent scenario today in which the client's data is stored unencrypted with the service provider. Client queries are executed entirely on the provider's side and only the results are transferred back to the client. The lower bound cost of query execution on unencrypted data is computed as follows[3]:

$$\begin{aligned} Cost_{unencrypted} &= Cost\ of\ addition\ operations\ on\ server\ + \\ &\quad Cost\ of\ transmitting\ results \\ &= \left(\frac{N}{D} - 1\right) \cdot C_{cycle\_server} \cdot \eta_{addition} + \\ &\quad 2 \cdot D \cdot C_{bit\_transmit} \end{aligned} \tag{2.1}$$

[3] The cost of reading data from storage into main memory is a common factor in all solutions and thus not included here.

where N is the size of the entire database in bits, $D = 32$ for 32 bit integer dataset, C_{cycle_server} is the cost of one CPU cycle on server hardware, $\eta_{addition} = 1$ is the average number of CPU cycles required for an addition operation [139]. $C_{bit\_transmit}$ is the cost of transmitting 1 bit of data from the service provider to the client.

Solution A: Transferring encrypted data to client

Solution A provides data confidentiality. In solution A, the entire encrypted database is transferred to the client. The client decrypts the database and performs the aggregation locally. The cost of solution A is as follows:

$$\begin{aligned} Cost_{transfer} &= Cost\ of\ data\ transmission\ + \\ &\quad\ Cost\ of\ decryption\ on\ client\ + \\ &\quad\ Cost\ of\ addition\ operations\ on\ client \\ &= N \cdot C_{bit\_transmit} + \\ &\quad\ N \cdot C_{bit\_decryption} + \\ &\quad\ \left(\frac{N}{D} - 1\right) \cdot C_{cycle\_client} \cdot \eta_{addition} \end{aligned} \tag{2.2}$$

Where $C_{bit\_decryption} = 8$ picocents is the (normalized) cost of decrypting one bit with AES-128 in a medium-sized (M) enterprise and $C_{cycle\_client} = 2$ picocents is the cost of a single client CPU cycle in medium sized enterprises. Here, the cost of transferring the database to the client dominates.

Solution B: Cryptography

For server-side processing of aggregation query, additive homomorphisms [198,199,223] have been proposed in existing work [123, 249]. Additive homomorphic encryption allows the computation of the encryption of the sum of a set of encrypted values without requiring decryption.

Homomorphic schemes require at least a modular multiplication in performing the addition operation. Moreover, for security, the modular multiplication needs to be performed in fields with a large modulus. For efficiency, Ge et al. [249] propose to perform aggregation in parallel by simultaneously adding multiple 32-bit integer values. In parallel addition, two 1024-bit chunks of encrypted data are added at a time. Due to the properties of the Paillier cryptosystem used, each 1024 bit addition involves one 2048-bit modular multiplication[4].

The server computes the encrypted sum of all 1024-bit integers and returns the 2048 bit result to the client. The client decrypts the result into a 1024 bit plaintext, splits the plaintext into 32 32-bit integers, and computes the final sum. The cost of solution B is as follows:

[4] To process n-bit plaintexts, Paillier operates in $n^2 = 2048$ bit fields for 1024 bit plaintexts. Ciphertexts are 2048 bit.

$$\begin{aligned} Cost_{homomorphic} &= Cost\ of\ Modular\ Multiplications\ on\ server\ + \\ &\quad Cost\ of\ data\ transmission\ + \\ &\quad Cost\ of\ decryption\ on\ client\ + \\ &\quad Cost\ of\ addition\ operations\ on\ client \\ &= \frac{B_h}{D} \cdot \left(\left(\frac{N}{B_h} - 1 \right) \cdot C_{modular\_mul} + \right. \qquad (2.3) \\ &\quad 2 \cdot B_h \cdot C_{bit\_transmit} + \\ &\quad C_{homomorphic\_dec} + \\ &\quad \left. \left(\frac{B_h}{D} - 1 \right) \cdot C_{cycle\_client} \cdot \eta_{addition} \right) \end{aligned}$$

where $B_h = 1024$ is the plaintext block size and $C_{modular\_mul}$ is the cost of performing a single modular multiplication modulo 2048 on the server. $C_{homomorphic\_dec}$ is the cost of performing the single decryption on client and involves modular multiplication and exponentiation.

Solution C: SCPUs

A possible use of a SCPU for aggregation is to perform the aggregation completely inside the server-side SCPU. The result is then reencrypted and transmitted to the client.

In addition to the core CPU processing costs, data transfer overheads are incurred to bring encrypted data into the SCPU and then to transfer the encrypted results back to the client via the host server. The total cost of solution C is as follows:

$$\begin{aligned} Cost_{scpu} &= Cost\ of\ data\ transmission\ between\ the\ host\ server\ and\ SCPU\ + \\ &\quad Cost\ of\ decryption\ inside\ SCPU\ + \\ &\quad Cost\ of\ addition\ operations\ inside\ SCPU\ + \\ &\quad Cost\ of\ encryption\ inside\ SCPU\ + \\ &\quad Cost\ of\ data\ transmission\ from\ server\ to\ client\ + \\ &\quad Cost\ of\ decryption\ at\ client \\ Cost_{scpu} &= \left\lceil \frac{N}{B_s} \right\rceil \cdot (\delta_{srv} \cdot C_{cycle\_srv} + \delta_{scpu} \cdot C_{cycle\_scpu})\ + \qquad (2.4) \\ &\quad N \cdot C_{bit\_decryption\_scpu} + \\ &\quad \left(\frac{N}{D} - 1 \right) \cdot C_{cycle\_scpu} \cdot \eta_{addition\_scpu} + \\ &\quad B_c \cdot C_{bit\_encryption\_scpu} + \\ &\quad B_c \cdot C_{bit\_transmit} + \\ &\quad B_c \cdot C_{bit\_decyption\_client} \end{aligned}$$

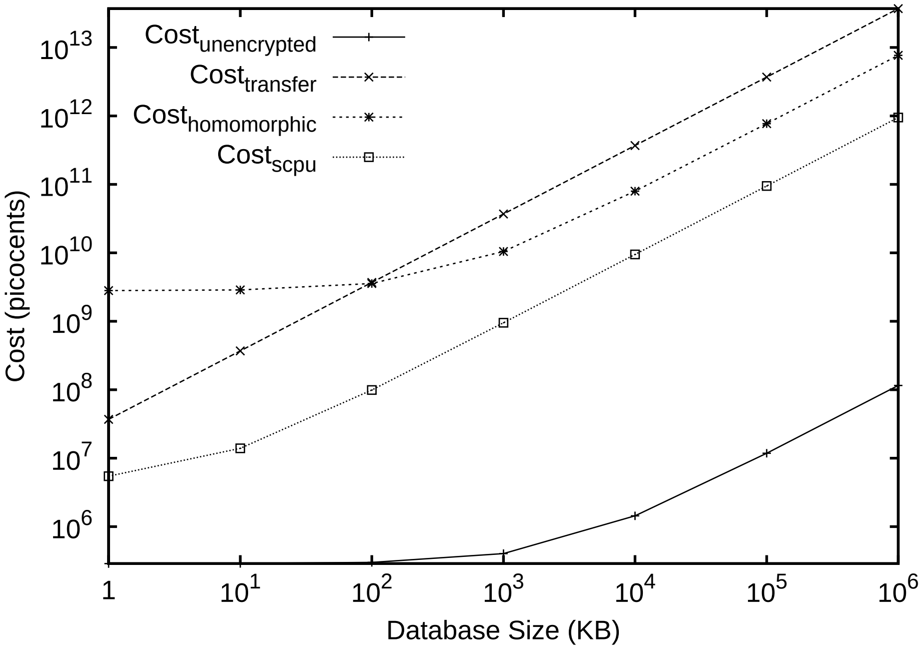


Figure 2.3: Comparison of outsourced aggregation query solutions.

Where δ_{srv} and δ_{scpu} are the server and SCPU cycles used to setup data transfer and include the cost of setting up and handling DMA interrupts. C_{cycle_scpu} is the cost of a SCPU cycle. $B_s = 64KB$ is the block size of data transmitted between the server and the SCPU in one round and B_c is the cipher block size (128 bits for AES). $\eta_{addition\_scpu} = 2$ is the number of cycles per addition operation in the SCPU for 64 bit addition (on a 32 bit architecture).

Cost Comparison Results

Figure 2.3 shows the cost relationship between solutions A, B, and C. As can be seen, for all data set sizes, $Cost_{scpu} < Cost_{homomorphic}$ and $Cost_{scpu} < Cost_{transfer}$. Also, note that for data sets of size $< 100KB$, the cost of client-side homomorphic decryptions, which involves modular exponentiation, dominates and exceeds the data transmission cost in $Cost_{transfer}$. Overall, the use of SCPUs is the most efficient from a cost-centric point by an order of magnitude as compared to cryptographic alternative.

Cost vs. Performance

Given the order of magnitude cost advantage of SCPU over cryptography-based mechanism, we expect that for the aggregation scenario [249], the SCPU's overall performance will also be at least comparable if not better. We experimentally evaluated the SCPU-based approach and achieved a throughput of about 1.07 million tuples/second for the SCPU. By contrast, best-case scenario throughputs of homomorphic scheme ranges between 0.58 and 0.92 million tuples/second [249].

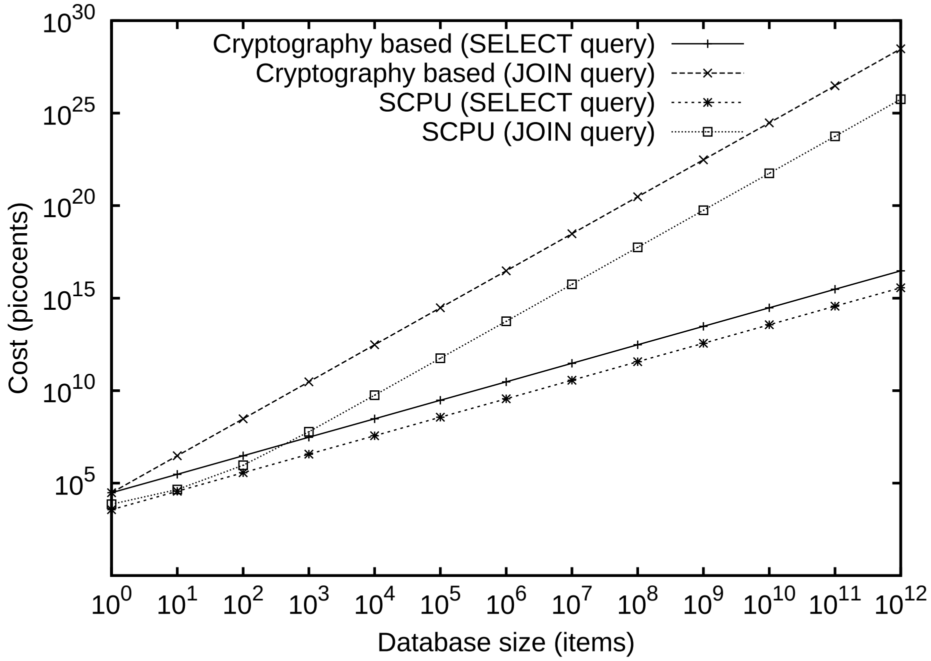


Figure 2.4: Cost comparison of SCPU-based and cryptography-based query processing. SCPU-based query processing is 1-3 orders of magnitude cheaper.

2.4.3 Generic Select and Join Query Processing Costs

Current cryptographic constructs that guarantee data confidentiality are based on trapdoor functions [110]. Currently viable trapdoors are based on modular multiplication and exponentiation in large fields, for example, 2048 bit modular operations. A single modular multiplication operation costs 30,000 picocents [61]. Thus, even if we assume that in the future, homomorphic schemes are invented that allow full Turing Machine languages to be run under the encryption envelope, unless new trapdoor math is discovered, each operation will cost at least 30,000 picocents when run on commodity servers. By comparison, SCPUs process data at a cost of 56 picocents per cycle[5].

To compute general query processing costs, we also need to account for the fact that SCPUs need to read data in before processing. The IBM 4764 SCPUs feature a decryption throughput of about 10-14 MB/s for AES decryption [17]. The decryption throughput limits the ability to process data. At 166-200 megacycles/second, the decryption latency results in the SCPU having to idly wait anywhere between 47 and 80 cycles for decryption to happen in the crypto engine module before the SCPU can process data. The wait time in effect results in an amortized SCPU decryption cost between 2632 and 4480 picocents, 3556 picocents on average.

The decryption cost of 3556 picocents is derived for the case wherein the SCPU has only

[5] While ECC signatures (e.g., even the weak ECC-192) may be faster, ECC-based trapdoors would be even more expensive, as they would require two point multiplications, coming at a price tag of least 780,000 cycles [81].

enough memory to store the operands of a binary operation. In the presence of significantly higher, realistic amounts of SCPU memory, for example, 32 MB for the 4764, optimizations can be achieved for certain types of queries such as relational joins. For joins, the SCPU can read in and decrypt data pages instead of individual data items and run the join query over as many of the decrypted data pages as would fit in SCPU memory at one time.

Loading bulk data pages at a time results in significant cost savings. To illustrate, consider a page size P of 32-bit words and a nested join algorithm for two tables of size N 32-bit integers each. For the nested join, the SCPU will perform $(N/P)^2 + (N/P)$ page fetches each fetch also involving a page decryption at a cost of $P \cdot 3556$ picocents[6]. Thus, we get a total join query processing cost of $(\frac{N^2}{P} + N) \cdot 3556 + N^2 \cdot 56$. As shown in Figure 2.4, for reasonable page sizes, such as $P = M/2/4 = 4$ million 32-bit words, join query processing cost becomes 3 orders of magnitude lower than the $N^2 \cdot 30000$ picocent cost incurred in the cryptography-based solution.

Cost-Analysis Summary

As Figure 2.4 illustrates, for linear processing queries, such as select queries, the SCPU-based solution is roughly 1+ order of magnitude cheaper than any cryptography-based mechanisms. For join queries, the SCPU-based costs drop even further even when assuming no available memory. Finally, in the presence of realistic amounts of SCPU memory, SCPU-based join processing is 3 orders of magnitude cheaper than software-only cryptographic solutions on commodity server hardware.

2.5 TrustedDB Architecture

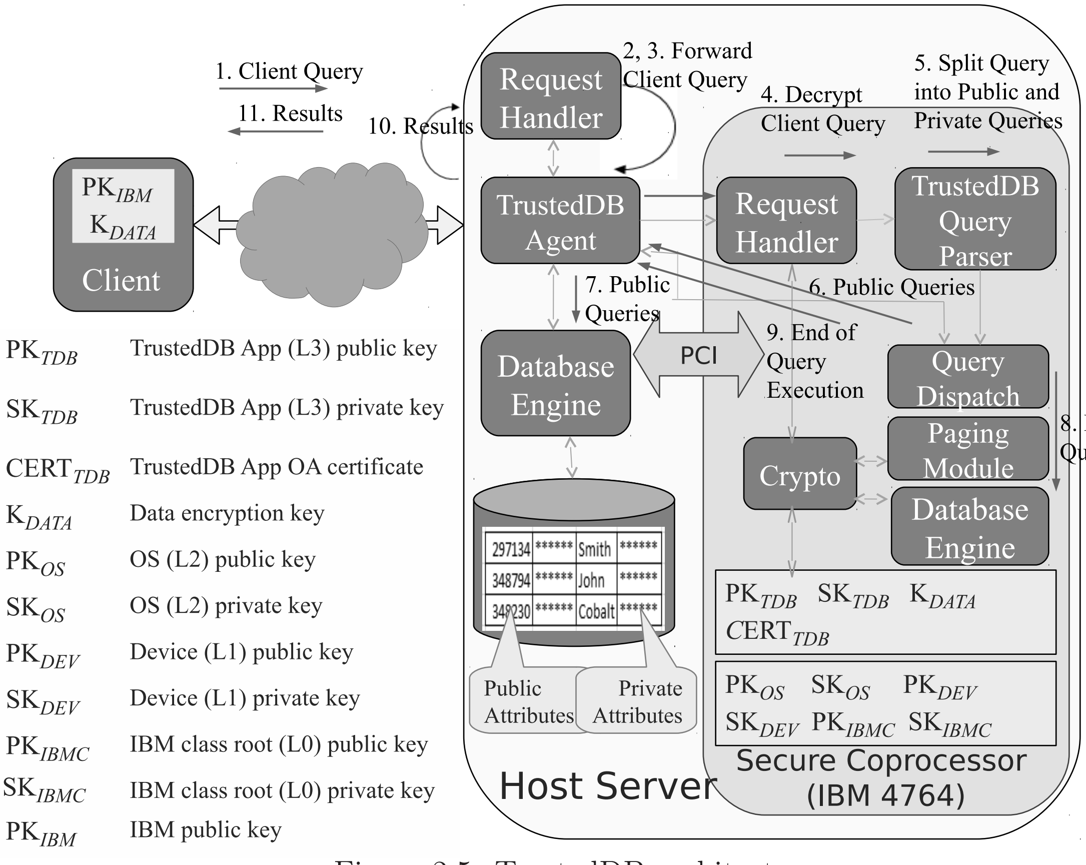
In the previous section, we showed the cost benefits of trusted hardware for data confidentiality. We now describe the TrustedDB architecture. TrustedDB uses the IBM 4764 SCPU for full-SQL query execution over encrypted data.

TrustedDB is built around a set of core components (Figure 2.5) including server and SCPU request handlers; a processing agent; a query parser; a paging module; a query dispatch module; a cryptography library; and server and SCPU-side database engines.

The SCPU DBMS engine is a heavily modified PowerPC ported SQLite core, which uses the paging module to extend the database buffer pool with external storage. The TrustedDB paging module traps I/O requests made by the SCPU DBMS engine. The paging module then interacts with the external TrustedDB agent to fetch and store the relevant database pages.

The main CPU DBMS is an unmodified MySQL 14.12 Distrib 5.0.45 engine. The main database engine can be substituted with any other SQL engine with changes related only to SQL syntax.

[6] Recall that decrypting 32-bits incurs an amortized cost of around 3556 picocents.



PK_{TDB}	TrustedDB App (L3) public key
SK_{TDB}	TrustedDB App (L3) private key
$CERT_{TDB}$	TrustedDB App OA certificate
K_{DATA}	Data encryption key
PK_{OS}	OS (L2) public key
SK_{OS}	OS (L2) private key
PK_{DEV}	Device (L1) public key
SK_{DEV}	Device (L1) private key
PK_{IBMC}	IBM class root (L0) public key
SK_{IBMC}	IBM class root (L0) private key
PK_{IBM}	IBM public key

Figure 2.5: TrustedDB architecture

2.5.1 Overview

A client defines a relational database schema and uploads data. Sensitive attributes are marked using the "SENSITIVE" keyword in DDL statements. For example, following is a create statement wherein customer name and address are marked as sensitive attributes.

```
CREATE TABLE customer(ID integer primary key,
Name char(72) SENSITIVE, Address char(120) SENSITIVE);
```

A client-side library transparently encrypts sensitive attributes.

Query execution comprises of the following steps (Figure 2.5).

- Step 1: A client sends a query request to the host server through a standard SQL interface. The query is transparently encrypted at the client site using the SCPU public key. The host server thus cannot decrypt the query.

- Steps 2,3: The host server forwards the encrypted query to the SCPU request handler.

- Step 4: The SCPU request handler decrypts and forwards the client query to the query parser.

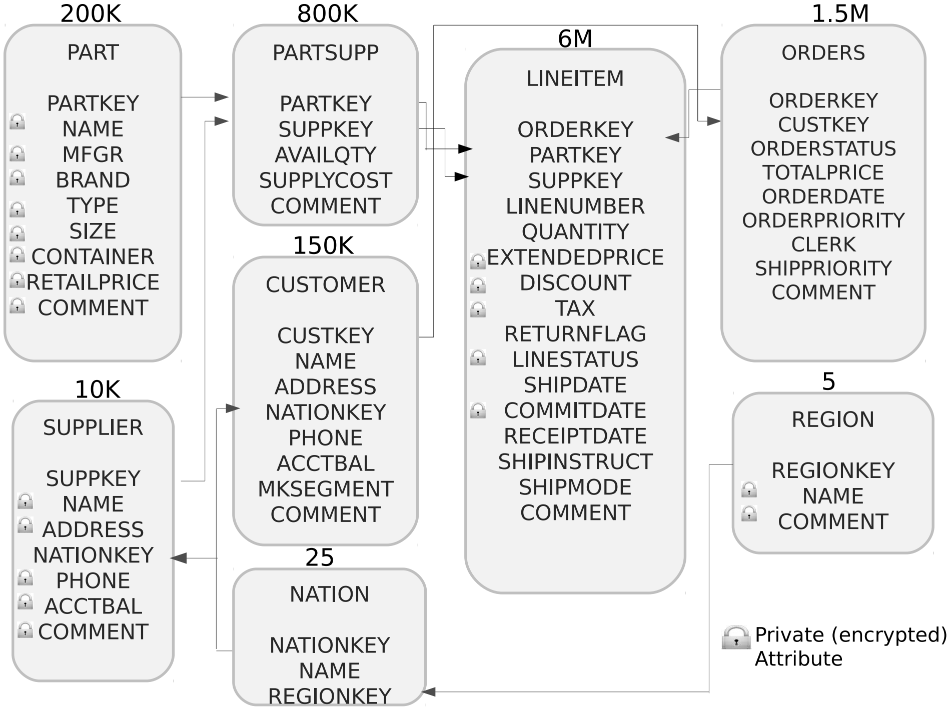


Figure 2.6: Database Schema

- Step 5: The decrypted query is parsed and rewritten as a set of sub-queries. Each subquery is identified as being either public or private. Public queries access only nonsensitive attributes and can be executed entirely on the host server.

- Steps 6,7,8: The query dispatcher forwards public queries to the host server and private queries to the SCPU database engine. The host server executes public queries. The SCPU database engine executes private queries. Dependencies are handled by the query dispatcher. The net result is that the maximum possible work is run on the host server's cheap cycles.

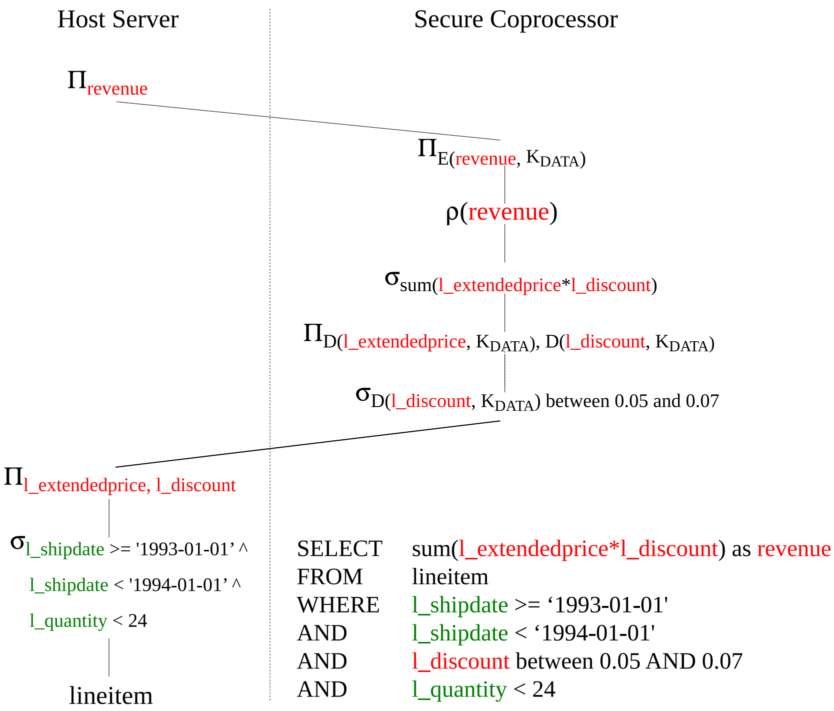
- Step 9: The final query result is assembled and encrypted by the SCPU query dispatcher.

- Steps 10,11: The encrypted query result is sent to the client.

2.5.2 Query Parsing

Outline

Sensitive attributes can occur anywhere within the select clause, where clause, group-by clause, aggregation operators, or within sub-queries. The TrustedDB query parser's job is then:


Host Server
Secure Coprocessor
$\Pi_{revenue}$
$\Pi_{E(revenue,\ K_{DATA})}$
$\rho(revenue)$
$\sigma_{sum(l\_extendedprice*l\_discount)}$
$\Pi_{D(l\_extendedprice,\ K_{DATA}),\ D(l\_discount,\ K_{DATA})}$
$\sigma_{D(l\_discount,\ K_{DATA})\ between\ 0.05\ and\ 0.07}$
$\Pi_{l\_extendedprice,\ l\_discount}$
$\sigma_{l\_shipdate >= '1993-01-01' \wedge l\_shipdate < '1994-01-01' \wedge l\_quantity < 24}$
lineitem

```
SELECT   sum(l_extendedprice*l_discount) as revenue
FROM     lineitem
WHERE    l_shipdate >= '1993-01-01'
AND      l_shipdate < '1994-01-01'
AND      l_discount between 0.05 AND 0.07
AND      l_quantity < 24
```

Figure 2.7: TrustedDB query plan for TPC-H query Q6

- To ensure that any processing involving private attributes is done within the SCPU. All private attributes are encrypted using a shared data encryption key between the client and the SCPU (Section 2.5.3). Hence, the host server cannot decipher these attributes.

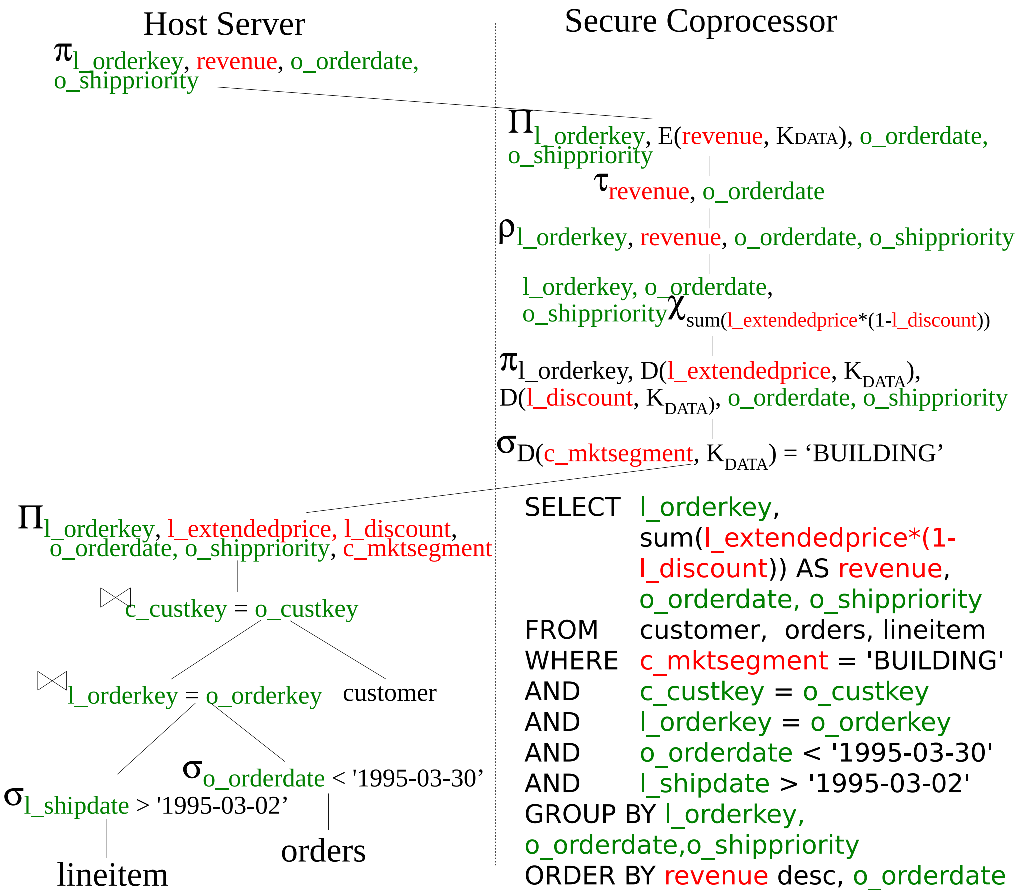
- To optimize query rewrites, such that processing on the host server is maximized.

To exemplify how public and sensitive queries are generated from the original client query, we use examples from the TPC-H benchmark [12]. TPC-H does not classify attributes based on security. Therefore, we define an attribute classification into sensitive and nonsensitive attributes. The resultant schema is listed in Figure 2.6. In summary, all attributes that convey identifying information about customers, suppliers and parts are considered private. The query plans for TPC-H queries Q3, Q4, and Q6 are illustrated in Figures 2.8,2.9 and 2.7, respectively.

Select Query

Queries with where clause conditions on nonsensitive attributes only, are processed entirely by the server. No private queries are generated by the query parser if sensitive attributes do not occur within a where clause and no computation is performed on sensitive attributes in the select clause other than projection.

Queries with where clause conditions on both sensitive and nonsensitive attributes are parsed into a public subquery and a private subquery. The public subquery is first processed


Host Server
Secure Coprocessor
π l_orderkey, revenue, o_orderdate, o_shippriority
Π l_orderkey, E(revenue, KDATA), o_orderdate, o_shippriority
τ revenue, o_orderdate
ρ l_orderkey, revenue, o_orderdate, o_shippriority
l_orderkey, o_orderdate, o_shippriority χ sum(l_extendedprice*(1-l_discount))
π l_orderkey, D(l_extendedprice, K DATA), D(l_discount, K DATA), o_orderdate, o_shippriority
σ D(c_mktsegment, K DATA) = 'BUILDING'
Π l_orderkey, l_extendedprice, l_discount, o_orderdate, o_shippriority, c_mktsegment
⋈ c_custkey = o_custkey
⋈ l_orderkey = o_orderkey
customer
σ o_orderdate < '1995-03-30'
σ l_shipdate > '1995-02-02'
orders
lineitem

```
SELECT   l_orderkey,
         sum(l_extendedprice*(1-
         l_discount)) AS revenue,
         o_orderdate, o_shippriority
FROM     customer,  orders, lineitem
WHERE    c_mktsegment = 'BUILDING'
AND      c_custkey = o_custkey
AND      l_orderkey = o_orderkey
AND      o_orderdate < '1995-03-30'
AND      l_shipdate > '1995-03-02'
GROUP BY l_orderkey,
o_orderdate,o_shippriority
ORDER BY revenue desc, o_orderdate
```

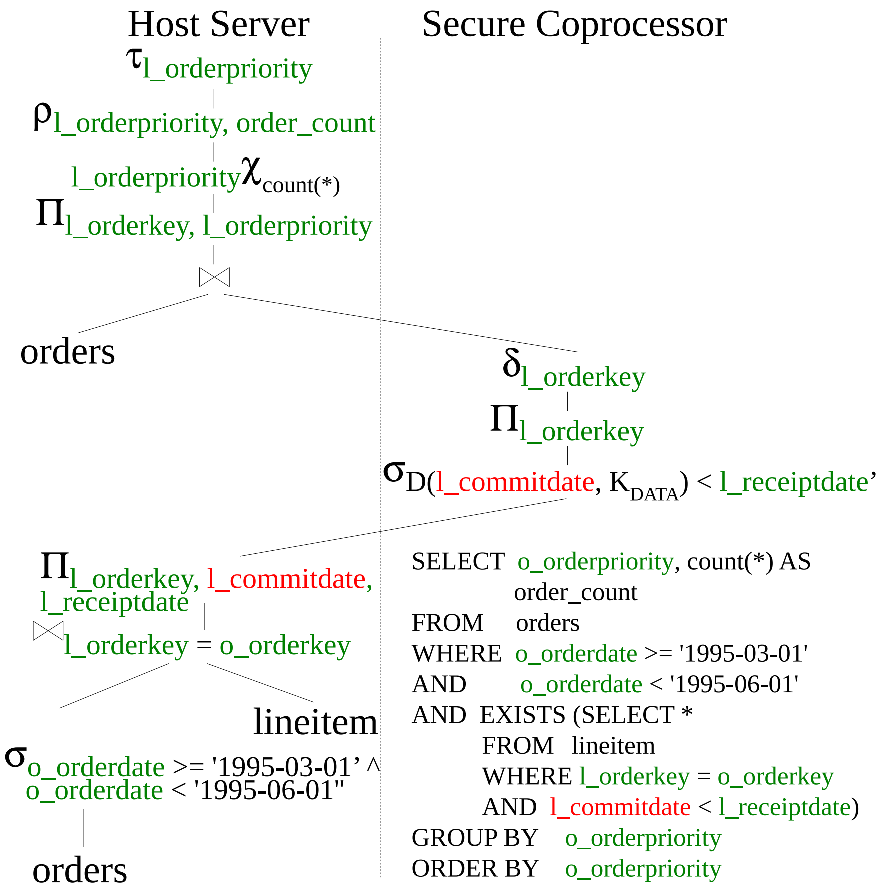
Figure 2.8: TrustedDB query plan for TPC-H query Q3

on the host server. The private subquery is executed by the SCPU database engine over the intermediate results of the public subquery. For example, query $Q6$ of the TPC-H benchmark is parsed as shown in Figure 2.7. The host server first executes a public query that filters all tuples which fall within the desired *ship date* and *quantity* range, both *ship date* and *quantity* being nonsensitive. The public query result is then used by the SCPU to filter the tuples that meet the condition on the sensitive *discount* attribute.

Note that the execution of private queries depends on the results from the execution of public queries and vice-a-versa even though they execute in separate database engines. The dependency is handled by the TrustedDB query dispatcher in conjunction with the paging module.

Aggregation

An aggregation clause involving sensitive attributes is processed by the SCPU database engine. For example, execution of TPC-H query Q6 as illustrated in Figure 2.7. The host server first executes a public query that filters all tuples which fall within the desired *ship date* and *quantity* range. The result public query result is then used by the SCPU to perform the aggregation on the private attributes *extended price* and *discount*. For aggregation,


Host Server
Secure Coprocessor
τ l_orderpriority
ρ l_orderpriority, order_count
l_orderpriority χ count(*)
Π l_orderkey, l_orderpriority
⋈
orders
δ l_orderkey
Π l_orderkey
σ D(l_commitdate, K DATA) < l_receiptdate'
Π l_orderkey, l_commitdate, l_receiptdate
⋈ l_orderkey = o_orderkey
lineitem
σ o_orderdate >= '1995-03-01' ^ o_orderdate < '1995-06-01"
orders

```
SELECT  o_orderpriority, count(*) AS
            order_count
FROM     orders
WHERE  o_orderdate >= '1995-03-01'
AND       o_orderdate < '1995-06-01'
AND  EXISTS (SELECT *
          FROM   lineitem
          WHERE l_orderkey = o_orderkey
          AND  l_commitdate < l_receiptdate)
GROUP BY    o_orderpriority
ORDER BY    o_orderpriority
```

Figure 2.9: TrustedDB query plan for TPC-H query Q4

sensitive attributes are decrypted inside the SCPU. Since the aggregation operation results in a new attribute computed from sensitive attributes, the result is reencrypted before being sent to the client. The reencryption is also done within the SCPU.

Group-by and Order-by

If the client query specifies a group-by or an order-by clause on nonsensitive attributes but the select clause includes an aggregation on sensitive attributes, the grouping or sort operation is performed inside the SCPU. Figure 2.8 illustrates the case for the TPC-H query $Q3$. If the aggregation does not involve any sensitive attributes, then the host server performs all the group-by and sort operations.

Nested Queries

Each nested query is parsed and processed separately. The input dataset for a nested query are the intermediate results of public sub-queries. For example, nested query execution of TPC-H query Q4 as shown in Figure 2.9.

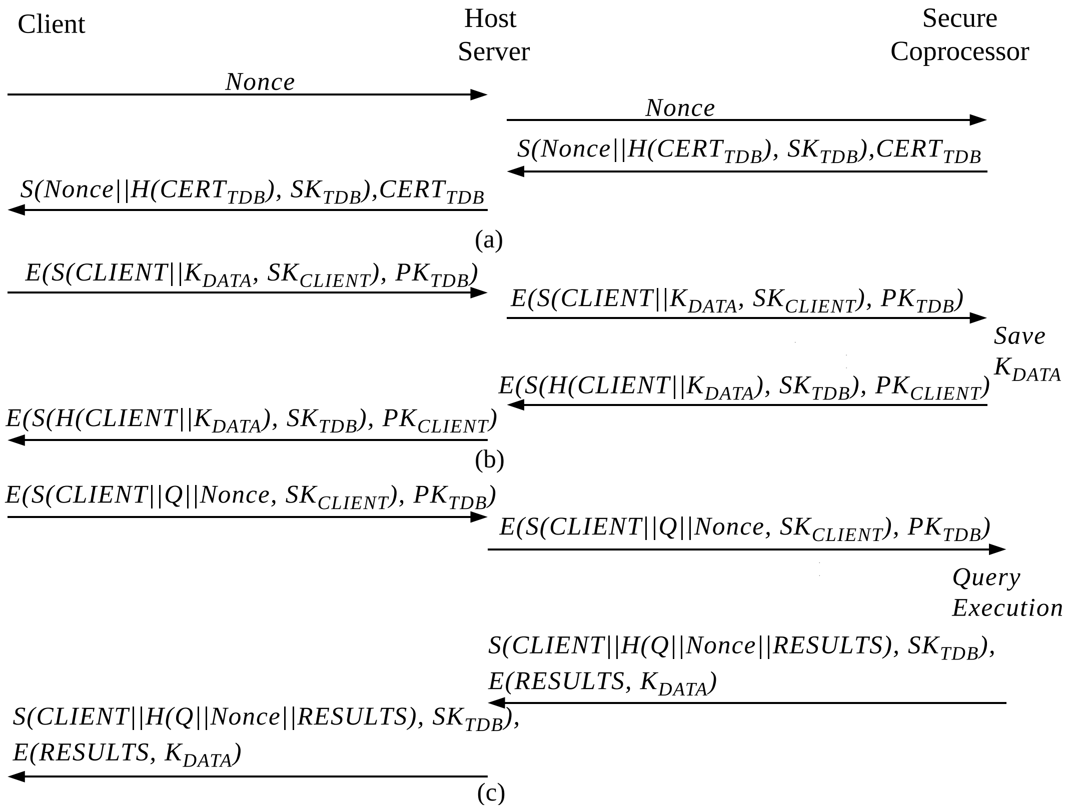

Client
Host Server
Secure Coprocessor
Nonce
Nonce
$S(Nonce||H(CERT_{TDB}), SK_{TDB}),CERT_{TDB}$
$S(Nonce||H(CERT_{TDB}), SK_{TDB}),CERT_{TDB}$
(a)
$E(S(CLIENT||K_{DATA}, SK_{CLIENT}), PK_{TDB})$
$E(S(CLIENT||K_{DATA}, SK_{CLIENT}), PK_{TDB})$
Save K_{DATA}
$E(S(H(CLIENT||K_{DATA}), SK_{TDB}), PK_{CLIENT})$
$E(S(H(CLIENT||K_{DATA}), SK_{TDB}), PK_{CLIENT})$
(b)
$E(S(CLIENT||Q||Nonce, SK_{CLIENT}), PK_{TDB})$
$E(S(CLIENT||Q||Nonce, SK_{CLIENT}), PK_{TDB})$
Query Execution
$S(CLIENT||H(Q||Nonce||RESULTS), SK_{TDB})$, $E(RESULTS, K_{DATA})$
$S(CLIENT||H(Q||Nonce||RESULTS), SK_{TDB})$, $E(RESULTS, K_{DATA})$
(c)

Figure 2.10: (a) Acquiring Outbound Authentication certificate. (b) Setup of Data Encryption Key. (c) Secure Transmission of query results. Notations: $E(M, K)$ denotes encryption of message M with key K. PK_{ALICE} denotes a public key that belongs to Alice while SK_{ALICE} represents Alice's private key. $S(M, K)$ denotes signature of message M with private key K. The cryptographic hash of message M is denoted by $H(M)$. $||$ represents concatenation.

For nested queries, additional care is taken to limit the data transfer between host server and the SCPU, which may result in sub-optimal performance. For instance, the query plan of Figure 2.9 runs the removal of duplicates on attribute *order key* within the SCPU. An alternative would be to perform duplicate removal on the host server. The choice for SCPU-side duplicate removal is made to reduce the traffic over the PCI interface.

2.5.3 TrustedDB Security

In Section 2.3.1, we discussed the physical SCPU security and client-side verification of the SCPU software stack. Here, we detail the client-SCPU communication protocols to setup data encryption keys.

The client database consists of nonsensitive and sensitive attributes. Sensitive attributes are transparently encrypted by the client layer as part of DML[7] query requests. The encryption is performed using a key K_{DATA} shared between the SCPU-side, TrustedDB database

[7] DML queries include insert and update queries

engine and the client. The shared key is generated using outbound authentication (Section 2.3.1). The precise key exchange protocol is listed in Figure 2.10(b).

Sensitive results generated by query execution are encrypted inside the SCPU before being sent to the client. For example, consider the query `SELECT avg(SALARY) FROM EMPLOYEES`. The result of `avg(SALARY)` is an aggregation over a sensitive attribute. Hence, the aggregation result is encrypted inside the SCPU using the shared key K_{DATA}. For authenticity, the result is also signed before transmission to the client. Figure 2.10(c) depicts the process of securing query results.

2.5.4 Choice of Encryption Algorithm

In TrustedDB, different avenues for data encryption are available. In the simplest case, the entire database is encrypted at page level and SCPU-side database engine fetches pages on demand, decrypting each page before processing. Page-level encryption features the highest throughput but does not utilize host server CPU cycles. Page-level encryption is suitable for data sets with only sensitive attributes.

Tuple-level encryption can be used for key-based lookup queries. In tuple-level encryption, each tuple is encrypted using a symmetric encryption algorithm, such as AES. The encryption key K_{DATA} is shared between clients and SCPU. Tuple-level encryption is suitable for blobs and large-sized attribute values but not for small data items such as integers due to encryption latencies and ciphertext blow-up factors.

Finally, for fine-grained-attribute-level encryption, each individual attribute value within each tuple is encrypted separately. Attribute-level encryption is achieved using random keys generated by a cryptographic hash function based cipher. The cipher is initialized with the key K_{DATA} shared between clients and SCPU. The attribute-level encryption scheme in TrustedDB is based on the NMAC construction [35, 110]. Each attribute value is encrypted as follows:

$$\begin{aligned} E(tbl.attr.val) &= ctr_{attr} \mid\mid tbl.pri\_key \mid\mid idx_K \mid\mid (tbl.attr.val \oplus k) \\ k &= F(K_{DATA}[idx_K] \ \mid\mid ctr_{attr} \mid\mid tbl.pri\_key \mid\mid F(K_{DATA}[idx_K])) \end{aligned} \tag{2.5}$$

where tbl is the table name, $tbl.attr$ is the attribute to be considered, $tbl.attr.val$ is the plaintext value of the current tuple, $tbl.pri\_key$ is the primary key of the current tuple in table tbl, ctr_{attr} is a unique identifying number associated with $tbl.attr$[8], idx_K is an index in a table of K_{DATA} keys which allows multiple such keys to exist simultaneously for increased security, and $F(\cdot)$ is a cryptographic hash function, such as SHA or MD5 [35][9].

2.5.5 Key Management

So far we have considered a single data encryption key shared between the SCPU and client(s). In a multi-client scenario, it is desired to have multiple distinct client-SCPU keys

[8] For storage efficiency we don't want to use the entire *tbl.attr* value in the result.

[9] Note that the known MD5 *collision - resistance* related vulnerabilities are not a problem here where it is used as a source of randomness only.

either for access control or for increased security in case one or more clients are compromised. The extensions to handle a multi-client scenario are discussed here.

The data stored on host server disk is encrypted using a single master encryption key known only to the SCPU. Since all update and insert operations are performed by the SCPU, the master key is stored within the SCPU and is never communicated to the outside. Now, all decryptions required as part of query processing use the master key. Only when a sensitive attribute value is to be communicated to the client, the SCPU encrypts the value using the specific client-SCPU encryption key. Use of the specific client-SCPU key ensures only the authorized client access to data.

Having a unique encryption key for each client is possible since the SCPU has specialized battery backed memory dedicated for the purpose of key storage. The available space, which is 128 KB for the 4764, enables the storage of up to 8K keys assuming a 128 bit key size. For larger key space, client keys can be generated from a master encryption key using the key generation technique described in Section 2.5.4.

2.6 Query Optimization

2.6.1 Model

As per section 2.5, due to the unavailability of storage within the SCPU the entire database is stored at the server. Hence, the attribute classification into sensitive and nonsensitive introduces a logical vertical partitioning of the database between the server and the SCPU. The partitioning of data resembles a federated database rather than a stand alone DBMS. Moreover, for data confidentiality, it needs to be ensured that sensitive attributes are processed only within the SCPU. Query optimization in TrustedDB therefore accounts for both performance and security. In the following, we describe TrustedDB query optimization techniques.

2.6.2 Overview

At a high level query optimization in a database system works as follows.

1. A query plan generator constructs possibly multiple plans for a client query.
2. For each constructed plan a query cost estimator estimates the plan's execution cost.
3. The best plan, that is, one with the least cost, is then selected and passed on to a query plan interpretor for execution.

The query optimization process in TrustedDB works similarly with key differences in the query cost estimator due to the logical partitioning of data into sensitive and nonsensitive attributes. Well-known query optimizations techniques [154] applicable in a traditional DBMS with no sensitive attributes are applied to public sub-queries executed on the host server.

In the following sections we only discuss query optimizations that are unique to TrustedDB and trusted hardware based designs alike.

	Server		SCPU	
CPU	η_s	3.4 GHz	η_t	233 MHz
Memory	Υ_s	4 GB	Υ_t	32 MB

Table 2.2: System Configuration parameters.

Query-Plan-Cost Metric

The key for query optimization is estimating the costs of various logical query plans. A common metric utilized in comparing query plan costs is disk I/O [231, 238]. Use of disk I/O as a metric is justified since disk access is the most expensive operation for databases and should be minimized. In the trusted hardware model, an additional significant I/O cost is introduced, which is the server↔SCPU data transfer cost. Moreover, disk access on the server and the Server↔SCPU communication have different costs. Further, we also need to consider the disparity between the computational abilities of the server and the SCPU. To combine all cost factors, we use execution time as the metric for cost estimation. From this point onwards, any reference to the cost of a query plan refers to the plan's execution time.

We emphasize that the goal of query optimizer is not necessarily to measure query execution times with high accuracy but only to correctly compare query plans based on estimation. To clarify, assume that a query Q has two valid execution plans P_A and P_B. Then, if the real execution times of P_A and P_B are such that $\mathcal{ET}_{real}(P_A) > \mathcal{ET}_{real}(P_B)$, then it suffices for the *Query Optimizer* to estimate $\mathcal{ET}_{est}(P_A) > \mathcal{ET}_{est}(P_B)$ although the values for $\mathcal{ET}_{real}(P_i)$ and $\mathcal{ET}_{est}(P_i)$ may not be close.

2.6.3 System Catalog

A query plan is composed of multiple steps. To estimate the cost of an entire plan, it is essential to estimate the cost of individual steps and aggregate them. In order to estimate individual steps' costs, the query cost estimator needs access to some key information. For instance, the availability of an index or the knowledge of possible distinct values of an attribute. Information needed by the query cost estimator is collected and stored in the system catalog. Most available DBMS today maintain some form of periodically updated system catalog. Figures 2.2-2.4 and tables 2.3-2.6 give a partial view of the system catalog maintained by TrustedDB. Later, in section 2.6.5 we will see how the system catalog information is used in estimating plan execution times. System Catalog content is classified in to four categories. We discuss each category in the following:

(a) System Configuration (Figure 2.2)

System configuration includes compute capacities of the system hardware. Configuration information is unlikely to change frequently and is configured during system setup.

Parameter		Value	Description
Disk Read	ϕ_s	0.02 ms	Average time to read 32 KB blk from server disk
Server↔SCPU	λ	5.26 ms	Average time to transfer a 32 KB blk between server and SCPU
Cycles / aggregation	δ_g	3	Number of cpu cycles per aggregate operation (e.g. group by)
Cycles / addition	δ_a	1	Number of cpu cycles per addition
Cycles / comparison	δ_c	1	Number of cpu cycles per comparison between two values
Crypto	ϵ_a	0.012 μs	Time to encrypt/decrypt a single (32 byte) attribute in SCPU

Table 2.3: Benchmarked parameters.

Parameter		Value
DB Page Size	ρ	32KB
Server Cache Size	μ_s	32768
SCPU Cache Size	μ_t	1024
B^+-Tree Order	θ	100

Table 2.4: Database Configuration parameters.

(b) Benchmarked Parameters (Table 2.3)

As part of query execution many basic operations are performed which add to the overall execution time. Benchmarks are needed to determine the average execution times for the basic operations. Unless system configuration is changed, benchmarked parameters remain unchanged.

(c) Database Configuration (Figure 2.4)

Database configuration parameters are directly set on the host server database to improve performance. In TrustedDB, the server DBMS is an off the shelf industrial quality database system (Section 2.8), which provides a large range of configuration parameters. With the TrustedDB design the host DBMS can be configured independently.

(d) Data Statistics (Table 2.5)

TrustedDB performs periodic data scans to collect data statistics. The statistics on public attributes are collected server side. However, private attributes are scanned via the SCPU, decrypted, and then analyzed. Since the collection process involves scan of the database, it is a time consuming task. As a result, data statistics collection needs to be scheduled during low workloads, for example, nightly or on weekends.

lineitem			
Attribute	**Values**	**Max**	**Size**
l_shipdate	$\upsilon_{l_{sd}} = 2526$	$\vartheta_{l_{sd}}$	$\kappa_{l_{sd}} = 4$
l_shipmode	$\upsilon_{l_{sm}} = 7$		$\kappa_{l_{sm}} = 10$
l_linestatus	$\upsilon_{l_{ls}} = 2$		$\kappa_{l_{ls}} = 16$
l_quantity			$\kappa_{l_{qt}} = 4$
l_discount	$\upsilon_{l_{dc}} = 11$		$\kappa_{l_{dc}} = 16$
l_orderkey			$\kappa_{l_{ok}} = 4$
l_linenumber			$\kappa_{l_{ln}} = 4$
Indexes			
Table	**Attribute(s)**	**Type**	**Organization**
lineitem	l_orderkey, l_linenumber	B^+-Tree	clustered
lineitem	l_shipdate	B^+-Tree	non-clustered

Table 2.5: Collected data statistics.

	lineitem		orders		part	
Number of tuples	φ_l	6 M	φ_o	1.5 M	φ_p	200 K
Tuple size	τ_l	120	τ_o	100	τ_p	160
Tuples per page ($\frac{\rho}{\tau}$)	ω_l	273	ω_o	328	ω_p	203

Table 2.6: Collected relation level statistics for sample TPC-H data set of Figure 2.6.

2.6.4 Estimating Cost of Basic Query Operations

In this section, we present cost estimation for certain basic query plan steps. The basic steps are reused in multiple plans and hence we group their cost analysis here.

(i) Index-based lookup

Consider the selection query $\mathcal{Q} = \sigma_{l\_shipdate=10/01/1998}$. As per the System Catalog (Table 2.5), there is a B^+-Tree index available on the attribute *l_shipdate*. Hence, the expected execution time to locate the first leaf page containing the result tuples will be

$$\mathcal{ET}(\mathcal{Q}) = \log_\theta \varphi_l \cdot \phi_s \tag{2.6}$$

Here, $\log_\theta \varphi_l$ is number of index pages read and ϕ_s is the time to read a single page from disk on server.

(ii) Selection

Estimating the execution time of selection queries requires estimation of the number of tuples that would comprise the query result. To estimate the number of tuples in query results, we use the following concepts from [154].

Definition 1. Values*: The* Values *of an attribute A, $Values(A)$ or υ_A is the number of distinct values of A.*

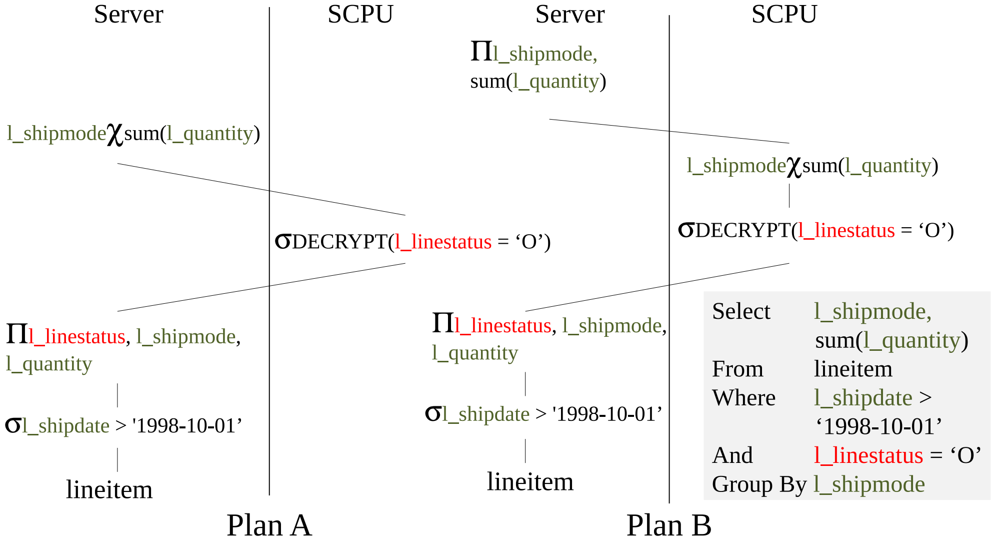

Server
SCPU
Server
SCPU
Πl_shipmode, sum(l_quantity)
l_shipmodeχsum(l_quantity)
l_shipmodeχsum(l_quantity)
σDECRYPT(l_linestatus = 'O')
σDECRYPT(l_linestatus = 'O')
Πl_linestatus, l_shipmode, l_quantity
Πl_linestatus, l_shipmode, l_quantity
σl_shipdate > '1998-10-01'
σl_shipdate > '1998-10-01'
lineitem
lineitem
Select l_shipmode, sum(l_quantity)
From lineitem
Where l_shipdate > '1998-10-01'
And l_linestatus = 'O'
Group By l_shipmode
Plan A
Plan B

Figure 2.11: Query plans for Case I: Group By on public attribute.

Definition 2. Reduction Factor*: The* Reduction Factor *of a condition C, $Reduction(C)$ or Θ_C is the reduction in size of the relation caused by the execution of a selection clause with that condition.*

For example,

$$Reduction(l\_shipdate = 10/01/1998) = \frac{1}{Values(l\_shipdate)} \quad OR \quad \Theta_{l\_shipdate=10/01/1998} = \frac{1}{\upsilon_{l_sd}} \tag{2.7}$$

Further,

$$\Theta_{l\_shipdate>10/01/1998} = \frac{\vartheta_{l_{sd}} -' 10/01/1998'}{\upsilon_{l_sd}} \tag{2.8}$$

Note that above definitions assume a uniform distribution of attribute values, that is, each distinct attribute value is equally likely to occur in a relation.

Now, consider the selection query $\sigma_{l\_shipdate>10/01/1998}$. For execution of the selection query, the index on *l_shipdate* is used to locate the first leaf page containing the result. Then, subsequent leaf pages are scanned to gather all tuples comprising the result. Hence the estimated execution time is $\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$. Here the term $\frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l}$ estimates the number of leaf pages containing all tuples that satisfy the query.

(iii) Server↔SCPU Data Transfer

The intermediate results from query plan execution are often transferred between the server and the SCPU. The server-SCPU data transfer involves exchange of fixed sized pages in a

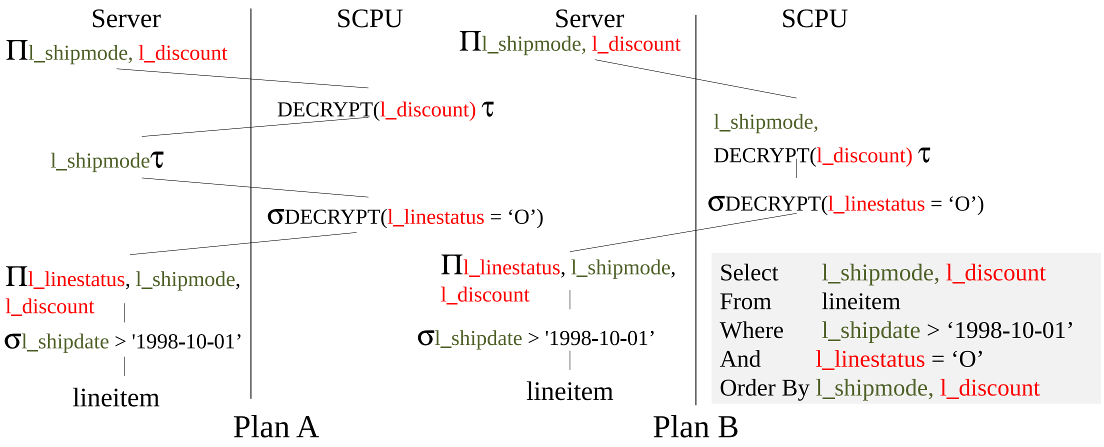

Server
SCPU
Πl_shipmode, l_discount
DECRYPT(l_discount) τ
l_shipmodeτ
σDECRYPT(l_linestatus = 'O')
Πl_linestatus, l_shipmode, l_discount
σl_shipdate > '1998-10-01'
lineitem
Plan A
Server
SCPU
Πl_shipmode, l_discount
l_shipmode, DECRYPT(l_discount) τ
σDECRYPT(l_linestatus = 'O')
Πl_linestatus, l_shipmode, l_discount
σl_shipdate > '1998-10-01'
lineitem
Plan B
Select l_shipmode, l_discount
From lineitem
Where l_shipdate > '1998-10-01'
And l_linestatus = 'O'
Order By l_shipmode, l_discount

Figure 2.12: Query plans for Case II: Order By on public and private attributes.

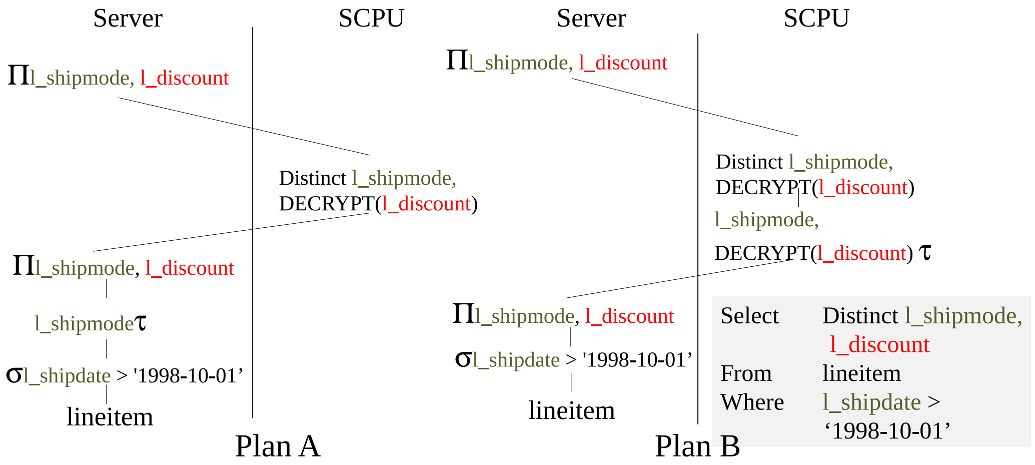

Server
SCPU
Πl_shipmode, l_discount
Distinct l_shipmode, DECRYPT(l_discount)
Πl_shipmode, l_discount
l_shipmodeτ
σl_shipdate > '1998-10-01'
lineitem
Plan A
Server
SCPU
Πl_shipmode, l_discount
Distinct l_shipmode, DECRYPT(l_discount)
l_shipmode, DECRYPT(l_discount) τ
Πl_shipmode, l_discount
σl_shipdate > '1998-10-01'
lineitem
Plan B
Select Distinct l_shipmode, l_discount
From lineitem
Where l_shipdate > '1998-10-01'

Figure 2.13: Query plans for Case III: Distinct clause on public and private attribute.

synchronous fashion. If the data to be transferred is B bytes, then the total transfer time is $\lceil \frac{B}{\rho*1024} \rceil \cdot \lambda$. Here, ρ is the page size and hence $\lceil \frac{B}{\rho*1024} \rceil$ gives the number of pages needed to transfer B bytes. λ is the time required to transfer a single page of size ρ. Refer to Table 2.3 and Figure 2.4 for values of ρ and λ.

Suppose that we need to transfer the results of a query $Q = \Pi_{sum(l\_quantity)}(\sigma_{l\_shipdate>10/01/1998 \text{ } and \text{ } l\_linestatus='O'})$. Then, we can estimate the intermediate query result size by multiplying the number of tuples in the query result with the total size of the projection operation. The size of the projection is simply the sum of the sizes of the individual attributes $l\_linestatus$ and $l\_quantity$. Hence, the total data transfer time for query Q is estimated as $\frac{(\kappa_{l_{ls}}+\kappa_{l_{qt}})\cdot\Theta_{l_{sd}}\cdot\varphi_l}{\rho*1024}\cdot\lambda$.

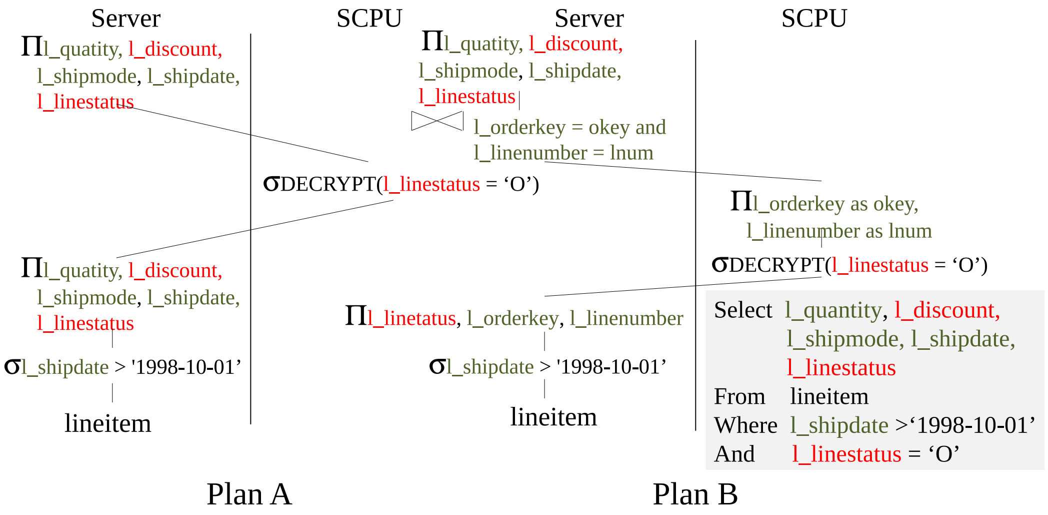

Server
SCPU
Server
SCPU
Πl_quatity, l_discount, l_shipmode, l_shipdate, l_linestatus
σDECRYPT(l_linestatus = 'O')
Πl_quatity, l_discount, l_shipmode, l_shipdate, l_linestatus
σl_shipdate > '1998-10-01'
lineitem
Πl_quatity, l_discount, l_shipmode, l_shipdate, l_linestatus
⋈ l_orderkey = okey and l_linenumber = lnum
Πl_orderkey as okey, l_linenumber as lnum
σDECRYPT(l_linestatus = 'O')
Πl_linetatus, l_orderkey, l_linenumber
σl_shipdate > '1998-10-01'
lineitem
Select l_quantity, l_discount, l_shipmode, l_shipdate, l_linestatus
From lineitem
Where l_shipdate >'1998-10-01'
And l_linestatus = 'O'
Plan A
Plan B

Figure 2.14: Query plans for Case IV: Projections.

(iv) External Sorting

External sorting is used when the relation(s) to be sorted cannot fit in memory. External sorting has been studied extensively [154] and the I/O cost for an external merge sort is given as $2 \cdot F \cdot log_{M-1}F$, where F is the total number of relation pages and M is the number of pages that can be stored in memory. Also, $M << F$. Using the well-known cost of external sorting, we can estimate the execution time for sorting a relation r on the server as

$$2 \cdot \frac{\varphi_r \cdot \mathcal{T}_r}{1024 \cdot \rho} \cdot \left(\log_{\frac{\Upsilon_s \cdot 1024}{\rho} - 1} \frac{\varphi_r \cdot \mathcal{T}_r}{1024 \cdot \rho} \right) \cdot \phi_s \tag{2.9}$$

and the time for the same sort from within the SCPU as

$$2 \cdot \frac{\varphi_r \cdot \mathcal{T}_r}{1024 \cdot \rho} \cdot \left(\log_{\frac{\Upsilon_t \cdot 1024}{\rho} - 1} \frac{\varphi_r \cdot \mathcal{T}_r}{1024 \cdot \rho} \right) \cdot (\phi_s + \lambda) \tag{2.10}$$

Here, φ_r is the total number of tuples in r and $\mathcal{T}_r$ is the size of an individual tuple.

2.6.5 Estimating Query Plan Costs

Overall Approach

The TrustedDB query optimizer selects optimization techniques on a case-by-case basis. To illustrate query optimization in TrustedDB, we take the following approach for each case.

- First, we present two alternative plans for the case.
- Next, we estimate the execution times ($\mathcal{ET}$) for each of the two plans.
- We analyze the estimations of step (2) for selection of the best plan.

Plan	Step	Execution Time ($\mathcal{ET}$)	$\mathcal{ET}$ (s)
I.A	1. $\sigma_{l\_shipdate > `1998-10-01'}$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$	0.57
	2. Server $\rightarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$	
	3. $\sigma_{DECRYPT(l\_linestatus = `O')}$	$\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$	
	4. Server $\leftarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$	
	5. ${}_{l\_shipmode} \mathcal{X}_{sum(l\_quantity)}$	$(\Theta_{l_{sd}} \cdot \varphi_l \cdot \delta_g + \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l \cdot \delta_a) \cdot \frac{1000}{\eta_s}$	
I.B	1. $\sigma_{l\_shipdate > `1998-10-01'}$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$	0.78
	2. Server $\rightarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$	
	3. $\sigma_{DECRYPT(l\_linestatus = `O')}$	$\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$	
	4. ${}_{l\_shipmode} \mathcal{X}_{sum(l\_quantity)}$	$2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda,\ F = \left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho * 1024} \right\rceil$	
	5. Server $\leftarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}}) \cdot \upsilon_{l_{sm}}}{\rho \cdot 1024} \right\rceil \cdot \lambda$	
II.A	1. $\sigma_{l\_shipdate > `1998-10-01'}$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$	0.68
	2. Server $\rightarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$	
	3. $\sigma_{DECRYPT(l\_linestatus) = `O'}$	$\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$	
	4. Server $\leftarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$	
	5. ${}_{l\_shipmode} \mathcal{T}$	$\frac{\Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l \cdot 1000}{\eta_s} \cdot \delta_c$	
	6. ${}_{DECRYPT(l\_discount)} \mathcal{T}$	$2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda,\ F = \left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil$	
II.B	1. $\sigma_{l\_shipdate > `1998-10-01'}$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$	1.29
	2. Server $\rightarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$	
	3. $\sigma_{DECRYPT(l\_linestatus) = `O'}$	$\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$	
	4. Server $\leftarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$	
	5. ${}_{l\_shipmode, DECRYPT(l\_discount)} \mathcal{T}$	$2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda,\ F = \left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil$	

Table 2.7: Cost computations for query plans shown in Figures 2.11 and 2.12 corresponding to cases I and II.

Note that the query optimizer may generate multiple plans for each client query. For brevity, we limit ourselves to discussion of only two plans for each case. In Section 2.8 we experimentally verify TrustedDB's query optimization techniques.

Case I: Group-By on Public Attribute

Figure 2.11 shows two alternative plans for a group-by operation on public attribute *l_shipmode*. The difference between the plans A and B is whether the grouping is performed by the server or the SCPU. If the grouping is performed by the server, then the cheap server cycles are utilized. However, if grouping is done within the SCPU, the SCPU→Server data transfer is reduced. The reduction in data transferred depends on the number of distinct values of attribute *l_shipmode*.

Table 2.7 shows the computation of execution times of plans A and B along with the estimation. Given the parameters and system catalog data from Figures 2.2-2.4 and Tables 2.3-2.6, it is more efficient to perform the group-by operation server side, that is, plan A has lower estimated execution time. The lower execution time for plan A results due to the high selectivity of *l_shipdate* attribute, which reduces the SCPU-server data transfer cost. If *l_shipdate* had low selectivity, then aggregation within the SCPU would be more efficient.

Plan	Step	Execution Time ($\mathcal{ET}$)	$\mathcal{ET}$ (s)
III.A	1. $\sigma_{l\_shipdate > \text{‘}1998-10-01'}$ 1. $l\_shipmode^{\mathcal{T}}$ 3. $\text{Distinct}_{l\_shipmode, DECRYPT(l\_discount)}$ 4. Server ← SCPU transfer	$\log_{\theta} \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$ $2 \cdot \frac{\varphi_l \cdot \Theta_{l_{sd}} \cdot \tau_l}{1024 \cdot \rho} \cdot \left(\log_{\frac{\Upsilon_s \cdot 1024}{\rho} - 1} \frac{\varphi_l \cdot \Theta_{l_{sd}} \cdot \tau_l}{1024 \cdot \rho} \right) \cdot \phi_s$ $\upsilon_{l_{sm}} \cdot 2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda,\ F = \left\lceil \frac{\kappa_{l_{dc}} \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil$ $\left\lceil \frac{(\kappa_{l_{sm}} + \kappa_{l_{dc}}) \cdot \upsilon_{l_{sm}} \cdot \upsilon_{l_{dc}}}{\rho \cdot 1024} \right\rceil \cdot \lambda$	0.23
III.B	1. $l\_shipmode, DECRYPT(l\_discount)^{\mathcal{T}}$ 2. $\text{Distinct}_{l\_shipmode, DECRYPT(l\_discount)}$ 3. Server ← SCPU transfer	$2 \cdot \frac{\varphi_l \cdot \tau_l}{1024 \cdot \rho} \cdot \left(\log_{\frac{\Upsilon_t \cdot 1024}{\rho} - 1} \frac{\varphi_l \cdot \tau_l}{1024 \cdot \rho} \right) \cdot (\phi_s + \lambda)$ $2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda,\ F = \left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil$ $\left\lceil \frac{(\kappa_{l_{sm}} + \kappa_{l_{dc}}) \cdot \upsilon_{l_{sm}} \cdot \upsilon_{l_{dc}}}{\rho \cdot 1024} \right\rceil \cdot \lambda$	0.20
IV.A	1. $\sigma_{l\_shipdate > \text{‘}1998-10-01'}$ 2. Server → SCPU transfer 3. $\sigma_{DECRYPT(l_{linestatus} = \text{‘}O')}$ 4. Server ← SCPU transfer	$\log_{\theta} \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$ $\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{dc}} + \kappa_{l_{sm}} + \kappa_{l_{sd}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$ $\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$ $\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{dc}} + \kappa_{l_{sm}} + \kappa_{l_{sd}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$	0.81
IV.B	1. $\sigma_{l\_shipdate > \text{‘}1998-10-01'}$ 2. Server → SCPU transfer 3. $\sigma_{DECRYPT(l\_linestatus = \text{‘}O')}$ 4. Server ← SCPU transfer 5. $\bowtie_{l\_orderkey = okey, l\_linenumber = lnum}$	$\log_{\theta} \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$ $\left\lceil \frac{(\kappa_{l_{ls}} + \kappa_{l_{ok}} + \kappa_{l_{ln}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$ $\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$ $\left\lceil \frac{(\kappa_{l_{ok}} + \kappa_{l_{ln}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$ $\left((2 \cdot F \cdot log_{\mu_t - 1} F) + \frac{\Theta_{l_{ls}} \cdot \varphi_l}{\omega_l} \right) \cdot \phi_s,\ F =$ $\left\lceil \frac{(\kappa_{l_{ok}} + \kappa_{l_{ln}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil$	0.32
V.A	1. $o\_totalprice^{\mathcal{T}}$ 2. $p\_retailprice^{\mathcal{T}}$ 3.$\bowtie_{o\_totalprice = p\_retailprice}$	$2 \cdot \frac{\varphi_o \cdot \tau_o}{1024 \cdot \rho} \cdot \left(\log_{\frac{\Upsilon_s \cdot 1024}{\rho} - 1} \frac{\varphi_o \cdot \tau_o}{1024 \cdot \rho} \right) \cdot \phi_s$ $2 \cdot \frac{\varphi_p \cdot \tau_p}{1024 \cdot \rho} \cdot \left(\log_{\frac{\Upsilon_t \cdot 1024}{\rho} - 1} \frac{\varphi_p \cdot \tau_p}{1024 \cdot \rho} \right) \cdot (\phi_s + \lambda) + \varphi_p \cdot \epsilon_a$ $\frac{\varphi_o}{\omega_o} \cdot \frac{\varphi_p}{\omega_p} \cdot (\phi_s + \lambda) + \varphi_p \cdot \epsilon_a$	39.8
V.B	1. $\bowtie_{o\_totalprice = p\_retailprice}$	$\frac{\varphi_p \cdot \tau_p}{1024 \cdot \rho} + \frac{\varphi_o \cdot \tau_o}{1024 \cdot \rho} \cdot \left\lceil \frac{\frac{\varphi_p \cdot \tau_p}{1024 \cdot \rho}}{\frac{\Upsilon_t \cdot 1024}{\rho} - 1} \right\rceil$	29.3

Table 2.8: Cost computations for query plans shown in Figures 2.12(a) and 2.12(b) corresponding to cases III and IV.

Case II: Order-By on Public and Private Attributes

If an order-by clause has a nonsensitive attribute followed by a sensitive attribute, the server can first order the intermediate results on the nonsensitive attribute leaving the sensitive attribute ordering to the SCPU (Figure 2.12 - plan A). As an alternative, the SCPU can process the entire order-by clause as shown in Figure 2.12, plan B.

Under the specific data statistics, plan A is preferred. In plan A, the SCPU orders a smaller intermediate result set making the the sort operation more efficient. Since the SCPU employs external sorting, any reduction in the size of intermediate results directly lowers the Server↔SCPU transfer cost.

Case III: Distinct clause on Public and Private Attributes

A distinct clause is processed by first sorting the relation on the sort attribute and then removing the duplicate values. Figure 2.13 depicts two plans for evaluating a distinct operation. In plan A (Figure 2.13), the server first sorts the intermediate results on the public attribute *l_shipmode*. Then, the SCPU processes the distinct clause. In plan B, the SCPU sorts and processes the distinct operation.

As seen from Table 2.8, the optimizer prefers plan A. The number of distinct values of *l_shipmode* play a critical role in plan selection here. In the dataset, *l_shipmode* has high selectivity, which means that after server-side sorting, the SCPU operates on a small data set. Low selectivity of *l_shipmode* instead would favor plan B since the advantage of server-side sorting will be reduced.

Case IV: Projections

Figure 2.14 shows two plans for evaluating a projection operation. In plan A, all projected attributes are passed to the SCPU. After SCPU-side processing all attributes are transferred back to the server. The server then performs the final projection.

In plan B, only the attributes needed for SCPU-side processing are passed into the SCPU. As a result, in plan B the server needs to perform additional lookups to locate the projected attributes for each tuple in the result set. To optimize the final lookup join, the server first sorts the intermediate results received from the SCPU on the tuple primary key. The sort on primary key greatly reduces the number of disk operations, thereby making plan B more efficient.

Case V: Public-Private Join

A join between a sensitive and a nonsensitive attribute can be processed via two plans. In plan A, the relation with the nonsensitive join attribute is sorted server-side. The relation with the sensitive join attribute is sorted by the SCPU. Then, the SCPU performs a sort-merge join on the sorted relations. In plan B, the SCPU performs the entire join using a block-nested mechanism. For efficiency, in plan B, the SCPU makes use of all available memory within.

The steps involved in both plans are listed in Table 2.8 along with the estimated plan costs. Plan B is preferred since the SCPU sort operation in plan A is expensive enough to not augment the advantage of server-side sorting on nonsensitive attribute. However, if the relation to be sorted had a clustered index on the sort attribute, then the server-side sort in plan A would be eliminated, thereby favoring plan A.

2.7 Attribute Exposure

For efficiency, TrustedDB relies on attribute classification into sensitive and non-sensitive attributes. Sensitive attributes are stored encrypted. Non-sensitive attributes are stored in clear. To ensure data confidentiality, sensitive attributes are decrypted only within the trusted hardware.

However, correlations between non-sensitive and sensitive attributes can yet violate data confidentiality. For example, consider a census database that stores *name*, *date of birth*, and *zip* of individuals. Also, suppose that attribute *name* is considered sensitive while *date* *of birth* and *zip* are both considered non-sensitive. Samarati et al. [227] have shown that the *date of birth* and *zip* combination can be used to infer an individual's identity. Hence, storing *date of birth* and *zip* as non-sensitive attributes can reveal individual names, thereby violating data confidentiality that TrustedDB aims to protect.

Information leakage due to correlation between non-sensitive and sensitive attributes can be prevented by a careful choice of sensitive attributes. For example, in a census database *name*, *date of birth*, and *zip* can all be classified as sensitive. We note that increasing the number of sensitive attributes lowers efficiency, since the cheap untrusted server cycles can only be utilized to process non-sensitive attributes. The following interesting question arises from this discussion. What is the optimal attribute classification that prevents information leakage from non-sensitive to sensitive attributes and permits the maximum possible number of non-sensitive attributes for efficiency. This question of optimal attribute classification has been answered by Ciriani et al. [69].

Ciriani et al. [69] refer to a set of correlated attributes as a $confidentiality\ constraint$. The requirement is that attributes with a constraint must never be jointly visible. Joint visibility can be prevented by encrypting one or more attributes in a constraint. For example, the sets {*name*} and {*date of birth*, *zip*} are constraints in a census database. Given a set of confidentiality constraints to be satisfied, Ciriani et al. address the problem of minimizing the number of sensitive attributes. They refer to this problem as $minimal\ fragmentation$. Ciriani et al. go on to show that minimal fragmentation is a NP-hard problem. Instead, they propose a heuristic approach to minimize fragmentation.

We suggest adoption of the approach by Ciriani et al. to select the set of sensitive attributes in TrustedDB. Once, attribute classification is done, TrustedDB provides an efficient, low-cost solution for query processing as compared to cryptography-based solutions.

2.8 Experiments

2.8.1 Setup

The SCPU of choice is the IBM 4764-001 PCI-X with the 3.30.05 release toolkit featuring 32MB of RAM and a PowerPC 405GPr at 233 MHz. The SCPU sits on the PCI-X bus of an Intel Xeon 3.4 GHz, 4GB RAM Linux box with kernel version 2.6.18. The server DBMS is a standard MySQL 14.12 Distrib 5.0.45 engine. The SCPU DBMS is a heavily modified SQLite custom port to the PowerPC. The entire TrustedDB stack is written in C.

2.8.2 TPC-H Query Load

To evaluate the runtime of generalized queries, we chose several queries of varying degrees of difficulty and privacy from the TPC-H benchmark [12] The TPC-H scale factor is 1, that is, the database size is $1GB$.

Figure 2.15 shows the TrustedDB query execution times as compared to a simple unencrypted MySQL setup. Figure 2.16(a) depicts the breakdown of times spent in execution

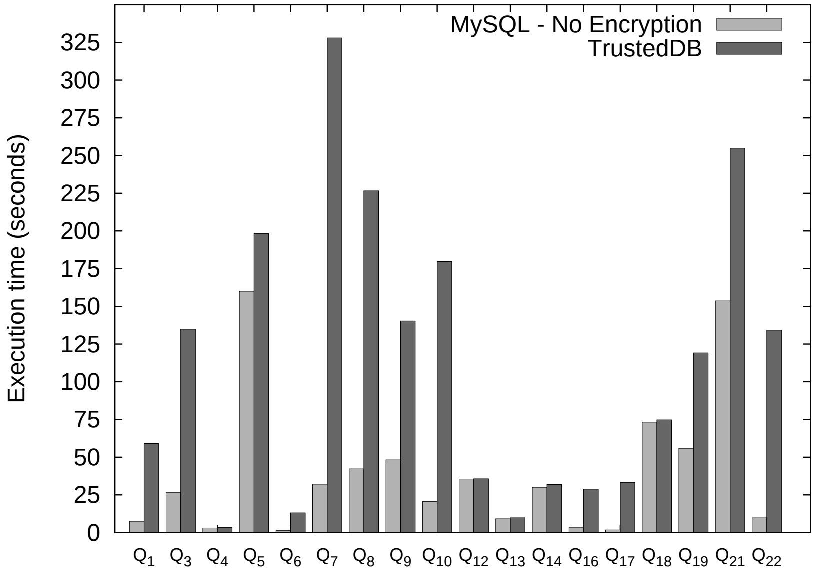


Figure 2.15: TPC-H query execution times.

of the public and private sub-queries. The execution times of private queries include the time required for encryption and decryption operations inside the SCPU. The public queries executed on the host server also include the server-SCPU data transfer time.

As seen from Figure 2.15, when compared with a completely unsecured baseline scenario, data confidentiality in TrustedDB does not come cheap. The query execution times in TrustedDB are higher by factors between 1.03 and 10. However, recall from Section 2.4 that the actual query processing costs are orders of magnitude lower than any cryptography-based solution.

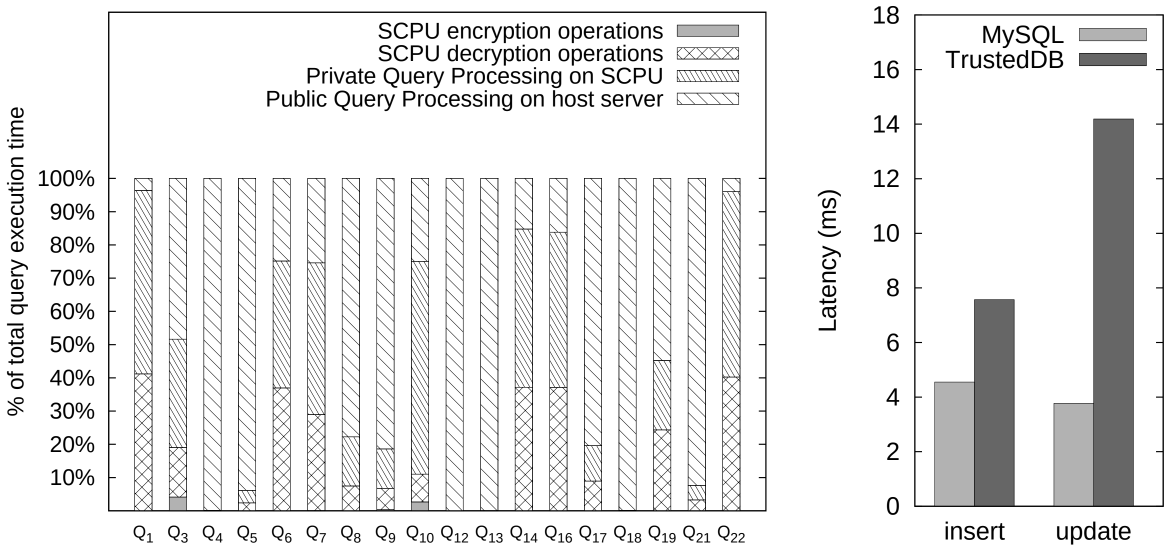
Figure 2.16(b) shows the latencies for insert and update queries. The reported times are for a random insert or update of a single tuple in the lineitems relation averaged over ten runs.

2.8.3 Query Optimization

In Section 2.6, we presented different query plans, analyzed query plans execution and showed how the optimizer computed query plan execution times. Tables 2.7 and 2.8 summarized the theoretical costs and estimated execution times. To verify whether the plan selected in each case is indeed the best plan, we executed each of the plans on the TPC-H dataset and measured execution times. Figure 2.17(a) lists the results.

We find that in each of the cases I-IV, the following holds: If $\mathcal{ET}_{est}(P_A)$ >$\mathcal{ET}_{est}(P_B)$ in table 2.7 or 2.8 then $\mathcal{ET}_{real}(P_A)$ >$\mathcal{ET}_{real}(P_B)$ in figure 2.17(a).

For more detailed evaluation, we compare the estimated and measured times for varying selectivity of the public attribute *l_shipdate* in case I. Note that the selectivity directly influences the amount of server↔SCPU data transfer and thus the overall processing costs.



(a) Query time profiles.

(b) DML ops.

Figure 2.16: Query time profiles and latencies for DML operations. $Q_i = i^{th}$ query from TPC-H [12].

As seen in Figure 2.17(b), the optimizer correctly estimates which plan would have lower execution time for most of the cases.

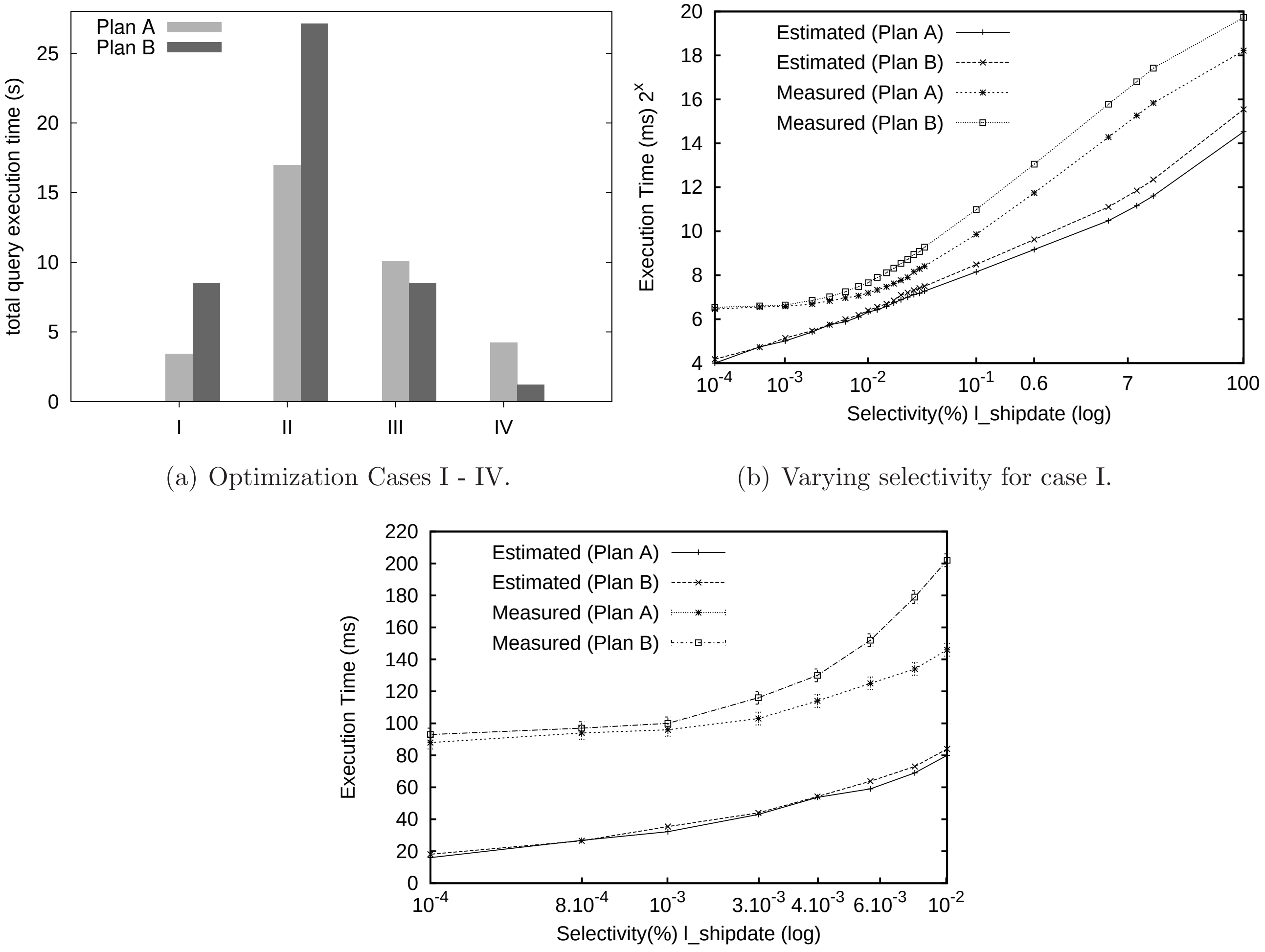
Figure 2.17(c) shows the results for very low selectivity of *l_shipdate*. At low selectivity the accuracy of estimation lowers. The low accuracy is due to experimental variance and data distribution. In the experiments, measured times vary by $\pm$3.5ms between runs. Thus, when the estimated times for two plans differ by <3.5ms they are practically equivalent. Also, the optimizer assumes a uniform distribution of attribute values. For the TPC-H data, uniform distribution does not hold, especially at low selectivity. The accuracy of estimation for in the case of low selectivity can be increased by simply populating the system catalog with more accurate information.

2.9 Related Work

2.9.1 Queries on Encrypted Data

Hacigumus et al. [125] propose division of data into secret partitions and re-writing of range queries over the original data in terms of the resulting partition identifiers. This balances a trade-off between client and server-side processing, as a function of the data segment size. In [134] the authors explore optimal bucket sizes for range queries.

[89] proposes using tuple-level encryption and indexes on the encrypted tuples to support equality predicates. The main contribution here is the analysis of attribute exposure caused by query processing leading to two insights. (a) the attribute exposure increases with the



(a) Optimization Cases I - IV.

(b) Varying selectivity for case I.

(c) Low selectivity for case I.

Figure 2.17: Measured execution times for optimization cases I-IV from section 2.6 and with varied selectivity for Case I.

number of attributes used in an index, and (b) the exposure decreases with the increase in database size. Range queries are processed by encrypting individual $B^+ - Tree$ nodes and having the client, in each query processing step, retrieve a desired encrypted $B^+ - Tree$ node from the server, decrypt and process it. However, this leads to minimal utilization of server resources thereby undermining the benefits of outsourcing. Moreover, transfer of entire $B^+ - Tree$ nodes to the client results in significant network costs.

[256] employs *Order Preserving* encryption for querying encrypted xml databases. In addition, a technique referred to as splitting and scaling is used to differ the frequency distribution of encrypted data from that of the plain-text data. Here, each plain-text value is encrypted using multiple distinct keys. Then, corresponding values are replicated to ensure that all encrypted values occur with the same frequency thereby thwarting any frequency-based attacks.

[260] uses a salted version of IDA scheme to split encrypted tuple data amongst multiple servers. In addition, a secure $B^+ - Tree$ is built on the key attribute. The client utilizes the $B^+ - Tree$ index to determine the IDA matrix columns that need to be accessed for data retrieval. To speed up client-side processing and reduce network overheads it is suggested to cache parts of the $B^+ - Tree$ index client-side.

Vertical partitioning of relations amongst multiple un-trusted servers is employed in [98]. Here, the privacy goal is to prevent access of a subset of attributes by any single server. E.g., {Name, Address} can be a privacy sensitive access-pair and query processing needs to ensure that they are not jointly visible to any single server. The client query is split into multiple queries wherein each sub-query fetches the relevant data from a server and the client combines results from multiple servers. [22] also uses vertical partitioning in a similar manner and for the same privacy goal, but differs in partitioning and optimization algorithms. TrustedDB is equivalent to both [22, 98] when the size of the privacy subset is one and hence a single server suffices. In this case each attribute column needs encryption to ensure privacy [70]. Hence [22,98] can utilize TrustedDB to optimize for querying encrypted columns since otherwise they rely on client-side decryption and processing.

[70] introduces the concept of logical fragments to achieve the same partitioning effect as in [22, 98] on a single server. A fragment here is simply a relation wherein attributes not desired to be visible in that fragment are encrypted. TrustedDB (and other solutions) are in effect concrete mechanisms to efficiently query any individual fragment from [70]. [70] on the other hand can be used to determine the set of attributes that should be encrypted in TrustedDB.

Ge et al. [101] propose an encryption scheme in a trusted-server model to ensure privacy of data residing on disk. The FCE scheme designed here is equivalently secure as a block cipher, however, with increased efficiency. [211], like [101] only ensures privacy of data residing on disk. In order to increase query functionality a layered encryption scheme is used and then dynamically adjusted (by revealing key to the server) according to client queries. TrustedDB on the other hand operates in an un-trusted server model, where sensitive data is protected, both on disk and during processing.

Data that is encrypted on disk but processed in clear (in server memory) as in [101,211] compromises privacy during the processing interval. In [52] the disclosure risks in such solutions are analyzed. [52] also proposes a new query optimizer that takes into account both performance and disclosure risk for sensitive data. Individual data pages are encrypted by secret keys that are managed by a trusted hardware module. The decryption of the data pages and subsequent processing is done in server memory. Hence the goal is to minimize the lifetime of sensitive data and keys in server memory after decryption. In TrustedDB there is no such disclosure risk since decryptions are performed only within the SCPU.

Aggregation queries over relational databases is provided in [123] by making use of homomorphic encryption based on Privacy Homomorphism [223]. The authors in [91] have suggested that this scheme is vulnerable to a cipher text only attack. Instead [91] proposes an alternative scheme to perform aggregation queries based on bucketization [125]. Here the data owner precomputes aggregate values such as SUM and COUNT for partitions and stores them encrypted at the server. Although this makes processing of certain queries faster it does not significantly reduce client side processing.

Ge et al. [249] discuss executing aggregation queries with confidentiality on an untrusted server. Due to the use of extremely expensive homomorphisms [198, 199] this scheme leads to impractically large costs by comparison, for any reasonable security parameter choices. This is discussed in more detail in section 2.4.

Above solutions are specialized for certain types of query operations on encrypted data. [89] for equality predicates, [125, 256, 260] for range predicates and [123, 249] for aggregation. In TrustedDB, all decryptions are performed within the secure confinements of the SCPU, thereby processing is done on plain-text data. This removes any limitation on the nature of predicates that can now be employed on encrypted attributes including arbitrary user defined functions. We note that certain solutions designed for a very specific set of predicates can be more efficient albeit at the loss of functionality.

2.9.2 Encrypted Storage

Encryption is one of the most common techniques used to protect the confidentiality of stored data. Several existing systems encrypt data before storing it on potentially vulnerable storage devices or network nodes. Blaze's CFS [37], TCFS [56], EFS [175], StegFS [170], and NCryptfs [265] are file systems that encrypt data before writing to stable storage. NCryptfs is implemented as a layered file system [133] and is capable of being used even over network file systems such as NFS. SFS [120] and BestCrypt [143] are device driver level encryption systems. Encryption file systems are designed to protect the data at rest, yet only partially solve the outsourcing problem. They do not allow for complex retrieval queries or client access privacy.

2.9.3 Keyword Searches on Encrypted Data

Song et al. [241] propose a scheme for performing simple keyword search on encrypted data in a scenario where a mobile, bandwidth-restricted user wishes to store data on an untrusted server. The scheme requires the user to split the data into fixed-size words and perform encryption and other transformations. Drawbacks of this scheme include fixing the size of words, the complexities of encryption and search, the inability of this approach to support access pattern privacy, or retrieval correctness. Eu-Jin Goh [107] proposes to associate indexes with documents stored on a server. A document's index is a Bloom filter [39] containing a codeword for each unique word in the document. Chang and itzenmacher [58] propose a similar approach, where the index associated with documents consists of a string of bits of length equal to the total number of words used (dictionary size). Boneh et al. [43] proposed an alternative for senders to encrypt e-mails with recipients' public keys, and store this email on untrusted mail servers. They present two search protocols: (1) a non-interactive search-able encryption scheme based on a variant of the Diffie-Hellman problem hat uses bilinear maps on elliptic curves; and (2) a protocol using only trapdoor permutations, requiring a large number of public-private key pairs. Both protocols are computationally expensive. Golle et al. [112] extend the above idea to conjunctive keyword searches on encrypted data. They propose two solutions. (1) The server stores capabilities for conjunctive queries, with sizes linear in the total number of documents. They claim that a majority of the capabilities can be transferred offline to the server, under the assumption that the client knows beforehand its future conjunctive queries. (2) Doubling the size of the data stored by the server, which reduces the communication overheads between clients and servers significantly. The scheme requires users to specify the exact positions where the search matches have to occur, and hence is impractical. Brinkman et al. [46] deploy secret splitting of polynomial expressions to search in encrypted XML.

2.9.4 Trusted Hardware

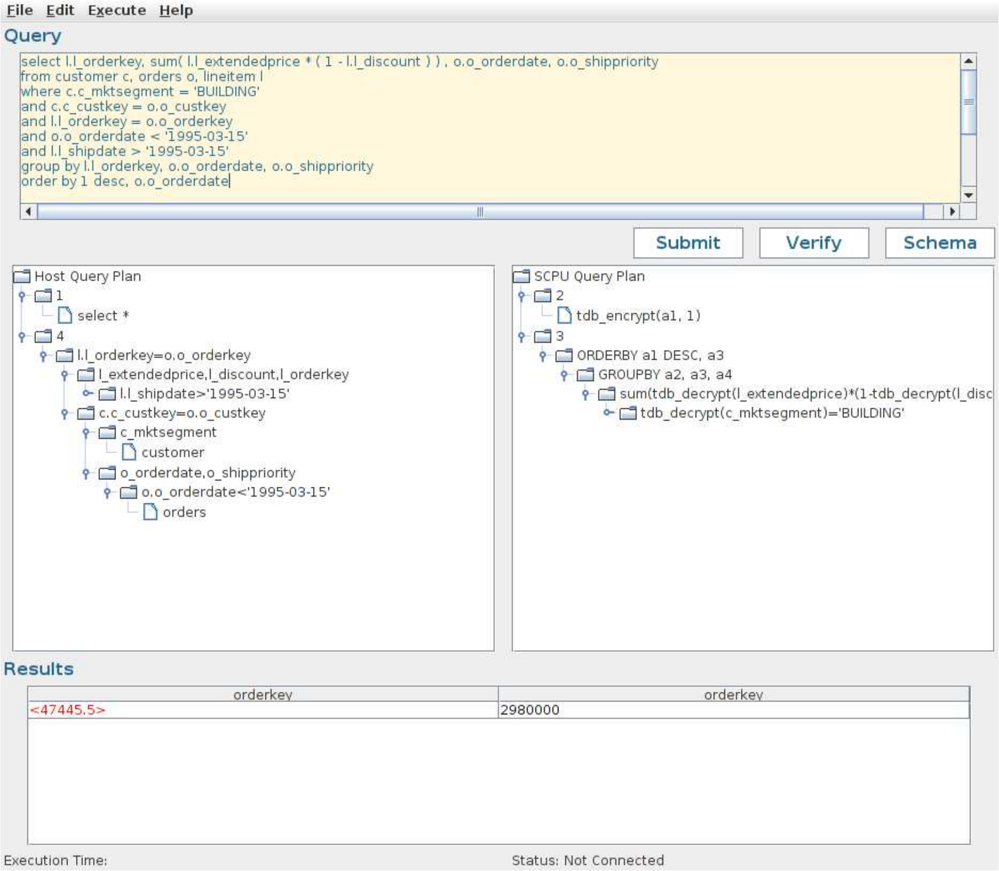
In [25] SCPUs are used to retrieve X509 certificates from a database. However, this only supports key based lookup. Each record has a unique key and a client can query for a record by specifying the key. [226] uses multiple SCPUs to provide key based search. The entire database is scanned by the SCPUs to return matching records.

[216] implements arbitrary joins by reading the entire database through the SCPU. Such as approach is clearly not practical for real implementations since it is lower bounded by the Server↔SCPU bandwidth (10 MBps in our setup).

Chip-Secured Data Access [165] uses a smart card for query processing and for enforcing access rights. The client query is split such that the server performs majority of the computation. The solution is limited by the fact that the client query executing within the smart card cannot generate any intermediate results since there is no storage available on the card. In follow-up work, GhostDB [189] proposes to embed a database inside a USB key equipped with a CPU. It allows linking of private data carried on the USB Key and public data available on a server. GhostDB ensures that the only information revealed to a potential spy is the query issued and the public data accessed.

Both [165] and [189] are subject to the storage limitations of trusted hardware which in turn limits the size of the database and the queries that can processed. In contrast TrustedDB uses external storage to store the entire database and reads information into the trusted hardware as needed which enables it to be used with large databases. Moreover, database pages can be swapped out of the trusted hardware to external storage during query processing.

In [36] a database engine is proposed inside a SCPU for data sharing and mining. The SCPU fetches data from external sources using secure jdbc connections. The entire data is treated as private with queries completely executed inside the coprocessor. We find that using the IBM 4764 for processing queries entirely within the trusted hardware module, without utilizing server cpu cycles, is up to 40x slower than traditional server query processing. This is so even when the trusted hardware has access to the local server file system using our *Paging Module* (section 2.5). Hence using jdbc connections as in [36] can only have higher processing overheads.



```
select l.l_orderkey, sum( l.l_extendedprice * ( 1 - l.l_discount ) ) , o.o_orderdate, o.o_shippriority
from customer c, orders o, lineitem l
where c.c_mktsegment = 'BUILDING'
and c.c_custkey = o.o_custkey
and l.l_orderkey = o.o_orderkey
and o.o_orderdate < '1995-03-15'
and l.l_shipdate > '1995-03-15'
group by l.l_orderkey, o.o_orderdate, o.o_shippriority
order by 1 desc, o.o_orderdate
```

```
Host Query Plan
  1
    select *
  4
    l.l_orderkey=o.o_orderkey
      l_extendedprice,l_discount,l_orderkey
        l.l_shipdate>'1995-03-15'
      c.c_custkey=o.o_custkey
        c_mktsegment
          customer
        o_orderdate,o_shippriority
          o.o_orderdate<'1995-03-15'
            orders
```

```
SCPU Query Plan
  2
    tdb_encrypt(a1, 1)
  3
    ORDERBY a1 DESC, a3
      GROUPBY a2, a3, a4
        sum(tdb_decrypt(l_extendedprice)*(1-tdb_decrypt(l_disc
          tdb_decrypt(c_mktsegment)='BUILDING'
```

orderkey	orderkey
<47445.5>	2980000

Figure 2.18: TrustedDB Demo Client.

2.10 Demo Application

The TrustedDB client application [31] illustrates how TrustedDB enables generalized query processing over encrypted data and covers the following.

- Running queries and performing data manipulation over outsourced encrypted data.
- Visualizing the workload schedule between the host server and the secure coprocessor, which is key in making the use of trusted hardware in query processing practical.
- Gauging the security mechanisms employed to ensure the execution of queries over sensitive data in a remote secure environment.

2.11 Conclusions

This chapter's inherent thesis is that in outsourced contexts, query processing inside secure hardware processors is 1-3 of magnitude cheaper than using cryptography despite the higher acquisition cost of trusted hardware. Further, the use of trusted hardware removes limitations on query expressiveness that are inherent in current cryptography-based mechanisms.

To realize the benefits of trusted hardware we designed, implemented and evaluated TrustedDB. TrustedDB is the first relational database that supports full SQL query execution over encrypted data.

Chapter 3

Query Authentication with CorrectDB

3.1 Introduction

3.1.1 Background and Motivation

Query authentication (QA) requires strict guarantees for both correctness and completeness of query results returned by potentially compromised providers. For users of cloud-based databases, QA offers the ability to prove non-compliance by service providers.

Existing research tackles the QA problem by designing authenticated data structures (ADS). An ADS is constructed and uploaded by the data owner to the service provider. At query execution time, ADS enables the service provider to construct a proof that a client verifies. The proof ensures the client that query results are correct and complete.

The proofs in existing QA solutions are large, making the server to client proof transmission a costly operation. Also, due to large proof sizes, the performance of QA solutions based on client-side checking is limited by network bandwidth and latency. Moreover, a single QA solution for all query types has not been proposed yet.

3.1.2 Our Contribution: Low-cost Query Authentication Using Trusted Hardware

We propose to look at the QA problem from a different angle. Specifically, we posit that server-hosted, close-to-data trusted hardware acting on behalf of clients can result in a general purpose QA solution that is also significantly cheaper and more efficient than prior work. We were particularly encouraged by the results from TrustedDB (Chapter 2) showing that for data confidentiality, a trusted hardware-based solution is significantly cheaper than cryptographic alternatives [64].

In this chapter, we show the cost benefits of trusted hardware for QA (Section 3.6.5). In addition to lowering costs, the use of trusted hardware for QA significantly increases the

QA functionality (Section 3.5). For example, support for update queries and replay attack detection.

To realize the benefits of trusted hardware in QA, we design CorrectDB, an SQL DBMS that deploys tamper proof secure coprocessors at the server's side to provide full QA guarantees cheaply and efficiently. CorrectDB achieves efficient, low-cost QA through its close proximity to the outsourced data, by minimizing the authentication data, and by reducing the client-server communication overheads.

3.1.3 Chapter Outline

Section 3.2 describes the adversarial model. QA requirements are listed in Section 3.3. Section 3.4 provides an overview of exiting QA work along with critical cost insights. The benefits of trusted hardware for QA are discussed in Section 3.5. Section 3.6 details the CorrectDB architecture. Experimental results are presented in Section 3.7. Section 3.8 discusses the relationship between data confidentiality and QA. Finally, Section 3.9 concludes the chapter.

3.2 Model

Data Owner

Data is placed by the data owner with a remote, untrusted *service provider*. For authenticity and integrity of query results, the data owner computes and places additional authentication data with the provider. The data owner issues search and update queries to the provider. In existing QA work, the data owner performs all updates to the provider-side authentication data. In CorrectDB, the server-side, trusted hardware manages the authentication data on behalf of the data owner.

Clients

Clients authorized by the data owner query the outsourced datasets through an interface exposed by the provider. A client query can either perform a search or a data-only-update operation. Clients cannot update the server-side authentication data.

Adversary

The service provider is not trusted. Due to compromise or malicious intent, the provider may violate one or more of the QA security requirements (Section 3.3).

3.3 Query Authentication

Query Authentication (QA) has two essential requirements – correctness and completeness.

Correctness

Correctness has two components. First, each tuple in the query result should be authentic, that is, all tuple's must originate from the original database that was uploaded to the provider's site. It must be impossible for the provider to introduce any spurious tuples in the result. Second, each tuple in the query result must satisfy the query predicates exactly, thus ensuring that query execution adhered to all predicates specified in clients' queries.

Completeness

For completeness, all tuples that are supposed to be part of the query result must be present in the result, that is, query execution should not exclude any valid tuples from the result.

3.4 Existing QA Solutions

In this section, we overview existing QA work. The overview identifies cost points of existing QA solutions. In addition, using published experimental result, we identify state-of-the-art in QA.

Existing QA solutions follow the deployment model of Section 3.2 with the following key difference. All database updates, including the modifications to the authentication data are performed by the data owner. Client queries are read-only.

At a high level, existing solutions work similarly using the following steps.

- Step 1: The data owner uploads the database to the service provider. The data owner then computes and uploads additional data structures known as *Authenticated Data Structures* (ADS). In addition, depending on the specific QA scheme, the owner may distribute certain information to the clients.
- Step 2: A client submits a query request to the service provider.
- Step 3: The provider executes the query to get the desired results. Using the ADS, the provider also computes a proof, which the client uses to verify correctness and completeness of query results. This proof is referred to as a *Verification Object* (VO).
- Step 4: The service provider delivers both the query results and the VO to the client.
- Step 5: Using information from the owner together with query results and VO from the service provider, the client determines whether QA assurances are met.

The properties of ADS and VO prevent the provider from compromising integrity of the query results.

Existing QA solutions can be classified as either *tree-based* or *signature-based*. The two categories differ in the data structures used for the ADS and the VO; and hence in the query execution and verification.

In the following we discuss the general strategy of tree and signature-based solutions. Discussion of a general strategy suffices to identify cost points. In Chapter 9, we detail existing tree and signature-based QA solutions.

3.4.1 Tree-based QA solutions

In tree-based approaches, the ADS is constructed as a tree. For example, MB-tree [155] and VB-tree [201]. As part of query execution, service provider traverses the tree and gathers the nodes that form the VO, which is sent to the client along with the query results. The client reconstructs the traversal path used in query execution to verify correctness and completeness.

The simplest example of a tree-based QA solution is a merkle hash tree [172]. A merkle hash tree (MHT) authenticates a set relationship between multiple items such as tuples in a database. Let t_1, t_2 ... t_n denote individual tuples in the dataset. Then, the leaves of the MHT are composed of the hash of each individual tuple, that is, $H(t_1)$, $H(t_2)$ ... $H(t_n)$. Each internal tree node is constructed as the hash of its two child nodes. For instance, the parent node of $H(t_1)$ and $H(t_2)$ is $H(H(t_1)||H(t_2))$. To subsequently prove that a tuple belongs to the dataset associated with the MHT, it suffices to recompute and verify the root of the tree from the tuple's value and the siblings of all the nodes in the traversal path up to the root of the MHT.

Cost Insights for Tree-based Solutions

Transferring additional ADS nodes to the clients for verification means that the VO sizes in tree-based approaches can become quite large. Large VO sizes increase both query latency and the cost of data transmission. From the results of Chen et al. [65] we know that cloud-to-client data transfer costs upwards of 3500 picocents[1] per bit, 2-3 orders of magnitude higher than processing costs.

3.4.2 Signature-based QA solutions

Signature-based approaches provide a mechanism to verify the ordering between tuples when using specific search attributes. For QA, an authenticated chain of unforgeable signatures is constructed by the data owner. At query time, the service provider gathers the signatures of all the tuples that comprise the contiguous range query result. The set of signatures comprises the VO. Since each tuple is now linked to its predecessor and successor in an unforgeable manner, the client can verify that no tuple is either illicitly inserted or omitted from the query result.

An example of a signature-based QA solution is DSAC [185]. In DSAC [185], the ADS consists of the signature of each individual tuple along with its immediate predecessor, that is, $S(H(t_i)||H(t_{i-1}), SK_{DO})$, where $S(\cdot)$ denotes a signature operation and t_{i-1} is the

[1] 1 US picocent = \$1 × 10^{-14}.

predecessor of t_i when sorted on the search attribute. By including the predecessor in the signature, a chain of all tuples is formed ordered on the search attribute. The VO then consists of the signature of all tuples in the query results. The client verifies that the set of tuples received in the result form a valid chain.

Cost Insights for Signature-based Solutions

The VO for signature-based solutions that employ signature aggregation [181,182] is a single signed message. Hence, signature-based solutions do not incur the high transmission costs of tree-based approaches, at least not for range queries. However, the operations needed to construct small VO*s* involve cryptographic operations, such as modular multiplications and exponentiations. Cryptographic operations are expensive to compute on the server side due to the large number of CPU cycles involved. Computing a single cryptographic trapdoor server-side costs up to 30,000 picocents [65].

3.4.3 Survey of Empirical Evaluation

QA performance entails the following three key metrics.

- **VO size (VOS):** The VO is transferred over the network and thus determines query latency and bandwidth usage. Hence, all QA solutions target minimizing the VO size.

- **Query Execution Time (QET):** The provider builds VO by computing on the ADS. This computation adds to the overall execution time.

- **Verification Time (VT):** Client-side computation is required to verify the authenticity of query results. Since clients may be limited in computing abilities it is imperative that client-side processing is minimal.

Over time, newer QA solutions have shown their benefits over prior work using experimental comparisons. By surveying all published experimental results, we draw a comparison map and identify the most efficient QA solutions. Figure 3.1 shows the comparative summary of existing QA research.

Survey Conclusions

For selection and range queries, signature-based schemes using signature aggregation [185] provide the smallest VO and thus have the least client-server network overhead. However, tree-based solutions [155] in 2006 were shown to perform better than signature-based schemes when evaluated on both QET and VT. The advantage reverses in 2009 with the speedup in crypto operations [203]. Crypto operations such as signature verification and aggregation are CPU but not I/O intensive. Faster processors in 2009 reverse the 2006 result and signature-based schemes now perform better on the metrics QET and VT as well. Signature-based

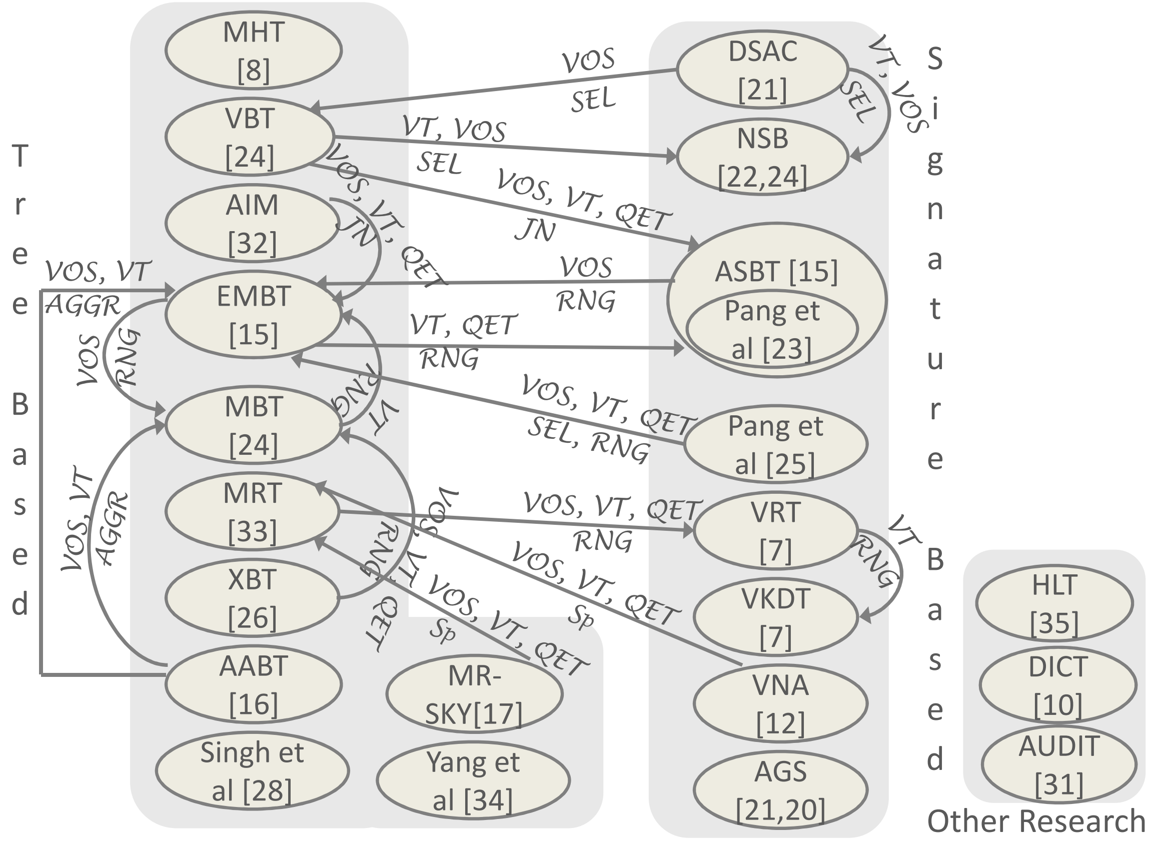


Figure 3.1: Comparison of published results. Metrics: VOS = Verification Object (VO) Size, VT = Client-side Verification Time, QET = Server-side Query Execution Time. Queries: SEL = Selection, AGGR = Aggregation, RNG = Range, JN = Join. A $\longrightarrow$ B = A outperforms B for the metrics denoted above the arrow and queries denoted below. Nodes with no edges = no experimental comparisons available.

schemes are not efficient for join processing. AIM [269] is the only comprehensive tree-based approach for join processing and is shown to perform better than other tree-based approaches.

3.5 The Case for Trusted Hardware in QA

In Section 2.3, we described secure coprocessors (SCPU) including scpu-based application development and security. We now discuss the benefits of SCPUs for QA. In section 3.6, we describe how the benefits are realized in CorrectDB.

Data Proximity

A major performance and cost consideration for QA is VO size. Since VO needs to be transferred from provider to client, VO size directly adds to network costs. VO cost factor specifically applies to tree-based solutions.

SCPUs communicate with the host server locally over the PCI bus. Hence, in SCPU-based QA, the VO can be processed server-side within the SCPU virtually eliminating the VO transmission costs. Later, in section 3.6.5 we show that despite the higher acquisition and processing costs of SCPU, the proximity factor leads to significant savings in overall query processing costs.

Query Expressiveness

The general-purpose SCPUs can be programmed to execute arbitrary queries. Thus, limitations on the query expressiveness can now be removed and a single solution utilizing SCPUs can be used for authenticating range, join, and aggregation queries even with complex predicates.

Database Updates

In existing QA solutions, there is no trusted component server-side. Hence, the data owner cannot issue update queries to the server. Instead, the data owner performs all update operations locally [185]. For an update operation, the data owner fetches the relevant tuples from the server, modifies them locally, constructs new ADS, and upload the new tuples and ADS back to the server. The data transfer overhead involved in owner-side updates is significant. Moreover, tree-based approaches require redistribution of the new ADS root hash computed by the owner to all clients. Distribution to clients increases transfer costs.

Using SCPUs, the data owner can issue an update query directly to the server-side SCPU. All updates are then performed by the SCPU incurring no additional data transfer overheads (Section 3.6.7). Also, the ADS can now completely be stored server-side. Hence, if the query result verification is also performed by the SCPU, redistribution to clients is avoided.

Access Control

If the provider is not trusted for QA, the provider can also not be trusted enforce access control policies. In a SCPU-based solution, access control can be efficiently enforced by the trusted, server-side SCPU.

Untrusted Clients

An update operation involves modifications to both data and ADS. In existing QA solutions, updates are limited to the data owner only. If clients were permitted to perform data updates, it would be necessary to give the clients access to the ADS as well. Hence, a compromised client could alter the ADS causing incorrect results to be computed for other clients' queries.

In a trusted hardware-based solution, the SCPU acts as a trusted entity on behalf of the owner and performs all updates server-side. Clients can now issue update queries but the underlying data and ADS are modified only by the server-side SCPU. Further, client update queries can be filtered by the SCPU enforced access control mechanisms, thereby avoiding malicious updates by compromised clients.

Client Synchronization

If clients store authentication information, such as the root hash in tree-based approaches, any change to the authentication data involves updates on all other clients in synchronization. If clients are not synchronized, some clients may end up with stale data. For large number of dispersed clients, synchronization becomes a problem. In a SCPU-based QA solution, the SCPU maintains and updates information for multiple clients. Hence, SCPU-based QA avoids client synchronization.

Replay Attacks

A replay attack occurs when the server sends old authentication data to clients. To prevent replay attacks, tree-based approaches require locking the entire database when the data owner performs updates. Further, the owner is required to securely distribute the new ADS root to all clients. Similar to tree-based schemes, signature-based schemes are also vulnerable to replay attacks on tuple chain signatures. By computing the ADS locally during updates, SCPU-based designs can avoid replay attacks entirely in an efficient manner (Section 3.6.7).

Querying without ADS

In existing work, queries with predicates on attributes that do not have an associated ADS require intermediate results to be transferred for client-side evaluation. Transfer of intermediate query results to clients incurs significant data transmission costs.

In comparison, server-side SCPUs can leverage an ADS on attributes other than the search attribute. Hence, intermediate results do not need client-side processing eliminating data transfer costs (Section 3.6.4).

Data Privacy

For privacy, data can be encrypted before uploading to the provider. However, encryption greatly limits the query predicates to very simple conditions [91, 123]. Within a SCPU however, data can be processed in plaintext and complex predicates can be evaluated (Section 3.8).

3.6 CorrectDB Architecture

CorrectDB is built around a set of core components (Figure 3.2) including a request handler; a query parser; server and SCPU-side query processors; and a crypto library.

3.6.1 Overview

In this section, we give an overview of CorrectDB query processing. In Sections 3.6.2-3.6.5, we detail processing of specific query types.

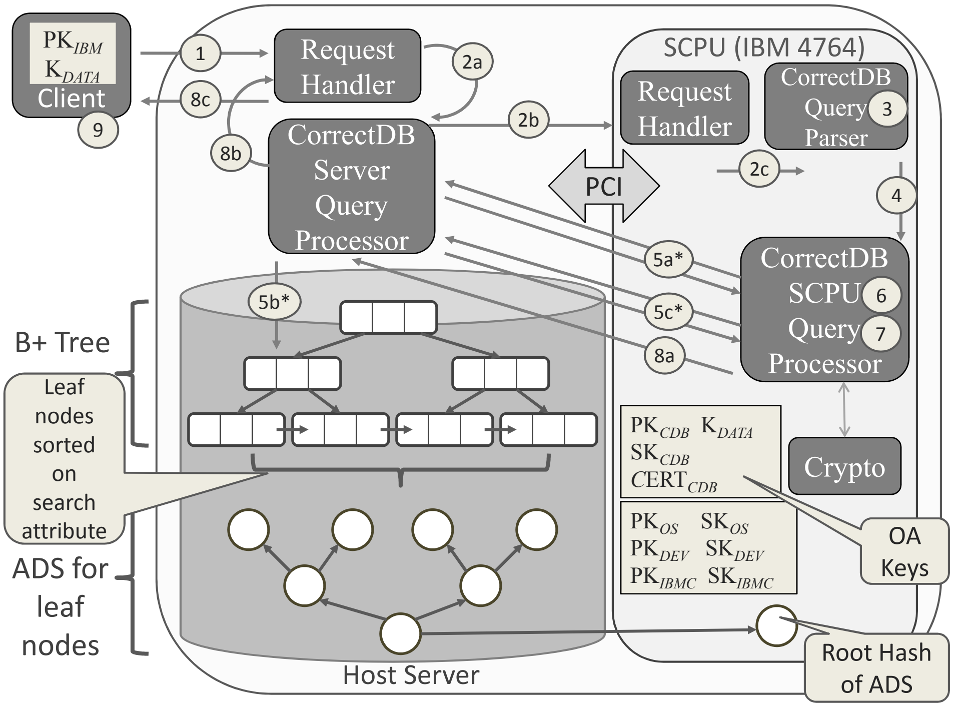


Figure 3.2: CorrectDB Architecture

The outsourced data is stored at the provider's site. For each relation R in the database, a B^+-tree is built on the search attribute(s) of R. In addition, a separate merkle hash tree (MHT) based ADS is built on the leaf nodes of each of the B^+-trees. Each leaf of the ADS is a hash of a B^+-tree leaf node's contents. An internal ADS node is computed as the hash of its children. Figure 3.3 illustrates the MHT ADS construction. The root hash of each ADS is stored inside the SCPU and is never accessible from the outside. Since the root hash is stored within the SCPU, the hash need not be signed. Not signing the root hash saves an expensive signature operation on each update and multiple signature verifications for each query.

Note that for static data sets, both the B^+-trees and their ADS can be constructed by the data owner prior to uploading the database. However, since CorrectDB supports insert and update queries, the structures are continuously updated by the server-side SCPU in response to client update queries.

Query processing in CorrectDB occurs as follows (Figure 3.2):

- A client queries the server through a standard SQL interface.
- The server forwards the query to the SCPU request handler. The SCPU request handler, in turn, forwards the query to the CorrectDB Query Parser.
- The parser rewrites the client query into two sub-queries: a server subquery and a SCPU subquery.

Step	Description	Details in
1	Client submits query	
2,3	Forward client query	
4	Parse into server and scpu-side sub-queries	section 3.6.1
5	Forward parsed queries	
6*	Request leaf nodes from server	Sections 3.6.2 to 3.6.5
7*	Find leaf and MHT nodes	
8*	Request MHT nodes from server	
9*	Verify leaf nodes and process query	
10	Sign digest of query results (VO)	
11,12	Forward results and signed VO	
13	Client-side verification	Section 3.6.6

Table 3.1: Legend for figure 3.2, * = multi-round steps.

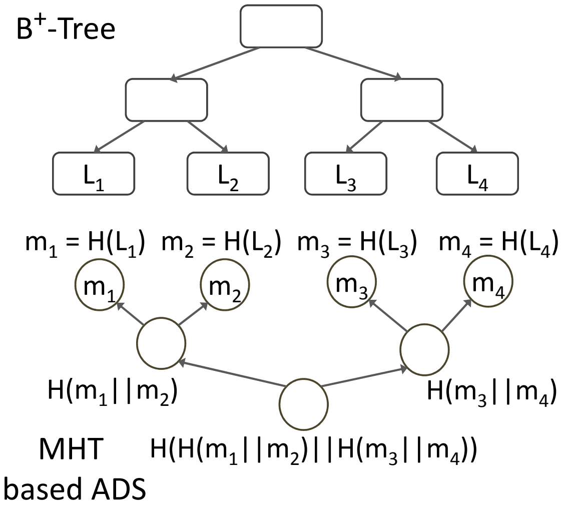


Figure 3.3: MHT based ADS.

- The parser then forwards the rewritten queries to the CorrectDB query processor.

- CorrectDB query processor forwards the server subquery to the server-side query processor. Server-side processor executes the server subquery on untrusted server host. The results of server-side execution are verified by the SCPU query processor. For verification, SCPU query processor using the MHT ADS.

- The SCPU subquery is processed entirely by the SCPU query processor to get the final results.

- The SCPU query processor signs the final query result.

- The signed result is sent to the client by the server-side request handler.

- The client verifies the signature on the final query result.

Query Parsing and Execution Example

A query consists of various SQL operations, such as selection, range predicate, projection, aggregation, order-by, and group-by. The CorrectDB query parser's job is to rewrite the original client query into sub-queries ensuring the following:

- Processing within the SCPU is minimized.
- Any intermediate results generated by server-side query processing can be validated by the SCPU using the ADS built on the leaf nodes of the B^+-tree indices of the relevant relations.
- Any server-side operations that cannot be authenticated by the SCPU are by the SCPU.
- The net result of the sub-queries is the same as if the original client query was executed without any rewrites.

To illustrate how queries are rewritten and processed, consider the following query derived from the TPC-H [12] benchmark.

```
SELECT sum(l_extendedprice*l_discount), o_priority
FROM   lineitem, orders
WHERE  l_shipdate >= '1993-01-01'
AND    l_shipdate < '1994-01-01'
AND    o_orderdate between '1992-01-01' AND '1993-01-01'
AND    l_discount between 0.05 AND 0.07
AND    l_orderkey = o_orderkey
AND    o_priority in ('W', 'R', 'Q')
```

Suppose we have B^+-tree indices and MHT based ADS on the attributes *l_shipdate* and *o_orderdate*. Then, the server-side subquery searches for all leaf nodes from the relations *lineitem* and *orders* that satisfy the following conditions.

```
    l_shipdate >= '1993-01-01'
AND l_shipdate < '1994-01-01'
AND o_orderdate between '1992-01-01' AND '1993-01-01'
```

For query execution, server uses the B^+-tree indices on attributes *l_shipdate* and *o_orderdate*. The server does not identify individual tuples that satisfy query predicates. Instead, the server locates only the B^+-tree leaf nodes that contain the tuples which may potentially form the query result. The server and the SCPU-side query processors then engage in a interactive protocol. In each round, the SCPU query processor reads a set of leaf nodes and verifies correctness and completeness using the MHT based ADS. On each set of leaf nodes read, the SCPU query processor evaluates the SCPU subquery:

```
SELECT sum(l_extendedprice*l_discount), o_priority
FROM   lineitem, orders
AND    l_discount between 0.05 AND 0.07
AND    l_orderkey = o_orderkey
AND    o_priority in ('W', 'R', 'Q')
```

The SCPU-side verification step ensures QA for the server subquery results (Section 3.6.2). Since the server subquery filters out unwanted tuples from relations *lineitem* and *orders*, the SCPU subquery only processes a subset of the data. The filtering is essential since otherwise the join condition $l\_orderkey = o\_orderkey$ would become an expensive operation to perform entirely within the SCPU.

3.6.2 Range Queries

Consider the execution of a range query for all tuples with keys in the range (L, U), $L \leq U$. Let R denote the set of tuples in the query result. Let L_1, L_2,..., L_n denote B^+-tree leaf nodes. Since the B^+-tree stores data sorted on the search keys, the same sort order applies to the leaf nodes as well including the ordering of tuples within a single leaf, that is, for two leaf nodes L_i and L_j where $i < j$, we have $\forall\ t \in \mathsf{L}_i$, $\forall\ t^{'} \in \mathsf{L}_j$, $t.key \leq t^{'}.key$. The server searches all tuples in the range and identifies the leaf nodes L_l, L_{l+1},...L_m, where $l \geq 1$, $m \leq$ n and $l \leq m$. The server then sends the leaf nodes to the SCPU query processor. The SCPU query processor computes the hash of the leaf nodes $H(\mathsf{L}_l)$, $H(\mathsf{L}_{l+1})$,...$H(\mathsf{L}_m)$. Using the computed hash values and by requesting additional ADS nodes from the server, the SCPU query processor constructs and verifies the root hash of the MHT ADS. Finally, the SCPU query processor scans the leaf nodes to find all tuples t, such that $t.key \in (L, U)$, which comprise the result set R.

The following properties hold:

Correctness: $\forall\ tuples\ t \in \mathsf{R}$**,** $t.key \in (L, U)$

Proof (sketch): The SCPU verifies evaluation of the range predicate. Correctness then reduces to the collision resistance of the MHT. Since the MHT based ADS is used to verify the integrity of each leaf node the server cannot alter any tuples to subvert query correctness.

Completeness: $\forall\ t$**,** $if\ \ t.key \in (L, U)\ then\ t \in \mathsf{R}$

Proof (sketch): Completeness is violated iff, $\exists$ tuple t such that $t.key \in (L, U)$ but $t \notin \mathsf{R}$. Note that the leaf nodes L_l, L_{l+1}, ... L_m are consecutive nodes at the lowest level of the B^+-tree. The sequence of leaf nodes is verified by the SCPU using the MHT since the same leaf nodes correspond to the leaves at the lowest level of the MHT. Thus, when the root hash of the MHT is constructed, the chain linking between the B^+-tree leaf nodes is automatically verified.

In addition, the SCPU checks the following two conditions.

- $min\{t.key,\, t \in \mathsf{L}_l\} < L$, and
- $max\{t.key,\, t \in \mathsf{L}_m\} > U$.

The above conditions ensure that all leaf nodes that potentially contain result tuples are included in the server-side subquery results. Hence, the above conditions together with chain linking of consecutive leaf nodes guarantee query completeness. Proof then reduces again to collision resistance of the MHT.

3.6.3 Projections

Projection operations are performed by the SCPU query processor. Thus, no additional ADS is needed to support projections. For each leaf node being processed, the SCPU query processor simply discards any attributes not required for current query processing. Supporting projections within the SCPU enables CorrectDB to build the MHT based ADS on B^+-trees leaf nodes rather than on individual tuples. An MHT on leaf nodes requires significantly less hash operations to verify intermediate server-side query results.

In effect, for the verification of L_n leaf nodes of average size L_s KB with an average tuple size of T_s bytes, the number of hash operations required during SCPU-side verification are reduced from $\frac{L_n * L_s * 1024}{T_s}$ to L_n. Moreover, hashing entire leaf nodes enables utilization of SCPU's crypto-hardware engine, which has high throughputs for bulk operations.

3.6.4 Joins

Join processing is a relatively straight-forward extension of range processing. A join query specifies a condition on two attributes. The conditions are of the form $R.A \circ S.B$, where R and S are relations; A and B are attributes of relations R and S, respectively; and $\circ$ is the join operator.

CorrectDB uses two methods for evaluating join queries. For a query with a ordering-based join operators such as $=, <$ and $>$, CorrectDB uses a sort-merge join. For all other join operators, CorrectDB uses a full nested join.

Ordering-based join operators ($=$, $<$, $>$, $\leq$, $\geq$)

CorrectDB uses a sort-merge join for ordering-based join operators.

To illustrate, consider authentication of the join query $\sigma_{P_r}(R) \bowtie_{R.a=S.b} \sigma_{P_s}(S)$. The server uses the B^+-trees on $R.a$ and $S.b$ to identify all the leaf pages of R and S that contain the query results. If the predicates P_r and P_s contain conditions on attributes other than $R.a$ and $S.b$ or if both P_r and P_s are empty, then all leaf pages of both relations potentially contain result tuples. If P_r and P_s contain predicates only on $R.a$ and $S.b$, then the server identifies the subset of leaf nodes by traversing the B^+-trees trees of relations R and S.

Once the server identifies leaf nodes, the server and SCPU-side query processors engage in an multi-round protocol. In each round, the server-side query processor sends a set of leaf nodes, in order, to the SCPU query processor. The SCPU query processor performs a sort-merge scan of leaf nodes and constructs the final result set. Just as in range query processing (Section 3.6.2), the SCPU query processor verifies both integrity and consecutive linking of all leaf nodes read using the respective MHT based ADS. Verification by SCPU-side query processor ensures both correctness and completeness. Projections and additional predicates if present, are processed by the SCPU query processor.

Arbitrary Joins

Nested loop joins are required when join operations are complex and cannot be computed using a sort-merge mechanism, or when there are no indices available on the join attributes. Nested loop joins can also be the preferred choice when the outer relation is small. Unfortunately, nested loop joins can be expensive. Suppose relations R and S participating in the join have n_r and n_s number of leaf nodes, respectively. A nested join operation would then require up to $n_r * n_s$ leaf nodes to be read into the SCPU.

The cost of nested joins can be reduced by utilizing the available SCPU memory. Let M be the size of SCPU memory and $M/2$ space inside the SCPU is dedicated to hold B^+-tree leaf nodes. Then, the SCPU query processor will read $n_r + (\frac{2*P*n_s}{M})^2$ leaf nodes to perform the join. The verification procedure described in Section 3.6.2 ensures correctness and completeness. If an ADS is not available for a particular join attribute, the ADS of another attribute of the same relation is used to perform the join within the SCPU. When using ADS on an attribute not specified in the join condition, all leaf nodes are scanned by the SCPU query processor.

3.6.5 Aggregations, Grouping and Ordering

Existing tree and signature-based mechanisms require the client to perform aggregation operations. Thus, additional data, which is not part of the final query results is transferred to the client incurring both query latency and data-transfer-cost overheads. CorrectDB however, performs all aggregation operations inside the SCPU and only the final result is sent to the client. Thus, CorrectDB significantly lowers the amount of data transferred. Later, in Section 3.7, we compare the performance and cost of aggregation operations for CorrectDB with the data transfer in other QA solutions.

Similar to aggregation operations, group-by clauses are processed entirely by the SCPU query processor. No additional tuples are transferred to the client other than the final query result. Suppose, on average a group-by clause aggregates n_g values and the total number of tuples satisfying the query predicates is n_r. Then, SCPU processing of group-by clauses reduces the number of tuples transferred from n_r to $\frac{n_r}{n_g}$.

Order-by clauses cannot be optimized like aggregation and group-by clauses. If an order-by clause is processed as the last step in query execution, then there is no reduction in the number of tuples that need to be transferred to the client. If the client has higher processing capacity than the SCPU, such as the case of desktop clients, then client-side ordering will perform better that CorrectDB at least in execution time.

3.6.6 Client Side Verification

In CorrectDB, unlike other QA solutions, client-side verification is minimal. The SCPU query processor computes and verifies final query results and transfers a signed hash of query results to client. Client-side processing is limited to r hash operations and a single signature verification per query, where r is the number of tuples in the query result.

The tuples comprising the result set R are identified by the SCPU query processor while processing the SCPU subquery. Let $\mathsf{R} = \{t_1, t_2, ..., t_r\}$. The SCPU then computes the digest of R, $D(\mathsf{R})$ as

$$D(\mathsf{R}) = H(C_{id}||Q_c||\ Nonce||H(t_1)||H(t_2)||...||H(t_r)).$$

Here, C_{id} is the client identifier and Q_c is the client query. $Nonce$ is a per query fresh random value sent by the client within the query request. The $Nonce$ establishes a unique association between the result R with the query Q_c. The unique association prevents the server from matching stale results with a recent query, thereby thwarting replay attacks.

The SCPU signs $D(\mathsf{R})$ using a private key. The signed message $S(D(\mathsf{R}), SK_{CDB})$ is sent to the client along with the query results R. SK_{CDB} is the CorrectDB application secret key. The client then recomputes $D(\mathsf{R})$ and verifies the signature using the CorrectDB application public key.

Standardized outbound authentication mechanisms exist for key setup and for communication of public keys to clients. We described outbound authentication in Section 2.3.

3.6.7 Database Updates

Clients issue update statements to request modifications to the server-hosted data. An update query requires secure modifications to the B^+-tree storing the tuples and to the MHT based ADS used by the SCPU for verification. Both B^+-tree and ADS are modified only by the SCPU via local interaction with the server-side query processor. Updates do not involve the data owner. Also, since root hashes for the ADS are stored within the SCPU, client synchronization is not needed.

To illustrate update query processing in CorrectDB, consider the following update query:

```
UPDATE lineitem SET l_discount = l_discount + 0.01
WHERE  l_shipdate >= '1993-01-01'
AND    l_shipdate < '1994-01-01'
AND    l_discount between 0.05 AND 0.07
```

The above query is processed as follows.

Step 1: The query is parsed by the SCPU query parser and rewritten into a server subquery

```
SELECT * FROM lineitem
WHERE  l_shipdate >= '1993-01-01'
AND    l_shipdate < '1994-01-01'
```

and a SCPU subquery

```
UPDATE lineitem SET l_discount = l_discount + 0.01
WHERE  l_discount between 0.05 AND 0.07
```

Step 2: The server then executes the first subquery and finds all the leaf nodes that require modifications. Additional leaf nodes may be identified if the server requires rebalancing of the index.

Step 3: The server transfers the leaf nodes to the SCPU query processor. The SCPU query processor verifies the leaf nodes using the MHT ADS. Verification ensures correctness and completeness.

Step 4: The SCPU query processor then modifies the nodes as per the second subquery. The SCPU query processor also modifies the MHT ADS recomputing and updating the root hash stored within the SCPU.

Step 5: Only after making changes to the ADS, the modified leaf nodes are transferred back to the server query processor and the final changes applied to the B^{+}-tree.

Steps 1-5 are repeated for each B^{+}-tree that has an MHT ADS.

3.7 Experiments

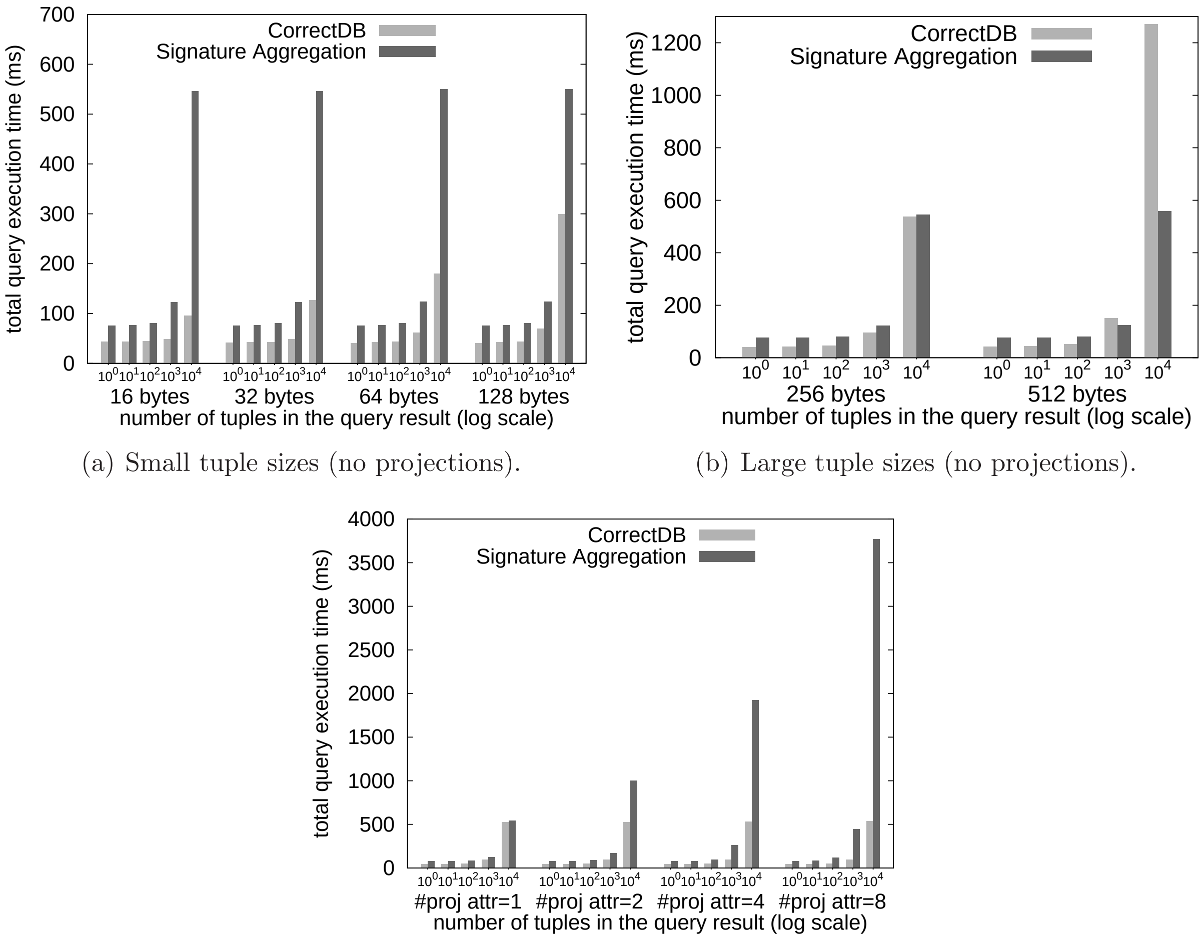
In this section, we present experimental evaluation of CorrectDB. In Section 3.4.3, we identified the best QA solutions for range and join query processing. We compare CorrectDB with the identified solutions. For range query execution we compare CorrectDB with DSAC [185, 203] using signature aggregation. For join query processing we compare with AIM [269].

3.7.1 Setup and Measurements

The SCPU used is the IBM 4764 with 32 MB RAM, and a PowerPC 405GPr at 233 MHz. The SCPU sits on the PCI-X bus of an Intel Xeon 3.4 GHz, 4GB RAM Linux box with kernel 2.6.18. The client is an Ubuntu VM with 1 GB RAM and 2 vCPUs. The CorrectDB stack is written in C.

Measurements are made for the three key metrics – verification object (VO) size, query execution time (QET), and client-side verification time (VT).

For comparative experiments (Figures 3.4-3.6), we combine both query execution time (QET) and client-side verification time (VT) into a single total query execution time. In Figure 3.7(a), we list each time component separately. CorrectDB has a constant VO size,



(a) Small tuple sizes (no projections).

(b) Large tuple sizes (no projections).

(c) Varying # projected attributes.

Figure 3.4: Comparison of CorrectDB and signature aggregation for range queries with and without projections

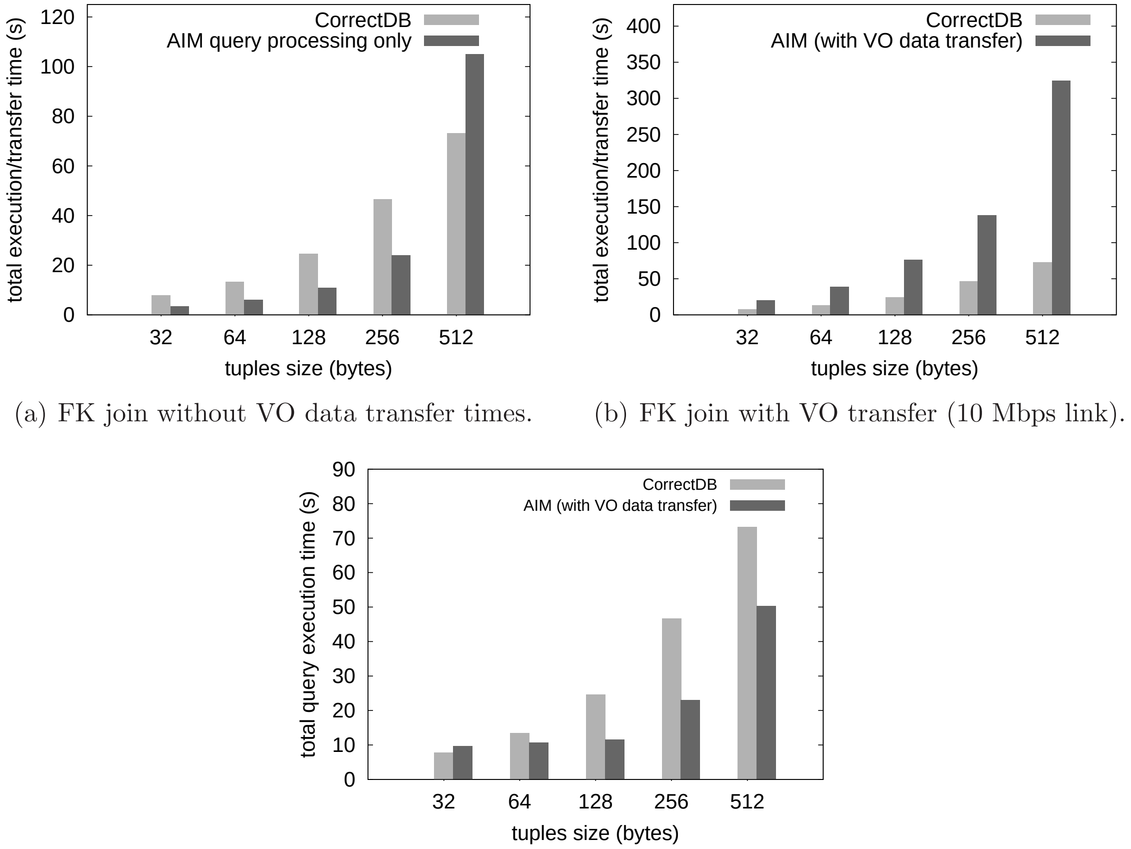
which is a single signed digest of all tuples that comprise the query result. The VO size is the same for all query types. Hence, we only mention the VO size once here.

3.7.2 Range Queries

To evaluate the performance of CorrectDB and signature aggregation, we set up multiple relations with varying tuple sizes. Each relation consists of 10^6 tuples. All relations are indexed using B^+-trees. Tuple keys are random integers between 1 and 10^7.

We evaluate the performance by varying the tuple size and the number of tuples in the query result. As we shall see, both tuple size of number of tuples in query result have varying effects on performance and it is hence important to consider both. Test queries are of the form

```
SELECT * FROM R where R.key > 'LB' AND R.key < 'UB'
```



(a) FK join without VO data transfer times.

(b) FK join with VO transfer (10 Mbps link).

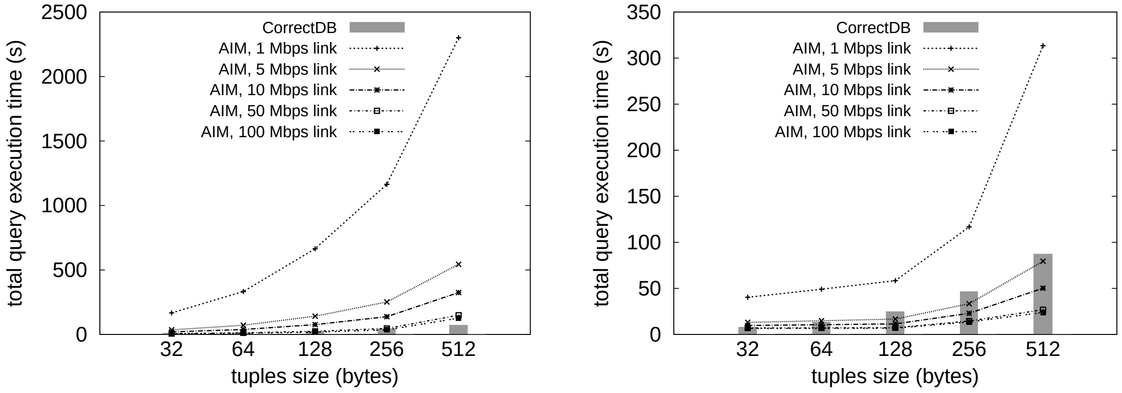
(c) Equi join - 10 Mbps link.

Figure 3.5: Comparison of CorrectDB and AIM for foreign Key and equi-join queries.

'LB' and 'UB' are varied to get query results of different sizes. Although CorrectDB supports a wide range of queries, we only use simple queries here for consistent comparison since other approaches [185, 203, 269] support and evaluate only such basic queries.

Figure 3.4(a) shows the total query execution times for CorrectDB and signature aggregation for varying tuple and result set size. For small tuple sizes, 32 bytes - 128 bytes, CorrectDB performs 2-6x better than signature aggregation. The performance of signature aggregation does not depend on tuple size since signature aggregation is performed only tuple signatures and not on the tuples. Thus signature aggregation has similar times for all tuple sizes. CorrectDB's performance however, varies with both tuple size and number of tuples in the result.

For larger tuple sizes, such as 512 bytes and large result sets, $\geq 10^3$, we observe a shift and signature mechanisms perform better by factors of 1.1-2x (Figure 3.4(b)). Increasing tuple size also increases the size of the leaf nodes that contain tuple data. Since leaf nodes are transferred from the server to the SCPU, the larger the leaf nodes, the higher is the transfer latency. Hence, CorrectDB has higher execution times for large tuples. The transfer latency is evident from Figure 3.7(a), which depicts the breakdown of execution times of different query processing stages of CorrectDB. As seen from Figure 3.7(a), for result sizes



(a) FK join query

(b) EQ join query

Figure 3.6: Effect of data link capacity on performance of AIM compared to CorrectDB.

$\geq 10^3$, the data transfer of leaf nodes from the server to the SCPU is significant.

Paying the penalty to transfer entire leaf nodes into the SCPU enables CorrectDB to support projections and complex selection predicates. The overhead of transfer is thus acceptable. However, note that in the experiments above we have not considered projection operations, that is, the entire tuple is part of the result. Once projections are introduced, CorrectDB regains the performance advantage even for large tuple sizes, as described next.

3.7.3 Projections

We now add projections to the test queries. We fix the tuple size to 512 bytes and vary the number of projected attributes in the select condition of the test query for each run. Each tuple is divided into 16 equal sized attributes.

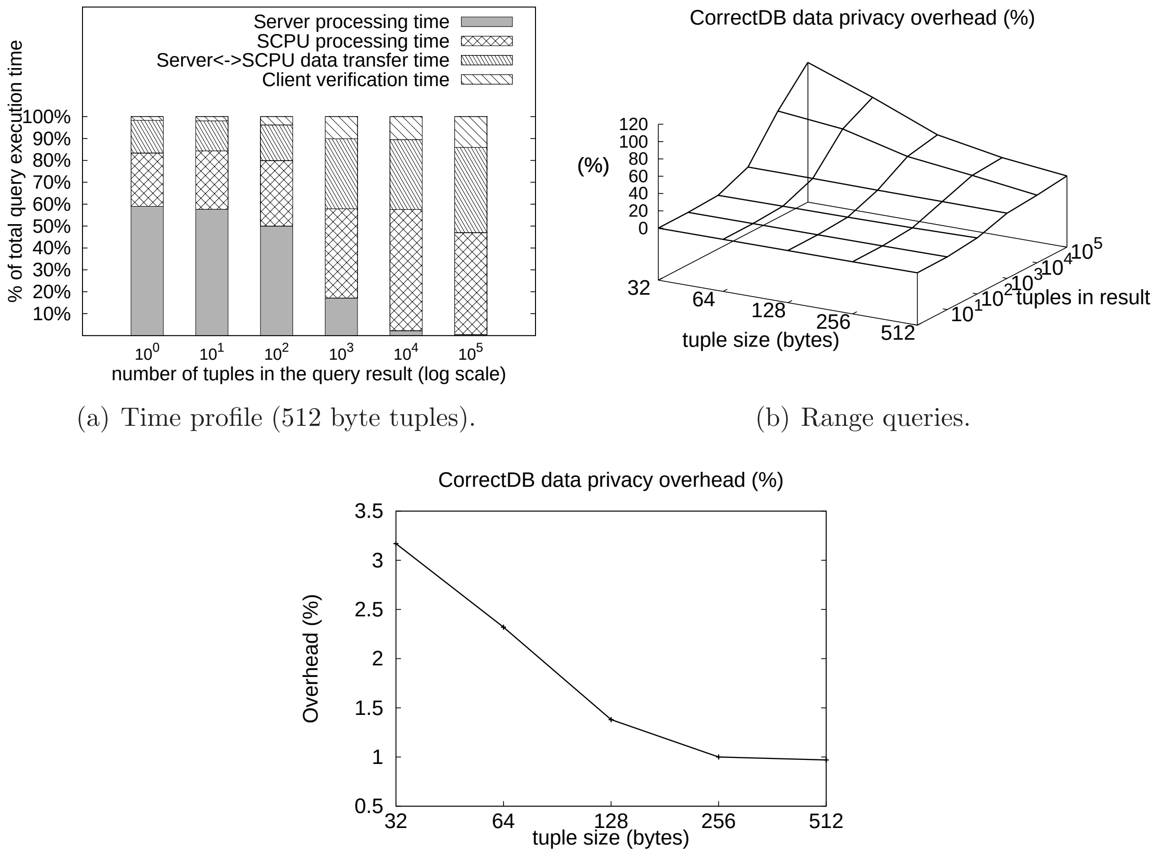
As Figure 3.4(c) illustrates, CorrectDB performs better for all cases by 1.5-7x. In case of signature aggregation, if p tuples are projected out, then the number of signatures to be aggregated increases by p per tuple in the query result [203]. As a result, projections cause higher execution times for signature aggregation as compared to CorrectDB even for large tuple sizes.

Both signature aggregation and CorrectDB have the same VO size and hence are equivalent on the VO size metric. Figure 3.8(a) summarizes which solution, CorrectDB or signature aggregation, performs better for each combination of the parameters, tuple size, result size, and number of projected attributes.

3.7.4 Join Queries

For join queries, we compare CorrectDB with AIM [269]. We use two relations R and S. Each relation has 10^6 tuples. We vary the tuple size from 32 to 512 bytes. Each relation has a B^+-tree on its key attribute and an MHT based ADS. The test queries are of the form

```
SELECT * FROM R, S WHERE R.key = S.key AND R.key > 'LB'
```



(a) Time profile (512 byte tuples).

(b) Range queries.

(c) Equi join.

Figure 3.7: CorrectDB query profile and data privacy overheads (as percentage of total query execution time).

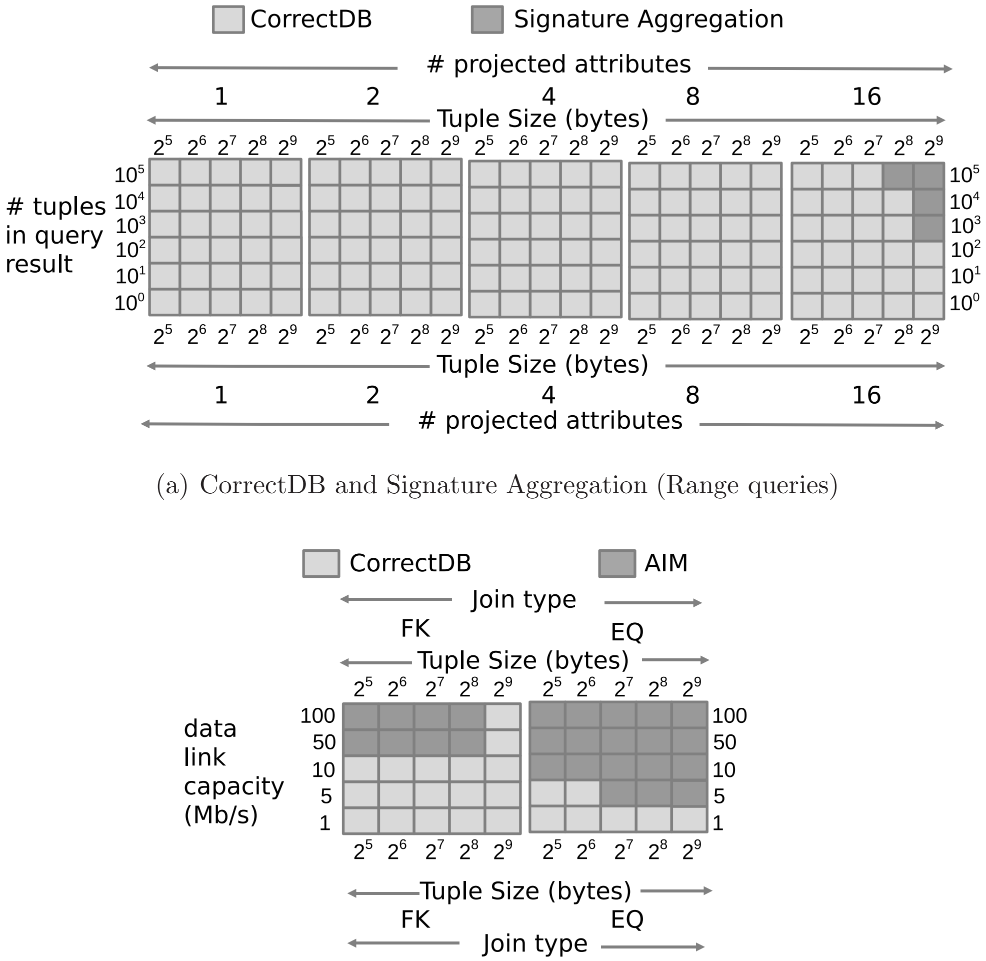
```
AND R.key < 'UB'
```

For AIM, the VO size is not constant but varies with the query result size. Since nontrivial VO sizes result in large server-client data transfers, we include the data transfer time as a measurement parameter. As in [269], we evaluate the performance on both foreign Key (FK) and equi (EQ) joins to keep the comparisons consistent.

Foreign Key (FK) Join

In a FK join, each tuple in relation R matches at least one tuple in the other relation S. FK joins therefore have large query result sizes. Figure 3.5(a) shows the total FK join query execution times for CorrectDB and AIM for various tuple sizes. Note that the data transfer times are not included in Figure 3.5(a). We see that when compared on processing times alone, AIM outperforms CorrectDB for tuple sizes up to 256 bytes.

However, note that the VO size for AIM ranges from 19 MB to 260 MB [269]. Hence, once we consider the data transfer times as well, it is observed that the performance relationship



(a) CorrectDB and Signature Aggregation (Range queries)

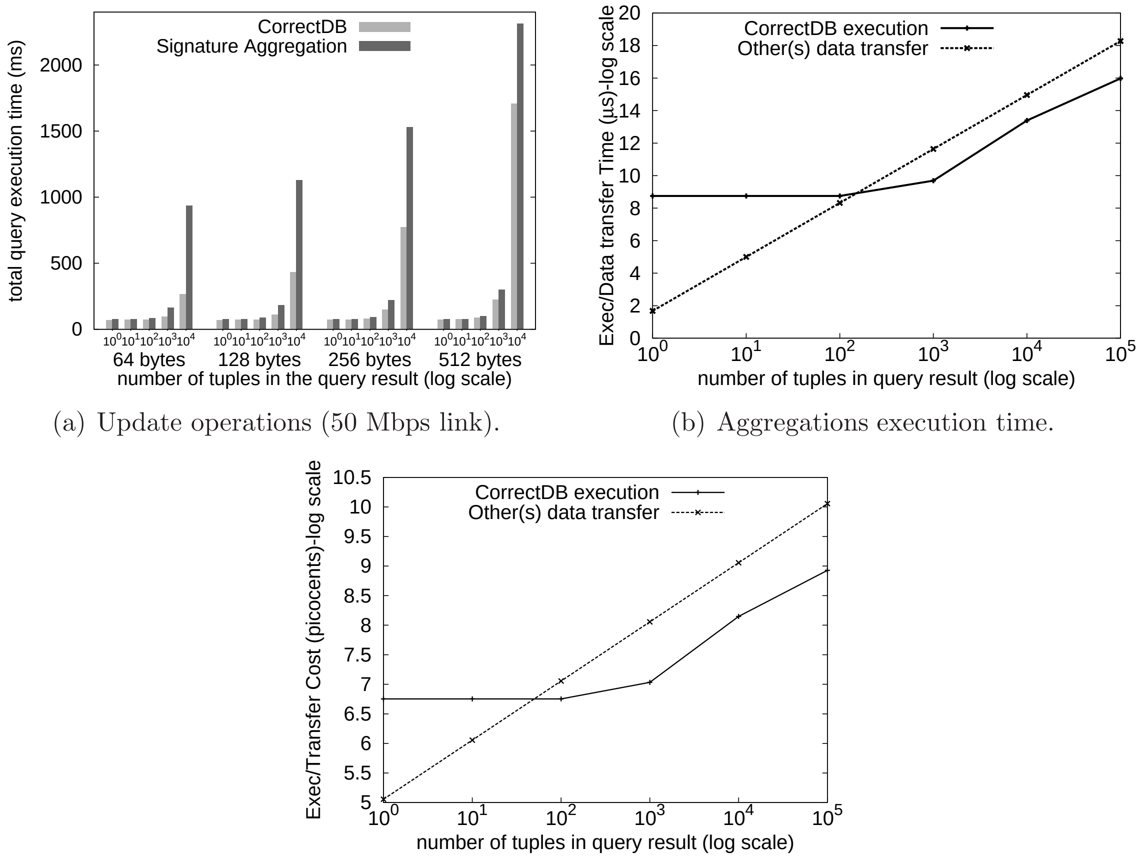
(b) CorrectDB and AIM (Join queries)

Figure 3.8: Performance maps for CorrectDB, Signature Aggregation and AIM. A box shaded with the color of a specific solution indicates that this solution has better performance, that is, lower total query execution time, for the set of parameters corresponding to that box.

inverses. Results including server to client data transfer times are depicted in Figure 3.5(b). As seen from Figure Figure 3.5(b), when data transfer times are also considered, CorrectDB features significantly lower overall execution times by factors up to 5x.

Figure 3.5(b) considers a link capacity of 10 Mbps. In certain settings, such as private clouds, the client-server link capacities may be larger favoring AIM. Hence, in Figure 3.6(a) we recompare for different link capacities. In conclusion, up to link capacities of 50 Mbps CorrectDB still performs better than AIM.

In most commercial settings today, available link capacities for home and businesses range from 1 Mbps to 30 Mbps. Increased capacity is available at increased costs [65]. For commercial cloud services, such as Amazon EC2, the available TCP bandwidth from external clients to cloud has been benchmarked in the 7-27 Mbps range [230]. Hence, for today's link



(a) Update operations (50 Mbps link).

(b) Aggregations execution time.

(c) Aggregations execution cost.

Figure 3.9: CorrectDB aggregate and update operations.

capacities CorrectDB performs better than AIM.

Equi (EQ) Join

Unlike FK join, where each tuple in relation R matches at least one tuple in the other relation S, equi join has a small result set, which we fix at 31000 as in [269]. A small result set reduces the processing times for AIM to construct the VO. Moreover, the VO size is also small. CorrectDB on the other hand, uses the same processing mechanisms for both FK and EQ joins, thereby having similar performance for both join types.

Figure 3.5(c) shows the execution times for EQ join queries for a link capacity of 10 Mbps. Figure 3.6(b) compares the times for varying data link capacities. In summary, for link capacities > 5 Mbps AIM performs similarly or better.

Figure 3.8(b) summarizes which solution ,CorrectDB or AIM, performs better for each combination of the parameters – join type, tuple size, and link capacity.

Conclusions

As seen from Figures 3.8(a) and 3.8(b), using trusted hardware makes CorrectDB the preferred QA solution for a wide range of the parameter space. In addition, use of trusted hardware as in CorrectDB significantly increases the QA functionality. For example, support for arbitrary joins, aggregation queries, access control, and client synchronization (Section 3.5).

3.7.5 Updates

Update operations are evaluated using the range query data sets from Section 3.7.2. Test queries of the form

```
UPDATE key=key+1 FROM R where R.key>'LB' AND R.key<'UB'
```

Updates are a highly favorable scenario for CorrectDB. As discussed in Section 3.5, existing QA solutions do not perform updates server-side. Instead, the data owner obtains the relevant tuples from the server, modifies the tuples locally along with the ADS, and reuploads modified tuples and ADS to the server. Hence, the data transfer overheads in existing QA solutions are significant. As a result, for update operations CorrectDB outperforms in the entire parameter space. Results for update operations are shown in Figure 3.9(a).

3.7.6 Aggregations

To evaluate aggregate operations, we use the data setup of range queries (Section 3.7.2). Test queries are of the form.

```
SELECT SUM(key) FROM R where R.key > 'LB' AND R.key < 'UB'
```

Figure 3.9(b) shows data transfer times in other QA approaches as compared to the time required for computation within the SCPU wherein only the final result is send to the client. The link capacity considered is 10 Mbps. As seen from Figure 3.9(b), total query execution time in CorrectDB is lower for result sizes $> 10^2$. Note that, here we only compare the data transfer time in other solutions and not the total query execution time, which includes the server processing and client verification times. We compare with data transfer times only, to specifically demonstrate the significant overhead of data transfer for aggregations in existing QA solutions. If overall query execution times are added to results of Figure 3.9(b), then CorrectDB outperforms for all result sizes.

We also compare the costs of aggregations in CorrectDB with just the data transfer costs in other approaches. For cost comparison, we use the data transmission costs derived in prior work [65]. To compute the cost of SCPU-based aggregations, we consider the SCPU cycle costs derived in Section 2.4.1. Figure 3.9(c) illustrates the cost comparison.

As seen from Figure 3.9(c), CorrectDB performs significantly better in terms of cost as long as the number of tuples aggregated is above 80. For aggregation set size < 80, a fixed lower bound cost is incurred in transferring a single data page from the server to the SCPU.

3.8 Discussion

3.8.1 Data Confidentiality and QA

In Chapter 2 we showcased TrustedDB [30], an efficient SCPU-based solution for data confidentiality in the presence of a curious but otherwise trusted server. TrustedDB partitions relational data into sensitive and nonsensitive attributes. Sensitive attributes are encrypted. A client query is then split in such a way that sensitive data is decrypted only inside the server-side. Processing on nonsensitive attributes is done by the host server. TrustedDB supports full SQL and leverages the use of fine-grained, attribute-level encryption to offload significant query operations to the server.

TrustedDB does not provide QA guarantees. Although it may be feasible to endow TrustedDB with an additional layer of QA, we chose not to do so for two reasons.

Firstly, since TrustedDB employs fine-grained, attribute-level encryption, it can keep the SCPU-side processing minimal by offloading nonsensitive range, projection and aggregation operations to the server. However, to date, we do not have a single authentication data structure (ADS) that can verify the integrity of all query operations. Hence, to add QA in TrustedDB, a separate ADS is needed for each query operation. Having a separate ADS for each operation increases server-side storage, the SCPU-server data transfer overheads and SCPU-side processing. In short, the overheads of adding QA to TrustedDB far exceed the original cost of data confidentiality, resulting in a solution that is efficient neither for confidentiality nor for QA.

Secondly, for illustration purposes, it was important to clearly outline the cost and performance benefits of trusted hardware over existing QA work without the additional overheads of data confidentiality.

However, CorrectDB allows for a certain degree of confidentiality at minimal cost by employing tuple-level encryption in update and insert operations. All tuple attributes are encrypted except for the search attribute(s) on which a B^+-tree index is built. The search attribute(s) are not encrypted to aid the server in query processing. Also, since projections, aggregations, and the final processing of queries are done inside the SCPU, tuple data is decrypted only within the SCPU.

We measure the overhead of data confidentiality in CorrectDB. Figure 3.7(b) shows the data confidentiality overhead for a set of range queries while Figure 3.7(c) shows the overhead for equi join queries. We observe that the confidentiality overhead decreases with tuple size. For small result set sizes as in the case of EQ join queries, confidentiality is added with very little overhead.

3.8.2 Optimized Solutions

At the expense of functionality, it is possible to design SCPU-based solutions targeted at particular query types. For example, for EQ join queries we can modify the leaf nodes to store the hash of individual tuples instead of entire tuple contents. Tuple-level hashing reduces the size of the leaf nodes and thereby the server-to-SCPU data transfer times. To

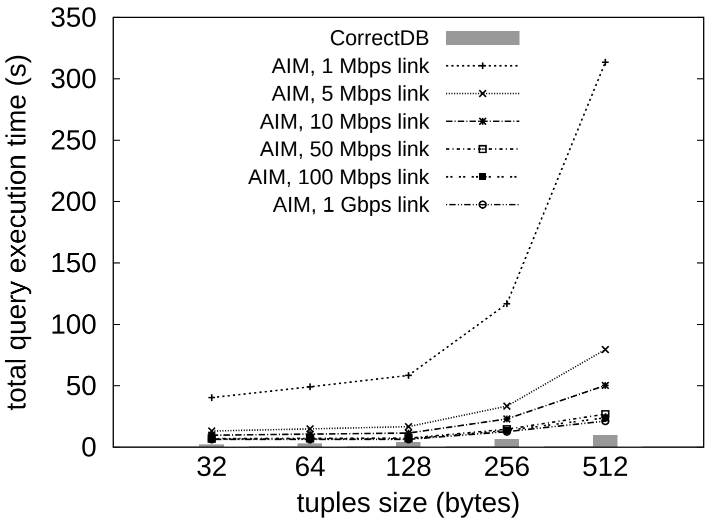


Figure 3.10: Equi Join Comparison for new 4765 SCPU which yields faster results for all cases.

illustrate, for tuple size of 512 bytes from above experiments the transfer times are reduced by up to 95%. However, we chose to opt out of targeted solutions since we posit that the benefits of utilizing trusted hardware are seen in the increased functionality offered (Section 3.5) at better or comparable performance.

New SCPU

Older SCPU technology is now being replaced by new and improved SCPUs, such as the recently announced IBM 4765 [19]. The 4765 features a larger 128 MB RAM, two faster 400MHz CPUs, and a significantly faster PCIe bus for increased 100 MB/s throughputs.

Initial intuitions suggest that the new 4765 will increase the current CorrectDB advantages by a factor of 4-6x over existing QA solutions. We know from the results of Figure 3.7(a) that the server-SCPU transfer latency often dominates query processing time. The higher PCIe throughputs of the 4765 will significantly decrease server-SCPU data transfer times, resulting in a 3.5x reduction of the overall query execution time. Figure 3.10 projects the expected performance of CorrectDB on the new SCPU. As seen, CorrectDB outperforms AIM in the entire parameter space.

3.9 Conclusions

In this chapter, we provide insights into the benefits of trusted hardware for query authentication (QA). The reach the insights, we survey existing QA work and identify cost-based inefficiencies. We show how trusted hardware significantly reduces the data transfer cost incurred by existing QA solutions for join queries. For range queries, use of trusted hardware avoids expensive cryptographic operations significantly saving processing costs.

The benefits of trusted hardware for QA include significant cost savings, better performance, and enhanced QA functionality. We realize the benefits in CorrectDB, a trusted hardware based DBMS that provides QA for a wide range of query types on both read-only and dynamic data sets.

Chapter 4

Concurrent Query Authentication with ConcurDB

4.1 Chapter Overview

4.1.1 Background and Motivation

As we briefly discussed in Chapter 3, *Query Authentication* (QA) complicates update operations since changes made to the outsourced database tuples involve modifications in the related server-side authenticated data structures (ADS). In case of concurrent updates by multiple clients, existing QA solutions have two key limitations. Firstly, current ADS designs limit transaction concurrency because in order to keep the authentication information small, they require updates to a common subset of data items in an ADS resulting in bottlenecks. Further, since current QA solutions do not synchronize clients, they are also subject to *replay attacks* (also referred to as *fork consistency* attack [157]). A replay attack occurs when the server hides updates of one client from other clients by using old authentication data.

Due to the limitations of concurrency and replay attack detection, existing QA solutions do not support update operations [240]; assume fairly static or infrequently updated databases [85, 182]; rely on the data owner to perform all updates on behalf of clients [185, 204, 276]; permit only periodic updates [203]; or suggest batching of update operations [155].

4.1.2 Our Contribution: Concurrent Query Authentication with Updates

We design and evaluate ConcurDB, a concurrent, multi-client QA scheme that eliminates bottlenecks on ADS updates and detects replay attacks efficiently. For concurrency, we decouple transaction execution and verification permitting transactions to execute concurrently and performing verifications in parallel. To detect replay attacks, we design novel communication protocols ensuring that updates by one client are visible to other clients even though clients have no direct knowledge of each other. As compared to existing QA approaches, ConcurDB shows a fourfold increase in performance for update operations.

ConcurDB can be utilized to augment the low-cost CorrectDB solution by adding concurrency. The protocols designed for ConcurDB can easily be ported for execution in the general-purpose trusted hardware devices utilized by CorrectDB, significantly increasing QA concurrency.

4.1.3 Chapter Outline

Section 4.2 describes concurrency-related limitations of existing QA solutions. Section 4.3 discusses the key ideas behind ConcurDB design. ConcurDB architecture is detailed in Section 4.4. Section 4.5 lists experimental results. Finally, Section 4.6 concludes the chapter.

4.2 Concurrency of Existing QA solutions

In Chapter 3, we described the classification existing QA solutions into tree and signature-based. In this section, we focus on the concurrency-related limitations of existing QA work.

4.2.1 Tree-based QA solutions

In tree-based approaches, the ADS is constructed as a tree. As part of range or join query execution, service provider traverses the tree and gathers query results and nodes that form the VO. Using the VO, client reconstructs the traversal path used in query execution to verify correctness and completeness.

The Merkle B-tree (MBT) [155, 202] forms the basis of tree-based approaches. All other tree-based approaches, such as EMBT [155], AIM [269], and XBT [204] are variations of MBT.

MBT is essentially a Merkle hash tree [172] applied to a B$^+$-tree. In a MBT, a hash value is computed for each B$^+$-tree node using a cryptographic hash function, such as SHA-1. The hash for a B$^+$-tree leaf node is computed as follows: The hash of each tuple contained in the leaf node if first computed. Then, the hashes of all the leaf node's tuples are concatenated. A hash of the concatenated value gives the leaf node's hash.

The hash of a non-leaf node is computed as the hash of concatenation of the hashes of the node's children. The root node's hash is signed by the data owner using a secret key. The root signature can be verified by clients using the owner's public key.

During query execution, the provider gathers a minimal set of tuple and B$^+$-tree node hashes that would enable the client to reconstruct the root node's hash. The additional tuple and node hashes constitute the VO. Using the VO and query results, client reconstructs the root hash. The client then compares the computed root hash to the original root hash signed by the owner. If the two hashes match, QA is ensured [202].

To illustrate, Figure 4.1 shows a MBT for a set of tuples along with the VO for a sample range query.

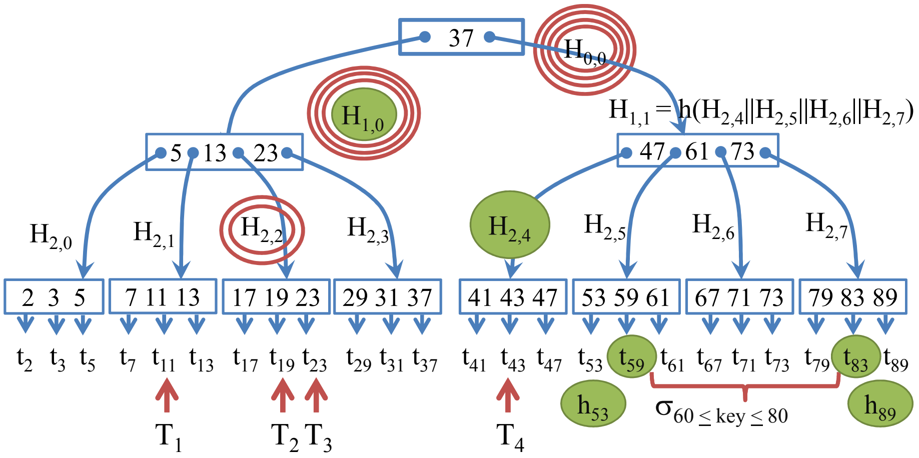

37
$H_{0,0}$
$H_{1,0}$
$H_{1,1} = h(H_{2,4}||H_{2,5}||H_{2,6}||H_{2,7})$
5 13 23
47 61 73
$H_{2,0}$
$H_{2,1}$
$H_{2,2}$
$H_{2,3}$
$H_{2,4}$
$H_{2,5}$
$H_{2,6}$
$H_{2,7}$
2 3 5
7 11 13
17 19 23
29 31 37
41 43 47
53 59 61
67 71 73
79 83 89
t_2 t_3 t_5 t_7 t_{11} t_{13} t_{17} t_{19} t_{23} t_{29} t_{31} t_{37} t_{41} t_{43} t_{47} t_{53} t_{59} t_{61} t_{67} t_{71} t_{73} t_{79} t_{83} t_{89}
h_{53}
$\sigma_{60 \leq key \leq 80}$
h_{89}
T_1
T_2 T_3
T_4

Figure 4.1: An example of a MBT ADS. t_i denotes a tuple with key i. h_i is the hash of tuple t_i. $H_{l,k}$ represents the hash of k^{th} node at level l. Root node is at level 0. Values circled in green constitute the VO for sample select query $\sigma_{61 \leq key \leq 80}$. T_1,T_2,T_3, and T_4 are update transactions. The number of red concentric circles indicate the number of transactions that are in contention for that value.

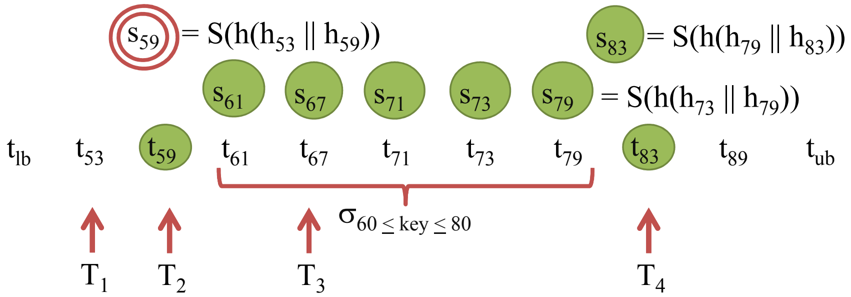

s_{59} = S(h(h_{53} || h_{59}))
s_{83} = S(h(h_{79} || h_{83}))
s_{61} s_{67} s_{71} s_{73} s_{79} = S(h(h_{73} || h_{79}))
t_{lb} t_{53} t_{59} t_{61} t_{67} t_{71} t_{73} t_{79} t_{83} t_{89} t_{ub}
$\sigma_{60 \leq key \leq 80}$
T_1
T_2
T_3
T_4

Figure 4.2: An example of a signature-based ADS. t_i denotes a tuple with key i. h_i is the hash of tuple t_i. t_{lb} and t_{ub} are fake boundary tuples added to the data set. s_i represents the signature for tuple t_i. Values circled in green constitute the VO for sample select query $\sigma_{61 \leq key \leq 80}$. T_1,T_2,T_3, and T_4 are update transactions. The number of red concentric circles indicate the number of transactions that are in contention for that value.

Concurrency

For read-only transactions, it is trivial to observe that tree-based approaches have good concurrency. However, for update operations, concurrency is limited due to the common set of hash values modified by all update transactions. An update of any tuple in the dataset requires recomputation of node hashes up to the root. Thus, in a multi-client scenario, each client update locks and modifies multiple node hashes significantly limiting transaction concurrency. Figure 4.1 shows an MBT example wherein transactions update independent tuples (from distinct leaf nodes) and yet have contention on node hashes.

We note that other tree-based approaches are derived from the MBT approach. Hence, they too have a common set of hash values modified on each update limiting concurrency. Concurrency limitations of tree-based approaches, such as Embedded-Merkle B-tree (EMBT) [155] have been identified and experimentally demonstrated in prior work [203].

Replay Attacks

In tree-based approaches, there are two options to store the root hash. The first option is to distribute the root hash to clients when the database and ADS are initially uploaded to the provider. Although this approach precludes server-side replay attacks, it necessitates synchronization between clients to communicate updated root hash values.

The second option is to store the signed root hash with the provider. On updates, a new root hash is computed and resigned using a secret key not known to the provider. However, storing the signed root hash with the provider enables replay attacks. Using old hash values, the provider can transfer stale results to clients.

To avoid replay attacks, most QA solutions limit updates to the data owner [185,204,276]. Jain et al. [141] designed a QA solution that permits updates by clients. However, all transactions are verified by the data owner. Moreover, Jain et al. use a MBT ADS limiting server-side concurrency due to contention for hash updates.

4.2.2 Signature-based QA solutions

Figure 4.2 illustrates a signature-based ADS approach [185]. Here, the ADS consists of a set of signatures forming a chain. For each tuple, a signature is computed on the concatenation of the tuple's hash along with the hash of the immediate predecessor tuple. For example, the signature for a tuple t_i is computed as $S(h(t_i)||h(t_{i-1}))$, where $S(\cdot)$ denotes a signature operation using the data owner's secret key, $h(\cdot)$ represents a cryptographic hash, and t_{i-1} is the predecessor of t_i when sorted on the search attribute. By including the predecessor in the signature, a chain of all tuples is formed ordered on the search attribute. The VO then consists of the signature of all tuples in the query result and the signature of two boundary tuples not in the result. Since each tuple is linked to its predecessor in an unforgeable manner, the client can verify that no tuple is either illicitly inserted or omitted from the query result.

Variations of the above signature-based scheme can be constructed by including both predecessor and successor tuples in the signature chain or by computing signatures using multiple search attributes [200].

Concurrency

Compared to tree-based approaches, signature-based solutions exhibit good concurrency for updates. In signature-based approaches, an update of a tuple requires updates to the tuple's and the neighboring tuples' signatures. Hence, two updates are in lock contention only if they update same or adjacent tuples as illustrated in Figure 4.2.

Replay Attacks

In signature-based QA, the ADS consists of as many signatures as the number of database tuples. Hence, replay attack detection in signature-based QA is challenging.

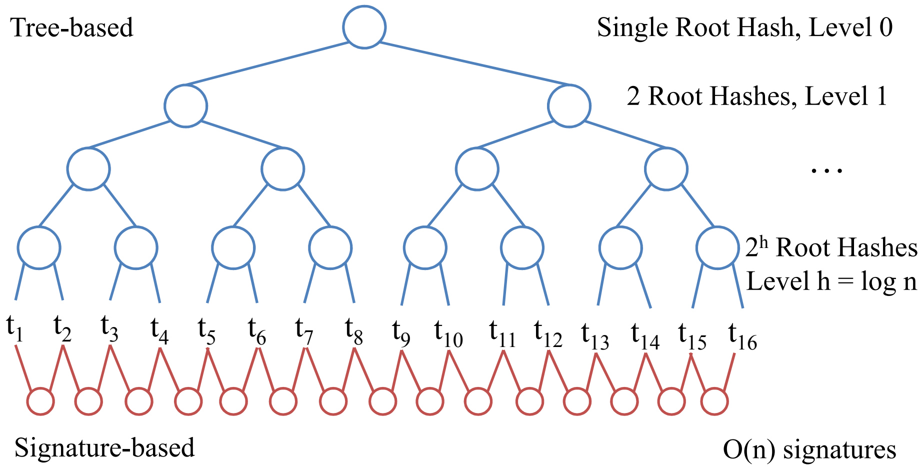


Figure 4.3: Relationship between tree and signature-based ADS. Number of tuples, $n = 16$.

Replay attack detection in signature-based approaches is challenging even if updates are limited to the data owner. For any update by the owner, all clients need to be notified of signatures that are no longer valid. If the number of updated tuples and clients is large, the problem of notifying clients becomes acute.

Pang et al. [203] address the problem of client notification to a certain extent by allowing the data owner to periodically publish a concise bitmap of signatures that were updated in the current period. The solution requires clients to check freshness of each signature received in the VO. However, replay attacks can only be detected for a period p, if the client has stored a bitmap for p. Hence, client-side storage can potentially become a limiting factor.

4.2.3 Relationship between Tree and Signature-based QA

As discussed in Section 4.2.1, tree-based approaches limit concurrency since any two update transactions will be in contention for updating the root hash. A straight-forward way to increase concurrency for tree-based solutions is to maintain multiple root hashes from lower levels. For example, consider the MBT from Figure 4.1. Instead of the single root hash $H_{0,0}$, the hashes $H_{1,0}$, $H_{1,1}$, and $H_{1,2}$ can be considered as root hashes for their respective sub-trees. Here, $H_{1,0}$, $H_{1,1}$, and $H_{1,2}$ will all be signed by the data owner or distributed to clients. The advantage of this approach will be that a transaction that updates tuples in subtree of $H_{1,0}$ will not be in contention with a transaction that updates tuples in subtrees of $H_{1,1}$ and $H_{1,2}$, thereby increasing concurrency. However, a server-side replay attack can now involve $H_{1,0}$, $H_{1,1}$, and $H_{1,2}$.

The approach of multiple root hashes can be extended to the lowest tree level. At the lowest level, a root hash will be maintained for each adjacent tuple pair giving maximum concurrency. This effectively is the signature-based QA approach, where each root hash is a signature chain between two adjacent tuples.

Figure 4.3 highlights the relationship between tree and signature-based QA. In summary, a signature-based approach can be viewed as a modification of tree-based QA for high concurrency. However, due to the large number of signatures (or root hashes) involved,

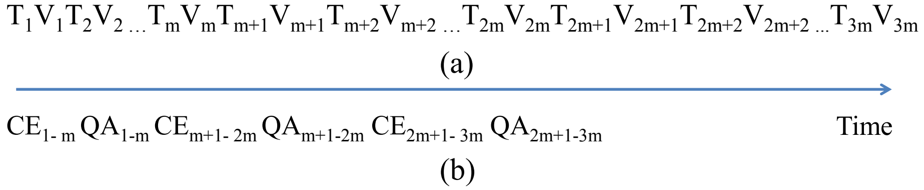

$T_1V_1T_2V_2 \ldots T_mV_mT_{m+1}V_{m+1}T_{m+2}V_{m+2} \ldots T_{2m}V_{2m}T_{2m+1}V_{2m+1}T_{2m+2}V_{2m+2} \ldots T_{3m}V_{3m}$
(a)
$CE_{1\text{-}m}\,QA_{1\text{-}m}\,CE_{m+1\text{-}2m}\,QA_{m+1\text{-}2m}\,CE_{2m+1\text{-}3m}\,QA_{2m+1\text{-}3m}$
Time
(b)

Figure 4.4: QA execution models. (a) Serial execution model in online QA. (b) ConcurDB's offline QA model. T_i denotes transaction i. V_i denotes online verification of T_i. CE_{i-j} represents concurrent execution of transactions $Ti, Ti+1, ...Tj$. QA_{i-j} indicates completion of offline verification for transactions $Ti, Ti+1, ...Tj$.

signature-based approaches make replay attack detection challenging.

4.3 ConcurDB: Key Insights and Ideas

In view of existing QA approaches, designing a particular QA solution in practice requires two design choices to be made. The first choice is between tree and signature-based QA. The second choice that has received less consideration in prior work is between *online* and *offline* QA. In online QA, a client query is verified at execution time, that is, a client verifies query correctness and completeness as soon as it receives query results and VO from the provider. In offline QA, a client query is verified at a later time, possibly after several other queries have executed.

As a result of the two choices, currently, four combinations for QA design are possible – tree-based, online; tree-based, offline; signature-based, online; and signature-based, offline.

We first discuss trade-offs for the choices. Then, in Section 4.3.4, we outline our choices in ConcurDB. In ConcurDB, we choose the best combination in terms efficiency and make further design improvements to realize concurrent QA.

4.3.1 Tree-based vs Signature-based

Trade-offs

In Chapter 3, we compared existing QA solutions based on published experimental results. We found that signature-based approaches perform better for range query processing. Tree-based approaches are more efficient for join processing. Therefore, knowledge of application query types can be a factor in the choice between tree or signature-based QA.

For cloud environments, where data transfer costs dominate [65], signature-based QA may be preferred due to the small, constant VO sizes. Analysis of tree-based QA has shown VO sizes to be $O(\log_b n)$, where n is the total number of database tuples and b is the B$^+$-tree branching factor [155].

For multiple updates by a single client, signature-based approaches have been shown to perform better than tree-based approaches [203]. No efficient solutions are available for updates in a multi-client scenario.

4.3.2 Online vs Offline QA

Most QA solutions [136, 155, 160, 201, 203, 204, 269] have been designed assuming the online QA model.

Jain et al. [141] designed an offline QA solution based on the MBT. During query execution, all relevant information is logged with the data owner, including query results and VO. If a particular transaction is suspected in the future, the logged data can be used for verification. Verification is entirely the data owner's responsibility.

The QA solution by Goodrich et al. [118] performs most query verification operations online except for replay attack detection, which is done offline using cryptographic accumulators. Both solutions [118, 141] use tree-based ADS and hence limit concurrency due to contention for hash updates. Goodrich et al. leave the extension of their protocols to a multi-client setting as an open problem.

Memory Checking and QA

The ability to detect replay attacks is necessary to support update operations in QA. Replay attacks are not just a concern for QA. Any application that updates data stored with an untrusted party is subject to replay attacks. Hence, general solutions for replay attack detection may also be applicable to QA.

The problem of replay attacks has been extensively studied under *memory checking* [41]. In memory checking, a client performs store (write) and retrieve (read) operations on an untrusted memory. The goal of memory checking is to ensure that a value retrieved by the user from a particular memory location was the latest value written to that location. Memory checking thus detects replay attacks at a level of simple store (write) and retrieve (read) operations.

We observe that since database operations are a sequence of reads and writes on tuples [150], memory checking schemes are applicable to replay attack detection in QA. To illustrate the connection between memory checking and QA, consider the MBT, which is the basis for tree-based QA solutions. The MBT in turn is based on the Merkle hash tree (MHT) [202], which is an online memory checking scheme [41].

Memory checking can be online or offline [41]. In online memory checking, each store and retrieve operation is immediately verified. In offline memory checking, verification is deferred until a number of store and retrieve operations have been performed.

We describe memory checking in detail in Section 4.4.1. In this section, we outline the theoretical bounds proved for memory checking that apply to all replay attack detection schemes including ones designed for QA.

Trade-offs

Dwork et al. [88] show that the complexity of online memory checking is $\Omega(\log n/\log\log n)$, where n is the total number of data items subject to replay attacks. This result is significant for signature-based QA since the number of signatures for potential replay attacks is large, as many as the number of database tuples. At first glance, it may seem that tree-based approaches are not limited by the theoretical bound since replay attacks involve only the root hash. However, this is not the case. In order to limit replay attacks to a single root hash, tree-based approaches by design have a built-in processing and VO size complexity of $O(b \cdot \log_b n)$[1] [155]. Thus the lower bounds by Dwork et al. are inherently factored into the ADS for tree-based approaches.

In conclusion, irrespective of tree or signature-based ADS*, the per query complexity of online QA cannot be better than* $\Omega(\log n / \log\log n)$*, where* n *is the number of database tuples.*

Further, for a multi-client scenario, online QA results in a serial dependency between client queries. In online QA, the results of a query are verified as soon the query is executed. As a result, in tree-based QA, if one client updates the root hash as part of verification, a subsequent client cannot execute its update query unless it has access to the updated root hash from the previous client. Therefore, in addition to lock contention over a common set of hash values, the single root hash results in a serialized dependency between client transactions. Figure 4.4(a) illustrates the serialized, online transaction execution and verification model.

Unlike online memory checking, offline memory checking schemes have $O(1)$ amortized complexity per operation [41]. Thus, offline memory checking is a promising alternative for replay attack detection in QA. However, to achieve amortized constant complexity, offline memory checking detects whether a replay attack has occurred in a batch of operations. The precise operation (or query in case of QA) that was subject to a replay attack is not identified.

4.3.3 Analysis Summary

Based on the analysis from Sections 4.2.1 and 4.3.2, we find that tree-based and online QA are not suitable for high concurrency. Tree-based ADS have higher processing and data transfer latencies; and limit concurrency due to contention for hash updates. Online QA introduces a serial dependency between client transactions and imposes a theoretical lower bound of $\Omega(\log n / \log\log n)$ for replay attack detection. As a result, we rule out tree-based and online designs for concurrent QA with updates.

Signature-based and offline QA exhibit greater potential for concurrency. However, signature-based solutions are more vulnerable to replay attacks and use expensive crypto operations. Offline QA detects replay attacks only for query batches.

4.3.4 ConcurDB: Signature-based and Offline

In ConcurDB, we take advantage of signature-based design for concurrency but overcome the associated limitations via the following:

- We do not use separate mechanisms to verify correctness and completeness; and to check for replay attacks. Instead, we design a single memory checking-based, offline

[1] b is the B-tree branching factor.

QA scheme that verifies both correctness and completeness of SQL query results; and detects replay attacks.

- Although ConcurDB design is based on signature-based QA, we do not use expensive crypto operations, such as RSA signatures. Instead, we use the properties of memory checking to guarantee unforgeability. Thus, we achieve same security properties as signature-based QA but avoid the use of expensive crypto operations (Section 4.4.2).
- For concurrency, our offline QA scheme decouples transaction execution and verification. Thus, when one transaction commits, a client does not wait to execute a subsequent transaction. QA operations for a transaction are initiated in a parallel client thread. Thus, transaction execution is not stalled by QA as is the case with online QA (Section 4.4.3). Figure 4.4(b) illustrates the ConcurDB transaction execution model.
- We extend offline QA to enable concurrent updates by multiple clients (Section 4.4.5). The extension involves new protocols that allow clients to securely exchange small, constant-sized authentication data in a fixed order while using the untrusted provider as a communication hub. Our protocols ensure that the provider cannot skip transaction execution for any client, cannot reuse old authentication data (for replay attacks), and cannot tamper client messages.
- We provide extensions to our offline QA scheme that enable precise identification of transactions for which QA requirements were violated (Section 4.4.6). Once our offline QA mechanism detects QA infringement for a batch of transactions, we permit a switch over to online QA. The online QA mode repeats verification for offline QA operations. By design, online QA identifies the faulty operation immediately on execution and hence the faulty transaction. For providers that malfunction or cheat occasionally the switch over condition ensures lower offline QA costs for most of the time. The higher price for online QA applies only when an occasional fault occurs. Moreover, even the online QA mode does not stall subsequent transaction executions and offline QA.

Experimental results (Section 4.5) validate the high concurrency in ConcurDB as compared to tree-based solutions showing a fourfold increase in performance for updates.

4.4 Architecture

We detail ConcurDB architecture as follows: Firstly, we explain offline memory checking as it was originally introduced for a single client scenario (Section 4.4.1). Then, we model QA as an offline memory checking problem and introduce a basic memory checking-based QA solution (Section 4.4.2). Next, we extend the basic QA solution to support QA for database transactions involving multiple SQL queries, such as range, updates, inserts, deletes, projections, and joins (Sections 4.4.3 - 4.4.4). Then, we present a novel protocol that extends offline QA to multiple clients with concurrency (Section 4.4.5). Finally, we present extensions to our offline scheme to precisely detect the transactions (if any) that violate QA (Section 4.4.6).

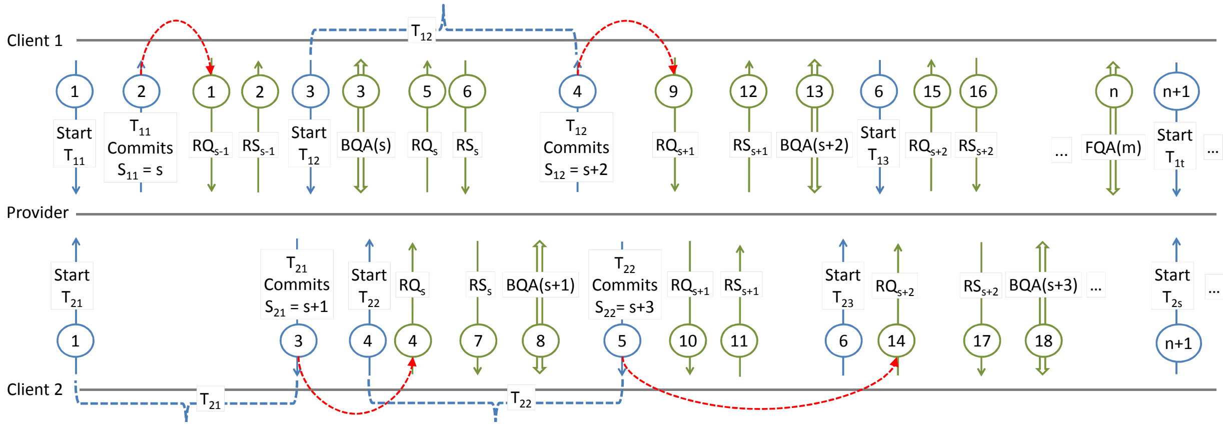


Figure 4.5: Sample execution of ConcurDB's offline QA mechanism for two concurrent clients. Blue steps represent transaction execution. Green steps represent offline QA operations. Steps with the same sequence number and color, or with different colors occur in parallel except for the dependency indicated by red arrows. Dotted blue lines indicate the interval during which queries of transaction T_{ij} execute.

4.4.1 Offline Memory Checking [41]

Model

Memory checking involves three entities: an untrusted memory $\mathcal{M}$, a trusted user $\mathcal{U}$, and a trusted checker $\mathcal{C}$. $\mathcal{M}$ consists of n registers. Each register has a b-bit unique address and can store a b-bit value. $\mathcal{C}$ is assumed to have a small amount of trusted memory of size $O(\log n)$.

User $\mathcal{U}$ requests two types of operations – read and write. Given a register address a, read returns the b-bit value stored at a. Given a register address a and a b-bit value v, write stores v at address a.

User $\mathcal{U}$ sends a sequence of read and write operations to $\mathcal{C}$. $\mathcal{C}$ translates a user operation into multiple memory read and write operations. $\mathcal{C}$'s job is to verify that $\mathcal{M}$ operates correctly. $\mathcal{M}$ is said to operate correctly, if the value read by user from any register a is the latest value that was written to a [41].

The offline checker by Blum et al. stores a timestamp of size $O(\log n)$ with each register in $\mathcal{M}$. A memory read operation thus accepts an address a and returns a tuple (v, t), where v and t are the value and timestamp stored at a, respectively. A memory write operation accepts a triple (a, v, t) and stores the value v and timestamp t at a.

Offline Checker

Checker $\mathcal{C}$ stores two hash values $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ in its memory. Here, $\mathcal{R}$ is the set of all triples (a, v, t) read from $\mathcal{M}$ and $\mathcal{W}$ is the set of all triples (a, v, t) specified to write operations. Initially, $\mathcal{R} = \mathcal{W} = \phi$.

The hash function $\mathcal{H}$ has the following two properties:

T_{ij}	j^{th} transaction of client i.
S_{ij}	Sequence number of transaction T_{ij}.
RQ_s	$RQ_s = E_k(N, s, C_i)$ is the request for authentication set (Section 4.4.5) of transaction with sequence number s (Section 4.4.5).
RS_s	$RS_s = E_k(N, s, C_i, C_j, \mathcal{A})$ is the response to request for authentication set of transaction with sequence number s (Section 4.4.5).
$BQA(s)$	BasicQA step for transaction with sequence number s (Section 4.4.2).
$FQA(s)$	Final QA step for a batch of m transactions (Section 4.4.2).

Table 4.1: Legend for Figure 4.5.

- *Incremental*: Given $\mathcal{H}(\mathcal{R})$ and a triple (a, v, t), $\mathcal{H}(\mathcal{R} \cup (a, v, t))$ can be computed efficiently.
- *Collision Resistant*: If $\mathcal{R} \neq \mathcal{W}$, then $\mathcal{H}(\mathcal{R}) \neq \mathcal{H}(\mathcal{W})$ with high probability.

For a user operation $write(a, v)$, the checker performs the following actions.

- Reads the value v' and time t' stored at a.
- Checks that current time is greater than t'.
- Writes current time t and value v to a.
- Replaces $\mathcal{H}(\mathcal{R})$ with $\mathcal{H}(\mathcal{R} \cup (a, v', t'))$
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (a, v, t))$

For a user operation $read(a)$, the checker performs the following actions.

- Reads the value v' and time t' stored at a.
- Checks that current time is greater than t'.
- Writes current time t and value v' to a.
- Replaces $\mathcal{H}(\mathcal{R})$ with $\mathcal{H}(\mathcal{R} \cup (a, v', t'))$
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (a, v', t))$

After a sequence of operations, $\mathcal{C}$ reads all n memory registers and updates $\mathcal{H}(\mathcal{R})$. Collision resistance of hash function $\mathcal{H}$ ensures that if $\mathcal{M}$ operates incorrectly, then $\mathcal{H}(\mathcal{R}) \neq \mathcal{H}(\mathcal{W})$.

Complexity

The checker performs a constant number of memory operations for each user read or write. Thus, the final step wherein the checker reads all memory registers to update $\mathcal{H}(\mathcal{R})$ dominates the runtime cost especially if $\mathcal{M}$ is large. To achieve $O(1)$ amortized cost per user operation, the checker performs the final step only after n user operations have completed. Thus, memory correctness is verified only after a long sequence of n operations.

4.4.2 Memory Checking-based QA

In this section, we model QA as the offline memory checking problem described in Section 4.4.1.

Overview

For each relation, we first sort the tuples on a search attribute. Then, we create links between adjacent tuples. The links are similar to the signatures in signature-based QA (Section 4.2.2). However, since we do not use expensive crypto operations, such as RSA signatures, we use the term links rather than signatures to differentiate from existing signature-based QA solutions.

We create links as a function of tuple attribute values. As a result, the links also guarantee tuple integrity (Section 4.4.4). Moreover, a SQL query can now be considered as a sequence of reads and writes on links. Therefore, memory checking can be applied to links (Section 4.4.2) along with checks for correctness and completeness (Section 4.4.4).

Since links are a function of tuple attributes, updates and inserts of database tuples correspond to link updates and insertions, respectively. By properties of memory checking, we ensure that, a link value read as a result of a SQL query is the latest value written to the database. Reuse of an old link value by the provider or fake tuples in the query result cannot go undetected.

Initialization and Hashing Tools

Data Upload and Links Creation

At initialization, the data owner uploads a relation $R = \{t_0, t_1, t_2, ..., t_n, t_{n+1}\}$ to the provider, where each t_i is a tuple, such that $t_i.a < t_{i+1}.a$ for some attribute a. t_0 and t_{n+1} are two fake boundary tuples, such that the values of $t_0.a$ and $t_{n+1}.a$ are never used for any other tuples. Additionally, the data owner uploads a set of triples $\mathcal{L}$ wherein each triple (l_i, v_i, ts_i) represents the following.

- l_i is a link between t_i and t_{i-1}. That is, $l_i = h(h(t_{i-1})||h(t_i)))$, where h is a cryptographic hash function, such as SHA.
- v_i takes two values. $v_i = 1$ means link l_i is valid and $v_i = 0$ indicates an invalid link.
- ts_i is the timestamp for link l_i. The timestamp is incremented on each read and write of l_i.

At initial upload time, $v_i = 1$ and $ts_i = 0$ for all triples (l_i, v_i, ts_i), $1 \leq i \leq n+1$.

Thus, the triples (l_i, v_i, ts_i) closely resemble the (a, v, t) triples in memory checking (Section 4.4.1).

Incremental Hashing

Similar to memory checking, the data owner records two hash values, $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$. Initially, $\mathcal{R} = \phi$ and $\mathcal{W} = \mathcal{L}$.

As in memory checking, we require the hash function $\mathcal{H}$ to be incremental. That is given $\mathcal{H}(\mathcal{R})$ and a triple (l, v, t), it should be efficient to compute $\mathcal{H}(\mathcal{R} \cup (l, v, t))$. To ensure collision resistance, we use incremental hash functions based on modular arithmetic [108, 109][2].

Using incremental hashing with a large modulus p [210], owner computes $\mathcal{H}(\mathcal{W})$ as

$$\mathcal{H}(\mathcal{W}) = \textstyle\sum_{i=1}^{n+1} h(l_i||v_i||t_i) \ mod \ p$$

For now, we assume that the owner distributes $\mathcal{H}(\mathcal{R})$, $\mathcal{H}(\mathcal{W})$, and a count $\mathcal{C} = |\mathcal{W}|$ to client. Later, in Section 4.4.5 we describe a new protocol that enables multiple clients to securely exchange the set $\{\mathcal{H}(\mathcal{R}), \mathcal{H}(\mathcal{W}), \mathcal{C}\}$ using the untrusted server as a communication hub.

BasicQA

We first detail our memory checking-based protocol for read and write operations on links. We term this scheme as *BasicQA*. Then, in Section 4.4.3 we describe use of the BasicQA protocol to support QA for entire transactions involving multiple SQL queries.

For a sequence of link reads and writes, our BasicQA protocol ensures correctness and detection of replay attacks. That is, any link returned by provider as a result of a read, is either the original link uploaded by the owner or a new link uploaded by client. If a link is made invalid by client, then the invalid link cannot be returned as a result of any future read without detection.

Provider and Client operations

In BasicQA, the provider supports two operations – $read(l)$ and $write(l, v, ts)$. Read returns the tuple (v, ts), where v is the value and ts is the timestamp, respectively, for link l. $write(l, v, ts)$ stores the triple (l, v, ts) at the provider.

Unlike an user in memory checking, a client in BasicQA functions as both a user and checker. To capture link invalidation (as part of update queries) and addition of new links (as part of insert and update queries) by client, BasicQA supports three operations on the client's side – *cwrite*, *cwrite_new*, and *cread*. *cwrite* invalidates an existing link and *cwrite_new* stores a new valid link with the provider.

On a $cwrite(l)$, a client performs the following.

- Reads the tuple (v, ts) for link l from the provider.

[2] Proved collision-resistant via equivalence to the weighted subset sum problem.

- Checks that v is 1, that is, the link is valid.
- Writes the triple $(l, 0, ts + 1)$ to the provider.
- Replaces $\mathcal{H}(\mathcal{R})$ with $\mathcal{H}(\mathcal{R} \cup (l, v, ts))$.
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (l, 0, ts + 1))$.

For a new link, under $cwrite\_new(l)$, client does the following.

- Writes the triple $(l, 1, 0)$ to the provider.
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (l, 1, 0))$.
- Increments count $\mathcal{C}$ by one.

For $cread(l)$, client actions involve the following.

- Reads the tuple (v, ts) for link l from the provider.
- Checks that v is 1, that is the link is valid.
- Writes the triple $(l, v, ts + 1)$ to the provider.
- Replaces $\mathcal{H}(\mathcal{R})$ with $\mathcal{H}(\mathcal{R} \cup (l, v, ts))$.
- Replaces $\mathcal{H}(\mathcal{W})$ with $\mathcal{H}(\mathcal{W} \cup (l, v, ts + 1))$.

After a sequence of operations, the client requests all triples from the server. For each triple received, the client updates $\mathcal{H}(\mathcal{R})$ and decrements $\mathcal{C}$. When $\mathcal{C} = 0$, client checks whether $\mathcal{H}(\mathcal{R}) = \mathcal{H}(\mathcal{W})$. The collision-resistant property of hash function $\mathcal{H}$ ensures that if the provider operates incorrectly, then $\mathcal{H}(\mathcal{R}) \neq \mathcal{H}(\mathcal{W})$ with high probability. As in memory checking (Section 4.4.1), longer operation sequences, result in lower amortized per-operation verification cost.

4.4.3 QA for SQL Queries : An Overview

We use the BasicQA protocol for transaction-level QA as follows. For each query Q in the transaction, client submits Q to the provider and receives some results as a response. The client then checks the results for partial correctness (not completeness). For example, client checks whether all result tuples satisfy the query predicates. Once partial correctness is verified, using the query results, client constructs and records the hashes for all tuples that were read or written as part of query execution. At transaction commit time, using the recorded tuple hashes, client initiates the BasicQA protocol. Section 4.4.4 details the derivation of BasicQA operation sequences from query results for SQL queries.

Note that in ConcurDB, a client does not wait to submit subsequent transactions to the provider. Execution of BasicQA operations for a transaction is initiated in a parallel client thread. Thus, transaction execution is not stalled by QA as is the case with online QA (Figure 4.4(a)). In a multi-client scenario, each client performs the BasicQA operations in a parallel thread and continues to execute new transactions. Figure 4.4(b) depicts the ConcurDB transaction execution model.

A client is required to store the tuple hashes only temporarily. As soon as the BasicQA operations are performed by the parallel thread for a particular transaction, client discards the tuple hashes recorded for that transaction. In practice, since the BasicQA operations involve far less computations than server-side transaction execution, the BasicQA operations for a transaction are expected to complete before a subsequent transaction commits.

Further, initiating the BasicQA protocol at transaction commit enables two optimizations. The first optimization is elimination of redundant operations. For example, if the same tuple was read twice in a transaction, with no updates in between, then we only perform the corresponding BasicQA operation once for the tuple link. The second optimization is reduction in client-server communication. All BasicQA operations for one transaction are batched and performed via a single round trip message to the provider.

When verification for completeness and replay attack detection is desired, a client or the data owner initiates a special transaction that executes the final memory checking step wherein all triples are downloaded from the provider. The final step verifies completeness and checks replay attacks for all committed transactions.

The timing of final QA step gives a tradeoff for overhead of QA. The longer the final step is delayed, lower is the per transaction QA cost. In real-world deployments, the final step can be initiated at times of low transaction load, such as nightly. For amortized constant QA cost per transaction, the final step must be performed after n/r transactions have committed. Here, n is the number of tuples initially uploaded by the owner; and r is the average number of tuples read and written per transaction.

Figure 4.5 illustrates ConcurDB's offline QA mechanism.

4.4.4 Mapping SQL Query Results to BasicQA Operations

Now that we have outlined the QA process for database transactions, we detail the derivation of BasicQA operation sequences from query results for various SQL query types.

Range (Select) Queries

Consider a range query for all tuples with keys in the range $[L, U]$, $L \leq U$. Let $\mathcal{R} = \{t_0, t_1, t_2, ..., t_r, t_{r+1}\}$ denote the set of tuples in the query result. t_0 and t_{r+1} are two boundary tuples included in the query result to ensure completeness.

For completeness, t_0 must be the immediate predecessor of t_1 and t_{r+1} must be the immediate successor of tuple t_r. That is, t_0 and t_{r+1} are required to satisfy the following conditions:

- $t_0.key < L$ and $\nexists\ t_i$, such that $t_0.key < t_i.key < L$.
- $t_{r+1}.key > U$ and $\nexists\ t_i$, such that $t_{r+1}.key > t_i.key > U$.

At query execution, client checks the following for correctness of result set $\mathcal{R}$:

- $t_0.key < L$.

- $t_{r+1}.key > U$.
- $L \leq t_i.key \leq U$, where $1 \leq i \leq r$.

Additionally, client records the hashes of all tuples in $\mathcal{R}$. In the parallel QA step, using the stored tuple hashes, client executes the following sequence of BasicQA operations.

- $cread(l)$, where $l = h(h(t_{i-1})||h(t_i))$ and $1 \leq i \leq r+1$.

In order to violate correctness or completeness the provider has three choices. The first choice is to introduce a fake tuple in the result. The second choice is to omit a valid tuple from the result. The final choice is to resend a tuple that was previously updated or deleted by client. First two choices would result in a BasicQA operation on a link that was neither written by the owner at database upload time nor written by the client using the *cwrite_new* operation. The third choice would result in a BasicQA operation on a link that was previously made invalid by client using the *cwrite* operation. As a result, in the final QA step, for all provider choices, $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ will not be equal. Therefore, QA infringements are guaranteed to be detected by ConcurDB's offline memory checking-based QA.

Update Queries

Consider an update query that modifies all tuples with keys in the range $[L, U]$, $L \leq U$. The client first executes a select query for all tuples in the range $[L, U]$, $L \leq U$. Let $\mathcal{R} = \{t_0, t_1, t_2, ..., t_r, t_{r+1}\}$ denote the set of tuples in the select query result. t_0 and t_{r+1} are two boundary tuples included for completeness as in the case of range queries (Section 4.4.4). Similar to the case of range queries, client performs checks for correctness on set $\mathcal{R}$.

The client then locally modifies the tuples to create the set $\mathcal{R}' = \{t_0, t'_1, t'_2, ..., t'_r, t_{r+1}\}$, where t'_i is the updated version of tuple t_i. Boundary tuples are not updated since they fall outside the query range.

After local modifications, client records the hashes of all tuples in sets $\mathcal{R}$ and $\mathcal{R}'$. Finally, client issues the update query to provider and the provider updates database tuples.

In the parallel QA step, client constructs and executes a sequence of BasicQA operations using the stored tuple hashes as follows.

- $cread(l)$, where $l = h(h(t_{i-1})||h(t_i))$ and $1 \leq i \leq r+1$.
- $cwrite(l)$, where $l = h(h(t_{i-1})||h(t_i))$ and $1 \leq i \leq r+1$.
- $cwrite\_new(l)$, where $l = h(h(t'_{i-1})||h(t'_i))$, $1 \leq i \leq r+1$ and $t'_0 = t_0$.

cwrite operations invalidate old tuple links. *cwrite_new* operations add new links for the updated tuples.

Insert Queries

Consider an insert query to add a new tuple t with key k. The client first executes a select query for the range $[k, k]$. Let $\mathcal{R} = \{t_l, t_u\}$ denote the set of tuples in the select query result. t_l and t_u are required to be the immediate predecessor and successor, respectively, of tuple t.

The client then records $h(t_l)$, $h(t_u)$, and $h(t)$. Finally, client submits insert query to the provider. The provider adds the new tuple to the database.

BasicQA operations for the insert query are the following.

- $cread(l)$, where $l = h(h(t_l)||h(t_u))$.
- $cwrite(l)$, where $l = h(h(t_l)||h(t_u))$.
- $cwrite\_new(l)$, where $l = h(h(l)||h(t))$.
- $cwrite\_new(l)$, where $l = h(h(t)||h(u))$.

The old link between t_l and t_u is invalidated and two new links are added for tuple t. For clarity, we illustrated an insert with a single tuple. Extensions to insert-select queries are straight-forward.

Delete Queries

Delete queries are processed similar to updates queries in that the client first executes a select query to fetch all tuples that would be deleted. The client then verifies correctness, records hashes, and submits the delete query to provider.

If $\mathcal{R} = \{t_0, t_1, t_2, ..., t_r, t_{r+1}\}$ is the select query result, then the BasicQA operation sequence for the delete query with range $[L, U]$ $(L \leq U)$ is the following:

- $cread(l)$, where $l = h(h(t_{i-1})||h(t_i))$ and $1 \leq i \leq r + 1$.
- $cwrite(l)$, where $l = h(h(t_{i-1})||h(t_i))$ and $1 \leq i \leq r + 1$.
- $cwrite\_new(l)$, where $l = h(h(t_0)||h(t_{r+1}))$.

Projections

In Section 4.4.2 we described the tuple link construction from tuple hashes. To support projections, we construct a tuple hash as a Merkle hash tree (MHT) on the tuple's attribute values. The MHT root hash is the tuple's hash.

When a subset of attributes are projected by a select query, for each tuple in the query result, provider returns only the projected attributes along with a set of hashes using which the client constructs the tuple hash. Incorrect projections will result in an invalid tuple hash. An invalid tuple hash will in turn result in an invalid link which will be detected by the BasicQA protocol.

The MHT node hashes are not stored at the provider. The provider computes the hashes as part of query execution. Hence, no extra storage is consumed on the provider's side.

Moreover, since most databases use row-level locking, concurrent transactions are in contention for entire tuples. Adding an MHT on tuple attributes does not add further contention.

Join Queries

We provide two mechanisms for join processing – client-side join and nested join. To illustrate the two mechanisms, consider a join query $\sigma_{R.a=S.b \textit{ and } R.a \in [L_1,U_1] \textit{ and } S.b \in [L_2,U_2]}$, where R and S are two relations; and a and b are search attributes of R and S, respectively.

If selectivity of both attributes is high, joins are processed client-side. The client executes and verifies two select queries $\sigma_{R.a \in [L_1,U_1]}$ and $\sigma_{S.b \in [L_2,U_2]}$. Using results of both queries, client performs the join locally. Verification of both queries is carried out as described in Section 4.4.4.

In the nested join mechanism, the relation with higher selectivity of the search attribute is first selected. In the example query suppose that $R.a$ has higher selectivity. The client first executes the range query $\sigma_{R.a \in [L_1,U_1]}$ performing checks for correctness and gathering tuple hashes as described in Section 4.4.4. For each tuple t in the query result, client executes and verifies a select query $\sigma_{t.a=S.b}$.

In case of low selectivity of search attributes join efficiency in ConcurDB will be lower than tree-based approaches. As discussed in Section 4.3.1, tree-based approaches are more efficient for join processing and thus more suitable for read-intensive OLAP applications. Our focus is OLTP applications with frequent data updates. Hence, we design ConcurDB for efficient multi-client update scenarios.

4.4.5 Multi-Client Offline QA

The security of ConcurDB's offline QA scheme relies on two hash values $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ (Section 4.4.2). Initially, $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ are computed by the data owner at database upload time. In a single client scenario, owner transfers the initial hashes and a count $\mathcal{C}$ to the only client. We collectively refer to the two hashes and count as the *authentication set* $\mathcal{A} = \{\mathcal{H}(\mathcal{R}), \mathcal{H}(\mathcal{W}), \mathcal{C}\}$. As part of the QA step for each transaction, client updates $\mathcal{A}$. The authentication set $\mathcal{A}$ remains with the single client until the final QA step.

In a multi-client scenario, transactions are executed concurrently by many clients. Each client performs the BasicQA operations for its transactions in a parallel thread. Since BasicQA involves updates to the set $\mathcal{A}$, all clients need access to $\mathcal{A}$. However, in order to ensure QA, only one copy of $\mathcal{A}$ can exists[3] and clients need to access the single copy. This is analogous to the root hash in tree-based $\mathcal{ADS}$. However, unlike online tree-based QA transaction execution is not stalled by QA.

Moreover, for consistency, databases operate under a serializable mode. In serializable mode, locking mechanisms ensure that a concurrent transaction schedule is equivalent to some serial schedule of transactions [150]. Therefore, to ensure QA consistency, the order of BasicQA steps must match a serial order of committed transactions.

Serializing the order of BasicQA steps requires the authentication set $\mathcal{A}$ to be exchanged by clients in a fixed serial order without clients having direct knowledge of each other. We designed a new protocol to enable the exchange using the untrusted provider as a communication hub.

Authentication Set Exchange Protocol

For each client transaction, at commit time, the provider assigns a unique incrementing sequence number. In effect, the provider commits to a serial transaction schedule. The

[3] For partitioned databases, one copy can exist for each partition.

BasicQA operations are then performed in the order of assigned sequence numbers. Our protocol design ensures that the provider cannot skip sequence numbers (i.e., cannot skip a transaction execution altogether), cannot reuse old sequence numbers (i.e, cannot perform a replay attack using old authentication set), and cannot tamper client messages.

As part of the exchange protocol, each client performs two essential operations.

- Before commencing the BasicQA operations for a transaction with sequence number s, a client acquires the authentication set (from another client) resulting from BasicQA for transaction with sequence number $s-1$.
- After completing the BasicQA operations for a transaction with sequence number s, a client responds to a request (from another client) for authentication set with sequence s.

Setup

All clients and the data owner share a key K. We denote by $E_K(M)$, the encryption of message M with key K. Additionally, each client stores a sequence number S. Initially, before any transactions are executed, $S = 0$ for all clients. The initial database upload by the owner is considered as a transaction with sequence number 0. Provider assigns sequence numbers to client transactions starting from 1.

Protocol (illustrated in Figure 4.5)

In the parallel QA step, when performing BasicQA operations for transaction with sequence number s, a client uses the following steps to acquire the authentication set for sequence number $s-1$.

- Generates a random nonce N.
- Pushes the message $E_k(N, s-1, C)$ to the provider, where C is the unique client identifier.
- Waits for the response message $E_k(N', s', C', C'', \mathcal{A})$ Here, C'' is the responding client's identifier and $\mathcal{A}$ is the authentication set for sequence s'.
- Decrypts the response message and verifies that $N' = N$, $s' = s-1$, and $C' = C$.

Using $\mathcal{A}$, the client performs BasicQA operations, resulting in the authentication set $\mathcal{A}'$. After BasicQA operations are performed, client C does the following to transfer $\mathcal{A}'$ to a client that has sequence number $s+1$.

- Waits for a request message for current sequence s. The request message is of the form $E_k(N, s, C')$, where C' is the requesting client's identifier.
- Decrypts the request message.
- Ensures that $s > S$, that is, the client has not previously responded for the sequence s.
- Pushes the message $E_k(N, s, C', C, \mathcal{A}')$ to the provider.
- Sets $S \leftarrow s$.

The nonce plays three important roles. Firstly, the nonce associates a request and a response message for a particular sequence number. Secondly, along with the client identifier, the nonce ensures message integrity. Finally, since each client only responds to a sequence number once, the nonce ensures that replay attacks are avoided. Security proof follows by induction from the intial database upload which is assigned sequence number zero.

4.4.6 Detection of Faulty Transactions

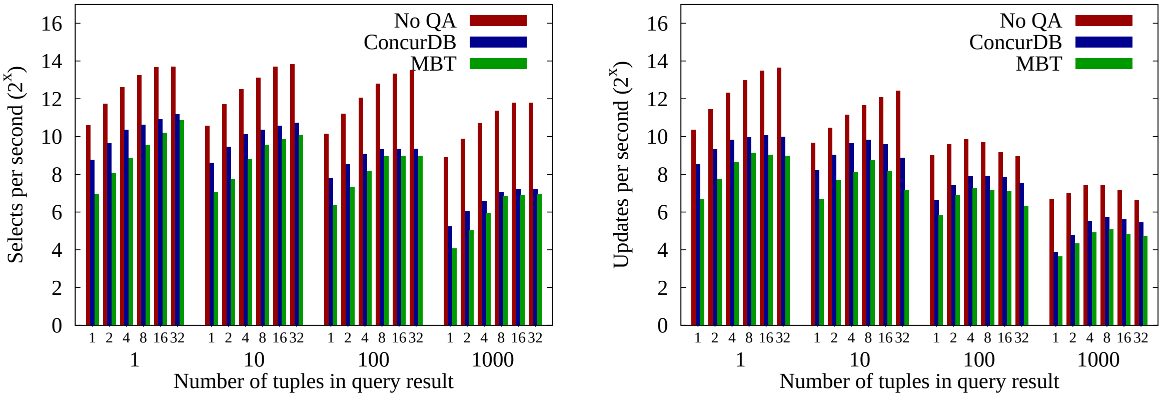
ConcurDB's offline QA mechanism described so far (Sections 4.4.2 - 4.4.5) only detects whether QA infringements for a batch of transactions. Specific transactions for which the provider cheated or malfunctioned are not identified. However, in certain applications, identification of faulty transactions may be necessary for corrections. In this section, we describe the augmentations to ConcurDB for retrospective identification of faulty transactions.

At a high level, faulty transactions are identified as follows. Once the offline QA mechanism detects a QA violation, that is, $\mathcal{H}(\mathcal{R}) \neq \mathcal{H}(\mathcal{W})$ in the final QA step (Section 4.4.2), a switch over to online QA occurs. The online QA mode repeats verification of all BasicQA operations (*cwrite*, *cwrite_new*, and *cread*). By design, online QA identifies the faulty operation immediately on execution.

Note that the switch over to online QA occurs only if a violation is detected by offline QA. The switch over condition is important for efficiency. Recall from Section 4.3.1 that online QA has a complexity of $O(\log n)$ per operation as compared to $O(1)$ complexity for offline QA. For providers that malfunction or cheat occasionally the switch over condition ensures lower offline QA costs for most of the time. The higher price for online QA applies only when an occasional fault occurs. We emphasize that the assumption for occasional violations is reasonable in practice. If a provider malfunctions or cheats often, then switching providers may be a more prudent decision than using a more expensive online QA solution considering that online QA would also limit transaction throughputs.

To enable a switch over to online QA, we make the following augmentations to our offline QA mechanism.

- Along with the hashes $\mathcal{H}(\mathcal{R})$ and $\mathcal{H}(\mathcal{W})$ we add a new hash $\mathbb{H}(\mathcal{O})$. $\mathbb{H}(\mathcal{O})$ records (in sequence) all BasicQA operations performed by clients. Initially, at upload time, sequence $\mathcal{O} = \langle (create\_new, l_i) | 1 \leq i \leq n+1 \rangle$, where $l_1, l_2, ..., l_{n+1}$ are the tuple links (Section 4.4.2). The hash function $\mathbb{H}$ is not incremental. For example, $\mathbb{H}(\mathcal{O} \cup (create\_new, l_i)) = h(\mathbb{H}(\mathcal{O}) || h(create\_new, l_i))$.
- $\mathbb{H}(\mathcal{O})$ is added to the authentication set exchanged by clients (in transaction commit order) using the authentication set exchange protocol (Section 4.4.5).
- Provider records the sequence of all BasicQA operations, that is, the sequence of *cwrite*, *cwrite_new*, and *cread* operations.
- After a QA violation is detected by the offline mechanism, online QA is initiated. In online QA, the owner or a client requests the BasicQA operations from provider. $\mathbb{H}(\mathcal{O})$ verifies that provider returns the exact same sequence of BasicQA operations as were performed in offline mode. For each operation, client and provider engage in an online QA protocol. We suggest the use of MBT-based online implementation by Jain et al. that also supports insert and update operations. *cwrite*, *cwrite_new*, and *cread* will correspond to the MBT *update*, *insert*, and *read* operations, respectively. Online QA



(a) Select queries.

(b) Update queries.

Figure 4.6: Throughput comparison of ConcurDB and MBT with varying number of concurrent clients (1,2,4,8,16,32) and result set size (1,10,100,1000). "No QA" indicates a plain MySQL database with no query authentication.

will identify the faulty operation.

The above augmentations add only a constant number of operations for clients and provider. Hence, the amortized constant cost of offline QA is preserved. Moreover, switching to an online mode does not necessarily stall transactions. Clients can continue to execute new transactions and perform offline QA for the new transactions. Online QA can be done in parallel for the faulty batch of transactions. However, the decision to stall transactions is application-specific since application logic dictates whether a faulty transaction causes inconsistencies in future transactions.

4.5 Experiments

We compare the performance of ConcurDB with MBT and separately evaluate ConcurDB using the TPC-C benchmark.

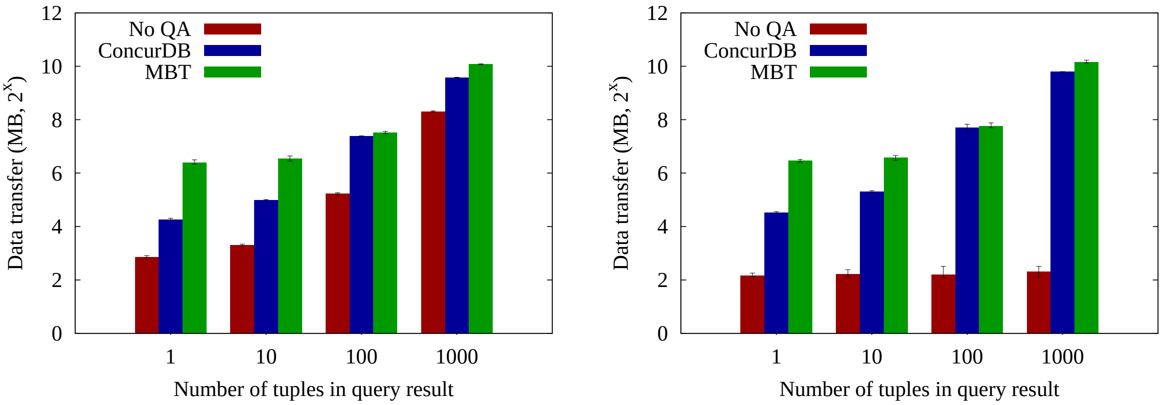
Implementation

ConcurDB implementation is split into client-side and server-side libraries. The client-side Java library transparently performs all QA related operations, including query rewriting; and BasicQA and concurrency protocols.

Server-side library comprises of a set of stored procedures that implement provider-side QA functionality. Using stored procedures instead of modifying the database makes ConcurDB easily portable to run with several DBMS systems. For the current implementation, the database of choice is MySQL version 5.6.17. For the authentication set exchange protocol (Section 4.4.5), we use the RabbitMQ message broker. The total LOC is ≈11K.

Setup

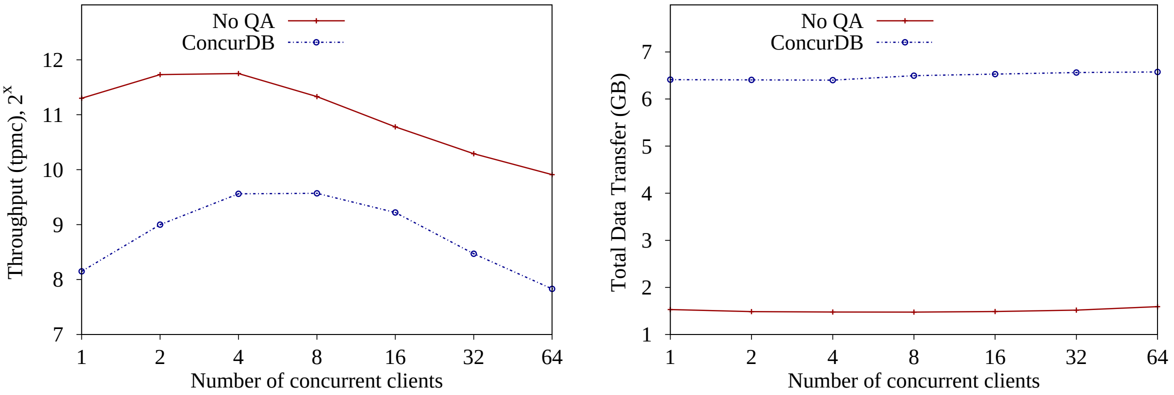
We use a test bed of three identical servers. Each server has 2 Intel Xeon quad-core CPUs at 3.16GHz, 8GB RAM, a Hitachi HDS72302 SCSI drive, and Linux kernel v3.13.0-24. One



(a) Select data transfer.

(b) Update data transfer.

Figure 4.7: Data transfer comparison of ConcurDB and MBT with varying number of concurrent clients (1,2,4,8,16,32) and result set size (1,10,100,1000). "No QA" indicates a plain MySQL database with no query authentication.



(a) TPC-C throughput.

(b) TPC-C Data transfer.

Figure 4.8: TPC-C benchmark comparison of ConcurDB and plain MySQL with no query authentication.

server is dedicated to host the MySQL database and the RabbitMQ message broker, thereby playing the role of service provider. The remaining two servers host clients. We simulate clients using the BenchmarkSQL tool.

4.5.1 Comparison with MBT

We implemented a MBT on top of MySQL following Jain et al.'s implementation [141]. We do not endow the MBT implementation with replay detection. Instead, multiple clients update MBT node hashes and replay attacks are permitted. Permitting replay attacks on MBT enables a comparison for concurrency-related characteristics only. The ConcurDB approach on the other hand, performs the full offline QA protocol. Hence, we note that in real deployments with replay attack detection, performance of MBT will be lower than the performance reported in our results.

We compare ConcurDB and MBT for both select (read-only) and update queries. The data set consists of a relation with 10 million random integer keys. Each client transaction performs a select or update query for a range of tuples. We measure the select and update throughputs with varying number of concurrent clients for a total of 10K transactions.

Figures 4.6(a) and 4.6(b) show the throughputs for selects and updates, respectively. To clearly indicate the overheads of QA, Figures 4.6(a) and 4.6(b) also show the performance of a traditional MySQL database with no QA.

Overall, ConcurDB performs 1.5x - 4x better than the MBT approach with no replay attack detection. Adding replay attack detection to MBT will widen the performance gap further. The results directly stem from the $O(\log n)$ complexity of MBT versus the $O(1)$ amortized execution and data transfer complexity of ConcurDB's offline mechanism.

Figures 4.7(a) and 4.7(b) report the total network-level data transferred for all clients. Since we keep the total number of transactions constant (10K) is each test, data transfer is similar for 1,2,4,8,16, or 32 concurrent clients. Hence, in Figures 4.7(a) and 4.7(b) we report the average data transfer with error bars indicating the minimum and maximum.

For small result set sizes, ConcurDB exhibits up to 4x reduction in data transfer as compared to MBT. For large result set sizes, the data transfer gap between ConcurDB and MBT is lower. This results from the large fanout of MBT. Large result sets include tuples from multiple leaf nodes. Since multiple leaf nodes share common parent nodes at upper levels, the per tuple $\mathcal{VO}$ overhead reduces. ConcurDB's data transfer on the other hand increases linearly with query result size.

In a database with no QA, updates do not return results to clients except for an acknowledgement. Hence, data transfer is nearly constant irrespective of result size size (Figure 4.7(b)). For QA solutions (MBT, ConcurDB, etc.) updates include search operations to identify the correct set of target tuples. Hence, in QA solutions, data transfer for updates is not constant but increases with result set size.

4.5.2 TPC-C Benchmark

We use a TPC-C scale factor of 100 giving a total database size of ≈10 GB. The distributions of TPC-C transactions were set in accordance with the specification. 45% new order, 43% payment, 4% order status, 4% delivery, and 4% stock level. A total of 100K transactions were executed for each test. As per the TPC-C benchmark, we measure throughputs as the number of new order transactions executed per minute (tpmc).

Figure 4.8(a) shows the throughput results in comparison with a plain MySQL database with no QA. We also measure the total data transferred for all clients during the entire test, reported in Figure 4.8(b).

Even for complex TPC-C transactions, ConcurDB maintains a QA overhead similar to the case of simple select and update queries (Figures 4.6(a) and 4.6(b)).

4.6 Conclusions

In this chapter, we introduced ConcurDB, a concurrent query authentication (QA) scheme that supports updates by multiple clients. Identifying an important relationship between query authentication and memory checking we showed that online QA is inherently limited in latency and concurrency. Therefore, we designed ConcurDB as an offline QA mechanism. ConcurDB eliminates bottlenecks on updates; increases transaction concurrency by decoupling transaction execution and verification; and detects replay attacks efficiently. For updates, ConcurDB performs up to 4x better that tree-based QA. Using the TPC-C benchmark, we also demonstrate that ConcurDB can achieve efficient QA for full-fledged OLTP applications.

Chapter 5

History Independence For Regulatory Compliance

5.1 Introduction

5.1.1 Background and Motivation

Retention regulations [83, 191, 252] desire that once data is deleted, no evidence about the past existence of deleted data should be recoverable. Typically, data is deleted using secure deletion [86]. Under secure deletion, data is physically deleted from the storage medium by overwriting. The number of overwrites required depends on the storage medium characteristics. However, evidence of past deletes cannot be eliminated by simply overwriting data as in secure deletion [86]. Even after secure deletion, deleted data can be recovered via current data organization.

Data structures are commonly used to organize data in systems. For a data structure, the organization of data is referred to as the data structure's state. Since the previous existence of deleted data impacts the current data structure state, the current state can be used to derive information about the past existence of deleted data.

We posit that for regulatory compliance, truly irrecoverable deletion can be achieved by utilizing history independent data structures for organizing data (Section 5.2). The current state of a history independent data structure is a function of current data only and not of the sequence of past operations. Hence, if data is deleted in the past, the current state carries no evidence of the delete. History independent data structures are therefore ideal to meet data retention Regulations.

Although we have identified the role of history independence in designing systems compliant with data retention Regulations, history independence has a wider scope. Applications such as incremental signature schemes [183] and e-voting [38, 178, 179, 183] rely on the use of history independent data structures for privacy. In the context of document editing, incremental signature schemes ensure that intermediate edits are not visible in the final document version. In e-voting, use of history independent data structures conceals the vote order protecting voter privacy.

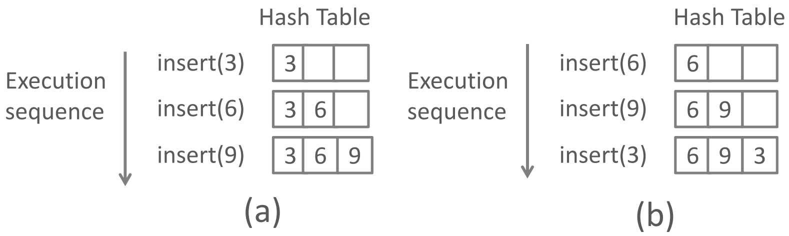


Figure 5.1: A history dependent hash table organizes the same data set differently depending on the sequence of operations (i.e., history). In this example, the hash table uses linear probing [171]. The number of hash table buckets is 3 and the hash function is modulo 3.

5.1.2 Our Contribution: Theoretical Foundations of History Independence

Weak history independence (WHI) and strong history independence (SHI) are the two existing notions of history independence. WHI assumes a rather weak adversary. SHI on the other hand is a very powerful notion of history independence, secure even against a computationally unbounded adversary [113]. Further, WHI does not protect against insider adversaries and SHI results in inefficiency [48].

Currently, applications are restricted to using data structures with either WHI or SHI characteristics. However, applications that do not fit into either WHI or SHI do exist. For example, a journaling system that reveals no historical information other than the last k operations[1]. Hence, there is a necessity for new notions of history independence targeted towards specific application scenarios.

To facilitate the design of new history independence notions, we introduce the Δ history independence (ΔHI) framework. ΔHI centers around a generic game-based definition of history independence and is malleable enough to accommodate WHI, SHI, and a broad spectrum of new history independence notions (Section 5.5.1). In addition, ΔHI helps to quantify the history revealed by existing data structures most of which have been designed without history independence in mind.

In the process of formalizing ΔHI, we explore the concepts of abstract data types, data structures, machine models, memory representations and history independence itself.

To summarize, in this chapter we understand history independence from a theoretical perspective. Then, in Chapters 6 and 7 we use the theoretical results to architect history independent file systems.

5.2 A Quick Informal Look at History Independence

History independence is concerned with the historical information preserved within data structure states. The preserved history may be illicitly used by adversaries to violate regulatory compliance. For example, an adversary may breach data retention laws by recovering

[1] We give additional examples in Section 5.5.1.

deleted data. Therefore, to understand history independence, we need to specify what we mean by state, what we mean by history, and what an adversary can do.

What is state?
A data structure's state is an organization of data on a physical medium such as memory or disk.

What is history?
History is the sequence of operations that led to the current data structure state.

What is the threat?
For many existing data structures, the current state is a function of both data and history [113]. Hence, by analyzing the current state an adversary can derive the state's history. Depending on the application the historical information includes the following:

- Evidence of past existence of delete data [32].
- The order in which votes were cast in a voting application [38, 183].
- The intermediate versions of a published document [183].

To illustrate, consider the sample hash table data structure of Figure 5.1. The sample hash table organizes the same data set differently depending on the sequence of operations used. Hence, an adversary that looks at the system memory can potentially detect which operation sequence was used to get to the current hash table state.

What is history independence?
History independence is a characteristic of a data structure. A data structure is said to be history independent if from the adversary's point of view, the current data structure state is a function of data only and not of history. Thus, the current state of a history independent data structure reveals no information to the adversary about its history other than what is inherently visible from the data itself.

We emphasize that history independence is concerned with historical information that is revealed from data organization and not from the data. In our hash table example of Figure 5.1, the fact that values {3,6,9} were inserted in the past is evident no matter how the data is organized. The data organization reveals the order of insertion.

Are there different kinds of history independence?
Naor et al. [184] introduced two notions of history independence – weak history independence (WHI) and strong history independence (SHI).

WHI and SHI differ in the number of data structure states an adversary is permitted to observe. Under WHI, an adversary is permitted to observe only the current data structure state. For example, as in case of a stolen laptop. Under SHI, an adversary is permitted several observations of data structure states throughout a sequence of operations. For example, as in case of an insider adversary who can obtain a periodic memory dump. For SHI, the adversary should be unable to identify which sequence of operations was applied between any two adjacent observations.

How does history independence achieve regulatory compliance?
The current state of a history independent data structure is a function of current data only. Data that was deleted in the past leaves no effect on the current state that an adversary can detect. History independent data structures are therefore ideal to organize data in compliance with data retention Regulations [83, 191, 252] that require truly irrecoverable data erasure.

5.3 Preliminaries

Formalizing history independence requires an understanding of data structures. A data structure itself can be viewed as an implementation of an abstract data type (ADT) on a machine model [113]. An abstract data type (ADT) is a specification of operations for data organization while a machine model represents a physical computing machine.

In the following, we will explore the aspects of ADTs, data structures, machine models, and memory representations that are relevant to history independence. Then, in Section 5.4 we formalize history independence.

5.3.1 Abstract Data Type (ADT)

The specification of data organization techniques is often done via abstract data types. The key characteristic of an ADT is that it specifies operations independently of any specific implementation. We build on the the ADT concept proposed by Golovin et al. [113], wherein an ADT is considered as a set of states together with a set of operations. Each operation maps the current state to a new state.

Definition 3. Abstract Data Type (ADT)
An ADT $\mathcal{A}$ is a pentuple $(\mathcal{S}, s_\phi, \mathcal{O}, \Gamma, \Psi)$, where $\mathcal{S}$ is a set of states; $s_\phi \in \mathcal{S}$ is the initial state; $\mathcal{O}$ is a set of operations; Γ is a set of inputs; Ψ is a set of outputs; and each operation $o \in \mathcal{O}$ is a function [2] *$o : \mathcal{S} \times \Gamma_o \to \mathcal{S} \times \Psi_o$, where $\Gamma_o \subseteq \Gamma$ and $\Psi_o \subseteq \Psi$.*

The ADT is initialized to state s_ϕ. When an operation $o \in \mathcal{O}$ with input $i \in \Gamma_o$ is applied to an ADT state s_1, the ADT outputs $\tau \in \Psi_o$ and transitions to a state s_2. The transition from state s_1 to s_1 is denoted as $o(s_1, i) \to (s_2, \tau)$.

The necessity of ADTs

History independence requires that from an adversary's point of view, the current data structure state is a function of data only and not of history. In the context of history independence, an ADT models the history revealed by data only. Since we view a data structure as an ADT implementation (Section 5.3.3), the ADT helps to clearly identify what

[2] For brevity, we model each ADT operation with an input and an output. ADT operations may accept no inputs or produce no outputs. Hence, an ADT operation can also be modeled as the following functions: $o : \mathcal{S} \to \mathcal{S}$, $o : \mathcal{S} \to \mathcal{S} \times \Psi_o$, or $o : \mathcal{S} \times \Gamma_o \to \mathcal{S}$.

the data structure is permitted or not permitted to reveal about past operations. Any history revealed by an ADT state can be revealed by the corresponding data structure state. Any history hidden by an ADT state must be hidden by the corresponding data structure state.

We will revisit and formalize the connection between ADT states and data structure states in Section 5.3.5.

ADT as a graph

We can imagine the ADT to be a directed graph $\mathcal{G}$, where each vertex represents an ADT state and each edge is labeled with an ADT operation along with an ADT input and an ADT output. The label for an edge between two vertices represents the operation that causes the transition between the corresponding states. We call the graph $\mathcal{G}$, the state transition graph of the ADT.

Viewing an ADT as a graph will be particularly useful when we take a deeper look into history independence in Section 5.3.4.

5.3.2 Models of Execution

An ADT is only a specification of operations for organizing data. For more practical use, such as for efficiency analysis, concrete implementations of the ADT operations are required. ADT implementations are provided via programs that can be executed on a given machine model. We refer to an ADT's implementation in a given machine model as a data structure (Section 5.3.3).

Several machine models have been proposed [233], such as logic circuits, machines with memory, and combinatorial circuits[3]. We focus on the RAM model of execution since we are concerned with history independent characteristics of complex software applications. Software applications such as databases, and file systems rely on data organization within the storage sub-systems of modern computers. The sub-systems can be accurately modelled using the RAM execution model.

RAM Model of Execution

The RAM model of execution models a traditional serial computer. The model consists of two components, a central processing unit (CPU) and a random access memory (RAM). Both the CPU and RAM are finite state machines (FSM) [233].

The RAM consists of $m = 2^u$ storage locations. Each location is a b-bit word and has a unique $\log_2 m$ bit address associated with it[4]. Two operations are permitted on a storage location in the RAM. First, a load operation to access the b-bit bit word stored at the location. Second, a store operation that copies a given b-bit word to the location. Typically, the b-bit words are copied to or copied from CPU registers.

[3] For a detailed survey of various machine models refer to the work of Savage et al. [233].

[4] This a bounded-memory RAM.

The CPU consists of n b-bit registers and operates on a fetch-and-execute cycle [233]. The CPU has an associated set of instructions that it can perform. CPU instructions are specified in a programming language. A program in a RAM model is a finite sequence of programming language instructions.

A machine model can itself be considered as an ADT [113]. In this case, the set of ADT states is the set of all machine states, and the set of ADT operations is the set of all machine programs. For the RAM model, the set of ADT states, the set of inputs, and the set of outputs are all represented as bit strings.

Definition 4. Bounded RAM Machine Model
A bounded RAM machine model $\mathcal{M}$ with m b-bit memory words and n b-bit CPU registers is a pentuple $(\mathcal{S}, s_\phi, \mathcal{P}, \Gamma, \Psi)$, where $\mathcal{S} = \{0,1\}^{b(m+n)}$ is the set of machine states; $s_\phi \in \mathcal{S}$ is the initial state; $\mathcal{P}$ is the set of all programs of $\mathcal{M}$; $\Gamma = \{0,1\}^*$ is a set of inputs; $\Psi = \{0,1\}^*$ is a set of outputs; and each program $p \in \mathcal{P}$ is a function $p : \mathcal{S} \times \Gamma_p \rightarrow \mathcal{S} \times \Psi_p$, where $\Gamma_p \subseteq \Gamma$ and $\Psi_p \subseteq \Psi$.

$\mathcal{M}$ is initialized to state s_ϕ. If a program $p \in \mathcal{P}$ with input $i \in \Gamma_p$ is executed by the CPU when $\mathcal{M}$ is in state s_1, $\mathcal{M}$ outputs $\tau \in \Psi_p$ and transitions to a state s_2. The transition from state s_1 to s_2 is denoted as $p(s_1, i) \rightarrow (s_2, \tau)$.

5.3.3 Data Structure

In the previous section, we hinted that a data structure is an ADT's implementation in a specific machine model. Now that we have defined both ADT and the RAM machine model we can formalize the data structure.

An implementation for an ADT in a given machine model is obtained as follows.

- A machine representation is chosen for each ADT input and output.
- For each ADT operation a machine program is selected that provides the functionality desired from the ADT operation.
- A unique machine state is selected to represent the initial ADT state.

We encapsulate the above steps in the following data structure definition.

Definition 5. Data Structure
A data structure implementation of an ADT $\mathcal{A}$ in a bounded RAM machine model $\mathcal{M}$ is a quadruple $(\alpha, \beta, \gamma, s_0^{\mathcal{M}})$, where $\mathcal{A} = (\mathcal{S}, s_\phi, \mathcal{O}, \Gamma, \Psi)$ as per definition 3, $\mathcal{M} = (\mathcal{S}^{\mathcal{M}}, s_\phi^{\mathcal{M}}, \mathcal{P}^{\mathcal{M}}, \Gamma^{\mathcal{M}}, \Psi^{\mathcal{M}})$ as per definition 4, $\alpha : \Gamma' \rightarrow \Gamma^{\mathcal{M}}$, $\beta : \Psi' \rightarrow \Psi^{\mathcal{M}}$, $\gamma : \mathcal{O} \rightarrow \mathcal{P}^{\mathcal{M}}$, $s_0^{\mathcal{M}} \in \mathcal{S}^{\mathcal{M}}$, $\Gamma' \subseteq \Gamma$ and $\Psi' \subseteq \Psi$.

α is a mapping from ADT inputs to machine inputs. That is, for any ADT input i, $\alpha(i)$ is the machine representation of the input. Similarly, β is the mapping from ADT outputs to machine outputs. γ is the mapping from ADT operations to machine programs. For an ADT operation o, $\gamma(o)$ is the machine program implementing o. Finally, just as the ADT $\mathcal{A}$ is initialized to a unique state s_ϕ, a unique machine state $s_0^{\mathcal{M}}$ is selected to represent the initial data structure state.

Example: Hash table as an ADT and its data structure

To clarify the concepts of ADT and data structure, we use the example of a hash table. First, we define a hash table ADT. Then, we describe a data structure implementation of the hash table ADT.

Let $\mathrm{H} = (\mathcal{S}, s_\phi, \mathcal{O}, \Gamma, \Psi)$ be a hash table ADT, where

- $\mathcal{S} = 2^{\mathbb{N}\times\mathbb{N}}$ is the set of states[5].
- $s_\phi = \varnothing$ is the initial state.
- $\Gamma = \mathbb{N} \cup (\mathbb{N} \times \mathbb{N})$ is the set of inputs.
- $\Psi = \mathbb{N} \cup \{ERROR, SUCCESS\}$ is the set of outputs.
- The set of operations $\mathcal{O} = \{\text{insert, search, delete}\}$, such that
  - insert : $\mathcal{S} \times \mathbb{N} \times \mathbb{N} \to \mathcal{S} \times \{ERROR, SUCCESS\}$.
  - search : $\mathcal{S} \times \mathbb{N} \to \mathcal{S} \times (\mathbb{N} \cup \{ERROR\})$.
  - delete : $\mathcal{S} \times \mathbb{N} \to \mathcal{S} \times \{ERROR, SUCCESS\}$.

An implementation of the above hash table ADT in the RAM model, that is, a data structure $\mathcal{D} = (\alpha, \beta, \gamma, s_0^{\mathcal{M}})$ can be obtained as follows.

- For all $n \in \mathbb{N}_b$, $\alpha(n) \in \{0,1\}^b$. Here, $\mathbb{N}_b = \{x | x \in \mathbb{N} \textit{ and } x \leq 2^b\}$, b is the machine word length, and $\alpha(n)$ is the bit string representing n. For all $(n_1, n_2) \in \mathbb{N}_b \times \mathbb{N}_b$, $\alpha((n_1, n_2)) = \alpha(n_1)||\alpha(n_2)$.
- For all $n \in \mathbb{N}_b$, $\beta(n) = \alpha(n)$. $SUCCESS$ is represented by $\{0\}^b$, and $ERROR$ is represented by $\{1\}^b$.
- $\gamma : \mathcal{O} \to \mathcal{P}^{\mathcal{M}}$. A machine program is provided for each hash table ADT operation. For example, implementation of the insert, search and delete algorithms of an array-based hash table using linear probing [171].
- The initial machine state $s_0^{\mathcal{M}}$ corresponding to the initial ADT state s_ϕ is obtained by first loading all machine programs implementing the ADT operations into memory and setting the memory locations reserved for the hash table to zero.

Note that the above data structure $\mathcal{D}$ is one possible implementation of the hash table ADT. Several other implementations are possible. In general, the same ADT can have several data structure implementations.

Abstract Data Types And Type Theory: The above hash table ADT definition assumes a hash table over natural numbers. In general, the hash table ADT can be defined over any type, such as real numbers, bit strings, or be composed from other basic types. In

[5] 2^A denotes the powerset of set A.

type theory [53], a type is defined as a set of values that share a common logical property or attribute. It is beyond the scope of this dissertation to formalize the notion of types. Instead, we refer the reader to relevant notes on type theory [53].

Data Structure State

A data structure state is a machine state. The set of all data structure states consists of all machine states that are reachable from the initial data structure state via execution of machine programs implementing the ADT operations.

State Transition Graph For Data Structure

In Section 5.3.1, we introduced the state transition graph for an ADT. Similarly, we can view a data structure in terms of a state transition graph. Graph-based view of a data structure helps to identify the machine states that constitute the set of data structure states, to precisely define the relationship between ADT states and data structure states (Section 5.3.5), and to understand history independence (Section 5.3.4).

A data structure can be considered to be a directed graph $\mathcal{G}$, where each vertex represents a data structure state and each edge is labeled with a machine program implementing an ADT operation along with a machine input and a machine output. The label for an edge between two vertices represents the program that causes the transition between the corresponding states. We call the graph $\mathcal{G}$, the state transition graph of the data structure.

5.3.4 A Semi-Formal Look At History Independence

Equipped with the concepts of ADT (Section 5.3.1), RAM machine model (Section 5.3.2), data structure (Section 5.3.3), and state transition graphs, we can gain a deeper insight into history independence. In Section 5.2, we briefly introduced the two existing history independence notions – weak history independence (WHI) and strong history independence (SHI). Both WHI and SHI are formalized in Section 5.4. In this section, we use the graph-based view of ADT and data structure to understand history independence. Later, in Section 5.4 we formalize history independence.

The nonisomorphism problem

Nonisomorphism between the state transition graph of an ADT and of its data structure implementation breaks SHI. WHI on the other hand can be achieved even when the ADT and data structure state transition graphs are nonisomorphic. First, we look at how nonisomorphism breaks SHI and then we discuss how to achieve WHI in the presence of nonisomorphism.

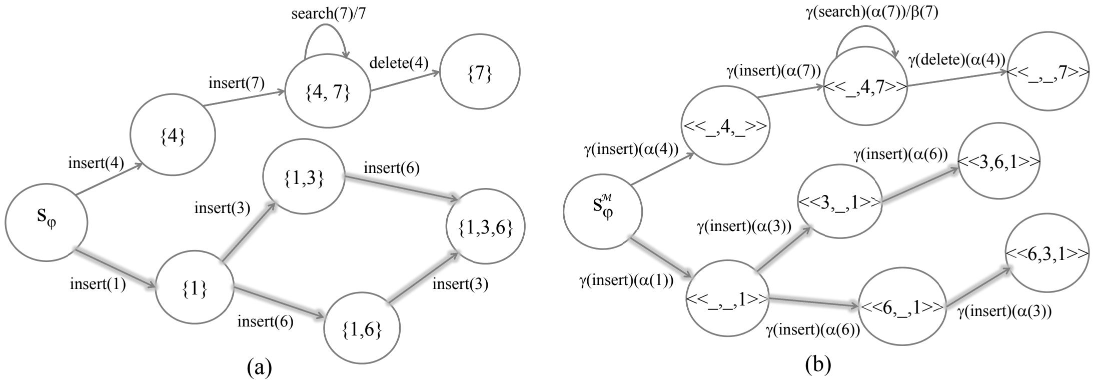


Figure 5.2: Example of nonisomorphism between ADT and data structure state transition graphs. (a) Partial state transition graph for sample hash table ADT. (b) Partial state transition graph for sample array-based hash table data structure implementation using linear probing. Number of hash table buckets is 3 and the hash function is $h(key) = key \% 3$. γ(insert), γ(search) and γ(delete) denote the machine programs implementing the ADT operations insert, search and delete, respectively. $o(i)/t$ denotes that ADT operation o takes input i and produces output t. Similarly, $\gamma(o)(\alpha(i))/\beta(t)$ denotes that program $\gamma(o)$ takes input $\alpha(i)$ and produces output $\beta(t)$. $\alpha(i)$ and $\beta(t)$ are the machine representations of the ADT input i, and ADT output t, respectively. Note that the vertices in figure (b) represent data structure states. In the RAM model, data structure states are bit strings. However, to convey data semantics we denote the hash table array as $<< a_0, a_1, a_2 >>$, where a_0, a_1, and a_2 are elements at buckets 0 , 1 and 2, respectively. Underscore denotes an empty bucket. Highlighted paths are referenced in Table 5.1 and in Section 5.3.4.

Why nonisomorphism breaks SHI?

The need for SHI arises due to nonisomorphism. Nonisomorphism occurs when an ADT state has multiple memory representations[6]. We will precisely define memory representations for ADT states in Section 5.3.5. For now, it suffices to say the following: A memory representation for an ADT state that is reachable from the initial ADT state via a sequence of ADT operations, is the machine state reachable from the initial data structure state via the corresponding program sequence.

To illustrate how nonisomorphism breaks SHI, consider the example graphs from Figure 5.2, example paths from Table 5.1, and an adversary with access to the following: the initial ADT state s_ϕ, the initial data structure state $s_\phi^\mathcal{M}$, the current ADT state $\{1, 3, 6\}$, and the current data structure state. The current data structure state is either $<< 3, 6, 1 >>$ or $<< 6, 3, 1 >>$. Both states $<< 3, 6, 1 >>$ and $<< 6, 3, 1 >>$ are memory representations of the ADT state $\{1, 3, 6\}$.

By looking at the ADT states alone, an adversary cannot determine which sequence of ADT operations, that is path $p_\mathcal{A}$ or path $p'_\mathcal{A}$ was used to arrive at the current ADT state

[6] Many existing data structures have this property and are hence, not history independent. Common examples include linked lists, hash tables, and B-trees. In these data structures, different insertion order of the same set of data elements, that is the same ADT state results in different memory representations.

Path	From Figure
$p_{\mathcal{A}} = s_\phi \rightarrow \{1\} \rightarrow \{1,3\} \rightarrow \{1,3,6\}$	5.2(a)
$p'_{\mathcal{A}} = s_\phi \rightarrow \{1\} \rightarrow \{1,6\} \rightarrow \{1,3,6\}$	5.2(a)
$p_{\mathcal{D}} = s_\phi^{\mathcal{M}} \rightarrow << \_, \_, 1 >> \rightarrow << 3, \_, 1 >> \rightarrow << 3, 6, 1 >>$	5.2(b)
$p'_{\mathcal{D}} = s_\phi^{\mathcal{M}} \rightarrow << \_, \_, 1 >> \rightarrow << 6, \_, 1 >> \rightarrow << 6, 3, 1 >>$	5.2(b)

Table 5.1: Sample paths from ADT and data structure state transition graphs of Figure 5.2.

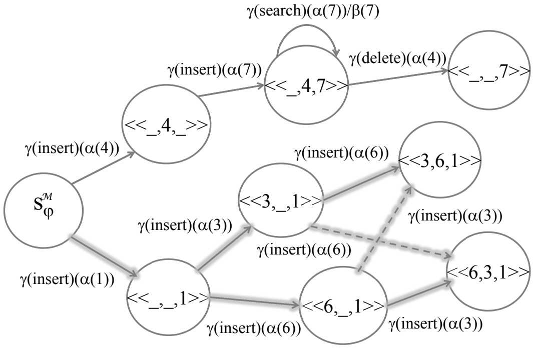


Figure 5.3: Using randomization to achieve history independence. The dotted lines indicate new transitions added to the hash table data structure state transition graph. Amongst all edges with the same starting node and the same label, the choice of edge for state transition is made at random.

$\{1, 3, 6\}$. Hence, the data alone gives the adversary no advantage in guessing which sequence of ADT operations was applied in the past. Now, by looking at the current data structure state, which is either $<< 3, 6, 1 >>$ or $<< 6, 3, 1 >>$, the adversary can clearly identify which sequence of machine programs, that is path $p_{\mathcal{D}}$ or path $p'_{\mathcal{D}}$ was used to arrive at the current data structure state. The sequence of machine programs, in turn, tells the adversary the sequence of ADT operations used. Hence, the data structure implementation gives the adversary additional advantage in identifying the history of past execution, thereby breaking history independence.

How can we achieve history independence?

Currently, there are two known ways to make data structures history independent.

1. *For SHI, make the ADT and the data structure state transition graphs isomorphic:* Data structures with state transition graphs isomorphic to their ADT's state transition graph are referred to as canonically (or uniquely) represented data structures. We discuss the necessity of canonical representations for SHI in Section 5.4.4. SHI implies WHI.

2. *For WHI, make the data structure state transitions randomized:*
   Randomization here refers to the selection of the data structure state representing the corresponding ADT state. To illustrate, consider the example graphs from Figure 5.2. Both data structure states $<< 3, 6, 1 >>$ and $<< 6, 3, 1 >>$ are valid memory representations of the ADT state $\{1, 3, 6\}$. For WHI, the choice of data structure state to represent the ADT state $\{1, 3, 6\}$ must be random.

   Randomization translates to addition of new paths in the data structure state transition graph (Figure 5.3) to ensure the following: For any two ADT states s_0 and s_1, if there is a path in the ADT state transition graph between s_0 and s_1, then there must be a path from all memory representations of ADT state s_0 to all memory representations of ADT state s_1 in the data structure's state transition graph. The choice of path in the data structure state transition graph between representations of ADT states s_0 and s_1 is then made at random.

   From the adversary's point of view, randomization makes all memory representations of an ADT state equally likely to occur. Hence, observation of a specific representation gives the adversary no advantage in guessing the sequence of machine programs that led to the current data structure state. Since the adversary cannot identify the sequence of machine programs used, the adversary is also unable to identify the sequence of ADT operations that led to the current ADT state.

5.3.5 Memory Representations

The last concept that remains to be formalized before we move on to formal definitions for history independence (Section 5.4) is that of memory representations.

In the discussion of nonisomorphism and history independence above, we informally introduced memory representations for ADT states. We also showed that history independence comes into picture when an ADT state has multiple memory representations. In short, the memory representation for an ADT state that is reachable from the initial ADT state via a sequence of ADT operations, is the machine state reachable from the initial data structure state via the corresponding program sequence. We formally define memory representations here and use them later in Section 5.4 for the game-based definitions of history independence.

Let $\delta = \langle o_1, o_2, ..., o_n \rangle$ be a sequence of ADT operations and $I = \langle i_1, i_2, ..., i_n \rangle$ be a sequence of ADT inputs. We denote by $\mathbb{O}(\delta, s_0, I)$ the application of the ADT operation sequence δ on ADT state s_0.

$$\mathbb{O}(\delta, s_0, I) = \begin{cases} s_0 & \text{if } |\delta| = 0 \\ (s_n, \tau_n) | o_k(s_{k-1}, i_k) \to (s_k, \tau_k); 1 \leq k \leq n & \text{otherwise} \end{cases}$$

If δ is empty no state transition occurs and no outputs are produced. For nonempty sequence δ, s_n and τ_n denote the ADT state and the ADT output, respectively, produced by the final operation in sequence δ.

To summarize, we denote by $\mathbb{O}(\delta, s_0, I) \rightarrow (s_n, \tau_n)$ that the ADT operation sequence δ when applied to the ADT state s_0 with ADT input sequence I, results in the ADT state s_n and ADT output τ_n.

Now, let $\delta^{\mathcal{M}} = \chi(\delta) = \langle \gamma(o_1), \gamma(o_2), ..., \gamma(o_n) \rangle$ be a sequence of machine programs corresponding to the ADT operation sequence δ. $\gamma(o_k)$ is the machine program implementing the ADT operation o_k. Then, we denote by $\mathbb{O}^{\mathcal{M}}(\delta^{\mathcal{M}}, s_0^{\mathcal{M}}, I)$ the application of program sequence $\delta^{\mathcal{M}}$ on a machine state $s_0^{\mathcal{M}}$.

$$\mathbb{O}^{\mathcal{M}}(\delta^{\mathcal{M}}, s_0^{\mathcal{M}}, I) = \begin{cases} s_0^{\mathcal{M}} & \text{if } |\delta^{\mathcal{M}}| = 0 \\ (s_n^{\mathcal{M}}, \beta(\tau_n)) | \gamma(o_k)(s_{k-1}^{\mathcal{M}}, \alpha(i_k)) \rightarrow \{s_k^{\mathcal{M}}, \beta(\tau_k)\}; 1 \leq k \leq n & \text{otherwise} \end{cases}$$

Here, $\alpha(i)$ and $\beta(\tau)$ denote the machine representations for an ADT input i and an ADT output τ, respectively. $s_n^{\mathcal{M}}$ and $\beta(\tau_n)$ are the machine state and the machine output, respectively, produced by the final program in sequence $\delta^{\mathcal{M}}$.

In summary, we denote by $\mathbb{O}^{\mathcal{M}}(\delta^{\mathcal{M}}, s_0^{\mathcal{M}}, I) \rightarrow (s_n^{\mathcal{M}}, \tau_n^{\mathcal{M}})$ that a program sequence $\delta^{\mathcal{M}}$ when applied to a machine state $s_0^{\mathcal{M}}$ with an ADT input sequence I, results in a machine state $s_n^{\mathcal{M}}$ and a machine output $\tau_n^{\mathcal{M}}$.

Definition 6. Memory Representations
The set of memory representations of an ADT state s, denoted by $m(s)$, is the set of data structure states, defined as

$$m(s) = \begin{cases} \{s_0^{\mathcal{M}}\} & \text{if } s = s_\phi \\ \{s^{\mathcal{M}} \mid \mathbb{O}^{\mathcal{M}}(\delta_k^{\mathcal{M}}, s_0^{\mathcal{M}}, I_k) \rightarrow \{s^{\mathcal{M}}, \beta(\tau_{|\delta_k|})\}; 1 \leq k \leq n\} & \text{otherwise} \end{cases}$$

where, $s_0^{\mathcal{M}}$ is the initial data structure state; $I_1, I_2, ..., I_n$ are sequences of ADT inputs; $\delta_1, \delta_2, ..., \delta_n$ are ADT operation sequences, each of which when applied to the initial ADT state s_ϕ results in state s, that is $\mathbb{O}(\delta_k, s_\phi, I_k) \rightarrow (s, \tau_k)$; $\delta_k^{\mathcal{M}} = \chi(\delta_k)$ denotes the program sequence corresponding to ADT operation sequence δ_k; $|I_k| = |\delta_k|$; $1 \leq k \leq n$.

Here m is the mapping $m : \mathcal{S} \rightarrow 2^{\mathcal{S}^{\mathcal{D}}}$, where S is the set of all ADT states, $\mathcal{S}^{\mathcal{D}}$ is the set of all data structure states, and $2^{\mathcal{S}^{\mathcal{D}}}$ denotes the power set of $\mathcal{S}^{\mathcal{D}}$.

Dealing With Infinite ADT State Space

The set of machine states for the bounded RAM model is finite since there are finite number of available bits. Hence, a data structure implementation on a bounded RAM model can only have a finite number of data structure states. The set of ADT states on the other hand can be infinite. For an ADT with infinite states, a data structure implementation will be unable to uniquely represent all the ADT states. The case of infinite ADT states is of particular importance for canonically represented data structures that require the state transition graphs of the ADT and of the data structure to be isomorphic, that is, each ADT state has a unique memory representation.

We will look at canonical representations in detail within the context of history independence in Section 5.4.4. Here, we list two work-arounds to dealing with infinite ADT state space.

1. Redefine the ADT, such that the number of ADT states is less than or equal to the number of machine states.

2. Design each machine program implementing an ADT operation, such that the program produces a special output when an ADT state cannot be represented using the available machine bits. For example, an out-of-memory error.

5.4 History Independence

Now that we are equipped with the necessary concepts (ADT, RAM machine model, data structure, and memory representations), we proceed to formalize history independence. We give new game-based definitions for both WHI and SHI (Sections 5.4.1 and 5.4.1). The new definitions are equivalent to existing proposals [127,183] but more appropriate for the security community since they follow the game-based construction of semantic security. Further, our new definitions naturally extend to accommodate other notions of history independence beyond WHI and SHI.

We generalize history independence by introducing Δ history independence (ΔHI), a generic game-based definition of history independence that is malleable enough to accommodate WHI, SHI, and a broad spectrum of new history independence notions. Using ΔHI, we define new practical notions of history independence and also cover both WHI and SHI (Section 5.5.1). Finally, we show how ΔHI helps to reason about the history preserved or hidden by data structures including ones that were designed without history independence in mind (Sections 5.5.2 and 5.5.3).

Summary of notations: In the previous section, we introduced several notations for states, state transitions, operations, and program sequences. We summarize the notations in Table 5.2. The summary serves as a quick reference for history independence definitions that follow.

5.4.1 Weak History Independence (WHI)

WHI was introduced for scenarios wherein an adversary observes only the current data structure state. For example, as in the case of a stolen laptop. The current data structure state is the memory representation of the current ADT state. WHI then requires that observation of the current data structure state reveals no additional historical information to the adversary other than what is inherently available from the current ADT state.

Informally, a data structure is said to be weakly history independent if for any two sequences of ADT operations δ_1 and δ_2, that take the ADT from initialization to a state s, observation of any memory representation of state s gives the adversary no advantage in guessing whether sequence δ_1 or δ_2 was used to get to s.

We define weak history independence (WHI) by the following game:

Notation	Description
s_ϕ	Initial ADT state
$s_\phi^\mathcal{M}$	Initial machine state
$s_0^\mathcal{M}$	Initial data structure state
s_1, s_2	ADT states
$s_1^\mathcal{M}, s_2^\mathcal{M}$	Data structure (machine) states
δ, δ_1, δ_2	Sequences of ADT operations
$\delta^\mathcal{M}$, $\delta_1^\mathcal{M}$, $\delta_2^\mathcal{M}$	Sequences of machine programs
i, i_1, i_2	ADT inputs
I, I_1, I_2	Sequences of ADT inputs
τ, τ_1, τ_2	ADT outputs
$i^\mathcal{M} = \alpha(i)$	Denotes the machine representation of the ADT input i
$\tau^\mathcal{M} = \beta(\tau)$	Denotes the machine representation of the ADT output τ
$\gamma(o)$	Denotes the machine program corresponding to the ADT operation o
$\delta^\mathcal{M} = \chi(\delta)$	Denotes that $\delta^\mathcal{M}$ is a sequence of machine programs corresponding to the ADT operation sequence δ. If $\delta = \langle o_1, o_2, ..., o_n\rangle$ then $\delta^\mathcal{M} = \langle\gamma(o_1), \gamma(o_2), ..., \gamma(o_n)\rangle$
$\mathbb{O}(\delta, s_0, I) \rightarrow (s_n, \tau_n)$	Denotes that the ADT operations sequence δ when applied to the ADT state s_0 with ADT input sequence I, results in the ADT state s_n and ADT output τ_n
$\mathbb{O}^\mathcal{M}(\delta^\mathcal{M}, s_0^\mathcal{M}, I) \rightarrow (s_n^\mathcal{M}, \tau_n^\mathcal{M})$	Denotes that the program sequence $\delta^\mathcal{M}$ when applied to the machine state $s_0^\mathcal{M}$ with ADT input sequence I, results in the machine state $s_n^\mathcal{M}$ and machine output $\tau_n^\mathcal{M}$

Table 5.2: Summary of notations.

Let $\mathcal{A} = (\mathcal{S}, s_\phi, \mathcal{O}, \Gamma, \Psi)$ be an ADT, $\mathcal{M} = (\mathcal{S}^\mathcal{M}, s_\phi^\mathcal{M}, \mathcal{P}^\mathcal{M}, \Gamma^\mathcal{M}, \Psi^\mathcal{M})$ be a bounded RAM machine model, and $\mathcal{D} = (\alpha, \beta, \gamma, s_0^\mathcal{M})$ be a data structure implementing $\mathcal{A}$ in $\mathcal{M}$, as per definitions 3, 4 and 5, respectively.

1. A probabilistic polynomial time-bounded adversary selects the following: An ADT state s; two sequences of ADT operations δ_0 and δ_1; and two sequences of ADT inputs I_0 and I_1; such that $\mathbb{O}(\delta_0, s_\phi, I_0) \rightarrow (s, \tau)$ and $\mathbb{O}(\delta_1, s_\phi, I_1) \rightarrow (s, \tau)$. Both δ_1 and δ_2 take the ADT from the initial state s_ϕ to state s producing the same output τ.

2. The adversary sends s, δ_0, δ_1, I_0 and I_1 to the challenger.

3. The challenger flips a fair coin $c \in \{0, 1\}$ and computes $\mathbb{O}^{\mathcal{M}}(\delta_c^{\mathcal{M}}, s_0^{\mathcal{M}}, I_c) \to (s^{\mathcal{M}}, \tau^{\mathcal{M}})$, where $\delta_c^{\mathcal{M}} = \chi(\delta_c)$ and $\tau^{\mathcal{M}} = \beta(\tau)$. That is, the challenger applies the program sequence $\delta_c^{\mathcal{M}}$ corresponding to the ADT operation sequence δ_c to the data structure initialization state $s_0^{\mathcal{M}}$, resulting in a memory representation $s^{\mathcal{M}}$ of ADT state s and a machine output $\tau^{\mathcal{M}}$.

4. The challenger sends the memory representation $s^{\mathcal{M}}$ to the adversary.

5. The adversary outputs $c' \in \{0, 1\}$.

The adversary wins the game if $c' = c$.

A data structure is said to be weakly history independent if the advantage of the adversary defined as $\left|Pr[c' = c] - \frac{1}{2}\right|$ is negligible.

Since WHI permits the adversary to make a single observation, the adversary is allowed to choose the end state only in step 1. The starting state for the chosen ADT operation sequences is always the initial ADT state s_ϕ. Recall from the data structure definition (Section 5.3.3) that the initial ADT state has a fixed memory representation, which is the initial data structure state $s_0^{\mathcal{M}}$. Hence, in step 3, the challenger applies the adversary-selected sequence to the memory representation $s_0^{\mathcal{M}}$ of s_ϕ.

If the adversary is able to identify the ADT operation sequence chosen by the challenger in step 3, then the adversary wins[7] the game. Winning the game implies the adversary was able to determine the operation sequence that led to the current ADT state by observing the state's memory representation, thereby breaking WHI.

5.4.2 Strong History Independence (SHI)

Unlike WHI, SHI is applicable when an adversary can observe multiple memory representations throughout a sequence of operations For example, as in case of an insider who can obtain a periodic memory dump. SHI requires that the adversary must not gain any additional information about the sequence of operations between any two adjacent observations than what is inherently available from the corresponding ADT states.

Informally, a data structure is said to be strongly history independent if for any two sequences of ADT operations δ_1 and δ_2, that take the ADT from a state s_1 to a state s_2, observations of any memory representations of states s_1 and s_2 give the adversary no advantage in guessing whether sequence δ_1 or δ_2 was used to go from s_1 to s_2.

We define strong history independence (SHI) by the following game:

[7] A function f is negligible if for every positive polynomial poly() there exists an integer $n > 0$ such that for all $x > n$, $f(x) < \frac{1}{poly(x)}$.

Let $\mathcal{A} = (\mathcal{S}, s_\phi, \mathcal{O}, \Gamma, \Psi)$ be an ADT, $\mathcal{M} = (\mathcal{S}^\mathcal{M}, s_\phi^\mathcal{M}, \mathcal{P}^\mathcal{M}, \Gamma^\mathcal{M}, \Psi^\mathcal{M})$ be a bounded RAM machine model, and $\mathcal{D} = (\alpha, \beta, \gamma, s_0^\mathcal{M})$ be a data structure implementing $\mathcal{A}$ in $\mathcal{M}$, as per definitions 3, 4 and 5 respectively.

1. A probabilistic polynomial time-bounded adversary selects the following.
   - Two ADT states s_1 and s_2; two sequences of ADT operations δ_0 and δ_1; and two sequences of ADT inputs I_0 and I_1; such that $\mathbb{O}(\delta_0, s_1, I_0) \to (s_2, \tau)$ and $\mathbb{O}(\delta_1, s_1, I_1) \to (s_2, \tau)$. Both δ_1 and δ_2 take the ADT from state s_1 to state s_2 producing the same output τ.
   - A memory representation $s_1^\mathcal{M}$ of ADT state s_1.
2. The adversary sends s_1, $s_1^\mathcal{M}$, δ_0, δ_1, I_0 and I_1 to the challenger.
3. The challenger flips a fair coin $c \in \{0, 1\}$ and computes $\mathbb{O}^\mathcal{M}(\delta_c^\mathcal{M}, s_1^\mathcal{M}, I_c) \to (s_2^\mathcal{M}, \tau^\mathcal{M})$, where $\delta_c^\mathcal{M} = \chi(\delta_c)$ and $\tau^\mathcal{M} = \beta(\tau)$. That is, the challenger applies the program sequence $\delta_c^\mathcal{M}$ corresponding to the ADT operation sequence δ_c to the data structure state $s_1^\mathcal{M}$, resulting in a memory representation $s_2^\mathcal{M}$ of state s_2 and a machine output $\tau^\mathcal{M}$.
4. The challenger sends the memory representation $s_2^\mathcal{M}$ to the adversary.
5. The adversary outputs $c' \in \{0, 1\}$.

The adversary wins the game if $c' = c$.

A data structure is said to be strongly history independent if the advantage of the adversary defined as $\left|Pr[c' = c] - \frac{1}{2}\right|$ is negligible.

Winning the game means that the adversary was able to determine the operation sequence that took the ADT from state s_1 to state s_2, thereby breaking SHI.

SHI implies WHI. If the ADT state s_1 chosen by the adversary in step 1 is the initial ADT state s_ϕ, then the SHI game reduces to the WHI game of Section 5.4.1.

5.4.3 Equivalence to Existing History Independence Definitions

WHI and SHI were first introduced by Naor et al. [184]. Later, Hartline et al. [127] introduced new definitions for WHI and SHI. However, Hartline et al. showed that their definitions although less complex are equivalent to the ones proposed by Naor et al. Our game-based definitions of WHI and SHI (Sections 5.4.1 and 5.4.2) differ slightly from the definitions by Hartline et al. Specifically, Hartline et al. assume a computationally unbounded adversary. We address history independence in the presence of computationally bounded adversaries to be more in-line with reality. Further, new definitions were necessary to overcome impreciseness in existing definitions and to develop a framework for new history independence notions beyond WHI and SHI. We detail in the following.

Hartline et al. defined weak history independence as follows.

Definition 7. Weak History Independence (WHI)
A data structure implementation is weakly history independent if, for any two sequences of operations X and Y that take the data structure from initialization to state A, the distribution over memory after X is performed is identical to the distribution after Y. That is:

$$(\phi \xrightarrow{X} A) \land (\phi \xrightarrow{Y} A) \Longrightarrow \forall\ \mathbf{a} \in A,\ \mathbf{Pr}\big[\phi \xrightarrow{X} \mathbf{a}\big] = \mathbf{Pr}\big[\phi \xrightarrow{Y} \mathbf{a}\big]$$

In the above definition, $\phi \xrightarrow{X} B$ denotes that a operation sequence X when applied to the initial state ϕ, results in state A. The notation $\mathbf{a} \in A$ means than $\mathbf{a}$ is a memory representation of state A. $\mathbf{Pr}\big[\phi \xrightarrow{X} \mathbf{a}\big]$ denotes the probability that a sequence X when applied to initial state ϕ, results in representation $\mathbf{a}$.

Reconciling terminology

Hartline et al. do not formalize the concepts of data structure, data structure state and memory representations. A data structure's state is referred to as the data structure's content. Memory representation of a data structure state is the physical contents of memory that represent that state. We note that Naor et al. also used the same terminology in their definitions.

The WHI definition by Hartline et al. is imprecise in the following.

- Operation inputs and outputs are not considered.
- The same operation sequences are considered applicable to both data structure states and to memory representations. The mechanisms for the applicability are not specified.
- The connection between a data structure's state and the state's memory representations is not precisely specified.

Following Golovin et al. [113] we use the ADT concept to model logical states (or content) and define a data structure as an ADT's implementation (Sections 5.3.1 - 5.3.3). A data structure state is therefore the memory representation of an ADT state. Separating ADT and data structure concepts enables us to precisely define memory representations (Section 5.3.5) for various machine models; understand history independence from the perspective of state transition graphs; and to build a framework for defining new history independence notions other than SHI and WHI (Section 5.5).

To summarize the differences in terminology, what Hartline et al. refer to as data structure state in definition 8 is an ADT state in our model. Further, we refer to a memory representation in definition 8 as a data structure state.

For WHI, Hartline et al. require a data structure implementation to satisfy the following:

$$(\phi \xrightarrow{X} A) \land (\phi \xrightarrow{Y} A) \Longrightarrow \forall \mathbf{a} \in A, Pr\big[\phi \xrightarrow{X} \mathbf{a}\big] = Pr\big[\phi \xrightarrow{Y} \mathbf{a}\big] \tag{5.1}$$

Our game-based definition of WHI poses the following slightly relaxed requirement:

$$(\phi \xrightarrow{X} A) \wedge (\phi \xrightarrow{Y} A) \Longrightarrow \forall \mathbf{a} \in A, |Pr\big[\phi \xrightarrow{X} \mathbf{a}\big] - Pr\big[\phi \xrightarrow{Y} \mathbf{a}\big]| \text{ } \mathit{is\ negligible} \qquad (5.2)$$

We will show that the game-based WHI definition (Section 5.4.1) is equivalent to statement 5.2, that is, a data structure preserves WHI only if statement 5.2 is true. However, before we show the equivalence we point out the necessity for the difference between conditions 5.1 and 5.2.

As discussed in Section 5.3.4, there are two known ways to achieve history independence. The first way is to make the ADT and data structure state transition graphs isomorphic. The second way is to make the data structure state transition graph randomized. The requirement for identical memory distributions as per statement 5.1 rules out the use of randomization to achieve history independence[8]. A randomized data structure implementation will rely on pseudo random generators. The security of pseudo random generators relies on computational indistinguishability [33]. Therefore, the relaxed requirement of negligibility introduced in statement 5.2 is in fact not a limitation, but rather a reconciliation of the definition by Hartline et al. with reality where we have computationally bounded adversaries.

Although Naor et al. proposed a WHI definition that requires identical distributions, they also used randomization to design a history independent data structure.

Equivalence of WHI definitions

We now show that our gamed-based WHI definition (Section 5.4.1) is equivalent to a WHI definition based on statement 5.2.

We rewrite statement 5.2 for consistent notations as follows.

$$(s_\phi \xrightarrow{\delta_0} s) \wedge (s_\phi \xrightarrow{\delta_1} s) \Longrightarrow \forall s^{\mathcal{M}} \in s, |Pr\big[s_\phi^{\mathcal{M}} \xrightarrow{\delta_0^{\mathcal{M}}} s^{\mathcal{M}}\big] - Pr\big[s_\phi^{\mathcal{M}} \xrightarrow{\delta_1^{\mathcal{M}}} s^{\mathcal{M}}\big]| \text{ } \mathit{is\ negligible} \qquad (5.3)$$

Here, δ_0 and δ_1 are two ADT operation sequences that take the ADT from initial state s_ϕ to state s. s_ϕ and $s_\phi^{\mathcal{M}}$ are the initial ADT and the initial data structure states, respectively. $\delta_0^{\mathcal{M}}$ and $\delta_1^{\mathcal{M}}$ are the machine programs corresponding to ADT operation sequences δ_0 and δ_1, respectively.

History independence only considers cases where the condition $(s_\phi \xrightarrow{\delta_0} s) \wedge (s_\phi \xrightarrow{\delta_1} s)$ is true, that is, both sequences δ_0 and δ_1 take the ADT to the same end state s. Otherwise, the ADT states themselves reveal history.

We therefore have two cases to consider

Case 1: The distributions are computationally distinguishable, that is,

$$\exists s^{\mathcal{M}} \in s \text{ } \mathit{such\ that} \text{ } |Pr\big[s_\phi^{\mathcal{M}} \xrightarrow{\delta_0^{\mathcal{M}}} s^{\mathcal{M}}\big] - Pr\big[s_\phi^{\mathcal{M}} \xrightarrow{\delta_1^{\mathcal{M}}} s^{\mathcal{M}}\big]| \text{ } \mathit{is\ non-negligible}.$$

Now consider the following adversarial strategy. Given a data structure state $s^{\mathcal{M}}$ in step 4 of the WHI game, the adversary outputs c such that $\delta_c^{\mathcal{M}}$ has a higher probability of

[8] The use of randomization to achieve weak history independence is discussed in Section 5.4.5.

producing $s^{\mathcal{M}}$. For such an adversarial strategy $\left|Pr[c' = c] - \frac{1}{2}\right|$ is non-negligible for some $s^{\mathcal{M}}$. Therefore, the data structure implementation does not preserve WHI.

Case 2: The distributions are computationally indistinguishable, that is,

$$\forall s^{\mathcal{M}} \in s, |Pr\big[s_{\phi}^{\mathcal{M}} \xrightarrow{\delta_0^{\mathcal{M}}} s^{\mathcal{M}}\big] - Pr\big[s_{\phi}^{\mathcal{M}} \xrightarrow{\delta_1^{\mathcal{M}}} s^{\mathcal{M}}\big]| \; is\ negligible$$

In this case, from a computationally bounded adversary's perspective, the representation $s^{\mathcal{M}}$ received in step 4 of the WHI game is equally likely to have been produced by either $\delta_0^{\mathcal{M}}$ or $\delta_1^{\mathcal{M}}$. Hence, observation of a data structure state gives the adversary a negligible advantage in guessing c. The data structure implementation therefore preserves WHI.

Equivalence of SHI definitions

For strong history independence Hartline et al. proposed the following definition.

Definition 8. Strong History Independence (SHI)
A data structure implementation is strongly history independent if, for any two (possibly empty) sequences of operations X and Y that take a data structure in state A to state B, the distribution over representations of B after X is performed on a representation **a** *is identical to the distribution after Y is performed on* **a**. *That is:*

$$(A \xrightarrow{X} B) \wedge (A \xrightarrow{Y} B) \Longrightarrow \forall\, \mathbf{a} \in A, \forall\, \mathbf{b} \in B,\ \mathbf{Pr}\big[\mathbf{a} \xrightarrow{X} \mathbf{b}\big] = \mathbf{Pr}\big[\mathbf{a} \xrightarrow{Y} \mathbf{b}\big]$$

In the above definition, $A \xrightarrow{X} B$ denotes that a operation sequence X when applied to state A, results in state B. The notation $\mathbf{a} \in A$ means than $\mathbf{a}$ is a memory representation of state A. $\mathbf{Pr}\big[\mathbf{a} \xrightarrow{X} \mathbf{b}\big]$ denotes the probability that a sequence X when applied to memory representation $\mathbf{a}$, results in representation $\mathbf{b}$.

Similar to the case for WHI, our game-based SHI definition (Section 5.4.2) differs from the above definition only by relaxing the requirement for identical distributions. That is, for SHI, we require the following:

$$(A \xrightarrow{X} B) \wedge (A \xrightarrow{Y} B) \Longrightarrow \forall\, \mathbf{a} \in A, \forall\, \mathbf{b} \in B,\ |\mathbf{Pr}\big[\mathbf{a} \xrightarrow{X} \mathbf{b}\big]\ \textbf{-}\ \mathbf{Pr}\big[\mathbf{a} \xrightarrow{Y} \mathbf{b}\big]| \text{ is negligible}$$

The equivalence of SHI definitions follows similarly to the case of WHI.

Summary of Differences

The main differences between our definitions and the the definitions by Hartline et al. are the following

- The definitions by Hartline et al. are imprecise about the concepts of data structures, states, and memory representations. We precisely formalize all of these concepts.
- Hartline et al. do not consider the case of computationally bounded adversaries. We permit computationally bounded adversaries and thus have the negligibility definition instead of equality for memory distributions.

Programs of $\mathcal{M}$	**Random bits hidden from adversary**	**Adversary computationally bounded**	**History Independence desired**	**Canonical representations needed?**
Randomized	Yes	Yes	WHI	No
N/A	N/A	No	N/A	Yes
N/A	N/A	N/A	SHI	Yes
Deterministic	N/A	N/A	N/A	Yes

Table 5.3: Identification of scenarios where canonical representations are necessary for history independence. N/A = not applicable.

5.4.4 Canonical Representations And History Independence

Canonically (or uniquely) represented data structures have the property that each ADT state has a unique memory representation. Unique representation implies that the ADT and data structure state transition graphs are isomorphic[9]. Canonically represented data structures give very strong guarantees for history independence and in many cases are the only way to achieve history independence.

We first define canonically represented data structures and then discuss several important results pertaining to canonical representations and history independence. We also summarize (Table 5.3) the scenarios where canonical representations are necessary for history independence across all combinations of types of programs, secrecy of random bits, adversarial computational ability, and the desired notion of history independence.

Definition 9. Canonically represented data structure
A data structure $\mathcal{D}$ implementing an ADT $\mathcal{A}$ on a bounded RAM machine model $\mathcal{M}$ is canonically represented if each ADT state has a unique memory representation, that is, the mapping $m : \mathcal{S} \rightarrow 2^{\mathcal{S}^{\mathcal{D}}}$ is injective and $|m(s)| = 1$, where $\mathcal{S}$ is the set of all ADT states, $\mathcal{S}^{\mathcal{D}}$ is the set of all data structure states, and $m(s)$ denotes the set of memory representations of an ADT state $s \in S$ as per definition 6.

Impossibility of canonical representations for ADTs with infinite states

In Section 5.3.5, we discussed how to handle the case when the set of ADT states is infinite. The case of infinite ADT states is of particular importance for canonically represented data structure implementations on a bounded RAM machine model. Since the bounded RAM machine model has a finite number of available bits, the machine state space is not large enough to provide a unique representation for each ADT state when the ADT state space is infinite. Impossibility of unique representations clearly suggests that canonical representations for infinite state set ADTs are not possible in practice since machines with infinite state space do not exists in reality. This straight-forwardly leads to the following theorem.

[9] Isomorphism is discussed in Section 5.3.4.

Theorem 1. *Canonically represented data structure implementations for ADTs with infinite states are impossible in practice.*

However, prior work [113, 183, 184] has claimed designs for canonically represented data structures for the RAM model in direct contradiction to Theorem 1. The contradiction arises from the fact that prior work has implicitly considered ADTs with finite state space. Specifically, the ADTs considered have have fewer states than the the total number of machine states.

The necessity of canonical representations for SHI

Since history independence was first proposed [184], it has been known that canonically represented data structures support SHI. An interesting question posed in this context was whether canonical representations are necessary to achieve SHI. The question about the necessity of canonical representations for SHI was answered by Hartline et al. Hartline et al. [127] showed that SHI cannot be achieved without canonical representations.

Thus, we have the following theorem

Theorem 2. *A data structure is strongly history independent iff it is canonically represented.*

The proof by Hartline et al. [127] builds on the case that if a data structure is not canonically represented, then an adversary can distinguish an empty sequence of operations from a nonempty sequence of operations.

Why canonical representations are not necessary for WHI?

In the absence of canonical representations, it has been shown that an adversary can distinguish an empty sequence of operations from a nonempty sequence of operations thereby breaking SHI [127]. If operation sequences are always assumed to be nonempty, canonical representations are not necessary [127]. We will define such a slightly relaxed notion of history independence that permits only nonempty sequences in Section 5.5.1. Here, we show that WHI is preserved even for empty operation sequences in the absence of canonical representations.

Consider the WHI game from Section 5.4.1. The case in which the adversary selects two empty ADT operation sequences in step 1 is trivial since empty sequences cause no state transitions and hence there is no history to be revealed.

Now, consider the case when the adversary selects an empty sequence δ_ϕ and a nonempty sequence δ_1 of ADT operations. Both δ_ϕ and δ_1 are required to take the ADT from the initial state to the same end state. Since the empty sequence δ_ϕ causes no state transitions, end state for both sequences δ_ϕ and δ_1 will be the initial ADT state itself.

Then, in step 3, the challenger chooses either δ_ϕ or δ_1 and sends the resulting memory representation to the adversary. Since the end state for the two operation sequences is the initial ADT state, the memory representation sent to the adversary in step 4 will be the data structure initialization state. From the data structure definition (Section 5.3.3), we know that the initial ADT state has a corresponding fixed unique memory representation. Hence, irrespective of the nonempty sequence that the adversary selects in step 1, the adversary receives the initial ADT state's memory representation in step 4. Since the adversary receives the same representation each time, the adversary gains no advantage in guessing whether δ_ϕ or δ_1 was chosen by the challenger in step 3.

ADT states other than the initial ADT state can have multiple memory representations. Multiple representations for ADT states does not break WHI as long it is ensured that from the adversary's perspective, all representations of the current ADT state are equally likely to be observed. Equal likelyhood for all representations of an ADT state can be achieved using randomization (Section 5.4.5).

Canonical representations and adversary models

Canonically represented data structures are history independent in the strongest sense, secure even against a computationally unbounded adversary [113]. For a computationally unbounded adversary, canonical representations are also necessary for WHI.

5.4.5 Randomization and History Independence

In the previous section, we discussed the necessity of canonical representations for SHI. In this section, we discuss the use of randomization to achieve history independence. We note that using randomization only gives WHI.

In Section 5.3.4, we introduced the use of randomization for WHI from the point of view of state transition graphs. We showed that randomization involves ensuring that for any two ADT states s_0 and s_1, if there is a path in the ADT state transition graph between s_0 and s_1, then there must be a path from all memory representations of s_0 to all memory representations of s_1 in the data structure's state transition graph. The choice of path in the data structure state transition graph between representations of s_0 and s_1 is then made at random.

In practice, randomization is achieved using the machine programs implementing the ADT operations. An ADT operation o takes the ADT from a state s_1 to a state s_2. A machine program implementing o takes the data structure from a memory representation of state s_1 to a memory representation of state s_2. Since each ADT state can have several memory representations (Section 5.3.4), the program has a choice amongst all representations of state s_2 and picks one representation as the result of a transition. Starting from a fixed memory representation of s_1, and a fixed input, if the program takes the data structure to a fixed resulting representation of s_2 on each execution, then the program is said to be deterministic. If on each execution the resulting representation is chosen uniformly at random from all possible representations of state s_2, then the program is said to be randomized.

To illustrate, consider an ADT operation o and a machine program p implementing o. Let $o(s_1, i) \rightarrow (s_2, \tau)$ denote the transition from ADT state s_1 to ADT state s_2 using an ADT input i and producing an ADT output τ. Also, let $m(s_1)$ and $m(s_2)$ denote the set of memory representations of states s_1 and s_2, respectively. Then, for history independence, the following must hold for program p:

$$Pr[p(s_1^{\mathcal{M}}, \alpha(i)) \rightarrow (s_2^{\mathcal{M}}, \beta(\tau))] = \frac{1}{|m(s_2)|}, \forall\ s_1^{\mathcal{M}} \in m(s_1) \text{ and } \forall\ s_2^{\mathcal{M}} \in m(s_2).$$

Here, $\alpha(i)$ and $\beta(\tau)$ are the machine representations of ADT input i and ADT output τ, respectively.

Note that randomization here refers to the selection of memory representations for ADT states and not to program outputs. A program's output is the machine representation of the corresponding ADT operation's output.

If randomization is used for history independence, then random choices made by the machine programs must be hidden from the adversary. If the adversary has knowledge of the random bits, then from the adversary's point of view the machine programs are deterministic. Data structures with deterministic machine programs require canonical representations.

5.5 Generalizing History Independence

SHI is a very strong notion of history independence requiring canonical representations [113, 127]. Canonically represented data structures are not efficient [48]. For heap and queue data structures Buchbinder et al. [48] show that certain operations that require logarithmic time under WHI take linear time under SHI. Hence, it is worth to question the need for canonical representations for history independence. Many scenarios may not require such a strong notion of history independence making the use of SHI data structures with canonical representations an inefficient solution.

Following are some scenarios that can be efficiently realized by new history independence notions weaker than SHI.

- Hiding evidence of specific operations only. For example, hiding only the fact that a specific data item has been deleted in the past. Eliminating evidence of past deletes is directly applicable for regulatory compliance. Retention Regulations [83, 191, 252] are only concerned with hiding the past existence of deleted data and not with other aspects of history, such as the insertion sequence of current data.

- A MRU caching or a journaling system by definition reveals the last k operations. Hence, journaling and caching require a new notion of history independence, wherein no history is revealed other than the last k operations [184].

- Revealing only the number of times each operation is performed [113]. For example, in a file-sharing application disclosing file-access counts may be permissible, but not the access order.

Existing work [113,184] has already suggested that for efficiency, it is important to benefit from new relaxed notions of history independence. A proper theoretical framework is needed to precisely define new history independence notions. In the following, we take first steps towards such a framework.

A straight-forward way to define new notions of history independence is to provide a new game-based definition for each scenario. However, defining distinct scenario-specific games can quickly become a tedious process. Instead, we introduce a definitional framework that can accommodate a broad spectrum of history independence notions. We term the new framework as Δ history independence (ΔHI), where Δ is the parameter determining the history independence flavor. As we shall see, ΔHI also captures both WHI and SHI. In addition, ΔHI helps to reason about the history revealed or concealed by existing data structures which were designed without history independence in mind.

5.5.1 Δ History Independence (ΔHI)

The WHI and SHI games (Sections 5.4.1 and 5.4.2 respectively) are defined over a subset of ADT operation sequences. For WHI, the adversary is permitted to select sequences that take the ADT from initialization to the same end state. For SHI, the permitted sequences are ones that take the ADT from the same starting state to the same ending state. The selection is made by the adversary in step 1 of both the WHI and SHI games. Hence, the initial selection permitted to the adversary determines the history that is desired to be revealed or hidden. By generalizing the selection step, we can accommodate a broad spectrum of history independence notions. We achieve the generalization in ΔHI, which is defined by the following game:

Let $\mathcal{A} = (\mathcal{S}, s_\phi, \mathcal{O}, \Gamma, \Psi)$ be an ADT, $\mathcal{M} = (\mathcal{S}^\mathcal{M}, s_\phi^\mathcal{M}, \mathcal{P}^\mathcal{M}, \Gamma^\mathcal{M}, \Psi^\mathcal{M})$ be a bounded RAM machine model, and $\mathcal{D} = (\alpha, \beta, \gamma, s_0^\mathcal{M})$ be a data structure implementing $\mathcal{A}$ in $\mathcal{M}$, as per definitions 3, 4 and 5, respectively. Also, let ζ be the set of all ADT operation sequences, Υ be the set of all ADT input sequences, and Δ be a function $\Delta : \mathcal{S} \times \mathcal{S} \times \zeta \times \zeta \times \Upsilon \times \Upsilon \rightarrow \{0, 1\}$.

1. A probabilistic polynomial time-bounded adversary selects the following.
   - Two ADT states s_1 and s_2; two sequences of ADT operations δ_0 and δ_1; and two sequences of ADT inputs I_0 and I_1; such that $\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = 1$.
   - A memory representation $s_1^\mathcal{M}$ of ADT state s_1.
2. The adversary sends s_1, $s_1^\mathcal{M}$, δ_0, δ_1, I_0 and I_1 to the challenger.
3. The challenger flips a fair coin $c \in \{0, 1\}$ and computes $\mathbb{O}^\mathcal{M}(\delta_c^\mathcal{M}, s_1^\mathcal{M}, I_c) \rightarrow (s^\mathcal{M}, \tau^\mathcal{M})$, where $\delta_c^\mathcal{M} = \chi(\delta_c)$. That is, the challenger applies the program sequence $\delta_c^\mathcal{M}$ corresponding to the ADT operation sequence δ_c to the data structure state $s_1^\mathcal{M}$, resulting in a memory representation $s^\mathcal{M}$, and a machine output $\tau^\mathcal{M}$.

4. The challenger sends the memory representation $s^{\mathcal{M}}$ and the machine output $\tau^{\mathcal{M}}$ to the adversary.

5. The adversary outputs $c' \in \{0, 1\}$.

The adversary wins the game if $c' = c$.

A data structure is said to be Δ history independent if the advantage of the adversary defined as $\left|Pr[c' = c] - \frac{1}{2}\right|$ is negligible.

Function Δ determines the pairs of ADT states, ADT operation sequences, and ADT input sequences that the adversary is permitted to select in step 1 of the ΔHI game. For the adversary-selected ADT states, operation sequences, and input sequences, the ΔHI game can be played and the data structure implementation is required to ensure that the advantage of the adversary is negligible. Thus, for a given ADT, Δ defines two sets,

$$H_{\Delta} = \{(s_1, s_2, \delta_0, \delta_1, I_0, I_1) \mid \Delta(s_1, s_2, \delta_1, \delta_2, I_0, I_1) = 1\}, \text{ and}$$
$$\overline{H}_{\Delta} = \{(s_1, s_2, \delta_0, \delta_1, I_0, I_1) \mid \Delta(s_1, s_2, \delta_1, \delta_2, I_0, I_1) = 0\}.$$

For all tuples in H_{Δ}, history independence is preserved, that is, neither the ADT nor the data structure implementation reveals the operation sequence selected by the challenger in step 3. For all tuples in $\overline{H}_{\Delta}$, history independence is not required to be preserved since the ADT itself reveals the sequence of operations used.

A careful choice of Δ allows us to precisely define both SHI and WHI, and a broad spectrum of new history independence notions. In the following, we illustrate the use of ΔHI framework to define some familiar history independence notions and a few previously unconsidered notions of history independence.

Strong History Independence (SHI)

We discussed SHI in Section 5.4.2. Here, we define the function Δ for SHI.

$$\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 1 & \text{if } \mathbb{O}(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \textit{ and } \mathbb{O}(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \\ 0 & \text{otherwise} \end{cases}$$

For SHI, the adversary's advantage in the ΔHI game must be negligible when in step 1, the adversary selects any two ADT operation sequences that take the ADT from a state s_1 to a state s_2 producing the same ADT output τ.

Weak History Independence (WHI)

Refer to Section 5.4.2 for discussion on WHI, which requires the following definition of Δ.

$$\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 1 & \text{if } s_1 = s_\phi \textit{ and } \mathbb{O}(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \textit{ and} \\ & \quad \mathbb{O}(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \\ 0 & \text{otherwise} \end{cases}$$

Since WHI permits the adversary to observe a single data structure state, the adversary chooses only the end state s_2 in step 1 of the ΔHI game. The starting state on which sequences δ_0 and δ_1 are applied is the initial ADT state s_ϕ.

Null history independence (ϕHI)

Under null history independence, a data structure conceals no history except for the trivial case when the ADT operation sequences and ADT input sequences selected by the adversary in the ΔHI game are identical. Example of a data structure with ϕHI is an append-only log. We can reflect ϕHI using the following.

$$\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 1 & \text{if } \mathbb{O}(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \textit{ and } \mathbb{O}(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \\ & \quad \textit{and } \delta_1 = \delta_2 \textit{ and } I_1 = I_2 \\ 0 & \text{otherwise} \end{cases}$$

SHI*

The necessity of canonical representations for SHI was proven by Hartline et al. [127]. The proof by Hartline et al. [127] builds on the case that if a data structure is not canonically represented, then an adversary can distinguish an empty sequence of operations from a nonempty sequence. Hartline et al. [127] then proposed SHI*, which is defined over nonempty ADT operation sequences. SHI* data structures were initially expected to more efficient than data structures providing SHI. However, Hartline et al. [127] found that SHI* still poses very strict requirements on a data structure and may not differ from SHI in asymptotic complexity.

Here, we give the Δ function for SHI*.

$$\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 1 & \text{if } \mathbb{O}(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \textit{ and } \mathbb{O}(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \textit{ and} \\ & \quad |\delta_0| > 0 \textit{ and } |\delta_1| > 0 \\ 0 & \text{otherwise} \end{cases}$$

SHI* closely resembles SHI except that the operations sequences δ_0 and δ_1 must be nonempty.

Reveal last k operations (MRU Cache, File System Journal)

System features such as caching and journaling by definition reveal the last k operations performed from the ADT state itself. Thus, for caching and journaling, we need to define a Δ function, such that no additional historical information is leaked from the memory representations other than the last k operations. We define the new notion as follows.

Let $\delta[i]$ denote the i^{th} operation in the sequence δ. Also, let $\delta[i, j]$ denote a subsequence of δ, $i \leq j$.

$$\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 1 & \text{if } \mathbb{O}(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \textit{ and } \mathbb{O}(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \textit{ and} \\ & \quad |\delta_0| \geq k \textit{ and } |\delta_1| \geq k \textit{ and } \delta_0[|\delta_0| - k, |\delta_0|] = \delta_1[|\delta_1| - k, |\delta_1|] \\ 0 & \text{otherwise} \end{cases}$$

Here, the adversary is permitted to choose two sequences δ_0 and δ_1, such that last k operations in δ_0 and δ_1 are the same. Other than the last k operations, sequences δ_0 and δ_1 may differ. Yet, the adversary should be unable to identify the sequence chosen by the challenger in step 3.

Operation-Agnostic History Independence (OAHI)

Consider a secure deletion application that wishes to destroy any evidence of a delete operation performed in the past. That is, an adversary should be unable to detect whether a delete operation was performed or not other than guessing. In general, any particular operation may require to be concealed, not just deletes. We introduce a new notion of history independence that conceals specific ADT operations. The new notion is referred to as operation-agnostic history independence (OAHI). A data structure that is Δ history independent given the following Δ function guarantees operation-agnostic history independence for an ADT operation o.

$$\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 1 & \text{if } \mathbb{O}(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \textit{ and } \mathbb{O}(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \textit{ and} \\ & \quad o \in \delta_0 \textit{ and } o \notin \delta_1 \\ 0 & \text{otherwise} \end{cases}$$

In OAHI, neither the presence of operation o in δ_0, nor the absence of o in δ_1 gives the adversary any advantage in guessing the sequence chosen by the challenger in step 3 of the ΔHI game.

Operation-Instance-Agnostic History Independence (OIAHI)

Consider a file system or a database index that features irrecoverable erase. That is, no evidence of the past deletion of a particular data item is preserved in the current memory representation. Concealing item-specific deletion requires hiding the fact that a particular ADT operation with a specific input was performed. For example, concealing a file delete operation with specific file name as input. For concealing item-specific deletes, we introduce operation-instance-agnostic history independence (OIAHI). Following is the Δ function for OIAHI.

$$\Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = \begin{cases} 1 & \text{if } \mathbb{O}(\delta_0, s_1, I_0) \rightarrow (s_2, \tau) \textit{ and } \mathbb{O}(\delta_1, s_1, I_1) \rightarrow (s_2, \tau) \textit{ and} \\ & \quad \delta_0[j] = o \textit{ and } \delta_1[k] = o \textit{ and } I_0[j] \neq I_1[k], \\ & \quad 1 \leq j \leq |\delta_0|, 1 \leq k \leq |\delta_1| \\ 0 & \text{otherwise} \end{cases}$$

For a given ADT operation o, OAHI hides the past execution of o. OIAHI on the other hand does not hide the past execution of o, but hides the input used. To illustrate a practical comparison, the history independent file system (HIFS) we propose in Chapter 6 supports OAHI. The Ext3 [5] file system supports OIAHI. B-trees [74] feature neither.

5.5.2 Measuring History Independence

We have seen that new notions of history independence can be easily derived from Δ history independence by defining the appropriate Δ function. In this section, we present an intuitive way of comparing Δ functions on the basis of the history they require to be concealed or preserved.

For a given Δ function we defined the set H_Δ (Section 5.5.1) that represents all combinations of ADT states, operation sequences, and ADT input sequences for which the adversary's advantage is negligible in the Δ history independence game. That is, for all members of H_Δ, history independence is preserved. One insight is to use the cardinality of H_Δ as a measure of history independence.

Recall from Section 5.3.3 that an ADT can have several data structure implementations. Let $\mathcal{D}$ and $\mathcal{D}'$ be two implementations of an ADT $\mathcal{A}$, such that $\mathcal{D}$ is Δ history independent and $\mathcal{D}'$ is Δ' history independent for two functions Δ and Δ'. Now, we say that $\mathcal{D}$ is more history independent than $\mathcal{D}'$ if $H_{\Delta'} \subset H_\Delta$.

Note that $|H_\Delta| > |H_{\Delta'}|$ alone does not imply that $\mathcal{D}$ is more history independent than $\mathcal{D}'$ since an application may be more sensitive to the history preserved by D' than the history preserved by D. Only in the case where $H_{\Delta'} \subset H_\Delta$ can we consider $\mathcal{D}$ to be a more history independent implementation than $\mathcal{D}'$.

5.5.3 Deriving History Independence

In order to provide a history independent implementation for an ADT, we first require the Δ function to be precisely defined. Then, a history independent data structure can be designed that satisfies the Δ function. Satisfying a Δ function means that the adversary's advantage is always negligible in the Δ history independence game. In effect, so far we have approached history independence as a define-then-design process.

However, data structures have been in use for a long time and most data structures have been designed for efficiency or functionality with no history independence in mind. A natural question then arises – are there any meaningful[10] Δ functions satisfied by existing data structures?.

A data structure can be Δ history independent for several Δ functions. For example, a data structure that satisfies SHI, also satisfies WHI, OAHI, and OIAHI. Hence, for a given data structure $\mathcal{D}$ finding a Δ function may not be a particularly difficult task. It may be more useful instead to determine an uncontained Δ function for $\mathcal{D}$. We define an uncontained Δ function for a data structure as follows.

Definition 10. Uncontained Δ function
A Δ function for a data structure $\mathcal{D}$ is uncontained if $\mathcal{D}$ is Δ history independent and $\nexists\, \Delta'$, such that $\mathcal{D}$ is also Δ' history independent and $H_\Delta \subset H_{\Delta'}$, where $H_\Delta = \{(s_1, s_2, \delta_0, \delta_1, I_0, I_1) \mid \Delta(s_1, s_2, \delta_0, \delta_1, I_0, I_1) = 1\}$; $H_{\Delta'} = \{(s'_1, s'_2, \delta'_0, \delta'_1, I'_0, I'_1) \mid \Delta'(s'_1, s'_2, \delta'_0, \delta'_1, I'_0, I'_1) = 1\}$; s_1, s_2,

[10] $\Delta = 0$ is satisfied by all data structures. Hence, we need to determine Δ functions that are more useful in practice.

s'_1, and s'_2 are ADT states; δ_0, δ_1, δ'_0, and δ'_1 are ADT operation sequences; and I_0, I_1, I'_0, and I'_1 are ADT input sequences.

We can determine an uncontained Δ function for existing data structures on a case-by-case basis. An open question is whether there exists a general mechanism for deriving an uncontained Δ function for a given data structure.

5.6 From Theory To Practice

5.6.1 Defining Machine States

The RAM model of execution described in Section 5.3.2 consists of two components, the RAM and the CPU. Hence, the machine state for the RAM model includes bits from both the RAM and the CPU. In general, the machine state for a system-wide machine model will comprise all system component states. A system-wide history independent implementation has to then consider each individual component's characteristics along the interaction between the components. Providing system-wide history independence is therefore challenging.

However, in practice an adversary may have access to only a subset of system components. In this case, for the purpose of history independence, the machine state can be defined over the adversary-accessible components only. For example, history independent data structures proposed in existing work (Section 9.3.1) are designed with the RAM model in mind. However, the machine states considered for history independence only include bits from the RAM and exclude the CPU.

5.6.2 Building History Independent Systems

Various techniques for designing history independent data structures for commonly used ADTs such as queues, stacks, and hash tables have been proposed [113]. Our focus on the other hand is designing *systems* with end-to-end history independent characteristics. The difference between history independent implementations for simple ADTs, such as stacks and queues versus a complete system, such as a database, or a file system is a matter of often exponentially increasing complexity. Fundamentally, any system can be modeled as an ADT and an history independent implementation can be sought for the system.

We introduce a general recipe for building history independent systems as follows:

1. Model the system as an ADT. For a specific example of file system as an ADT, refer to Chapter 6.

2. Select a machine model for implementation. While defining the machine state identify all machine components that the adversary has access to and define the machine state associated with the adversary-accessible components.

3. Depending on the application scenario, fix a desired notion of history independence and the corresponding Δ function.

4. Based on the definition of Δ, provide an implementation over the selected machine model. For complex systems, the implementation will likely require the most effort since the machine programs implementing the ADT operations must provably ensure that the advantage of the adversary is negligible in the ΔHI game.

In Chapter 6, we follow the above recipe to design a history independent file system.

5.7 On A Philosophical Note

At a very high level, the motivation for history independence can be stated as follows.

> *For any logical state S_L, the physical state S_P representing S_L may reveal information about the history leading to S_L, that is otherwise not discernible via solely S_L.*

So far, we have considered the logical state to be the ADT state and the physical state to be the underlying machine state representing the ADT state, that is, the physical state is the set of all bits of the machine. Our selection of logical and physical states seems rather arbitrary. We do this specific selection due to our adversary model, which assumes that the adversary can interpret information at the level of bits. An adversary, that can for example, examine the electric charge in individual capacitors used to represent the bits will require a different choice of logical and physical state descriptions. A straight-forward choice would be to consider a bit as a logical state and the precise capacitor state as the physical state.

The following interesting question arises from this discussion – *is history independence only a matter of perspective*?. The short answer is *yes*, history independence is a matter of perspective. There is no universal history independence.

To clarify, consider the universe as a whole from the viewpoint of classical physics. Under the classical viewpoint, knowledge of current state of all objects in the universe enables determination of any past or future universal state since the laws of physics work both forwards and backwards in time. Hence, the past is never hidden and history independence is impossible. For example, using the currently observed movement of galaxies, the past states of the universe can be inferred up to the very initial moments of the big bang.

Physical phenomena at the subatomic scale is explained by quantum physics. At the quantum level, the universe appears nondeterministic. Further, the uncertainty principle [215] restricts the ability to accurately measure the current state of a quantum system. Since the current state cannot be accurately known, it may seem the past states cannot be determined either and history independence can be achieved at the quantum level.

However, even at the quantum level history independence is still a matter of perspective. The perspective is governed by the interpretation of quantum physics used. Under the many-worlds interpretation, the multiverse as a whole is deterministic [93]. The probabilistic nature at the quantum level is only our perception since our observations are limited to a single universe. A hypothetical all-powerful adversary that can view the entire multiverse would have a full view of the past and the future similar to the case of classical physics making history independence in the presence of such an adversary impossible.

5.8 Conclusions

In this chapter, we took a deep look into history independence from both a theoretical and a systems perspective. We explored the concepts of abstract data types, machine models, data structures and memory representations. We identified the need for history independence from the perspective of ADT and data structure state transition graphs. Then, we introduced Δ history independence, which serves as a general framework to define a broad spectrum of history independence notions including strong and weak history independence. We also outlined a general recipe for building history independent systems, which we will use in designing a history independent file system in Chapter 6.

Chapter 6

History Independent File System

6.1 Chapter Overview

6.1.1 Background and Motivation

In Chapter 5, we identified the role of history independence for data retention Regulations. Although compliance with retention Regulations is our prime motivation, we note that any application that relies on persistent history independent data structures cannot be realized in practice without an underlying file system that is also history independent. If the file system is not history independent, then the file system's organization of data will break application-level history independence (Section 6.2). Examples of applications that rely on persistent history data structures include incremental signature schemes [183] and e-voting [38].

Existing file systems, such as Ext3 [5] are not history independent because they organize data on disk as a function of both files' data and the sequence of file operations. The exact same set of files can be organized differently on disk depending on the sequence of file system operations that created the set. As a result, observations of data organization on disk can potentially reveal file system's history. Moreover, file system metadata also contains historical information, such as list of allocated blocks. Therefore, when observations of data organization are combined with file system metadata, and with knowledge of application logic, significantly more historical information can be derived, for example, full recovery of deleted data. It is therefore imperative to hide file system history.

File system history can be hidden by making file system implementations history independent. A straight-forward way to achieve history independence is to use existing history independent data structures to organize files' data on disk. Current techniques to make history independent data structures persistent require the use of history independent hash tables [113]. The history independent hash tables [38] in-turn use uniform hash functions, distributing files' data on storage with no consideration to data locality. Hence, existing history independent data structures destroy data locality. Since data locality is critical for file system efficiency, the direct adoption of current history independent data structures in file system design is impractical.

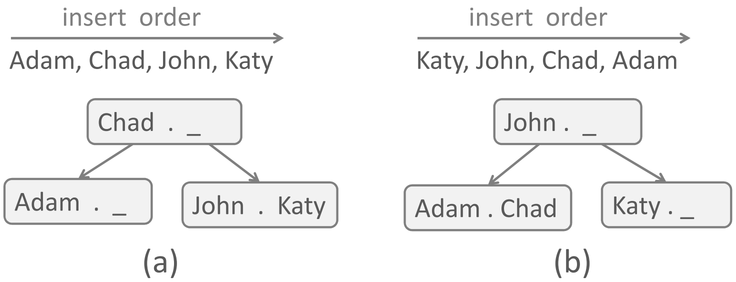


Figure 6.1: B-Tree, a non history independent data structure.

6.1.2 Our Contribution: History Independent File System (HIFS)

In Chapter 5 we layed the theoretical foundations for history independence. We explored the concepts of ADTs, machine models, data structures, and memory representations. We then formalized history independence and introduced the Δ history independence framework. In this chapter, we apply the theoretical concepts and results towards practical history independent system design. Using the recipe outlined in Section 5.6.2, we design, implement, and evaluate a history independent file system (HIFS).

In HIFS, we overcome the challenge of providing history independence while preserving data locality. Moreover, HIFS is configurable to different data locality scenarios, such as block group locality and complete sequential. Data locality is preserved in the presence of strong history independence (SHI).

6.1.3 Chapter Outline

Section 6.2 illustrates the need for history independence in file systems through a practical example. Section 6.3 describes the adversarial model. Theoretical concepts are introduced in Section 6.4. Section 6.5 details HIFS architecture including proofs of history independence. Issues, such as maintenance of temporal metadata and in-memory history independence are discussed in Section 6.6. Section 6.7 presents experimental evaluation of HIFS. HIFS demo application is introduced in Section 6.8. Finally, Section 6.9 concludes the chapter.

6.2 Illustrative Example

6.2.1 File Systems

To illustrate, the need for history independence in file systems we consider an admissions management application at a hospital. The application records patients' data as new patients are admitted. The application API permits the hospital staff to add new patient records, lookup existing records and delete patient records on discharge. We use this example for illustration. However, most existing applications of history independent data structures cannot be securely realized in practice without an underlying history independent file system.

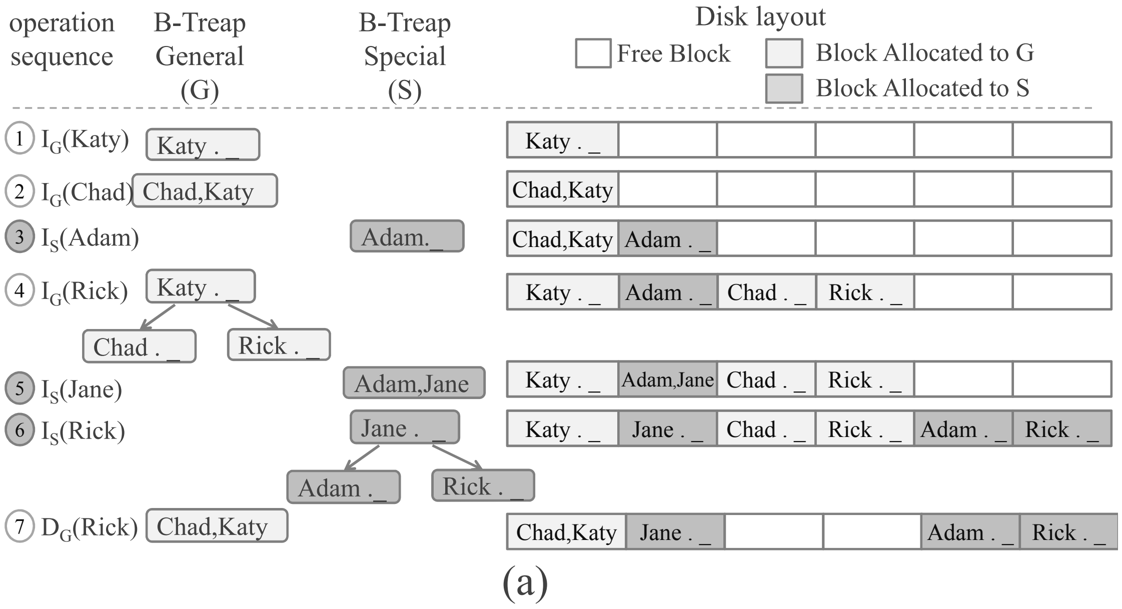
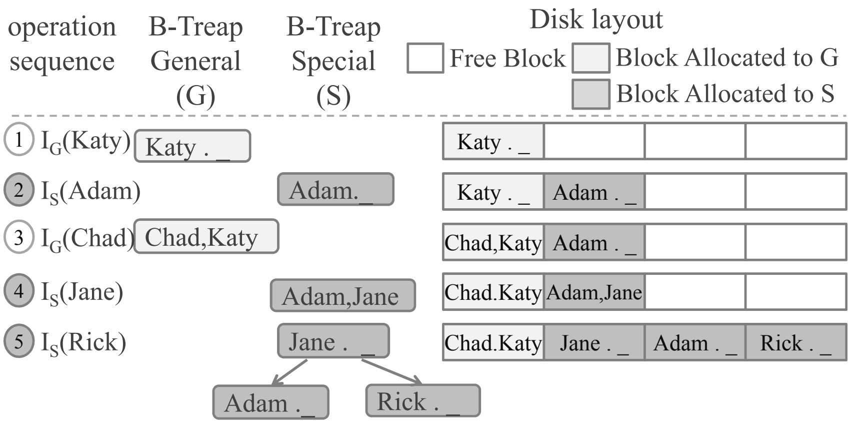


Figure 6.2: B-Treaps and file system layouts. $I_R(t)$ and $D_R(t)$ denote insertion and deletion respectively of element t in relation R.

The admissions management application will typically utilize a database to manage its data. Now, a database in turn stores and manipulates data by utilizing efficient data structures of which B-Trees [74] are the most common example. However, the use of B-Trees causes several privacy concerns. The concerns arise because the storage layout of B-Trees or of variations such as B$^+$-Trees often depends on the order in which operations are performed due to deterministic insertions and deletions. For example, Figure 6.1 shows two B-Trees that store the exact same elements. Yet the layouts of the two B-Trees differ due to different insertion orders. Therefore, by a simple examination of the tree layout (Figure 6.1(a)) one can ascertain with a probability of 75% that *Chad* was admitted before *John*. The deduction of admission order is possible due to the fact that out of total 4! ways of insertion for the four elements *Adam, Chad, John and Katy*, *Chad* is the root in only twelve, and inserted before *John* in nine of those twelve sequences.

To ensure privacy, B-Trees can be replaced by corresponding history independent versions, such as B-Treaps [114]. The treaps will yield the same layout irrespective of the insertion order. However, a simple replacement of application data structures with their history independent versions does not suffice unless the underlying file system is also history independent. To clarify, suppose that the B-Trees are replaced by the history independent B-Treaps [114]. Also, suppose that the application has two relations, one for admissions to the General ward and another for admissions to the Special ward. Let both relations be persisted using a simple file system that allocates the first available free block on request. The file system is therefore not history independent. For simplicity, assume that the B-Treap node size is equal to the file system block size.

Now, consider the sequence of operations, the resultant B-Treaps, and the space allocation by the underlying file system as shown in Figure 6.2. At the end of operation 7, the disk



(b)

Figure 6.3: B-Treaps and file system layouts. $I_R(t)$ and $D_R(t)$ denote insertion and deletion respectively of element t in relation R.

layout (Figure 6.2) reveals the following.

i. The fact that a delete operation was performed. The gaps left by the file system allocation form evidence for a delete.

ii. That the deleted node belonged to the General relation. Since otherwise there would be no gaps in the file system.

iii. The first patient was not admitted to the Special ward, since the root node of Special relation is not the first block on storage.

Note that neither the B-Treaps' layout nor the application API reveal any of (i) - (iii). The leaks are solely due to file system allocation.

Figure 6.3 shows an alternate sequence of operations that yield the exact same trees as in Figure 6.2. However, even though the tress match, the disk layouts are significantly different showing that the disk layouts produced by the file system allocation heavily depend on file system operation sequencing. Hence, underlying storage mechanisms can defeat the history independence of higher level data structures.

Existing file systems, such as Ext3 [5] are not history independent. For efficiency and to preserve locality, existing file systems allocate new blocks to files based on existing state, resulting in heavily history dependent layouts. A file system with strong history independence would carry no evidence of a delete, such as gaps in block allocation. Thus, a strongly history independent file system would reveal none of (i) - (iii) above, resulting in the exact same disk layouts for the operation sequences of both Figures 6.2 and 6.3.

6.3 Model

Adversary

We assume an insider adversary with full access to the storage medium, such as the system disk. By analyzing data organization on disk, the adversary aims to derive file system's history.

We assume that the adversary can make multiple observations of disk state. Recall from Section 5.4.4 that thwarting such an adversary requires SHI with canonical representations. Hence, our HIFS design targets canonical representations for file storage.

Storage Medium

The underlying storage device is assumed to be a mechanical disk drive, not flash storage.

6.4 Concepts

In Section 5.6.2, we outlined a general recipe for building history independent systems. In the following, we use the recipe to design HIFS. First, we define a file system as an ADT (Section 6.4.1). Then, we describe the machine model over which we seek an history independent implementation for the file system ADT (Section 6.4.2). In Section 5.6.2 we have already outlined the need of canonical representations for SHI. Hence, the Δ function for HIFS is the same as that for SHI defined in Section 5.5.1. Finally, we detail our HIFS implementation (Sections 6.5 - 6.7).

6.4.1 File System as an ADT

A file system organizes data as a set of files. We consider a file to consists of some metadata and a bit string. That is, a file $f = \{m_f, b_f\}$, where m_f is the file metadata and $b_f \in \{0,1\}^*$. We define file system as an ADT using the file type. Refer to Section 5.3.3 for a discussion on ADTs and types.

A file system is an ADT, that is, a pentuple $(\mathcal{S}, s_\phi, \mathcal{O}, \Gamma, \Psi)$, where

- $\mathcal{S} = 2^{\mathcal{F}}$, is the set of states. Here $\mathcal{F}$ is the set of all files.
- $s_\phi \in \mathcal{S}$ is the initial state.
- $\Gamma = \mathbb{N} \cup \{0,1\}^* \cup (\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \cup (\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \{0,1\}^*)$ is the set of inputs.
- $\Psi = \mathbb{Z} \cup (\{0,1\}^* \times \mathbb{Z})$ is the set of outputs.
- The set of operations $\mathcal{O} = \{\text{open, read, write, delete, close}\}$, such that
  - open : $\mathcal{S} \times \{0,1\}^* \to \mathcal{S} \times \mathbb{Z}$.
  - read : $\mathcal{S} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathcal{S} \times \{0,1\}^* \times \mathbb{Z}$.

- write : $\mathcal{S} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \{0,1\}^* \rightarrow \mathcal{S} \times \mathbb{Z}$.
- delete : $\mathcal{S} \times \mathbb{N} \rightarrow \mathcal{S} \times \mathbb{Z}$.
- close : $\mathcal{S} \times \mathbb{N} \rightarrow \mathcal{S} \times \mathbb{Z}$.

File systems including HIFS, support several additional operations. We have included only a small subset of the operations here for brevity.

6.4.2 RAMDisk Machine Model

In Section 5.3.2, we introduced the RAM machine model. The RAM model consists of two components, a central processing unit (CPU) and a random access memory (RAM). However, a file system is generally used to store and manage data over a secondary storage device. Hence, we define the RAMDisk model which in addition to the CPU and memory also includes the storage disk.

Definition 11. RAMDisk Machine Model
A RAMDisk machine model $\mathcal{M}_{\mathcal{D}}$ with m b-bit memory words, n b-bit CPU registers, and c k-bit disk blocks is a pentuple $(\mathcal{S}, s_\phi, \mathcal{P}, \Gamma, \Psi)$, where $\mathcal{S} = \{0,1\}^{b(m+n)+c\cdot k}$ is a set of machine states; $s_\phi \in \mathcal{S}$ is the initial state; $\mathcal{P}$ is the set of all programs of $\mathcal{M}_{\mathcal{D}}$; $\Gamma = \{0,1\}^*$ is a set of inputs; $\Psi = \{0,1\}^*$ is a set of outputs; and each program $p \in \mathcal{P}$ is a function $p : \mathcal{S} \times \Gamma_p \rightarrow \mathcal{S} \times \Psi_p$, where $\Gamma_p \subseteq \Gamma$ and $\Psi_p \subseteq \Psi$.

$\mathcal{M}_{\mathcal{D}}$ is initialized to state s_ϕ. If a program $p \in \mathcal{P}$ with input $i \in \Gamma_p$ is executed by the CPU when $\mathcal{M}_{\mathcal{D}}$ is in state s_1, $\mathcal{M}_{\mathcal{D}}$ outputs $\tau \in \Psi_p$ and transitions to a state s_2. The transition from state s_1 to state s_2 is denoted as $p(s_1, i) \rightarrow (s_2, \tau)$.

According to our model (Section 6.3), the adversary has access to the storage disk. Recall from Section 5.6.1 that for the purpose of history independence, we need to consider the machine states associated with the adversary-accessible components only. Hence, from this point onwards we refer to the storage device state as the machine state. Since the adversary does not access CPU and RAM components, we permit the CPU and RAM states to reveal history.

6.4.3 File System Implementation (Data Structure)

The objectives of HIFS design are three-fold.

1. For a given set of files, the organization of files' data and files' metadata on disk must be the same independent of the sequence of file operations. That is, file system implementation must be canonically represented and thereby preserve SHI.

2. Despite history independent storage, data locality must be preserved.

3. The implementation must be easily customizable to suit a wide range of data locality scenarios.

HIFS is a history independent implementation of the file system ADT from Section 6.4.1. That is, HIFS is a data structure $\mathcal{D} = (\alpha, \beta, \gamma, s_0^{\mathcal{M}})$ obtained as follows.

- For all $n \in \mathbb{N}_b$, $\alpha(n) \in \{0,1\}^b$. Here, $\mathbb{N}_b = \{x | x \in \mathbb{N}\ and\ x \leq 2^b\}$, b is the machine word length, and $\alpha(n)$ is the bit string representing n. For all $t_s \in \{0,1\}^{c \cdot k}, \alpha(t_s) = t_s$, For all $(n_1, n_2, n_3) \in \mathbb{N}_b \times \mathbb{N}_b \times \mathbb{N}_b, \alpha((n_1, n_2, n_3)) = \alpha(n_1)||\alpha(n_2)||\alpha(n_3)$. For all $(n_1, n_2, n_3, t_s) \in \mathbb{N}_b \times \mathbb{N}_b \times \mathbb{N}_b \times \{0,1\}^{c \cdot k}$, $\alpha((n_1, n_2, n_3, t_s)) = \alpha(n_1)||\alpha(n_2)||\alpha(n_3)||t_s$.
- For all $z \in \mathbb{Z}_b$, $\alpha(z) \in \{0,1\}^b$. Here, $\mathbb{Z}_b = \{x | x \in \mathbb{Z}\ and\ x \leq 2^b\}$, b is the machine word length, and $\alpha(z)$ is the bit string representing z.
- $\gamma : \mathcal{O} \to \mathcal{P}^{\mathcal{M}}$. The programs that we provide for each file system operation are the key to achieving SHI. We discuss the HIFS programs in Section 6.5.
- The initial data structure state $s_0^{\mathcal{M}}$ corresponding to the initial file system ADT state is obtained by initializing all file system metadata[1].

6.5 Architecture

6.5.1 Overview

A file system ADT state contains two pieces of information for each file – file data and file metadata. Both files' metadata and files' data need representation in the underlying machine state[2]. In HIFS, separate areas on disk are reserved to store both files' data and files' metadata. Moreover, HIFS ensures that both files' data and files' metadata are stored in a canonical form. Canonical form gives strong history independence (SHI).

To ensure canonical representations, we first select an existing history independent data structure implementation for a hash table ADT (Section 6.5.2). Then, we redesign the hash table implementation to endow it with data locality properties (Section 6.5.3). Finally, we use two instances of the redesigned hash table implementation, one for files' data and one for files' metadata (Sections 6.5.5-6.5.6). File system level metadata is handled separately as discussed in Section 6.5.4.

HIFS closely resembles existing Linux file systems such as Ext3 [5] exposing the exact same API and utilizing a similar disk structure (Figure 7.1). The key difference is that unlike Ext3, the allocation of disk blocks to files is not based on history. Hence, HIFS does not use indirect and double indirect blocks to map file blocks to disk blocks. Instead, the entire data blocks section on disk is managed as a history independent data structure (Section 6.5.5).

In the following sections we detail.

[1] File system metadata includes superblock, group descriptors, inode tables, and disk buckets map. The loading of file system programs, and memory management, is done by the operating system.

[2] Machine state is the disk layout.

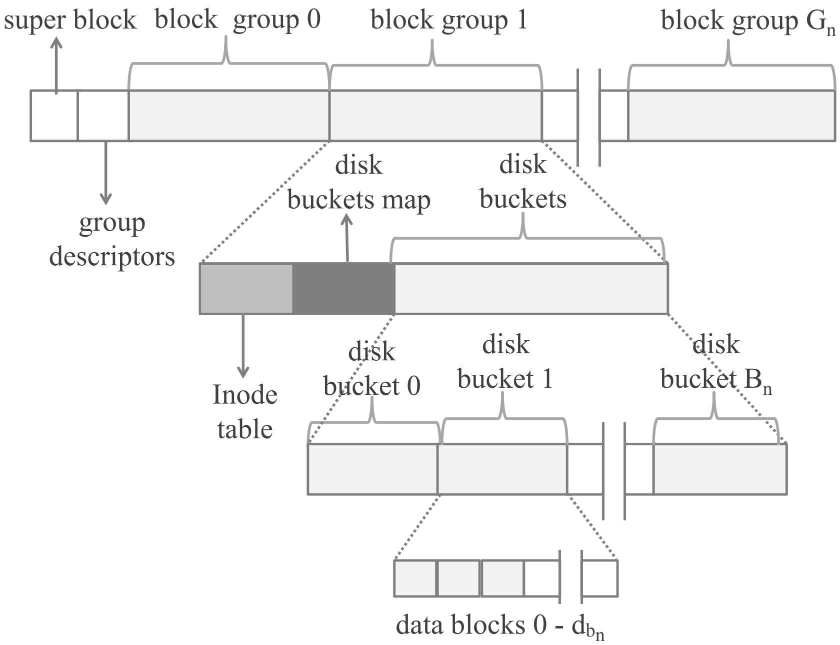


Figure 6.4: *HIFS* disk layout. Key parameters: $\mathsf{G_n}$ ← number of block groups, $\mathsf{B_n}$ ← number of disk buckets per block group, $\mathsf{d_s}$ ← Data block size in bytes, $\mathsf{d_{b_n}}$ ← number of data blocks per disk bucket.

6.5.2 History Independent Hash Table [38]

The key feature of HIFS is the replacement of all file system disk structures with history independent versions that we then endow with data locality properties. The data structure of choice here is the history independent hash table designed by Golovin et al. [38]. Hence, first we describe the hash table construction and in subsequent sections illustrate its transformation and use in various HIFS components.

Golovin et al. designed a history independent hash table based on the stable matching property of the *Gale-Shapley Stable Marriage* algorithm [97] detailed in the following.

Stable Marriage Algorithm

Let M and W be a set of men and women respectively, $|M| = |W| = n$. Also, let each man in M rank all women in W as per his set of preferences. Similarly, each women in W ranks all men in M.

The goal of the stable marriage algorithm is to create n matchings (m, w), where $m \in M$ and $w \in W$, such that no two distinct pairs (m_i, w_j) and (m_k, w_l) exist where m_i ranks w_l higher than w_j and w_l ranks m_i higher than m_k. If no such pairings exists, then all matchings are considered stable.

The algorithm works as follows. In each round, a man m proposes to one woman at a time based on his ranking of W. If a woman w being proposed to is unmatched, then a new match (m, w) is created. If the woman w is already matched to some other man m', then one of the following two occurs.

1. If w ranks m higher than m', then the match (m', w) is broken and a new match (m, w) is created.

2. If w ranks m lower than m', then m proposes to the next woman based on his rankings.

The algorithm terminates when all men are matched.

Gale et al. [97] show that if all the men propose in decreasing order of their preferences (ranks) then the resulting stable matching is unique. The matchings are unique even if the selection of a man m who gets to propose in each round is arbitrary.

History Independent Hash Table

Golovin et al. [38] use the above unique matching property of the Stable Marriage algorithm to construct a history independent hash table as follows.

1. The set of keys to be inserted are considered as the set of men.

2. The set of hash table buckets are considered as the set of women.

3. Each key has an ordered preference of buckets and vice versa.

4. The preference order of each key is the order in which the buckets are probed for insertion, deletion and search.

5. In case of a collision between two keys, the key which ranks higher on the bucket's preference takes the slot. The lower ranked key is relocated to the next bucket in its preference list.

(1) - (5) ensure that the layout of keys in the hash table is the same irrespective of the sequence of key insertions and deletions, thereby making the hash table history independent.

6.5.3 Key Insights

A simple replacement of all file system structures with the above history independent hash table will suffice to yield a history independent layout of files on disk. However, a simple replacement neither preserves data locality nor gives the flexibility to choose different layouts based on application characteristics both of which are key goals in HIFS design.

A key observation to achieve data locality is the following. In the Stable Marriage algorithm each man in M can rank the n women in W in $n!$ ways, and vice-versa. Hence, several sets of preferences from keys to buckets and buckets to keys are possible. Each preference set results in a distinct history independent hash table instance. Therefore, by changing the preference order of keys and buckets we can control the layout of keys within the hash table.

Procedure Set 1 History Independent Hash Table

Procedure: INSERT
Desc: insert the given key in to the hash table.
Input: Tables $H^{0-m}[n]$, key k

```
$<i, r> \leftarrow$ GET_MOST_PREFERRED_BUCKET($k$)
$c \leftarrow 0$
while $c < (n * (m + 1))$ do
    if $H^r[i]$ is null then
        $H^r[i] \leftarrow k$
        return $<i, r>$
    if BUCKET_PREFERS($i, r, k, H^r[i]$) then
        SWAP($k, H^r[i]$)
    $<i, r> \leftarrow$ GET_NEXT_BUCKET($k, i, r$)
    $c \leftarrow c + 1$
return $<null, null>$   {tables are full}
```

Procedure: SEARCH
Desc: search for the given key in the hash table.
Input: Tables $H^{0-m}[n]$, key k

```
$<i, r> \leftarrow$ GET_MOST_PREFERRED_BUCKET($k$)
$c \leftarrow 0$
while $c < (n * (m + 1))$ AND $H^r[i]$ is not null do
    if k == $H^r[i]$ then
        return $<i, r>$   {key found at $H^r[i]$}
    $<i, r> \leftarrow$ GET_NEXT_BUCKET($k, i, r$)
    $c \leftarrow c + 1$
return $<null, null>$   {key not found}
```

Procedure: DELETE
Desc: delete the given key from the hash table.
Input: Tables $H^{0-m}[n]$, key k

```
$<i, r> \leftarrow$ SEARCH($k$)
while $i$ is not null AND $H^r[i]$ is not null do
    $<j, s> \leftarrow$ GET_NEXT_BUCKET($k, i, r$)
    if $H^s[j]$ is not null AND KEY_PREFERS( $H^s[j], i, j, r, s$) then
        $H^r[i] \leftarrow H^s[j]$
        $k \leftarrow H^s[j]$
        $i \leftarrow j, r \leftarrow s$
```

The reordering of preferences leads to the realization that we can rewrite the hash table algorithms to enable easy-custom selection of history independent layouts with minimal modifications. For customization, we categorize the hash table operations in two procedure sets – a *generic* set and a *customizable* set. The generic procedures implement the overall hash table search, insert, and delete operations.

Generic procedures can be used unaltered for all application scenarios. The customizable procedures determine the specific key and bucket preferences thereby governing the resultant hash table layouts. The generic procedures include INSERT, SEARCH, and DELETE (Procedure Set 1). The customizable procedures include GET_MOST_ PREFERRED_BUCKET, GET_NEXT_BUCKET, BUCKET_PR EFERS, and KEY_PREFERS. We list the scenario specific customizable procedures later in Sections 6.5.5 and 6.5.5 while discussing file system operations.

In summary, the new procedure classification and rewrite enables distinct history independent layouts for the same data set through modifications to the customizable procedures. Moreover, modifications can now be targeted to favor other application characteristics, such as read-only and sequential access (Section 6.5.5).

6.5.4 Disk Layout

Super Block and Block Groups

Similar to Ext3 [5], HIFS divides the disk into block groups. Each block group contains an inode table, a map, and data blocks. The super block contains information about the overall file system, such as the number of block groups and disk block size. Each group descriptor describes one block group. Information stored in a group descriptor includes the inode table size, and location of the disk buckets map.

All parameters governing the disk layout can be set up at file system creation time. Figure 7.1 summarizes the HIFS configuration parameters.

The superblock and group descriptors have fixed sizes and occupy the same locations on disk independent of file contents. Hence, no special history independent designs are needed for the superblock and group descriptors. The disk bucket maps and the inode tables however, play a critical role in history independent file storage. We discuss the maps and inode table in detail below.

6.5.5 File Storage

Disk Buckets

File data is stored in blocks on disk. Blocks are grouped into units. Each unit consists of a fixed number of data blocks. Each unit is termed as a disk bucket (Figure 7.1). Although read and write operations access individual data blocks, space is allocated to files in multiples of disk buckets.

Procedure Set 2 Customizable Procedures for Case A (Block Group Locality) from Section 6.5.5

Procedure: GDB

Desc: get the logical file bucket number from file offset.

Input: file offset f_o : $f_o \in \mathbb{N}$

```
return $\left(\left\lfloor\left\lfloor\frac{f_o}{d_s}\right\rfloor/d_{b_n}\right\rfloor\right)$
```

Procedure: GET_MOST_PREFERRED_BUCKET

Input: key k : k = {file path f_p, file offset f_o}

```
return $<h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n},\ h(f_p) \bmod \mathsf{G_n}>$
```

Procedure: GET_NEXT_BUCKET

Input: key k : $\{f_p, f_o\}$, bucket i, block group r : $(i, r) \in \mathbb{N}$

```
$i \leftarrow (i+1) \bmod \mathsf{B_n}$
if $i == (h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n})$ then
    $r \leftarrow (r+1) \bmod \mathsf{G_n}$, $i \leftarrow h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n}$
return $<i, r>$
```

Procedure: BUCKET_PREFERS

Input: bucket i, block group r : (i,r) $\in \mathbb{N}$, key a : $\{f_{p_a}, f_{o_a}\}$, key b : $\{f_{p_b}, f_{o_b}\}$

```
return $h(f_{p_a}||\text{GDB}(f_{o_a})) > h(f_{p_b}||\text{GDB}(f_{o_b}))$
```

Procedure: KEY_PREFERS

Input: key k : $\{f_p, f_o\}$, bucket i, bucket j, block group r, block group s : (i,j,r,s) $\in \mathbb{N}$

```
if r <>s then
    return $((h(f_p) \bmod \mathsf{G_n})$ - r + $\mathsf{G_n}) \bmod \mathsf{G_n} <$ $((h(f_p) \bmod \mathsf{G_n})$ - s + $\mathsf{G_n}) \bmod \mathsf{G_n}$
return $((h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n})$ - i + $\mathsf{B_n}) \bmod \mathsf{B_n} <$ $((h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n})$ - j + $\mathsf{B_n}) \bmod \mathsf{B_n}$
```

Disk Buckets Map

To allocate new disk buckets to files and to locate disk blocks in read and write operations, HIFS relies on a special region in each block group referred to as the disk buckets map.

Each entry within the disk buckets map has a one-one mapping to the corresponding disk bucket within the same block group (Figure 7.1). The entry in the map contains metadata about the corresponding disk bucket, such as whether the bucket is free or occupied.

All file system operations first locate the target entry in the disk buckets map. Once the map entry is located, the actual read or write operation performed on the corresponding disk bucket. The size of a disk buckets map entry is much smaller than the size of the corresponding disk bucket. Hence, using the map to locate disk buckets avoids expensive seek operations on actual file contents. Also, the significantly smaller size of the maps means

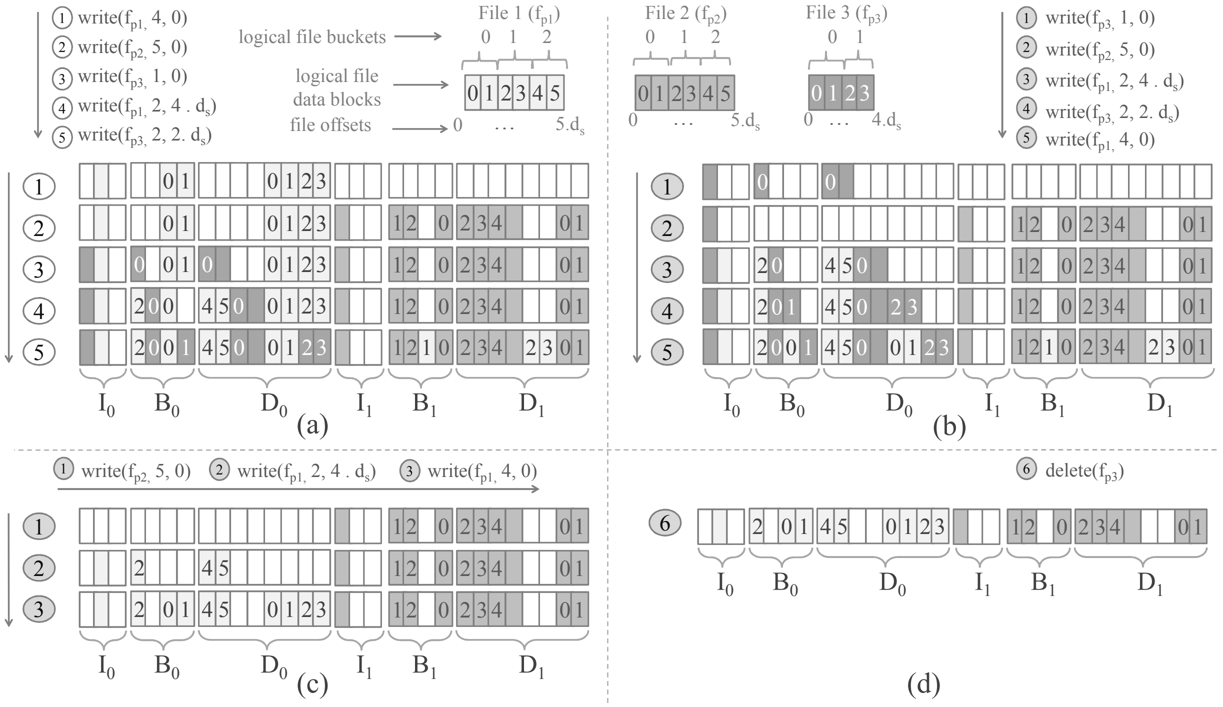


Figure 6.5: Sample executions and corresponding disk layouts for Case A from Section 6.5.5. Here $\mathsf{G}_\mathsf{n} = 2$, $\mathsf{B}_\mathsf{n} = 4$, $\mathsf{d}_{\mathsf{b}_\mathsf{n}} = 2$. I_i, B_i and D_i denote the inode table, disk buckets map, and the disk blocks respectively of block group i. Also, $h(f_{p_1}) = 2$, $h(f_{p_2}) = 3$ and $h(f_{p_3}) = 4$. $write(f_p, f_d, f_o)$ represents a write operation of f_d blocks on file f_p at offset f_o. The disk blocks occupied by files with paths f_{p1}, f_{p2} and f_{p3} are shaded with their respective colors. Blocks that are not shaded indicate free blocks. The number within a block represents the data block number of the corresponding file.

that map entries are often cached in memory for faster access.

History Independent Layouts

Existing file systems such as Ext3 [5] maintain a list of allocated blocks within the file inode which renders the disk space allocation history dependent. HIFS on the other hand, does not rely on allocation lists. Instead, in HIFS, location of data blocks are derived directly from file attributes. Thus, for each read or write operation in HIFS, the locations of data blocks on disk are determined only by parameters to the current operation and do not depend on history.

To ensure history independent file storage, the disk bucket maps from all block groups are collectively treated as a single history independent hash table. Hash table keys are derived from file attributes such as the file paths and read-write offsets. The entries in the maps are the hash table buckets.

As discussed in Section 6.5.3, by altering the derivation of keys and the keys↔buckets preference sets we can attain customized layouts of the hash table to suit different locality scenarios. Hence, file system operations use the generic hash table procedures from Procedure

Set 1 to locate free blocks for allocation or to find existing blocks for reading or writing. Different history independent disk layouts are realized solely by altering the customizable procedure set. Thus, file system operations, such as read and write have implementations independent of the underlying history independent layouts. The history independent layouts, in turn, are governed by specific customizable procedure implementations. To illustrate, the actual file system write operation is included in Procedure Set 7 (Appendix).

We now describe a specific history independent data locality scenario in detail and then briefly discuss other scenarios.

Case A: Block Group Locality

For block group locality, it is desired that data blocks of the same file are located close together on disk, ideally within the same block group. To realize block group locality, we tailor the customizable procedures as shown in Procedure Set 2. The generic history independent hash table procedures from Set 1, and the generic file system operations are unchanged.

To clarify the implementation of history independent block group locality in HIFS, consider the example in Figure 6.5. The example illustrates history independent space allocation for a set of three files. Each of the Figures 6.5(a)-6.5(d) lists a sequence of file system operations and depicts the resultant disk layout at the end of each operation. The depicted disk layouts include files' data and metadata. Note that although the sequence of operations in Figures 6.5(a) and 6.5(b) differ, the resultant disk layouts at the end of either operation sequence are exactly the same. The example thus illustrates canonical SHI file storage.

To achieve canonical disk layouts, file system operations are translated in to hash table operations as follows:

1. Keys are derived from the full file path f_p and the read or write offset f_o.

2. The hash table buckets are the disk buckets map entries.

3. Key preferences are set such that each key first prefers all buckets from one specific block group in a fixed order. Then, buckets from the next adjacent block group and so on.

4. Finally, buckets simply prefer keys with higher numerical values.

We now explain how steps (1) - (4) above preserve locality and history independence.

Consider the file system write operation listed in Procedure Set 7 in the chapter Appendix. The write operation first needs to locate the correct entry in the disk buckets maps. Once the map entry is located, the actual file write will be performed on the corresponding disk bucket. Since the disk buckets maps from all block groups are treated as a single history independent hash table, locating the correct buckets map entry is equivalent to finding the corresponding hash table bucket. Hence, locating the disk buckets map entry requires probing of the disk buckets maps. The probe order is exactly what is determined by the key preferences (Step 3).

Block Group Locality. The probe order for a write operation starts with the most preferred bucket as determined by the procedure GET_MOST_PREFERRED_BUCKET (Procedure Set 2). For a given file f_p, the block group of the most preferred bucket for all file buckets are derived from only the file path's hash $h(f_p)$ (line 1, second return parameter). Hence, all write operations of the same file begin their probe from the same block group independent of the target file offsets. Although the probing for two write operations that target different offsets may start from two different buckets but both buckets will be located within the same block group.

Subsequent map entries in the probe order are determined using the GET_NEXT_BUCKET procedure. The GET_NEXT_BUCKET procedure ensures that all buckets in the current block group are probed (line 1) before moving to the next adjacent block group (lines 2-4). Hence, data blocks of the same file prefer to be located in the same block group preserving locality.

History Independence. For each bucket map entry in the probe sequence, following two cases are possible.

(a) The bucket map entry is free.

(b) The bucket map entry is occupied by a previous write.

For case (a), data is written to the corresponding disk bucket and the key is stored in the bucket map entry. The bucket map entry is also marked as occupied. In case (b), a collision has occurred. Now it is exactly the collision resolution process that gives HIFS its history independence.

We clarify the collision resolution through an illustration. Let w_1 and w_2 be two write operations on files f_{p_1} and f_{p_2}, respectively. Also, let w_1 target offset f_{o_1} of f_{p_1} and w_2 target offset f_{o_2} of f_{p_2}. Hence, the keys for w_1 and w_2 for probing the disk bucket maps are $h(f_{p_1}||\text{GDB}(f_{o_1}))$ and $h(f_{p_2}||\text{GDB}(f_{o_2}))$, respectively (procedure GET_NEXT_BUC KET, line 2). In addition, let $h(f_{p_1}||\text{GDB}\ (f_{o_1})) > h(f_{p_2}||\text{GDB}(f_{o_2}))$. Finally, suppose that the keys for both w_1 and w_2 prefer the same entry in the disk bucket map of the same block group and hence result in a collision. Now, there are two possible write sequences for w_1 and w_2.

(i) w_1 *occurs before* w_2*:* The bucket is already occupied by the key of w_1 when w_2 executes. Also, since $h(f_{p_1}||\text{GDB}(f_{o_1})) > h(f_{p_2}||\text{GDB}(f_{o_2}))$, the bucket prefers key of w_1 more than that of w_2 (procedure BUCKET_PREFERS, line 1). Hence, w_2 looks to the next bucket in the key's preference list and the bucket map entry remains unchanged.

(ii) w_2 *occurs before* w_1*:* The bucket entry is already occupied by the key of w_2 when w_1 executes. The bucket however, prefers the key of w_1 over that of w_2. Hence, the key of w_2 is now evicted from the bucket entry and is replaced by w_1's key. The key of w_2 is relocated to a new bucket based on its preference list and probe order. Also, in this case, the data written by w_1 is now placed in the corresponding disk bucket, while the data previously written by w_2 is moved to the new disk bucket corresponding to the new bucket map entry determined for its key's relocation.

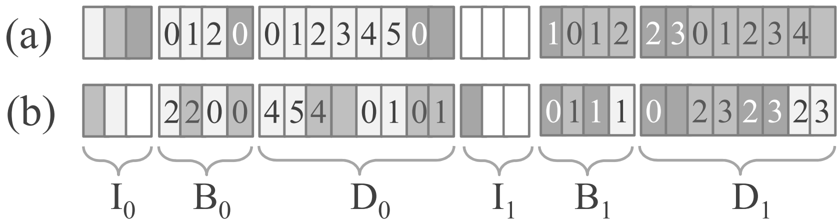


Figure 6.6: *HIFS* disk layouts for operation sequences of Figure 6.5. (a) Case B ← Complete Sequential and (b) Case C ← External Parameters, from Section 6.5.5. Also, $h(user(f_{p_1}))$ = 2, $h(user(f_{p_2})) = 4$ and $h(user(f_{p_3})) = 3$.

Figure 6.5 lists several examples of the two cases (i) and (ii) from above. Specifically, consider execution of operation 5 of Figure 6.5(b). In operation 5, the write of data blocks zero and one of file f_{p_1} replaces and relocates the data blocks two and three of f_{p_3} which were previously written by operation 4.

As a result of the collision resolution process, the bucket entry and hence the corresponding disk bucket contents are the same irrespective of the write sequence. Generalizing the above example gives the following. For a given set of files F, a file f in F is always stored at the exact same locations on disk irrespective of the operation sequence that creates F. That is, HIFS supports canonical representations and is history independent as per SHI definition in Section 5.4.2.

Customizing History Independence

Block group locality is one specific scheme for a history independent disk layouts for a given set of files. Different applications may prefer different locality properties. Hence, we list two additional locality scenarios that can be achieved using different implementations of the customizable procedures. The two scenarios are complete sequential locality and locality based on external parameters. The relevant customizable procedures for the two scenarios are listed in Procedure Sets 4 and 5 (Appendix).

Case B: Completely Sequential

Applications, such as data mining have very few writes as compared to read operations. Read-intensive applications can greatly benefit if the entire file is stored sequentially on disk giving maximal locality. To realize sequential file layouts, only the key and bucket preferences need to be modified. The keys of all write operations prefer buckets within the same block group as in block group locality. However, the probe sequence starts with the first bucket in the block group and probes linearly henceforth. The buckets in-turn prefer blocks in increasing order of file offsets.

Case C: External Parameters

Many applications access multiple files simultaneously [126]. I/O performance of multi-file applications can be greatly enhanced if the files are located close together on disk. For example, files created by a single user or process are located in adjacency within the same block group. In HIFS, multi-file locality can be realized with very minimal changes. In fact, the only change required is that key's preferences for block groups are now based on external parameters such as the user (Procedure Set 5 in Appendix).

Figure 6.6 gives the resultant layouts for both complete sequential and locality based on external parameters. The layouts result from the operation sequences of Figures 6.5(a) and 6.5(b). For all locality scenarios, resultant disk layouts will be the same irrespective of the operation sequence.

Several other locality scenarios are possible. Individual scenarios can also be combined to create more complex ones. For example, cases B and C can be combined to have a distribution wherein files from the same user are located adjacently on disk with each file laid out sequentially. A new scenario is realized by providing an appropriate set of customizable procedures. A new set of customizable procedures needs to ensure that the key↔buckets preferences are unambiguous, that is, no key should prefer any two buckets equally and vice-versa. In Section 6.5.8, we show that as long as unambiguity of preferences is maintained, the resulting layouts will be history independent.

6.5.6 Inode Table

Each inode is of fixed size and contains file metadata such as the file type, access rights, and file size. Each inode table contains a fixed number of inodes. Inodes are allocated to files at creation time.

Similar to the disk buckets maps, the inode tables from all block groups are collectively treated as a single history independent hash table. Then, inode allocation and search is done using the generic procedures from Procedure Set 1. History independent layouts for the inode tables are also determined by the customizable procedures. One such set of inode-specific customizable procedures are listed in Procedure Set 6 (Appendix). Here, the file inode is located in the same block group as that preferred by the file data blocks. The intuition and reasoning for history independence is similar to that described for block group locality.

6.5.7 File Delete and Rename Operations

HIFS hides all evidence of a file delete operation. To illustrate, consider the sample executions and disk layouts from Figures 6.5(c) and 6.5(d). Figure 6.5(c) shows the resulting layouts of a sequence of write-only operations. Figure 6.5(d) shows the layout after execution of all operations of 6.5(b) plus the delete operation on file f_{p_3}.

The disk layout after the delete operation (Figure 6.5(d)) is exactly the same as the layout after the write-only sequence of Figure 6.5(c). Same layouts occur since HIFS relocates data blocks on delete to their more preferred locations (procedure DELETE in Set 1). For example, in Figure 6.5(d) blocks 2 and 3 of file f_{p_1} are relocated to more preferred locations when file f_{p_3} is deleted by operation 6. Overall, when a file is deleted, the resultant disk layout carries no traces of the delete operation as if the file was never created in the first place.

A file delete is realized by execution of the delete procedure from Set 1 for each disk bucket allocated to the file. Thus, a delete operation is cost-wise equivalent to a write of the entire file. A rename or move operation for a particular file is a delete of each allocated disk bucket followed by a write of the same bucket with the new file path.

6.5.8 Proofs of History Independence

Let K denote the set of keys and β denote the set of buckets within the hash table. Also, let $k.pref(b)$ denote bucket b's position on key k's preference list, where $k \in K$ and $b \in \beta$. Lower value of $k.pref$ indicates higher preference. Then, we have the following definition

Definition 12. Unambiguous Preferences
A set of preferences from $K \to \beta$ and $\beta \to K$ are unambiguous *if both conditions (a) and (b) below are met*

(a) $\forall k \in K$, $\nexists\ b_i, b_j \in \beta$, *such that* $i \neq j$ *and* $k.pref(b_i) = k.pref(b_j)$

(b) $\forall b \in \beta$, $\nexists\ k_i, k_j \in K$, *such that* $i \neq j$ *and* $b.pref(k_i) = b.pref(k_j)$.

Theorem 3. *The customizable procedures listed in Procedure Set 2 result in a history independent hash table with unique representation.*

Proof. According to the Gale-Shapley Stable Marriage algorithm [97], if men propose in decreasing order of their preferences and if all preferences are unambiguous, then the resulting stable matching is unique. The key preferences of the history independent hash table [38] are decreasing and unambiguous. Golovin et al. [38] proved that as a result of decreasing and unambiguous preferences, the hash table layout is canonical, that is, the hash table is strongly history independent. Therefore, to show that the layouts produced by a given set of customizable procedures are history independent, we only need to show that the customizable procedures result in decreasing and unambiguous preferences.

In the following we show that the customizable procedures in Procedure Set 2 applicable to block group locality (Section 6.5.5) produce decreasing and unambiguous preferences.

(a) Proposal Order

Each time a new key is inserted, the first attempt is to place the key in its most preferred bucket, that is, bucket b where $k.pref(b)$ is minimum. The hash table insert procedure (Set 1) uses the GET_MOST_PREFERRED_BUCKET customizable procedure to locate a key's most preferred bucket. If further probing is needed, each new bucket in the probe sequence is selected by the procedure GET_NEXT_BUCKET. Given a bucket b_i, procedure GET_NEXT_BUCKET finds the bucket b_j, such that $\nexists\ b_l$ where $k.pref(b_i) < k.pref(b_l) < k.pref(b_j)$. In other words, b_j is always the next preferred bucket on key k's preference list. Thus, analogous to men in the stable marriage algorithm, the matching of keys to buckets is attempted in decreasing order of key preferences.

(b) Unambiguous preferences

A bucket's preference between two keys is resolved by the customizable procedure BUCKET_PREFERS. The condition for resolution is $h(f_{p_a}||\text{GDB}(f_{o_a})) > h(f_{p_b}||\text{GDB}(f_{o_b}))$. If the condition is true, key $h(f_{p_a}||\text{GDB}(f_{o_a}))$ is preferred. If the condition is false, key $h(f_{p_b}||\text{GDB}(f_{o_b}))$ is preferred. Since the comparison operator is $>$, the comparison is always unambiguous for two distinct keys.

For any given set of files F, the combination of the file path and the logical file bucket number is unique, that is, $\forall f \in F$, $<f_p$,GDB$(f_o)>$is unique. Hence, the hash value $h(f_p||\text{GDB}(f_o))$ is unique. The proof then reduces to collision resistance of the hash function.

Similarly, a key's preference amongst two buckets is resolved by the customizable procedure KEY_PREFERS. If the two buckets belong to separate block groups (line 1), then the key simply prefers the bucket in the block group with the lower index (line 2). Hence, the case for separate block groups is unambiguous.

If the two buckets are in the same block group, then the key prefers the bucket closer to its most preferred bucket in that block group (line 3). Again, since the comparison is based on the hash of the unique combination $<f_p$,GDB$(f_o)>$, the comparison is unambiguous. The proof again reduces to collision resistance of the hash function. □

Similar proofs exist for the customizable procedures for complete sequential (Case B) and for locality based on external parameters (Case C) from Section 6.5.5. Also, using the same approach, history independence can be proved for the inode table.

6.6 Discussion

6.6.1 The Role of Secure Deletion

Under secure deletion, data is physically deleted from the storage medium via overwriting [219]. The number of overwrites needed depends on the storage medium being used.

Secure deletion plays a role in history independence to avoid direct retention of deleted data. When data is deleted from a history independent implementation, it is imperative that the data location on disk is cleared by overwriting. Any residual data artifacts can compromise the history of operations. Hence, HIFS does not mark hash table entries as deleted but immediately performs an overwrite on delete. Overwriting applies to all history independent disk structures including the inode table, disk bucket maps, and disk buckets.

6.6.2 Temporal Metadata

File systems typically maintain temporal metadata, such as modification times for files and directories. The temporal metadata are then made available to applications. For certain composite applications, it may be desired that history independence is preserved across files. Cross-file history independence implies that file creation and modification order must be

Parameter	**Database profile**	**Web Server profile**
File system size	100 GB	10 GB
Mean file size	1 GB	512 KB
No. of files	$L \cdot 100$	$(L \cdot 10) \cdot 2^{11}$
Disk block size ($\mathsf{d_s}$)	4 KB	4 KB
No. of block groups ($\mathsf{G_n}$)	8	24
Disk blocks / bucket ($\mathsf{d_{b_n}}$)	5120	128
Inode size	281 bytes	281 bytes
IO Size	32KB	512KB

Table 6.1: Experimental parameters. L ← File System Load factor.

hidden. If configured, HIFS achieves cross-file history independence by not maintaining any temporal metadata for files and by using history independent directories.

A directory file consists of a set of directory entries one entry for each file in the directory. If cross-file history independence is desired, then HIFS maintains the contents of each directory file in a history independent manner. To ensure history independence, no additional data structures are used. Instead, the directory entries are always stored sorted. Since a simple sorted list is history independent [184], the sorting of directory contents suffices.

6.6.3 In-Memory History Independence

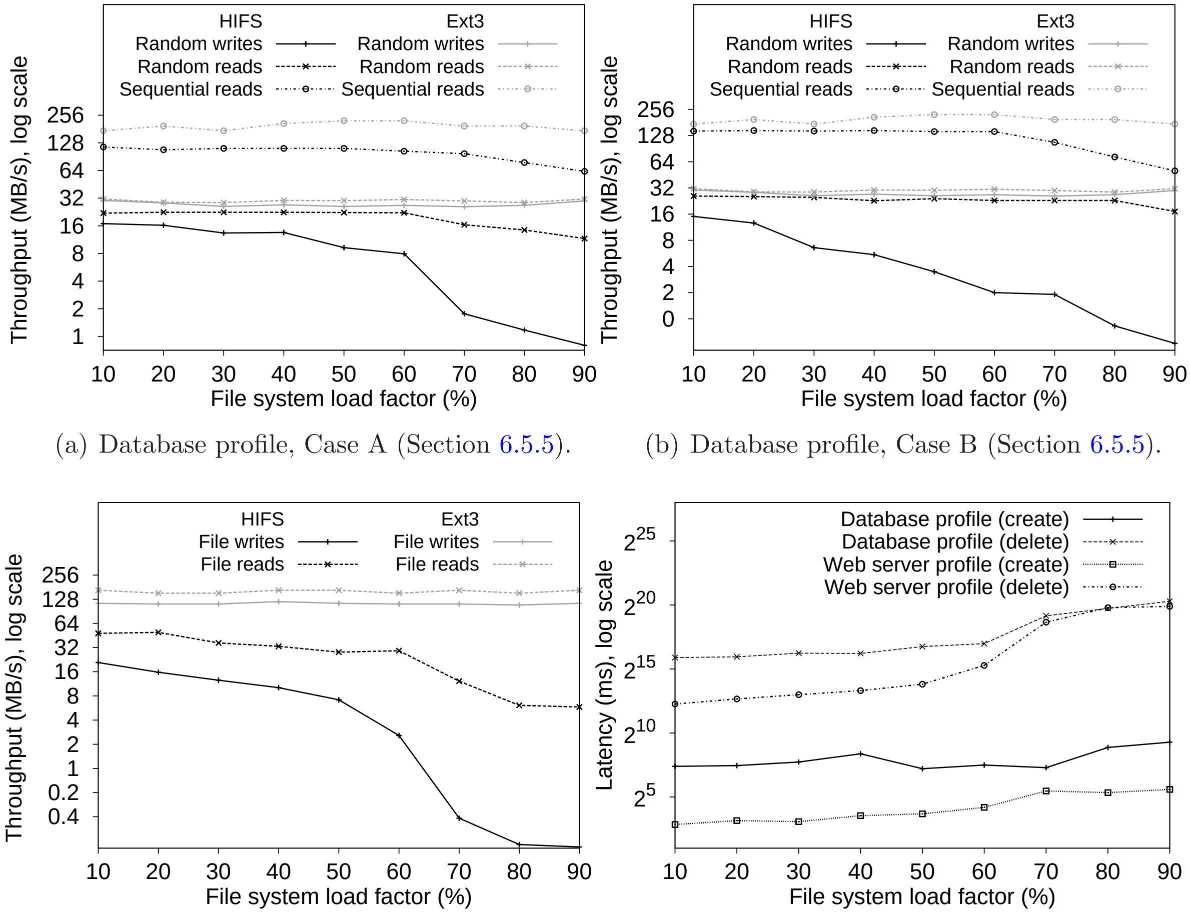
HIFS protects history independence only for data residing on disk. HIFS does not target in-memory structures such as the file system cache.

Although the need for a history independent memory allocator for in-memory data structures has been voiced [113], we posit that extension of history independence to data in memory requires further careful examination. Simply replacing in-memory caches with history independent versions will not suffice.

An in-memory system cache cannot treat each disk block independently. Instead, interblock associations need to be maintained. For example, the cache must avoid scenarios, such as the case wherein disk buckets map is written to disk but the modified file data blocks still reside in the cache. Discrepancy between cache and disk can potentially reveal the last file system operation performed.

6.7 Experiments

We benchmarked HIFS to understand the impact of history independence on performance.



(a) Database profile, Case A (Section 6.5.5).

(b) Database profile, Case B (Section 6.5.5).

(c) Web Server profile, Case A (Section 6.5.5).

(d) HIFS latencies for file create and delete operations for Case A (Section 6.5.5).

Figure 6.7: HIFS throughputs and latencies. A load factor of L indicates that the file system is $L\%$ full.

6.7.1 Setup

All experiments were conducted on servers with 8 Intel i7 CPUs at 3.4GHz, 16GB RAM, and kernel v3.2.0-37. The storage devices of choice are Hitachi HDS72302 SCSI drives. The benchmark tool used is Filebench [95].

6.7.2 Implementation

HIFS is implemented as a C++ based user-space Fuse [246] file system. All data structures, including customizable procedures for history independence were written from scratch. The entire HIFS code is ≈10K LOC.

6.7.3 Measurements

Each test run commences with an empty file system, then creates and writes new files to storage. The number of files stored is increased until the file system is 90% full. Throughputs are measured at specific load factors[3] ranging from 10% to 90%. To minimize the effect of caching, the writes and subsequent read operations are separated by a complete clearing of the system cache.

6.7.4 Results

Database Profile. Databases typically feature access to a few large files with random access patterns. To simulate a database profile, we evaluate random reads and writes on multiple files with a mean file size of 1 GB. The number of files is varied from 10 to 90 to reflect the file system load factor. Table 6.1 summarizes all file system parameters while Figure 6.7(a) shows the results.

As detailed in Section 6.5.5, the allocation of a new disk bucket to a file potentially causes the displacement of other files' data due to the history independent collision resolution. The number of collisions increases with the load factor. For load factors beyond 60% a sharp decrease in throughputs is observed for random writes. Read operations do not modify data. As a result, read operations are not signifcantly affected by collisions showing less declines in throughput with increasing load factors. Also, since the customizable procedures for block group locality (Case A from Section 6.5.5) are employed here, data locality is preserved. Write operations incur the overheads to maintain locality while reads benefit from colocated data.

The tradeoff between write and read performance is further evident from the results shown in Figure 6.7(b). Figure 6.7(b) lists file system throughputs when customizable procedures for complete sequential locality (Case B in Section 6.5.5) are used. In the case of complete sequential storage, each file is laid out sequentially on disk. For the sequential scenario, read operations show a 13% average increase in throughput for sequential reads and 17% for random reads as compared to block group locality (Figure 6.7(a)). To maintain higher locality for sequential storage, write throughputs incur an average decline of 90% compared to block group locality (Case A).

All of the above results hold under history independence assurances.

Web Server Profile

Web servers are characterized by access to a large number of very small files [247]. To model a web server profile, we use a mean file size of 512 KB but increase the number of files up to ≈18,500 for a load factor of 90%. Refer to Table 6.1 for a full parameter list. The read and write operations access entire files. Figure 6.7(c) summarizes the results for the web server profile.

[3] Load factor is the file system disk space utilization.

File create and delete operations

Figure 6.7(d) shows the latencies of file create and delete operations.

A create operation involves locating a free inode for the new file, writing the file metadata to the new inode, locating the parent directory, and writing the file entry to the parent directory. The writes to the inode table and to the directory are both history independent. Since the inode table employs a history independent hash table (Section 6.5.6), create latency increases with the load factor.

As discussed in Section 6.5.7, a file delete is equivalent to a write of the entire file in order to preserve history independence. As a result, deletes have high latencies.

Overhead of history independence

Figures 6.7(a) and 6.7(c) also illustrate the throughputs for the Ext3 file system. For consistent comparisons all Ext3 file system operations are also routed via Fuse.

The performance of HIFS for read operations is comparable to read throughputs of Ext3 for load factors up to 70%. To maintain history independence with locality, write operations sustain significant overheads especially for higher load factors. Once the write operations provide locality, reads incur significantly lower disk seek operations and are hence efficient.

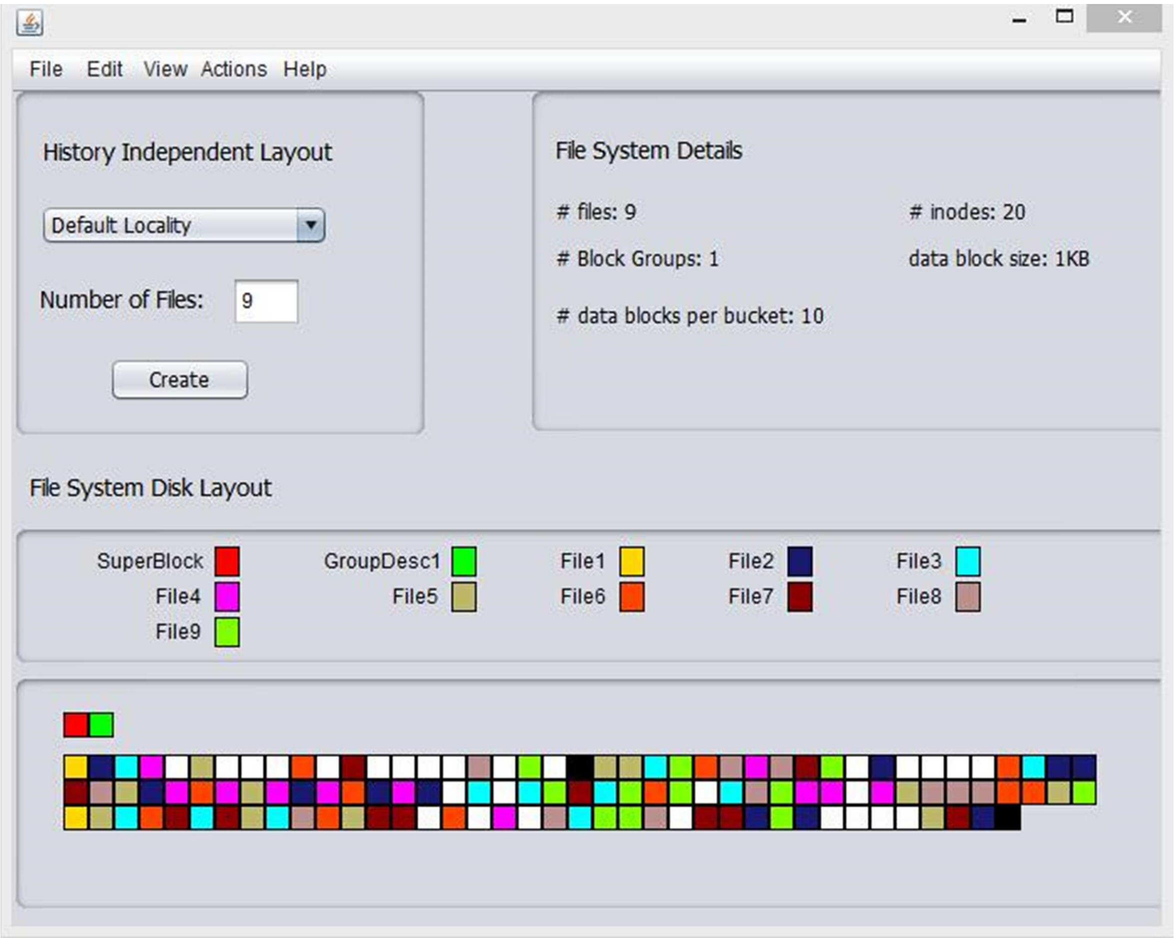
For the web server profile, the overhead of history independence is higher even for reads. Each operation in the web server profile accesses the entire file at once. Although the locality of data blocks withing a file is maintained the files themselves are distributed over the entire disk.

6.7.5 Summary and Analysis

HIFS employs history independent hash tables for all its data structures including files' metadata and files' data. The performance of hash tables in turn depends on the load factor. In fact, the asymptotic performance per operation of the history independent hash table employed is $O(1/(1-\alpha)^3)$ [38]. The asymptotic result shows exponential decrease in performance with increasing load factor α. Hence, for load factors >60% the performance degrades for writes are significant. The same behavior is clearly evident in the experimental results for HIFS (Figures 6.7(a)-6.7(d)).

6.8 HIFS Visualizer (Demo)

The HIFSVisualizer (Figure 6.8) aids in the demonstration and understanding of history independent storage in HIFS. For a given locality scenario and file set, the HIFSVisualizer depicts the entire disc layout. Users can also select different locality scenarios and see how changes in the file creation sequence result in the same canonical, history independent layouts.



History Independent Layout

Default Locality

Number of Files: 9

File System Details

files: 9

inodes: 20

Block Groups: 1

data block size: 1KB

data blocks per bucket: 10

File System Disk Layout

Figure 6.8: HIFS Visualizer.

6.9 Conclusions

In this chapter, we approached history independence from a systems perspective bridging the gap between theory and practice. We used our theoretical concepts from Chapter 5 to design a history independent file system (HIFS). Once data is deleted, HIFS guarantees that no evidence of the deleted data is recoverable via data organization. HIFS provides full history independence across both file system and disk layers of the storage stack. Additionally, HIFS preserves data locality and is configurable to suit different data locality scenarios.

6.10 Chapter Appendix

6.11 Procedure Sets

Procedure Set 3 Customizable Procedures for HIHT Hash Table by Golovin et al. [38]

Procedure: GET_MOST_PREFERRED_BUCKET
Desc: get the top most bucket on key's preference list.

Input: key k : $k \in \mathbb{N}$

```
return $<k \bmod n, 0>$
```

Procedure: GET_NEXT_BUCKET
Desc: get the next bucket on key's preference list.

Input: key k, bucket i, table r : $(k, i, r) \in \mathbb{N}$

```
$i \leftarrow (i + 1) \bmod n$
if $i == (k \bmod n)$ then
    $r \leftarrow r + 1$
    $i \leftarrow k \bmod n$
return $<i, r>$
```

Procedure: BUCKET_PREFERS
Desc: find which of two keys the given bucket prefers.

Input: bucket i, table r, key a, key b : $(i, r, a, b) \in \mathbb{N}$

```
return $a > b$
```

Procedure: KEY_PREFERS
Desc: find which of two buckets the given key prefers.

Input: key k, bucket i, bucket j, table r, table s : $(k, i, j, r, s) \in \mathbb{N}$

```
if $r \neq s$ then
    return $r < s$
return $((k \bmod n) - i + n) \bmod n < ((k \bmod n) - j + n) \bmod n$
```

Procedure Set 4 Customizable Procedures for Case B (Complete Sequential) from Section 6.5.5

Procedure: GET_MOST_PREFERRED_BUCKET
Desc: get the top most bucket on key's preference list.
Input: key $k = \{$file path f_p, file offset $f_o\}$

```
return  $<0, h(f_p) \bmod \mathsf{G_n}>$
```

Procedure: GET_NEXT_BUCKET
Desc: get the next bucket on key's preference list.
Input: key $k = \{f_p, f_o\}$, bucket i, block group r

```
$i \leftarrow (i+1) \bmod \mathsf{B_n}$
if $i == 0$ then
    return  $<0, (r+1) \bmod \mathsf{G_n}>$
return  $<i, r>$
```

Procedure: BUCKET_PREFERS
Desc: find which of two keys the given bucket prefers.
Input: bucket i, block group r, key $a = \{f_{p_a}, f_{o_a}\}$, key $b = \{f_{p_b}, f_{o_b}\}$

```
if $f_{p_a} == f_{p_b}$ then
    return  GDB($f_{o_a}$) <GDB($f_{o_b}$)
return  $h(f_{p_a}) > h(f_{p_b})$
```

Procedure: KEY_PREFERS
Desc: find which of two buckets the given key prefers.
Input: key $k = \{f_p, f_o\}$, bucket i, bucket j, block group r, block group s

```
if $r \neq s$ then
    return  $((h(f_p) \bmod \mathsf{G_n}) - r + \mathsf{G_n}) \bmod \mathsf{G_n} < ((h(f_p) \bmod \mathsf{G_n}) - s + \mathsf{G_n}) \bmod \mathsf{G_n}$
return  $i < j$
```

Procedure Set 5 Customizable Procedures for Case C (External parameters) from Section 6.5.5

```
Procedure: GET_MOST_PREFERRED_BUCKET
Input: key $k : k = \{$file path $f_p$, file offset $f_o$, user $u\}$
return $<h(f_p||\mathrm{GDB}(f_o)) \bmod \mathsf{B_n}, h(u) \bmod \mathsf{G_n}>$
```

GET_NEXT_BUCKET and BUCKET_PREFERS same as in procedure set 2.

```
Procedure: KEY_PREFERS
Input: key $k : \{f_p, f_o, u\}$, bucket $i$, bucket $j$, block group $r$, block group $s : (i, j, r, s) \in \mathbb{N}$
if $r \neq s$ then
    return $((h(u) \bmod \mathsf{G_n}) - r + \mathsf{G_n}) \bmod \mathsf{G_n} < ((h(u) mod \mathsf{G_n}) - s + \mathsf{G_n}) \bmod \mathsf{G_n}$
return $((h(f_p||\mathrm{GDB}(f_o)) \bmod \mathsf{B_n}) - i + \mathsf{B_n}) \bmod \mathsf{B_n} < ((h(f_p||\mathrm{GDB}(f_o)) \bmod \mathsf{B_n}) - j + \mathsf{B_n}) \bmod \mathsf{B_n}$
```

Procedure Set 6 Inode Table Customizable Procedures

```
Procedure: GET_MOST_PREFERRED_BUCKET
Input: key $k =$ file path $f_p$
return $<h(f_p) \bmod \mathsf{I_n}, h(f_p) \bmod \mathsf{G_n}>$
```

```
Procedure: GET_NEXT_BUCKET
Input: key $k = f_p$, bucket $i$, block group $r$
$i \leftarrow (i + 1) \bmod \mathsf{I_n}$
if $i == (h(f_p) \bmod \mathsf{I_n})$ then
    $r \leftarrow (r + 1) \bmod \mathsf{G_n}$, $i \leftarrow h(f_p) \bmod \mathsf{I_n}$
return $<i, r>$
```

```
Procedure: BUCKET_PREFERS
Input: bucket $i$, block group $r$, key $a = f_{p_a}$, key $b = f_{p_b}$
return $h(f_{p_a}) > h(f_{p_b})$
```

```
Procedure: KEY_PREFERS
Input: key $k = f_p$, bucket $i$, bucket $j$, block group $r$, block group $s$
if $r \neq s$ then
    return $((h(f_p) \bmod \mathsf{G_n}) - r + \mathsf{G_n}) \bmod \mathsf{G_n} < ((h(f_p) \bmod \mathsf{G_n}) - s + \mathsf{G_n}) \bmod \mathsf{G_n}$
return $((h(f_p) \bmod \mathsf{I_n}) - i + \mathsf{I_n}) \bmod \mathsf{I_n} < ((h(f_p) \bmod \mathsf{I_n}) - j + \mathsf{I_n}) \bmod \mathsf{I_n}$
```

Procedure Set 7 *HIFS* write operation

```
**Procedure:** write
**Input:** file path $f_p$, file offset $f_o$, data $\partial$, data length $\partial_l$
    $\mathsf{I}_{\mathsf{t}}^{\mathsf{G_n}}$ : Inode tables, $\mathsf{B}_{\mathsf{t}}^{\mathsf{G_n}}$ : Disk Bucket Maps, $\mathsf{GD_t}$ : Group Descriptor table
$<i_i, i_n> \leftarrow$ SEARCH($\mathsf{I}_{\mathsf{t}}^{\mathsf{G_n}}$, $f_p$)
check file permissions in inode data
$b_s \leftarrow (\mathsf{d}_{\mathsf{b_n}} * \mathsf{d_s})$, $w_l \leftarrow 0$
**while** $w_l < \partial_l$ **do**
    $<b_i, g_i> \leftarrow$ SEARCH($\mathsf{B}_{\mathsf{t}}^{\mathsf{G_n}}$, $\{f_p, f_o\}$)
    **if** $b_i$ is null **then**
        $<b_i, g_i> \leftarrow$ INSERT($\mathsf{B}_{\mathsf{t}}^{\mathsf{G_n}}$, $\{f_p, f_o\}, \{\}$)
    **if** $(\partial_l - w_l) > (b_s - (f_o \bmod b_s))$ **then**
        $\partial_i \leftarrow b_s - (f_o \bmod b_s)$
    **else**
        $\partial_i \leftarrow \partial_l - w_l$
    $s_d \leftarrow$ start offset of disk buckets in $\mathsf{GD}_t[g_i]$
    write $\partial[w_l : w_l + \partial_i]$ bytes to disk at offset $s_d + (b_s * b_i) + (f_o \bmod b_s)$
    $f_o \leftarrow f_o + \partial_i$, $w_l \leftarrow w_l + \partial_i$
```

Chapter 7

Delete Agnostic File System (DAFS): Journaling and OAHI

7.1 Chapter Overview

7.1.1 Background and Motivation

The HIFS implementation from Chapter 6 supports strong history independence (SHI). As seen from the experimental results in Section 6.7, for higher file system load factors[1], HIFS write efficiency is low. Lower write efficiency results from the necessity of canonical representations for SHI. To ensure canonical representations for files' data and metadata, HIFS relocates data on each write operation. The amount of data relocated increases exponentially with the file system load factor. Hence, the write throughputs are significantly lower for load factors greater than 60%.

SHI is a very powerful notion of history independence, secure even against a computationally unbounded adversary [113]. SHI also conceals the maximum possible history. In practice, hiding partial history may be sufficient for certain applications. Applications that do not require SHI can be made highly efficient using new targeted history independence notions. It is therefore important to explore the efficiency benefits of new history independence notions that are weaker that SHI and potentially more suitable for practical use.

7.1.2 Our Contribution: Delete Agnostic File System (DAFS)

In Section 5.5.1, using ΔHI we defined several new history independence notions that unlike SHI do not require canonical representations. In this chapter, we redesign the HIFS file system layer to support two new history independence notions – revealing last k operations for journaling (JHI)[2] and operation-agnostic history independence (OAHI) for regulatory compliance. The new file system is called *Delete Agnostic File System (DAFS)*.

[1] Load factor refers to file system space utilization.

[2] We refer to history independence in the presence of journaling as *journaled history independence*.

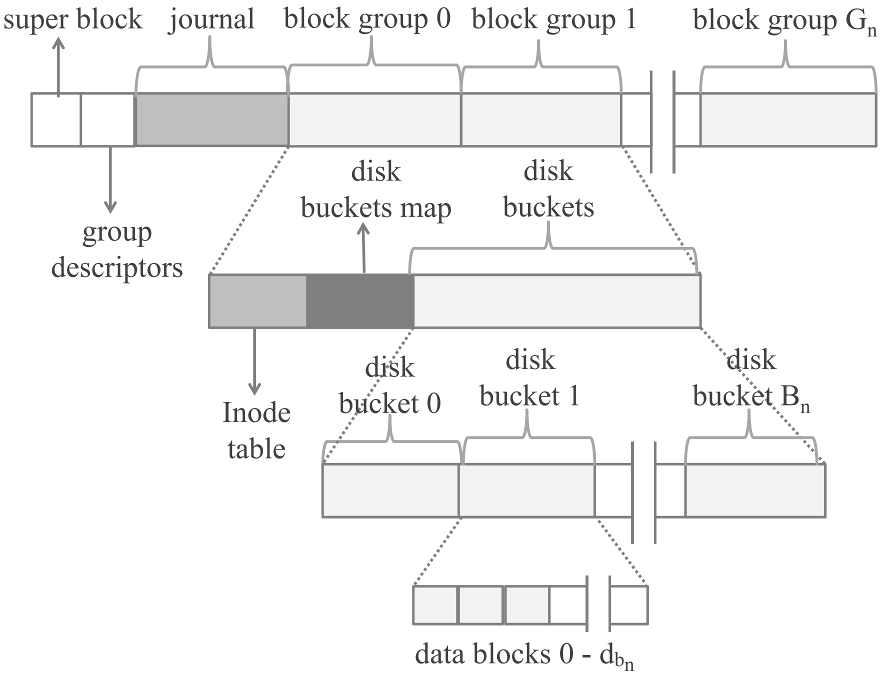


Figure 7.1: *DAFS* disk layout. Key parameters: $\mathsf{G_n}$ ← number of block groups, $\mathsf{B_n}$ ← number of disk buckets per block group, $\mathsf{d_s}$ ← Data block size in bytes, $\mathsf{d_{b_n}}$ ← number of data blocks per disk bucket.

We demonstrate that relaxing the history independence notion from SHI to JHI or from SHI to OAHI significantly improves file system performance.

7.2 Model

We assume an insider adversary with full access to the storage medium, such as the system disk. By analyzing data organization on disk, the adversary aims to derive file system's history. We also assume that the adversary can make multiple observations of disk state.

Both JHI and OAHI were formally defined in Section 5.5.1. For JHI, it is permissible to reveal only the last k file system operations performed. For OAHI, the adversary should not be able to detect whether a delete operation was performed between any two adjacent observations unless the delete is evident from the file system ADT states itself.

7.3 Journaled History Independence (JHI): Reveal Last k Operations

In the event of a system failure, it is imperative that the file system state is not corrupted. To ensure consistent state, file systems typically employ a journal. File system operations are first recorded in the journal and then applied to the file storage area. If a failure occurs while writing to the journal, the operations can be ignored on system recovery. On the other hand, if failure occurs while writing to file storage, operations are replayed from the journal on recovery. Thus, each write request to the file system causes two writes on disk, one to the journal and one to file storage.

Recall from Section 5.5.1 that journaling by definition reveals the last k operations. For journaling, revealing the last k operations is a necessary tradeoff for resilience.

7.3.1 DAFS Journaling

In DAFS, a separate region on disk is reserved for a journal in the form of a circular log. The journal contains information for a finite number of file system operations, say k operations. Operations are recorded in the journal in the order in which they are received by the file system. To restore consistency after system failure, it is essential to maintain operation order. Hence, the sequence of k operations recorded in the journal cannot be hidden. The file storage areas, such as the inode tables, disk bucket maps, and the disk buckets provide SHI just as in the case of HIFS. Hence, once a file system operation is applied to file storage and removed from the journal, its timing can no longer be identified.

In summary, DAFS journaling provides consistent failure recovery while revealing the sequence of only last k file system operations.

7.3.2 Apparent Paradox: Why Journaling Increases Efficiency

History independence relaxations that come with journaling allow significantly more efficient file system operations due to batching. The positive effect of batching on efficiency for write-intensive workloads is explained in the following.

To maintain canonical representations, in HIFS, data is potentially relocated on each file system write operation. The frequency of data relocation increases exponentially with the file system load factor. Hence, for higher load factors, the number of disk writes for each write request to the file system is much greater that the two disk writes required for journaling. Further, the same data blocks may be relocated several times in consecutive write operations. If write operations can be batched, then the number of times a data block is relocated can be reduced by avoiding redundant moves.

Hence, in DAFS, we choose to use the journal not only for failure recovery, but also as a buffer to batch write operations. Write operations are applied to file storage areas only when the journal is full. During the application to file storage area, redundant disk writes are eliminated significantly improving write throughputs.

7.4 Operation-Agnostic History Independence (OAHI)

Regulations [83,191,252] that are specifically concerned with irrecoverable data erasure and not with other artifacts of history can be met by systems that support OAHI for the delete operation. As discussed in Section 5.5.1, unlike SHI, OAHI for deletes can be achieved

File System Operation	**SHI (HIFS)**	**OAHI (DAFS)**
File write	$O(1/(1-L)^3)$	$O(1)$
File delete	$O(1/(1-L)^3)$	$O(1/(1-L)^3)$

Table 7.1: Number of disk bucket writes per file system operation. L is the file system load factor (space utilization), $0 \leq L \leq 1$.

without canonical representations. Relaxing the requirement to noncanonical representations presents significant efficiency benefits.

To make DAFS preserve OAHI only, we first transform the SHI hash table [38] into an OAHI hash table. Then, we use the OAHI hash table to organize files' data and files' metadata.

7.4.1 OAHI hash table

The SHI hash table [38] can be transformed into an OAHI hash table as follows. The hash table insert operation is modified to not maintain canonical representations. Instead, the insert operation uses linear probing [171] and inserts a key in the first available bucket. The OAHI hash table operations are listed in Procedure Sets 8 and 9 in the chapter appendix.

The hash table delete operation[3] alone provides OAHI. Deletion of a key from the hash table leaves an empty bucket, say bucket b_1. The delete operation then finds a key that prefers bucket b_1 more than the bucket it is located in, say bucket b_2. If such a key is found, it is moved from b_2 to b_1 making b_2 empty. The process is then repeated for bucket b_2 and so on, until no key is found for relocation. The net effect of the key relocation process is that a sequence of hash table operations containing a delete, results in the same hash table state as an insert-only sequence, thereby hiding all evidence of the delete.

7.4.2 OAHI in DAFS

DAFS uses the OAHI hash table for file storage. The OAHI hash table insert operation is not required to maintain canonical representations. The hash table insert operation is used by file system write operation. Hence, the overhead of maintaining canonical representations on file writes is eliminated.

When a file is deleted in DAFS, for each disk bucket allocated to the file, the same effect is achieved as that for a key deleted from the OAHI hash table. As a result, no evidence of a delete remains in the file system state and OAHI is preserved.

Changing the history independence notion from SHI in HIFS, to OAHI in DAFS, has significant potential for efficiency. As shown in Table 7.1, the number of writes to disk buckets needed for OAHI is significantly lower as compared to the number of writes needed for SHI. Fewer writes are needed for OAHI since the write operations are not required to

[3] For complete listing of SHI hash table operations refer to Chapter 6.

Parameter	**Value**
File system size	10 GB
Mean database size	1 GB
No. of databases	$L \cdot 10$
Disk block size ($\mathsf{d_s}$)	4 KB
No. of block groups ($\mathsf{G_n}$)	4
Disk blocks per bucket ($\mathsf{d_{b_n}}$)	512
Inode size	281 bytes

Table 7.2: Experimental parameters. L ← File System Load factor.

maintain canonical representations. Under noncanonical representations, when disk buckets are allocated to a file, other files' data needs no relocation. Relocation of data to ensure canonical representations was precisely the reason for lower throughputs of HIFS writes.

7.5 Experiments

DAFS implements two new history independence notions, JHI and OAHI. Both JHI and OAHI are aimed to increase file system efficiency. In this section, we compare the performance of DAFS and HIFS.

7.5.1 Setup

All experiments were conducted on servers with 2 Intel Xeon Quad-core CPUs at 3.16GHz, 8GB RAM, and kernel v3.13.0-24. The storage devices of choice are Hitachi HDS72302 SCSI drives.

7.5.2 Implementation

DAFS is implemented as a C++ based user-space Fuse [246] file system. All data structures, including OAHI hash table were written from scratch. File system setup parameters are listed in Table 7.2.

7.5.3 Measurements

To experiment for a real-world scenario we use the TPCC [11] database benchmark. The database of choice is Sqlite. Sqlite data files are stored using HIFS, DAFS, and Ext3. The BenchmarkSQL [1] tool is used to generate the TPCC workload.

Each test run commences with an empty file system and creates new databases on file system storage. The number of databases is increased until the file system is 90% full. The

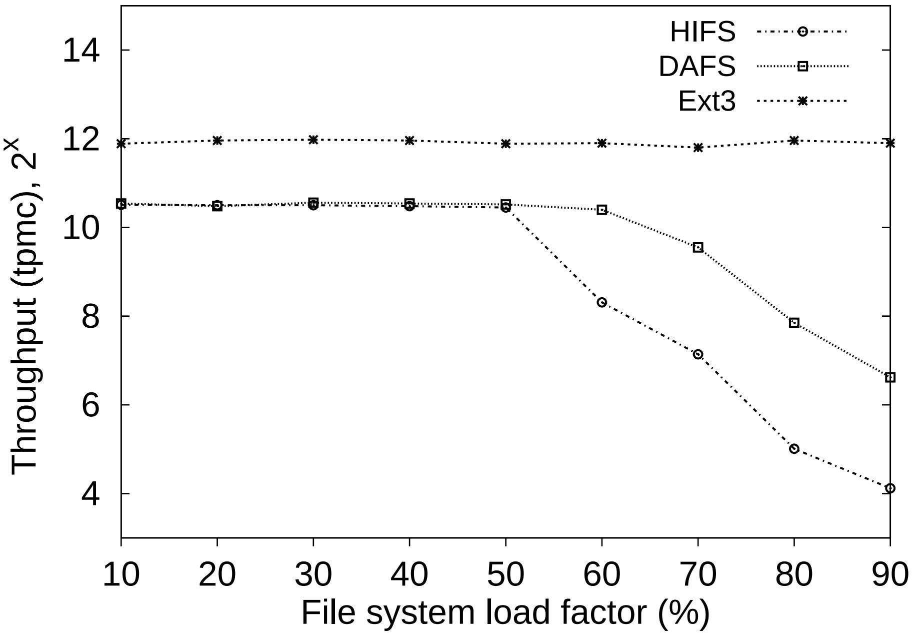


Figure 7.2: TPCC throughputs for Ext3, HIFS, and DAFS with file system load factor.

TPCC scale factor is 10 giving a size of 1GB for each database. Throughputs are measured at specific load factors[4] ranging from 10% to 90%.

7.5.4 Results

Figure 7.2 reports the throughputs for HIFS, DAFS, and Ext3. As per the TPCC specification [11], throughputs are reported as new order transactions executed per minute (tpmc). As seen, the performance of DAFS is up to 4x times better than HIFS for load factors >50%. Note that the performance of Ext3 is included as a reference. Ext3 does not provide OAHI.

For load factors $\leq$ 50%, HIFS and DAFS exhibit similar performance. At lower load factors fewer collisions occur as new files are added to file system storage. Fewer collisions mean that the frequency of data relocation to maintain canonical representations is low at load factors $\leq$ 50%. Hence, performance of DAFS and HIFS is similar at low load factors.

7.6 Conclusions

The system design and experimental results of this chapter validate our theoretical results from Chapter 5. In Chapter 5, using the new ΔHI framework we defined both JHI and OAHI. We also hypothesized that relaxing the history independence notion from SHI to JHI

[4] Load factor is the file system disk space utilization.

and OAHI will eliminate the need for canonical representations and significantly improve system performance.

In this chapter, we demonstrate the effect of history independence notions on file system performance. We design, implement, and evaluate the delete agnostic file system (DAFS). DAFS supports two new history independence notions - JHI and OAHI. JHI makes DAFS resilient to system failures and OAHI targets regulatory compliance. As shown by our experimental results, changing the history independence notion from SHI in HIFS, to OAHI in DAFS significantly increases file system performance.

7.7 Chapter Appendix

7.7.1 Procedure Sets

Procedure Set 8 OAHI Hash Table

Procedure: INSERT

Desc: insert the given key in to the hash table.

Input: Tables $H^{0-m}[n]$, key k

```
$<i, r> \leftarrow$ GET_MOST_PREFERRED_BUCKET($k$)
$c \leftarrow 0$
while $c < (n * (m + 1))$ do
    if $H^r[i]$ is null then
        $H^r[i] \leftarrow k$
        return $<i, r>$
    $c \leftarrow c + 1$
return $<null, null>$   {tables are full}
```

Procedure: SEARCH

Desc: search for the given key in the hash table.

Input: Tables $H^{0-m}[n]$, key k

```
$<i, r> \leftarrow$ GET_MOST_PREFERRED_BUCKET($k$)
$c \leftarrow 0$
while $c < (n * (m + 1))$ AND $H^r[i]$ is not null do
    if k == $H^r[i]$ then
        return $<i, r>$   {key found at $H^r[i]$}
    $<i, r> \leftarrow$ GET_NEXT_BUCKET($k, i, r$)
    $c \leftarrow c + 1$
return $<null, null>$   {key not found}
```

Procedure: DELETE

Desc: delete the given key from the hash table.

Input: Tables $H^{0-m}[n]$, key k

```
$<i, r> \leftarrow$ SEARCH($k$)
while $i$ is not null AND $H^r[i]$ is not null do
    $<j, s> \leftarrow$ GET_NEXT_BUCKET($k, i, r$)
    if $H^s[j]$ is not null AND KEY_PREFERS( $H^s[j], i, j, r, s$) then
        $H^r[i] \leftarrow H^s[j]$
        $k \leftarrow H^s[j]$
        $i \leftarrow j, r \leftarrow s$
```

Procedure Set 9 Customizable Procedures for OAHI Hash Table

```
Procedure: GDB
Desc: get the logical file bucket number from file offset.
Input: file offset $f_o$ : $f_o \in \mathbb{N}$
return $\left(\left\lfloor\left\lfloor\frac{f_o}{d_s}\right\rfloor/d_{b_n}\right\rfloor\right)$
```

```
Procedure: GET_MOST_PREFERRED_BUCKET
Input: key $k$ : $k = \{$file path $f_p$, file offset $f_o\}$
return $<h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n},\ h(f_p) \bmod \mathsf{G_n}>$
```

```
Procedure: GET_NEXT_BUCKET
Input: key $k : \{f_p, f_o\}$, bucket $i$, block group $r$ : $(i, r) \in \mathbb{N}$
$i \leftarrow (i+1) \bmod \mathsf{B_n}$
if $i == (h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n})$ then
    $r \leftarrow (r+1) \bmod \mathsf{G_n}$, $i \leftarrow h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n}$
return $<i, r>$
```

```
Procedure: BUCKET_PREFERS
Input: bucket i, block group r : (i,r) $\in \mathbb{N}$, key a : $\{f_{p_a}, f_{o_a}\}$, key b : $\{f_{p_b}, f_{o_b}\}$
return $h(f_{p_a}||\text{GDB}(f_{o_a})) > h(f_{p_b}||\text{GDB}(f_{o_b}))$
```

```
Procedure: KEY_PREFERS
Input: key k : $\{f_p, f_o\}$, bucket i, bucket j, block group r, block group s : (i,j,r,s) $\in \mathbb{N}$
if r <>s then
    return $((h(f_p) \bmod \mathsf{G_n})$ - r + $\mathsf{G_n}) \bmod \mathsf{G_n} < ((h(f_p) \bmod \mathsf{G_n})$ - s + $\mathsf{G_n}) \bmod \mathsf{G_n}$
return $((h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n})$ - i + $\mathsf{B_n}) \bmod \mathsf{B_n} < ((h(f_p||\text{GDB}(f_o)) \bmod \mathsf{B_n})$ - j + $\mathsf{B_n}) \bmod \mathsf{B_n}$
```

Chapter 8

Un-Traceable Deletion and Ficklebase

8.1 Introduction

8.1.1 Background and Motivation

The delete operation in modern computer systems can be an illusion [224]. Although once deleted, data may no longer be accessible via legitimate system interfaces, several instances [50, 239, 245] have demonstrated that presumably erased data can be recovered with simple mining techniques. Preserving deleted data violates retention policies set forth by legislations such as HIPAA [77], FERPA [49], FISMA [14], EU Data Protection Directive [4] and the Gramm–Leach–Bliley Act [20].

Prior work has proposed secure deletion to prevent direct recovery of deleted data from storage. Under secure deletion, system components where deleted data is preserved are first identified. For example, system subcomponents, such as memory [68], storage mediums [99] and file systems [55] have been shown to preserve deleted data. Applications such as databases hold on to deleted data in transaction logs, error logs, temporary tables, deallocated data pages, index entries and audit logs [94, 242]. Once the existence of deleted data is identified, the physical storage locations where deleted data resides are overwritten. Overwriting ensures that deleted data cannot be directly recovered from storage. Secure deletion mechanisms have been proposed for general storage media [99, 121, 151, 266], file systems [34, 153], and database applications [242]. Also, off-the-shelf [3] tools can now be used to perform secure deletion.

In Chapter 6, we showed that secure deletion alone is insufficient. Even after secure deletion, deleted data can recovered by analyzing data organization. We then proposed history independence as a solution to eliminate inferences about deleted data via data organization.

The storage layout of history independent data structures[1] [128] is a function of current state only and not of past operations. If a delete operation is part of a system interface, then utilizing a history independent data structure ensures that an adversary subsequently gaining access to the system storage is unable to infer whether a delete operation was performed.

[1] Also referred to as uniquely represented data structures.

Currently, history independent variants are available for hash tables [183], 2-3 Trees [174], B-Trees [114] and Skip Lists [115].

A third, largely ignored aspect in preventing recovery of deleted data concerns the relationship of deleted data to data that is present in the system now. The main observation here is that side effects of deleted data persist within current data. Analyzing current data can potentially reveal information about the data deleted in the past.

We posit that, for true regulatory compliance, in addition to secure delete and history independence, full erasure of postdeletion data side effects is required. We term the removal of data side effects as untraceable[2] deletion.

8.1.2 Our Contribution: Untraceable Deletion for Databases

We formalize untraceable deletion for relational databases (Section 8.2) and discuss various functional aspects of untraceable deletion. We then design Ficklebase, a relational database that achieves untraceable deletion. In Ficklebase, once a tuple is deleted, all of the deleted tuple's side effects are removed. Removal of all side effects achieves the same effect as if the deleted tuple was never inserted in the database. Ficklebase thus eliminates all traces of deleted data, rendering data unrecoverable and also guaranteeing that the deletion itself is undetectable.

8.1.3 Chapter Outline

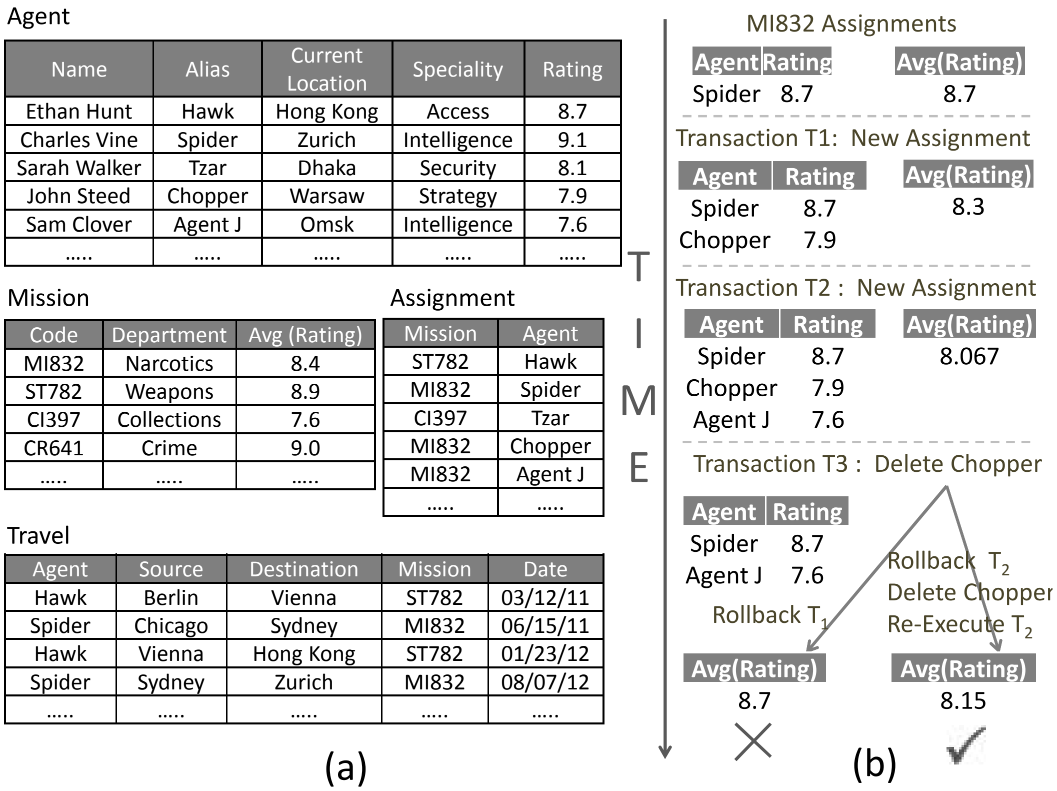
Data side effects and untraceable deletion are formalized in Section 8.2. Section 8.2 also identifies the scenarios where untraceable deletion is suitable or not suitable. Adversarial model is discussed in Section 8.3. Section 8.4 describes the Ficklebase architecture. Ficklebase security is analyzed in Section 8.5. Section 8.7 presents experimental results for Ficklebase performance. Ficklebase demo application is introduced in Section 8.8. Finally, Section 8.9 concludes the chapter.

8.2 Concepts

8.2.1 Example Illustrating Data side effects

To illustrate how deleted data can leave behind side effects, consider a snippet from a hypothetical intelligence agency database shown in Figure 8.1(a). Suppose that once agent Sarah leaves the agency all evidence of her existence in the database needs to be eliminated. The evidence includes tuples from the $Agent$ relation, travel information from $Travel$ relation, and mission assignments from the $Mission$ relation. In addition, the $Avg(Rating)$ attribute in $Mission$ relation also carries evidence of past existence of agent Sarah since the average was computed using agent Sarah's rating.

[2] To differentiate from a "secure deletion" performed by overwriting.



Agent

Name	Alias	Current Location	Speciality	Rating
Ethan Hunt	Hawk	Hong Kong	Access	8.7
Charles Vine	Spider	Zurich	Intelligence	9.1
Sarah Walker	Tzar	Dhaka	Security	8.1
John Steed	Chopper	Warsaw	Strategy	7.9
Sam Clover	Agent J	Omsk	Intelligence	7.6
.....	.....	.....	.....	.....

Mission

Code	Department	Avg (Rating)
MI832	Narcotics	8.4
ST782	Weapons	8.9
CI397	Collections	7.6
CR641	Crime	9.0
.....	.....	.....

Assignment

Mission	Agent
ST782	Hawk
MI832	Spider
CI397	Tzar
MI832	Chopper
MI832	Agent J
.....	.....

Travel

Agent	Source	Destination	Mission	Date
Hawk	Berlin	Vienna	ST782	03/12/11
Spider	Chicago	Sydney	MI832	06/15/11
Hawk	Vienna	Hong Kong	ST782	01/23/12
Spider	Sydney	Zurich	MI832	08/07/12
.....	.....	.....	.....	.....

Figure 8.1: (a) Intelligence Agency Database Snippet (b) Rollback vs Transaction reexecution

To remove all evidence of agent Sarah's past existence, relevant tuples from $Agent$, $Travel$ and $Mission$ relations need to be deleted. Further, the $Avg(Rating)$ needs to be recomputed to remove the effect of agent Sarah's rating from the average. The effect on the average rating can be removed by reexecuting transactions that computed the average since agent Sarah was inserted into the database. Figure 8.1(b) illustrates that the reexecution is necessary to ensure that the database reflects the correct average rating after agent deletion.

At first glance, it may appear that removal of all data side effects can be performed by application logic. However, deletion of all data linked to an agent can potentially be a complex task. For example, in the case of a transaction that uses agent information and mission data to generate new travel assignments for other agents. Also, removal of data side effects via application logic requires detailed semantic knowledge of all database transactions increasing the burden on database application developers. Ideally, removal of all evidence of the deleted agent should be supported transparently by the underlying database as Ficklebase does.

8.2.2 Notations

Consider the following notation for a transaction T_j in a relational database - $T_j(R) \rightarrow (M)$, where R represents the read operations performed by T_j while M indicates the data modification operations[3] of T_j, $|R| \geq 0, |M| \geq 0$. Let r_{t_i}, u_{t_i} and i_{t_i} denote the read, update, and insert operations, respectively, of a tuple t_i. Also let $\mathcal{TS}(o_i)$ denote the commit timestamp of the transaction that performs operation o_i.

Now suppose that the following transactions have been executed and committed in sequence:

$$T_1() \rightarrow (i_{t_1}), T_2(r_{t_1}) \rightarrow (i_{t_2}), T_3(r_{t_2}) \rightarrow (u_{t_5}, u_{t_6}), T_4(r_{t_5}, r_{t_4}) \rightarrow (u_{t_7}, i_{t_{10}}). \tag{8.1}$$

8.2.3 Data Side Effects

We first determine tuple side effects for transactions in schedule 8.1. Transaction T_2 read tuple t_1 and inserted t_2. Hence, the insertion of t_2 is a side-effect of t_1. Similarly, transaction T_3 read tuple t_2 and updated tuples t_5 and t_6. Hence, updates to t_5 and t_6 are side effects of t_2. In addition, T_4 read tuple t_5 updated by T_3. Hence, modifications made by T_4 i.e. update of t_7 and insertion of t_{10} are also side effects of t_2 and so on.

Overall, the side effects are as follows:
$\mathcal{SA}(t_1) = (i_{t_2}$ by T_2, u_{t_5}, u_{t_6} by T_3, $u_{t_7}, i_{t_{10}}$ by $T_4)$
$\mathcal{SA}(t_2) = (u_{t_5}, u_{t_6}$ by T_3, $u_{t_7}, i_{t_{10}}$ by $T_4)$
$\mathcal{SA}(t_5) = \mathcal{SA}(t_4) = (u_{t_7}, i_{t_{10}}$ by $T_4)$.

It is to be noted from the above illustration that data side effects go beyond simple primary and foreign key relationships. In fact, any modification after reading of a data item constitutes the data item's side effects and needs to be hidden after deletion of the read data item.

Definition 13. Side effects *of a tuple t_i ($\mathcal{SA}(t_i)$) are represented by the set of all data modifications (update and insert) operations, such that*

1. *If $\exists T_j(R_j) \rightarrow (M_j)$, such that $r_{t_i} \in R_j$ and $\mathcal{TS}(T_j) > \mathcal{TS}(i_{t_i})$, then $\forall o_{t_m} \in M_j, o_{t_m} \in \mathcal{SA}(t_i)$.*

2. *$\forall o_{t_m} \in \mathcal{SA}(t_i)$, If $\exists T_j(R_j) \rightarrow (M_j)$, such that $r_{t_m} \in R_j$ and $\mathcal{TS}(T_j) > \mathcal{TS}(o_{t_m})$, then $\forall o_{t_n} \in M_j, o_{t_n} \in \mathcal{SA}(t_i)$.*

Note that the side effects definition is recursive but not circular. The reasons for noncircularity as as follows. Under a fully serializable mode of execution there exists a serial schedule of database transactions, that is, a sequential schedule with no overlapping transactions. A serial schedule is by definition noncircular. Further, $\mathcal{SA}(t_i)$ includes database operations and not the tuples themselves. Since each database operation is unique, circularity is avoided. To illustrate, consider the sequence $T_1(r_{t_1}) \rightarrow (u_{t_2})$, $T_2(r_{t_2}) \rightarrow (u_{t_1})$. Although it may appear circular at first glance, the operations u_{t_1} and u_{t_2} are distinct, thereby $\mathcal{SA}(t_1) = \{u_{t_2}\}$ and $\mathcal{SA}(t_2) = \{u_{t_1}\}$.

[3] Data modification operations include updates and inserts.

8.2.4 Untraceable Deletion

Consider the transaction schedule 8.1. Now, suppose that tuple t_2 expires and is to be deleted. An untraceable delete of t_2 should leave the database in a state such that no trace of t_2 is left behind, not even the effects of t_2 on other data. Untraceable delete of t_2 thus requires the following:

- Rollback of transactions T_4 and T_3.
- Deletion of t_2, which is rollback of transaction T_2.
- Reexecution of transactions T_3 and T_4. The reexecution is necessary as illustrated by the example in Figure 8.1).

The above steps results in the execution schedule $T_1T_3T_4$. A database that performs operations equivalent to the above steps and achieves a schedule where the transaction that inserted t_2 never took place achieves untraceable deletion of t_2 [4].

We define untraceable deletion for relational databases as follows:

Definition 14. Untraceable delete. *Let the current database state be achieved by the transaction execution sequence $\Gamma^{\mathcal{S}} = ...T_{j-2}T_{j-1}T_jT_{j+1}T_{j+2}$, where tuple t_i was inserted by transaction T_j. Then, an* untraceable delete *of t_i is a set of operations that changes the current database state into a state computationally indistinguishable*[5] *from a state resulting from the execution sequence $\Gamma^{\mathcal{E}}$=$...T_{j-2}T_{j-1}T'_jT_{j+1}T_{j+2}$, where $T'_j = \phi$ or $T'_j = T_j - i_{t_i}$.*

$T'_j = \phi$ when the application logic dictates that noninsertion of t_i means complete rollback of transaction T_j making $\Gamma^{\mathcal{E}} = \Gamma^{\mathcal{S}} - T_j$. Otherwise, $T'_j = T_j - i_{t_i}$, for example, when T_j inserts t_i using an insert-select query.

8.2.5 Applicability of Untraceable Deletion

It is important to note that untraceable deletion is not desirable in certain scenarios. For example, consider a banking application that records money transfer between clients. If a client A has transfers recorded with another client B, then deletion of client A does not justify deletion of $A \leftrightarrow B$ transfers and their side effects.

On the other hand, consider a privacy sensitive application that maintains confidential documents, records user access to documents, and generates statistical or cross-document intelligence information. Once a document D is to be purged, it is important to properly erase all associated access records and intelligence information deduced from D. Erasing all information deduced from document D eliminates all side effects of document D.

[4] Secure deletion and history independence would still be required to truly erase t_2 as discussed in Section 8.4.7.

[5] No nonuniform probabilistic polynomial time algorithm exists that can distinguish between the states [146]. untraceable deletion in fact, offers stronger information theoretic guarantees but we formulate the definition in terms of computational adversaries to allow for the deployment of cryptography in the underlying mechanisms.

A third category of applications where an equivalent of untraceable delete operation is desired is not privacy but rather functionality-centric. Economic data, such as the Current Population Survey (CPS) [2] are permitted to undergo revisions. A simple case for revision could be that an individual I is wrongly classified. Correcting the classification requires deletion of I's information from the data set and its effects on computed statistics, such as average earnings.

8.2.6 Caveats of Untraceable Deletion

A database providing untraceable deletion will in certain aspects function differently than a traditional database without untraceable deletion. Here we discuss some of the differences.

Time-sensitive Queries

If a query run over past data is repeated, then the query result should be unchanged since the past has already occurred. For example, order is shipped, patient is discharged, etc. However, untraceable deletion of one or more tuples that comprised the result set of the query can cause the query response to differ at a later time.

To illustrate, consider a query Q = "find the number of agents that travelled on date d_t" on the sample database from Figure 8.1. Query Q in SQL can be written as – SELECT COUNT(DISTINCT AGENT) FROM TRAVEL WHERE DATE = d_t. If an agent was made untraceable on date d_e, where $d_e > d_t$, then the responses for query Q will be different on two dates d_1 and d_2, $d_t < d_1 < d_e < d_2$. In a traditional database, the expected response for query Q on both dates d_1 and d_2 would be the same.

Committed Transactions

Traditionally, once committed, transactions are treated as permanent and irreversible. However, with untraceable deletion effects of committed transactions are no longer permanent. In fact, in order to support untraceable deletion, a database must employ mechanisms to change the effects of committed transactions.

External Application Logic

To provide untraceable deletion transparently, a database must be able to reexecute transactions. To reexecute transactions a database must have access to and understand all application logic. For instance, in the example of Figure 8.1, if the database did not know that the *Rating* statistic was an average, the database would be unable to correctly remove the effects of deleted agent *Chopper*.

Therefore, untraceable deletion may be impossible for databases that are agnostic to application logic semantics. For example, when most of the business logic or functionality resides in application programs, which in turn access the database externally via a standard SQL interface.

8.3 Model

Data Expiration

Each database tuple has an associated expiration time. The tuple expiration time is specified or computed at tuple insertion time. At expiration, a tuple needs to be deleted untraceably.

Tuple expiration times occur at fixed time intervals. For example, daily, weekly, monthly, etc. We assume that expiration times of all tuples coincides with the end of an interval. Delete queries can be executed at any time by clients. However, a tuple is deleted untraceably only at expiration.

Adversary

We assume an adversary with full access to current and future database states. The adversary uses mining or forensic techniques to recover information about tuples deleted in the past.

Suppose that a tuple t_i expires and is made untraceable at time E_t. Let D_{c_t} denote the database state at time c_t. Then the goal of untraceable deletion is to prevent the adversary from recovering any information about t_i when the adversary has full access to any database state D_{c_j}, such that $c_j > E_t$.

Note that the case where the adversary gains access to two database states D_{c_m} and D_{c_n}, where $t_i \in D_{c_m}$ and $c_m < E_t < c_n$ is trivial since the adversary can detect the deletion of t_i by merely computing the difference between the states D_{c_m} and D_{c_n}.

Transactions

A client transaction T_j is a sequence of SQL statements, that is,

$$T_j=\{\text{begin,commit}\}, \text{ or } T_j=\{\text{begin},Q_1,Q_2,Q_3,...,Q_m,\text{commit}\},$$

where Q_i is a create, drop, insert, update, delete or select SQL statement, $1 \leq i \leq m$, $m \geq 1$.

8.4 Ficklebase Architecture

8.4.1 Key Ideas

Ficklebase achieves untraceable deletion through versioning. Ficklebase maintains multiple logical database versions. Each version has an associated expiration time. A given tuple is only inserted in versions with expiration time less than or equal to the tuple expiration time. Hence, a database version contains only tuples that expire on or before the version expiration time.

Each client transaction is applied to all versions. When current time approaches a version expiration time, the entire database version is deleted. Deletion of database version causes all tuples with expiration time on or before the version expiration time to be untraceably

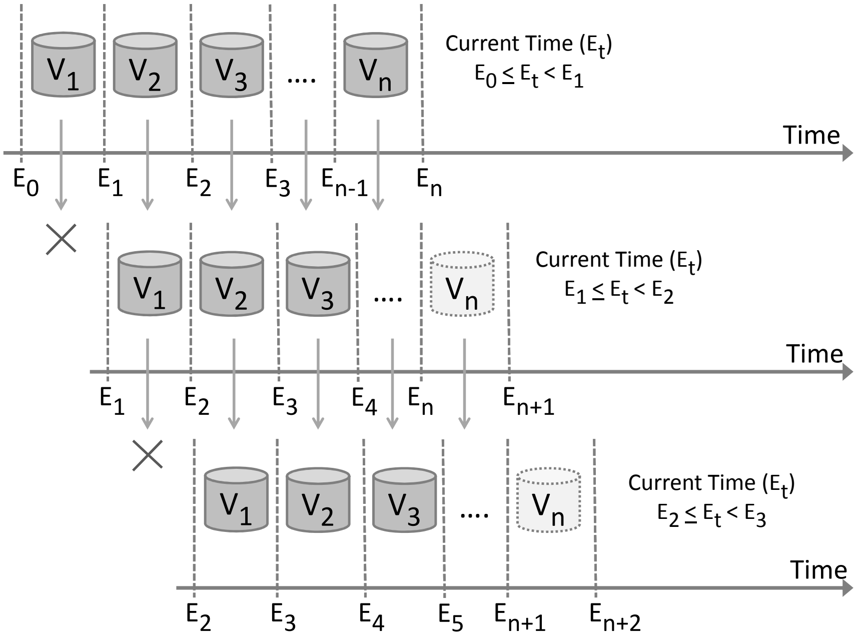


Figure 8.2: Version maintenance and expiration with progression of time. V_j = Version$_j$.

deleted. Since a tuple is never inserted in a version that the tuple is not supposed to exist in, the tuple leaves no side effects in that version. After tuple expiration time, only the versions in which the tuple was never inserted exist. Thus, after tuple expiration time, no evidence of expired tuple's past existence is present including tuple side effects.

Achieving untraceable deletion by versioning avoids the need to keep track of all system-wide side effects. Further, the versioning approach does not explicitly need retroactive rollbacks of committed transactions and their reexecution.

In Ficklebase, maintenance of database versions, transaction application, and version expiration are achieved using runtime query rewriting.

8.4.2 Overview

Recall from Section 8.3 that tuples expire at fixed intervals of time. Let E_i denote the end time of interval i. For each time interval, Ficklebase maintains a separate logical database version V_i (Figure 8.2). Version V_i contains tuples with an expiration time $\geq \mathcal{E}_x(V_i)$, where $\mathcal{E}_x(V_i)$ is the time when V_i will be fully expired. At a given time E_t, such that $E_0 \leq E_t < E_n$, database versions V_1 to V_n exist with $\mathcal{E}_x(V_1) = E_1$, $\mathcal{E}_x(V_2) = E_2$ and so on.

Each client transaction T_j is transparently applied to all versions with the following two restrictions:

- When applied to version V_i, only tuples with expiration times $\geq \mathcal{E}_x(V_i)$ are visible to queries in T_j.

- Insertion of a tuple t by T_j in V_i is ignored if expiration time of t is $\leq \mathcal{E}_x(V_{i-1})$.

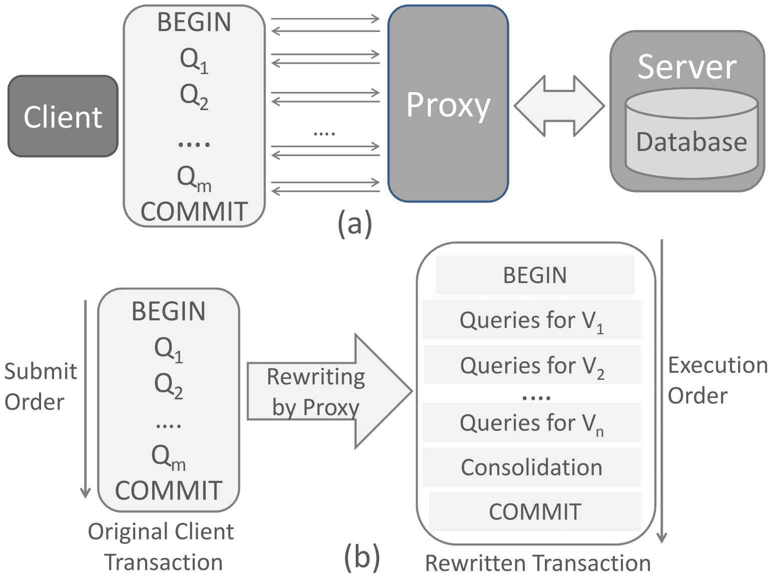


Figure 8.3: Overview of (a) Architecture (b) Query Re-writing. Q_i = Query$_i$, V_j = Version$_j$.

The net effect of above restrictions is that for a version V_i, all tuples with expiration times $\leq \mathcal{E}_x(V_{i-1})$ are never inserted in V_i. As a result, tuple side effects are never propagated to any transactions and data structures in V_i. In effect, all tuples with expiration time $\leq \mathcal{E}_x(V_{i-1})$ are untraceably delete when version V_{i-1} expires.

In summary, at a given time E_t, only versions with expiration time $>E_t$ exist, $i \geq 1$. A version V_i is expired at its expiration time $\mathcal{E}_x(V_i)$ (Section 8.4.7). To illustrate, at any time E_t, $E_0 < E_t < E_1$ versions $V_1, V_2, ..., V_n$ exist, with V_1 being the current version visible to clients (Figure 8.2). Once the current time approaches E_1, version V_1 is deleted, V_2 become V_1, V_3 becomes V_2 and so on. Also, $\mathcal{E}_x(V_1) \leftarrow E_2$, $\mathcal{E}_x(V_2) \leftarrow E_3$ and so on.

The client application is not aware of the existence of versions other than the current version V_1. A new version V_{n+1} is created when a tuple with expiration time $>E_n$ is inserted by a client transaction T_j.

8.4.3 Components

Figure 8.3 (a) illustrates the main Ficklebase components. Query rewriting logic resides in the Ficklebase proxy. The proxy intercepts all client queries and communicates with the server on behalf of the clients. The database server is an off-the-shelf DBMS.

8.4.4 Consolidated Versioning

All versions are maintained within a single database instance. To limit storage overheads, tuple copies are combined, that is, if tuple attributes have the same value across multiple versions then only a single copy of the tuple is maintained for the multiple versions.

To consolidate tuple copies, a special *VERSION* attribute is transparently added to each relation by query rewriting (Section 8.4.5). The *VERSION* attribute is not visible to clients. The *VERSION* attribute is a bit field of size β_v wherein a bit b_i, $0\underline{<}i\underline{<}\beta_v$, is set only if the tuple is valid in version V_i, that is, the tuple expiration time is $\leq \mathcal{E}_x(V_i)$ [6].

Client queries specify tuple expiration times on insertion via the *EXPIRATION TIME* tuple attribute. Rewriting of insert queries (Figure 8.6) converts the client-specified expiration time into the correct value of the *VERSION* attribute[7]. The conversion is done by the *EXDT2VER* function (Procedure Set 10). Note that the function *EXDT2VER* listed in Procedure Set 10 is a sample that implements daily, monthly, quarterly, and yearly expirations. For additional functionality, such as hourly expiration, necessary modifications should be made.

Tuples are copied on write only, when an update query modifies one or more tuple attributes causing the attribute values to differ between versions. The *VERSION* attribute is automatically modified by query rewrites to indicate distinct tuple versions.

To illustrate how tuple copies are created and maintained in Ficklebase, consider a tuple t_j with k attributes, such that $t_j = \{VERSION{=}0000..11,\ ATTR_1{=}value_1,\ ATTR_2{=}value_2,\ \ldots\ ,\ ATTR_k{=}value_k\}$. The *VERSION* attribute has bits b_1 and b_2 set indicating that the same tuple copy is valid in both versions V_1 and V_2, that is, expiration time of $t_j \leq \mathcal{E}_x(V_2)$. Now, suppose that an update query applied to V_2 modifies $ATTR_1$ from $value_1$ to $value_1'$. Then a new copy of t_j is created such that
$t_j = \{VERSION{=}0000..01,\ ATTR_1{=}value_1,\ ,\ ATTR_2{=}value_2,\ \ldots\ ,\ ATTR_k{=}value_k\}$ and
$t_j' = \{VERSION{=}0000..10,\ ATTR_1{=}value_1',\ ,\ ATTR_2{=}value_2,\ \ldots\ ,\ ATTR_k{=}value_k\}$
The version fields of the original and copied tuples are updated to correctly maintain distinct version copies.

8.4.5 Query Rewriting

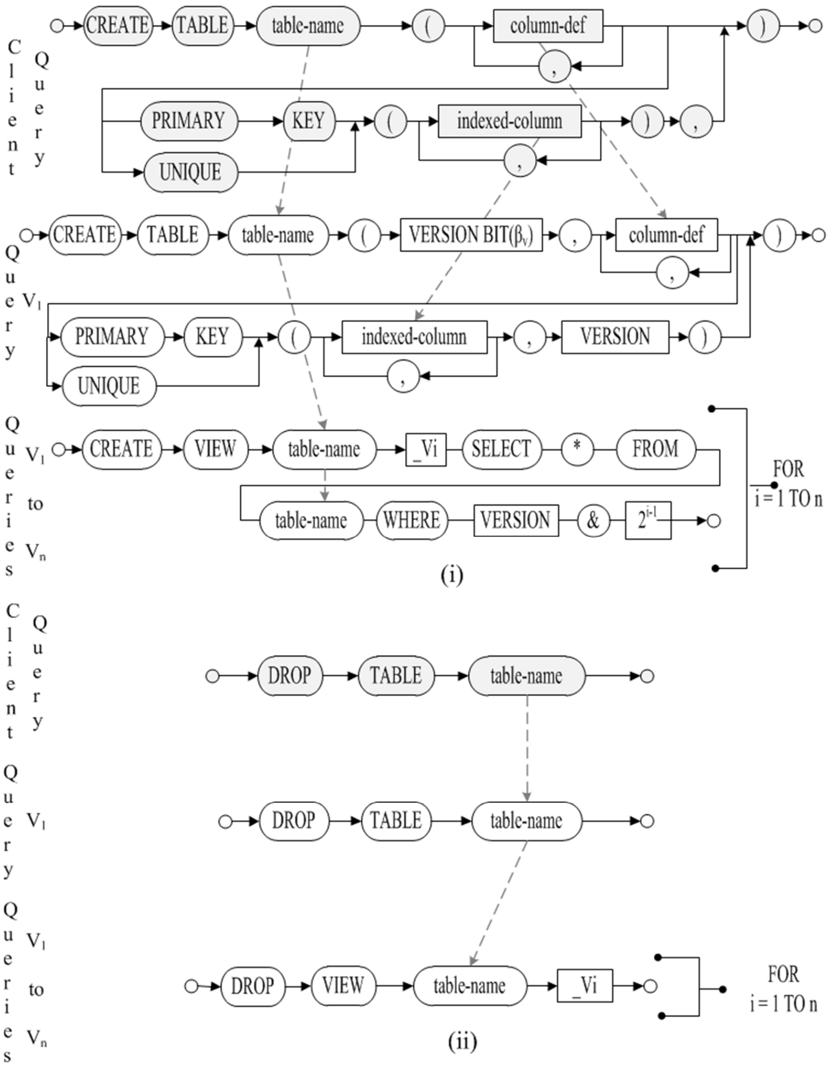
As discussed in Section 8.4.2, Ficklebase achieves untraceable deletion via versioning. In Ficklebase, database versioning is achieved using query rewriting. Client queries are transparently rewritten by the Ficklebase proxy to achieve the effect of versioning.

Each client query is rewritten into a set of queries. Each query in the set is classified as a *version-specific* or a *consolidation* query. *Version-specific* queries only affect the version that they are applicable to while *consolidation* queries ensure compact storage by combining tuple copies across versions. Figure 8.3 (b) gives an overview of query rewriting along the order of client query submission and the order of query execution after rewriting. *BEGIN* and *COMMIT* statements are executed as is at the start and end of the rewritten transaction.

Figures 8.4 - 8.7 detail query rewriting. We discuss the rewrites for each SQL statement below.

[6] The last bit b_{β_v} is used for consolidation and does not represent any version.

[7] Expiration times are only specified/computed in insert queries and cannot be updated at a later time – an almost pervasive requirement of most information life-cycle regulations.



(i)

(ii)

Figure 8.4: Query rewrites for create table and drop table statements.

Create Table and Drop Table Statements [Figure 8.4]

Create and drop statements are rewritten to achieve the following:

- To transparently add the *VERSION* attribute to the relation being created. The *VERSION* attribute is also added as the terminal field on table indexes.

- To create and drop version-specific views. Version-specific views are used in rewriting select, update, and insert queries. Each version-specific view is applicable to a specific relation in a specific database version. If there are r relations and n database versions, then there are $r \cdot n$ version-specific views. A version-specific view on a relation R for a version V selects only the tuples from R that are valid in version V.

- Create additional indexes on the *VERSION* attribute to improve overall transaction

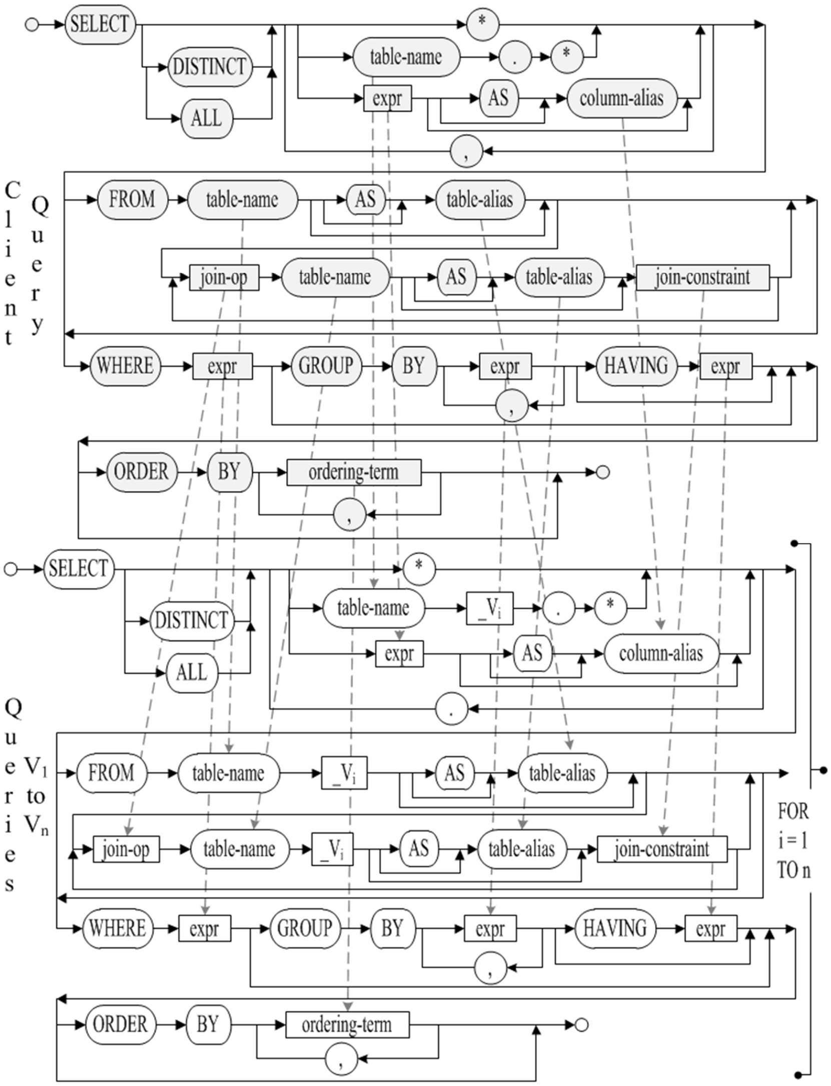


Figure 8.5: Query rewrites for select statements.

performance.

Select Statements [Figure 8.5]

A select statement is rewritten into n select statements, where n is the number of database versions present. Hence, rewriting of a select statement results is in a new unique select statement for each database version. For a select statement rewritten for version V, all table references in the select are replaced with the corresponding version-specific views. A version-specific view on a relation R for a version V selects only the tuples from R that are valid in version V. Hence, a select statement for version V reads only tuples valid in version V.

Only the results of the select statement executed on version V_1 are returned to the client.

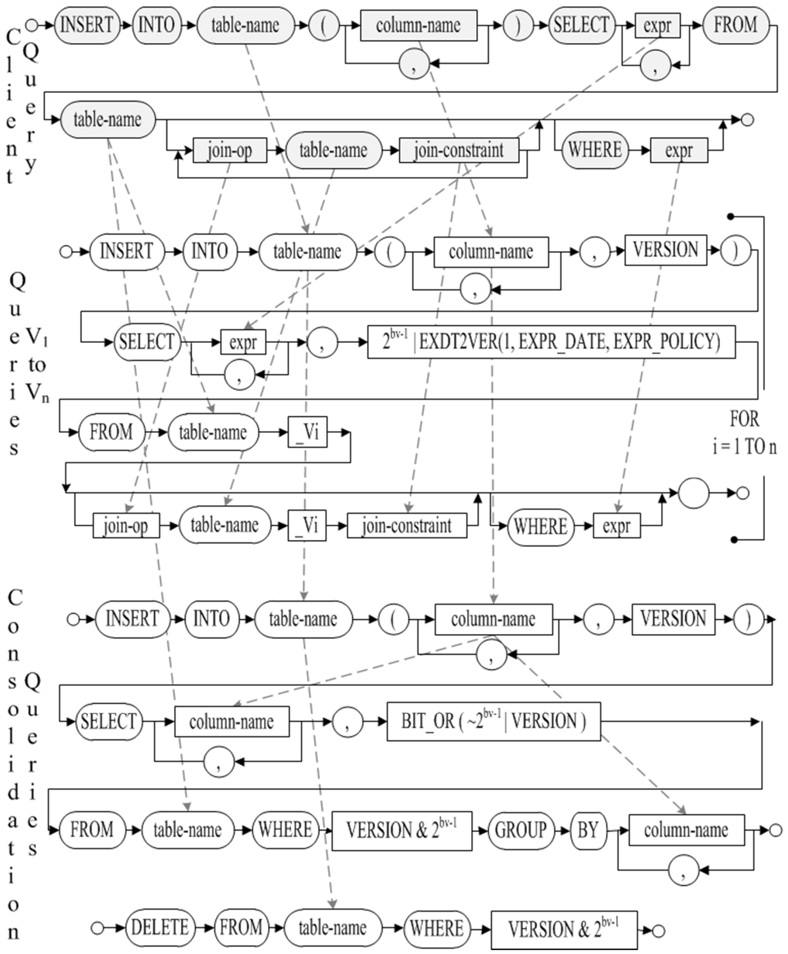


Figure 8.6: Query rewrites for insert statements.

The results of select statements executed on all other versions are filtered out by the proxy component. Thus, the existence of versions other than version V_1 is hidden from client applications.

The case where a transaction is read-only is handled differently. A read-only transaction comprises of only select statements. All queries within a read-only transaction are applied only to the current version V_1 and the results of each query are returned to the client. Since read-only transactions do not modify any tuples, they produce no side effects. Hence, the application of read-only transactions to the current version suffices.

Insert and Update Statements [Figures 8.6,8.7]

Similar to select, insert and update statements are rewritten to replace table references with version-specific views. Insert statements create new tuples and update statements modify existing tuples. Hence, new tuple copies come into existence as a result of inserts and updates.

The new tuple copies generated by insert and update statements are combined together by consolidation queries. Consolidation queries are generated for each database relation in

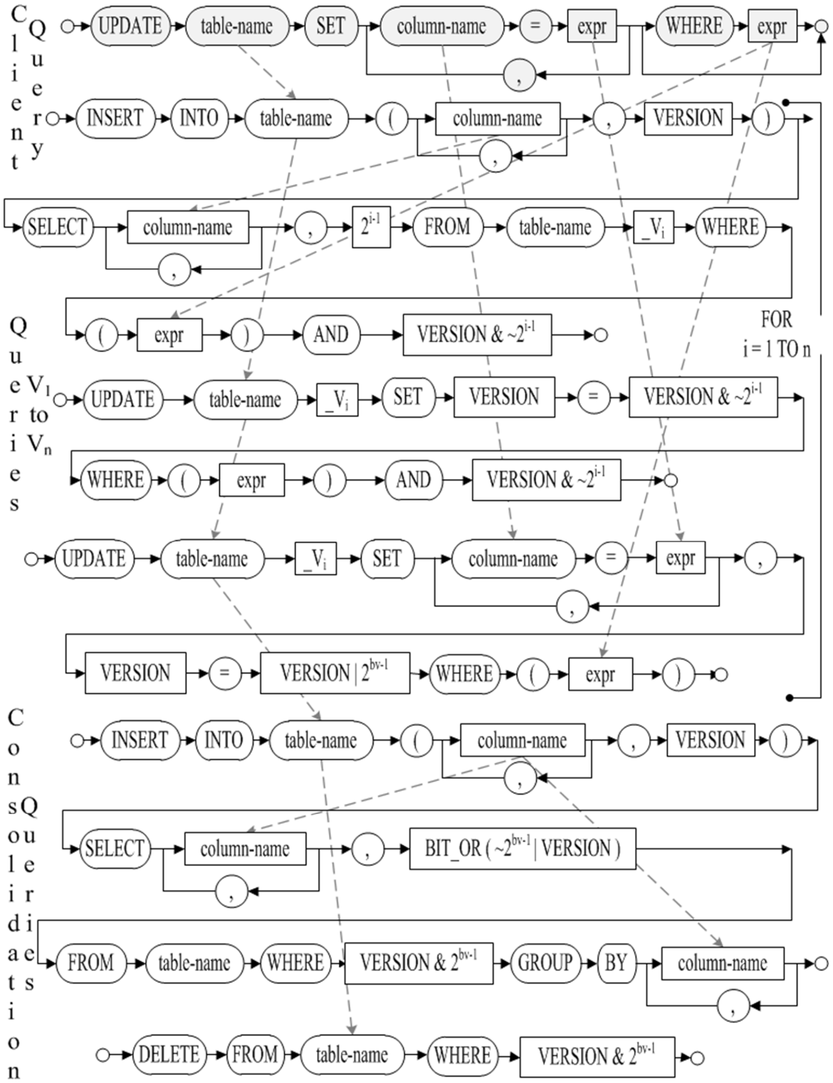


Figure 8.7: Query rewrites for update statements.

which either a tuple is inserted or modified by the client transaction. Also, similar to select, only results of statements executed on current version V_1 are seen by clients.

8.4.6 Version-Specific Rollbacks

It is often desired by application logic that transactions be rolled back under certain conditions. For example, tuple not found. Note that rollbacks explicitly requested by applications are not error or failure conditions, such as duplicate key or deadlock. In case of error or failure, a transaction is implicitly rolled back by the DBMS.

To illustrate how Ficklebase handles application-requested rollbacks, consider the following two queries submitted as part of a client transaction.

```
SELECT @d_next_o_id := d_next_o_id, d_tax
```

Procedure Set 10 EXDT2VER

Input: sver INT, exdt DATE, policy INT

Output: version BIT(β_v)

```
ver ← 0
ever ← 0
switch(policy)
  case 1:
    ever = datediff(exdt, curdate()) + 1
  case 2:
    ever = period_diff(extract(year_month from exdt),
           extract(year_month from curdate())) + 1
  case 3:
    ever = (period_diff(extract(year_month from exdt),
           extract(year_month from curdate())) div
           3) + 1
  case 4:
    ever = year(exdt) - year(curdate()) + 1
end switch
if ever $\geq$ sver then
  ver ← $2^{sver-1}$
return ver
```

```
FROM DISTRICT WHERE d_id = 1 AND d_w_id = 1

ROLLBACK(ISNULL(@d_next_o_id))
```

Here, the application desires that if no tuple is selected by the first select query, then the transaction be rolled back. The *ROLLBACK* syntax is specially provided by Ficklebase for the purpose of requesting a transaction rollback. Ficklebase query rewriting handles a rollback request as follows. The client *ROLLBACK* statement is rewritten for each version. For user defined variables, such as @d_next_o_id in the example above, a separate copy is created for each version. Also, a savepoint is created on the database before execution of queries for each version.

When a rollback occurs for any version, that is, the condition in the *ROLLBACK* statement evaluates to true, the Ficklebase proxy issues a SQL *ROLLBACK* statement to the database. The SQL *ROLLBACK* statement rolls back the transaction up to the previous savepoint. Rolling back to the previous savepoint undoes the effects of all queries executed on that specific version. The effects of queries on other versions remain intact.

Rollbacks in Ficklebase are similar to nested transactions [221] with distinct subtransactions. In nested transactions, individual subtransactions can be rolled back without affecting other subtransactions.

8.4.7 Expiration

As illustrated in Figure 8.2, when the current time E_t approaches $\mathcal{E}_x(V_1)$, version V_1 expires. To expire version V_1, all tuples that were valid until version V_1, that is, all tuples with expiration time $\leq E_t = \mathcal{E}_x(V_1)$ are deleted. Then, the next version V_2 is set as the current version V_1 visible to clients. For all versions V_i, where $i > 0$, $V_i \leftarrow V_{i+1}$ and thus $\mathcal{E}_x(V_i) \leftarrow \mathcal{E}_x(V_{i+1})$.

The expiration of version V_1 is achieved by a scheduled transaction that executes at the expiration interval $\mathcal{E}_x(V_1)$. The transaction comprises of the following queries for each relation R_i.

```
UPDATE Ri SET VERSION = VERSION >> 1

DELETE FROM Ri WHERE VERSION = 0
```

The net effect of the above queries is the deletion of all tuples that were valid only in version V_1.

8.4.8 Storage Analysis

Suppose the database comprises of N tuples and the number of active versions is n. Then, in the worst case, every tuple has a distinct copy in each version $V_i, 1 \leq i \leq n$ giving a overall storage requirement of $N \cdot n$ tuples. In the best case, each tuple has the same attribute values for all of versions and storage for only N tuples is required.

Now, suppose that each tuple is equally likely to expire at any interval $E_i, 1 \leq i \leq n$. Also, suppose that each tuple has distinct copies for each version V_j, such that, $\mathcal{E}_x(V_j) \leq E_i$. Then, the total number of database tuples is $N \cdot \frac{(n+1)}{2}$.

If we further assume a random distribution of client queries, such that each tuple expiring at $E_i, 1 \leq i \leq n$ is equally likely to have j copies $(1 \leq j \leq i)$, then the average storage requirement is $N \cdot \frac{(n+3)}{4}$.

In summary, the overall storage complexity of Ficklebase is $O(N \cdot n)$.

8.5 Untraceability

In this section, we show that Ficklebase query rewriting and versioning achieves untraceable deletion as defined in Section 8.2.

Let $\Gamma_{\mathcal{S}}$ denote the set of all transactions submitted by the client until time ℓ. Let tuple t_k with expiration time $E_t > \ell$ be inserted by a transaction T_j, where $T_j \in \Gamma_{\mathcal{S}}$. As per the expiration model in Section 8.3, the expiration of tuple t_k will coincide with the expiration of some version V_i, that is, $\mathcal{E}_x(t_k) = \mathcal{E}_x(V_i) = E_t$ Also, let $\Gamma_{\mathcal{E}}^{V_i}$ denote the set of transactions successfully committed on version i until time ℓ. Then, Ficklebase guarantees that queries for a version V_i do not access any tuples that have expired before version V_i. That is, the following theorem holds for Ficklebase.

Untraceable deletion of tuple t_k: $\forall V_i$, $T_j \in \Gamma_{\mathcal{E}}^{V_i}$ iff. $\mathcal{E}_x(t_k) \geq \mathcal{E}_x(V_i)$.

Proof. For each version V_i, function EXDT2VER (Figure 8.6) determines whether tuple t_k is valid in V_i, that is, whether $\mathcal{E}_x(t_k) \geq \mathcal{E}_x(V_i)$. If t_k is not valid in version V_i, then t_k is inserted with corresponding bit in the *VERSION* attribute set to zero. Setting the bit to zero ensures that t_k is not visible to any subsequent query on V_i. Before transaction T_j commits, consolidation queries delete all tuples, which have zero value for the bit corresponding to version V_i (Figure 8.6). Thus, an invalid tuple is never accessed by queries for a version V_i.

Further, if transaction T_j rolls back on version V_i, then the rollback mechanism of Section 8.4.6 ensures that T_j's queries have no effect on V_i.

Finally, when the current time is $> E_t$, all versions V_j where $\mathcal{E}(V_j) \leq E_t$ expire and are securely deleted (Section 8.4.7). The only versions left after time E_t would have expiration time $> E_t$ and not contain any side effects of tuple t_k. □

8.6 Untraceable Deletion + Secure Deletion + History Independence = Truly Irrecoverable Deletion

Untraceable deletion eliminates deleted data's side effects from current data. Secure deletion ensures target data is physically deleted from the storage medium. History independence erases all evidence of past existence from data organization. When combined, untraceable deletion, secure deletion, and history independence can together achieve truly irrecoverable data deletion.

In this section we discuss how to incorporate secure deletion and history independence into Ficklebase.

Secure Deletion

Version expiration as described in Section 8.4.7 can leave behind deleted data since many database systems do not physically delete tuple data at delete query execution time [242]. . Instead, deleted tuples are marked for later deletion [242]. Moreover, the database transaction log may retain deleted tuples.

To avoid direct retention of deleted data, we suggest adoption of secure deletion mechanisms suggested by Stahlberg et al. [242]. Stahlberg et al. recommend that after delete query execution, the storage area where deleted tuple content resides be overwritten with zeros.

To prevent retention of deleted data in transaction log, we suggest encryption of individual log entries with unique keys [242]. When a tuple is deleted the key for the corresponding log entry must be erased by overwriting.

History Independence

Data structures such as B-Trees are commonly used by database storage engines. The storage layouts of B-Trees or variations such as B$^+$-Trees are history dependent. Thus, even if

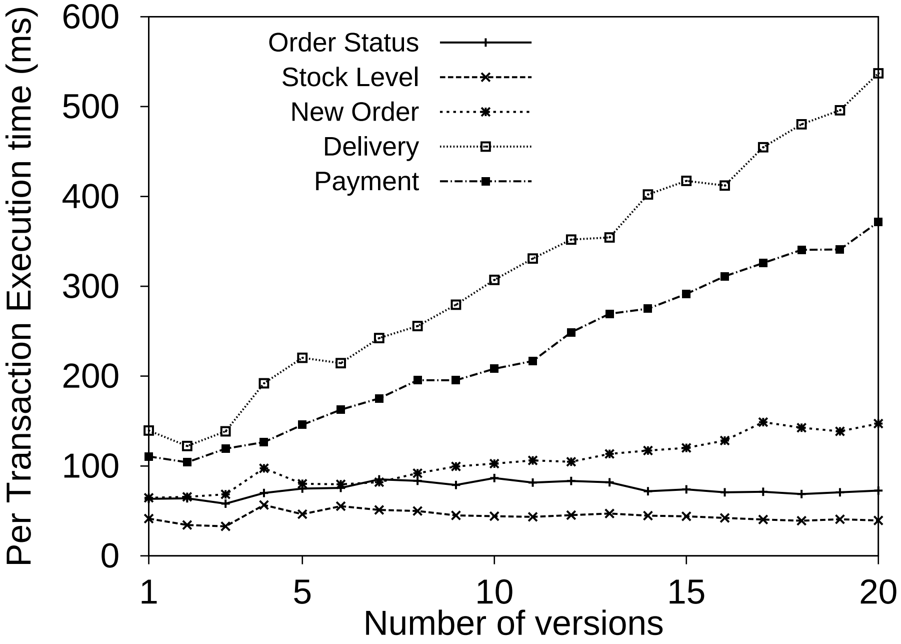


Figure 8.8: Execution times for TPC-C transactions.

untraceable deletion and secure deletion are used, data organization may yet reveal evidence for the past existence of deleted data. We note that although deductions of past existence via data organization are difficult in practice [149,242], they are nonetheless possible and in certain specific cases trivial [149]. For example, an index based on incrementing values. For B^+-Trees in particular, the amount of information regarding past operations decreases as the fanout increases and fanouts in typical usages are usually large. Ideally a fanout of N, where N is the total number of database tuples, results in all index values being stored sorted in a single root node making the B^+-Trees history independent. However, unreasonably large fanouts are not practical.

Data organization can be made delete-evidence-free by using history independent data structures for database indexes. For instance, B-Treaps [114] or B-SkipLists [115]. History independent data structures can be made persistent using the history independent file system described in Chapter 6.

8.7 Experiments

Benchmark

We evaluate the performance of Ficklebase using the TPC-C benchmark [11]. The benchmark data is set up with 16 warehouses giving a total database disk size of 1.5 GB. The database buffer pool size is 200MB. In the initial versioned database, tuples in relations *oorder*, *order line* and *new order* are given random expiration times. The tuples in other relations have fixed maximum expiration times. New tuples inserted during the benchmark transactions are also given random expiration times.

Setup

The database server runs on an Intel Xeon 3.4 GHz, 4GB RAM Linux box with kernel 2.6.18. The server DBMS is off-the-shelf MySQL version 14.12 Distrib 5.0.45. The client system is an Ubuntu VM running on an Intel core i5 at 1.60 GHz with 2 GB RAM. The Ficklebase proxy is implemented in Lua [8] and runs within the MySQL proxy [9] component version 0.8.2. To simulate the TPC-C clients, we use the BenchmarkSQL tool [1]. We modified BenchmarkSQL so that all TPC-C logic is comprised in SQL queries.

Measurements

To measure the TPC-C transaction execution times, we execute $\mathrm{n}_i \times 50$ runs of each TPC-C transaction using a single client and record the average execution time. n_i is the target number of versions the test database instance is set up for. The multiplicative factor $\times$ 50 ensures that targets of insert and update queries are distributed across all versions of the test database. Figure 8.8 shows the results for each of the TPC-C transactions with varying number of versions.

Results

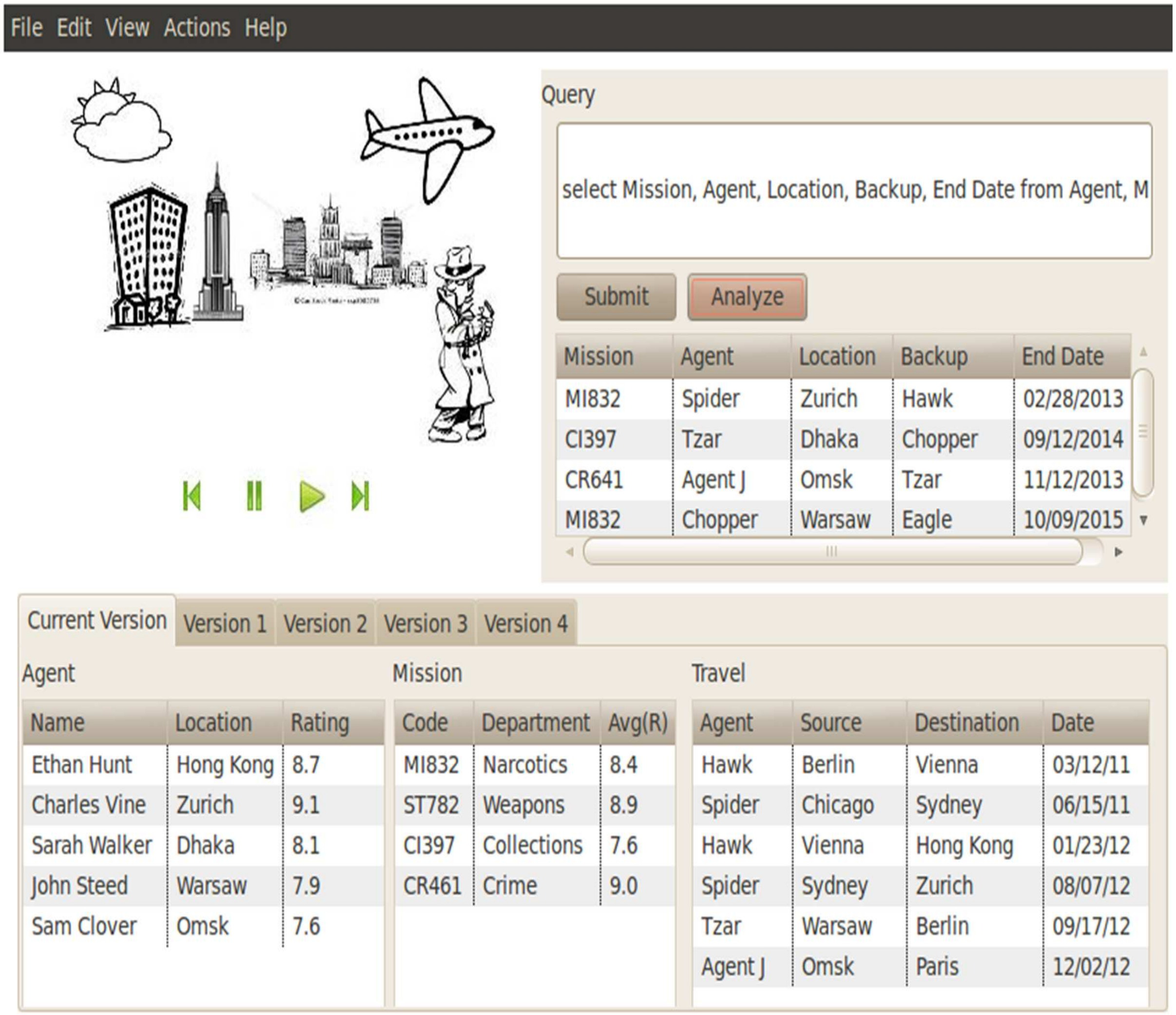
We observe the following overheads for each added version: *New Order* : $\approx$4.9 %, *Delivery* : $\approx$7.8 %, *Payment* : $\approx$6.7 %. *Stock level* and *order status* are both read-only transactions. Note from section 8.4.5 that read-only transactions are executed only on version one. Hence increasing number of versions to not contribute any overheads on these transactions.

Analysis

Version maintenance and query rewriting (Section 8.4) may suggest that each added version should result in an overhead of 1x. If a transaction takes time t to complete execution on one version, then on two versions it would require $2t$ time, on three versions $3t$ and so on. Expectation for 1x overhead is justified since each client transaction is applied to all logical database versions. In practice however, the overheads are far lower as shown in Figure 8.8.

The lower overheads result from database caching and colocation of tuple versions. Updates to individual tuples potentially cause distinct copies to be present in the database. However, in Ficklebase, tuple copies reside close together and are very often located in the same storage node, that is, the same leaf node of the underlying B^+-tree. Hence, queries applicable to a specific version often locate their target tuples in the database caches since the queries were processed for other versions.

In addition, rewriting of create statements adds the *VERSION* attribute only as the terminal field of primary keys or other indexes (Figure 8.4). Thus, even if tuples are valid



select Mission, Agent, Location, Backup, End Date from Agent, M

Mission	Agent	Location	Backup	End Date
MI832	Spider	Zurich	Hawk	02/28/2013
CI397	Tzar	Dhaka	Chopper	09/12/2014
CR641	Agent J	Omsk	Tzar	11/12/2013
MI832	Chopper	Warsaw	Eagle	10/09/2015

Agent

Name	Location	Rating
Ethan Hunt	Hong Kong	8.7
Charles Vine	Zurich	9.1
Sarah Walker	Dhaka	8.1
John Steed	Warsaw	7.9
Sam Clover	Omsk	7.6

Mission

Code	Department	Avg(R)
MI832	Narcotics	8.4
ST782	Weapons	8.9
CI397	Collections	7.6
CR461	Crime	9.0

Travel

Agent	Source	Destination	Date
Hawk	Berlin	Vienna	03/12/11
Spider	Chicago	Sydney	06/15/11
Hawk	Vienna	Hong Kong	01/23/12
Spider	Sydney	Zurich	08/07/12
Tzar	Warsaw	Berlin	09/17/12
Agent J	Omsk	Paris	12/02/12

Figure 8.9: Ficklebase demo client.

in different versions and differ in their *VERSION* attribute, tuples are not dispersed within the storage index.

8.8 Demo Application

The Ficklebase client application (Figure 8.9) serves to demonstrate the following.

- The effects of providing untraceable deletion in a database application.
- Inner workings of Ficklebase including the runtime query rewrites and version maintenance.

- Application of untraceable deletion in a real-world scenario using an animated, storyboard style illustration.

8.9 Conclusions

In this chapter, we introduced untraceable deletion. Untraceable deletion removes all data side effects, thereby eliminating all evidence of the past existence of deleted data from current data. Along with secure deletion and history independence, untraceable deletion is integral to ensure truly irrecoverable data deletion.

We formalize untraceable deletion for relational databases and provide insights into the new functional aspects of untraceable deletion. We also present the design and evaluation of Ficklebase, a relational database which achieves untraceable deletion via versioning and query rewriting.

Chapter 9

Related Work

This chapter discusses related work. In Section 1.4, we gave a brief overview of regulatory compliant data management systems research. In this chapter, we focus on research that is directly related to our contributions.

9.1 Related Work Addressing Privacy Regulations

9.1.1 Queries over Encrypted Data

Range Queries

Hacigümüs et al. [125] propose range query execution over encrypted data using partitioning. Data is partitioned by attribute values. Each partition contains a subset of the total attribute value range. Partitions are stored encrypted with the service provider.

Tuples are then transformed to associate partition identifiers with tuple attribute values. To illustrate, consider a tuple $t =< a_1, a_2, ..., a_n >$, where $a_1, a_2, ..., a_n$ are tuple attributes. The tuple t is transformed to $t' =< E(t), I_1, I_2, ...I_n >$, where $E(t)$ is the encrypted value of tuple t, and I_k is a partition identifier. Each partition identifier I_k identifies the partition that contains the value for attribute a_k.

Mapping functions are used to map attribute values to partition identifiers. Given an attribute value, a mapping function determines the partition that contains the value. To process a range query, the server evaluates mapping functions to identify the partitions that contain the query results. The identified partitions are transferred to the client. The client decrypts and processes the partitions to get the final query results.

The information leaked to the server is claimed to be 1-out-of-s, where s is the partition size. The partitioning scheme thus provides a tradeoff between security and performance. Large partition sizes favor security and small partition sizes favor performance. At one extreme, a single partitioned is maintained. For query processing, the entire partition would be transferred to client for client-side processing. At the other extreme, each attribute value constitutes a partition. Such fine-grained partitioning maximizes performance since only the values that are part of final query results would be sent to client. However, fine-grained partitioning leaks the attribute value distribution to the server.

Damiani et al. [89] propose using tuple-level encryption for outsourced data. Their main contribution is an analysis for the attribute exposure. Attribute exposure is the risk that an untrusted server can deduce an attribute's value for a database that indexes encrypted data. Damiani et al. concluded that the attribute exposure increases with the number of attributes used in an index. Further, the exposure decreases with the increase in database size. Damiani et al. also propose a range query solution, wherein data is stored using a B^+-tree. For confidentiality, the B^+-tree nodes are encrypted. For query processing, a client retrieves the desired encrypted B^+-tree nodes from the server. The client then performs decryption and processing locally.

Client-side processing of queries leads to minimal utilization of server resources, thereby undermining the benefits of outsourcing. Moreover, transfer of entire $B^+ - Tree$ nodes to the client results in significant data transfer costs.

Wang et al. [256] use *order preserving* encryption for querying encrypted xml databases. Order preserving encryption was ruled out by Damiani et al. due to the high attribute exposure. To reduce the attribute exposure, Wang et al. us a technique referred to as splitting and scaling. Splitting and scaling differentiates the frequency distribution of encrypted data from plaintext data. Each plaintext value is encrypted using multiple distinct keys. Then, corresponding values are replicated to ensure that all encrypted values occur with the same frequency, thereby thwarting frequency-based attacks.

Vertical partitioning of relations amongst multiple untrusted servers is suggested by Ganapathy et al. [98]. The goal of vertical partitioning is to prevent access of a subset of attributes by any single server. For example, {*name, address*} can be a privacy-sensitive access pair and query processing should ensure that *name* and *address* are not jointly visible to any single server. The client query is split into multiple sub-queries. Each subquery fetches data from a single server. The client then combines results from multiple servers. TrustedDB is equivalent to the vertical partitioning scheme when the size of the privacy subset is one and hence a single server suffices. In the single-server case, each attribute is encrypted to ensure privacy [70]. Hence, vertical partitioning-based schemes can utilize TrustedDB to optimize query execution at each individual server.

Ciriani et al. [70] introduce the concept of logical fragments to achieve vertical partitioning on a single server. A fragment is a relation. Sensitive attributes in a fragment are encrypted. The paritioning into sensitive and non-sensitive attributes in TrustedDB is equivalent to fragmentation. TrustedDB then is a concrete mechanism to efficiently query a set of fragments. The appropriate set of sensitive attributes in TrustedDB can be determined using the fragmentation concept.

Aggregation Queries

For aggregation queries over encrypted relational databases, Hacigümüs et al. [123] use homomorphic encryption based on privacy homomorphisms [223]. Mykletun et al. [91] have showed that the homomorphic scheme for aggregations [123] is vulnerable to a ciphertext-only attack. Instead, Mykletun et al. [91] propose an alternative scheme based on bucketization [125]. Here, the data owner precomputes aggregate values such as SUM and COUNT for data partitions and stores the precomputed values encrypted at the server. Although precomputation makes processing of certain queries faster it does not significantly reduce client-side processing.

Ge et al. [249] propose to perform aggregation in parallel by simultaneously adding multiple 32-bit encrypted integer values. In parallel addition, two 1024-bit chunks of encrypted data are added at a time. Due to the properties of the Paillier cryptosystem used, each 1024-bit addition involves one 2048-bit modular multiplication. The server computes the encrypted sum of all 1024-bit integers and returns the 2048-bit result to the client. The client decrypts the result into a 1024-bit plaintext, splits the plaintext into 32 32-bit integers, and computes the final sum. Due to the use of extremely expensive homomorphisms [198, 199], the parallel aggregation scheme is expensive (Section 2.4).

Keyword-based Search

Song et al. [241] propose a probabilistic scheme for keyword-based search on encrypted documents. A document is split into fixed-size words. Each word is encrypted using a distinct key. For query processing, the server scans a whole document and matches the encrypted keyword to each encrypted word. Since the matching is probabilistic, false positives are included in the result. False positives are eliminated client-side.

Goh et al. [107] use indexes for keyword search on encrypted data. An encrypted document's index is a Bloom filter [39], which contains all document keywords. Query processing is then equivalent to a bloom filter lookup.

Above solutions are specialized for specific query operations, such as equality predicates [89] , range predicates [125, 256, 260], aggregations [123, 249], and keyword-based search [107, 241]. In TrustedDB, all decryptions are performed within the server-side SCPU and data is processed in plaintext. Plaintext processing of data removes any limitation on the nature of predicates that can be supported. The tamper-proof-SCPU enclosure ensures that no data is leaked even if an adversary gains complete physical access to the server. Hence, in TrustedDB a single strong encryption scheme is used for all query types.

Support for multiple query types

CryptDB [211] uses layered encryption to support multiple query types. The encryption is dynamically adjusted according to client queries. To illustrate, consider an encrypted attribute $l\_quantity$ and two queries, a query $Q_1 = \sigma_{l\_quantity<100}$ and a join query $Q_2 = \bowtie_{l\_quantity>o\_shipqty}$. Both queries share the attribute $l\_quantity$. However, query Q_1 can be efficiently processed if $l\_quantity$ was encrypted using order preserving encryption [217] and query Q_2 requires nondeterministic encryption. Therefore, to enable server-side execution for both queries, attribute $l\_quantity$ needs to be encrypted using two different encryption schemes.

For multiple encryptions on a single attribute, CryptDB uses layered encryption. For example, *l_quantity* is first encrypted using order preserving encryption and then using nondeterministic encryption – $E_{RAND}(E_{OPE-JOIN}(l\_quantity))$. To process query Q_1, the nondeterministic encryption layer is removed and the inner order preserving encryption is used server-side. To remove the nondeterministic encryption layer, decryption keys are communicated to the server. Restoration of the nondeterministic layer then requires reencryption of the entire relation.

In CryptDB, the encryption layers need to be decided before data is uploaded. Hence, prior knowledge of client queries is essential. The encryption layers used by CryptDB include homomorphism, order-preserving encryption, deterministic encryption, and nondeterministic encryption. Deterministic encryption is used to process equality predicates. Homomorphic encryption is used for aggregation queries.

In contrast to CryptDB, TrustedDB uses a single strong encryption scheme for all query types. Hence, TrustedDB does not weaken the encryption as client queries are processed. Moreover, in TrustedDB, no prior knowledge of client queries is required to encrypt data.

9.1.2 Disk-based confidentiality

An alternative adversarial model for an outsourced setting is discussed by Ge et al. [101]. The alternative model requires protecting the confidentiality of data residing on disk. For processing, data is permitted to be decrypted in memory. Data that is encrypted on disk but decrypted for server-side processing compromises confidentiality during the processing interval. Canim et al. [166] analyze the disclosure risks in solutions that decrypt data server-side. Canim et al. also propose a new query optimizer that takes into account both performance and disclosure risk for sensitive data. Individual data pages are encrypted by secret keys that are managed by server-side trusted hardware. The decryption of the data pages and subsequent processing is done in server memory. Here, the goal is to minimize the lifetime of sensitive data and keys in server memory after decryption. In TrustedDB on the other hand, there are no such disclosure risks since decryptions are performed only within the SCPU.

9.1.3 Use of Trusted Hardware for Data Confidentiality

Iliev et al. [25] use SCPUs to retrieve X509 certificates from an outsourced database. Each certificate has a unique key and a client queries for a certificate by specifying a key. To improve performance, Smith et al. [226] use multiple SCPUs for key-based search. The entire database is scanned by the SCPUs to locate matching records.

Agrawal et al. [216] propose a SCPU-based solution for join queries. The entire database is scanned by the SCPU to process a join. The approach is limited by available server-SCPU bandwidth, which is low in practice, ≈10 MBps in our setup.

Chip-Secured Data Access [165] uses a smart card for query processing and to enforce access rights. The client query is split, such that the server performs majority of computation. In follow-up work, GhostDB [189] proposes to embed a database inside a USB key equipped with a CPU. GhostDB allows linking of private data carried on the USB Key and public data available on a server. GhostDB ensures that the only information revealed to the server is the query issued and the public data accessed. For efficiency, GhostDB organizes the schema as a tree and builds additional indexes on data.

Both Chip-Secured Data Access and GhostDB are limited to small datasets since all query-processing data resides within the trusted hardware module used. Queries executing within the trusted hardware cannot use external storage to store intermediate query results. In TrustedDB on the other hand, the SCPU database engine can generate intermediate results. The intermediate results are encrypted and stored externally. Database pages can be swapped out of the trusted hardware to external storage during query processing. Therefore, TrustedDB can process queries that access large amounts of data.

Bhattacharjee et al. [36] use SCPUs for cross-enterprise data mining. They port a database engine to the SCPU. The SCPUs fetch data from external enterprises. Queries are processed entirely by the SCPUs. Host server cycles are not used. To fetch data from external sources, the SCPUs use secure jdbc connections. Processing queries entirely within the SCPU is up to 40x slower than plaintext query processing. The 40x factor holds when the SCPU access data from the host server. Hence, for jdbc connections the slowdown is expected to be much higher than 40x. In contrast, query execution times in TrustedDB are 1.03x-10x slower than plaintext query processing.

In Matchbox [147], requests are made to the SCPU in the form of contracts. A contract is a predefined list of operations to be performed by the SCPU. All parties including data owners and potential recipients of processing results must sign the contract before-hand. Defining contracts requires prior knowledge of all application logic and thus is less suitable for dynamic outsourced databases.

NetDB [124] simply deploys a SCPU as a cryptographic accelerator. The SCPU hardware module is used to perform cryptographic operations efficiently. The SCPU is not used for query processing.

9.2 Related Work Addressing Audit Regulations

9.2.1 Query Authentication (QA)

Existing QA solutions can be classified as either *tree-based* or *signature-based*. The two categories differ in the data structures used for the ADS and the VO and hence in the query execution and verification.

In tree-based approaches, the ADS is constructed as a tree. For example, MB-tree [155] and VB-tree [201]. As part of query execution, service provider traverses the tree and gathers the nodes that form the VO, which is sent to the client along with the query results. The client reconstructs the traversal path used in query execution to verify correctness and completeness.

Signature-based approaches provide a mechanism to verify the ordering between tuples when using specific search attributes. For QA, an authenticated chain of unforgeable signatures is constructed by the data owner. At query time, the service provider gathers the

Name	Year	QCT	QCP	Ops	Updates
Tree Based					
MHT [85]	2003	✓	✓	S,R	×
VBT [201]	2004	✓	✓	S,R	✓
EMBT [155]	2006	✓	✓	S,R,J	✓
Singh et al [240]	2008	✓	✓	S	✓
XBT [204]	2009	✓	✓	R	✓
AIM [269]	2009	✓	✓	S,R,J	×
MRT [270]	2009	✓	✓	S_p	✓
AABT [156]	2010	✓	✓	A	✓
MR-SKY [160]	2011	✓	✓	S_p	×
Yang et al [271]	2011	✓	✓	R	×
Jain et al [141]	2012	✓	✓	R	✓
Signature Based					
AGS [181, 182]	2004	✓	×	S,R	×
DSAC [185]	2005	✓	✓	S,R	✓
Pang et al [200]	2005	✓	✓	S,R,J	✓
VKDT,VRT [66]	2006	✓	✓	S_p	×
Goodrich et al [118]	2008	✓	✓	R	✓
Pang et al [203]	2009	✓	✓	S,R,J	✓
VNA [137]	2010	✓	✓	S_p	✓
Others					
DICT [117]	2001	✓	✓	K	✓
AUDIT [268]	2007	✓	×	S,J	✓
Nath et al. [186]	2009	✓	✓	R,A	✓
HLT [275]	2012	✓	✓	S,R,J,A	✓
Zhou et al. [276]	2013	✓	✓	R	✓

Table 9.1: Summary of existing approaches (QCT - Query Correctness, QCP - Query Completeness, S - Select, R - Range, J - Join (Equi,<,>), A - Aggregation, S_p - Spatial, K - Key lookup).

signatures of all tuples that comprise the contiguous range query result. The set of signatures comprises the VO. Since each tuple is now linked to its predecessor and successor in an unforgeable manner, the client can verify that no tuple is either illicitly inserted or omitted from the query result.

Table 9.1 summarizes existing QA solutions.

Tree-based Solutions

The QA approach designed by Devanbu et al. [85] forms the basis for most tree-based range query approaches. Devanbu et al. utilize a Merle hash tree (MHT) for query processing on the provider's site. The MHT root node is signed by the owner and distributed to the clients before uploading the MHT to the provider. In response to a client query, the provider delivers the actual query results and relevant nodes from the MHT such that the client can reconstruct the root node. The client then verifies the signature on the root node and is thereby assured that the query was processed correctly.

The MHT approach can be extended and applied to a B-tree. The ADS is then referred to as a MB-tree (MBT). In a MBT, the hash for a leaf node is constructed by concatenating the hashes all tuples contained in the leaf, that is, $H_{li} = H(H(t_1)||H(t_2)||...||H(t_k))$, where li is a leafd node, $t_1, t_2, ..., t_k$ are tuples contained in l_i, and $H(\cdot)$ is a cryptographic hash function. Each non-leaf node's hash is concatenation of the hashes of its children. Then, the VO consists of all the additional node hashes required by the client to reconstruct and verify the root hash.

The first approach to deploy an MHT-design-based B-tree is Verifiable B-tree (VBT) [201]. VBT considers an edge computing model, wherein all hashes are computed, signed, and later updated by a trusted central server and then distributed to the edge servers. The VO then consists of all authenticated nodes up to the root node of the subtree that do not envelope the tuples present in the query result. The client constructs the subtree and verifies the signature on the subtree's root. The optimization here lies in the fact that the VO need not contain the path up to the root of the entire B-tree, like in MHT.

An MBT-based ADS is also used by Li et al. [155]. Here, each individual node of the MBT in-turn stores an embedded MBT. The approach is thus referred as Embedded Merkle B-tree (EMBT). The embedded tree aids in quickly constructing the composite hash of the node being traversed as part of query execution, thereby reducing the number of B^+-tree read operations required to construct the VO.

A standard MHT is used by Singh et al. [240] in a slightly different context. Here, the provider periodically commits the state of the database to the client and QA is performed against the last committed version. Limiting QA to the last committed version provides weaker security but allows more frequent updates. The ADS used is a MHT and the hash for a leaf node is computed as $H(id||H(A_{i1}||S(A_{i1}, SK_{DO}) \dots H(A_{ik}||S(A_{ik}, SK_{DO}))$, where id is a unique identifier for the tuple, $A_{i1}, ...A_{ik}$ are the tuple attribute values, and SK_{DO} is the data owner's secret key. Support for projections is added by including the tuple attribute values in the composite hash.

For join processing, Pang et al. [201] suggest the use of materialization. Here, the entire cross product is materialized and an ADS is built on the materialized result. Materialization is inefficient both in terms of storage and for update operations.

Pang et al. [200] extend range query authentication to joins, explained as follows. Consider the relations R_1, R_2, and the join query $R_1 \bowtie_{R_1.A=R_2.B} R_2$. The data owner then constructs an ADS on both relations $R_1.A$ and $R_2.B$. Also, assume that R_1 is smaller. First the provider sends R_1 to the client along with the authentication data for R_1. Then, for each tuple in R_1 the provider performs a range query on R_2 to find the matching tuples. The VO resulting from each range query is appended to construct the VO for the entire join.

The first comprehensive QA solution for joins is Authenticated Index Merge Join (AIM) [269]. According to our survey of published experimental results (Section 3.4.3), AIM is also the most efficient QA solution for join queries.

To understand AIM, consider the join query $R_1 \bowtie_{R_1.A=R_2.B} R_2$. In AIM, the ADS used is MBT. A MBT is constructed on the join attribute of each relation. In our example, a separate MBT is constructed for $R_1.A$ and $R_2.B$. Join query processing proceeds as follows. The provider locates the first tuple from R_1 that is part of the query result. For the first tuple in R_1, the provider performs an index traversal and leaf scan on the ADS for R_2 locating the matching and boundary tuples, which are included in the VO. The server then uses each boundary tuple located in R_2 as a target and performs a search on R_1. Now, boundary tuples from R_1 are added to the VO. The process is repeated until all result tuples are found.

A tree-based QA solution for aggregation queries is proposed by Li et al. [156]. The solution is referred to as Authenticated Aggregation B-tree (AABT). In AABT, each tree node stores the aggregated sum α of its child nodes on the value of the search attribute, that is, $\alpha = \alpha_1 + \alpha_2 + ... + \alpha_k$ where α_1, α_2 ... α_k are the aggregated values of the individual child nodes. In addition, a node stores the hash $H(\eta_1||\alpha_1||...||\eta_k||\alpha_k)$, where η_1, η_2 ... η_k are the child nodes. The hash $H(\eta_1||\alpha_1||...||\eta_k||\alpha_k)$ is also included in the VO. The precomputed aggregation values allow client-side verification of aggregation queries. QA solutions other than AABT and TrustedDB do not authenticate aggregations server-side. Instead, data is transferred for client-side aggregation.

Signature-based Solutions

Mykletun [182] designed the first approach to use signatures for range QA. The approach is referred to as NBS. NBS addresses only query correctness. In NBS, the data owner signs individual tuples before uploading to the provider. The signatures of all tuples constitute the ADS. The provider includes tuple signatures in the query result sent to clients. Tuple signatures in query result constitute the VO. Client-side processing then involves signature verifications.

NBS uses signature aggregation [181, 182] to combine multiple tuple signatures into a single signed message, thereby resulting in a small, constant-sized VO. Signature aggregation applies to both RSA and BGLS [44] signature schemes. We describe signature aggregation for RSA below.

Consider a query Q with a result that includes the set of tuples $\{t_1,t_2...t_k\}$. Then a single aggregated signature-based VO is constructed by the service provider as follows

$$\mathsf{S}_{\mathsf{VO}} = \prod_{i=1}^{k} S(H(t_i), SK_{DO})(mod\ n) \tag{9.1}$$

where SK_{DO} is the data owner's secret key. Client verification then tests the following equality

$$(\mathsf{S}_{\mathsf{VO}})^e = \prod_{i=1}^{k} H(t_i)(mod\ n) \tag{9.2}$$

The tuple signatures $S(H(t_i), SK_{DO})$ are computed by the data owner and uploaded to the provider.

DSAC [185] extends NBS [182] to provide completeness for range queries. In NBS, the ADS consists of individual tuple signatures. In DSAC, each tuple signature also includes the tuple's immediate predecessor. For example, the signature of tuple t_i is computed as $S(H(t_i)||H(t_{i-1}), SK_{DO})$, where t_{i-1} is the predecessor of t_i when sorted on the search attribute. Including the predecessor forms an authenticated chain of all database tuples ordered on the search attribute. A client then verifies that the set of tuples received in the result do form a valid chain.

Pang et al. [200] designed a signature-based scheme for range and join QA. Unlike DSAC [185], Pang et al. include a tuple's immediate predecessor and successor in the query result. For example, the signature of tuple t_i is computed as $S(H(t_{i-1})||H(t_i)||H(t_{i+1})), SK_{DO})$, where t_{i-1} and t_{i+1} are the predecessor and successor, respectively, of tuple t_i when sorted on the search attribute. Signature aggregation is also applicable here to reduce the VO size.

Signature-based schemes are inefficient for join processing as they result in large VO sizes. To illustrate, consider the join query $R_1 \bowtie_{R_1.A=R_2.B} R_2$. For each tuple from R_1 present in the result, the VO contains the boundary tuples for the matching tuple in R_2. Also, for each tuple in R_1 that is not part of the result we need a proof that no matching tuple exists in R_2. Again, the VO will contain two boundary tuples showing that no matching tuple exists in R_2. The resulting VO thus becomes large.

QA for key-based lookups is proposed by Goodrich et al. [117]. The ADS is constructed using a skip list [213]. Verification is similar to tree-based approaches. Client verifies the skip-list traversal path used to locate tuples. The VO is kept constant using commutative hashing.

Xie et al. [268] propose a probabilistic QA scheme for range and join queries. The data owner incorporates fake tuples in the encrypted dataset. The secret function used to distinguish between fake and real tuples is shared only with clients. Clients probabilistically verify completeness by checking whether all fake tuples satisfying the query are present in the result set. Unfortunately, a probabilistic design makes it difficult to detect omissions of a very small fraction of tuples from a large dataset.

Update Operations in QA

QA complicates update operations on the outsourced database since changes made to the database tuples involve modifications in the related ADS. Hence, existing QA solutions either do not address update operations [181, 240] or assume fairly static or infrequently updated databases [85, 169, 182, 185].

Solutions that do provide QA, only permit the data owner to perform updates as follows. The data owner queries and downloads the tuples to be updated along with the ADS. The data owner then performs updates to both tuples and ADS locally. The updated tuples and ADS are reuploaded to the provider. Finally, the data owner recomputes a new signature for the ADS root and distributes it to clients. As discussed in Section 3.5, owner-side updates is inefficient for large number of clients.

Further, avoiding replay attacks in case of owner-side updates also results in inefficiency. A replay attacks occurs when the service provider uses old signatures to return stale results

Data Structure	Year	Ops	Runtime
2-3 Tree [173]	1997	I,L,D	$O(\log N)$
Hash Table [184]	2001	I,L	$O(log(1/(1-\alpha)))$
Hash Table [38]	2007	I,L,D	$O(1/(1-\alpha)^3)$
Ordered Dictionary [38]	2007	I,P,D	$O(\log\log N)$
Order Maintenance [38]	2007	I,C,D	$O(1)$
Hash Table [183]	2008	I,L,D	I,D → $O(\log N)$ S → $O(1)$
B-Treaps [114]	2009	I,D,R	$O(log_B N)$
B-SkipList [115]	2010	I,D,R	$O(log_B N)$
R-Trees [250]	2012	I,D,R	not available

Table 9.2: Summary of history independent data structures. $\alpha \leftarrow$ load factor, $N \leftarrow$ number of keys, $B \leftarrow$ block transfer size. Also, I : insert, L : lookup, D : delete, R : range, P : predecessor, C : compare.

to clients. To avoid replay attacks, the data owner locks the entire database for the duration of updates. Since locking the entire databases is inefficient, Pang et al. [201] permit only intermittent updates.

To overcome inefficiencies of database locking, Li et al. [155] suggest batching of updates. Updates to tuples that reside close together, that is, stored in the same or adjacent ADS leaf nodes are performed in a single batch. Since adjacent leaf nodes share parent nodes on the path to the tree root node, the number of ADS tree nodes modified is reduced.

For signature-based approaches, an update to a tuple requires recomputing the signatures of neighboring tuples. A concrete update mechanism is designed in DSAC [185]. In DSAC, the owner downloads the tuple and signatures to be modified, performs modifications locally, and reuploads modified tuples and ADS to the provider.

In Section 3.5, we discussed the advantages of SCPU-based QA over existing tree and signature-based solutions. We do not repeat the discussion here and instead point to Section 3.5.

9.3 Related Work Addressing Data Retention Regulations

9.3.1 History Independence

Existing history independent data structures are summarized in Table 9.2. History independent data structures have several applications including incremental signature schemes [183], privacy in voting systems [38, 178, 179, 183], performing updates without revealing intermediate states [184], debugging parallel computations [38], reconciliation of dynamic sets [183], and untraceable deletion (Section 8.6).

Micciancio et al. [173] designed a history independent 2-3 tree, referred to as an oblivious 2-3 tree. In a 2-3 tree, each non-leaf leaf node has either two or three children. In the oblivious 2-3 tree, the choice of whether to have two or three child nodes for each non-leaf is randomized. When the oblivious 2-3 tree undergoes a local modification, random choices are made for a small number of neighboring nodes in the leaf-to-root path. The tree is rebalanced based on the new random choices. Micciancio et al. show that the probability distribution of nodes in an oblivious 2-3 tree is independent of the sequence of operations.

Naor et al. [184] were the first to introduce weak and strong history independence. Naor et al. [184] then designed a strongly history independent hash table that supports search and insert operations. The hash table construction is similar to linear probing [171] except for the collision resolution. In case of collision between two keys a priority function is used. The key with higher priority takes the bucket. The key with lower priority is relocated to the next bucket in the probe sequence. Naor et al. [184] show that for a given set of keys, the hash table layout is same irrespective of the insert sequence.

In follow-up-work, Golovin et al. [38] designed a history independent hash table that also supports deletion. The hash table is based on the stable matching property of the *Gale-Shapley Stable Marriage* algorithm [97] detailed in the following.

Stable Marriage Algorithm: Let M and W be a set of men and women respectively, $|M| = |W| = n$. Also, let each man in M rank all women in W as per his set of preferences. Similarly, each women in W ranks all men in M.

The goal of the stable marriage algorithm is to create n matchings (m, w), where $m \in M$ and $w \in W$, such that no two distinct pairs (m_i, w_j) and (m_k, w_l) exist where m_i ranks w_l higher than w_j and w_l ranks m_i higher than m_k. If no such pairings exists, then all matchings are considered stable.

The algorithm works as follows. In each round, a man m proposes to one woman at a time based on his ranking of W. If a woman w being proposed to is unmatched, then a new match (m, w) is created. If the woman w is already matched to some other man m', then one of the following two occurs.

1. If w ranks m higher than m', then the match (m', w) is broken and a new match (m, w) is created.

2. If w ranks m lower than m', then m proposes to the next woman based on his rankings.

The algorithm terminates when all men are matched.

Gale et al. [97] show that if all the men propose in decreasing order of their preferences (ranks) then the resulting stable matching is unique. The matchings are unique even if the selection of a man m who gets to propose in each round is arbitrary.

Golovin et al. [38] use the above unique matching property of the Stable Marriage algorithm to construct a history independent hash table as follows.

1. The set of keys to be inserted are considered as the set of men.

2. The set of hash table buckets are considered as the set of women.

3. Each key has an ordered preference of buckets and vice versa.

4. The preference order of each key is the order in which the buckets are probed for insertion, deletion and search.

5. In case of a collision between two keys, the key which ranks higher on the bucket's preference takes the slot. The lower ranked key is relocated to the next bucket in its preference list.

(1) - (5) ensure that the layout of keys in the hash table is the same irrespective of the sequence of key insertions and deletions, thereby making the hash table history independent.

Hartline et al. provide new definitions for weak and strong history independence based on probability distributions. The new definitions are equivalent to the ones provided by Naor et al. [184]. Hartline et al. then briefly analyze various properties of history independence. The significant result by Hartline et al. is the proof for necessity of canonical representations for strong history independence (SHI). The proof builds on the case that in the absence of canonical representations, an adversary can distinguish an empty operations sequence from a non-empty sequence, thereby breaking SHI.

History Independence on Write Once Storage

The history independent data structures in Table 9.2 are designed for the RAM model and assume a rewritable storage medium. Molnar et al. [178] designed a history independent solution for write-once storage. For example, storage in voting machines. The construction is based on the observation by Naor et al. [184], which states that a lexicographic ordering of elements in a list is history independent. However, write-once memories do not allow in-place sorting of elements and hence cannot maintain a sorted list. Instead, Molnar et al. use copy-over lists [184]. In a copy-over list, when a new element is inserted, the current list is deleted and a new sorted list is stored. Copy-over lists therefore require $O(K^2)$ space to store K keys.

An alternative solution designed by Molnar et al. uses randomization. Each new element is inserted at a random location on the write-once storage. If a location is free, the element is inserted in the free location. If a location is already occupied, a new random location is selected. Although simple and space-efficient, the random approach requires random bits to be hidden from the adversary. As Molnar et al. suggest hiding random bits from the adversary may not be possible in the target scenario involving voting machines in poll booths.

Moran et al. [179] design a history independent solution for write-once storage that requires only linear storage. Specifically, their solution requires $O(K \cdot polylog(N))$ space, where K is the number of keys to be stored and N is the total number of keys in the keys' domain. To reduce storage requirement, Moran et al. organize the write-once storage as a history independent hash table. Each hash table bucket is a copy-over list. As suggested by

Naor et al. [184], a copy-over list is history independent. Hence, the hash table as a whole also preserves history independence. In case a bucket overflows, one key from the bucket is relocated to the adjacent bucket. For a given set of keys, the key selected for relocation is always the same.

9.3.2 Secure Deletion

The goal of secure deletion is to prevent direct recovery of deleted data from the storage medium. Prior work achieves secure deletion either by overwriting or by using encryption. We cover both techniques in the following.

Secure Deletion by Overwriting

Secure deletion by overwriting was first suggested by Gutmann et al. [121]. Gutmann et al. concluded that storage locations need to be overwritten multiple times to ensure secure deletion. In fact, Gutmann et al. suggest up to 35 overwrites to prevent recovery of deleted data from storage.

Later Joukov et al. [145] claim that a single overwrite is sufficient to prevent software-based recovery of delete data. Software-based recovery implies that an adversary does not have physical access to the storage medium. For example, an insider that can obtain a dump of entire storage over a remote connection but cannot physically access the storage device. Joukov et al. [145] also propose extensions to the Ext3 [5] file system for overwriting files' data and metadata on deletion.

Wright et al. [266] investigated the possibility of recovering deleted data using an electron microscope. They concluded that although recovery of an individual bit is possible, the likelihood of recovering sizeable data using an electron microscopy is negligible. Wright et al. suggest that the case for multiple overwrites [121] holds only when a copy of deleted data is available. Then, the recovery process entails determining whether the storage contained the same data in the past. Without an available copy, recovery of deleted data is infeasible.

An extension for the Ext2 file system is provided by Bauer et al. [34]. Unlike the Ext3 extension by Joukov et al. [145], Bauer et al. use asynchronous overwriting. Asynchronous overwriting causes less interference with user tasks by delaying overwriting. Security is thus compromised for a short interval from delete time to overwrite time.

Chow et al. [68] target secure deletion for main memory. They propose to reduce the lifetime of data in main memory. The solution is referred to as *secure deallocation*. In secure deallocation, the heap and stack contents are overwritten with zeros to prevent recovery of deleted data.

Secure Deletion by Encryption

Lee et al. [153] use encryption to achieve secure deletion for a NAND flash file system. Each data block is encrypted using a distinct encryption key. A special area on disk is reserved for key storage. To delete a data block, the corresponding key is overwritten by zeroes. For a strong encryption system, overwriting the key suffices to ensure a secure data block delete.

Zhu et al. [278] also use encryption for secure deletion. The solution by Zhu et al. is similar to the work by Lee et al. [153]. The only difference is that Zhu et al. [278] target deletion of document index entries and not file system data blocks.

Peterson et al. [209] provide secure deletion in a versioning file system. Similar to the work of Lee et al. each data block is encrypted using a random key. Random keys are authenticated using a single per-file master key. To delete an entire file, the master key is securely deleted by overwriting.

9.3.3 Compensating Transactions

A *compensating transaction* undoes the effect of a previously committed transaction without resorting to cascading aborts. Hence, compensating transactions can potentially be utilized to undo the side effects of deleted tuples as in Ficklebase. However, compensating transactions are application-dependent [152], need to be predefined, and can only be minimally automated. Ficklebase on the other automates untraceable deletion at the database level.

Compensating transactions also need to be manually predefined. The existence of an automated technique to generate a compensating transaction for any given transaction is yet unknown [152]. Hence, use of compensating transactions requires increased development effort from database application developers.

Korth et al. [152] discuss various guidelines for designing compensating transactions. Colombo et al. [73] use an online bookstore example to review several notations for compensating transactions including syntax and semantics.

Sagas [47] is a flow composition language which achieves atomicity based on compensation. In case of a long running transactions that fails to complete, compensation is used to undo the transaction's effects. Sagas also addresses parallel composition, nesting and exception handling.

Similar to compensating transactions Ataullah et al. [29] design a system wherein application developers can specify destructive policies for business records. The policies are stored and later executed as database stored procedures. Policy execution is triggered by predefined meta-policies. The meta-policies take data lifetime into account. The drawback is that destructive policies need to be predefined not unlike compensating transactions.

9.3.4 Statistical Databases

Statistical databases [45] are used to maintain statistical data in an online analytical processing (OLAP) model. The security concern for statistical databases is to prevent an adversary from deducing specific information by issuing statistical queries. Typical approaches to prevent data leaks include limiting support for aggregation queries, refusal to answer queries with small result sets, and returning range results instead of specific values [67, 82].

At first glance it may seem that statistical databases achieve untraceable deletion. The perception may arise since deletion of a particular data item will result in new updated statistics that exclude the deleted data. However, statistical databases are designed for the OLAP model and do not support delete operations.

9.3.5 Data Degradation

Data Degradation [28] is a work-in-progress to address deletion of sensitive data. The goal of data degradation is to gradually degrade sensitive information eventually making it unrecoverable.

A solution providing data degradation is InstantDB [27]. In InstantDB, a data item is degraded in steps from specific to more general values. For example, an address field may initially contain the entire detailed address. In the next iteration the street part is erased. A following iteration erases the state and zip leaving only the country code and so on.

9.3.6 Information Flow Control

Although not dealing with removal of side effects, *information flow control* and related implementations [79,225,273] can potentially play a complimentary role to Ficklebase. Information flow control enables tracking of sensitive data across system components. Information flow control can therefore be used to extend Ficklebase for cross-system untraceable deletion. When a tuple is deleted in Ficklebase the flow control information can be used to remove remote side effects.

Chapter 10

Conclusions

This thesis has been driven by the motivation for low-cost, efficient, and increasingly automated regulatory compliance in data management. This thesis describes the design and implementation of several relational databases and file systems that target specific requirements from privacy, audit, and retention Regulations. The systems increase efficiency and lower costs of regulatory compliance through the use of novel cryptographic and system security constructs.

Through TrustedDB and CorrectDB, this thesis shows the cost and efficiency benefits of trusted hardware for query processing in outsourced databases. To make the use of trusted hardware practical, this thesis overcomes several challenges in trusted hardware-based application development. This thesis also presents new query optimization techniques for a trusted hardware model.

The contributions of this thesis are not limited to system designs but also cover theoretical results. This thesis develops the theoretical foundations for history independence via an exploration of basic concepts, such as abstract data types, data structures and memory representations. The thesis then expands history independence by introducing the ΔHI framework. ΔHI helps conceptualizing new history independence notions and also incorporates both weak and strong history independence. Using ΔHI, the thesis outlines several new history independence notions and presents mechanisms to analyze the history preserved in existing data structures that were designed without history independence in mind.

This thesis bridges the gap between theory and practice of history independence by outlining a generic process for history independent system design. The process is used in the thesis itself to design and evaluate two history independent file systems, HIFS and DAFS.

The thesis focuses on both security and practicality simultaneously. For query authentication (QA), this thesis proposes ConcurDB, a concurrent QA solution for outsourced databases that meets practicality by selecting the best available design tradeoffs for efficiency and security. For file systems, along with history independence, the thesis ensures data locality for efficiency and journaling for failure recovery.

Finally, to achieve truly irrecoverable deletion, the thesis introduces the concept of untraceable deletion and designs Ficklebase, a relational database that provides untraceable deletion via versioning and query rewriting.

Bibliography

[1] BenchmarkSQL. Online at http://sourceforge.net/projects/benchmarksql/. 167, 190

[2] Current Population Survey (CPA). Online at http://www.bls.gov/cps/. 177

[3] Deleted but not gone. Online at http://www.nytimes.com/2005/11/03/technology/circuits/03basics.html. 172

[4] EU's Data Protection Directive. Online at http://ec.europa.eu/justice/data-protection/index_en.htm. 172

[5] Ext3 file system, http://en.wikipedia.org/wiki/ext3. 130, 135, 138, 141, 145, 147, 205

[6] FIPS PUB 140-2, Security Requirements for Cryptographic Modules. Online at http://csrc.nist.gov/groups/STM/cmvp/standards.html#02. 19

[7] List of corporate collapses and scandals. http://en.wikipedia.org/wiki/List_of_corporate_collapses_and_scandals. 7

[8] Lua programming language. Online at http://www.lua.org/. 190

[9] MySQL Proxy. Online at http://forge.mysql.com/wiki/MySQL_Proxy. 190

[10] The UBENCH Toolkit. Online at http://www.phystech.com/download/ubench.html. 23

[11] TPC-C Benchmark. Online at http://www.tpc.org/tpcc/default.asp. 167, 168, 189

[12] TPC-H Benchmark. Online at http://www.tpc.org/tpch/. 31, 46, 48, 65

[13] Data Protection Act. http://www.legislation.gov.uk/ukpga/1998/29/contents, 1998. 4, 6

[14] Federal Information Security Management Act. https://www.dhs.gov/federal-information-security-management-act-fisma, 2002. 3, 172

[15] Corporate Law Economic Reform Program Act. http://www.asic.gov.au/clerp9, 2004. 4, 5

[16] Financial Instruments and Exchange Act.
http://www.fsa.go.jp/common/law/fie01.pdf, 2006. 5

[17] IBM 4764 PCI-X Cryptographic Coprocessor. Online at
http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml, 2007. 18, 27

[18] Massachusetts identity theft protection regulation.
http://www.mass.gov/ocabr/docs/idtheft/201cmr1700reg.pdf, 2008. 3, 6

[19] IBM 4765 PCIe Cryptographic Coprocessor. Online at
http://www-03.ibm.com/security/cryptocards/pciecc/overview.shtml, 2010. 79

[20] 106th United States Congress. Gramm-Leach-Bailey Act.
http://www.gpo.gov/fdsys/pkg/PLAW-106publ102/pdf/PLAW-106publ102.pdf, 1999. 5, 6, 172

[21] Nabil R. Adam and John C. Worthmann. Security-control methods for statistical databases: a comparative study. *ACM Comput. Surv.*, 21(4):515–556, December 1989. 8

[22] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu 0002. Two can keep a secret: A distributed architecture for secure database services. In *CIDR*, pages 186–199, 2005. 50

[23] Dakshi Agrawal and Charu C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 247–255, New York, NY, USA, 2001. ACM. 8

[24] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 439–450, New York, NY, USA, 2000. ACM. 8

[25] Alexander Iliev and Sean W Smith. Protecting Client Privacy with Trusted Computing at the Server. *IEEE, Security and Privacy*, 3(2), Apr 2005. 52, 196

[26] Nicolas Anciaux, Luc Bouganim, Harold van Heerde, Philippe Pucheral, and Peter M. G. Apers. Instantdb: Enforcing timely degradation of sensitive data. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 1373–1375, Washington, DC, USA, 2008. IEEE Computer Society. 10

[27] Nicolas Anciaux, Luc Bouganim, Harold van Heerde, Philippe Pucheral, and Peter M. G. Apers. Instantdb: Enforcing timely degradation of sensitive data. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*,

ICDE '08, pages 1373–1375, Washington, DC, USA, 2008. IEEE Computer Society. 207

[28] Nicolas Anciaux, Luc Bouganim, Harold van Heerde, Philippe Pucheral, and Peter M.G. Apers. Data degradation: making private data less sensitive over time. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 1401–1402, New York, NY, USA, 2008. ACM. 10, 207

[29] Ahmed A. Ataullah, Ashraf Aboulnaga, and Frank Wm. Tompa. Records retention in relational database systems. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 873–882, New York, NY, USA, 2008. ACM. 10, 206

[30] Sumeet Bajaj and Radu Sion. Trusteddb: a trusted hardware based database with privacy and data confidentiality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 205–216, New York, NY, USA, 2011. ACM. 77

[31] Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware based outsourced database engine. *PVLDB*, 4(12):1359–1362, 2011. 53

[32] Sumeet Bajaj and Radu Sion. Ficklebase: Looking into the future to erase the past. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, 0:86–97, 2013. 106

[33] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 203–212, New York, NY, USA, 2005. ACM. 121

[34] Steven Bauer and Nissanka B. Priyantha. Secure data deletion for linux file systems. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 12–12, Berkeley, CA, USA, 2001. USENIX Association. 10, 172, 205

[35] Mihir Bellare. New proofs for nmac and hmac: Security without collision-resistance. pages 602–619. Springer-Verlag, 2006. 35

[36] Bishwaranjan Bhattacharjee, Naoki Abe, Kenneth Goldman, Bianca Zadrozny, Chid Apte, Vamsavardhana R. Chillakuru and Marysabel del Carpio. Using secure coprocessors for privacy preserving collaborative data mining and analysis. In *Proceedings of DaMoN*, 2006. 52, 197

[37] M. Blaze. A Cryptographic File System for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, VA, 1993. ACM. 51

[38] Guy E. Blelloch and Daniel Golovin. Strongly history-independent hashing with applications. In *Proceedings of IEEE Symposium on Foundations of Computer Science*, FOCS '07, pages 272–282. IEEE Computer Society, 2007. 104, 106, 135, 142, 143, 152, 157, 159, 166, 202, 203

[39] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. 51, 195

[40] Avrim Blum, Katrina Ligett, and Aaron Roth. A learning theory approach to non-interactive database privacy. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, pages 609–618, New York, NY, USA, 2008. ACM. 8

[41] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. SFCS, 1991. 86, 87, 89

[42] Alexandra Boldyreva, Nathan Chenette, and Adam O'Neill. Order-preserving encryption revisited: improved security analysis and alternative solutions. In *Proceedings of the 31st annual conference on Advances in cryptology*, CRYPTO'11, pages 578–595, Berlin, Heidelberg, 2011. Springer-Verlag. 8, 16

[43] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proceedings of Eurocrypt 2004*, pages 506–522. LNCS 3027, 2004. 51

[44] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques*, pages 416 – 432. Springer-Verlag, 2003. 200

[45] Claus Boyens, Oliver Gnther, and Hans j. Lenz. Statistical databases. 206

[46] R. Brinkman, J. Doumen, and W. Jonker. Using secret sharing for searching in encrypted data. In *Secure Data Management*, 2004. 52

[47] Roberto Bruni, Hernán Melgratti, and Ugo Montanari. Theoretical foundations for compensations in flow composition languages. *SIGPLAN Not.*, 40(1):209–220, January 2005. 206

[48] Niv Buchbinder and Erez Petrank. Lower and upper bounds on obtaining history independence. *Inf. Comput.*, 204(2):291–337, February 2006. 105, 126

[49] James L. Buckley. Family Educational Rights and Privacy Act. http://www.ed.gov/policy/gen/guid/fpco/ferpa/index.html, 1974. 3, 5, 172

[50] Simon Byers. Scalable exploitation of, and responses to information leakage through hidden data in published documents. *ATT Research*, 2003. 172

[51] Ran Canetti, Uri Feige, Oded Goldreich, and Moni Naor. Adaptively secure multi-party computation. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 639–648, New York, NY, USA, 1996. ACM. 18

[52] Mustafa Canim, Murat Kantarcioglu, Bijit Hore, and Sharad Mehrotra. Building disclosure risk aware query optimizers for relational databases. *Proc. VLDB Endow.*, 3(1-2):13–24, September 2010. 50

[53] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985. 111

[54] Maeve P. Carey. Counting regulations: An overview of rulemaking, types of federal regulations, and pages in the federal register. http://www.fas.org/sgp/crs/misc/R43056.pdf, May 2013. 7

[55] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, 2005. 172

[56] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 245–252, Boston, MA, June 2001. 51

[57] J. G. Cederquist, R. Corin, M. A. C. Dekker, S. Etalle, J. I. den Hartog, and G. Lenzini. Audit-based compliance control. *Int. J. Inf. Secur.*, 6(2):133–151, March 2007. 9

[58] Y. Chang and M. Mitzenmacher. Privacy preserving keyword searches on remote encrypted data. Cryptology ePrint Archive, Report 2004/051, 2004. http://eprint.iacr.org/. 51

[59] Shuchi Chawla, Cynthia Dwork, Frank McSherry, Adam Smith, and Hoeteck Wee. Toward privacy in public databases. In *Proceedings of the Second international conference on Theory of Cryptography*, TCC'05, pages 363–385, Berlin, Heidelberg, 2005. Springer-Verlag. 8

[60] Shuchi Chawla, Cynthia Dwork, Frank Mcsherry, and Kunal Talwar. On the utility of privacy-preserving histograms. In *In 21st Conference on Uncertainty in Artificial Intelligence (UAI*. AUAI Press, 2005. 8

[61] Yao Chen and Radu Sion. On the (Im)Practicality of Securing Untrusted Computing Clouds with Cryptography. Online at http://www.cs.sunysb.edu/~sion/research/. 21, 22, 23, 27

[62] Yao Chen and Radu Sion. To Cloud or Not To. Online at http://www.cs.sunysb.edu/~sion/research/. 21, 22, 23

[63] Yao Chen and Radu Sion. On securing untrusted clouds with cryptography. In *WPES '10: Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, pages 109–114, New York, NY, USA, 2010. ACM. 21, 22, 23

[64] Yao Chen and Radu Sion. On securing untrusted clouds with cryptography. In Ehab Al-Shaer and Keith B. Frikken, editors, *WPES*, pages 109–114. ACM, 2010. 55

[65] Yao Chen and Radu Sion. To cloud or not to cloud?: musings on costs and viability. In *Proceedings of SOCC*, pages 29:1–29:7. ACM, 2011. 58, 59, 75, 77, 85

[66] Weiwei Cheng, Hweehwa Pang, and Kian lee Tan. Authenticating multi-dimensional query results in data publishing. In *In DBSec*, 2006. 198

[67] Francis Y. Chin. Security in statistical databases for queries with small counts. *ACM Trans. Database Syst.*, 3(1):92–104, March 1978. 206

[68] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 22–22, Berkeley, CA, USA, 2005. USENIX Association. 10, 172, 205

[69] Valentina Ciriani, Sabrina De Capitani Di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Combining fragmentation and encryption to protect privacy in data storage. *ACM Trans. Inf. Syst. Secur.*, 13(3):22:1–22:33, July 2010. 46

[70] Valentina Ciriani, Sabrina De Capitani Di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Combining fragmentation and encryption to protect privacy in data storage. *ACM Trans. Inf. Syst. Secur.*, 13(3):22:1–22:33, July 2010. 50, 194

[71] CNN. Feds seek Google records in porn probe. Online at http://www.cnn.com, January 2006. 16

[72] CNN. YouTube ordered to reveal its viewers. Online at http://www.cnn.com, July 2008. 16

[73] Christian Colombo and Gordon J. Pace. A compensating transaction example in twelve notations. Technical Report CS2011-01, Department of Computer Science, University of Malta, 2011. Available from http://www.um.edu.mt/ict/cs/research/technical_reports. 206

[74] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. 130, 137

[75] Federal Trade Commission. The Children's Online Privacy Protection Act. http://www.coppa.org/comply.htm, 1998. 3, 5

[76] Government Commission. German Corporate Governance Code. http://www.corporate-governance-code.de/index-e.html, 2002. 4, 5

[77] United States Congress. Health Insurance Portability and Accountability Act. http://www.hhs.gov/ocr/privacy/index.html, 1996. 3, 9, 172

[78] Security Standards Council. Pci data security standard, version 2.0, Feb 2013. 7

[79] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. *SIGARCH Comput. Archit. News*, 35(2):482–493, June 2007. 207

[80] Fida Kamal Dankar and Khaled El Emam. The application of differential privacy to health data. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 158–166, New York, NY, USA, 2012. ACM. 8

[81] Tom Denis. *Cryptography for Developers*. Syngress. 27

[82] Dorothy E. Denning and Jan Schlörer. A fast procedure for finding a tracker in a statistical database. *ACM Trans. Database Syst.*, 5(1):88–102, March 1980. 206

[83] Departments and Part 240. Agencies of the Federal Government. Code of Federal Regulations. http://www.law.cornell.edu/cfr/text/17/240.17a-4. 3, 104, 107, 126, 165

[84] Departments and Agencies of the Federal Government. Code of Federal Regulations, Part 1226 – Implementing Disposition. http://www.law.cornell.edu/cfr/text/36/1226.24. 6

[85] Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart G. Stubblebine. Authentic data publication over the internet. pages 291 – 314, 2003. 10, 80, 198, 201

[86] Sarah M. Diesburg and An-I Andy Wang. A survey of confidential data storage and deletion methods. *ACM Comput. Surv.*, 43(1):2:1–2:37, December 2010. 10, 104

[87] Cynthia Dwork. Differential privacy. In *Proceedings of the 33rd international conference on Automata, Languages and Programming - Volume Part II*, ICALP'06, pages 1–12, Berlin, Heidelberg, 2006. Springer-Verlag. 8

[88] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be. Technical report, 2008. 86

[89] Damiani E., Vimercati C., Jajodia S., Paraboschi S., and Samarati P. Balancing confidentiality and efficiency in untrusted relational dbmss. In *Proceedings of ACM CCS*, 2003. 48, 51, 194, 195

[90] Hon Janet Ecker. Keeping the Promise for a Strong Economy Act. http://www.e-laws.gov.on.ca/html/source/statutes/english/2002/elaws_src_s02022_e.htm, 2003. 5

[91] Einar Mykletun and Gene Tsudik. Aggregation Queries in the Database-As-a-Service Model. *Data and Applications Security*, 4127, 2006. 50, 62, 194, 195

[92] Elaine Shi, John Bethencourt, T-H. Hubert Chan, Dawn Song and Adrian Perrig. Multi-Dimensional Range Query over Encrypted Data. *IEEE Symposium on Security and Privacy*, 2007. 9, 17

[93] Hugh Everett. The theory of the universal wavefunction, 1956. PhD Thesis. 133

[94] Kevvie Fowler. *SQL Server Forensic Analysis*. Addison-Wesley Professional, 2008. 172

[95] Stony Brook University FSL Lab. Filebench, http://www.fsl.cs.sunysb.edu/project-fsbench.html, 2011. Benchmarking tool. 155

[96] Benjamin C. M. Fung, Ke Wang, Rui Chen, and Philip S. Yu. Privacy-preserving data publishing: A survey of recent developments. *ACM Comput. Surv.*, 42(4):14:1–14:53, June 2010. 8

[97] D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69(1):9–15, 1962. 142, 143, 152, 203

[98] Vignesh Ganapathy, Dilys Thomas, Tomas Feder, Hector Garcia-Molina, and Rajeev Motwani. Distributing data for secure database services. In *Proceedings of PAIS*, pages 8:1–8:10, New York, NY, USA, 2011. ACM. 50, 194

[99] Simson L. Garfinkel and Abhi Shelat. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security and Privacy*, 1(1):17–27, January 2003. 172

[100] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: theory, implementation and applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 151–162, New York, NY, USA, 2011. ACM. 9

[101] Tingjian Ge and Stan Zdonik. Fast, secure encryption for indexing in a column-oriented dbms. In *ICDE*, 2007. 50, 196

[102] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: outsourcing computation to untrusted workers. In *Proceedings of the 30th annual conference on Advances in cryptology*, CRYPTO'10, pages 465–482, Berlin, Heidelberg, 2010. Springer-Verlag. 9

[103] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010. 16

[104] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Annual ACM Symposium on Theory of Computing*, volume 3193 of *Lecture Notes in Computer Science*, pages 169–178. ACM, 2009. 8, 16

[105] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, March 2010. 8, 16

[106] Craig Gentry, Shai Halevi, and Nigel P. Smart. Better bootstrapping in fully homomorphic encryption. In *Proceedings of the 15th international conference on Practice and Theory in Public Key Cryptography*, PKC'12, pages 1–16, Berlin, Heidelberg, 2012. Springer-Verlag. 8, 16

[107] E. Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003. http://eprint.iacr.org/2003/216/. 51, 195

[108] Bok-Min Goi, M. U. Siddiqi, and Hean-Teik Chuah. Computational complexity and implementation aspects of the incremental hash function. *IEEE Trans. on Consum. Electron.'03*. 92

[109] Bok-Min Goi, M. U. Siddiqi, and Hean-Teik Chuah. Incremental hash function based on pair chaining & modular arithmetic combining. INDOCRYPT, pages 50–61, 2001. 92

[110] O. Goldreich. *Foundations of Cryptography I*. Cambridge University Press, 2001. 27, 35

[111] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, STOC '08, pages 113–122, New York, NY, USA, 2008. ACM. 9

[112] P. Golle, J. Staddon, and B. Waters. Secure conjunctive keyword search over encrypted data. In *Proceedings of ACNS*, pages 31–45. Springer-Verlag; Lecture Notes in Computer Science 3089, 2004. 52

[113] Daniel Golovin. *Uniquely represented data structures with applications to privacy*. PhD thesis, Pittsburgh, PA, USA, 2008. AAI3340637. 105, 106, 107, 109, 120, 124, 125, 126, 132, 135, 154, 163

[114] Daniel Golovin. B-treaps: A uniquely represented alternative to b-trees. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I*, ICALP '09, pages 487–499, Berlin, Heidelberg, 2009. Springer-Verlag. 137, 173, 189, 202

[115] Daniel Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *CoRR*, abs/1005.0662, 2010. 173, 189, 202

[116] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Practical oblivious storage. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 13–24, New York, NY, USA, 2012. ACM. 9

[117] Michael T. Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DISCEX*, pages 68 – 82. IEEE Computer Society Press, 2001. 198, 201

[118] Michael T. Goodrich, Roberto Tamassia, and Nikos Triandopoulos. Super-efficient verification of dynamic outsourced databases. CT-RSA'08, pages 407–424, 2008. 86, 198

[119] Enterprise Storage Group. ESG compliance report: The effect on information management and the storage industry. 2

[120] P. C. Gutmann. Secure filesystem (SFS) for DOS/Windows. www.cs.auckland.ac.nz/~pgut001/sfs/index.html, 1994. 51

[121] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*, SSYM'96, pages 8–8, Berkeley, CA, USA, 1996. USENIX Association. 10, 172, 205

[122] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 216–227, New York, NY, USA, 2002. ACM. 9, 17

[123] Hakan Hacigumus, Bala Iyer, and Sharad Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *9th International Conference on Database Systems for Advanced Applications, DASFAA*, volume 2973, pages 633–650. Springer Berlin Heidelberg, 2004. 9, 17, 24, 50, 51, 62, 194, 195

[124] Hacigumus H, Iyer, B and Mehrotra S. Providing Database as a Service. In *Proceedings of ICDE*, 2002. 197

[125] Hakan Hacigumus, Bala Iyer, Chen Li and Sharad Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of SIGMOD*, pages 216–227, 2002. 48, 50, 51, 193, 195

[126] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. *ACM Trans. Comput. Syst.*, 30(3):10:1–10:39, August 2012. 150

[127] Jason Hartline, Edwin Hong, Alexander Mohr, Er E. Mohr, William Pentney, and Emily Rocke. Characterizing history independent data structures, 2002. 116, 119, 124, 126, 129

[128] Jason D. Hartline, Edwin S. Hong, Alexander E. Mohr, William R. Pentney, and Emily Rocke. Characterizing history independent data structures. In *Proceedings of the 13th International Symposium on Algorithms and Computation*, ISAAC '02, pages 229–240, London, UK, UK, 2002. Springer-Verlag. 172

[129] Ragib Hasan, Radu Sion, and Marianne Winslett. Introducing secure provenance: problems and challenges. In *Proceedings of the 2007 ACM workshop on Storage security and survivability*, StorageSS '07, pages 13–18, New York, NY, USA, 2007. ACM. 5, 9

[130] Ragib Hasan, Radu Sion, and Marianne Winslett. The case of the fake picasso: preventing history forgery with secure provenance. In *Proccedings of the 7th conference on File and storage technologies*, FAST '09, pages 1–14, Berkeley, CA, USA, 2009. USENIX Association. 9

[131] Ragib Hasan, Radu Sion, and Marianne Winslett. Preventing history forgery with secure provenance. *Trans. Storage*, 5(4):12:1–12:43, December 2009. 5, 9

[132] Ragib Hasan and Marianne Winslett. Trustworthy vacuuming and litigation holds in long-term high-integrity records retention. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 621–632, New York, NY, USA, 2010. ACM. 10

[133] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994. 51

[134] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proceedings of ACM SIGMOD*, 2004. 48

[135] Justin Hsu, Aaron Roth, and Jonathan Ullman. Differential privacy for the analyst via private equilibrium computation. In *Proceedings of the 45th annual ACM symposium on Symposium on theory of computing*, STOC '13, pages 341–350, New York, NY, USA, 2013. ACM. 8

[136] Haibo Hu, Qian Chen, and Jianliang Xu. Verdict: Privacy-preserving authentication of range queries in location-based services. In *ICDE*, pages 1312–1315. 86

[137] Ling Hu, Wei-Shinn Ku, Spiridon Bakiras, and Cyrus Shahabi. Verifying spatial queries using voronoi neighbors. In *Proceedings of GIS*, pages 350–359. ACM, 2010. 198

[138] Legal Information Institute. Right of privacy: Access to personal information. http://www.law.cornell.edu/wex/personal_information. 2

[139] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2008. 24

[140] KPMG International. The cost of compliance, global hedge fund survey, 2013. 7

[141] Rohit Jain and Sunil Prabhakar. Trustworthy data from untrusted databases. *2014 IEEE 30th International Conference on Data Engineering*, 0:529–540, 2013. 83, 86, 101, 198

[142] Sen. Jim Jeffords. Patient Safety and Quality Improvement Act. http://www.hhs.gov/ocr/privacy/index.html, 2005. 3

[143] Jetico, Inc. BestCrypt software home page. www.jetico.com, 2002. 51

[144] Wei Jiang and Chris Clifton. Privacy-preserving distributed k-anonymity. In *Proceedings of the 19th annual IFIP WG 11.3 working conference on Data and Applications Security*, DBSec'05, pages 166–177, Berlin, Heidelberg, 2005. Springer-Verlag. 8

[145] Nikolai Joukov, Harry Papaxenopoulos, and Erez Zadok. Secure deletion myths, issues, and solutions. In *Proceedings of the second ACM workshop on Storage security and survivability*, StorageSS '06, pages 61–66, New York, NY, USA, 2006. ACM. 205

[146] J. Katz and Y. Lindell. *Introduction to modern cryptography*. Chapman & Hall/CRC cryptography and network security. Chapman & Hall/CRC, 2008. 176

[147] Kenneth Goldman and Enriquillo Valde. Matchbox: Secure Data Sharing. *Internet Computing*, 8(6), 2004. 197

[148] Peter Kieseberg, Sebastian Schrittwieser, Martin Mulazzani, Markus Huber, and Edgar Weippl. Trees cannot lie: Using data structures for forensics purposes. In *Proceedings of the 2011 European Intelligence and Security Informatics Conference*, EISIC '11, pages 282–285, Washington, DC, USA, 2011. IEEE Computer Society. 9

[149] Peter Kieseberg, Sebastian Schrittwieser, Martin Mulazzani, Markus Huber, and Edgar Weippl. Trees cannot lie: Using data structures for forensics purposes. In *Proceedings of the 2011 European Intelligence and Security Informatics Conference*, EISIC '11, pages 282–285, Washington, DC, USA, 2011. IEEE Computer Society. 189

[150] M. Kifer, A.J. Bernstein, and P.M. Lewis. *Database Systems: An Application-oriented Approach*. Number v. 2. Pearson/Addison-Wesley, 2006. 86, 97

[151] Marek Klonowski, MichałPrzykucki, and Tomasz Strumiński. Information security applications. chapter Data Deletion with Provable Security, pages 240–255. Springer-Verlag, Berlin, Heidelberg, 2009. 10, 172

[152] Henry F. Korth, Eliezer Levy, and Avi Silberschatz. A formal approach to recovery by compensating transactions. Technical report, Austin, TX, USA, 1990. 206

[153] Jaeheung Lee, Junyoung Heo, Yookun Cho, Jiman Hong, and Sung Y. Shin. Secure deletion for nand flash file system. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1710–1714, New York, NY, USA, 2008. ACM. 10, 172, 205, 206

[154] Philip Lewis, Arthur Bernstein, and Michael Kifer. *Databases and Transaction Processing.* Addison-wesley, 2002. 36, 39, 42

[155] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of SIGMOD*, pages 121 – 132. ACM, 2006. 58, 59, 80, 81, 82, 85, 86, 87, 197, 198, 199, 202

[156] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Authenticated index structures for aggregation queries. *ACM Trans. Inf. Syst. Secur.*, 13(4):32:1–32:35, December 2010. 198, 200

[157] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *OSDI*, 2004. 80

[158] Ninghui Li and Tiancheng Li. t-closeness: Privacy beyond k-anonymity and -diversity. In *In Proc. of IEEE 23rd Intl Conf. on Data Engineering (ICDE07*, 2007. 8

[159] Tiancheng Li, Xiaonan Ma, and Ninghui Li. Worm-seal: trustworthy data retention and verification for regulatory compliance. In *Proceedings of the 14th European conference on Research in computer security*, ESORICS'09, pages 472–488, Berlin, Heidelberg, 2009. Springer-Verlag. 10

[160] Xin Lin, Jianliang Xu, and Haibo Hu. Authentication of location-based skyline queries. In *Proceedings of CIKM*, pages 1583–1588. ACM, 2011. 86, 198

[161] Ponemon Institute LLC. Compliance cost assiciated with the storage of unstructured information, May 2011. 2

[162] Ponemon Institute LLC. The true cost of compliance: A benchmark study of multinatinal organizations, January 2011. 2, 7

[163] Ponemon Institute LLC. 2014 cost of data breach study: Global analysis, May 2014. 1

[164] Rongxing Lu, Xiaodong Lin, Xiaohui Liang, and Xuemin (Sherman) Shen. Secure provenance: the essential of bread and butter of data forensics in cloud computing. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 282–292, New York, NY, USA, 2010. ACM. 9

[165] Luc Bouganim and Philippe Pucheral. Chip-secured data access: confidential data on untrusted server. In *Proceedings of VLDB*, pages 131–141. VLDB Endowment, 2002. 52, 196

[166] Canim M., Kantarcioglu M., and Malin B. Secure management of biomedical data with cryptographic hardware. In *IEEE Transactions on Information Technology in Biomedicine*, volume 16, pages 166–175, 2012. 196

[167] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007. 8

[168] Murali Mani. Enabling secure query processing in the cloud using fully homomorphic encryption. In *Proceedings of the Second Workshop on Data Analytics in the Cloud*, DanaC '13, pages 36–40, New York, NY, USA, 2013. ACM. 9, 17

[169] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart Stubblebine. A general model for authenticated data structures. *Algorithmica, Volume 39 Issue 1*, pages 21 – 41, 2004. 201

[170] A. D. McDonald and M. G. Kuhn. StegFS: A Steganographic File System for Linux. In *Information Hiding*, pages 462–477, 1999. 51

[171] Dinesh P. Mehta and Sartaj Sahni. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.)*. Chapman & Hall/CRC, 2004. 105, 110, 166, 203

[172] R. Merkle. A Certified Digital Signature. In *Proceedings of Crypto*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer-Verlag, 1990. 58, 81

[173] Daniele Micciancio. An oblivious data structure and its applications to cryptography. In *In Proceedings of ACM Symposium on the Theory of Computing*, pages 456–464. ACM Press, 1997. 10, 202, 203

[174] Daniele Micciancio. Oblivious data structures: applications to cryptography. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 456–464, New York, NY, USA, 1997. ACM. 173

[175] Microsoft Research. Encrypting File System for Windows 2000. Technical report, Microsoft Corporation, July 1999. www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp. 51

[176] Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 650–661, New York, NY, USA, 2012. ACM. 8

[177] Soumyadeb Mitra, Windsor W. Hsu, and Marianne Winslett. Trustworthy keyword search for regulatory-compliant records retention. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1001–1012. VLDB Endowment, 2006. 10

[178] David Molnar, Tadayoshi Kohno, Naveen Sastry, and David Wagner. Tamper-evident, history-independent, subliminal-free data structures on prom storage-or-how to store ballots on a voting machine (extended abstract). In *Proceedings of IEEE Symposium on Security and Privacy*, SP '06, pages 365–370. IEEE Computer Society, 2006. 104, 202, 204

[179] Tal Moran, Moni Naor, and Gil Segev. Deterministic history-independent strategies for storing information on write-once memories. In *Proceedings of International Colloquium on Automata, Languages and Programming*, pages 303–315. Springer, 2007. 104, 202, 204

[180] Ira S. Moskowitz and Liwu Chang. A decision theoretical based system for information downgrading. In *In Proceedings of the 5th Joint Conference on Information Sciences*, 2000. 8

[181] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Signature bouquets: Immutability for aggregated/condensed signatures. In *ESORICS*, 2004. 59, 198, 200, 201

[182] Einar Mykletun, Maithili Narasimha, and Gene Tsudik. Authentication and integrity in outsourced databases. *ACM TOS, Volume 2 Issue 2*, pages 107 – 138, 2006. 10, 59, 80, 198, 200, 201

[183] Moni Naor, Gil Segev, and Udi Wieder. History-independent cuckoo hashing. In *Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 631–642, Berlin, Heidelberg, 2008. Springer-Verlag. 104, 106, 116, 124, 135, 173, 202, 203

[184] Moni Naor and Vanessa Teague. Anti-persistence: History independent data structures. In *In Proceedings of ACM symposium on Theory of computing*, pages 492–501. ACM Press, 2001. 106, 119, 124, 126, 154, 202, 203, 204, 205

[185] Maithili Narasimha and Gene Tsudik. Dsac: integrity for outsourced databases with signature aggregation and chaining. In *Proceedings of CIKM*. ACM, 2005. 58, 59, 61, 70, 71, 80, 83, 198, 201, 202

[186] Suman Nath, Haifeng Yu, and Haowen Chan. Secure outsourced aggregation via one-way chains. SIGMOD '09, pages 31–44, New York, NY, USA, 2009. ACM. 198

[187] Mehmet Ercan Nergiz, Maurizio Atzori, and Chris Clifton. Hiding the presence of individuals from shared databases. In *Proceedings of the 2007 ACM SIGMOD*

international conference on Management of data, SIGMOD '07, pages 665–676, New York, NY, USA, 2007. ACM. 8

[188] Mehmet Ercan Nergiz, Christopher Clifton, and Ahmet Erhan Nergiz. Multirelational k-anonymity. *IEEE Trans. on Knowl. and Data Eng.*, 21(8):1104–1117, August 2009. 8

[189] Nicolas Anciaux, Mehdi Benzine, Luc Bouganim, Philippe Pucheral and Dennis Shasha. GhostDB: Querying Visible and Hidden Data Without Leaks. In *Proceedings of SIGMOD*, 2007. 52, 196

[190] Commonwealth of Australia. Corporations Act. http://www.austlii.edu.au/au/legis/cth/consol_act/ca2001172/sch3.html, 2001. 5

[191] Parliament of Canada. Personal Information Protection and Electronic Documents Act. http://laws-lois.justice.gc.ca/eng/acts/P-8.6/, 2000. 4, 104, 107, 126, 165

[192] Department of Defense. DoD Records Management Program. http://www.defense.gov/webmasters/policy/dodd50152p.pdf, 2000. 1, 3

[193] Government of India. Information Technology Amendment Act. http://pib.nic.in/newsite/erelease.aspx?relid=53617, 2000. 4

[194] North Carolina Department of Justice. Protect your identity. http://www.ncdoj.gov/Protect-Yourself/2-4-3-Protect-Your-Identity.aspx. 1

[195] Office of Management and Budget. On the benefits and costs of federal regulations and agency compliance with the unfunded mandates reform act. http://www.whitehouse.gov/omb/inforeg_regpol_reports_congress, 2013. 7

[196] Parliament of Singapore. Personal Data Protection Act. http://www.pdpc.gov.sg/personal-data-protection-act/the-act, 2012. 4

[197] Parliament of the United Kingdom. Regulation of Investigatory Powers Act. http://www.legislation.gov.uk/ukpga/2000/23/contents, 2000. 4

[198] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of EuroCrypt*, 1999. 11, 24, 51, 195

[199] Pascal Paillier. A trapdoor permutation equivalent to factoring. In *Proceedings of PKC*, pages 219–222. Springer-Verlag, 1999. 11, 24, 51, 195

[200] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and KianLee Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of SIGMOD*, pages 407 – 418. ACM, 2005. 83, 198, 199, 201

[201] HweeHwa Pang and Kian-Lee Tan. Authenticating query results in edge computing. In *Proceedings of ICDE*, page 560. IEEE Computer Society, 2004. 58, 86, 197, 198, 199, 202

[202] HweeHwa Pang and Kian-Lee Tan. *Query Answer Authentication*. Synthesis Lectures on Data Management. 2012. 81, 86

[203] HweeHwa Pang, Jilian Zhang, and Kyriakos Mouratidi. Scalable verification for outsourced dynamic databases. *Proceedings of the VLDB Endowment, Volume 2 Issue 1*, pages 802 – 813, 2009. 10, 59, 70, 71, 73, 80, 82, 84, 85, 86, 198

[204] Stavros Papadopoulos, Dimitris Papadias, Weiwei Cheng, and Kian-Lee Tan. Separating authentication from query execution in outsourced databases. In *Proceedings of ICDE*, pages 1148 – 1151. IEEE Computer Society, 2009. 10, 80, 81, 83, 86, 198

[205] European Parliament and of the Council. The EU E-Privacy Directive (2002/58/EC). http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32002L0058:en:NOT, 2002. 6

[206] Kyriacos E. Pavlou and Richard T. Snodgrass. Forensic analysis of database tampering. *ACM Trans. Database Syst.*, 33(4):30:1–30:47, December 2008. 9

[207] Senator Peace and Assembly Member Simitian. California Security Breach Information Act, 2002. 3

[208] Jian Pei, Man Ki Mag Lau, and Philip S. Yu. Ts-trees: A non-alterable search tree index for trustworthy databases on write-once-read-many (worm) storage. In *Proceedings of the 21st International Conference on Advanced Networking and Applications*, AINA '07, pages 54–61, Washington, DC, USA, 2007. IEEE Computer Society. 10

[209] Zachary N. J. Peterson, Randal Burns, Joe Herring, Adam Stubblefield, and Aviel D. Rubin. Secure deletion for a versioning file system. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association. 10, 206

[210] Raphael C. W. Phan and David Wagner. Security considerations for incremental hash functions based on pair block chaining. *Comput. Secur. '06, 131-136*. 92

[211] Raluca Ada Popa, Catherine Redfield, and Nickolai Zeldovich. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of SOSP*, 2011. 50, 195

[212] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing.

In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM. 9, 17

[213] William Pugh. Skip lists: A probabilistic alternative to balanced trees, 1990. 201

[214] M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical report, 1979. 16, 21

[215] A.I.M. Rae. *Quantum physics: a beginner's guide*. Oneworld beginners' guides. Oneworld, 2005. 133

[216] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, Yaping Li. Sovereign Joins. In *Proceedings of the 22nd International Conference on Data Engineering*, page 26. IEEE Computer Society, 2006. 52, 196

[217] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant and Yirong Xu. Order Preserving Encryption for Numeric Data. In *Proceedings of the ACM SIGMOD international conference on Management of data*, pages 563–574. ACM, 2004. 195

[218] Vibhor Rastogi, Dan Suciu, and Sungho Hong. The boundary between privacy and utility in data publishing. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 531–542. VLDB Endowment, 2007. 8

[219] Joel Reardon, David Basin, and Srdjan Capkun. Sok: Secure data deletion. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 301–315. IEEE Computer Society, 2013. 153

[220] Federal regulations. DODD-FRANK WALL STREET REFORM AND CONSUMER PROTECTION ACT. http://www.gpo.gov/fdsys/pkg/PLAW-111publ203/pdf/PLAW-111publ203.pdf, 2010. 7

[221] R. F. Resende and A. El Abbadi. On the serializability theorem for nested transactions. *Inf. Process. Lett.*, 50(4):177–183, May 1994. 186

[222] Richard T. Snodgrass and Stanley Yao and Christian Collberg. Tamper Detection in Audit Logs. In *Proceedings of VLDB*, 2004. 9

[223] Ronald Rivest, Len Adleman and Michael Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 1978. 8, 11, 16, 24, 50, 194

[224] James M. Rosenbaum. In defence of the delete key. *The Green Bag*, 3(4), 2000. 172

[225] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. *SIGPLAN Not.*, 44(6):63–74, June 2009. 207

[226] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM SYSTEMS JOURNAL*, 40(3), 2001. 52, 196

[227] P. Samarati. Protecting respondents' identities in microdata release. *IEEE Trans. on Knowl. and Data Eng.*, 13(6):1010–1027, November 2001. 46

[228] Pierangela Samarati and Latanya Sweeney. Generalizing data to provide anonymity when disclosing information (abstract). In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 188–, New York, NY, USA, 1998. ACM. 8

[229] Pierangela Samarati and Latanya Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. Technical report, 1998. 8

[230] Sumit Sanghrajka, Nilesh Mahajan, and Radu Sion. Cloud performance benchmark series, network performance: Amazon ec2. Online at www.cloudcommons.org. 75

[231] Sanjay Agrawal and Vivek Narasayya and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of SIGMOD*, pages 359 – 370. ACM, 2004. 37

[232] U.S. Senator Paul Sarbanes and U.S. Representative Michael G. Oxley. Sarbanes-Oxley Act. http://www.sec.gov/about/laws.shtml#sox2002, 2002. 3, 6, 9

[233] John Savage. Models of computation: Exploring the power of computing, 1998. 108, 109

[234] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2):159–176, May 1999. 9

[235] Sean W. Smith. Outbound authentication for programmable secure coprocessors. Online at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.4066. 19

[236] Securities and Exchange Commission. Electronic Storage of Broker-Dealer Records, 17 CFR Part 241. http://www.sec.gov/rules/interp/34-47806.htm, 2002. 5

[237] Securities and Exchange Board of India. Listing Agreement to the Indian stock exchange. http://www.nseindia.com/getting_listed/content/clause_49.pdf, 2005. 4

[238] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. In *Proceedings of VLDB*, pages 481 – 492. Morgan Kaufmann Publishers Inc., 1990. 37

[239] Jitesh Shetty and Jafar Adibi. The enron email dataset database schema and brief statistical report, 2004. 172

[240] Sarvjeet Singh and Sunil Prabhakar. Ensuring correctness over untrusted private database. In *Proceedings of EDBT*, pages 476 – 486. ACM, 2008. 10, 80, 198, 199, 201

[241] D. Xiaodong Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P 2000)*. IEEE Computer Society, 2000. 51, 195

[242] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 91–102, New York, NY, USA, 2007. ACM. 172, 188, 189

[243] Washington state office of the Attorney General. Identity theft. http://www.atg.wa.gov/InternetSafety/IDTheft.aspx#.UfnmbZHYU8o. 1

[244] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. *CoRR*, abs/1106.3652, 2011. 9

[245] Latanya Sweeney. Protecting job seekers from identity theft. *IEEE Internet Computing*, 10(2):74–78, March 2006. 172

[246] M. Szeredi. Filesystem in Userspace. http://fuse.sourceforge.net, February 2005. 155, 167

[247] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. File size distribution on unix systems: then and now. *SIGOPS Oper. Syst. Rev.*, 40(1):100–104, January 2006. 156

[248] Maolin Tang and Colin Fidge. Reconstruction of falsified computer logs for digital forensics investigations. In *Proceedings of the Eighth Australasian Conference on Information Security - Volume 105*, AISC '10, pages 12–21, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc. 9

[249] Tingjian Ge and Stan Zdonik. Answering Aggregation Queries in a Secure System Model. In *Proceedings of VLDB*, pages 519–530. VLDB Endowment, 2007. 24, 26, 51, 195

[250] Theodoros Tzouramanis. History-independence: a fresh look at the case of r-trees. In *Proceedings of ACM Symposium on Applied Computing*, SAC '12, pages 7–12. ACM, 2012. 202

[251] European Union. EU Data Protection Directive. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:31995L0046:en:HTML, 1998. 4, 6

[252] European Union. EU Data Retention Directive. http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2006:105:0054:0063:EN:PDF, 2006. 2, 4, 5, 6, 104, 107, 126, 165

[253] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'10, pages 24–43, Berlin, Heidelberg, 2010. Springer-Verlag. 8, 16

[254] V. S. Verykios, A. K. Elmagarmid, E. Bertino, Y. Saygn, and E. Dasseni. Association rule hiding. *IEEE Transactions on Knowledge and Data Engineering*, 16:434–447, 2004. 8

[255] Michael Walsh and Andrew J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. In *Electronic Colloquium on Computational Complexity, Revision 2 of Report No. 165*, 2013. 10

[256] Hui Wang and Laks V.S. Lakshmanan. Efficient secure query evaluation over encrypted xml databases. In *VLDB*, 2006. 49, 51, 194, 195

[257] Ke Wang and Benjamin C. M. Fung. Anonymizing sequential releases. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 414–423, New York, NY, USA, 2006. ACM. 8

[258] Ke Wang, Benjamin C. M. Fung, and Guozhu Dong. Integrating private databases for data analysis. In *Proceedings of the 2005 IEEE international conference on Intelligence and Security Informatics*, ISI'05, pages 171–182, Berlin, Heidelberg, 2005. Springer-Verlag. 8

[259] Ke Wang, Benjamin C. M. Fung, and Philip S. Yu. Handicapping attacker's confidence: an alternative to k-anonymization. *Knowl. Inf. Syst.*, 11(3):345–368, April 2007. 8

[260] Shiyuan Wang, Divyakant Agrawal, and Amr El Abbadi. A comprehensive framework for secure query processing on relational data in the cloud. In *Secure Data Management*, pages 52–69, 2011. 50, 51, 195

[261] Peter Williams and Radu Sion. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 293–304, New York, NY, USA, 2012. ACM. 9

[262] Peter Williams, Radu Sion, and Miroslava Sotakova. Practical oblivious outsourced storage. *ACM Trans. Inf. Syst. Secur.*, 14(2):20:1–20:28, September 2011. 9

[263] Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: a parallel oblivious file system. In *Proceedings of the 2012 ACM conference on Computer and*

communications security, CCS '12, pages 977–988, New York, NY, USA, 2012. ACM. 9

[264] Raymond Chi wing Wong, Jiuyong Li, Ada Wai chee Fu, and Ke Wang. , k)-anonymity: an enhanced k-anonymity model for privacy preserving data publishing. In *In ACM SIGKDD*, pages 754–759, 2006. 8

[265] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association. 51

[266] Craig Wright, Dave Kleiman, and Shyaam Sundhar R.S. Overwriting hard drive data: The great wiping controversy. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 243–257, Berlin, Heidelberg, 2008. Springer-Verlag. 172, 205

[267] Xiaokui Xiao and Yufei Tao. Personalized privacy preservation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 229–240, New York, NY, USA, 2006. ACM. 8

[268] Min Xie, Haixun Wang, Jian Yin, and Xiaofeng Meng. Integrity auditing of outsourced data. In *Proceedings of VLDB*, pages 782 – 793. VLDB Endowment, 2007. 198, 201

[269] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of SIGMOD*, pages 5 – 18. ACM, 2009. 59, 70, 71, 73, 74, 75, 81, 86, 198, 199

[270] Yin Yang, Stavros Papadopoulos, Dimitris Papadias, and George Kollios. Authenticated indexing for outsourced spatial databases. pages 631 – 648, 2009. 198

[271] Ziwei Yang, Shen Gao, Jianliang Xu, and Byron Choi. Authentication of range query results in mapreduce environments. In *Proceedings of CloudDB*, pages 25–32. ACM, 2011. 198

[272] Gahi Youssef, Guennoun Mouhcine, and El-Khatib Khalil. A secure database system using homomorphic encryption schemes. In *The Third International Conference on Advances in Databases, Knowledge, and Data Applications*, DBKDA '11, 2011. 9

[273] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. *Commun. ACM*, 54(11):93–101, November 2011. 207

[274] Qing Zhang, Nick Koudas, Divesh Srivastava, and Ting Yu. Aggregate query answering on anonymized tables. In *In VLDB*, 2007. 8

[275] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. Efficient query integrity for outsourced dynamic databases. 2012. 198

[276] Ying Zhou and Chen Wang. An online query authentication system for outsourced databases. In *TrustCom'13*. 80, 83, 198

[277] Qingbo Zhu and Windsor W. Hsu. Fossilized index: the linchpin of trustworthy non-alterable electronic records. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 395–406, New York, NY, USA, 2005. ACM. 10

[278] Qingbo Zhu and Windsor W. Hsu. Fossilized index: the linchpin of trustworthy non-alterable electronic records. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 395–406, New York, NY, USA, 2005. ACM. 206

[279] Youwen Zhu, Rui Xu, and Tsuyoshi Takagi. Secure k-nn computation on encrypted cloud data without sharing key with query users. In *Proceedings of the 2013 international workshop on Security in cloud computing*, Cloud Computing '13, pages 55–60, New York, NY, USA, 2013. ACM. 9