QUERY AND DATA SECURITY IN THE DATA OUTSOURCING MODEL

FEIFEI LI

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

BOSTON
UNIVERSITY

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

QUERY AND DATA SECURITY IN THE DATA OUTSOURCING MODEL

by

FEIFEI LI

B.S., Nanyang Technological University, Singapore, 2002

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2007

Approved by

First Reader

George Kollios, PhD
Associate Professor of Computer Science

Second Reader

Leonid Reyzin, PhD
Assistant Professor of Computer Science

Third Reader

Donghui Zhang, PhD
Assistant Professor of Computer Science

Acknowledgments

First of all I thank my advisor, George Kollios, for his continuous support and guidance throughout my PhD studies. I am especially grateful to him for giving me the freedom to pursue research topics that are interesting to me. My quest towards the PhD will not be successful without his constant encouragement and valuable advice at various points in last five years. I thank my thesis co-advisor, Leonid Reyzin, for his advice and discussion on subjects related to cryptography. Without his insightful feedbacks I may not have found this wonderful thesis topic.

I also thank Azer Bestavros for giving me a nice research problem in data stream processing. This experience enabled me to identify and work on security issues in data stream management systems. I thank John Byers and Shang-Hua Teng for collaborating on interesting research problems which sharpened my research skills and also led to a lot of fun moments. I am grateful to my other committee member, Donghui Zhang, for taking time out of his schedule to participate in my defense. My thank also goes to Maithili Narasimha and Gene Tsudik for pointing out a discrepancy associated with measuring the cost of signing operations in the signature chaining approach in the early draft of this dissertation.

My undergraduate advisors, Wee Keong Ng and Ee-Peng Lim, deserve special thanks for their encouragement and advice during my final year project. This research introduced me to the world database research and led me into the PhD program. My close friend, Zehua Liu, shared this wonderful period with me and together we explored the interesting world of computer science research.

I appreciate the joint work with the co-authors of my other research papers: Azer Bestavros, Ching Chang, Dihan Cheng, Jeffrey Considine, Yanfeng Huang, Ee-Peng Lim, Zehua Liu, George Mihaila, Wee Keong Ng, Dimitris Papadias, Spiros Papdimitriou, Divesh Srivastava, Ioana Stanoi, Jimeng Sun, Yufei Tao and Philip S. Yu. I have enjoyed working with all of you greatly! Although the papers that I have with them are not included in

this thesis, they are an integral part of my PhD research, and lead me to many interesting directions for future research. I would like to thank Ioana Stanoi, George Mihaila, Yuan-Chi Chang, Marios Hadjieleftheriou, Divesh Srivastava and David Lomet, who served as my mentors and gave me opportunities to work at the world's leading industrial research labs. All these valuable experience have helped me grow into a versatile scholar. In particular, these experience enabled me to conduct practical research with impact on the real world.

I want to thank my close friends from the computer science department for providing me with such an open and stimulating environment for both work and life. It is impossible to name everyone who deserves a note. However, I especially thank Jingbin Wang for being my roommate, Sa Cui, Vijay Erramilli, Nahur Fonseca, Rui Li, Panagiotis Papapetrou, Michalis Potamias, Niky Riga, Rui Shi, Georgios Smaragdakis, Yangui Tao, Taipeng Tian, Ashwin Thangali, Zheng Wu, Quan Yuan, Giorgos Zervas and Jing Zhong for leaving me uncountable joyful memories. I also want to thank the students, professors, and staff, especially, Joan Butler, Stephanie Cohen, Ellen Grady, Paul Stauffer and Jennifer Streubel, that I have interacted with within the department, for all the times they have helped and supported me.

I am grateful to my mother, father, and brother, for their unwavering support during my studies. I am also grateful to Xing Ma, who accompanied me during the most challenging part of my Phd studies.

Finally, I am especially thankful to Rui Li and Taipeng Tian for proof reading this dissertation. I want to thank Marios Hadjieleftheriou and Ke Yi, my good friends and primary research collaborators over the last two years. The core contributions of this thesis have evolved through the numerous discussions with them. This thesis, and my entire research work, would have been impossible without their valuable contribution. Thank you!

QUERY AND DATA SECURITY IN THE DATA OUTSOURCING MODEL

(Order No.)

FEIFEI LI

Boston University, Graduate School of Arts and Sciences, 2007

Major Professor: George Kollios, Department of Computer Science

ABSTRACT

An increasing number of applications adopts the model of outsourcing data to a third? party for query and storage. Under this model, database owners delegate their database management needs to third?arty data servers. These servers possess both the necessary resources, in terms of computation, communication and storage, and the required expertise to provide efficient and effective data management and query functionalities to data and information consumers.

However, as servers might be untrusted or can be compromised, query and data security issues must be addressed in order for this model to become more practical. An important challenge in this realm is to enable the client to authenticate the query results returned from a server. To authenticate is to verify that the query results are correctly computed from the same data as the data owner has published and all results have been honestly returned. Existing solutions for this problem concentrate mostly on static, relational data scenarios and are based on idealistic properties for certain cryptographic primitives, looking at the problem mostly from a theoretical perspective. This thesis proposes practical and efficient solutions that address the above challenge for both relational and streaming data. Specifically, this dissertation provides dynamic authenticated index structures for authenticating range and aggregation queries in both one and multiple dimensional spaces. The authentication of sliding window queries over data streams is then discussed to sup?

port data streaming applications. We also study the problem of query execution assurance over data streams where the data owner and the client are the same entity. A probabilistic verification algorithm is presented that has minimal storage and update costs and achieves a failure probability of at most δ, for any small $\delta > 0$. The algorithm is generalized to handle the scenarios of load shedding and multiple queries.

An extensive experimental evaluation for all the proposed methods over both synthetic and real data sets is presented. The findings of this evaluation demonstrate both the correctness and effectiveness of the proposed methods.

Contents

List of Tables

List of Figures

List of Abbreviations

1D	.....	One-dimensional
2D	.....	Two-dimensional
3D	.....	Three-dimensional
AAB Tree	.....	Authenticated Aggregation B^+ Tree
AAR Tree	.....	Authenticated Aggregation R Tree
APS Tree	.....	Authenticated Prefix Sum Tree
ASB Tree	.....	Aggregated Signature B^+ Tree
CM Sketch	.....	Count-Min Sketch
CQV	.....	Continuous Query Verification
DBMS	.....	Database Management System
DSMS	.....	Data Stream Management System
EMB Tree	.....	Embedded Merkle B^+ Tree
DMkd Tree	.....	Dyadic Merkle kd-tree
EMkd Tree	.....	Exponential Merkle kd-tree
$\mathcal{LC}$	.....	Label Cover
$\mathcal{LCA}$	.....	Least Covering Ancestor
LRU	.....	Least Recent Used
MB Tree	.....	Merkle B^+ Tree
$\mathcal{MCS}$	.....	Minimum Covering Set
MHT	.....	Merkle Hash Tree
NIST	.....	National Institute of Standards and Technology

ODB	.....	Outsourced Databases
PK	.....	Public Key
PIRS	.....	Polynomial Identity Random Summation
PS	.....	Prefix Sums
SK	.....	Secret Key
TM Tree	.....	Tumbling Merkle Tree
TMkd Tree	.....	Tumbling Merkle kd-tree
$\mathcal{VO}$	.....	Verification Object
WORM	.....	Write-Once-Read-Many

Chapter 1

Introduction

The latest decade has witnessed tremendous advances in computer systems, networking technologies and, as a consequence, the Internet, sparkling a new information age where people can access and process data remotely, accomplish critical tasks from the leisure of their own home, or do business on the go. In order to guarantee availability, reliability and good performance for a variety of network services, service providers are forced to distribute and replicate both data and services across multiple servers at the edge of the network, not necessarily placed under their direct control. Advantages of distributing and replicating data include bringing the data closer to the users and reducing network latency, increasing fault tolerance, making the services more resilient to denial of service attacks. As a concrete example, according to the Census Bureau of the U.S. Department of Commerce, Electronic Commerce in the United States generated sales of approximately 100 billion USD in the year 2006 [Bureau, 2006]. Today, there is a large number of corporations that use electronic commerce as their primary means of conducting business. As the number of customers using the Internet for acquiring services increases, the demand for providing fast, reliable and secure transactions increases accordingly. But overwhelming data management and information processing costs typically outgrow the capacity of individual businesses to provide the level of service required. This naturally requires the business owner to outsource its data to a third party for the guaranteed level of quality in providing services to its clients.

Database outsourcing [Hacigümüs et al., 2002b] is a new paradigm that has been proposed recently and received considerable attention for addressing the issues outlined above. The basic idea is that data owners delegate their database needs and functionalities to a

third-party provider. There are three main entities in the data outsourcing model: the data owner, the database service provider (a.k.a. server) and the client. In general, many instances of each entity might exist. In practice, usually there is a single or a few data owners, a few servers, and many clients. The data owner first creates the database, along with the associated index and authentication structures and uploads it to the servers. It is assumed that the data owner might update the database periodically or in an ad-hoc manner, and that the data management and retrieval happens only at the servers. Clients submit queries about the owner's data to the servers and get back results through the network. In certain scenarios, the data owner and the clients could be the same entity.

Remotely accessing data through the network, under both a centralized or a distributed setting, inherently raises important security issues, as the third party can be untrusted or can be compromised. Servers are prone to hacker attacks that can compromise the legitimacy of the data residing therein and the processing performed. The chances of even a meticulously maintained server being compromised by an adversary should not be underestimated, especially as the application environment and interactions become more complex and absolute security is harder, if not impossible, to achieve. This problem is exacerbated by the fact that time is in favor of attackers, since scrutinizing a system in order to find exploits proactively is an expensive and difficult task not commonly employed in practice. The hardness of enforcing a given level of security measures explodes in widely distributed settings, where many servers and diverse entities are involved. Furthermore, communication between the servers and the clients are vulnerable to various man-in-middle types of attacks as well.

An adversary gaining access to the data is free to deceive users who query and process the data, possibly remaining unobserved for a substantial amount of time. For non-critical applications, simple audits and inspections will eventually expose any attack. On the other hand, for business critical or life critical applications such sampling based solutions are not good enough. For example, consider a client taking business critical decisions concerning financial investments (e.g., mutual funds and stocks) by accessing information from what

he considers to be a trusted financial institution. A single illegitimate piece of information might have catastrophic results when large amounts of money are at stake. Similar threats apply to online banking applications and more. Most importantly, use of illegitimate information has dire consequences when health issues are involved. For example, planning future vaccination strategies for a subpopulation in Africa based on recent vaccination history, or deciding a course of action for a given individual based on patients with similar profiles. Notice that in these examples information can be tampered with not only due to a compromised sever, but also due to insecure communication channels and untrusted entities that already have access to the data (e.g., system administrators).

It is reasonable to assume that in critical applications users should be provided with absolute security guarantees on a per transaction basis, even at a performance cost. Such demands cannot be satisfied by putting effort and money into developing more stringent security measures and audits — in the long run subversion is always possible. Hence, a fail-safe solution needs to be engineered for tamper-proofing the data against all types of manipulation from unauthorized individuals at all levels of a given system, irrespective of where the data resides or how it was acquired, as long as it has been endorsed by the originator. Techniques for guaranteeing against illegitimate processing need to be put forth. It is exactly in this direction that this work focuses on, and more precisely, on query authentication: Assuring end-users that query execution on any server, given a set of authenticated data, returns correct results, i.e., unaltered and complete answers (no answers have been modified, no answers have been dropped, no spurious answers have been introduced). Towards that goal, the owner must give the clients the ability to authenticate the answers they receive without having to trust the servers. In that respect, query authentication means that the client must be able to validate that the returned records do exist in the owner's database and have not been modified in any way, i.e., they are the correct query results as if the data owner answers the query itself.

Existing work concentrate on offline, relational databases and study the problem mostly from a theoretical perspective. In this work, we address the query authentication prob-

lems for the relational database with an emphasis on combining cryptographic primitives with efficient disk-based indexing structures without compromising query efficiency. Additional challenges are posted by new models of databases emerging recently, such as the important streaming database [Chandrasekaran and et al., 2003, Hammad and et al., 2004, Arasu and et al., 2003, Cranor et al., 2003a]. We extend our discussions to streaming databases that has not been studied before. In addition, query execution assurance, as a special form of the query authentication problem for outsourced database (ODB) systems where the data owner and the clients are the same entity, has been studied for the offline, relational data model [Sion, 2005], however, there is no prior work addressing this issue in the context of the streaming database. This thesis closes the gap in this important area as well.

Furthermore, similar security concerns must be addressed in data publishing using content delivery networks [Saroiu et al., 2002], peer-to-peer computing [Huebsch et al., 2003] and network storage [Fu et al., 2002]. We would like to point out that there are other security issues in ODB systems that are orthogonal to the problems considered here. Examples include privacy-preservation issues [Hore et al., 2004, Agrawal and Srikant, 2000, Evfimievski et al., 2003], private query execution [Hacigümüs et al., 2002a] and security in conjunction with access control requirements [Miklau and Suciu, 2003, Rizvi et al., 2004, Pang et al., 2005].

1.1 Query Authentication for Relational Databases

The relational database is the most common and important type of model for representing, storing and querying large scale of structured data. A remarkable amount of effort has been coordinated in providing query authentication capability for relational databases[1]. However, as we will soon point out, existing work look at the problem mostly from a theoretical perspective and assume idealistic properties for certain cryptographic primitives, which makes them suitable for main-memory based, small scaled data. There is a lack of

[1]Readers are referred to Chapter 3 for a detailed review.

study for disk-based, large scale datasets, where the main challenge is how to incorporate cryptographic techniques into existing disk-based indexing structures so that query efficiencies are not dramatically compromised by enabling query authentication. That is the first focus of this thesis.

In the current framework, the owner creates a database along with the necessary query authentication structures and forwards the data to the servers. The clients issue queries, which are answered by the servers using the query authentication structures to provide provably correct results. Correctness is evaluated on a per query basis. Every answer is accompanied by a verification proof — an object that can be used to reconstruct the endorsement of the data originator, and build in a way that any tampering of the answer will invalidate an attempt to verify the endorsement. Verification objects are constructed using a form of chained hashing, using ideas borrowed from Merkle trees [Merkle, 1978] (cf. Chapter 2). A by-product of ensuring query authentication is that the data is also protected from unauthorized updates, i.e., changes made by individuals that do not have access to the secret key used to endorse the data. Data owners may update their databases periodically and, hence, authentication structures should support dynamic updates. An important dimension of query authentication in a dynamic environment is result freshness. Notice that an adversary acquiring an older version of an authenticated database can provide what may appear to be correct answers, that contain in reality outdated results. Authenticated structures should protect against this type of attack as well. It should be noted here that query authentication mechanisms do not guard against corrupt individuals at the originator side who, for example, could introduce phony updates to the data. In the rest, we assume that the originator is trusted and there are no corrupt insiders with access to the secret key. This also implies that the internal systems where data processing and endorsement is taking place have not been compromised.

Clearly, security does not come for free. Any security measure has an associated cost. There are a number of important metrics for evaluating query authentication mechanisms:
1. The computation overhead for the owner for constructing authentication structures;

2. The owner-server communication cost; 3. The storage overhead for the server; 4. The computation overhead for the server; 5. The client-server communication cost; and 6. The computation cost for the client to verify the answers.

Previous work has addressed the problem of query authentication for selection and projection queries only (e.g., "Find all stocks with bids between $50 and $500 within the last hour"), mostly for static scenarios. Existing solutions take into account only a subset of the metrics proposed here. Furthermore, previous work was mostly of theoretical nature, analyzing the performance of the proposed techniques using analytical cost models and not taking into account the fact that certain cryptographic primitives do not feature idealistic properties in practice. For example, trying to minimize the size of verification object does not take into account the fact that the time to generate a signature using popular public signature schemes is very costly and it also incurs additional I/Os due to the large size of individual signature. Our first contribution is to introduce novel disk-based authenticated indexing structures based on Merkle trees [Merkle, 1989] and B-trees [Comer, 1979] for authenticating selection and projection queries in static and dynamic settings.

Another important aspect of query authentication that has not been considered in the past is handling efficiently aggregation queries (e.g., "Retrieve the total number of stocks with bids between $50 and $500 within the last hour"). Currently available techniques for selection and projection queries can be straightforwardly applied for answering aggregation queries on a single selection attribute. Albeit, they exhibit very poor performance in practice and cannot be generalized to multiple selection attributes without incurring even higher querying cost. Our second contribution is to formally define the aggregation authentication problem and provide efficient solutions that can be deployed in practice, both for dynamic and static scenarios. Concentrating on SUM aggregates, we show that in static scenarios authenticating aggregation queries is equivalent to authenticating prefix sums [Ho et al., 1997]. For dynamic scenarios, maintaining the prefix sums and the corresponding authentication structure becomes expensive. Hence, we propose more involved techniques for efficiently handling updates, which are based on authenticated B-

trees [Comer, 1979] and R-trees [Guttman, 1984]. Finally, we extend the techniques for aggregates other than SUM, and discuss some issues related to result freshness and data encryption for privacy preservation.

1.2 Authenticating Sliding Window Query over Data Streams

Online services, like electronic commerce and stock market applications, are now permeating our modern way of life. Due to the overwhelming volume of data that can be produced by these applications, the amount of required resources, as the market grows, exceeds the capacity of a relational database that stores and queries all historical data. To deal with such challenges, data stream has been proposed as a new model of data management [Babcock et al., 2002] where emphasis is put on recent data and requiring fast and reliable query processing using limited system resources, in terms of both the memory consumption and the CPU computation. Outsourced data stream processing is even more appealing as the large amount of data generated by a streaming source could easily exceed the capacity of individual businesses to provide fast and reliable services. Consider the following example. A stock broker needs to provide a real-time stock trading monitoring service to clients. Since the cost of multicasting this information to a large audience might become prohibitive, as many clients could be monitoring a large number of individual stocks, the broker could outsource the stock feed to third-party providers, who would be in turn responsible for forwarding the appropriate sub-feed to clients (see Figure 1·1). Evidently, especially in critical applications, the integrity of the third-party should not be taken for granted, as the latter might have malicious intent or might be temporarily compromised by other malicious entities. Deceiving clients can be attempted for gaining business advantage over the original service provider, for competition purposes against important clients, or for easing the workload on the third-party servers, among other motives. In that respect, assuming that the original service provider can be trusted, an important consideration is to give clients the ability to prove the service furnished by the third-party (a.k.a. the server).

Consider an end-user that issues a monitoring query for the moving average of a specific

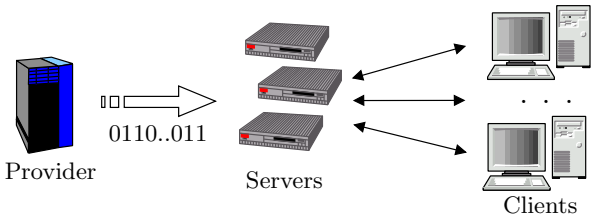


Figure 1·1: The outsourced stream model.

stock within a sliding window. This monitor can be viewed as a selection-aggregation query over the original stock feed. A third-party server is assigned to forward the aggregate over all qualifying trades to this client. Given that the server has total access over the feed, it can 1. drop trades; 2. introduce spurious trades; or 3. modify existing trades. Similar threats apply for selection queries, e.g., reporting all bids larger than a threshold within the last hour, and group by queries, e.g., reporting the potential value per market of a given portfolio. It will be the job of the stock broker to proof-infuse the stream such that the clients are guarded against such threats, guaranteeing both *correctness* and *completeness* of the results.

In the second part of our thesis we concentrate on *authenticated one-shot and sliding window queries on data streams*. To the best of our knowledge, no work has studied the authentication of exact selection and aggregation queries over streaming outsourced data. We introduce a variety of authentication algorithms for answering multi-dimensional (i.e., on multiple attributes) selection and aggregation queries. One-shot window queries report answers computed once over a user defined temporal range. Sliding window queries report answers continuously as they change, over user defined window sizes and update intervals. We assume that clients register a multiplicity of ad-hoc queries with the servers, which in turn compute the results and forward the answers back to the clients along with the necessary signatures and authentication information needed to construct a proof of correctness. For one-shot queries the servers construct answers and proofs, and send the results back to clients. For sliding window queries, the servers update the query results incrementally, communicating only authenticated changes to the clients. By ad-hoc queries

we mean that clients can register and cancel queries at any time, using arbitrary window sizes (sliding or not) and update intervals.

It should be emphasized here that one-shot queries are not only interesting in their own right, but also because they are an essential building block for answering sliding window queries: The initial answer of a sliding window query is constructed using a one-shot algorithm. We design one set of authentication structures optimized for one-shot window queries, and another for ad-hoc sliding window queries. Then, we combine both solutions to provide efficient algorithms for all settings. Our solutions are based on Merkle hash trees over a forest of space partitioning data structures, and balance key features, like update, query, signing and authentication cost, from the perspective of the service provider, the server and the client.

1.3 Continuous Query Execution Assurance on Data Streams

A large number of commercial Data Stream Management Systems (DSMS) have been developed recently [Chandrasekaran et al., 2003, Hammad et al., 2004, Arasu et al., 2003, Cranor et al., 2003b, Carney et al., 2003, Abadi et al., 2005], mainly driven by the continuous nature of the data being generated by a variety of real-world applications, like telephony and networking, and the stock trading example we have provided in last Section. Many companies deploy such DSMSs for the purpose of gathering invaluable statistics about their day-to-day operations. Not surprisingly, due to the overwhelming data flow observed in most data streams, some companies do not possess and are not willing to acquire the necessary resources for deploying a DSMS. Hence, in these cases outsourcing the data stream and the desired computations to a third-party is the only alternative. For example, an Internet Service Provider (e.g. Verizon) outsources the task of performing essential network traffic analysis to another company (e.g. AT&T) that already posses the appropriate tools for that purpose (e.g. Gigascope). Clearly, data outsourcing and remote computations intrinsically raise issues of trust. As a consequence, outsourced query execution assurance on data streams is a problem with important practical implications. This

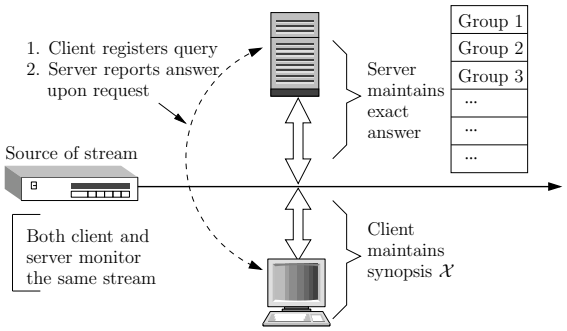


Figure 1·2: System architecture.

problem is a special form of the query authentication problem where the data owner and the clients are the same entity and it has been studied before in the context of static outsourced data [Sion, 2005]. To the best of our knowledge, this is the first work to address query execution assurance on data streams.

Consider a setting where continuous queries on a data stream are processed using a remote, untrusted server (that can be compromised, malicious, running faulty software, etc). A client with limited processing capabilities observing exactly the same input as the server, registers queries on the DSMS of the server and receives results upon request (Figure 1·2). Assuming that the server charges clients according to the computation resources consumed and the volume of traffic processed for answering the queries, the former has an incentive to deceive the latter for increased profit. Furthermore, the server might have a competing interest to provide fraudulent answers to a particular client. Hence, a passive malicious server can drop query results or provide random answers in order to reduce the computation resources required for answering queries, while a compromised or active malicious server might be willing to spend additional computational resources to provide fraudulent results (by altering, dropping, or introducing spurious answers). In other cases, incorrect answers might simply be a result of faulty software, or due to load shedding strategies, which are essential tools for dealing with bursty streaming data [Tatbul and Zdonik, 2006, Arasu and Manku, 2004, Babcock et al., 2004, Tatbul et al., 2003].

Ideally, the client should be able to verify the integrity of the computations performed by the server using significantly fewer resources than evaluating the queries locally. Moreover, the client should have the capability to tolerate errors caused by load shedding algorithms or other non-malicious operations, while at the same time being able to identify mal-intended attacks. To the best of our knowledge, this is the first work to address outsourced query execution assurance on streams, introducing a novel problem and proposing efficient solutions for a wide range of queries and failure models.

The present work concentrates on selection queries and aggregate queries, like sum and count. We develop solutions for verifying such aggregates on any type of grouping imposed on the input data (e.g., as a `GROUP BY` clause in standard SQL). First, we provide a solution for verifying the absolute correctness of queries in the presence of any error, and second, an algorithm for supporting *semantic load shedding*, which allows the server to drop tuples in a selected small number of groups. In the latter case we need to design techniques that can tolerate a small number of inconsistent answers while guaranteeing that the rest are correct. We also discuss the hardness of supporting *random load shedding*, where *small* errors are allowed for a *wide range* of answers.

Clearly, if a client wants to verify the query results with absolute confidence the only solution is to compute the answers exactly, which obviates the need of outsourcing. Hence, we investigate high-confidence probabilistic solutions and develop verification algorithms that significantly reduce resource consumption.

1.4 Summary of Contributions

This thesis answers the challenges, as we have illustrated above, posed by various aspects of the query authentication problem in the ODB system. We provide practical solutions that either improve existing approaches or closing the gap in important fields that no prior solutions exist. In summary the contributions of this thesis are: 1. Novel authentication structures for selection, projection and aggregation queries that best leverage all cost metrics; 2. Designing a variety of authentication algorithms for multi-dimensional selection

and aggregation queries on data streams, concentrating on both one-shot and variable-sized sliding window queries; 3. Detailed analytical cost models for existing and proposed approaches across all metrics, that take into account not only the structural maintenance overheads of the indices but the cost of cryptographic operations as well; 4. A probabilistic synopsis for query execution assurance over data streams that raises an alarm with very high confidence if there exists at least one error in the query results. This algorithm has constant space cost (three words) and low processing cost per update ($O(1)$ for count queries and $O(\log n)$ or $O(\log \mu)$ for sum queries, where n is the number of possible groups and μ is the update amount per tuple), for an arbitrarily small probability of failure δ (as long as n/δ fits in one word). We also provide a theoretical analysis of the algorithm that proves its space optimality on the bits level; 5. Extensions of this synopsis for handling multiple simultaneous queries, raising alarms when the number of errors exceeds a predefined threshold, e.g., when semantic load shedding needs to be supported, and dealing with random load shedding; 6. Finally, a fully working prototype and an comprehensive experimental evaluation and comparison of all solutions using both synthetic and real life datasets, showing that our algorithms not only provide strong theoretical guarantees, but also work extremely well in practice.

Chapter 2

Background

Existing solutions for the query authentication problem work as follows. The data owner creates specialized authenticated data structures over the original database and uploads them at the servers together with the database itself. These structures are used by the servers for answering queries, as well as providing special verification objects $\mathcal{VO}$ which clients can use for authenticating the results. Verification usually occurs by means of using collision-resistant hash functions and digital signature schemes. Note that in any solution, some information that is known to be authentically published by the owner must be made available to the client directly; otherwise, from the client's point of view, the owner cannot be differentiated from any other potentially malicious entity. For example, this information could be the owner's public key of any public signature scheme. For any authentication method to be successful it must be computationally infeasible for a malicious server to produce an incorrect query answer along with a verification object that will be accepted by a client that holds the correct authentication information of the owner.

2.1 Collision-resistant Hash Functions.

For our purposes, a hash function $\mathcal{H}$ is an efficiently computable function that takes a variable-length input x to a fixed-length output $y = \mathcal{H}(x)$. *Collision resistance* states that it is computationally infeasible to find two inputs $x_1 \neq x_2$ such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$. Collision-resistant hash functions can be built provably based on various cryptographic assumptions, such as hardness of discrete logarithms [McCurley, 1990]. However, in this work we concentrate on using heuristic hash functions that have the advantage of being very fast to evaluate, and focus on SHA1 [National Institute of Standards and Technology, 1995],

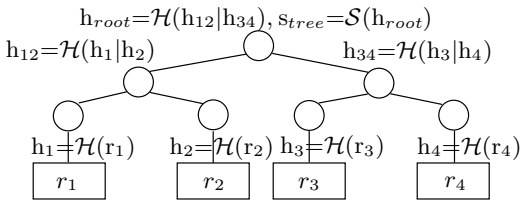


Figure 2·1: Example of a Merkle hash tree.

which takes variable-length inputs to 160-bit (20-byte) outputs. SHA1 is currently considered collision-resistant in practice. We also note that any eventual replacement to SHA1 developed by the cryptographic community can be used instead of SHA1 in our solution.

2.2 Public-key Digital Signature Schemes.

A public-key digital signature scheme, formally defined in [Goldwasser et al., 1988], is a tool for authenticating the integrity and ownership of the signed message. In such a scheme, the signer generates a pair of keys (SK, PK), keeps the secret key SK secret, and publishes the public key PK associated with her identity. Subsequently, for any message m that she sends, a signature s_m is produced by $s_m = \mathcal{S}(SK, m)$. The recipient of s_m and m can verify s_m via $\mathcal{V}(PK, m, s_m)$ that outputs "valid" or "invalid." A valid signature on a message assures the recipient that the owner of the secret key intended to authenticate the message, and that the message has not been modified. The most commonly used public digital signature scheme is RSA [Rivest et al., 1978]. Existing solutions [Pang et al., 2005, Pang and Tan, 2004, Mykletun et al., 2004a, Narasimha and Tsudik, 2005] for the query authentication problem chose to use this scheme, hence we adopt the common 1024-bit (128-byte) RSA. Its signing and verification cost is one hash computation and one modular exponentiation with 1024-bit modulus and exponent.

2.3 Aggregating Several Signatures.

In the case when t signatures $s_1, \ldots, s_t$ on t messages $m_1, \ldots, m_t$ signed by the same signer need to be verified all at once, certain signature schemes allow for more efficient communication and verification than t individual signatures. Namely, for RSA it is possible to combine the t signatures into a single aggregated signature $s_{1,t}$ that has the same size as an individual signature and that can be verified (almost) as fast as an individual signature. This technique is called Condensed-RSA [Mykletun et al., 2004b]. The combining operation can be done by anyone, as it does not require knowledge of SK; moreover, the security of the combined signature is the same as the security of individual signatures. In particular, aggregation of t RSA signatures can be done at the cost of $t-1$ modular multiplications, and verification can be performed at the cost of $t-1$ multiplications, t hashing operations, and one modular exponentiation (thus, the computational gain is that $t-1$ modular exponentiations are replaced by modular multiplications). Note that aggregating signatures is possible only for some digital signature schemes.

2.4 The Merkle Hash Tree.

The straightforward solution for verifying a set of n values is to generate n digital signatures, one for each value. An improvement on this straightforward solution is the Merkle hash tree (see Figure 2·1), first proposed by [Merkle, 1989]. It solves the simplest form of the query authentication problem for point queries and datasets that can fit in main memory. The Merkle hash tree is a binary tree, where each leaf contains the hash of a data value, and each internal node contains the hash of the concatenation of its two children. Verification of data values is based on the fact that the hash value of the root of the tree is authentically published (authenticity can be established by a digital signature). To prove the authenticity of any value, the prover provides the verifier with the data value itself and the hash values of the siblings of the nodes that lie in the path that connects the root of the tree with the data value. The verifier, by iteratively computing and concatenating the appropriate

hashes, can recompute the hash of the root and verify its correctness using the owner's signature. Correctness is guaranteed due to the collision-resistance of the hash functions. By hashing a given node, it becomes computationally infeasible for an adversary to modify the node in a way that ultimately preserves the hash value of the root. The correctness of any data value can be proved at the cost of computing $\log n$ hash values. Extensions for dynamic environments have appeared as well, such that insertions and deletions can be handled in $O(\log n)$ time [Naor and Nissim, 1998]. The Merkle tree concept can also be used to authenticate range queries using binary search trees (where data entries are sorted), and it has been shown how to guarantee completeness of the results as well (by including boundary values in the results) [Martel et al., 2004]. External memory Merkle B-trees have also been proposed [Pang and Tan, 2004]. Finally, it has been shown how to apply the Merkle tree concepts to more general data structures [Martel et al., 2004].

2.5 Cost Models for SHA1, RSA and Condensed-RSA.

Since all existing authenticated structures are based on SHA1 and RSA, it is imperative to evaluate the relative cost of these operations, in order to design efficient solutions in practice. Based on experiments with two widely used cryptography libraries, Crypto++ [Crypto++ Library, 2005] and OpenSSL [OpenSSL, 2005], we obtained results for hashing, signing, verifying and performing modulo multiplications. Evidently, one hashing operation for input up to 500 bytes on our testbed computer takes approximately 1 to 2 μs. Modular multiplication, signing and verifying are, respectively, approximately 50, 1000 and 100 times slower than hashing.

Thus, it is clear that multiplication, signing and verification operations are very expensive. The cost of these operations needs to be taken into account when designing authenticated index structures. In addition, since the cost of hashing is orders of magnitude smaller than that of singing, it is essential to design structures that use as few signing operations as possible, and hashing instead.

Chapter 3

Related Work

We have presented the essential cryptographic tools in Chapter 2. This Chapter provides a detailed review of other related work to different topics covered by our thesis.

3.1 Query Authentication for Offline, Relational Data

General issues for outsourced databases first appeared in [Hacigümüs et al., 2002b]. A large corpus of related work has appeared on authenticating queries for outsourced databases ever since [Devanbu et al., 2003, Martel et al., 2004, Bertino et al., 2004, Pang and Tan, 2004, Pang et al., 2005, Mykletun et al., 2004a, Sion, 2005, Goodrich et al., 2003, Miklau, 2005, Narasimha and Tsudik, 2005, Cheng et al., 2006]. The first set of attempts to address query authentication in ODB systems appeared in [Devanbu et al., 2000, Martel et al., 2004, Devanbu et al., 2003]. The focus of these works is on designing solutions for query correctness only, creating main memory structures that are based on Merkle trees. The work of [Martel et al., 2004] generalized the Merkle hash tree ideas to work with any DAG (directed acyclic graph) structure. With similar techniques, the work of [Bertino et al., 2004] uses the Merkle tree to authenticate XML documents in the ODB model. The work of [Pang and Tan, 2004] further extended the idea to the disk-based indexing structure and introduced the *VB-tree* which was suitable for datasets stored on secondary storage. However, this approach is expensive and was later subsumed by [Pang et al., 2005]. Several proposals for signature-based approaches appear in [Narasimha and Tsudik, 2005, Mykletun et al., 2004a, Pang et al., 2005]. Furthermore, most of previous work focused on the one dimensional queries except [Narasimha and Tsudik, 2005, Cheng et al., 2006]. Both works utilize the signature chaining idea developed in [Pang et al., 2005] and ex-

tend it for multi-dimensional range selection queries. There has been a lot of work on main memory authenticated data structures following the work in [Naor and Nissim, 1998]. However, these works, for example, [Anagnostopoulos et al., 2001, Goodrich et al., 2003, Tamassia and Triandopoulos, 2005a], concentrate on main memory structures and are not directly applicable to databases resided on secondary storage medias. Furthermore, most of these works presented techniques that are mostly theoretical focus. A general discussion has been provided by [Tamassia and Triandopoulos, 2005a] to bound the cost of Merkle tree based main memory index structures for authenticating selection queries.

Previous work has focused on studying the general selection and projection queries. The proposed techniques can be categorized into two groups, signature-based approaches and index-based approaches. In general, projection queries are handled using the same techniques as selection queries, the only difference being the granularity level at which the signature/hash of the database tuple is computed (i.e. tuple level or attribute level). Therefore, in the rest we focus on selection queries only.

In signature-based approaches [Narasimha and Tsudik, 2005, Pang et al., 2005], the general idea is to produce a signature for each tuple in the database. Suppose that there exists an authenticated index with respect to a query attribute A_q. The signatures are produced by forcing the owner to sign the hash value of the concatenation of every tuple with its right neighbor in the total ordering of A_q. To answer a selection query for $A_q \in [a, b]$, the server returns all tuples t with $A_q \in [a-1, b+1]$ along with a $\mathcal{VO}$ that contains a set of signatures $s(t_i)$ corresponding to tuples in $[a-1, b]$. The client authenticates the result by verifying these signatures for all consecutive pairs of tuples t_i, t_{i+1} in the result. It is possible to reduce the number of signatures transmitted by using special aggregation signature techniques [Mykletun et al., 2004b]. Index-based approaches [Devanbu et al., 2000, Martel et al., 2004, Mykletun et al., 2004a] utilize the Merkle hash tree to provide authentication. The owner builds a Merkle tree on the tuples in the database, based on the query attribute. Subsequently, the server answers the selection query using the tree, by returning all tuples t covering the result. In addition,

the server also returns the minimum set of hashes necessary for the client to reconstruct the subtree of the Merkle tree corresponding to the query result and compute the hash of the root. The reconstruction of the whole subtree is necessary for providing the proof of completeness.

We are not aware of work that specifically addresses the query freshness issue and to the best of our knowledge none of the existing works has explored aggregation queries.

The work of [Bouganim et al., 2003, Bouganim et al., 2005] offers a promising research direction for designing query authentication schemes with special hardware support. Distributed content authentication has been addressed in [Tamassia and Triandopoulos, 2005b], where a distributed version of the Merkle hash tree is applied. In [Miklau and Suciu, 2005], a new hash tree is designed based on the relational interval tree that is implemented on top of a database management system. However, despite its simplicity is expected to be much more expensive than the techniques presented here since the tree is stored on tables and the traversal is done through SQL queries. Techniques for integrity in data exchange can be applied for query authentication and we refer readers to an excellent thesis [Miklau, 2005] for more details. Also, [Mykletun and Tsudik, 2006] has studied the problem of computing aggregations over encrypted databases and [Ge and Zdonik, 2007] presented a technique to index encrypted data in a column-oriented DBMS. We discuss encrypted databases in more depth in Chapter 4.5. Another interesting related work investigates trustworthy index structures based on Write-Once-Read-Many (WORM) storage devices [Zhu and Hsu, 2005, Mitra et al., 2006]. In these work the main goal is to make sure that all the records or documents are maintained forever (or at least for a predefined retention period) and they are never deleted.

We would like to point out that there are other security issues that are orthogonal to the problems considered here. Examples include privacy-preservation [Hore et al., 2004, Agrawal and Srikant, 2000, Evfimievski et al., 2003], watermarking databases to prove ownerships [Sion et al., 2003, Sion, 2004, Sion et al., 2004], security in conjunction with access control requirements [Miklau and Suciu, 2003, Rizvi et al., 2004, Bouganim et al., 2003,

Pang et al., 2005] and secure query execution [Hacigümüs et al., 2002a] .

3.2 Query Authentication for Streaming Data

To the best of our knowledge, this thesis is the first to address query authentication issues on sliding window queries for outsourced streams. Nevertheless, previous work utilizes similar cryptographic primitives as the techniques proposed in this thesis, however, for selection, projection and aggregation queries on offline, relational data, see our review in last Section. Our solutions leverage on incorporating Merkle tree into kd-tree for providing necessary authentication information for multi-dimensional queries over data streams. These extensions for the kd-tree (and similarly for other multi-dimensional structures like the R-tree) are straightforward but new. Most importantly, arranging these structures into hierarchies, making them suitable for sliding windows over data streams, and studying their performance over a variety of cost metrics has not been addressed before.

Sliding window queries over data streams have been studied extensively recently. Readers are referred to an excellent survey and related work therein [Babcock et al., 2002]. Techniques that are in particular interested to us include dyadic ranges [Gilbert et al., 2002, Cormode and Muthukrishnan, 2003] and exponential hierarchies [Datar et al., 2002] that have been utilized by a number of algorithms to solve sliding window problems. As we will learn in Chapter 5, our improved versions of the Mkd-trees use the similar idea. Readers are referred to an excellent thesis [Golab, 2006] for more details.

We would like to point out that other security issues for secure computation/querying over streaming and sensor data start to receive attention recently. For example, orthogonal to our problem, [Chan et al., 2006, Garofalakis et al., 2007] have studied the problem of secure in-network aggregation for aggregation queries in sensor networks. Both works utilize cryptographic tools, such as digital signatures, as building blocks for their algorithms and assume the man-in-middle attack model. Furthermore, the verifier does not have access to the original data stream. Hence, they are fundamentally different from our work. Nevertheless, they attest to the fact that secure computation of aggregation queries

has a profound impact in many real applications. Our problem is also orthogonal to watermarking techniques for proving ownership of streams [Sion et al., 2006]

3.3 Query Execution Assurance on Data Streams

Our solutions to query execution assurance on data streams are a form of randomized synopses. These randomized synopses are a way of summarizing the underlying data streams. In that respect our work is related with the line of work on sketching techniques [Alon et al., 1996, Flajolet and Martin, 1985, Ganguly et al., 2004, Babcock et al., 2003, Manku and Motwani, 2002, Cormode and Muthukrishnan, 2005]. As it will be discussed in Chapter 6.2, since these sketches are mainly designed for estimating certain statistics of the stream (e.g. the frequency moments), they cannot solve the query execution assurance problem.

Another approach for solving our problem is to use program execution verification techniques [Blum and Kannan, 1995, Wasserman and Blum, 1997] on the DSMS of the server. We briefly discuss two representatives from this field [Chen et al., 2002, Yang et al., 2005]. The main idea behind these approaches is for the client to precompute hash values in nodes of the *control flow graph* of the query evaluation algorithm on the server's DSMS. Subsequently, the client can randomly select a subset of nodes from the expected execution path and ask the server for the corresponding hashes. If the sever does not follow the correct execution it will be unable to provide the correct hashes. With guaranteed probability (with respect to the sampling ratio) the client will be able to verify whether the query has been honestly executed by the server or not. Clearly, these techniques do not solve our problem since, first, the server can honestly execute the algorithm throughout and only modify the results before transmitting them to the client, and second, there is no way for the client to compute the correct hashes of the program execution, unless if it is recomputing the same algorithm in real-time, or storing the whole stream for later processing.

Various authentication techniques are also relevant to the present work. The idea is that by viewing $\mathbf{v}$, the dynamic stream vector which is the the query result computed on the fly

with the incoming stream, as a message, the client could compute an authenticated signature $s(\mathbf{v})$ and any alteration to $\mathbf{v}$ will be detected. Here the authenticated signature refers to any possible authentication techniques, such as RSA or Message Authentication Code (MAC). However, a fundamental challenge in applying such techniques to our problem is how to perform incremental updates using $s(\mathbf{v})$ alone, without storing $\mathbf{v}$, i.e., in the present setting the message $\mathbf{v}$ is constantly updated. Cryptography researchers have devoted considerable effort in designing *incremental cryptography* [Bellare et al., 1994]. Among them incremental signature and incremental MAC [Bellare et al., 1994, Bellare et al., 1995] are especially interesting for this work. However, these techniques only support updates for *block edit operations* such as insert and delete, i.e., by viewing $\mathbf{v}$ as blocks of bits, they are able to compute $s(\mathbf{v}')$ using $s(\mathbf{v})$ alone if $\mathbf{v}'$ is obtained by inserting a new block (or deleting an old block) into (from) $\mathbf{v}$. However, in our setting the update operation is arithmetic: $v_i^\tau = v_i^{\tau-1} + u^\tau$ where τ denotes any particular time instance, which cannot be handled by simply deleting the old entry followed by inserting the new one, since we only have u^τ (the incoming tuple) as input, and no access to either $v_i^{\tau-1}$ or v_i^τ. Hence, such cryptographic approaches are inapplicable.

There is also considerable work on authenticating query execution in an outsourced database setting. Readers are referred to the detailed discussion in Section 3.1. Here, the client queries the publisher's data through a third party, namely the server, and the goal is to design efficient solutions to enable the client to authenticate the query results. Therefore, the model is fundamentally different from our problem. The closest previous work appears in [Sion, 2005], where it is assumed that the client possess a copy of the data resided on the servers and a random sampling based approach (by posing test queries with known results) is used. This model is essentially same as it is in our problem, however, its discussion limits to offline settings and not for online, one-pass streaming scenarios. Authentication of sliding window queries over data streams has been studied in this thesis as well, however, the outsourced model is different from what we have presented for this problem. The client does not possess the data stream and the data publisher is responsible

for injecting "proofs" (in the forms of some cryptographic primitives) into its data stream so that the server could construct a verification object associated with a query requested by the client. Hence, it is fundamentally different from the problem we have studied in query execution assurance (where the client and the data publisher are the same entity).

Finally and most related to our proposed techniques, verifying the identity of polynomials is a fingerprinting technique [Motwani and Raghavan, 1995]. Fingerprinting is a method for efficient, probabilistic checking of equality between two elements x, y from a large universe U. Instead of testing the equality using x, y deterministically with complexity at least $\log |U|$, a probabilistic approach is to pick a random mapping from U to a significantly smaller universe V such that with high probability x, y are identical if and only if their images in V are identical. The images of x and y are their fingerprints and their equality can be verified in $\log |V|$ time. Fingerprint techniques generally employ algebraic techniques combined with randomization. Classical examples include verifying univariate polynomial multiplication [Freivalds, 1979], multivariate polynomial identities [Schwartz, 1980], and verifying equality of strings [Motwani and Raghavan, 1995]. We refer readers for an excellent discussion on these problems to [Motwani and Raghavan, 1995]. Although the general technique of polynomial identity verification is known, our use of it in the setting of query verification on data streams appears to be new.

Chapter 4

Authenticated Index Structures for Relational Databases

4.1 Problem Definition

This Chapter focuses on authentication of range selection queries of the following form[1]:

$$\text{SELECT } A_1, \cdots, A_c \text{ FROM } \mathbf{T} \text{ WHERE } a_1 \leq S_1 \leq b_1, \cdots, a_d \leq S_d \leq b_d,$$

and range aggregate queries of the following form[2]:

$$\text{SELECT } \otimes(A_1), \cdots, \otimes(A_c) \text{ FROM } \mathbf{T} \text{ WHERE } a_1 \leq S_1 \leq b_1, \cdots, a_d \leq S_d \leq b_d,$$

where $\bigotimes$ denotes an aggregation operation, A_i correspond to the projection/aggregation attributes and S_j to the selection attributes. Attributes A_i, S_j may correspond to any of the fields of $\mathbf{T}$. For simplicity and without loss of generality, we assume that the schema of $\mathbf{T}$ consists of fields $S_1, \ldots, S_d$ with domains $D_1, \ldots, D_d$, respectively. We refer to a query Q with d selection predicates as a d-dimensional query.

The database owner compiles a set of authenticated structures for $\mathbf{T}$, and disseminates them, along with $\mathbf{T}$, to a set of servers. Clients pose selection and aggregation queries Q to the servers, which use the authenticated structures to provide query answers $\mathcal{ANS}(Q)$ along with verification objects $\mathcal{VO}$. Denote with $\mathcal{SAT}(Q)$ the set of tuples from $\mathbf{T}$ that satisfy Q. The $\mathcal{VO}$ is constructed w.r.t. $\mathcal{ANS}(Q)$ and it enables the client to verify the correctness, completeness and freshness of the answer. The problem is to design efficient

[1]The discussion for authenticating range selection queries is based on [Li et al., 2006b].
[2]The discussion for authenticating range aggregate queries is based on [Li et al., 2006a].

Table 4.1: Notation used in Chapter 4.

Symbol	Description
r	A database record
k	A B^+-tree key
p	A B^+-tree pointer
h	A hash value
s	A signature
$\|x\|$	Size of object x
D_i	Domain size in the ith dimension
N_D	Total number of database records
N_R	Total number of query results
P	Page size
$\mathcal{ANS}(Q)$	Final answer to an aggregation query
$\mathcal{SAT}(Q)$	Tuples satisfying the query predicate for an aggregation query
f_x	Node fanout of structure x
d_x	Height of structure x
$\mathcal{H}_l(x)$	A hash operation on input x of length l
$\mathcal{S}_l(x)$	A signing operation on input x of length l
$\mathcal{V}_l(x)$	A verifying operation on input x of length l
$\mathcal{C}_x$	Cost of operation x
$\mathcal{VO}$	The verification object

authentication structures for constructing the $\mathcal{VO}$. The efficiency of each solution should be measured with respect to the cost metrics introduced in Chapter 1.1.

In our model, only the data owner can update the database. For aggregation queries we concentrate on distributive (SUM, COUNT, MAX, MIN) and algebraic (AVG) aggregates. We also support HAVING operations. We do not considerthe issue of the data confidentiality as it is orthogonal to our problem. Finally, access control constraints on top of query authentication as discussed in [Pang et al., 2005] are not considered.

4.2 Authenticated Index Structures for Selection Queries

4.2.1 The Static Case

We illustrate three approaches for query correctness and completeness: a signature-based approach similar to the ones described in [Pang et al., 2005, Narasimha and Tsudik, 2005],

a Merkle-tree-like approach based on the ideas presented in [Martel et al., 2004], and our novel embedded tree approach. We present them for the *static scenario* where no data updates occur between the owner and the servers on the outsourced database. We also present *analytical cost models* for all techniques, given a variety of performance metrics. As already mentioned, detailed analytical modelling was not considered in related literature and is an important contribution of this work. It gives the ability to the owners to decide which structure best satisfies their needs, using a variety of performance measures.

In the following, we derive models for the *storage*, *construction*, *query*, and *authentication* cost of each technique, taking into account the overhead of hashing, signing, verifying data, and performing expensive computations (like modular multiplications of large numbers). The analysis considers range queries on a specific database attribute A indexed by a B^+-tree [Comer, 1979]. The size of the structure is important first for quantifying the storage overhead on the servers, and second for possibly quantifying the owner/server communication cost. The construction cost is useful for quantifying the overhead incurred by the database owner for outsourcing the data. The query cost quantifies the incurred server cost for answering client queries, and hence the potential query throughput. The authentication cost quantifies the server/client communication cost and, in addition, the client side incurred cost for verifying the query results. The notation used is summarized in Table 4.1. In the rest, for ease of exposition, it is assumed that all structures are bulk-loaded in a bottom-up fashion and that all index nodes are completely full. Extensions for supporting multiple selection attributes are discussed in Section 4.5.

Aggregated Signatures with B$^+$-trees

The first authenticated data structure for static environments is a direct extension of aggregated signatures and ideas that appeared in [Narasimha and Tsudik, 2005, Pang et al., 2005]. To guarantee correctness and completeness the following technique can be used: First, the owner individually hashes and signs all consecutive pairs of tuples in the database, assuming some sorted order on a given attribute A. For example, given two consecutive tuples

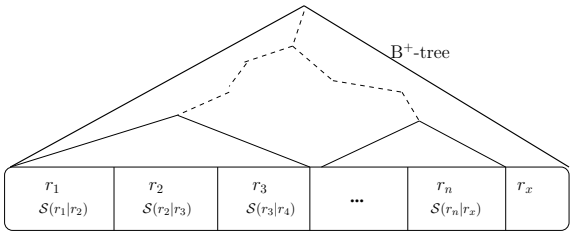


Figure 4·1: The signature-based approach.

r_i, r_j the owner transmits to the servers the pair (r_i, s_i), where $s_i = \mathcal{S}(r_i|r_j)$ ('|' denotes some canonical pairing of strings that can be uniquely parsed back into its two components; e.g., simple string concatenation if the lengths are fixed). The first and last tuples can be paired with special marker records. Chaining tuples in this way will enable the clients to verify that no in-between tuples have been dropped from the results or modified in any way. An example of this scheme is shown in Figure 4·1.

In order to speed up query execution on the server side a B^+-tree is constructed on top of attribute A. To answer a query the server constructs a $\mathcal{VO}$ that contains one pair $r_q|s_q$ per query result. In addition, one tuple to the left of the lower-bound of the query results and one to the right of the upper-bound is returned, in order for the client to be able to guarantee that no boundary results have been dropped. Notice that since our completeness requirements are less stringent than those of [Pang et al., 2005] (where they assume that database access permissions restrict which tuples the database can expose to the user), for fairness we have simplified the query algorithm substantially here.

There are two obvious and serious drawbacks associated with this approach. First, the extremely large $\mathcal{VO}$ size that contains a linear number of signatures w.r.t. N_R (the total number of query results), taking into account that signature sizes are very large. Second, the high verification cost for the clients. Authentication requires N_R verification operations which, as mentioned earlier, are very expensive. To solve this problem one can use the aggregated signature scheme discussed in Section 2.3. Instead of sending one signature per query result the server can send one *combined signature* s^π for all results,

and the client can use an aggregate verification instead of individual verifications.

By using aggregated RSA signatures, the client can authenticate the results by hashing consecutive pairs of tuples in the result-set, and calculating the product $m^{\pi} = \prod_{\forall q} h_q \pmod{n}$ (where n is the RSA modulus from the public key of the owner). It is important to notice that both s^{π} and m^{π} require a linear number of modular multiplications (w.r.t. N_R). The cost models of the aggregated signature scheme for the metrics considered are as follows:

Node fanout: The node fanout of the B^+-tree structure is:

$$f_a = \frac{P - |p|}{|k| + |p|} + 1\,. \tag{4.1}$$

where P is the disk page size, $|k|$ and $|p|$ are the sizes of a B^+-tree key and pointer respectively.

Storage cost: The total size of the authenticated structure (excluding the database itself) is equal to the size of the B^+-tree plus the size of the signatures. For a total of N_D tuples the height of the tree is equal to $d_a = \log_{f_a} N_D$, consisting of $N_I = \frac{f_a^{d_a}-1}{f_a-1}$ $(= \sum_{i=0}^{d_a-1} f_a^i)$ nodes in total. Hence, the total storage cost is equal to:

$$\mathcal{C}_s^a = P \cdot \frac{f_a^{d_a} - 1}{f_a - 1} + N_D \cdot |s|. \tag{4.2}$$

The storage cost also reflects the initial communication cost between the owner and servers. Notice that the owner does not have to upload the B^+-tree to the servers, since the latter can rebuild it by themselves, which will reduce the owner/server communication cost but increase the computation cost at the servers. Nevertheless, the cost of sending the signatures cannot be avoided.

Construction cost: The cost incurred by the owner for constructing the structure has three components: the signature computation cost, bulk-loading the B^+-tree, and the I/O cost for storing the structure. Since the signing operation is very expensive, it dominates

the overall cost. Bulk-loading the B^+-tree in main memory is much less expensive and its cost can be omitted. Hence:

$$\mathcal{C}_c^a = N_D \cdot (\mathcal{C}_{\mathcal{H}_{|r|}} + \mathcal{C}_{\mathcal{S}_{2|h|}}) + \frac{\mathcal{C}_s^a}{P} \cdot \mathcal{C}_{IO}. \tag{4.3}$$

$\mathcal{VO}$ construction cost: The cost of constructing the $\mathcal{VO}$ for a range query depends on the total disk I/O for traversing the B^+-tree and retrieving all necessary record/signature pairs, as well as on the computation cost of s^π. Assuming that the total number of leaf pages accessed is $N_Q = \frac{N_R}{f_a}$, the $\mathcal{VO}$ construction cost is:

$$\mathcal{C}_q^a = (N_Q + d_a - 1 + \frac{N_R \cdot |r|}{P} + \frac{N_R \cdot |s|}{P}) \cdot \mathcal{C}_{IO} + \mathcal{C}_{s^\pi}, \tag{4.4}$$

where the last term is the modular multiplication cost for computing the aggregated signature, which is linear to N_R. The I/O overhead for retrieving the signatures is also large.

Authentication cost: The size of the $\mathcal{VO}$ is equal to the result-set size plus the size of one signature:

$$|\mathcal{VO}|^a = N_R \cdot |r| + |s|. \tag{4.5}$$

The cost of verifying the query results is dominated by the hash function computations and modular multiplications at the client:

$$\mathcal{C}_v^a = N_R \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + \mathcal{C}_{m^\pi} + \mathcal{C}_{\mathcal{V}_{|n|}}, \tag{4.6}$$

where the modular multiplication cost for computing the aggregated hash value is linear to the result-set size N_R, and the size of the final product has length in the order of $|n|$ (the RSA modulus). The final term is the cost of verifying the product using s^π and the owner's public key.

It becomes obvious now that one advantage of the aggregated signature scheme is that it features small $\mathcal{VO}$ sizes and hence small client/server communication cost. On the other hand it has the following serious drawbacks: 1. Large storage overhead on the servers,

dominated by the large signature sizes, 2. Large communication overhead between the owners and the servers that cannot be reduced, 3. A very high initial construction cost, dominated by the cost of computing the signatures, 4. Added I/O cost for retrieving signatures, linear to N_R, 5. An added modular multiplication cost, linear to the result-set size, for constructing the $\mathcal{VO}$ and authenticating the results. Our experimental evaluation shows that this cost is significant compared to other operations, 6. The requirement for a public key signature scheme that supports aggregated signatures. For the rest of this thesis, this approach is denoted as Aggregated Signatures with B^+-trees (ASB-tree). The ASB-tree has been generalized to work with multi-dimensional selection queries in [Narasimha and Tsudik, 2005, Cheng et al., 2006].

The Merkle B-tree

Motivated by the drawbacks of the ASB-tree, we present a different approach for building authenticated structures that is based on the general ideas of [Martel et al., 2004] (which utilize the Merkle hash tree) applied in our case on a B^+-tree structure. We term this structure the Merkle B-tree (MB-tree).

As already explained in Section 2.4, the Merkle hash tree uses a hierarchical hashing scheme in the form of a binary tree to achieve query authentication. Clearly, one can use a similar hashing scheme with trees of *higher fanout and with different organization algorithms*, like the B^+-tree, to achieve the same goal. An MB-tree works like a B^+-tree and also consists of ordinary B^+-tree nodes that are extended with one hash value associated with every pointer entry. The hash values associated with entries on leaf nodes are computed on the database records themselves. The hash values associated with index node entries are computed on the concatenation of the hash values of their children. For example, an MB-tree is illustrated in Figure 4·2. A leaf node entry is associated with a hash value $h = \mathcal{H}(r_i)$, while an index node entry with $h = \mathcal{H}(h_1|\cdots|h_{f_m})$, where $h_1, \ldots, h_{f_m}$ are the hash values of the node's children, assuming fanout f_m per node. After computing all hash values, the owner has to sign the hash of the root using its secret key SK.

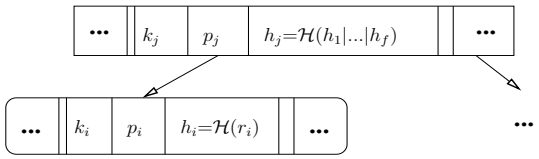


Figure 4·2: An MB-tree node.

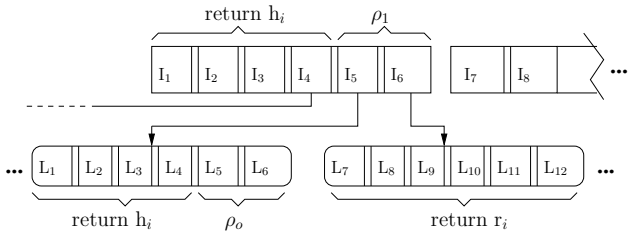


Figure 4·3: A query traversal on an MB-tree. At every level the hashes of the residual entries on the left and right boundary nodes need to be returned.

To answer a range query the server builds a $\mathcal{VO}$ by initiating two top-down B^+-tree like traversals, one to find the left-most and one the right-most query result. At the leaf level, the data contained in the nodes between the two discovered boundary leaves are returned, as in the normal B^+-tree. The server also needs to include in the $\mathcal{VO}$ the hash values of the entries contained in each index node that is visited by the lower and upper boundary traversals of the tree, except the hashes to the right (left) of the pointers that are traversed during the lower (upper) boundary traversals. At the leaf level, the server inserts only the answers to the query, along with the hash values of the residual entries to the left and to the right parts of the boundary leaves. The result is also increased with one tuple to the left and one to the right of the lower-bound and upper-bound of the query result respectively, for completeness verification. Finally, the signed root of the tree is inserted as well. An example query traversal is shown in Figure 4·3.

The client can iteratively compute all the hashes of the sub-tree corresponding to the query result, all the way up to the root using the $\mathcal{VO}$. The hashes of the query results are

computed first and grouped into their corresponding leaf nodes[3], and the process continues iteratively, until all the hashes of the query sub-tree have been computed. After the hash value of the root has been computed, the client can verify the correctness of the computation using the owner's public key PK and the signed hash of the root. It is easy to see that since the client is forced to recompute the whole query sub-tree, both correctness and completeness is guaranteed. It is interesting to note here that one could avoid building the whole query sub-tree during verification by individually signing all database tuples as well as each node of the B^+-tree. This approach, called VB-tree, was proposed in [Pang and Tan, 2004] but it is subsumed by the ASB-tree. The analytical cost models of the MB-tree are as follows:

Node fanout: The node fanout in this case is:

$$f_m = \frac{P - |p| - |h|}{|k| + |p| + |h|} + 1. \tag{4.7}$$

Notice that the maximum node fanout of the MB-tree is considerably smaller than that of the ASB-tree, since the nodes here are extended with one hash value per entry. This adversely affects the total height of the MB-tree.

Storage cost: The total size is equal to:

$$\mathcal{C}_s^m = P \cdot \frac{f_m^{d_m} - 1}{f_m - 1} + |s|. \tag{4.8}$$

An important advantage of the MB-tree is that the storage cost does not necessarily reflect the owner/server communication cost. The owner, after computing the final signature of the root, does not have to transmit all hash values to the server, but only the database tuples. The server can recompute the hash values incrementally by recreating the MB-tree. Since hash computations are cheap, for a small increase in the server's computation cost this technique will reduce the owner/sever communication cost drastically.

[3]Extra node boundary information can be inserted in the $\mathcal{VO}$ for this purpose with a very small overhead.

Construction cost: The construction cost for building an MB-tree depends on the hash function computations and the total I/Os. Since the tree is bulk-loaded, building the leaf level requires N_D hash computations of input length $|r|$. In addition, for every tree node one hash of input length $f_m \cdot |h|$ is computed. Since there are a total of $N_I = \frac{f_m^{d_m}-1}{f_m-1}$ nodes on average (given height $d_m = \log_{f_m} N_D$), the total number of hash function computations, and hence the total cost for constructing the tree is given by:

$$\mathcal{C}_c^m = N_D \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + N_I \cdot \mathcal{C}_{\mathcal{H}_{f_m|h|}} + \mathcal{C}_{\mathcal{S}_{|h|}} + \frac{\mathcal{C}_s^m}{P} \cdot \mathcal{C}_{IO}. \tag{4.9}$$

$\mathcal{VO}$ construction cost: The $\mathcal{VO}$ construction cost is dominated by the total disk I/O. Let the total number of leaf pages accessed be equal to $N_Q = \frac{N_R}{f_m}$, $d_m = \log_{f_m} N_D$ and $d_q = \log_{f_m} N_R$ be the height of the MB-tree and the query sub-tree respectively. In the general case the index traversal cost is:

$$\mathcal{C}_q^m = [(d_m - d_q + 1) + 2(d_q - 2) + N_Q + \frac{N_R \cdot |r|}{P}] \cdot \mathcal{C}_{IO}, \tag{4.10}$$

taking into account the fact that the query traversal at some point splits into two paths. It is assumed here that the query range spans at least two leaf nodes. The first term corresponds to the hashes inserted for the common path of the two traversals from the root of the tree to the root of the query sub-tree. The second term corresponds to the cost of the two boundary traversals after the split. The last two terms correspond to the cost of the leaf level traversal of the tree and accessing the database records.

Authentication cost: Assuming that ρ_0 is the total number of query results contained in the left boundary leaf node of the query sub-tree, σ_0 on the right boundary leaf node, and ρ_i, σ_i the total number of entries of the left and right boundary nodes on level $i, 1 \leq i \leq d_q$,

that point towards leaves that contain query results (see Figure 4·3), the size of the $\mathcal{VO}$ is:

$$|\mathcal{VO}|^m = (2f_m - \rho_0 - \sigma_0)|h| + N_R \cdot |r| + |s| + (d_m - d_q) \cdot (f_m - 1)|h| + \sum_{i=1}^{d_q-2} (2f_m - \rho_i - \sigma_i)|h| + (f_m - \rho_{d_q-1} - \sigma_{d_q-1})|h|. \tag{4.11}$$

This cost does not include the extra boundary information needed by the client in order to group hashes correctly, but this overhead is very small (one byte per node in the $\mathcal{VO}$) especially when compared with the hash value size. Consequently, the verification cost on the client is:

$$\mathcal{C}_v^m = N_R \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + \sum_{i=0}^{d_q-1} f_m^i \cdot \mathcal{C}_{\mathcal{H}_{f_m|h|}} + (d_m - d_q) \cdot \mathcal{C}_{\mathcal{H}_{f_m|h|}} + \mathcal{C}_{\mathcal{V}_{|h|}}. \tag{4.12}$$

Given that the computation cost of hashing versus signing is orders of magnitude smaller, the initial construction cost of the MB-tree is expected to be orders of magnitude less expensive than that of the ASB-tree. Given that the size of hash values is much smaller than that of signatures and that the fanout of the MB-tree will be smaller than that of the ASB-tree, it is not easy to quantify the exact difference in the storage cost of these techniques, but it is expected that the structures will have comparable storage cost, with the MB-tree being smaller. The $\mathcal{VO}$ construction cost of the MB-tree will be much smaller than that of the ASB-tree, since the ASB-tree requires many I/Os for retrieving signatures, and also some expensive modular multiplications. The MB-tree will have smaller verification cost as well since: 1. Hashing operations are orders of magnitude cheaper than modular multiplications, 2. The ASB-tree requires N_R modular multiplications for verification. The only drawback of the MB-tree is the large $\mathcal{VO}$ size, which increases the client/server communication cost. Notice that the $\mathcal{VO}$ size of the MB-tree is bounded by $f_m \cdot \log_{f_m} N_D$. Since generally $f_m \gg \log_{f_m} N_D$, the $\mathcal{VO}$ size is essentially determined by f_m, resulting in large sizes.

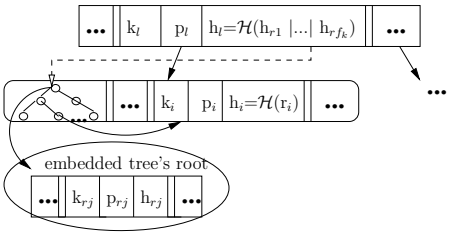


Figure 4·4: An EMB-tree node.

The Embedded Merkle B-tree

In this section we propose a novel data structure, the Embedded Merkle B-tree (EMB-tree), that provides a nice, adjustable trade-off between robust initial construction and storage cost versus improved $\mathcal{VO}$ construction and verification cost. The main idea is to have different fanouts for storage and authentication and yet combine them in the same data structure.

Every EMB-tree node consists of regular B^+-tree entries, augmented with an embedded MB-tree. Let f_e be the fanout of the EMB-tree. Then each node stores up to f_e triplets $k_i|p_i|h_i$, and an embedded MB-tree with fanout $f_k < f_e$. The leaf level of this embedded tree consists of the f_e entries of the node. The hash value at the root level of this embedded tree is stored as an h_i value in the parent of the node, thus authenticating this node to its parent. Essentially, we are collapsing an MB-tree with height $d_e \cdot d_k = \log_{f_k} N_D$ into a tree with height d_e that stores smaller MB-trees of height d_k within each node. Here, $d_e = \log_{f_e} N_D$ is the height of the EMB-tree and $d_k = \log_{f_k} f_e$ is the height of each small embedded MB-tree. An example EMB-tree node is shown in Figure 4·4.

For ease of exposition, in the rest of this discussion it will be assumed that f_e is a power of f_k such that the embedded trees when bulk-loaded are always full. The technical details if this is not the case can be worked out easily. The exact relation between f_e and f_k will be discussed shortly. After choosing f_k and f_e, bulk-loading the EMB-tree is straightforward: Simply group the N_D tuples in groups of size f_e to form the leaves and build their embedded trees on the fly. Continue iteratively in a bottom up fashion.

When querying the structure the server follows a path from the root to the leaves of the external tree as in the normal B^+-tree. For every node visited, the algorithm scans all $f_e - 1$ triplets $k_i|p_i|h_i$ on the data level of the embedded tree to find the key that needs to be followed to the next level. When the right key is found the server also initiates a point query on the embedded tree of the node using this key. The point query will return all the needed hash values for computing the concatenated hash of the node, exactly like for the MB-tree. Essentially, these hash values would be the equivalent of the $f_e - 1$ sibling hashes that would be returned per node if the embedded tree was not used. However, since now the hashes are arranged hierarchically in an f_k-way tree, the total number of values inserted in the $\mathcal{VO}$ per node is reduced to $(f_k - 1)d_k$.

To authenticate the query results the client uses the normal MB-tree authentication algorithm to construct the hash value of the root node of each embedded tree (assuming that proper boundary information has been included in the $\mathcal{VO}$ for separating groups of hash values into different nodes) and then follows the same algorithm once more for computing the final hash value of the root of the EMB-tree.

The EMB-tree structure uses extra space for storing the index levels of the embedded trees. Hence, by construction it has increased height compared to the MB-tree due to smaller fanout f_e. A first, simple optimization for improving the fanout of the EMB-tree is to avoid storing the embedded trees altogether. Instead, each embedded tree can be instantiated by computing fewer than $f_e/(f_k - 1)$ hashes on the fly, only when a node is accessed during the querying phase. We call this the EMB^--tree. The EMB^--tree is logically the same as the EMB-tree, however its physical structure is equivalent to an MB-tree with the hash values computed differently. The querying algorithm of the EMB^--tree is slightly different than that of the EMB-tree in order to take into account the conceptually embedded trees. With this optimization the storage overhead is minimized and the height of the EMB^--tree becomes equal to the height of the equivalent MB-tree. The trade-off is an increased computation cost for constructing the $\mathcal{VO}$. However, this cost is minimal as the number of embedded trees that need to be reconstructed is bounded by the height

of the EMB^--tree.

As a second optimization, one can create a slightly more complicated embedded tree to reduce the total size of the index levels and increase fanout f_e. We call this the EMB^*-tree. Essentially, instead of using a B^+-tree as the base structure for the embedded trees, one can use a multi-way search tree with fanout f_k while keeping the structure of the external EMB-tree intact. The embedded tree based on B^+-trees has a total of $N_i = \frac{f_k^{d_k}-1}{f_k-1}$ nodes while, for example, a B-tree based embedded tree (recall that a B-tree is equivalent to a balanced multi-way search tree) would contain $N_i = \frac{f_e-1}{f_k-1}$ nodes instead. A side effect of using multi-way search trees is that the cost for querying the embedded tree on average will decrease, since the search for a particular key might stop before reaching the leaf level. This will reduce the expected cost of $\mathcal{VO}$ construction substantially. Below we give the analytical cost models of the EMB-tree. The further technical details and the analytical cost models associated with the EMB^*-tree and EMB^--tree are similar to the EMB-tree case and can be worked out similarly.

Node fanout: For the EMB-tree, the relationship between f_e and f_k is given by:

$$P \geq \frac{f_k^{\log_{f_k} f_e - 1} - 1}{f_k - 1}[f_k(|k| + |p| + |h|) - |k|] + [f_e(|k| + |p| + |h|) - |k|]. \tag{4.13}$$

First, a suitable f_k is chosen such that the requirements for authentication cost and storage overhead are met. Then, the maximum value for f_e satisfying (4.13) can be determined.

Storage cost: The storage cost is equal to:

$$\mathcal{C}_s^e = P \cdot \frac{f_e^{d_e} - 1}{f_e - 1} + |s|. \tag{4.14}$$

Construction cost: The total construction cost is the cost of constructing all the embedded trees plus the I/Os to write the tree back to disk. Given a total of $N_I = \frac{f_e^{d_e}-1}{f_e-1}$ nodes

in the tree and $N_i = \frac{f_k^{d_k}-1}{f_k-1}$ nodes per embedded tree, the cost is:

$$\mathcal{C}_c^e = N_D \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + N_I \cdot N_i \cdot \mathcal{C}_{\mathcal{H}_{f_k|h|}} + \mathcal{C}_{\mathcal{S}_{|h|}} + \frac{\mathcal{C}_s^e}{P} \cdot \mathcal{C}_{IO}. \tag{4.15}$$

It should be mentioned here that the cost for constructing the EMB^--tree is exactly the same, since in order to find the hash values for the index entries of the trees one needs to instantiate all embedded trees. The cost of the EMB^*-tree is somewhat smaller than (4.15), due to the smaller number of nodes in the embedded trees.

$\mathcal{VO}$ construction cost: The $\mathcal{VO}$ construction cost is dominated by the total I/O for locating and reading all the nodes containing the query results. Similarly to the MB-tree case:

$$\mathcal{C}_q^e = [(d_e - d_q + 1) + 2(d_q - 2) + N_Q + \frac{N_R \cdot |r|}{P}] \cdot \mathcal{C}_{IO}, \tag{4.16}$$

where d_q is the height of the query sub-tree and $N_Q = \frac{N_R}{f_e}$ is the number of leaf pages to be accessed. Since the embedded trees are loaded with each node, the querying computation cost associated with finding the needed hash values is expected to be dominated by the cost of loading the node in memory, and hence it is omitted. It should be restated here that for the EMB^*-tree the expected $\mathcal{VO}$ construction cost will be smaller, since not all embedded tree searches will reach the leaf level of the structure.

Authentication cost: The embedded trees work exactly like MB-trees for point queries. Hence, each embedded tree returns $(f_k - 1)d_k$ hashes. Similarly to the MB-tree the total size of the $\mathcal{VO}$ is:

$$|\mathcal{VO}|^e = N_R \cdot |r| + |s| + \sum_{0}^{d_q-2} 2|\mathcal{VO}|^m + |\mathcal{VO}|^m + \sum_{d_q}^{d_m-1} (f_k - 1)d_k|h|, \tag{4.17}$$

where $|\mathcal{VO}|^m$ is the cost of a range query on the embedded trees of the boundary nodes contained in the query sub-tree given by equation (4.11), with a query range that covers all pointers to children that cover the query result-set.

The verification cost is:

$$\mathcal{C}_v^e = N_R \cdot \mathcal{C}_{\mathcal{H}_{|r|}} + \sum_{i=0}^{d_q-1} f_e^i \cdot \mathcal{C}_k + (d_e - d_q) \cdot \mathcal{C}_k + \mathcal{C}_{\mathcal{V}_{|h|}}, \tag{4.18}$$

where $\mathcal{C}_k = N_i \cdot \mathcal{C}_{\mathcal{H}_{f_k|h|}}$ is the cost for constructing the concatenated hash of each node using the embedded tree.

For $f_k = 2$ the authentication cost becomes equal to a Merkle hash tree, which has the minimal $\mathcal{VO}$ size but higher verification time. For $f_k \geq f_e$ the embedded tree consists of only one node which can actually be discarded, hence the authentication cost becomes equal to that of an MB-tree, which has larger $\mathcal{VO}$ size but smaller verification cost. Notice that, as f_k becomes smaller, f_e becomes smaller as well. This has an impact on $\mathcal{VO}$ construction cost and size, since with smaller fanout the height of the EMB-tree increases. Nevertheless, since there is only a logarithmic dependence on f_e versus a linear dependence on f_k, it is expected that with smaller f_k the authentication related operations will become faster.

There are two possible ways of utilizing our index structures to authenticate multi-dimensional selection queries. One option is to build one authenticated index structure for each selection attribute, and authenticate all tuples satisfying each predicate separately, filtering the results at the client side. Another option is to use an authenticated R-tree structure, a variant of which will be discussed in Section 4.3.2.

4.2.2 The Dynamic Case

In this section we analyze the performance of all approaches given dynamic updates between the owner and the servers. In particular, for simplicity, we assume that only insertions and deletions can occur to the database. The performance of updates in the worst case can be considered as the cost of a deletion followed by an insertion. There are two contributing factors for the update cost, computation cost such as creating new signatures and computing hashes, and I/O cost.

Aggregated Signatures with B$^+$-trees

Suppose that a single database record r_i is inserted in or deleted from the database. Assuming that in the sorted order of attribute A the left neighbor of r_i is r_{i-1} and the right neighbor is r_{i+1}, for an insertion the owner has to compute signatures $\mathcal{S}(r_{i-1}|r_i)$ and $\mathcal{S}(r_i|r_{i+1})$, and for a deletion $\mathcal{S}(r_{i-1}|r_{i+1})$. For k consecutive updates in the best case a total of $k+2$ signature computations are required for insertions and 1 for deletions if the deleted tuples are consecutive. In the worst case a total of $2k$ signature computations are needed for insertions and k for deletions, if no two tuples are consecutive. Given k updates, suppose the expected number of signatures to be computed is represented by $E\{k\}(k \leq E\{k\} \leq 2k)$. Then the additional I/O incurred is equal to $\frac{E\{k\}\cdot|s|}{P}$, excluding the I/Os incurred for updating the B^+-tree structure. Since the cost of signature computations is larger than even the I/O cost of random disk accesses, a large number of updates is expected to have a very expensive updating cost. Our experimental evaluation verifies this claim. The total update cost for the ASB-tree is:

$$\mathcal{C}_u^a = E\{k\} \cdot \mathcal{C}_s + \frac{E\{k\} \cdot |s|}{P} \cdot \mathcal{C}_{IO}. \tag{4.19}$$

The Merkle B-tree

The MB-tree can support efficient updates since only hash values are stored for the records in the tree and, first, hashing is orders of magnitude faster then signing, second, for each tuple only the path from the affected leaf to the root need to be updated. Hence, the cost of updating a single record is dominated by the cost of I/Os. Assuming that no reorganization to the tree occurs the cost of an insertion is $\mathcal{C}_u^m = \mathcal{H}_{|r|} + d_m(\mathcal{H}_{f_m|h|} + \mathcal{C}_{IO}) + \mathcal{S}_{|h|}$.

In realistic scenarios though one expects that a large number of updates will occur at the same time. In other cases the owner may decide to do a delayed batch processing of updates as soon as enough changes to the database have occurred. The naive approach for handling batch updates would be to do all updates to the MB-tree one by one and update the path from the leaves to the root once per update. Nevertheless, in case that a large

number of updates affect a similar set of nodes (e.g., the same leaf) a per tuple updating policy performs an unnecessary number of hash function computations on the predecessor path. In such cases, the computation cost can be reduced significantly by recomputing the hashes of all affected nodes only once, after all the updates have been performed on the tree. A similar analysis holds for the incurred I/O as well.

Clearly, the total update cost for the per tuple update approach for k insertions is $k \cdot \mathcal{C}_u^m$ which is linear to the number of affected nodes $k \cdot d_m$. The expected cost of k updates using batch processing can be computed as follows. Given k updates to the MB-tree, assuming that all tuples are updated uniformly at random and using a standard balls and bins argument, the probability that leaf node X has been affected at least once is $P(X) = 1 - (1 - \frac{1}{f_m^{d_m-1}})^k$ and the expected number of leaf nodes that have been affected is $f_m^{d_m-1} \cdot P(X)$. Using the same argument, the expected number of nodes at level i (where $i = 1$ is the leaf level and $1 \leq i \leq d_m$) is $f_m^{d_m-i} \cdot P_i(X)$, where $P_i(X) = [1 - (1 - \frac{1}{f_m^{d_m-i}})^k]$. Hence, for a batch of k updates the total expected number of affected nodes is:

$$E\{X\} = \sum_{i=0}^{d_m-1} f_m^i [1 - (1 - \frac{1}{f_m^i})^k]. \tag{4.20}$$

Hence, the expected MB-tree update cost for batch updates is

$$\mathcal{C}_u^m = k \cdot \mathcal{H}_{|r|} + E\{X\} \cdot (\mathcal{H}_{f_m|h|} + \mathcal{C}_{IO}) + \mathcal{S}_{|h|}. \tag{4.21}$$

We can find the closed form for $E\{X\}$ as follows:

$$\begin{aligned}
& \textstyle\sum_{i=0}^{d_m-1} f_m^i (1 - (\frac{f_m^i - 1}{f_m^i})^k) \\
= \; & \textstyle\sum_{i=0}^{d_m-1} f_m^i (1 - (1 - \frac{1}{f_m^i})^k) \\
= \; & \textstyle\sum_{i=0}^{d_m-1} f_m^i [1 - \sum_{x=0}^{k} \binom{k}{x} (-\frac{1}{f_m^i})^x] \\
= \; & \textstyle\sum_{i=0}^{d_m-1} f_m^i - \sum_{i=0}^{d_m-1} \sum_{x=0}^{k} \binom{k}{x} (-1)^x (\frac{1}{f_m^i})^{x-1} \\
= \; & \textstyle k d_m - \sum_{x=2}^{k} \binom{k}{x} (-1)^x \sum_{i=0}^{d_m-1} (\frac{1}{f_m^i})^{x-1} \\
= \; & \textstyle k d_m - \sum_{x=2}^{k} \binom{k}{x} (-1)^x \frac{1 - (\frac{1}{f_m^{d_m}})^{x-1}}{1 - (\frac{1}{f_m})^{x-1}}
\end{aligned}$$

The second term quantifies the cost decrease afforded by the batch update operation, when compared to the per update cost. For non-uniform updates to the database, the batch updating technique is expected to work well in practice given that in real settings updates exhibit a certain degree of locality. In such cases one can still derive a similar cost analysis by modelling the distribution of updates.

The Embedded MB-tree

The analysis for the EMB-tree is similar to the one for MB-trees. The update cost for per tuple updates is equal to $k \cdot \mathcal{C}_u^e$, where $\mathcal{C}_u^e = \mathcal{H}_{|r|} + d_e \log_{f_k} f_e \cdot (\mathcal{H}_{f_k|h|} + \mathcal{C}_{IO}) + \mathcal{S}_{|h|}$, once again assuming that no reorganizations to the tree occur. Similarly to the MB-tree case the expected cost for batch updates is equal to:

$$\mathcal{C}_u^e = k \cdot \mathcal{H}_{|r|} + E\{X\} \cdot \log_{f_k} f_e \cdot (\mathcal{H}_{f_k|h|} + \mathcal{C}_{IO}) + \mathcal{S}_{|h|}. \tag{4.22}$$

Discussion

For the ASB-tree, the communication cost for updates between owner and servers is bounded by $E\{K\}|s|$, and there is no possible way to reduce this cost as only the owner can compute signatures. However, for the hash based index structures, there are a number of options that can be used for transmitting the updates to the server. The first option is for the owner to transmit only a delta table with the updated nodes of the MB-tree (or EMB-tree) plus the signed root. The second option is to transmit only the signed root and the updates themselves and let the servers redo the necessary computations on the tree. The first approach minimizes the computation cost on the servers but increases the communication cost, while the second approach has the opposite effect.

4.3 Authenticated Index Structures for Aggregation Queries

Any technique for authenticating selection queries can be used as a straightforward but very inefficient solution for authenticating aggregation queries. The server simply answers the aggregation query Q as a selection query and returns $\mathcal{SAT}(Q)$ along with the $\mathcal{VO}$.

The client first verifies $\mathcal{SAT}(Q)$, then computes the aggregation locally. This approach is inefficient because the communication and verification costs are linear to $|\mathcal{SAT}(Q)|$ (e.g., if the query is a SELECT * statement the cost might be prohibitive), and also because the cost of answering multi-dimensional aggregation queries is extremely high since all tuples satisfying each selection predicate independently have to be send to the client before filtering can be performed. It is thus desirable to design a solution that has communication and verification costs sub-linear to $|\mathcal{SAT}(Q)|$ and, in addition, that can support multi-dimensional aggregation queries efficiently.

4.3.1 The Static Case

The APS-tree: Authenticated Prefix Sums

Consider a one dimensional SUM query,

$$Q = \text{SELECT SUM}(A_q) \text{ FROM } \mathbf{T} \text{ WHERE } a_1 \leq S_1 \leq b_1.$$

The prefix sum array $PS[x]$ for attribute A_q with respect to selection attribute S_1 stores for every x the sum of values A_q for all tuples with $S_1 \leq x$. If we assume that we have pre-computed the prefix sums array PS for attribute A_q with respect to S_1, the answer to Q is equal to $PS[b_1] - PS[a_1 - 1]$. Prefix sum arrays can be used for answering multi-dimensional queries also. Assume for simplicity discrete domains $D_j = [0, M_j)$, [4] and a query

$$\begin{aligned} Q = \text{SELECT SUM}(A_q) &\text{ FROM } \mathbf{T} \\ &\text{WHERE } a_1 \leq S_1 \leq b_1 \text{ AND } \ldots \text{ AND } a_d \leq S_d \leq b_d. \end{aligned}$$

Each tuple in the database can be viewed as a point in a d-dimensional space $D_1 \times \cdots \times D_d$. The d-dimensional space can be represented as a $D_1 \times \cdots \times D_d$ array C and the points (tuples) are mapped to elements in C. Every element $C[i_1, \ldots, i_d]$ of the array contains

[4] For continuous or categorical domains existing values are ordered and assigned distinct identifiers. Next section presents structures that do not require the domains to be discrete.

those A_q values that appear in tuples whose $S_1, \ldots, S_d$ attributes have values corresponding to coordinates $i_1, \ldots, i_d$. If there are no such tuples for element $C[i_1, \ldots, i_d]$, its value is set to zero. The prefix sum array PS of C has the same structure as C and in every coordinate it stores $\forall x_j \in [0, M_j), j \in [1, d]$:

$$PS[x_1, \ldots, x_d] = \sum_{i_1=0}^{x_1} \ldots \sum_{i_d=0}^{x_d} C[i_1, i_2, \ldots, i_d],$$

It has been shown in [Ho et al., 1997] that answering any d dimensional range sum query Q defined as above using PS requires accessing 2^d elements as follows. For all $j \in [1, d]$, let $I(j) = 1$ if $x_j = b_j$ and $I(j) = -1$ if $x_j = a_j - 1$, and $PS[x_1, \ldots, x_d] = 0$ if $x_j = -1$, then:

$$\mathcal{ANS}(Q) = \sum_{\forall x_j \in \{a_j - 1, b_j\}} \{PS[x_1, \ldots, x_d] \cdot \prod_{i=1}^{d} I(i)\}. \tag{4.23}$$

Having computed PS for the aggregation attribute A_q, any aggregation query with the same form as Q can be answered efficiently. Furthermore, authenticating the answers becomes equivalent to authenticating the 2^d prefix sums required by Equation (4.23). However, as we discuss next, we need to authenticate **both** their values and their locations in the array. To authenticate the elements of PS, we convert PS into a one-dimensional array PS_{1d}, where element $PS[i_1, \ldots, i_d]$ corresponds to element $PS_{1d}[k], k = \sum_{j=1}^{d-1}(i_j \prod_{n=j+1}^{d} M_n) + i_d$, and build an f-way Merkle tree on top of PS_{1d}, as described in Section 2.4. We call this structure the authenticated prefix sums tree (APS-tree).

Suppose that a single element $PS_{1d}[k]$ needs to be authenticated. Traversing the tree in order to find the k-th element requires computing the correct path from the root to the leaf containing the k-th entry, and can happen efficiently by using the following tree encoding scheme. Let h be the height of the tree (with 0 being the height of the root) and f its fanout. Every node is encoded using a unique label, which is an integer in a base-f number system. The root node has no label. The 1st level nodes have base-f

representations $1, 2, \ldots f$, from left to right respectively.[5] The 2nd level nodes have labels $11, \ldots, 1f, 21, \ldots, 2f, \ldots, f1, \ldots, ff$, and so on all the way to the leaves. An example is shown in Figure 4·5 with $f = 2$. Straightforwardly, a leaf entry with PS_{1d} offset k is converted into a base-f number $\lambda_1 \cdots \lambda_h$ with h digits (each digit, in our notation, ranging from 1 to f, and computed as $\lambda_i = 1 + \lfloor k/f^{h-i} \rfloor \bmod f$). Since the tree is full, element k lies in the k/f leaf node, and this leaf node lies in the k/f^2 index node one level up, and so on. Retrieving $PS_{1d}[k]$ is now possible by computing all prefixes of the label of k and following the corresponding nodes.

Given a query Q, the server computes all the 2^d prefix sums needed to answer the query and constructs a $\mathcal{VO}$ that contains all the hash values needed for authenticating those entries. In addition, the $\mathcal{VO}$ should contain the encoding of the path followed for every prefix sum (i.e., the label of every entry). The client needs the path encoding in order to be able to reconstruct the hash values of index nodes correctly.

Similarly to the authenticated index structures for range queries it can be shown that any change to the structure of the APS-tree or the $\mathcal{VO}$ will cause the authentication procedure to fail. In addition, the fact that the encoding path of an element must be included in the $\mathcal{VO}$ for successful verification ensures the following:

Lemma 4.3.1. *Given $\prod_{i=1}^{d} M_i$ ordered elements in PS_{1d}, the APS-tree can authenticate both the value and the position of the k-th element for $k \in [1, \prod_{i=1}^{d} M_i]$.*

In order for the client to be able to authenticate a query correctly the following information is needed: 1. The signed root of the APS-tree, 2. The domain size of the selection attributes, and 3. The fanout of the APS-tree. [6] Using the hashes and the path encodings first the client verifies each prefix sum by computing the hash of the root once for every path and comparing it against the singed root. During this step, the client also infers the position k of every prefix sum in PS_{1d} based on its encoding and the fanout f. Next, using the query ranges $[a_i, b_i]$ and the domain sizes, the client maps each element's position k

[5] We use numbers from 1 to f instead of from 0 to $f-1$ to simplify notation for the purposes of exposition.

[6] The domain sizes and the fanout of the tree are static information and could be authenticated directly by the owner at system initialization.

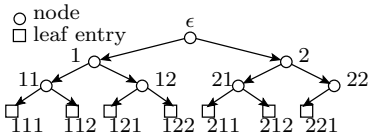


Figure 4·5: The tree encoding scheme.

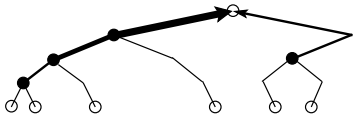


Figure 4·6: Merging the overlapping paths in the APS-tree. At every black node the remaining hashes can be inserted in the $\mathcal{VO}$ only once for all verification paths towards the root.

back to the original coordinate space of the d-dimensional prefix sums array. Finally, if all the elements are verified correctly, the client confirms that the correct prefix sum have been returned and computes the answer to the query using Equation (4.23).

Correctness and Completeness: Based on Lemma 4.3.1 and Equation (4.23), we can claim that the APS-tree guarantees both completeness and correctness of the provided results.

Optimizations: A naive $\mathcal{VO}$ construction algorithm would return an individual $\mathcal{VO}$ for each of the prefix sum entries returned. Since the authentication paths of these values may share a significant number of common edges (as shown in Figure 4·6), a substantial improvement in the communication and authentication costs can be achieved by constructing a common $\mathcal{VO}$ using only one tree traversal. A sketch of this process is shown in Algorithm 1. In this algorithm, $\mathcal{SAT}(Q)$ refers to the set of prefix sums required to answer the aggregation query, which is equivalent to the actual set of tuples satisfying the query, according to Equation (4.23).

Let $k^1, \ldots, k^n$ be the indices of the PS_{1d} values that need to be authenticated. The single-pass construction algorithm first computes the base-f labels of indices $k^1, \ldots, k^n$,

Algorithm 1: APSQuery(Query Q; APS-tree T; Stack $\mathcal{VO}$; Stack $\mathcal{SAT}$)

```
1  Let node N and N.k the key, N.α the aggregate value, N.η the hash
2  VO = ∅, SAT = ∅, h = T.height
3  VO.push(h), VO.push(domain sizes)
4  for k^x = i_1, ..., i_d ∈ {a_1 − 1, b_1}, ..., {a_d − 1, b_d} do
5      Compute matrix K = λ_1^1 ··· λ_h^n for keys k^1, ..., k^n
6  Compute G_1 = {11, ..., 1x} using K
   // set G_1 contains x groups named 11, ..., 1x
7  for S ∈ G_1 do
8      Recurse(root, 2, S, K)
9  Recurse(Node N, Level l, Set S, Matrix K):
10 begin
11     Compute G = {S1, ..., Sy} using K
       // group names will become {111, ..., 11z}, {1111, ..., 111w}, and so on
12     VO.push(l), VO.push(N.children - |G|)
13     for λ_l ≡ S' ∈ G do
           // for λ_ls corresponding to each S' in G
14         VO.push(λ_l)
15         if l = h then SAT.push(N[λ_l].k)
           ;                    // value λ_h^x is the offset of key k^x in the leaf
16     for 1 ≤ i ≤ N.children do
17         if i ≠ λ_l, ∀λ_l ∈ G then VO.push(N[i].η)
18     if l < h then
19         for λ_l ≡ S' ∈ G do  Recurse(N[λ_l], l + 1, S')
20 end
```

and conceptually defines a $n \times h$ matrix K as follows:

$$K = \begin{bmatrix} \lambda_1^1 & \lambda_2^1 & \dots & \lambda_h^1 \\ \lambda_1^2 & \lambda_2^2 & \dots & \lambda_h^2 \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_1^n & \lambda_2^n & \dots & \lambda_h^n \end{bmatrix}$$

The paths that need to be followed at every level of the tree for constructing the $\mathcal{VO}$ can be identified by calculating the longest common prefixes of the labels contained in every column of K. These prefixes denote the longest common paths that can be followed for a combination of prefix sum entries, before the paths of those entries split. The verification

procedure at the client follows a similar reasoning, but in a bottom-up fashion.

Extending the APS-tree to support multiple aggregate attributes is achieved by associating a set of aggregate values and hash values with every entry of $PS_{1d}[k]$ at the leaf level of the APS-tree. Every value/hash pair corresponds to an aggregate attribute one wishes to be able to authenticate. Answering multi-aggregate queries using this structure requires only one traversal of the tree. (To reduce storage costs at the expense of larger $\mathcal{VO}$ sizes, a single hash value for all aggregate attributes can be computed; in this case, for the client to authenticate correctly, the $\mathcal{VO}$ must include all aggregate attributes, irrespective of which attributes the query refers to.) Finally, the APS-tree can be used to authenticate COUNT and AVG as well, as COUNT is a special case of SUM and AVG is authenticated as SUM/COUNT. Authentication of MIN and MAX is not possible with the APS-tree.

Cost Analysis

It is easy to see that the APS-tree has constant communication and verification cost regardless of $|\mathcal{SAT}(Q)|$. However, it has high storage overhead for sparse databases and extremely high update cost. For example, updating a tuple that belongs in $PS[0, \ldots, 0]$ (having the smallest values in all domains), will necessitate an update to the whole APS-tree. A detailed cost analysis is presented next.

Query cost: The query cost can be broken up into communication and verification costs. The communication cost depends on the size of the $\mathcal{VO}$ and $\mathcal{SAT}$:

$$\mathcal{C}_{communication} \leq 2^d(f-1)\lceil \log_f N \rceil |\mathcal{H}| + 2^d|I|, \tag{4.24}$$

where N $(= M_1 \times \ldots \times M_d)$ is the total size of the PS_{1d} array, $|\mathcal{H}|$ is the size of a hash value, $|I|$ the size of an integer value (all in bytes).

The verification cost at the client in the worst case is:

$$\mathcal{C}_{verification} \leq 2^d \lceil \log_f N \rceil \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{V}}, \tag{4.25}$$

where $\mathcal{C}_{\mathcal{H}}$ and $\mathcal{C}_{\mathcal{V}}$ denote the cost of one hashing operation and the cost of one verification operation respectively.

Storage cost: The size of an APS-tree is equal to:

$$\mathcal{C}_{storage} = \sum_{l=0}^{\lceil log_f N \rceil} f^l(|\mathcal{H}| + |I|) + N|I|, \tag{4.26}$$

including one hash and one pointer per tree entry. The APS-tree is storage-expensive if the original d-dimensional array C is sparse (the database is sparse w.r.t the full domain size), i.e., when only a few coordinates contain database tuples.

Update cost: Updating a single element of the prefix sums array requires updating the values of all other elements that dominate this entry. Assume that element $PS[i_1, \ldots, i_d]$ is updated. Then, elements $PS[x_1, \ldots, x_d]$ for $i_j < x_j < M_j, 1 \leq j \leq d$ also need to be updated, for a total of $\prod_{j=1}^{d}(M_j - i_j)$ values. Hence, the cost of updating the APS-tree is:

$$\mathcal{C}_{update} = \prod_{j=1}^{d}(M_j - i_j)\lceil log_f N \rceil \cdot \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{S}}, \tag{4.27}$$

where $\mathcal{C}_{\mathcal{S}}$ denotes the cost of a signing operation.

4.3.2 The Dynamic Case

The APS-tree is a good solution for non-sparse, static environments and even though it features constant query cost, it has its limitations. More specifically, it cannot handle MIN/MAX aggregates, has considerable storage overhead for sparse database, and its update cost is very high. This section presents structures that overcome these limitations.

One dimensional Queries: Authenticated Aggregation B-tree

Consider an one-dimensional aggregation query that contains only one selection predicate with continuous or discrete domain D_1, where the *distinct number* of values of S_1 in $\mathbf{T}$ is $N \leq M_1$. An Authenticated Aggregation B-tree (AAB-tree) is an extended B^+-tree

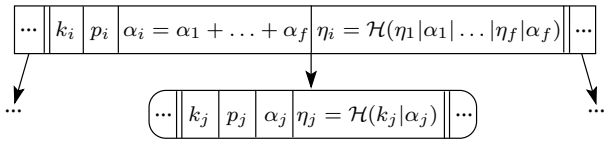


Figure 4·7: The AAB-tree. On the top is an index node. On the bottom is a leaf node. k_i is a key, α_i the aggregate value, p_i the pointer, and η_i the hash value associated with the entry.

structure of fanout f with key attribute S_1, bulk-loaded bottom-up on the base table tuples. The AAB-tree nodes are extended with one hash value and one aggregate value per entry. The exact structure of a leaf and an index node is shown in Figure 4·7. Each leaf entry corresponds to one database tuple t with key $k = t.S_1$, aggregate value $\alpha = t.A_q$, and an associated hash value $\eta = \mathcal{H}(k|\alpha)$. If tuples with duplicate keys exist, then the tree stores only one entry for that key and aggregates all A_q values in α. Hence the AAB-tree has exactly N data entries. Index entries have keys k computed in the same way as in the normal B$^+$-tree and each key is associated with an aggregate value $\alpha = \alpha_1 \otimes \ldots \otimes \alpha_f$ (which is the aggregation of the aggregate values of its children), and a hash value $\mathcal{H}(\eta_1|\alpha_1|\ldots|\eta_f|\alpha_f)$, which is a concatenation of both the hash values and the aggregate values of the children.

To locate an entry with key k, a point B$^+$-tree query is issued. Authenticating this entry is done in a Merkle tree fashion. The only difference is that the $\mathcal{VO}$ includes both the hash values η and aggregate values α associated with every index entry, and the key values k associated with every leaf entry. In addition, auxiliary information is stored in the $\mathcal{VO}$, so that the client can find the right location of each hash value during the verification phase. For ease of exposition, we use the same tree encoding scheme as in the previous section (see Figure 4·8). The only difference is that in an AAB-tree any node could be incomplete and contain fewer than f entries. However, the labelling scheme is imposed on a conceptually complete tree. As the auxiliary information enables the client at each level of the tree to place the computed hash values correctly, the AAB-tree can ensure that:

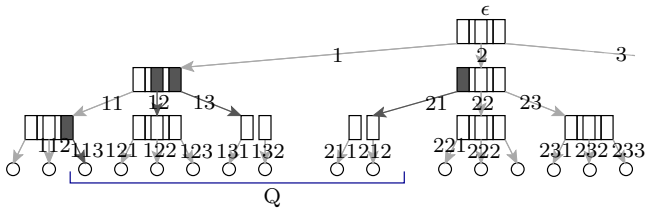


Figure 4·8: Labelling scheme and the $\mathcal{MCS}$ entries.

Lemma 4.3.2. *The AAB-tree can authenticate both the value associated with the aggregate attribute and the label of any entry in the tree, including entries at the index nodes.*

Next, we present a method for authenticating aggregation queries efficiently using the AAB-tree. The basic idea is that the aggregate information at the index nodes of the tree can be used to answer and authenticate range queries without having to traverse the tree all the way to the leaves. The following statements apply to aggregation trees in general, with or without authentication. As we shall see shortly, however, authentication interacts very nicely with the aggregation-related structures.

Definition 4.3.3. *The Label Cover $\mathcal{LC}$ of entry $\lambda = \lambda_1 \cdots \lambda_h$ is the range of labels of all* data entries *(i.e., leaf level entries) that have λ as an ancestor. The label cover of a data entry is the label of the entry itself.*

Given a label λ, the range of labels in its $\mathcal{LC}$ can be computed by right padding it with enough 1s to get an h-digit number for the lower bound, and enough fs to get an h-digit number for the upper bound. For example, the $\mathcal{LC}$ of $\lambda = 12$ in Figure 4·8 is $\{121, \ldots, 123\}$.

Definition 4.3.4. *The Minimum Covering Set $\mathcal{MCS}$ of the* data entries *in query range Q is the set of entries whose $\mathcal{LC}$s are: 1. Disjoint; 2. Their union covers S completely; 3. Their union covers only entries in S and no more.*

Let λ^-, λ^+ be the entries corresponding to the lower and upper bound entries of $\mathcal{ANS}(Q)$ (e.g., $\lambda^- = 113$ and $\lambda^+ = 212$ in the example of Figure 4·8). Set $\mathcal{MCS}(Q)$ can be computed by traversing the tree top-down and inserting in the $\mathcal{MCS}$ all entries whose $\mathcal{LC}$ is completely contained in $[\lambda^-, \lambda^+]$ (and whose ancestors have not been included

already in $\mathcal{MCS}$). An entry with $\mathcal{LC}$ that intersects with $[\lambda^-, \lambda^+]$ is followed to the next level. An entry with $\mathcal{LC}$ that does not intersect with $[\lambda^-, \lambda^+]$ is ignored. An example is shown in Figure 4·8. The range of labels $[\lambda^-, \lambda^+]$ of Q is $[113, 212]$. The $\mathcal{MCS}(Q)$ consists of the entries with labels $\{113, 12, 13, 21\}$. It can be shown that:

Proposition 4.3.5.

$$\mathcal{ANS}(Q) = \sum_{N \in \mathcal{MCS}(Q)} N.\alpha, \tag{4.28}$$

$$\bigcup_{N \in \mathcal{MCS}(Q)} \mathcal{LC}(N) = [\lambda^-, \lambda^+], \tag{4.29}$$

$$\mathcal{LC}(M) \cap \mathcal{LC}(N) = \varnothing, \forall N, M \in \mathcal{MCS}(Q), M \neq N. \tag{4.30}$$

Based on Proposition 4.3.5, the authentication of Q can now be converted to the problem of authenticating $\mathcal{MCS}(Q)$. Next, we discuss the algorithm for retrieving $\mathcal{MCS}(Q)$ and the sibling set $\mathcal{SIB}$ needed to verify the former, in one pass of the tree. The server first identifies the boundary labels λ^-, λ^+ of the query range using two point B$^+$-tree queries (note that these labels might correspond to keys with values $a \leq k^-$ and $k^+ \leq b$). Starting from the root, the server follows the following modified algorithm for constructing the $\mathcal{MCS}$, processing entries using a pre-order traversal of the tree. When a node N is visited, the algorithm compares $\mathcal{LC}(N)$ with $[\lambda^-, \lambda^+]$: if $\mathcal{LC}(N) \in [\lambda^-, \lambda^+]$, then the node is added to the $\mathcal{MCS}$. If $\mathcal{LC} \cap [\lambda^-, \lambda^+] \neq \varnothing$, then the node's children are visited recursively. If $\mathcal{LC}(N) \cap [\lambda^-, \lambda^+] = \varnothing$ then the node's hash value (or key value for leaf nodes) and aggregate value is added to the $\mathcal{SIB}$. In the previous example, the hash values (or key values for leaf entries) and the aggregate values of entries $\{111, 112, 22, 23, 3\}$ are included in $\mathcal{SIB}$.

The $\mathcal{VO}$ for the aggregation query contains sets $\mathcal{MCS}$ and $\mathcal{SIB}$, as well as the label of every entry contained therein; this will enable the client to infer an entry's correct position in the tree and reconstruct the hash value of its ancestor. Finally, to ensure completeness, the server also includes verification information for the two boundary data entries that lie exactly to the left of λ^- and to the right of λ^+. Denote these entries by λ_l^-, λ_r^+ respectively[7].

[7]For the left-most and right-most entries in the tree, dummy records are used.

Before discussing the verification algorithm at the client side the following definitions are necessary:

Definition 4.3.6. *Two entries (or their labels) are* neighbors *if and only if: 1. They are at the same level of the tree and no other entry at that level exists in-between, or 2. Their $\mathcal{LC}$s are disjoint and the left-most and right-most labels in the $\mathcal{LC}$ of the right and left entry respectively are neighbors.*

For example, in figure 4·8 entries $11, 12$ as well as $113, 12$ are neighbors. An interesting observation is that:

Lemma 4.3.7. *All consecutive $\mathcal{MCS}$ entries (in increasing order of labels) are neighbors in the tree.*

Proof. Suppose that two *consecutive* $\mathcal{MCS}$ entries M, N are not neighbors. Let x be the largest data value under M, and z the smallest data value under N. Since M and N are not neighbors, by the definition of $\mathcal{LC}$ there exists a value y and a node P s.t. $x < y < z$ and P is a neighbor of M. Clearly, if such y and P exist, y is an answer to the query and P (or a descendant of P that is a neighbor of M) must be a member of $\mathcal{MCS}$. This is a contradiction, since M and N are consecutive in label order. Hence, assuming that the $\mathcal{MCS}$ has been constructed correctly such y cannot exist and M, N are neighbors.

□

The authentication at the client side proceeds as follows. First the boundary entries $\{\lambda^-, \lambda_l^-, \lambda^+, \lambda_r^+\}$ are authenticated. After successful authentication, the client checks that $k_l^- < a \leq k^-$ and $k^+ \leq b < k_r^+$ and that the entries $\{k_l^-, k^-\}$ (similarly for $\{k^+, k_r^+\}$) are neighbors. The second step is to verify the correctness of all entries in the $\mathcal{MCS}$ and that consecutive entries are neighbors (by using their labels). If this step is successful, the $\mathcal{MCS}$ has been constructed correctly and no entries are missing according to Lemma 4.3.7. The last step is to infer the $\mathcal{LC}$s of all $\mathcal{MCS}$ entries using their labels and confirm Proposition 4.3.5. On success, the client computes the final aggregate result. On failure, the client rejects the answer.

The complete query and $\mathcal{VO}$ construction algorithm is presented in Algorithm 2 and the algorithm for client side verification is presented in Algorithm 3.

Algorithm 2: AABQuery(Query Q; AAB-tree T; Stack $\mathcal{VO}$)

```
1  Let node N and N.k the key, N.α the aggregate value, N.η the hash
2  Compute [λ⁻, λ⁺] from Q
3  Recurse(T.root, VO, [λ⁻, λ⁺])
4  Push information for verifying λ_l⁻, λ_r⁺ into VO
5
6  Recurse(Node N, Stack VO, Range R):
7  begin
8      VO.push(∗); VO.push(N.children)
9      for N.children ≥ i ≥ 1 do
10         if LC(N[i]) ∈ R then
11             if N is a leaf then
12                 VO.push(N[i].k);
13             else VO.push(N[i].η)
14         else if LC(N[i]) ∩R ≠ ∅ then
15             Recurse(N[i], VO, R)
16         else VO.push(N[i].η);
17         VO.push(N[i].α)
18 end
```

The AAB-tree can be used for authenticating one-dimensional aggregate queries in a dynamic setting since the owner can easily issue deletions, insertions and updates to the tree, which handles them similarly to a normal B^+-tree. In addition, extending the AAB-tree for multiple aggregate attributes A_q can happen similarly to the APS-tree. Straightforwardly, the AAB-tree can also support COUNT, AVG, MIN and MAX.

Correctness and Completeness: Based on lemmata 4.3.2, 4.3.7, and proposition 4.3.5, the AAB-tree ensures both correctness and completeness of the query results.

Optimizations: A potential optimization for reducing the $\mathcal{VO}$ size of a given query Q, is to authenticate the complement of Q. Define the following:

Definition 4.3.8. *The* Least Covering Ancestors $\mathcal{LCA}$ *of the data entries in the range of Q are the two entries whose $\mathcal{LC}$s are: 1. Disjoint, 2. Completely cover the data entries in $[\lambda^-, \lambda^+]$, 3. The union of their $\mathcal{LC}$s contains the minimum possible number of data entries.*

Denote with R the range of data entries covered by $\mathcal{LCA}(Q)$. In Figure 4·8, $\mathcal{LCA}(Q)$

Algorithm 3: AABAuthenticate(Query Q; Stack $\mathcal{VO}$)

```
1  Retrieve and verify λ⁻, λ⁺ from VO
2  MCS = ∅
3  η = Recurse(VO, MCS)
4  Remove entries from MCS according to [λ⁻, λ⁺]
5  Verify neighbor integrity of MCS or Reject
6  Verify η or Reject
7
8  Recurse(Stack VO, Stack MCS):
9  begin
10     c = VO.pop()
11     η = ∅
12     for 1 ≤ i ≤ c do
13         e = VO.pop()
14         if e = * then η = η| Recurse(VO, MCS) α = VO.pop()
15         η = η|e|α
16         MCS.push(e)
17     Return H(η)
18 end
```

contains entries 1 and 21 and range R data entries $[111, 212]$. Depending on the size of $\mathcal{MCS}(Q)$, it might be beneficial to answer the query by authenticating the aggregate of range R, then the aggregate of range $\overline{Q} = R - Q$ (using $\mathcal{MCS}(\overline{Q})$, and subtract the result for the final answer. In the running example, $\mathcal{MCS}(\overline{Q})$ is simply $\{111, 112\}$. It is possible to estimate the size of these sets using statistical information about the average per level utilization of the tree, meaning that the server can make a decision without having to traverse the tree. Furthermore, if the tree is complete, the exact size of these sets can be analytically computed.

Cost Analysis of AAB-Tree

To authenticate any *aggregate value* either in a leaf entry or an index entry, or the *key* of a leaf entry, in the worst case the $\mathcal{VO}$ constructed by the AAB-tree has size:

$$|\mathcal{VO}| \leq \lceil \log_f N \rceil [f(|\mathcal{H}| + 2|I|) + 2|I|], \tag{4.31}$$

The size of the $\mathcal{MCS}$ can be upper bounded as well. For any key range $[a, b]$:

$$|\mathcal{MCS}| \leq 2(f-1)\lceil \log_f (b-a+1) \rceil. \tag{4.32}$$

The subtree containing all entries in range $[a, b]$ has height $\lceil \log_f (b-a+1) \rceil$. In the worst case at every level of the tree the $\mathcal{MCS}$ includes $f-1$ entries for the left sub-range, and $f-1$ for the right sub-range, until the two paths meet.

Query cost: Combining Equations (4.31) and (4.32) the communication cost is bounded by:

$$\mathcal{C}_{communication} \leq 2|\mathcal{VO}| + |\mathcal{MCS}| \cdot |\mathcal{VO}|, \tag{4.33}$$

for the $\mathcal{VO}$s corresponding to the boundary labels, and the $\mathcal{VO}$ for the $\mathcal{MCS}$. The verification at the client, counting hashing and verification operations only, is bounded by:

$$\mathcal{C}_{verification} \leq \quad (|\mathcal{MCS}| + \lceil \log_f \tfrac{N}{b-a+1} \rceil) \cdot \mathcal{C}_{\mathcal{H}} + 2 \log_f N \cdot \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{V}}, \tag{4.34}$$

including the hashes for the nodes containing $\mathcal{MCS}$ entries, the remaining hashes in the path to the root, and the authentication cost of the boundary entries. Only one verification is necessary, since whenever the computed hash value of the root of the tree is rejected the answer is rejected immediately.

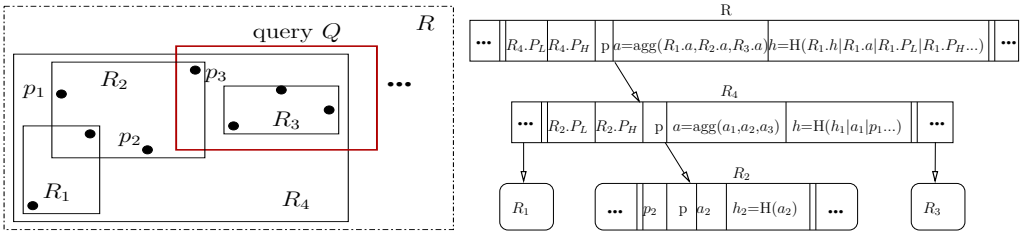
Storage cost: The size of the AAB-tree is:

$$\mathcal{C}_{storage} = \sum_{l=1}^{\lceil log_f N \rceil} f^l (|\mathcal{H}| + 4|I|), \tag{4.35}$$

which includes the hash value, aggregate value, key and one pointer per entry.

Update cost: Updating the AAB-tree is similar to updating a normal B^+-tree with the additional cost of recomputing the hash values and aggregate values when nodes merge or split. The cost is bounded by:

$$\mathcal{C}_{update} \leq 2\lceil log_f N \rceil \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{S}}, \tag{4.36}$$



(a) Data Indexed by AAR-tree. (b) AAR-tree Nodes.

Figure 4·9: AAR-tree.

given the worst case update cost of a B^+-tree.

Multi-dimensional Queries: Authenticated Aggregation R-tree

The AAB-tree can answer only one-dimensional queries. To answer multi-dimensional aggregation queries we extend the Aggregate R-tree (AR-tree)[Jürgens and Lenz, 1999, Lazaridis and Mehrotra, 2001, Tao and Papadias, 2004b] to get the Authenticated Aggregation R-tree (AAR-tree).

Let

$$Q = \text{SELECT SUM}(A_q) \text{ FROM } \mathbf{T}$$
$$\text{WHERE } a_1 \leq S_1 \leq b_1 \text{ AND } \ldots \text{ AND } a_d \leq S_d \leq b_d,$$

be a d-dimensional aggregate query. Similarly to the AAB-tree, the AAR-tree indexes the tuples in $\mathbf{T}$ on attributes $S_i, 1 \leq i \leq d$. Every dimension of the tree corresponds to an attribute S_i, and every node entry is associated with an aggregate value α and a hash value η. The hash value is computed on the concatenation of the node MBRs of the entry's children m_i, aggregate values α_i and hash values η_i ($\eta = \mathcal{H}(\ldots|m_i|\alpha_i|\eta_i|\ldots)$). The structure of an AAR-tree node is shown in Figure 4·9. The MBR of each entry is included in the hash computation in order for the client to be able to authenticate the extent of the tree nodes, which will enable verification of the completeness of the results, as will be seen shortly. Notice that in a d-dimensional space, the MBR m is simply a d-dimensional rectangle represented by two d-dimensional points. The query Q becomes a d-dimensional

query rectangle.

The $\mathcal{MCS}$ set can be defined similarly to the AAB-tree. It consists of the minimum set of nodes whose MBRs totally contain the points covered by the query rectangle. The $\mathcal{VO}$ construction is similar to that of the AAB-tree, and uses the concept of computing the answer by authenticating the $\mathcal{MCS}$ entries. Even though correctness verification for any range query can be achieved simply by authenticating the root of the tree, completeness in the AAR-tree requires extra effort by the client. Specifically, the client needs to check if the MBR associated with each node in the $\mathcal{VO}$ intersects or is contained in the query MBR. If it is contained, it belongs to the $\mathcal{MCS}$. If it intersects, then the client expects to have already encountered a descendant of this node which is either fully contained in the query range or disjoint. This check is possible since the MBRs of all entries are included in the hash computation. The server has to return all MBRs of the entries encountered during the querying phase in order for the client to be able to recompute the hash of the root, and to be able to check for completeness. The AAR-tree can be extended to support multi-aggregate queries similarly to the APS-tree.

Correctness and Completeness: The verification method described above gives an authentication procedure that guarantees correctness and completeness. The proof is a special case of [Martel et al., 2004, Theorem 3], which holds for more general DAG structures.

Cost Analysis

We present the cost analysis for AAR-tree next.

Query cost: Let an AAR-tree indexing N d-dimensional points, with average fanout f and height $h = \log_P(\frac{N}{f})$ (where the level of the root node is zero and the level of the data entries is h). The size of the $\mathcal{VO}$ for authenticating one AAR-tree entry at level l (equivalent to a node at level $l+1$), either a data entry or an index entry, is upper bounded by:

$$|\mathcal{VO}| \leq fl[2d \cdot |I| + |I| + |\mathcal{H}|], \tag{4.37}$$

(assuming that MBRs are represented by $|I|$-byte floating point numbers). The cost is equal to the level of the entry times the amount of information needed for computing the hash of every node on the path from the entry to the root.

The size of the $\mathcal{MCS}$ is clearly related to the relationship between the query range MBR q and the MBRs of the tree nodes. Let m be a node MBR, and $P(m \odot q)$ represent the probability that the query MBR fully contains m. Assuming uniformly distributed data, it has been shown in [Jürgens and Lenz, 1999] that:

$$P(m \odot q) = \begin{cases} \prod_{j=1}^{d}(q_j - m_j) & \text{, if } \forall j : q_j > m_j \\ 0 & \text{, otherwise} \end{cases},$$

where q_j, m_j represent the length of q and m on the j-th dimension. Furthermore, it has been shown in [Kamel and Faloutsos, 1993] that the probability of intersection between q and m is $P(m \oplus q) = \prod_{j=1}^{d}(q_j + m_j)$. Thus, the probability of an intersection but not containment is equal to $P(m \ominus q) = P(m \oplus q) - P(m \odot q)$. Let m_l be the average length of the side of an MBR at the l-th level of the tree. It has been shown by [Theodoridis and Sellis, 1996, Tao and Papadias, 2004a] that $m_l = \min\{(f^{h-l}/N)^{1/d}, 1\}, 0 \leq l \leq h-1$, which enables us to estimate the above probability.

The expected number of nodes at level l is $N_l = \frac{N}{f^{h-l}}$. It can be shown that $J_l = J_{l-1} \cdot P(m \ominus q), 1 \leq l \leq h, J_0 = f \cdot P(m \ominus q)$ is the number of nodes that intersect with query q at level l, given that all its ancestors also intersect with q. The size of the $\mathcal{MCS}$ is upper bounds by $|\mathcal{MCS}| \leq [N_1 + \sum_{l=0}^{h-1} J_l] \cdot P(m \odot q)$. Essentially, we estimate the number of children entries that are contained in the query, for every node that intersects with the query at a given level.

Hence, the communication cost in the worst case can be expressed by the following estimate:

$$\mathcal{C}_{communication} \leq |\mathcal{MCS}| \cdot |\mathcal{VO}|. \tag{4.38}$$

The verification cost is similarly bounded by:

$$\mathcal{C}_{verification} \leq |\mathcal{MCS}| \cdot \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{V}}. \tag{4.39}$$

Storage cost: The storage cost is:

$$\mathcal{C}_{storage} = \sum_{l=0}^{h-1} \frac{N}{f^{h-l}} \cdot f \cdot [|\mathcal{H}| + 2d \cdot |I| + |I| + |I|], \tag{4.40}$$

since we extend every entry with a hash value and an aggregate value, and include the d-dimensional MBRs and one pointer per entry.

Update cost: Updating the AAR-tree is similar to updating an R-tree. Hence:

$$\mathcal{C}_{update} = \log_P(\frac{N}{f}) \cdot \mathcal{C}_{\mathcal{H}} + \mathcal{C}_{\mathcal{S}}. \tag{4.41}$$

4.4 Query Freshness

The dynamic scenarios considered here reveal a third dimension of the query authentication problem that has not been sufficiently explored in previous work, that of *query result freshness*. When the owner updates the database, a malicious or compromised server may still retain an older version of the data. Since the old version was authenticated by the owner already, the client will still accept any query results originating from an old version as authentic, unless the latter is informed by the owner that this is no longer the case. In fact, a malicious server may choose to answer queries using any previous version, and in some scenarios even a combination of older versions of the data. If the client wishes to be assured that queries are answered using the latest data updates, additional work is necessary.

This issue is similar to the problem of ensuring the freshness of signed documents, which has been studied extensively in the context of certificate validation and revocation. There are many solutions which we do not review here. The simplest is to publish a list of revoked signatures, one for every expired version of the database. More sophisticated ones

are: 1. Including the time interval of validity as part of the signed root of the authenticated structures and reissuing the signature after the interval expires, 2. Using hash chains to confirm validity of signatures at frequent intervals [Micali, 1996].

Clearly, all signature freshness techniques impose a cost which is linear to the number of signatures used by any authentication structure. The advantage of all novel Merkle tree based solutions presented here is that they use one signature only — that of the root of the tree — which is sufficient for authenticating the whole database. Straightforwardly, database updates will also require re-issuing only the signature of the root. This is a tremendous improvement compared to the ASB-tree approach which uses multiple signatures. It is unclear how to solve the freshness problem for the ASB-tree without applying signature freshness techniques to each signature individually, which will be prohibitively expensive in practice.

4.5 Extensions

This section extends our discussion to other interesting topics that are related to the query authentication problem.

General Query Types. The proposed authenticated structures can support other query types as well. We briefly discuss here a possible extension of these techniques for join queries. Other query types that can be supported are projections and multi-attribute queries.

Assume that we would like to provide authenticated results for join queries such as $R \bowtie_{A_i=A_j} S$, where $A_i \in R$ and $A_j \in S$ (R and S could be relations or result-sets from other queries), and authenticated structures for both A_i in R and A_j in S exist. The server can provide the proof for the join as follows: 1. Select the relation with the smaller size, say R, 2. Construct the VO for R (if R is an entire relation then the $\mathcal{VO}$ contains only the signature of the root node from the index of R), 3. Construct the $\mathcal{VO}$s for each of the following selection queries: for each record r_k in R, $q_k =$"SELECT * FROM S WHERE

$r.A_j = r_k.A_i$". The client can easily verify the join results. First, it authenticates that the relation R is complete and correct. Then, using the $\mathcal{VO}$ for each query q_k, it makes sure that it is complete for every k (even when the result of q_k is empty). After this verification, the client can construct the results for the join query and be sure that they are complete and correct.

Authenticating Selection Queries and Aggregation Queries using One Unified Structure. It is not hard to see that both the AAB-tree and AAR-tree can support the authentication of selection queries as well. Furthermore, the embedding idea used for the EMB-tree and its variants could be easily applied to the AAB-tree and AAR-tree in order to reduce the $\mathcal{VO}$ size.

Handling Encrypted Data: In some scenarios it might be necessary for the data owner to publish only encrypted versions of its data for privacy preservation and data security purposes [Hacigümüs et al., 2002a, Agrawal et al., 2005, Mykletun and Tsudik, 2006]. It should be made clear that an encrypted database does not provide a solution to the query authentication problem: the servers could still purposely omit from the results tuples that actually satisfy the query conditions. It is interesting to mention that the APS-tree can work without modifications with encrypted data. The server does not need to access the data or perform any computations or comparisons on the data. It only needs to retrieve the encrypted data at a specific location of the one-dimensional prefix sums array. This location information can be provided by the client that can infer it from the query, the domain sizes and the fanout f of the APS tree. On the other hand, the AAB-tree and the AAR-tree structures cannot support encrypted databases. The difficulty arises in requiring the server to perform comparisons on encrypted data. Techniques like order-preserving encryption [Agrawal et al., 2004], if acceptable under certain security models, can be used with the proposed structures.

Chapter 5

Enabling Authentication for Sliding Window Query over Data Streams

5.1 Problem Formulation

Stream outsourcing. We adopt the data outsourcing model in a streaming environment. Formally, we define three entities, the *service provider* who is the originator of the stream, the *server* who answers queries, and the *client* who registers queries and receives authenticated results (see Figure 1·1)[1]. The service provider constructs special authentication structures that can be updated in real-time over the stream and that are tailored for answering one-shot and sliding window selection and aggregation queries. The provider forwards the original stream to the server along with the necessary information to enable reconstruction of the authentication structures at the server side. The server uses these structures to generate verifiable query answers, and forwards the final results to the clients.

The data stream model. We model a data stream S as an infinite sequence of tuples $S = (a_1, a_2, \dots)$. Tuples arrive one at a time, i.e., in the i-th time unit tuple a_i arrives. A sliding window of size n consists of elements $(a_{t-n+1}, \dots, a_t)$ where a_t is the last tuple received so far, and n is in number of tuples. This model is typically referred to as *tuple-based sliding windows* [Datar et al., 2002, Babcock et al., 2002] (i.e., querying the most recent n tuples). In some applications it might be desirable to use *time-based sliding windows*, where each tuple is associated with a *timestamp* and we are interested in querying all tuples within time interval $[t_{\text{now}} - T, t_{\text{now}}]$ where t_{now} is the current time and T is the

[1]This Chapter is extended from [Li et al., 2007].

window size in timestamps. For ease of exposition we focus on tuple-based sliding windows, and discuss extensions for time-based sliding windows in Section 5.5.

Queries. Assume that each tuple consists of multiple attributes and clients issue continuous selection queries of the following form:

`SELECT * FROM Stream WHERE`

$l_1 \leq A_1 \leq u_1$ `AND ...` `AND` $l_d \leq A_d \leq u_d$

`WINDOW SIZE` n`, SLIDE EVERY` σ

where (l_i, u_i) are the selection ranges over attributes A_i.[2] We will also consider aggregation queries of a similar form:

`SELECT AGG(`A_x`) FROM Stream WHERE ...`

A_x is any tuple attribute and `AGG` is any distributive aggregate function, like SUM, COUNT, MIN, and MAX.

Assume that a query is issued at time t. The answer to a selection query consist of those tuples that fall within the window $(a_{t-n+1}, \ldots, a_t)$ and whose attributes satisfy the selection predicates. For one-shot queries the server constructs the answer once and reports the result. For sliding window queries, the server constructs the initial answer at time t, which is again a one-shot query, and incrementally communicates the necessary changes to the clients, as tuples expire from the window and new tuples become available.

Our authentication algorithms will guarantee that the server does not introduce any spurious tuples, does not drop any tuples, and does not modify any tuples. In other words, our techniques guarantee both *correctness* and *completeness* of the results. Similarly for aggregation queries, we will guarantee that the aggregate is computed over the correct set of tuples, and properly updated over the sliding window.

An important observation for authenticating queries in a streaming setting is that any solution that can provide authenticated responses on a per tuple basis will have to trigger

[2]The selection attributes can be categorical as well, but without loss of generality, we will concentrate on numerical attributes in this work.

a signing operation at the provider on a per tuple arrival basis, which is very costly (see Chapter 2). The only alternative is to amortize the signing cost by performing signing operations across several tuples. This approach will lower the update overhead at the cost of providing delayed query responses. In many applications, delayed responses can often be tolerated and, given the complexity of this problem, our main interest will be to design algorithms that minimize signing, authentication and querying costs, given a maximum permissible response delay b. For one-shot window queries, clients will receive replies in the worst case b tuple arrivals after the time that the query was issued. For sliding window queries, clients will receive necessary updates with at most a b-tuple delay. Even though we introduce delays, we do not change the ad-hoc window semantics: The answers provided are with respect to the user defined window specifications t, n, σ. In critical applications (e.g., anomaly detection) preserving window semantics is very important.

A straightforward approach for answering sliding window queries would be to re-compute the exact answer every σ tuple arrivals, which is unacceptable. Alternatively, we could maintain the answers of all registered queries at the server side, update them incrementally as the window evolves, and communicate only the necessary changes to clients. This solution is unacceptable in streaming settings with a large number of queries and clients, putting unnecessary storage burden on the server. Hence, we explore solutions that incrementally compute result updates without explicitly keeping state on a per query/per client basis. In addition, we will assume that a maximum permissible window size N has been determined, and we will require our structures to have storage linear or near-linear in N.

Finally, we will evaluate our solutions, both analytically and experimentally, using the following metrics: 1. The query cost for the server; 2. The communication overhead for the authentication information, i.e., the size of the *verification object*, or the $\mathcal{VO}$. 3. The authentication cost for the client; 4. The update cost for the provider and the server; 5. The storage cost for the provider and the server; 6. Support for multi-dimensional queries.

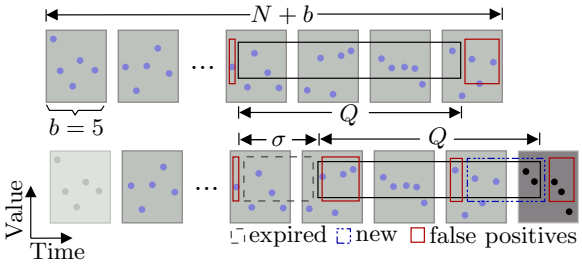


Figure 5·1: The Tumbling Merkle Tree.

5.2 The Tumbling Merkle Tree

In this section we present a structure based on the Merkle binary search tree for answering one-dimensional sliding window queries. Consider a predefined maximum window size N, a maximum permissible response delay b, and ad-hoc queries with window sizes $n < N$, on selection attribute A. The provider builds one Merkle binary search tree on the values of the attribute A for every b tuples within the window of the most recent $N + b$ tuples $(a_{t-N+1-b}, \ldots, a_t)$. Note that tuples in the tree are sorted by value A and not by order of arrival. An example is shown at the top of Figure 5·1, where every rectangle represents a Merkle tree on attribute A. Then, the provider signs the $\lceil N/b \rceil + 1$ Merkle trees. Each signature also includes $t_{\vdash}$ and $t_{\dashv}$, the indices of the oldest and newest tuple contained in the tree, i.e., $s = \mathcal{S}(SK, h_{\text{root}}|t_{\vdash}|t_{\dashv})$. All signatures are forwarded to the server. We call this structure the *Tumbling Merkle Tree* (TM-tree). Updating the TM-tree is easy. On every b new tuple arrivals the provider and the server bulk load a new Merkle tree with the new values and discard the oldest tree. In addition, the provider signs the tree and propagates the signature to the server. An advantage of this solution is that it amortizes the signing cost over every b tuples. The penalty is that authenticated results are provided with a delay up to b tuples. When the server receives a query but has not obtained the up-to-date signature from the provider, it temporarily buffers the query. When the next signature is received, the oldest buffered query refers to a sliding window with a low boundary at most $N + b$ tuples in the past.

One-Shot Queries. Consider a one-shot window query of size n. To answer the query the server traverses the $\lceil n/b \rceil + 1$ Merkle trees whose corresponding time intervals together cover the query time window, reporting all tuples satisfying the selection predicates (see Figure 5·1). For every tree, a *verification object* ($\mathcal{VO}$) is constructed using the original Merkle tree algorithm, and it is extended with the indices of the oldest and newest tuple in the tree. For trees fully contained in the query window, all tuples satisfying the selection predicate need to be reported (plus two boundary tuples for proof of completeness [Martel et al., 2004]). For the (at most) two boundary Merkle trees, up to $2b$ false positive answers might be returned (false positives are introduced in the temporal dimension though they are all satisfying tuples in the value dimension defined by the selection attribute A; see Figure 5·1). The server includes the false positives in the result, and leaves the filtering task to the client—this is necessary for proper authentication. Now, upon receiving a reply the client verifies correctness by authenticating the result of every tree individually; only then it filters out false positives from tuples in the two boundary Merkle trees whose indices are outside the query window. Correctness stems from the properties of Merkle trees discussed in Chapter 2. Completeness is guaranteed by proving that all qualifying values have been returned and that the right $\lceil n/b \rceil + 1$ trees have been traversed. Completeness of the values returned from each individual tree is guaranteed from the original Merkle binary search tree verification algorithm by including the left and right boundary values in the result [Martel et al., 2004]. A detailed discussion could be found in Chapter 4. Verifying that the correct trees have been traversed is possible by examining the indices of the oldest and newest tuples in the trees, whose values have been verified using the signature of each tree. Since trees do not intersect in the temporal dimension, we expect those indices to be consecutive and cover the query window completely.

Assuming that k is the result size of the query, the TM-tree achieves the following.

Lemma 5.2.1. *Given a maximum permissible response delay b, the TM-tree uses $O(N)$ space, and requires $O(\log b)$ time and $O(1/b)$ signing operations amortized to process a tuple. It takes $O(n/b \cdot \log b + b + k)$ time to answer a one-shot window query, and provides a $\mathcal{VO}$ of size $O(n/b \cdot \log b + b)$. The client takes $O(n/b \cdot \log b + b + k)$ time to authenticate*

the results.

Note that we interpret the $\mathcal{VO}$ as all information transmitted to the client besides the query results, so the $\mathcal{VO}$ size above also includes the false positives.

Sliding Window Queries. Consider a sliding window query issued at time t, with window size n and sliding period σ. The server first constructs the initial answer to the query for the window starting at t as described for one-shot queries, and sends the results to the client with delay at most b. Then, the server has to keep the results up-to-date as follows. Every b tuples, if a sliding period has ended, it constructs a $\mathcal{VO}$ that contains the tuples that have expired from the query window and the new tuples that have been generated since the last update to the client. The expired tuples can be found simply by traversing the (at most) $\lceil \sigma/b \rceil + 1$ Merkle trees at the left-most boundary of the sliding window; the new tuples can be found by querying the (at most) $\lceil \sigma/b \rceil + 1$ new Merkle trees that have been created since the last update. The cost of both operations is $O(\lceil \sigma/b \rceil \log b + b + k)$, where k is the total number of new result tuples in the current query window and expiring tuples in the previous query window. Note that the server may return up to $4b$ false positives (see Figure 5·1), which need to be filtered out by the client. False positives correspond to values from the boundary Merkle trees (expired and new) and also values that have appeared in the result already and are reported again. In practice, the query efficiency could be improved by querying and reporting results only from the new boundary Merkle trees without worrying about the expiring boundary trees. This is true because that the client, already possessing results in the expiring trees, could filter out the expiring tuples by checking their timestamps. Notice that in order to construct an update the server is oblivious to the previous state of the query answer. Hence, no per query/client state needs to be retained. Furthermore, for large n, updating the result is more efficient than reconstructing the answer from scratch, since a large number of intermediate Merkle trees do not have to be traversed. Correctness and completeness is guaranteed given that the answers provided by every Merkle tree can be authenticated individually, verifying that

both the expired and the new tuple sets are correct. Clearly, if $\sigma < b$, the server cannot do better than reporting updates only once every b tuples. If $\sigma > b$, the server reports updates every σ tuples with a maximum delay of b.

Lemma 5.2.2. *The TM-tree can support sliding window queries with a per period cost* $O(\lceil \sigma/b \rceil \log b + b + k)$ *and* $\mathcal{VO}$ *size* $O(\lceil \sigma/b \rceil \log b + b)$. *The client spends* $O(\lceil \sigma/b \rceil \log b + b + k)$ *time to authenticate the results.*

Supporting aggregations. Now we consider aggregate queries. Take SUM as an example. The Merkle binary search tree can support range-sum queries by combining with standard techniques [Lazaridis and Mehrotra, 2001] as we have discussed in Chapter 4. Assume that all data values are stored on the leaf nodes. We associate with every internal node u, the sum ($sum(u)$) of the aggregation attribute of all tuples stored in the subtree rooted at u. To enable authentication, we include this sum into the hash value stored in the parent node. More precisely, for an internal node u with children v and w, the hash value of u is computed as $h_u = \mathcal{H}(h_v|h_w|sum(u))$. Now, authenticating a SUM aggregate can be done efficiently by retrieving the covering set of leaf and internal nodes that span the query range, and computing the aggregate without having to traverse all the leaf nodes satisfying the selection predicate. It is easy to show that the covering set has a logarithmic size. The correctness and completeness proof is a technicality, and full details are in Chapter 4. Similar techniques can be used for all other distributive aggregates.

The TM-tree can be used for answering aggregation queries as well. For every Merkle tree that is completely contained in the window, we return the sum of all tuples falling in the selection range. However, for the two trees crossing the boundary, the server needs to return all the tuples in the selection range, instead of just the sum, so that the client can compute the sum of the tuples that are both in the selection range and in the query window. The same analysis applies to sliding window queries and for all other distributive aggregates, except MIN and MAX. For these aggregates, in addition the client has to maintain the MIN/MAX of the (at most) $\lceil n/b \rceil$ trees that do not contain any expiring tuples. These values will be needed for recomputing the total aggregate after an update is received.

Without giving the details of the derivation, which is rather straightforward, we conclude that for aggregation queries, all the aforementioned bounds hold by setting $k = 1$.

5.3 The Tumbling Mkd-tree

So far we have an efficient solution for one-shot and sliding window queries that has three drawbacks: it introduces $O(b)$ false positive answers (leading to a large $\mathcal{VO}$ size for large b), supports only one-dimensional queries, and has a high one-shot query cost. In this section we address the first two drawbacks and the third one in the next section. Notice that selection queries on variable window sizes with multiple selection predicates are essentially a range searching problem. This motivates the use of multi-dimensional range searching structures, like kd-trees [Bentley, 1975, de Berg et al., 2000], range trees [de Berg et al., 2000], or R-trees [Guttman, 1984]. In this work we will use authenticated *Merkle kd-trees* (Mkd-trees) as a building block for our solutions. The Mkd-tree is of independent interest for general authenticated range searching problems, but also a very good candidate for streaming settings, since it is a main memory data structure that can be bulk-loaded very efficiently and provides *guaranteed worst case performance* (as opposed to R-trees). Nevertheless, our techniques are not designed specifically for kd-trees; any space partitioning structure, such as R-trees, can be authenticated in the same manner.

5.3.1 The kd-tree

We briefly review the kd-tree in two dimensions; the extension to higher dimensions is straightforward. For simplicity we assume that all coordinates are distinct; if this is not the case, standard techniques such as *symbolic perturbation* [de Berg et al., 2000] can be used to resolve degeneracy. The kd-tree is a balanced binary tree $\mathcal{T}$. To build $\mathcal{T}$ on a set P of n points, we first divide P into two subsets P_l and P_r using a vertical line that divides the points into two sets of approximately equal size. We store the dividing line at the root node of $\mathcal{T}$ and continue with sets P_l and P_r as the left and right children of the root, by dividing each set into two subsets using a horizontal line. We continue recursively

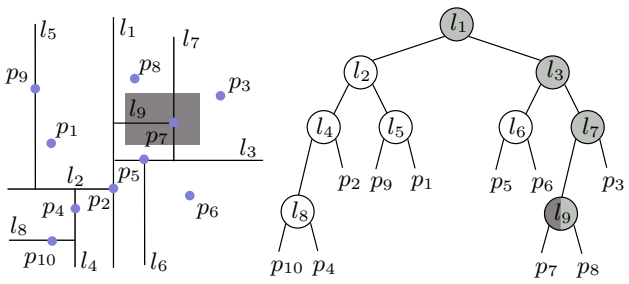


Figure 5·2: A kd-tree.

for all sets, alternating the direction of the dividing line for every level of the tree. The recursion stops when a set consists of only one point, which is stored as a leaf node of the tree. Each node u of the tree is naturally associated with a *bounding box*, denoted $box(u)$, which encloses all the points stored in the subtree rooted at u (see Figure 5·2).

To answer an orthogonal range search query Q, we start from the root of $\mathcal{T}$ and traverse all nodes whose associated bounding boxes intersect with or are contained in the query range Q. Please refer to Figure 5·2 for a query example where the query range is indicated by the small dark rectangle and the nodes accessed in the kd-tree are marked by the gray color. It is known that the kd-tree has excellent performance in practice, and furthermore, the following guarantee:

Lemma 5.3.1 ([Bentley, 1975])**.** *Let $\mathcal{T}$ be the kd-tree built on a set of n points. For any orthogonal query Q, the number of nodes in $\mathcal{T}$ whose bounding boxes intersect Q is at most $O(\sqrt{n}+k)$, where k is the number of points in the result, and the number of nodes whose bounding boxes intersect any side of Q (i.e., the boundary $\partial(Q)$ of Q) is at most $O(\sqrt{n})$.*

5.3.2 The Mkd-tree

We can extend the kd-tree with authentication information similarly to the Merkle tree. The authenticated Mkd-tree stores one hash value for every node. A leaf node v that contains point p stores hash value $h_v = \mathcal{H}(p)$. An internal node u with children v and w and dividing line l_u stores hash value $h_u = \mathcal{H}(h_v|h_w|l_u)$. The hash value of the root of the tree is signed by the data provider, producing signature $s = \mathcal{S}(SK, h_{\text{root}})$.

The query algorithm. To answer a range query Q, we recursively visit all nodes whose bounding boxes intersect with or are contained in Q. In addition to the query results, we also return the following $\mathcal{VO}$ in order for the client to authenticate the results: 1. The hash value h_u for each unvisited node u that has a visited sibling; 2. The dividing line l_u for each u where $box(u) \cap \partial(Q) \neq \varnothing$; 3. The point p of a visited leaf u if $p \notin Q$ (note that all $p \in Q$ are included in the query results); 4. The signature s. We also include a label for each of these nodes, for identifying its level and position in the tree. The label is described by $O(\log n)$ bits, or $O(1)$ words.

Lemma 5.3.2. *For any query Q, the Mkd-tree $\mathcal{T}$ built on n points returns a $\mathcal{VO}$ of size at most $O(\sqrt{n})$.*

Proof. We only need to bound the total number of nodes in the three categories defined above. According to Lemma 5.3.1, the number of nodes in $\mathcal{T}$ whose bounding boxes intersect $\partial(Q)$ is $O(\sqrt{n})$. This naturally bounds the number of type (2) and (3) elements. For a type (1) node u, its parent must have a bounding box that intersects $\partial(Q)$, so the number of type (1) nodes is also $O(\sqrt{n})$.

□

The authentication algorithm. Using the $\mathcal{VO}$ and the query results the client can recompute the hash value of the root and verify the signature s of the tree. Let R be the set of points in the query result. The client computes h_{root} using a recursive function starting from the root node. The recursive call has three inputs: a node u (i.e., the label of u), $box(u)$ and all $p \in R$ s.t. $p \in box(u)$. The algorithm (shown as Algorithm 4) is initialized with ComputeHash(root, $\mathbb{R}^2$, R). When the recursive call completes, the hash value of the root has been computed. The basic idea is to reconstruct the part of the Mkd-tree that was traversed for constructing the query answer at the server.

Since there are $O(\log n)$ levels of recursion, and each level involves $O(k)$ operations, the total running time is $O(k \log n)$. Correctness of the results is guaranteed similarly to the original Merkle tree, due to the collision-resistance of the hash function. Completeness is guaranteed by the following.

Algorithm 4: ComputeHash$(u, box(u), R_u)$

input: u: a node in $\mathcal{T}$, $box(u)$: u's bounding box, l_u: the dividing line of u, R_u: $p \in R$ s.t. $p \in box(u)$.
return: h_u: the hash value of u.

```
1  if u is a leaf then
2      if |R_u| ≠ 1 then report error;
3      let p be the only point in R_u;
4      if p ∉ box(u) then report error;
5      return H(p);
6  else
7      let v and w be u's children;
8      compute box(v) and box(w) from box(u) and l_u;
9      R_v := R_u ∩ box(v);
10     R_w := R_u ∩ box(w);
11     for z = v, w do
12         if box(z) ∩ Q = ∅ then
13             if h_v not available then report error;
14         else if box(z) ∩ ∂(Q) ≠ ∅ then
15             if l_z not available then report error;
16             h_z :=ComputeHash(z, box(z), R_z);
17         else
18             // box(z) is contained in Q;
19             build the Mkd-tree T_z of R_z;
20             h_z := hash value of the root of T_z;
21     return H(h_v|h_w|l_u);
```

Lemma 5.3.3. *Successful $\mathcal{VO}$ authentication of the Mkd-tree implies completeness of query results.*

Proof. If any node fully contained in Q is missing from the $\mathcal{VO}$, authentication will fail. Consider nodes that intersect with $\partial(Q)$. For any such node, the construction algorithm will have to traverse all the way to the leaves to identify potential query results. At some level of this recursion there will be one path fully disjoint with Q and another fully contained in Q. Clearly, if any fully contained node or its children is omitted from the $\mathcal{VO}$ authentication will fail; a needed hash for constructing the hash of the parent is missing. Similarly, if any node that has a parent that intersects with $\partial(Q)$ is missing, authentication will fail; the hash of this parent will have to be computed due to a child that is fully contained in Q. Hence, if the hash of the root authenticates correctly, no points below nodes that are

contained or intersect with Q have been omitted from the result.

□

The following holds for the Mkd-tree:

Theorem 5.3.4. *Given a set of n points in the plane, an Mkd-tree takes $O(n)$ space and can be built in $O(n \log n)$ time. Given an orthogonal range search query Q, it takes $O(\sqrt{n}+k)$ time to construct the answer, with a $\mathcal{VO}$ size $O(\sqrt{n})$. The client needs $O(\sqrt{n}+k\log n)$ time to authenticate the results.*

The kd-tree can be extended into an aggregate structure similarly to the discussion in Section 5.2, by hashing a node with $h_u = \mathcal{H}(h_v|h_w|l_u|sum(u))$. The bounds in Theorem 5.3.4 hold by setting $k = 1$ for aggregation queries.

5.3.3 The Tumbling Mkd-tree

We can now extend the TM-tree into a Tumbling Mkd-tree (TMkd-tree), by replacing the Merkle trees with Mkd-trees. Clearly, the new structure eliminates false positives, since now for the boundary kd-trees that intersect with the query range we can issue two-dimensional range queries that will report qualifying tuples within both the selection and temporal axes. The TMkd-tree significantly reduces the sliding window query cost, especially the $\mathcal{VO}$ size.

Theorem 5.3.5. *Given a maximum permissible response delay b, the TMkd-tree uses $O(N)$ space, and requires $O(\log b)$ time and $O(1/b)$ amortized signing operations to process a tuple. It supports sliding window queries with per period cost $O(\lceil\sigma/b\rceil\sqrt{b}+k)$, and $\mathcal{VO}$ size $O(\lceil\sigma/b\rceil\sqrt{b})$. For aggregation queries, the same bounds hold by setting $k = 1$.*

In addition, the structure naturally extends to higher dimensions. For queries with d selection attributes, we use $(d+1)$-dimensional kd-trees (time is always one dimension).

5.4 Reducing One-shot Query Cost

The TM-tree and TMkd-tree are robust solutions for sliding window queries. Nevertheless, they suffer from high one-shot query cost, especially for large n, since a large number

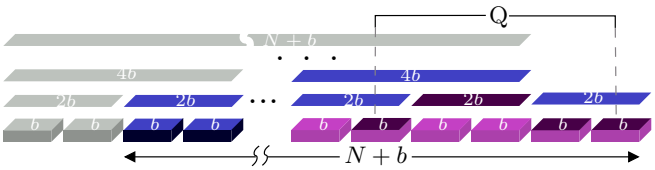


Figure 5·3: The DMkd-tree. Trees with boundaries outside of the maximum window can be discarded. A one-shot query can be answered by accessing only a logarithmic number of trees.

(n/b) of trees need to be queried for constructing the answer. In the following we focus on algorithms for reducing one-shot query cost at the expense of either slightly increased storage cost or slightly increased update cost. Clearly, the same solutions can be used for constructing the initial results of sliding window queries (recall that constructing the initial answer to a sliding window query is a one-shot query), which is a very important improvement especially in streaming scenarios where a large number of users pose queries dynamically. In the rest, we do not discuss extensions for aggregate queries, since they are similar to the solutions discussed already for the TMkd-tree.

5.4.1 The Dyadic Mkd-tree

We present a structure that combines the Merkle tree and the Mkd-tree, yielding a solution that has much faster one-shot query cost than the TM-tree for one-dimensional queries, at the expense of slightly increased maintenance cost at the provider and server, and slightly increased storage cost at the server.

The structure. We call the new structure the *Dyadic Mkd-tree* (DMkd-tree). Assume that $(N + b)/b$ is a power of 2 and let $\ell = \log((N + b)/b) - 1$. A DMkd-tree consists of $\ell + 1$ levels of forests of trees arranged in dyadic ranges over the time dimension. On level 0, we build a *Mkd-tree* for every b consecutive tuples in the stream (denoted with boxes in Figure 5·3). Levels 1 through ℓ consist of a forest of Merkle binary search trees over the values of attribute A (rectangles in the figure). More precisely, on level i, we build a Merkle binary search tree for every $2^i b$ tuples.

The maintenance algorithm. The DMkd-tree is fairly easy to maintain. The structure is initialized with the first $N + b$ tuples, building all levels in the dyadic hierarchy. After b new tuples have been received we build one new Mkd-tree on level 0 and discard the last kd-tree. At the same time we discard all Merkle trees that contain tuples outside the maximum window $N+b$. In general, after $2^i b$ tuples have been received, we build one new Merkle tree on level $i, 1 \leq i \leq \ell$ and discard all Merkle trees that fall outside the maximum window. For example, in Figure 5·3, after $2b$ tuples, the two left-most kd-trees and the left-most Merkle tree on all levels can be deleted.

A very important observation is that to build a new Merkle binary search tree $\mathcal{T}$ on level $i, 2 \leq i \leq \ell$, we do not need to re-sort any tuples. We can simply retrieve the tuples by scanning the leaf levels of the level-$(i-1)$ Merkle trees that fully span the range of $\mathcal{T}$. Hence, we can build tree $\mathcal{T}$ in a bottom-up fashion in time $O(2^i b)$.

Lemma 5.4.1. *The amortized maintenance cost per tuple for the DMkd-tree is* $O(\log N)$. *The amortized number of signing operations is* $O(1/b)$.

Proof. On level 0, we spend $O(b \log b)$ time to build each Mkd-tree. The amortized cost is $O(\log b)$ per tuple. Similarly, on level 1 we spend $O(b \log b)$ time to build a Merkle tree for every $2b$ tuples, so the amortized cost is also $O(\log b)$. On level $i, 2 \leq i \leq \ell$, we spend $O(2^i b)$ time to build the Merkle tree for every $2^i b$ tuples (as mentioned already the sorting operation can be avoided). This yields an amortized cost of $O(1)$. So the overall cost per tuple is $O(\log b + \ell) = O(\log N)$.

Applying similar arguments, the amortized number of signing operations on level i is $O(1/(2^i b))$, which yields a total of $O(1/b)$ signing operations per tuple.

□

Since each level occupies $O(N)$ space the entire structure at the server side takes $O(N \log(N/b))$ space. On the other hand, the service provider uses linear space for storing the DMkd-tree. Once a kd-tree or a Merkle tree at any level of the hierarchy has been covered by a higher level Merkle tree and its signature has been propagated to the server, it can be discarded (at the provider side the trees are only useful for producing signatures

and not for answering queries).

Query and authentication. The main idea behind using dyadic ranges is that constructing a query answer requires accessing only a logarithmic number of trees. Given a one-shot window query of size n, we decompose its time range into a series of sub-ranges, and answer each of them individually. First, we query the two level-0 Mkd-trees at the end-points of the query range. Then by a standard argument, the remaining portion of the range can be decomposed into $O(\log(n/b))$ sub-ranges, each of which is exactly covered by one of the Merkle trees. For example, in Figure 5·3 only three kd-trees and one Merkle tree need to be traversed. Trees at level 0 might be contained fully in the query time range or not. Trees at higher levels are always fully contained in the query. Hence, for higher levels in the hierarchy we only need to maintain one-dimensional structures on the selection attribute. For level 0 we need to maintain two-dimensional kd-trees to be able to filter out false positives.

To authenticate the results the client first individually authenticates the $\mathcal{VO}$ of each tree as in the original Merkle tree and Mkd-tree. This verifies correctness. To authenticate completeness, the client needs to verify that the appropriate trees have been traversed. First, the provider signs each tree individually creating a signature that contains the hash value of the root node and the indices of the oldest and newest tuple contained therein. Second, the server includes in the $\mathcal{VO}$s the indices of the newest and oldest tuple of the traversed trees. The client verifies that the indices returned by the server are consecutive and also cover the query window. Since no trees overlap in the temporal dimension this invariant has to hold. The authenticity of the timestamps returned is established by the signature of each tree. The following holds for the DMkd-tree:

Theorem 5.4.2. *For a maximum response delay b, the DMkd-tree uses $O(N \log(N/b))$ space, and requires $O(\log N)$ time and $O(1/b)$ signing operations amortized to process a tuple. It takes $O(\log n \log(n/b) + \sqrt{b} + k)$ time to answer a one-shot query, and provides a $\mathcal{VO}$ of size $O(\log n \log(n/b) + \sqrt{b})$. The client takes $O(\log n \log(n/b) + \sqrt{b} + k \log b)$ time to authenticate the results.*

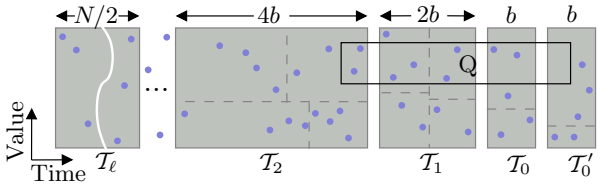


Figure 5·4: Querying the EMkd-tree.

Note that level-0 of DMkd-tree is essentially the same as TMkd-tree. Hence for sliding window queries, once the initial answers have been reported, subsequent updates can be handled by using the level-0 TMkd-tree, as in Section 5.3.3.

5.4.2 The Exponential Mkd-tree

So far, the DMkd-tree can be used for one-dimensional queries only. In this section, we present an algorithm that arranges a forest of Mkd-trees in an exponential hierarchy and can answer multi-dimensional queries. We call this structure the *Exponential Mkd-tree* (EMkd-tree). This approach can construct initial query answers much faster than the TMkd-tree, with a slight increase in the amortized per tuple update cost.

The structure. For simplicity assume that N/b is a power of 2 and let $\ell = \log(N/b) - 1$. The EMkd-tree consists of up to 2ℓ Mkd-trees: $\mathcal{T}_0, \mathcal{T}_0', \mathcal{T}_1, \mathcal{T}_1', \ldots, \mathcal{T}_\ell$. Each $\mathcal{T}_i$ is always present, but a $\mathcal{T}_i'$ may be present or absent, as will become clear shortly. The tree $\mathcal{T}_i$ or $\mathcal{T}_i'$ (if present) stores $2^i b$ consecutive tuples, hence $\sum_{i=0}^{\ell} |\mathcal{T}_i| = b(2^{\log N/b} - 1) = N - b$. For any $i < j$, all tuples stored in $\mathcal{T}_i$ are newer than any tuple stored in $\mathcal{T}_j$; and for any i, if $\mathcal{T}_i'$ is present, all tuples in $\mathcal{T}_i'$ are newer than any tuple in $\mathcal{T}_i$, such that no trees overlap in the time dimension (see Figure 5·4). The EMkd-tree structure is initialized with the first N tuples in the stream as follows. Tree $\mathcal{T}_\ell$ receives the first $N/2$ tuples, $\mathcal{T}_{\ell-1}$ the following $N/4$ tuples, until tree $\mathcal{T}_0$ which receives b tuples, for a total of $N - b$ tuples. The remaining b tuples are assigned to tree $\mathcal{T}_0'$, for a total of $\ell + 2$ Mkd-trees. No other $\mathcal{T}_i'$ exists yet (see Figure 5·5).

The update algorithm. After initializing the EMkd-tree we update it every b new

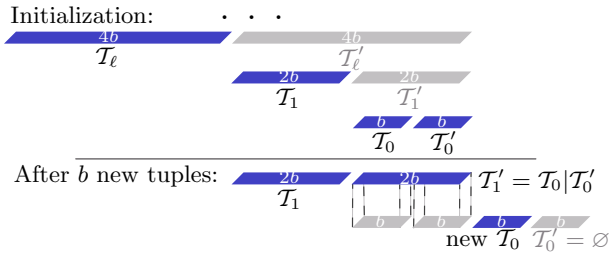


Figure 5·5: Initializing and updating the EMkd-tree.

arrivals. The update procedure first combines trees $\mathcal{T}_0$ and $\mathcal{T}'_0$ into tree $\mathcal{T}'_1$, then creates a new tree $\mathcal{T}_0$ using the latest b tuples and stops (see Figure 5·5). The next update proceeds similarly by creating a new tree $\mathcal{T}'_0$ using the latest b tuples and stops. This procedure continues by merging and propagating trees on consecutive levels of the hierarchy, until a level with a non-existent $\mathcal{T}'_i$ is found, at which point the propagation stops. Notice that we merge two trees $\mathcal{T}_i, \mathcal{T}'_i$ only if a third tree has been propagated from level $i-1$. Otherwise, we leave the trees intact. It is easy to see that after any update, the invariant that no trees overlap in the time dimension is preserved. A special case occurs at level ℓ. When tree $\mathcal{T}'_\ell$ is created the update procedure immediately discards tree $\mathcal{T}_\ell$ containing the oldest $N/2$ tuples, and sets $\mathcal{T}_\ell = \mathcal{T}'_\ell$. This iterative procedure is detailed in Algorithm 5.

Algorithm 5: EMkd-tree-Maintain

input: The b newest tuples in the stream.

```
1  build Mkd-tree R_0 on the b new tuples;
2  i := 0;
3  while true do
4      if T'_i present then
5          merge T_i and T'_i into R_{i+1};
6          remove T_i and T'_i;
7          T_i = R_i;
8          i := i + 1;
9      else
10         if i = ℓ then T_i = R_i;
11         else T'_i = R_i;
12         return;
```

Although the maintenance algorithm in the worst case may affect all the trees, its amortized cost can be effectively bounded by the following lemma:

Lemma 5.4.3. *$O(\log(N/b)\log N)$ is the amortized maintenance cost per tuple for the EMkd-tree. The amortized number of signing operations is $O(1/b)$.*

Proof. We use a charging argument. According to Theorem 5.3.4, the construction cost of a Mkd-tree built on n tuples is $O(n \log n)$, so we can charge an $O(\log n)$ cost to each tuple. Consider each tuple in the stream. It is involved in the construction of each $\mathcal{T}_i'$ exactly once, so it is charged by a total cost at most (asymptotically)

$$\begin{aligned}\sum_{i=0}^{\ell} \log(2^i b) &= \sum_{i=0}^{\ell} (i + \log b) = O(\log \frac{N}{b}(\log \frac{N}{b} + \log b)) \\ &= O(\log(N/b)\log N).\end{aligned}$$

For the signing cost, we use a similar charging scheme. Since building a Mkd-tree on n tuples requires only one signing operation, we can charge $O(1/n)$ to each tuple. Every tuple is involved in the construction of each $\mathcal{T}_i'$ exactly once, so it is charged with a total of $\sum_{i=1}^{\ell} 1/(2^i b) = O(1/b)$ signing operations.

□

It is easy to verify that in the worst case this algorithm will maintain $3N/2$ tuples, hence the storage cost is $O(N)$. In addition, there is an easy optimization for merging two Mkd-trees efficiently, without having to bulk load the structures from scratch. For all trees $\mathcal{T}_i, \mathcal{T}_i'$, and R_i, we first divide the data along the attribute dimension if i is even, and along the time dimension if i is odd (see Figure 5·4). Now, for even i, given that $\mathcal{T}_i$ and $\mathcal{T}_i'$ have no overlap in the temporal dimension and their roots are split on the attribute dimension, we can merge them by creating a new kd-tree root node and attaching $\mathcal{T}_i$ and $\mathcal{T}_i'$ as its two subtrees. On even levels the merging operation has constant cost. On odd levels (since the attribute domains of the two trees might overlap) bulk loading from scratch is unavoidable.

The query and authentication algorithms. To answer the query the server traverses the kd-trees starting from $\mathcal{T}_0$, until the query window is entirely covered. For kd-trees that

are entirely contained in the query window, the server poses a 2-sided range query. For the first and last kd-trees the server poses a 3-sided query, to filter out false positives. The server returns an individual $\mathcal{VO}$ for each tree using the original construction algorithm of the Mkd-tree. In addition, it includes the indices of the oldest and newest tuple in each tree. Authentication at the client side proceeds in exactly the same way as for the DMkd-tree.

The following can be stated for the EMkd-tree:

Lemma 5.4.4. *The EMkd-tree spends $O(\sqrt{n+b}+k)$ to construct an answer and returns a $\mathcal{VO}$ of size $O(\sqrt{n+b})$. It takes the client $O(\sqrt{n+b}+k\log(n+b))$ time to authenticate the results.*

Proof. Since the size of the Mkd-trees doubles every level, given a query with window size $n \geq b$, it is not difficult to see that the biggest tree that needs to be traversed to construct the answer has size at most n. For each level of the exponential hierarchy, at most two trees are traversed. So the total $\mathcal{VO}$ size is at most

$$O(\sqrt{n}) + O(\sqrt{n/2}) + \cdots + O(\sqrt{b}) = O(\sqrt{n}).$$

When $n < b$, at most two Mkd-trees of size b need to be traversed, so the overall $\mathcal{VO}$ size is $O(\sqrt{n+b})$. The same analysis applies for the total query time, plus an additional term linear to the size of the query results.

The bound on the authentication cost can be easily obtained by observing that the maximum height of the kd-trees that need to be traversed is $O(\max\{\log n, \log b\})$.

□

In order to answer sliding window queries, the provider and the server also need to maintain a TMkd-tree, concurrently with the EMkd-tree. The EMkd-tree is used for constructing the initial answers, while the TMkd-tree is used for constructing subsequent updates. Notice here that even if the initial answer from the EMkd-tree is delayed for some reason, the TMkd-tree can still provide updates unhindered (deltas can be computed even without knowing the answer of the previous window). The updates will have to be buffered

	TM-tree	TMkd-tree	DMkd-tree	EMkd-tree
Space	N	N	$N \log \frac{N}{b}$	N
Update cost	$\log b$	$\log b$	$\log N$	$\log \frac{N}{b} \log N$
Signing operations	$1/b$	$1/b$	$1/b$	$1/b$
Sliding query cost	$\lceil \sigma/b \rceil \log b + b + k$	$\lceil \sigma/b \rceil \sqrt{b} + k$	-	-
Sliding $\mathcal{VO}$ size	$\lceil \sigma/b \rceil \log b + b$	$\lceil \sigma/b \rceil \sqrt{b}$	-	-
Sliding authen. cost	$\lceil \sigma/b \rceil \log b + b + k$	$\lceil \sigma/b \rceil \sqrt{b} + k \log b$	-	-
One-shot query cost	$\frac{n}{b} \log b + b + k$	$\frac{n}{b}\sqrt{b} + k$	$\log n \log \frac{n}{b} + \sqrt{b} + k$	$\sqrt{n+b} + k$
One-shot $\mathcal{VO}$ size	$\frac{n}{b} \log b + b$	$\frac{n}{b}\sqrt{b}$	$\log n \log \frac{n}{b} + \sqrt{b}$	$\sqrt{n+b}$
One-shot authen. cost	$\frac{n}{b} \log b + b + k$	$\frac{n}{b}\sqrt{b} + k$	$\log n \log \frac{n}{b} + \sqrt{b} + k \log b$	$\sqrt{n+b} + k \log (n+b)$
Dimensions (*)	One	Multiple	One	Multiple
Aggregation (**)	Yes	Yes	Yes	Yes

Table 5.1: Summary of the (asymptotic) results for various solutions. (*) For d dimensions, all the $\sqrt{b}$ terms become $b^{1-1/(d+1)}$; all the $\sqrt{n+b}$ terms become $(n+b)^{1-1/(d+1)}$. (**) For aggregation queries, all the bounds hold by setting $k = 1$.

by the client until the initial answer arrives. The following summarizes the performance of the EMkd-tree:

Theorem 5.4.5. *For a maximum response delay b, the EMkd-tree uses $O(N)$ space, takes $O(\log(N/b) \log N)$ time and $O(1/b)$ signing operations amortized to process a tuple. It takes $O(\sqrt{n+b} + k)$ time to answer a one-shot query, with a $\mathcal{VO}$ size $O(\sqrt{n+b})$. The client takes $O(\sqrt{n+b} + k \log(n+b))$ time to authenticate the results.*

5.4.3 Comparison of DMkd-tree and EMkd-tree

Readers may be curious about the design principles behind the DMkd-tree and EMkd-tree. The intuition that motivates the design of both trees is to query only logarithm number of smaller trees. However, the goal of supporting either one dimensional queries or multi-dimensional queries differentiates the structures of these two trees. Noticeably, DMkd-tree has overlapping Merkle trees across various levels while the EMkd-tree has its trees from different levels covering disjoint segments along the time axis. The fact that the Merkle tree indexes one dimensional data leads to false positives when answering queries with both a query selection attribute and a time range. Hence, in the DMkd-tree Merkle trees are placed across different levels, with exponentially increasing overlapped ranges along the time axis, in order to increase the chance that there is a single Merkle tree at certain level that is fully covered by the given query in a specified time range. In fact, it is not

hard to see that this overlapping strategy ensures that at most four visits to small Mkd-trees at the bottom level, those appear at the two boundary sides, are necessary for any one-shot query if no false positive is desired. On the other hand, since Mkd-tree supports multi-dimensional queries without producing any false positives, it is not necessary to have overlapping trees in order to achieve the similar goal. Hence, trees are still designed to have exponentially increasing cover range along the time axis when level increase, however, they are not arranged to have overlaps.

5.5 Discussions

Supporting time-based windows. Supporting time-based windows is also possible. First, we use standard symbolic perturbation techniques to uniquely associate each tuple with a timestamp, if there are multiple tuples per time instant. Then, we generate dummy tuples for time instants without activity (a time instant can be defined as the smallest permissible window slide). The rest poses only technical difficulties that are not hard to overcome. Dummy tuples in general can be ignored and are used only for triggering signing operations. The theoretical bounds of the solutions hold, where N now expresses the maximum number of tuples within a maximum window.

Summary of various solutions. Table 5.1 summarizes the performance of various solutions. We can see that for sliding window queries, the TMkd-tree is better than the TM-tree for typical values of σ and b. The difference in the $\mathcal{VO}$ size could be significant when the sliding period σ is smaller than or comparable to b, which is common in real scenarios. However, for one-shot queries (or equivalently the initialization cost for sliding window queries), the two tumbling approaches both perform badly. The proposed DMkd-tree and EMkd-tree structures complement nicely in this respect. For one-dimensional queries, the DMkd-tree is the structure of preference, as it has excellent query performance, for a small penalty in the server's storage cost. When the server's memory is limited or for multi-dimensional queries, the EMkd-tree can be used. Notice that for sliding window queries,

the server needs to maintain both an EMkd-tree and a TMkd-tree, which doubles the storage cost in the worst case. For most practical cases though, the storage cost will still be smaller than the DMkd-tree. Finally, all structures can be extended easily to support aggregates like SUM, COUNT, AVG, MIN, and MAX.

Chapter 6

Query Execution Assurance on Data Streams

6.1 Problem Formulation

The queries examined in this Chapter for query execution assurance have the following structure[1]:

```
SELECT AGG(A_1), ..., AGG(A_N) FROM T
WHERE ... GROUP BY G_1, ..., G_M
```

This is a rather general form of SQL queries, as any "`SELECT, FROM, WHERE`" query can be written in a `GROUP BY` aggregate form by using the primary key of the input relation T as the `GROUP BY` attribute G (a `GROUP BY` clause that groups every tuple by itself). For example "`SELECT A,B FROM T WHERE B>10`" can be written as "`SELECT SUM(A),SUM(B) FROM T WHERE B>10 GROUP BY PK`". Note also that `GROUP BY` aggregate queries have wide applications in monitoring and statistical analysis of data streams (e.g., in networking and telephony applications). Previous work has addressed exactly these types of queries numerous times ([Zhang et al., 2005] and related work therein). For example, a query that appears frequently in network monitoring applications is the following:

```
SELECT SUM(packet_size) FROM IP_Trace
GROUP BY source_ip, destination_ip   (*)
```

In the rest of the thesis we will use this query as our main motivating example and concentrate on sum and count. Other aggregates that can be converted to these two (e.g., average, standard deviation, etc.) can be easily supported. Any solution to the above problem naturally supports the tumbling window semantics. Our proposed scheme, as we

[1]This Chapter extends from [Yi et al., 2007].

will see shortly, has the appealing feature of being easily extended to the sliding window semantics, which will be discussed in Section 6.6.

Data Stream Model. Following the example query of the previous section, the `GROUP BY` predicate partitions the streaming tuples into a set of n groups, computing one sum per group. The data stream can be viewed as a sequence of additions (or subtractions) over a set of items in $[n] = \{1, \ldots, n\}$. Denote this data stream as $\mathcal{S}$ and its τ-th tuple as $s^\tau = (i, u^\tau)$, an update of amount u to the ith group. Formally, the query answer can be expressed as a dynamic vector of non-negative integers $\mathbf{v}^\tau = [v_1^\tau, \ldots, v_n^\tau] \in \mathbb{N}^n$, containing one component per group aggregate. Initially, $\mathbf{v}^0$ is the zero vector. A new tuple $s^\tau = (i, u^\tau)$ updates the corresponding group i in $\mathbf{v}^\tau$ as $v_i^\tau = v_i^{\tau-1} + u^\tau$. We allow u^τ to be either positive or negative, but require $v_i^\tau \geq 0$ for all τ and i. When count queries are concerned, we have $u^\tau = 1$ for all τ. We assume that the L_1 norm of $\mathbf{v}^\tau$ is always bounded by some large m, i.e., at any τ, $\|\mathbf{v}^\tau\|_1 = \sum_{i=1}^n v_i^\tau \leq m$. Our streaming model is the same as the general Turnstile model of [Muthukrishnan, 2003], and our algorithms are designed to work under this model. The readers are referred to two excellent papers [Muthukrishnan, 2003, Babcock et al., 2002] for detailed discussions of data stream models.

Problem Definition. The problem of *Continuous Query Verification on data streams (CQV)*[2] is defined as follows:

Definition 6.1.1. *Given a data stream $\mathcal{S}$, a continuous query $\mathcal{Q}$ and a user defined parameter $\delta \in (0, \frac{1}{2})$, build a synopsis $\mathcal{X}$ of $\mathbf{v}$ such that for any τ, given any $\mathbf{w}^\tau$ and using $\mathcal{X}(\mathbf{v}^\tau)$, we: 1. raise an alarm with probability at least $1 - \delta$ if $\mathbf{w}^\tau \neq \mathbf{v}^\tau$; 2. shall not raise an alarm if $\mathbf{w}^\tau = \mathbf{v}^\tau$.*

Here $\mathbf{w}^\tau$, for example, could be the answer provided by the server, while $\mathcal{X}(\mathbf{v}^\tau)$ is the synopsis maintained by the client for verifying vector $\mathbf{v}$.

With this definition the synopsis raises an alarm with high probability if any component (or group answer) v_i^τ is inconsistent. Consider a server that is using semantic load shedding,

[2] We use the words Verification and Assurance interchangeably.

i.e., dropping tuples from certain groups, on bursty stream updates. In this scenario the aggregate of a certain, small number of components will be inconsistent without malicious intent. We would like to design a technique that allows a certain degree of tolerance in the number of erroneous answers contained in the query results, rather than raising alarms indistinctly. The following definition captures the semantics of *Continuous Query Verification with Tolerance for a Limited Number of Errors (CQV$^\gamma$)*:

Definition 6.1.2. *For any* $\mathbf{w}, \mathbf{v} \in \mathbb{Z}^n$*, let* $E(\mathbf{w}, \mathbf{v}) = \{i \mid w_i \neq v_i\}$*. Then* $\mathbf{w} \neq_\gamma \mathbf{v}$ *iff* $|E(\mathbf{w}, \mathbf{v})| \geq \gamma$ *and* $\mathbf{w} =_\gamma \mathbf{v}$ *iff* $|E(\mathbf{w}, \mathbf{v})| < \gamma$*. Given a data stream* $\mathcal{S}$*, a continuous query* $\mathcal{Q}$*, and user defined parameters* $\gamma \in \{1, \ldots, n\}$ *and* $\delta \in (0, \frac{1}{2})$*, build a synopsis* $\mathcal{X}$ *of* $\mathbf{v}$ *such that, for any* τ*, given any* $\mathbf{w}^\tau$ *and using* $\mathcal{X}(\mathbf{v}^\tau)$*, we: 1. raise an alarm with probability at least* $1 - \delta$*, if* $\mathbf{w}^\tau \neq_\gamma \mathbf{v}^\tau$*; 2. shall not raise an alarm if* $\mathbf{w}^\tau =_\gamma \mathbf{v}^\tau$*.*

Note that CQV is the special case of CQV$^\gamma$ with $\gamma = 1$. Similarly, we would like to design techniques that can support random load shedding, i.e., which can tolerate small absolute or relative errors on any component irrespective of the total number of inconsistent components. The following definition captures the semantics of *Continuous Query Verification with Tolerance for Small Errors (CQV$^\eta$)*:

Definition 6.1.3. *For any* $\mathbf{w}, \mathbf{v} \in \mathbb{Z}^n$*, let* $\mathbf{w} \not\approx_\eta \mathbf{v}$ *iff there is some* i *such that* $|w_i - v_i| > \eta$*, and* $\mathbf{w} \approx_\eta \mathbf{v}$ *iff* $|w_i - v_i| \leq \eta$ *for all* $i \in [n]$*. Given a data stream* $\mathcal{S}$*, a continuous query* $\mathcal{Q}$*, and user defined parameters* η *and* $\delta \in (0, \frac{1}{2})$*, build a synopsis* $\mathcal{X}$ *of* $\mathbf{v}$ *such that, for any* τ*, given any* $\mathbf{w}^\tau$ *and using* $\mathcal{X}(\mathbf{v}^\tau)$*, we: 1. raise an alarm with probability at least* $1 - \delta$*, if* $\mathbf{w}^\tau \not\approx_\eta \mathbf{v}^\tau$*; 2. shall not raise an alarm if* $\mathbf{w}^\tau \approx_\eta \mathbf{v}^\tau$*.*

Note that the definition above requires the *absolute* errors for each v_i^τ to be no larger than η. It is also possible to use *relative* errors, i.e., raise an alarm iff there is some i such that $|w_i^\tau - v_i^\tau|/|v_i^\tau| > \eta$. Thus CQV is also a special case of CQV$^\eta$ with $\eta = 0$.

We will work under the standard RAM model. Under this model, it is assumed that an addition, subtraction, multiplication, division, or taking mod involving two words takes one unit of time. We also assume that n/δ and m/δ fit in a word. In the rest of the thesis, we drop the superscript τ when there is no confusion.

6.2 Possible Solutions

This section presents some intuitive solutions and discusses why they do not solve the CQV problem. We focus on count queries only; the discussion extends to sum queries since count is a special case of sum. Abusing notations, we use $|\mathbf{v}|$ to denote the number of non-zero entries of $\mathbf{v}$. A naïve algorithm that always maintains $\mathbf{v}$ exactly would use $\Theta(|\mathbf{v}| \log m)$ bits of space. One might think of the following two simple solutions in order to reduce the high space requirement.

Random sampling. A first attempt is random sampling. Assuming a sampling rate r, the client randomly chooses rn groups. Clearly, with probability r this method will raise an alarm if $\mathbf{w} \neq \mathbf{v}$. In order to satisfy the problem statement requirements we need to set $r = 1 - \delta$. For CQV$^\gamma$, if the server modifies exactly γ answers, then the probability of raising an alarm is only roughly r^γ, which is obviously too small for practical r's and γ's. Thus, random sampling can at most reduce the space cost by a tiny fraction.

Sketches. Recent years have witnessed a large number of sketching techniques (for examples, [Flajolet and Martin, 1985, Cormode and Muthukrishnan, 2005, Alon et al., 1996, Bar-Yossef et al., 2002]) that are designed to summarize high-volume streaming data with small space. It is tempting to maintain such a sketch $\mathcal{K}(\mathbf{v})$ for the purpose of verification. When the server returns some $\mathbf{w}$, we compute $\mathcal{K}(\mathbf{w})$, which is possible since $\mathbf{w}$ exactly tells us what the elements have appeared in the stream and their frequencies. Then we check if $\mathcal{K}(\mathbf{v}) = \mathcal{K}(\mathbf{w})$.

It is imaginable that such an approach would likely to catch most unintentional errors such as malfunctions of the server or package drops during communication. However, the fact that they are not designed for verification leaves them vulnerable under certain attacks. For instance, let us consider the two AMS sketches from the seminal work of Alon et al. [Alon et al., 1996]. Their F_0 sketch uses a pairwise independent random hash function r and computes the maximum number of trailing zeros in the binary form of $r(i)$

for all tuples in the stream. This sketch is oblivious in the number of times a tuple appears, so will not detect any errors as long as $\mathbf{w}$ and $\mathbf{v}$ have the same set of locations on the groups with nonzero entries.

Their F_2 sketch computes the sum $\sum_{i=1}^{n} h(i)v_i$, where $h : \{1, \ldots, n\} \rightarrow \{-1, 1\}$ is chosen randomly from a family of 4-wise independent hash functions, and then repeat the process for a certain number of times independently. Below we will argue that for certain $\mathbf{w} \neq \mathbf{v}$, the chance that $\sum_{i=1}^{n} h(i)v_i = \sum_{i=1}^{n} h(i)w_i$ is high, thus the sketch will miss $\mathbf{w}$ unless many repetitions are used. This AMS sketch uses the BCH4 scheme (c.f. [Rusu and Dobra, 2007]) to construct a 4-wise independent random hash function $f : [n] \rightarrow \{0, 1\}$, and then set $h(i) = 2f(i) - 1$. Since $\sum_{i=1}^{n} h(i)(v_i - w_i) = 2\sum_{i=1}^{n} f(i)(v_i - w_i) - \sum_{i=1}^{n}(v_i - w_i)$, it is sufficient to construct a $\mathbf{w} \neq \mathbf{v}$ where $\sum_{i=1}^{n}(v_i - w_i) = 0$, such that $\sum_{i=1}^{n} f(i)(v_i - w_i) = 0$ are likely to happen.

Without loss of generality we assume $n = 2^r - 1$. Let S_0 and S_1 be two random r-bit integers. The BCH4 scheme computes $f(i)$ as $f(i) = (S_0 \odot i) \oplus (S_1 \odot i^3)$, where $\oplus$ is the vector dot product over the last r bits evaluated on $\mathbb{Z}_2$, i.e., assuming the last r bits of x (resp. y) is $x_1, \ldots, x_r$ (resp. $y_1, \ldots, y_r$), then $x \oplus y = (\sum_{i=1}^{r} x_i y_i) \mod 2$. We construct $\mathbf{w}$ as follows. For all odd i, and $i = 2^{r-1}$, $w_i = v_i$; for even $i \neq 2^{r-1}$, $v_i - w_i = -1$ if $i < 2^{r-1}$, and $v_i - w_i = 1$ if $i > 2^{r-1}$. It is clear that $\sum_{i=1}^{n}(v_i - w_i) = 0$. We will show that if $S_0 < 2^{r-1}$, then $\sum_{i=1}^{n} f(i)(v_i - w_i) = 0$. Consider any odd $i < 2^{r-1}$, and $j = i + 2^{r-1}$. We have

$$\begin{aligned} f(j) &= (S_0 \odot j) \oplus (S_1 \odot j^3) \\ &= (S_0 \odot (i + 2^{r-1})) \oplus (S_1 \odot (i + 2^{r-1})^3) \\ &= (S_0 \odot i) \oplus (S_1 \odot (i + 2^{r-1})^3), \end{aligned}$$

where the last equality is due to the fact that the first bit of S_0 is zero. On the other hand,

for even i, since

$$\begin{aligned}(i+2^{r-1})^3 &= i^3 + 3\cdot i^2\cdot 2^{r-1} + 3\cdot i\cdot 2^{2r-2} + 2^{3r-3} \\ &\equiv i^3 \qquad (\mod 2^r),\end{aligned}$$

we have $f(i) = f(j)$. Thus, the pair $f(i)(v_i - w_i)$ and $f(j)(v_j - w_j)$ cancel out, and we have $\sum_{i=1}^{n} f(i)(v_i - w_i) = 0$. So when $S_0 < 2^{r-1}$, which happens with probability 1/2, the sketch cannot catch this erroneous $\mathbf{w}$. This implies that at least $\Omega(\log \frac{1}{\delta})$ independent copies of sketches is needed to drive down the overall failure probability to the δ, giving a space complexity of $\Omega(\log \frac{1}{\delta} \log n)$ and update time $\Omega(\log \frac{1}{\delta})$, with potentially large hidden constants. As the typical δ values are very small in applications that require query assurance, these bounds are not satisfying. More importantly, 1/2 is merely lower bound on its failure probability as we do not know the exact probability it fails when $S_0 \geq 2^{r-1}$. For some specific values of $n = 2^r - 1$, we enumerated all values of S_0 and S_1 to compute the exact failure probabilities, shown in the following table.

r	3	4	5	6	7	8	9	10
fail prob	.75	.86	.84	.90	.94	.92	.93	.94

From the table we can see that the actual failure probability can be much larger than the lower bound of 1/2, and it is not clear if it is bounded away from 1, so it is not safe at all to just use $O(\log \frac{1}{\delta})$ copies. To summarize, although we have not theoretically ruled out the possibility that the AMS sketch may work, we can find cases where it is very likely to fail. Even if it could work (by proving an upper bound on the failure probability), it has to use at least $\Omega(\log \frac{1}{\delta} \log n)$ bits of space with a large hidden constant.

Finally, the AMS sketches have many variants developed in recent years, but to our knowledge, none of them has the guarantee of assurance required by the CQV problem. In the next section, we present our solution, which does not only solve the CQV problem, but also uses much less space than all the known sketches.

6.3 PIRS: Polynomial Identity Random Synopsis

This section presents two synopses, called *Polynomial Identity Random Synopses* (PIRS) and denote by $\mathcal{X}(\mathbf{v})$, for solving the CQV problem (Definition 6.1.1). The synopses, as the name suggests, are based on testing the identity of polynomials by evaluating them at a randomly chosen point. The technique of verifying polynomial identities can be traced back to the seventies [Freivalds, 1979]. It has found applications in e.g. verifying matrix multiplications and pattern matching [Motwani and Raghavan, 1995].

PIRS-1. Let p be some prime such that $\max\{m/\delta, n\} < p \leq 2\max\{m/\delta, n\}$. According to Bertrand's Postulate [Nagell, 1981] such a p always exists. We will work in the field $\mathbb{Z}_p$, i.e., all additions and multiplications are done modulo p. For the first PIRS, denoted PIRS-1, we choose α from $\mathbb{Z}_p$ uniformly at random and compute

$$\mathcal{X}(\mathbf{v}) = (\alpha - 1)^{v_1} \cdot (\alpha - 2)^{v_2} \cdot \cdots \cdot (\alpha - n)^{v_n}.$$

Having computed $\mathcal{X}(\mathbf{v})$ and given any input $\mathbf{w}$, PIRS is able to check if $\mathbf{w} = \mathbf{v}$ with high probability and without explicitly storing $\mathbf{v}$: We first check if $\sum_{i=1}^{n} w_i > m$, if so we reject $\mathbf{w}$ immediately; otherwise we compute $\mathcal{X}(\mathbf{w})$ as:

$$\mathcal{X}(\mathbf{w}) = (\alpha - 1)^{w_1} \cdot (\alpha - 2)^{w_2} \cdot \cdots \cdot (\alpha - n)^{w_n}.$$

If $\mathcal{X}(\mathbf{w}) = \mathcal{X}(\mathbf{v})$, then we declare that $\mathbf{w} = \mathbf{v}$; otherwise we raise an alarm. It is easy to see that we never raise a false alarm. Therefore we only need to show that we miss a true alarm with probability at most δ.

Theorem 6.3.1. *Given any $\mathbf{w} \neq \mathbf{v}$, PIRS raises an alarm with probability at least $1 - \delta$.*

Proof. Consider the polynomials $f_{\mathbf{v}}(x) = (x-1)^{v_1}(x-2)^{v_2} \cdots (x-n)^{v_n}$ and $f_{\mathbf{w}}(x) = (x-1)^{w_1}(x-2)^{w_2} \cdots (x-n)^{w_n}$. Since a polynomial with 1 as its leading coefficient, i.e., the coefficient of the term with the largest degree, is completely determined by its zeroes (with multiplicities), we have $f_{\mathbf{v}}(x) \equiv f_{\mathbf{w}}(x)$ iff $\mathbf{v} = \mathbf{w}$. If $\mathbf{v} \neq \mathbf{w}$, since both $f_{\mathbf{v}}(x)$ and $f_{\mathbf{w}}(x)$ have degree at most m, $f_{\mathbf{v}}(x) = f_{\mathbf{w}}(x)$ happens at no more than m values of x,

due to the fundamental theorem of algebra. Since we have $p \geq m/\delta$ choices for α, the probability that $\mathcal{X}(\mathbf{v}) = \mathcal{X}(\mathbf{w})$ happens is at most δ.

□

Note that once we have chosen α, $\mathcal{X}(\mathbf{v})$ can be incrementally maintained easily. For count queries, each tuple increments one of the v_i's by one, so the update cost is constant (one addition and one multiplication). For sum queries, a tuple $s = (i, u)$ increases v_i by u, so we need to compute $(\alpha - i)^u$, which can be done in $O(\log u)$ (exponentiation by squaring) time. To perform a verification with $\mathbf{w}$, we need to compute $(x - i)^{w_i}$ for each nonzero entry w_i of $\mathbf{w}$, which takes $O(\log w_i)$ time, so the time needed for a verification is $O(\sum \log w_i) = O(|\mathbf{w}| \log \frac{m}{|\mathbf{w}|})$. Since both $\mathcal{X}(\mathbf{v})$ and α are smaller than p, the space complexity of the synopsis is $O(\log \frac{m}{\delta} + \log n)$ bits.

Theorem 6.3.2. *PIRS-1 occupies $O(\log \frac{m}{\delta} + \log n)$ bits of space, spends $O(1)$ (resp. $O(\log u)$) time to process a tuple for count (resp. sum) queries, and $O(|\mathbf{w}| \log \frac{m}{|\mathbf{w}|})$ time to perform a verification.*

Some special care is needed when u is negative (or handling deletions for count queries), as the field $\mathbb{Z}_p$ is not equipped with division. We need first to compute $(\alpha - i)^{-1}$, the multiplicative inverse of $(\alpha - i)$ in modulo p, in $O(\log p)$ time (using Euclid's gcd algorithm [Knuth, 1997]), and then compute $(\alpha - i)^{-1|u|}$.

PIRS-2. When $n \ll m$ we can actually do slightly better with PIRS-2. Now we choose the prime p between $\max\{m, n/\delta\}$ and $2\max\{m, n/\delta\}$. For α chosen uniformly at random from $\mathbb{Z}_p$, we compute

$$\mathcal{X}(\mathbf{v}) = v_1\alpha + v_2\alpha^2 + \cdots + v_n\alpha^n.$$

By considering the polynomial $f_{\mathbf{v}}(x) = v_1 x + v_2 x^2 + \cdots + v_n x^n$, we can use the same proof to show that Theorem 6.3.1 still holds. Nevertheless, PIRS-2 has an $O(\log n)$ update cost for both count and sum queries, since we need to compute $u\alpha^i$ for a tuple (i, u) in the stream. Without repeating the details, we conclude with the following.

Theorem 6.3.3. *PIRS-2 occupies $O(\log m + \log \frac{n}{\delta})$ bits of space, spends $O(\log n)$ time to process a tuple, and $O(|\mathbf{w}| \log n)$ time to perform a verification.*

Since the space complexities of PIRS-1 and PIRS-2 are comparable, while PIRS-1 has a better update time, we recommend using PIRS-1 unless n is small compared to m and typical u.

Another nice property of PIRS is that the verification can also be performed in one pass of $\mathbf{w}$ using a constant number of words of memory. This is especially useful when $|\mathbf{w}|$ is large. The client will be able to receive $\mathbf{w}$ in a streaming fashion, verifies it online, and either forward it to a dedicated server for further processing, or a network storage device for offline use.

Space optimality. Below We give a lower bound showing that PIRS is space-optimal on the bits level for almost all values of m and n.

Theorem 6.3.4. *Any synopsis solving the CQV problem with error probability at most δ has to keep $\Omega(\log \frac{\min\{m,n\}}{\delta})$ bits.*

Proof. We will take an information-theoretic approach. Assume that $\mathbf{v}$ and $\mathbf{w}$ are both taken from a universe $\mathcal{U}$, and let $\mathcal{M}$ be the set of all possible memory states the synopsis might keep. Any synopsis $\mathcal{X}$ can be seen as a function $f : \mathcal{U} \to \mathcal{M}$; and if $\mathcal{X}$ is randomized, it can be seen as a function randomly chosen from a family of such functions $\mathcal{F} = \{f_1, f_2, \dots\}$, where f_i is chosen with probability $p(f_i)$. Without loss of generality, we assume that $p(f_1) \geq p(f_2) \geq \cdots$. Note that $\mathcal{X}$ needs at least $\log |\mathcal{M}|$ bits to record the output of the function and $\log |\mathcal{F}|$ bits to describe the function chosen randomly from $\mathcal{F}$.

For any $\mathbf{w} \neq \mathbf{v} \in \mathcal{U}$, let $\mathcal{F}_{\mathbf{w},\mathbf{v}} = \{f \in \mathcal{F} \mid f(\mathbf{w}) = f(\mathbf{v})\}$. For a randomized synopsis $\mathcal{X}$ to solve CQV with error probability at most δ, the following must hold for all $\mathbf{w} \neq \mathbf{v} \in \mathcal{U}$:

$$\sum_{f \in \mathcal{F}_{\mathbf{w},\mathbf{v}}} p(f) \leq \delta. \tag{6.1}$$

Let us focus on the first $k = \lceil \delta \cdot |\mathcal{F}| \rceil + 1$ functions $f_1, \dots, f_k$. It is easy to see that $\sum_{i=1}^{k} p(f_i) > \delta$. Since there are a total of $|\mathcal{M}|^k$ possible combinations for the outputs of

these k functions, by the pigeon-hole principle, we must have

$$|\mathcal{U}| \leq |\mathcal{M}|^k \tag{6.2}$$

so that no two $\mathbf{w} \neq \mathbf{v} \in \mathcal{U}$ have $f_i(\mathbf{w}) = f_i(\mathbf{v})$ for all $i = 1, \ldots, k$; otherwise we would find $\mathbf{w}, \mathbf{v}$ that violate (6.1).

Taking log on both sides of (6.2), we have

$$\log |\mathcal{U}| \leq (\lceil \delta \cdot |\mathcal{F}| \rceil + 1) \log |\mathcal{M}|.$$

Since $\mathbf{v}$ has n entries whose sum is at most m, by simple combinatorics, we have $|\mathcal{U}| \geq \binom{m+n}{n}$, or $\log |\mathcal{U}| \geq \min\{m, n\}$. We thus obtain the following tradeoff:

$$|\mathcal{F}| \cdot \log |\mathcal{M}| = \Omega(\min\{m, n\}/\delta).$$

If $\log |\mathcal{F}| \leq (1 - \epsilon) \log(\min\{m, n\}/\delta)$ (i.e., $|F| \leq (\min\{m, n\}/\delta)^{1-\epsilon}$) for any constant $\epsilon \in (0, 1)$, then $\mathcal{X}$ has to use super-polylogarithmic space $\log |\mathcal{M}| = \Omega((\min\{m, n\}/\delta)^{\epsilon})$; else $\mathcal{X}$ has to keep $\log |\mathcal{F}| \geq \log (\min\{m, n\}/\delta)$ random bits.

□

Therefore, when $m \leq n$, PIRS-1 is optimal as long as $\log n = O(\log \frac{m}{\delta})$; when $m > n$, PIRS-2 is optimal as long as $\log m = O(\log \frac{n}{\delta})$. Our bounds are not tight when $\log \frac{m}{\delta} = o(\log n)$ or $\log \frac{n}{\delta} = o(\log m)$.

Practical issues. The theoretical analysis above focuses on the bit-level space complexity. When implemented, however, both PIRS-1 and PIRS-2 use three words (p, α, and $\chi(\mathbf{v})$), and thus do not seem to have any difference. Nevertheless, there are some technical issues to be considered in practice.

First, we shall choose p to be the maximum prime that fits in a word, so as to minimize δ. Note that $\delta = m/p$ for PIRS-1 and $\delta = n/p$ for PIRS-2. For instance if we use 64-bit words and $m < 2^{32}$, then δ is at most 2^{-32} for PIRS-1, which practically means no

error at all. Second, since we need to extract the group id i from each incoming tuple directly, without the use of a dictionary (which would blow up the memory cost), the size of the group space, n, needs to be large for certain queries. For example, the query (*) of Section 6.1 has a group space of $n = 2^{64}$ (the combination of two IP addresses), although the actual number of nonzero entries $|\mathbf{v}|$ may be nowhere near n. In this case, since m is typically much smaller, PIRS-1 would be the better choice.

Information Disclosure on Multiple Attacks. Theorem 6.3.1 bounds the success rate for detecting a single attack attempted by the server. After an error has been detected, the client can choose to disclose this information to the server. If the error is not reported, then Theorem 6.3.1 will continue to hold. However, errors can occur due to faulty software or bad communication links, and may not be intentional. In this case we would like to give a warning to the server. Since a compromised, smart server can extract knowledge from this warning (e.g., it knows at least that the same attack will always fail), the guarantee of Theorem 6.3.1 is not applicable any more. In order to restore the $1 - \delta$ success rate after a reported attack, the synopsis has to be recomputed from scratch, which is impossible in a streaming setting. Hence, it is important to rigorously quantify the loss of guarantee after a series of warnings have been sent out without resetting the synopsis.

Let $e_k = 1$ if the k-th attack goes undetected and $e_k = 0$ otherwise. Let p_k be the probability that the server succeeds in its k-th attack after $k - 1$ failed attempts, i.e., $p_k = \Pr[e_k = 1 \mid e_1 = 0, \ldots, e_{k-1} = 0]$. From Theorem 6.3.1 we know that $p_1 \leq \delta$. In what follows we upper bound p_k with respect to the most powerful server, denoted as *Alice*, to demonstrate the strength of PIRS. We assume that Alice: 1. Knows how PIRS works except its random *seed*; 2. Maximally explores the knowledge that could be gained from one failed attack; and 3. Possesses infinite computational power.

Next, we precisely quantify the best Alice could do to improve p_k over multiple attacks. Denote by $\mathcal{R}$ the space of seeds used by PIRS. For any $\mathbf{w}, \mathbf{v}$ denote the set of *witnesses* $\mathcal{W}(\mathbf{w}, \mathbf{v}) = \{r \in \mathcal{R} \mid \text{PIRS raises an alarm on } r\}$ and the set of *non-witnesses* $\overline{\mathcal{W}}(\mathbf{w}, \mathbf{v}) =$

$\mathcal{R} - \mathcal{W}(\mathbf{w}, \mathbf{v})$. Note that $|\overline{\mathcal{W}}(\mathbf{w}, \mathbf{v})| \leq \delta|\mathcal{R}|$ if $\mathbf{w} \neq \mathbf{v}$, and $\overline{\mathcal{W}}(\mathbf{w}, \mathbf{v}) = \mathcal{R}$ if $\mathbf{w} = \mathbf{v}$. Suppose the seed PIRS uses is r. If Alice returns a correct answer $\mathbf{w} = \mathbf{v}$, she cannot infer anything about r. If she returns some $\mathbf{w} \neq \mathbf{v}$ and gets a warning, it is possible that Alice can determine $r \notin \overline{\mathcal{W}}(\mathbf{w}, \mathbf{v})$. However, even if we assume that Alice has enough computational power to compute both the sets of witnesses and non-witnesses, it is impossible for her to infer which witness PIRS is using as r. After $k-1$ failed attacks using $\mathbf{w}_1, \ldots, \mathbf{w}_{k-1}$, the set of seeds that Alice has ruled out is $\bigcup_{i=1}^{k-1} \overline{\mathcal{W}}(\mathbf{w}_i, \mathbf{v}_i)$, whose cardinality is at most $(k-1)\delta|\mathcal{R}|$. Thus, we have:

Lemma 6.3.5. $p_k \leq \frac{\delta}{1-(k-1)\delta}$.

Proof.

$$\begin{aligned} p_k &= \frac{|\text{set of non-witnesses}|}{|\text{set of remaining seeds}|} \\ &= \frac{|\overline{\mathcal{W}}(\mathbf{w}_k, \mathbf{v}_k)|}{|\mathcal{R} - \bigcup_{i=1}^{k-1} \overline{\mathcal{W}}(\mathbf{w}_i, \mathbf{v}_i)|} \leq \frac{\delta}{1-(k-1)\delta}. \end{aligned}$$

□

Theorem 6.3.6. *Assuming that Alice has made a total of k attacks to PIRS for any k, the probability that none of them succeeds is at least $1 - k\delta$.*

Proof. This probability is

$$\begin{aligned} & \Pr[e_1 = 0 \wedge \cdots \wedge e_k = 0] \\ = & \prod_{i=1}^{k} (1 - \Pr[e_i = 1 \mid e_1 = 0, \ldots, e_{i-1} = 0]) \\ \geq & \prod_{i=1}^{k} \left(1 - \frac{\delta}{1-(i-1)\delta}\right) = \prod_{i=1}^{k} \frac{1-i\delta}{1-(i-1)\delta} \\ = & \frac{1-\delta}{1} \cdot \frac{1-2\delta}{1-\delta} \cdot \cdots \cdot \frac{1-k\delta}{1-(k-1)\delta} = 1 - k\delta. \end{aligned}$$

□

Theorem 6.3.6 shows that PIRS is very resistant towards coordinated multiple attacks

even against an adversary with *unlimited computational power*. For a typical value of $\delta = 2^{-32}$, PIRS could tolerate millions of attacks before the probability of success becomes noticeably less than 1. Most importantly, the drop in the detection rate to $1 - k\delta$ occurs only if the client chooses to disclose the attacks to the server. Of course, such disclosure is not required in most applications.

6.4 Tolerance for Few Errors

This section presents a synopsis for solving the CQV$^\gamma$ problem (Definition 6.1.2). Let γ be the number of components in $\mathbf{v}$ that are allowed to be inconsistent. First, we present a construction that gives an exact solution that satisfies the requirements of CQV$^\gamma$, and requires $O(\gamma^2 \log \frac{1}{\delta} \log n)$ bits of space, which is practicable only for small γ's. Then, we provide an approximate solution which uses only $O(\gamma \log \frac{1}{\delta}(\log m + \log n))$ bits. Both solutions use PIRS as a black box, and therefore can choose either PIRS-1 or PIRS-2. We state all the results using PIRS-1 for count queries. The corresponding results for sum queries and PIRS-2 can be obtained similarly.

6.4.1 PIRS$^\gamma$: An Exact Solution

By using PIRS as a building block we can construct a synopsis that satisfies the requirements of CQV$^\gamma$. This synopsis, referred to as PIRS$^\gamma$, consists of multiple *layers*, where each layer contains $k = c_1\gamma^2$ *buckets* ($c_1 \geq 1$ is a constant to be determined shortly). Each component of $\mathbf{v}$ is assigned to one bucket per layer, and each bucket is represented using only its PIRS synopsis (see Figure 6·1). PIRS$^\gamma$ raises an alarm if at least γ buckets in any layer raise an alarm. The intuition is that if there are less than γ errors, no layer will raise an alarm, and if there are more than γ errors, at least one of the layers will raise an alarm with high probability (when the γ inconsistent components do not collide on any bucket for this layer). By choosing the probability of failure of the individual PIRS synopsis carefully, we can guarantee that PIRS$^\gamma$ achieves the requirements of Definition 6.1.2.

Concentrating on one layer only, let $b_1, \ldots, b_n$ be n γ-wise independent random num-

$$\begin{bmatrix} \underset{\{v_3\}}{\mathcal{X}_{11}} & \underset{\{v_2,v_5\}}{\mathcal{X}_{12}} & \underset{\{v_1,v_4,v_6\}}{\mathcal{X}_{13}} \\ \underset{\{v_1,v_3\}}{\mathcal{X}_{21}} & \underset{\{v_2,v_5\}}{\mathcal{X}_{22}} & \underset{\{v_4,v_6\}}{\mathcal{X}_{23}} \end{bmatrix}$$

Figure 6·1: The PIRS$^\gamma$ synopsis.

Algorithm 6: PIRS$^\gamma$-INITIALIZE(Prime p, Threshold γ)

```
1 c = 4.819, k = ⌈cγ²⌉
2 Generate β_{ℓ,j} uniformly at random from Z_p, for 1 ≤ ℓ ≤ log 1/δ, 1 ≤ j ≤ k
3 for ℓ = 1, ..., ⌈log 1/δ⌉ do
4     Layer L_ℓ = [X_1(v) := 0, ··· , X_k(v) := 0]
      // X_j(v) is a PIRS synopsis with δ' = 1/cγ
```

bers, uniformly distributed over $\{1,\ldots,k\}$. PIRS$^\gamma$ assigns v_i to the b_i-th bucket, and for each bucket computes the PIRS synopsis of the assigned subset of v_i's with probability of failure $\delta' = 1/(c_2\gamma)$ ($c_2 \geq 1$ is a constant to be determined shortly). According to Theorem 6.3.2 each of these k synopses occupies $O(\log \frac{m}{\delta'} + \log n) = O(\log m + \log n)$ bits. Given some $\mathbf{w} =_\gamma \mathbf{v}$, since there are less than γ errors, the algorithm will not raise an alarm. We can choose constants c_1 and c_2 such that if $\mathbf{w} \neq_\gamma \mathbf{v}$, then the algorithm will raise an alarm with probability at least $1/2$ for this layer. In this case there are two cases when the algorithm will fail to raise an alarm: 1. There are less than γ buckets that contain erroneous components of $\mathbf{w}$; 2. There are at least γ buckets containing erroneous components but at least one of them fails due to the failure probability of PIRS. We show that by setting constants $c_1, c_2 = 4.819$ either case occurs with probability at most $1/4$. Consider the first case. Since the v_i's are assigned to the buckets in a γ-wise independent fashion, by considering just γ of them, the probability that less than γ buckets get at least one of the erroneous v_i's is at most

$$\begin{aligned} & 1 - \frac{k}{k} \cdot \frac{k-1}{k} \cdot \cdots \cdot \frac{k-\gamma+1}{k} \\ \leq \;& 1 - \left(\frac{k-\gamma}{k}\right)^\gamma = 1 - \left(1 - \frac{\gamma}{k}\right)^{\frac{k}{\gamma}\cdot\frac{\gamma^2}{k}} \leq 1 - 2^{-\frac{2}{c_1}} \leq \frac{1}{4}, \end{aligned} \tag{6.3}$$

Algorithm 7: PIRS$^\gamma$-UPDATE(Tuple $s = (i, u)$)

1 **for** $\ell = 1, \ldots, \lceil \log 1/\delta \rceil$ **do**
2 $\quad b_{\ell,i} = (\beta_{\ell,\gamma} i^{\gamma-1} + \ldots + \beta_{\ell,2} i + \beta_{\ell,1}) \bmod k+1$
3 $\quad$ Update $L_\ell.\mathcal{X}_{b_{\ell,i}}(\mathbf{v})$ using s

Algorithm 8: PIRS$^\gamma$-VERIFY(Vector $\mathbf{w}$)

1 **for** $\ell = 1, \ldots, \lceil \log 1/\delta \rceil$ **do**
2 $\quad$ Layer $M_\ell = [\mathcal{X}_1(\mathbf{w}) := 0, \cdots, \mathcal{X}_k(\mathbf{w}) := 0]$
$\quad$ `//` $\mathcal{X}_j(\mathbf{w})$ `is a PIRS synopsis with` $\delta' = 1/c\gamma$
3 $\quad$ **for** $i = 1, \ldots, n$ **do**
4 $\quad\quad$ Generate $b_{\ell,i}$ as line 2, Algorithm 7
5 $\quad\quad$ Update $M_\ell.\mathcal{X}_{b_{\ell,i}}(\mathbf{w})$ by $s = (i, w_i)$
6 $\quad$ **if** $|\{j \mid L_i.\mathcal{X}_j(\mathbf{v}) \neq M_i.\mathcal{X}_j(\mathbf{w}), 1 \leq j \leq k\}| \geq \gamma$ **then** Raise an alarm

where the last inequality holds as long as $c_1 \geq 2/\log \frac{4}{3} = 4.819$.

Next, consider the second case. The probability that some of the γ buckets that are supposed to raise an alarm fail is:

$$1 - (1-\delta')^\gamma = 1 - \left(1 - \frac{1}{c_2\gamma}\right)^{c_2\gamma/c_2} \leq 1 - 2^{-\frac{2}{c_2}} < \frac{1}{4}, \tag{6.4}$$

which holds as long as $c_2 \geq 4.819$.

Therefore, using one layer PIRS$^\gamma$ will raise an alarm with probability at least 1/2 on some $\mathbf{w} \neq_\gamma \mathbf{v}$, and will not raise an alarm if $\mathbf{w} =_\gamma \mathbf{v}$. By using $\log \frac{1}{\delta}$ layers and reporting an alarm if at least one of these layers raises an alarm, the probability is boosted to $1 - \delta$.

Theorem 6.4.1. *For any $\mathbf{w} \neq_\gamma \mathbf{v}$, PIRS$^\gamma$ raises an alarm with probability at least $1 - \delta$. For any $\mathbf{w} =_\gamma \mathbf{v}$, PIRS$^\gamma$ will not raise an alarm.*

In addition to the $k\log \frac{1}{\delta}$ PIRS synopses, we also need to generate the γ-wise independent random numbers. Using standard techniques we can generate them on-the-fly using $O(\gamma \log n)$ truly random bits. Specifically, the technique of [Wegman and Carter, 1981] for constructing *k-universal hash families* can be used. Let p be some prime between n and $2n$, and $\alpha_0, \ldots, \alpha_{\gamma-1}$ be γ random numbers chosen uniformly and independently from $\mathbb{Z}_p$.

Then we set

$$b_i = ((\alpha_{\gamma-1} i^{\gamma-1} + \alpha_{\gamma-2} i^{\gamma-2} + \cdots + \alpha_0) \mod p) \mod k + 1,$$

for $i = 1, \ldots, n$. For an incoming tuple $s = (i, u)$, we compute b_i using the α_j's in $O(\gamma)$ time, and then perform the update to the corresponding PIRS. To perform a verification, we can compute in parallel for all the layers while making one pass over $\mathbf{w}$. The detailed initialization, update and verification algorithms for PIRS$^\gamma$ appear in Algorithms 6, 7, and 8. The next theorem bounds both the space and time complexity of PIRS$^\gamma$.

Theorem 6.4.2. *PIRS$^\gamma$ requires $O(\gamma^2 \log \frac{1}{\delta}(\log m + \log n))$ bits, spends $O(\gamma \log \frac{1}{\delta})$ time to process a tuple in the stream, and $O(|\mathbf{w}|(\gamma + \log \frac{m}{|\mathbf{w}|})\log \frac{1}{\delta})$ time to perform a verification.*

With careful analysis a smaller constant in the big-Oh above can be achieved in practice. For a given γ, we choose the minimum k such that (6.3) is at most $1/2$, and choose $1/\delta'$ very large (close to the maximum allowed integer) so that (6.4) is almost zero. For instance if $\gamma = 2$ and 3, then $2\log \frac{1}{\delta}$ and $6\log \frac{1}{\delta}$ words suffice, respectively. For arbitrary γ, the storage requirement is $2\gamma^2 \log \frac{1}{\delta}$ words in the worst case.

Remark. Note that computing the γ-wise independent random numbers b_i is the bottleneck for the time bounds. We can trade space for faster update times by using other γ-wise independent random number generation schemes. For instance by using an extra $O(n^\epsilon)$ words per layer, the technique of [Siegel, 1989] can generate a b_i in $O(1)$ time provided that $\gamma \leq n^{\epsilon^3/2}$, for $\epsilon > 0$. The update and verification times become $O(\log \frac{1}{\delta})$ and $O(n\log \frac{1}{\delta})$, and the space bound $O(n^\epsilon \log \frac{1}{\delta} \log n)$ bits.

6.4.2 PIRS$^{\pm\gamma}$: An Approximate Solution

The exact solution works when only a small number of errors can be tolerated. In applications where γ is large, the quadratic space requirement is prohibitive. If we relax the definition of CQV$^\gamma$ to allow raising alarms when *approximately* γ errors have been observed, we can design more space-efficient algorithms. This approximation is often acceptable since

when γ is large, users probably will not concern too much if the number of errors detected deviates from γ by a small amount. This section presents such an approximate solution, denoted with PIRS$^{\pm\gamma}$, that guarantees the following:

Theorem 6.4.3. *PIRS$^{\pm\gamma}$: 1. raises no alarm with probability at least $1-\delta$ on any $\mathbf{w} =_{\gamma^-} \mathbf{v}$ where $\gamma^- = (1-\frac{c}{\ln\gamma})\gamma$; and 2. raises an alarm with probability at least $1-\delta$ on any $\mathbf{w} \neq_{\gamma^+} \mathbf{v}$ where $\gamma^+ = (1+\frac{c}{\ln\gamma})\gamma$, for any constant $c > -\ln\ln 2 \approx 0.367$.*

Note that this is a very sharp approximation; the multiplicative approximation ratio $1 \pm \frac{c}{\ln\gamma}$ is close to 1 for large γ.

PIRS$^{\pm\gamma}$ also contains multiple *layers* of *buckets*, where each bucket is assigned a subset of the components of $\mathbf{v}$ and summarized using PIRS (Figure 6·1). Focusing on one layer only, our desiderata is on any $\mathbf{w} =_{\gamma^-} \mathbf{v}$ not to raise an alarm with probability at least $1/2+\epsilon$ for some constant $\epsilon \in (0, 1/2)$, and on any $\mathbf{w} \neq_{\gamma^+} \mathbf{v}$ to raise an alarm with probability at least $1/2 + \epsilon$. By using $O(\log\frac{1}{\delta})$ independent layers and reporting the *majority* of the results, the probabilistic guarantee will be boosted to $1 - \delta$ using Chernoff bounds [Motwani and Raghavan, 1995].

Let k be the number of buckets per layer. The components of $\mathbf{v}$ are distributed into the k buckets in a γ^+-wise independent fashion, and for each bucket the PIRS sum of those components is computed using $\delta' = 1/\gamma^2$. Given some $\mathbf{w}$, let this layer raise an alarm only if all the k buckets report alarms. The intuition is that if $\mathbf{w}$ contains more than γ^+ erroneous members, then the probability that every bucket gets at least one such component is high; and if $\mathbf{w}$ contains less than γ^- erroneous members, then the probability that there exists some bucket that is not assigned any erroneous members is also high.

The crucial factor that determines whether a layer could possibly raise an alarm is the distribution of erroneous components into buckets. The event that all buckets raise alarms is only possible if each bucket contains at least one inconsistent component. Let us consider all the inconsistent components in $\mathbf{w}$ in some order, say $w_1, w_2, \ldots$, and think of each of them as a collector that randomly picks a bucket to "collect". Assume for now that we have enough inconsistent elements, and let the random variable Y denote

the number of inconsistent components required to collect all the buckets, i.e., Y is the smallest i such that $w_1, \ldots, w_i$ have collected all the buckets. Then the problem becomes an instantiation of the *coupon collector's problem* [Motwani and Raghavan, 1995] (viewing buckets as coupons and erroneous components as trials). With k buckets, it is known that $E(Y) = k \ln k + O(k)$, therefore we set k such that $\gamma = \lceil k \ln k \rceil$. It is easy to see that $k = O(\gamma / \ln \gamma)$, hence the desired storage requirement.

We need the following sharp bounds showing that Y cannot deviate too much from its mean.

Lemma 6.4.4 ([Motwani and Raghavan, 1995], Theorem 3.8)**.** *For any constant c',*

$$\Pr[Y \leq k(\ln k - c')] \leq e^{-e^{c'}} + o(1),$$
$$\Pr[Y \geq k(\ln k + c')] \leq 1 - e^{-e^{-c'}} + o(1),$$

where $o(1)$ is in terms of k.

Notice that $\ln \gamma \leq 2 \ln k$ for any $k \geq 2$, so Lemma 6.4.4 also infers that for any real constant c:

$$\Pr[Y \leq \gamma - c\frac{\gamma}{\ln \gamma} = \gamma^-] \leq e^{-e^c} + o(1), \tag{6.5}$$
$$\Pr[Y \geq \gamma + c\frac{\gamma}{\ln \gamma} = \gamma^+] \leq 1 - e^{-e^{-c}} + o(1). \tag{6.6}$$

Now, consider the following two cases. If $\mathbf{w} =_{\gamma^-} \mathbf{v}$, then the probability that these less than γ^- independent erroneous components cover all buckets is bounded by (6.5), which is also the upper bound for the probability that the layer raises an alarm. Thus, there exists some constant ϵ such that the probability of raising a false alarm is (for large γ)

$$e^{-e^c} \leq 1/2 - \epsilon,$$

for any $c > \ln \ln 2$. If $\mathbf{w} \neq_{\gamma^+} \mathbf{v}$, then considering only γ^+ of the inconsistent components which are independently distributed to the buckets, there are two cases in which a true alarm is not raised: 1. These γ^+ components do not cover all buckets; and 2. All the buckets are covered but at least one of them fails to report an alarm. The probability that the

first case occurs is bounded by (6.6); while the probability that the second case happens is at most $1-(1-\delta')^k$. By the union bound, the total probability that we produce a false negative is at most

$$1-e^{-e^{-c}}+o(1)+1-(1-\delta')^k \leq 2-e^{e^{-c}}-2^{-\frac{2}{\gamma}}+o(1).$$

For γ large enough, there exists a constant $\epsilon > 0$ such that this probability is at most $1/2-\epsilon$ for any $c > -\ln\ln 2$.

To summarize, if $c > \max\{\ln\ln 2, -\ln\ln 2\} = -\ln\ln 2$, then both the false positive and false negative probabilities are at most $1/2-\epsilon$ for some constant ϵ at one layer with $k = O(\gamma/\log\gamma)$ buckets. Below we analyze the error probabilities of using $\ell = O(\log\frac{1}{\delta})$ independent layers.

To drive down the error probabilities for both false positives and false negatives to δ, we use $\ell = O(\log\frac{1}{\delta})$ layers and report simple majority. We quantify this probability for false negatives; the other case is symmetric.

Each layer can be viewed as a coin flip that raises a true alarm with probability at least $1/2+\epsilon$. Let the random variable Z denote the number of layers that raise alarms. This process is a sequence of independent *Bernoulli trials*, hence Z follows the binomial distribution. For ℓ independent layers, the expectation of Z is at least $\mu = (1/2+\epsilon)\ell$. By the Chernoff bound, the probability that a majority of layers raise alarms is

$$\Pr[Z < \frac{1}{2}\ell] = \Pr[Z < \left(1-\frac{2\epsilon}{1+2\epsilon}\right)\mu] < e^{-\frac{\mu}{2}(\frac{2\epsilon}{1+2\epsilon})^2}. \tag{6.7}$$

Therefore, we need to ensure that $e^{-\frac{\mu}{2}(\frac{2\epsilon}{1+2\epsilon})^2} \leq \delta$, which can be satisfied by taking $\ell = \lceil\frac{1+2\epsilon}{\epsilon^2}\ln\frac{1}{\delta}\rceil$.

Finally, we use the technique discussed in Section 6.4.1 to generate γ^+-wise independent random numbers, by storing $O(\gamma^+) = O(\gamma)$ truly random numbers per layer. We have thus obtained the desired results:

Theorem 6.4.5. *PIRS$^{\pm\gamma}$ uses $O(\gamma\log\frac{1}{\delta}(\log m+\log n))$ bits of space, spends $O(\gamma\log\frac{1}{\delta})$ time to process an update and $O(|\mathbf{w}|(\gamma+\log\frac{m}{|\mathbf{w}|})\log\frac{1}{\delta})$ time to perform a verification.*

As mentioned before, the technique of [Siegel, 1989] can be used to obtain space-time trade-offs with respect to generating the γ^+-wise independent random numbers.

6.4.3 Information Disclosure on Multiple Attacks

Similarly to the analysis in Section 6.3, since PIRS$^\gamma$ has false negatives only, it is a randomized algorithm with one-sided errors like PIRS. It is easy to prove that Theorem 6.3.6 holds for PIRS$^\gamma$ as well.

PIRS$^{\pm\gamma}$ is a randomized algorithm with two-sided errors, which is in favor of attackers and may be exploited by a smart server, since a portion of the seeds can be excluded both in the case that Alice is sending an incorrect answer and when she is sending a correct answer but PIRS$^{\pm\gamma}$ reports a false alarm. Using arguments similar to Theorem 6.3.6 the following can be stated:

Theorem 6.4.6. *Assuming that Alice has returned a total of k results $\mathbf{w}_1, \ldots, \mathbf{w}_k$ to PIRS$^{\pm\gamma}$, including both honest answers and attacks, the probability that PIRS$^{\pm\gamma}$ correctly identifies all consistent and inconsistent $\mathbf{w}_i$'s is at least $1 - k\delta$.*

Theorem 6.4.6 is slightly weaker than Theorem 6.3.6, since a new possible attack strategy for Alice is to simply return correct results and wait until a sufficient portion of the seeds have been ruled out before launching an attack. However, since the success probability drops linearly to the number of rounds, one can always intentionally set δ extremely small. For instance, if we choose $\delta = 2^{-30}$, it will take Alice a million rounds in order to have a 0.1% chance of launching a successful attack. As always, information disclosure by the clients is not necessary for most applications.

6.5 Tolerance for Small Errors

In this section we prove the hardness of solving CQV$^\eta$ (Definition 6.1.3) using sub-linear space, even if approximations are allowed. This problem can be interpreted as detecting if there is any component of $\mathbf{w}$ that has an absolute error exceeding a specified threshold η. We show that this problem requires at least $\Omega(n)$ bits of space.

Theorem 6.5.1. *Let η and $\delta \in (0, 1/2)$ be user specified parameters. Given a data stream $\mathcal{S}$, let $\mathcal{X}$ be any synopsis built on $\mathbf{v}$ that given $\mathbf{w}$: 1. raises an alarm with probability at most δ if $\mathbf{w} \approx_\eta \mathbf{v}$; and 2. raises an alarm with probability at least $1 - \delta$ if $\mathbf{w} \not\approx_{(2-\epsilon)\eta} \mathbf{v}$ for any $\epsilon > 0$. Then $\mathcal{X}$ has to use $\Omega(n)$ bits.*

Proof. We will reduce from the problem of approximating the infinity frequency moment, defined as follows. Let $A = (a_1, a_2, \dots)$ be a sequence of elements from the set $\{1, \dots n\}$. The *infinity frequency moment*, denoted by F_∞, is the number of occurrences of the most frequent element. Alon et al. [Alon et al., 1996] showed that any randomized algorithm that makes one pass over A and computes F_∞ with a relative error of at most $1/3$ and a success probability greater than $1 - \delta$ for any $\delta < 1/2$, has to use $\Omega(n)$ memory bits. In particular, they proved that even if each element appears at most twice, it requires $\Omega(n)$ bits in order to decide if F_∞ is 1 or 2 with probability at least $1 - \delta$.

Let $\mathcal{X}$ by a synopsis solving the problem stated in Theorem 6.5.1. We will show how to use $\mathcal{X}$ to compute the infinity frequency moment for any A in which each element appears at most twice. We will make one pass over A. For any element i that we encounter, we update $\mathcal{X}$ with the tuple $s = (i, \eta)$. In the end, we verify $\mathbf{w} = 0$ using $\mathcal{X}(\mathbf{v})$. If $\mathcal{X}$ asserts that $\mathbf{w} \approx_\eta \mathbf{v}$, we return $F_\infty = 1$; if $\mathcal{X}$ asserts that $\mathbf{w} \not\approx_{(2-\epsilon)\eta} \mathbf{v}$, we return $F_\infty = 2$. It is not difficult to see that we have thus computed the correct F_∞ with probability at least $1 - \delta$. □

If we allow relative errors instead of absolute errors, the problem is still difficult, as can be shown by setting $s = (i, n)$ for element i, and doing the verification with $\mathbf{w} = (n/(1+\eta), \cdots, n/(1+\eta))$ in the proof above.

Given the hardness of solving CQV$^\eta$, we are interested in seeking alternative methods that might be able to give us approximate answers using space less than the exact solution. Here we briefly discuss one such method.

The CM sketch. The CM sketch [Cormode and Muthukrishnan, 2005] uses $O(\frac{1}{\epsilon} \log \frac{1}{\delta'})$ words of space and provides an approximate answer $\widetilde{v}_i$ for any $i \in [n]$, that satisfies $v_i - 3\epsilon||\mathbf{v}||_1 \leq \widetilde{v}_i \leq v_i + 3\epsilon||\mathbf{v}||_1$ with probability $1 - \delta'$, for any $\epsilon \in (0, 1)$. However, this does not make it applicable for solving CQV$^\eta$ as: 1. The estimation depends on $||\mathbf{v}||_1$ and it

only works well for skewed distributions. Even in that case, in practice the estimation works well only for the large v_i's; and 2. $||\mathbf{v}||_1$ is not known in advance. However, if we can estimate an upper bound on $||\mathbf{v}||_1$, say $||\mathbf{v}||_1 \leq \Gamma$, then by setting $\epsilon = \frac{1}{3}\frac{\eta}{\Gamma}$ and $\delta' = \delta/n$, we can use the CM sketch to get approximate answers $\widetilde{v}_i$ such that $|\widetilde{v}_i - v_i| \leq \eta$ holds for all i *simultaneously* with probability at least $1 - \delta$. Now, given some $\mathbf{w}$, we generate an alarm iff there exists some i such that $|w_i - \widetilde{v}_i| \geq 2\eta$. This way, we give out a false alarm with probability at most δ if $\mathbf{w} \approx_\eta \mathbf{v}$, and generate an alarm with probability $1 - \delta$ if $\mathbf{w} \not\approx_{3\eta} \mathbf{v}$. For other $\mathbf{w}$'s, no guarantee can be made (see Figure 6·2). Especially, some false negatives may be observed for some range of $\mathbf{w}$. This solution uses $O(\frac{1}{\epsilon}\log\frac{n}{\delta}\log W)$ bits of space and $O(\log\frac{n}{\delta})$ time per update (W is the largest expressible integer in one word of the RAM model). The space dependence on $\frac{1}{\epsilon}$ is expensive, as $\frac{1}{\epsilon} = \frac{\Gamma}{\eta}$ in this case and the upper bound on $||\mathbf{v}||_1$ in practice might be large.

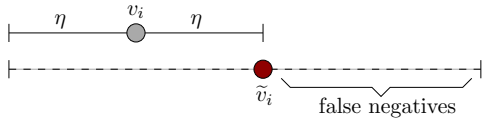


Figure 6·2: False negatives for the CM sketch approach.

6.6 Extensions

In this section we discuss several extensions of PIRS. We will focus on PIRS-1 for count queries only; the same arguments apply to sum queries, as well as to PIRS-2, PIRS$^\gamma$, and PIRS$^{\pm\gamma}$.

Handling Multiple Queries. The discussion so far focused on handling a single query per PIRS synopsis. Our techniques though can be used for handling multiple queries simultaneously. Consider a number of aggregate queries on a single attribute (e.g., packet size) but with different partitioning on the input tuples (e.g., source/destination IP and source/destination port). Let $\mathcal{Q}_1, \ldots, \mathcal{Q}_k$ be k such queries, and let the i-th query partition the incoming tuples into n_i groups for a total of $n = \sum_{i=1}^{k} n_i$ groups. A simple solution for

this problem would be to apply the PIRS algorithm once per query, using space linear in k. Interestingly, by treating all the queries as one unified query of n groups we can use one PIRS synopsis to verify the combined vector $\mathbf{v}$. The time cost for processing one update increases linearly in k, since each incoming tuple is updating k components of $\mathbf{v}$ at once (one group for every query in the worst case):

Corollary 6.6.1. *PIRS-1 for k queries occupies $O(\log \frac{m}{\delta} + \log n)$ bits of space, spends $O(k)$ time to process an update, and $O(|\mathbf{w}| \log \frac{m}{|\mathbf{w}|})$ time to perform a verification.*

Clearly, this is a very strong result, since we can effectively verify multiple queries with a few words of memory.

Handling sliding windows. Another nice property of PIRS-1 is that it is decomposable, i.e., for any $\mathbf{v}_1, \mathbf{v}_2, \mathcal{X}(\mathbf{v}_1 + \mathbf{v}_2) = \mathcal{X}(\mathbf{v}_1) \cdot \mathcal{X}(\mathbf{v}_2)$. (For PIRS-2, we have $\mathcal{X}(\mathbf{v}_1 + \mathbf{v}_2) = \mathcal{X}(\mathbf{v}_1) + \mathcal{X}(\mathbf{v}_2)$) This property allows us to extend PIRS for periodically sliding windows using standard techniques [Datar et al., 2002]. Again using our earlier example, one such sliding window query might be the following.

```
SELECT SUM(packet_size) FROM IP_Trace
GROUP BY source_ip, destination_ip
WITHIN LAST 1 hour SLIDE EVERY 5 minutes
```

In this case, we can build a PIRS-1 for every 5-minute period, and keep it in memory until it expires from the sliding window. Assume that there are k such periods in the window, and let $\mathcal{X}(\mathbf{v}_1), \ldots, \mathcal{X}(\mathbf{v}_k)$ be the PIRS for these periods. When the server returns a result $\mathbf{w}$, the client computes the overall $\mathcal{X}(\mathbf{v}) = \prod_{i=1}^{k} \mathcal{X}(\mathbf{v}_i)$, and then verifies the result.

Corollary 6.6.2. *For a periodically sliding window query with k periods, our synopsis uses $O(k(\log \frac{m}{\delta} + \log n))$ bits of space, spends $O(1)$ time to process an update, and $O(|\mathbf{w}| \log \frac{m}{|\mathbf{w}|})$ time to perform a verification.*

Synchronization. In our discussions we omitted superscript τ for simplicity. Hence, an implicit assumption was made that the result $\mathbf{w}^{\tau_s}$ returned by the server was synchronized with $\mathcal{X}(\mathbf{v}^{\tau_c})$ maintained by the client, i.e., $\tau_s = \tau_c$. Correct verification can be performed

only if the server and the client are synchronized. Obviously, such perfect synchronization is hard to obtain in practice, especially in a DSMS. Also, if n is large, transmitting the result itself takes non-negligible time. The solution to this problem is as follows. Suppose that the client sends out a request to the server asking for the query result at time τ, which is either a time instance at present or in the future. When the client has received s^τ from the stream $\mathcal{S}$ and has computed the synopsis for $\mathbf{v}^\tau$, it makes a copy of $\mathcal{X}(\mathbf{v}^\tau)$, and continues updating PIRS. When the server returns the answer $\mathbf{w}^\tau$, the client can do the verification using the snapshot. The synchronization problem once again illustrates the importance of using small space, as keeping a copy (or potentially many copies if there are significant delays in the server's response) could potentially become very expensive. Similar ideas can be used on the server side for dealing with queries referring to the past.

Exploiting locality. In many practical situations data streams tend to exhibit a large degree of locality. Simply put, updates to $\mathbf{v}$ tend to cluster to the same components. In this setting, it is possible to explore space/time trade-offs. We can allocate a small buffer used for storing exact aggregate results for a small number of groups. With data locality, a large portion of updates will hit the buffer. Whenever the buffer is full and a new group needs to be inserted, a victim is selected from the buffer using the simple *least recently used (LRU)* policy. Only then does the evicted group update PIRS, using the overall aggregate value computed within the buffer. We flush the buffer to update PIRS whenever a verification is required. Since we are aggregating the incoming updates in the buffer and update the synopsis in bulk, we incur a smaller, amortized update processing cost per tuple.

Chapter 7

Experiments

7.1 Experimental Evaluation on Authenticated Index Structures for Selection Queries

For our experimental evaluation we have implemented the aggregated signatures technique using a B^+-tree (ASB-tree), the MB-tree, the EMB-tree and its two variants, EMB$^-$-tree and EMB*-tree.

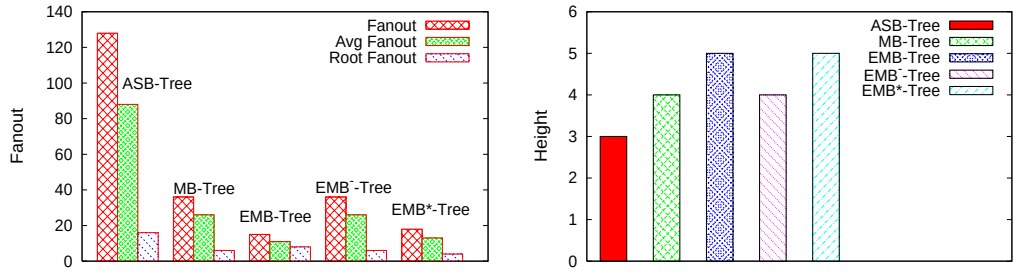
7.1.1 Setup

We use a synthetic database that consists of one table with $100,000$ tuples. Each tuple contains multiple attributes, a primary key A, and is 500 bytes long. For simplicity, we assume that an authenticated index is build on A, with page size equal to 1 KB. All experiments are performed on a Linux machine with a 2.8GHz Intel Pentium 4 CPU.

The cost $\mathcal{C}_{IO}$ of one I/O operation on this machine using 1KB pages for the indexes is on average 1 ms for a sequential read and 15 ms for random access. The costs $\mathcal{C}_{\mathcal{H}}$ of hashing a message with length 500 bytes is approximately equal to 1 to 2 μs. In comparison, the cost $\mathcal{C}_{\mathcal{S}}$ of signing a message with any length is approximately equal to 2 ms. The cost of one modular multiplication with 128 byte modulus is close to 100 μs. To quantify these costs we used the publicly available Crypto++ [Crypto++ Library, 2005] and OpenSSL [OpenSSL, 2005] libraries.

7.1.2 Performance Analysis

We run experiments to evaluate the proposed solutions under all metrics. First, we test the initial construction cost of each technique. Then, we measure their query and verifi-



(a) Fanout (b) Height

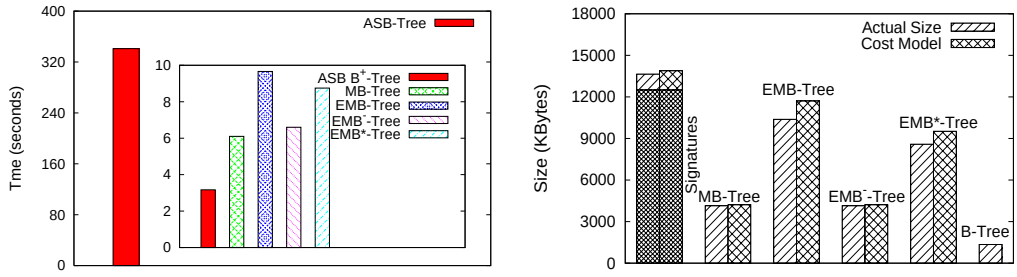
Figure 7·1: Index parameters.

cation performance. Finally, we run simulations to analyze their performance for dynamic scenarios. For various embedded index structures, the fanout of their embedded trees is set to 2 by default, except if otherwise specified.

Construction Cost

First we evaluate the physical characteristics of each structure, namely the maximum and average fanout, and the height. The results are shown in Figure 7·1. As expected, the ASB-tree has the maximum fanout and hence the smallest height, while the opposite is true for the EMB-tree. However, the maximum height difference, which is an important measure for the number of additional I/Os per query when using deeper structures, is only 2. Of course this depends on the database size. In general, the logarithmic relation between the fanout and the database size limits the difference in height of the different indices.

Next, we measure the construction cost and the total size of each structure, which are useful indicators for the owner/server computation overhead, communication cost and storage demands. Figure 7.2(a) clearly shows the overhead imposed on the ASB-tree by the excessive signature computations. Notice on the enclosed detailed graph that the overhead of other authenticated index structures in the worst case is twice as large as the cost of building the B^+-tree of the ASB-tree approach. Figure 7.2(b) captures the total size of each structure. Undoubtedly, the ASB-tree has the biggest storage overhead. The EMB-tree is storage demanding as well since the addition of the embedded trees decreases the index



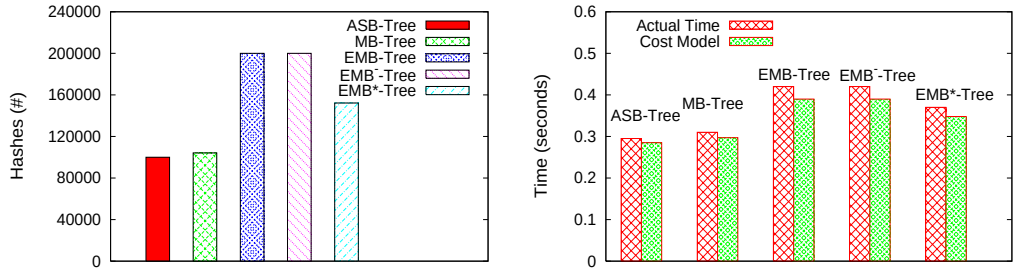
(a) Time (b) Storage

Figure 7·2: Index construction cost.

fanout substantially. The MB-tree has the least storage overhead and the EMB^*-tree is a good compromise between the MB-tree and the EMB-tree for this metric. In the same graph we also plot the size of a B-tree structure without authentication information. We can see that the MB-tree and the EMB^--tree are 3 times larger, while the EMB-tree is 7 times larger. Nevertheless, all structures are significantly smaller than the total database size, which is 47 MB. Notice that the proposed cost models capture very accurately the tree sizes.

The client/server communication cost using the simplest possible strategy is directly related to the size of the authenticated structures. It should be stressed however that for the hash based approaches this cost can be reduced significantly by rebuilding the trees at the server side. In contrast, the ASB-tree is not equally flexible since all signatures have to be computed by the owner.

The construction cost of our new authenticated index structures has two components. The I/O cost for building the trees and the computational cost for computing hash values. Figure 7·3 shows the total number of hash computations executed per structure, and the total time required. Evidently, the EMB-tree approaches increase the number of hashes that need to be computed. However, the additional computation time increases by a small margin as hashing is cheap, especially when compared with the total construction overhead (see Figure 7·2). Thus, the dominating cost factor proves to be the I/O operations of the index.



(a) Hash computations (b) Time

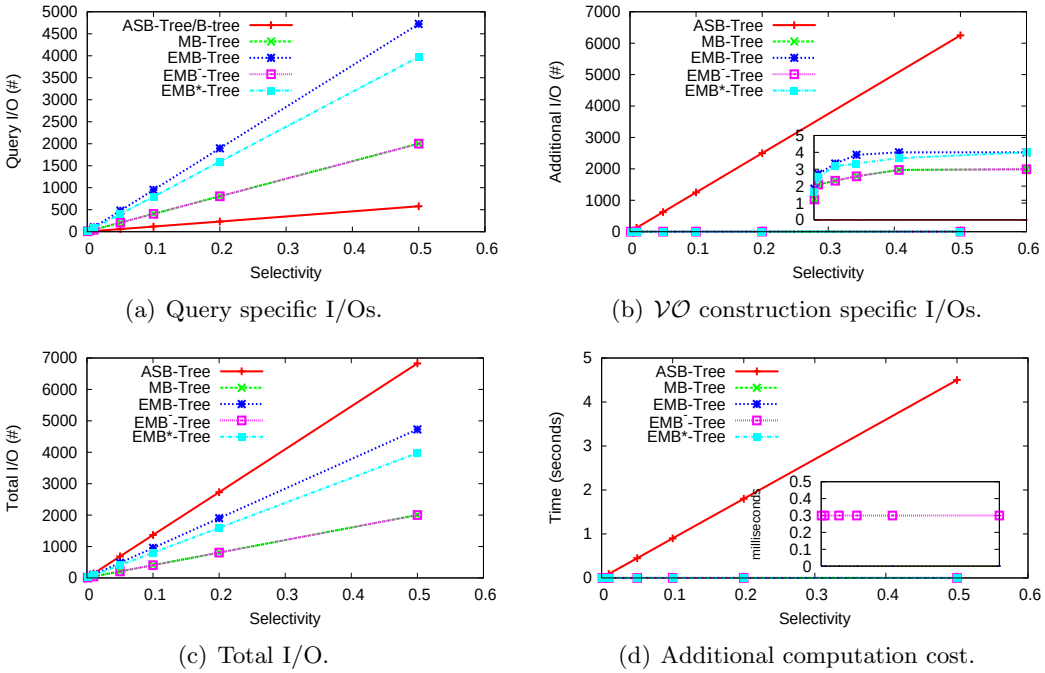
Figure 7·3: Hash computation overhead.

Query and Verification Cost

In this section we study the performance of the structures for range queries. We generate a number of synthetic query workloads with range queries of given selectivity. Each workload contains 100 range queries generated uniformly at random over the domain of A. Results reflect the average case, where the cost associated with accessing the actual records in the database has not been included. A 100 page LRU buffer is used, unless otherwise specified. In the rest of the experiments we do not include the cost model analysis not to clutter the graphs.

The results are summarized in Figure 7·4. There are two contributing cost factors associated with answering range queries. The first is the total number of I/Os. The second is the computation cost for constructing the $\mathcal{VO}$. The number of I/Os can be further divided into query specific I/Os (i.e., index traversal I/Os for answering the range query) and $\mathcal{VO}$ construction related I/Os.

Figure 7.4(a) shows the query specific I/Os as a function of selectivity. Straightforwardly, the number of page access is directly related to the fanout of each tree. Notice that the majority of page access is sequential I/O at the leaf level of the trees. We include in the graph the cost of answering a range query on an unauthenticated B-tree as well, which is exactly the same as the pure query I/O cost of the ASB-tree. Figure 7.4(b) shows the additional I/O needed by each structure for completing the $\mathcal{VO}$. Evidently, the ASB-tree has to perform a very large number of sequential I/Os for retrieving the signatures of the



(a) Query specific I/Os.

(b) $\mathcal{VO}$ construction specific I/Os.

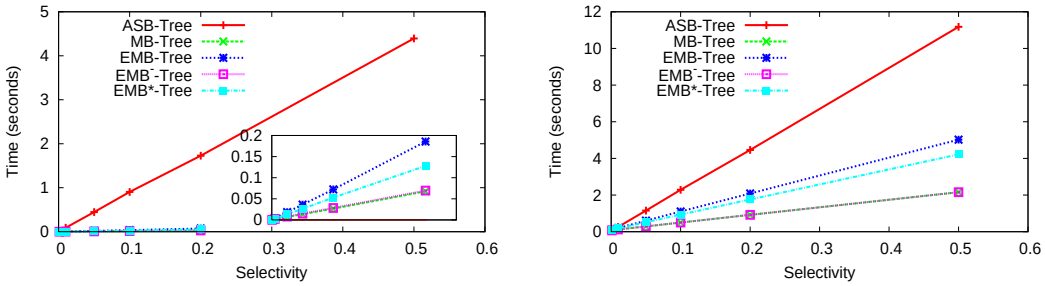
(c) Total I/O.

(d) Additional computation cost.

Figure 7·4: Performance analysis for range queries.

results, even though it's pure query cost is the same as an unauthenticated B-tree. Our authenticated index structures need to do only a few (upper bounded by the height of the index) extra random accesses for traversing the path that leads to the largest tuple in the query result. Overall, the EMB$^-$-tree and the MB-tree do 3 times more I/O access than the unauthenticated B-tree, while the EMB-tree 8 times more. Figure 7.4(c) shows the total I/O incurred by the structures. It is clear that the ASB-tree has the worst performance overall, even though its query specific performance is the best.

Figure 7.4(d) shows the runtime cost of additional computations that need to be performed for modular multiplications and hashing operations. The ASB-tree has an added multiplication cost for producing the aggregated signature. This cost is linear to the query result-set size and cannot be omitted when compared with the I/O cost. This observation is instructive since it shows that one cannot evaluate analytically or experimentally authenticated structures correctly only by examining I/O performance. Due to expensive cryptographic computations, I/O operations are not always a dominating factor. The



(a) Query runtime with LRU buffer.

(b) Query runtime without LRU buffer.

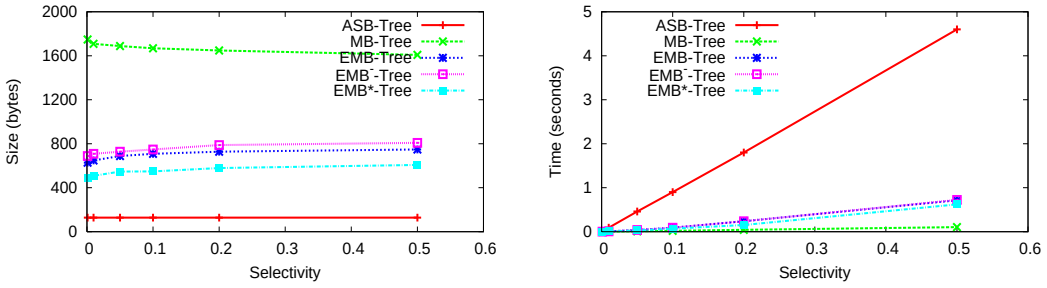
Figure 7·5: The effect of the LRU buffer.

EMB^--tree has a minor computation overhead, depending only on the fanout of the conceptual embedded tree. The rest of the structures have no computation overhead at all.

Interesting conclusions can be drawn by evaluating the effects of the main memory LRU buffer. Figure 7·5 shows the total query runtime of all structures with and without the LRU buffer. We can deduce that the LRU buffer reduces the query cost substantially for all techniques. We expect that when a buffer is available the computation cost is the dominant factor in query runtime, and the ASB-tree obviously has much worse performance, while without the buffer the I/O cost should prevail. However, since overall the ASB-tree has the largest I/O cost, the hash based structures still have better query performance.

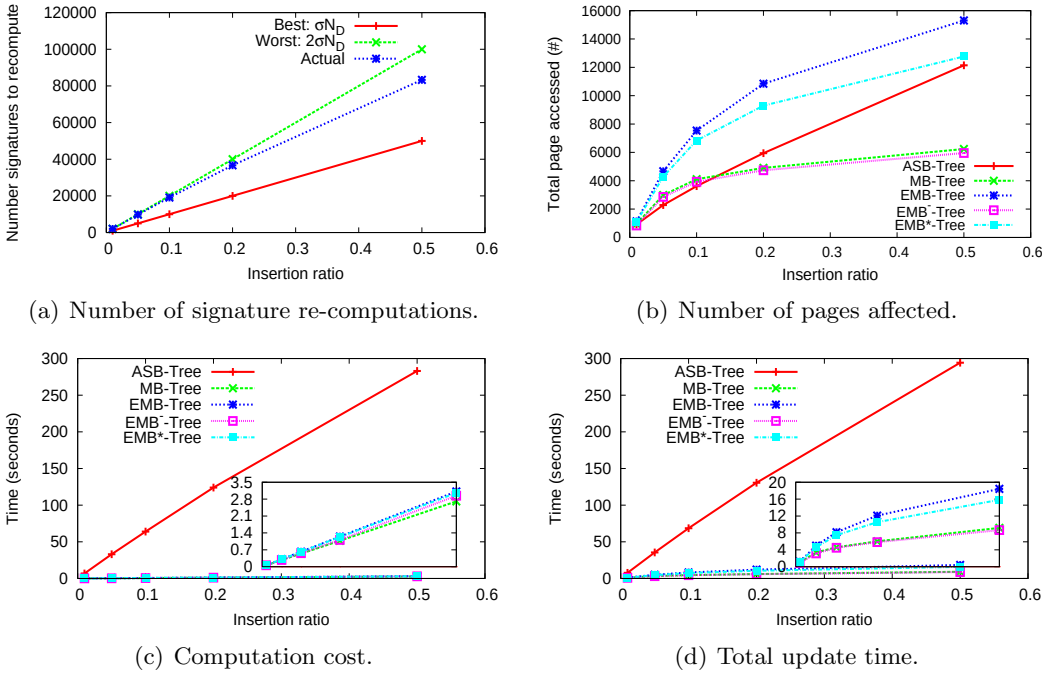
Finally, we measure the $\mathcal{VO}$ size and verification cost at the client side. The results are shown in Figure 7·6. The ASB-tree, as a result of using aggregated signatures always returns only one signature independent of the result-set size. The MB-tree has to return $f_m \log_{f_m} N_D$ number of hashes plus one signature. As $f_m \gg \log_{f_m} N_D$ the fanout is the dominating factor, and since the MB-tree has a relatively large fanout, the $\mathcal{VO}$ size is large. The EMB-tree and its variants, logically work as an MB-tree with fanout f_k and hence their $\mathcal{VO}$ sizes are significantly reduced, since $f_m \gg f_k$. Notice that the EMB^*-tree has the smallest $\mathcal{VO}$ among all embedded index structures, as the searches in its embedded multi-way search trees can stop at any level of the tree, reducing the total number of hashes.

The verification cost for the ASB-tree is linear to the size of the query result-set due to



(a) $\mathcal{VO}$ size. (b) Verification time.

Figure 7·6: Authentication cost.



(a) Number of signature re-computations. (b) Number of pages affected.

(c) Computation cost. (d) Total update time.

Figure 7·7: Performance analysis for insertions.

the modular multiplication operations, resulting in the worst performance. For the other approaches the total cost is determined by the total hashes that need to be computed. Interestingly, even though the MB-tree has the largest $\mathcal{VO}$ size, it has the fastest verification time. The reason is that for verification the number of hash computations is dominated by the height of the index, and the MB-tree has much smaller height compared to the other structures.

Update Cost

There are two types of update operations, insertions and deletions. To expose the behavior of each update type, we perform experiments on update workloads that contain either insertions or deletions. Each update workload contains 100 batch update operations, where each batch operation consists of a number of insertions or deletions, ranging from a ratio $\sigma = 1\%$ to 50% of the total database size before the update occurs. Each workload contains batch operations of equal ratio. We average the results on a per batch update operation basis. Two distributions of update operations are tested. Ones that are generated uniformly at random, and ones that exhibit a certain degree of locality. We present here results for uniform insertions only for clarity, since deletions worked similarly. Skewed distributions exhibit a somewhat improved performance and have similar effects on all approaches. Finally, results shown here include only the cost of updating the structures and not the cost associated with updating the database.

Figure 7·7 summarizes the results for insertion operations. The ASB-tree requires computing between $\sigma N_D + 1$ and $2\sigma N_D$ signatures. Essentially, every newly inserted tuple requires two signature computations, unless if two new tuples are consecutive in order in which case one computation can be avoided. Since the update operations are uniformly distributed, only a few such pairs are expected on average. Figure 7.7(a) verifies this claim. The rest of the structures require only one signature re-computation.

The total number of pages affected is shown in Figure 7.7(b). The ASB-tree needs to update both the pages containing the affected signatures and the B^+-tree structure. Clearly, the signature updates dominate the cost as they are linear to the number of update operations. Other structures need to update only the nodes of the index. Trees with smaller fanout result in larger number of affected pages. Even though the EMB$^-$-tree and MB-tree have smaller fanout than the ASB-tree, they produce much smaller number of affected pages. The EMB-tree and EMB*-tree produce the largest number of affected pages. Part of the reason is because in our experiments all indexes are bulk-loaded with 70% utilization and the update workloads contain only insertions. This will quickly lead

to many split operations, especially for indexes with small fanout, which creates a lot of new pages.

Another contributing factor to the update cost is the computation overhead. As we can see from Figure 7.7(c) the ASB-tree obviously has the worst performance and its cost is order of magnitudes larger than all other indexes, as it has to perform linear number of signature computations (w.r.t the number of update operations). For other indexes, the computation cost is mainly due to the cost of hashing operations and index maintenance. Finally, as Figure 7.7(d) shows, the total update cost is simply the page I/O cost plus the computation cost. Our proposed structures are the clear winners. Finally the communication cost incurred by update operations is equal to the number of pages affected.

7.1.3 Authentication Overhead

Based on the previous experiments we can easily quantify the overhead of authentication against an approach that does not provide any authentication at all. In that case, a single B+-tree will be used with the maximum fanout and small storage cost. However, as we can see from Figure 7.2(a), the size of the most space efficient authentication structures, i.e., MB-tree and EMB$^-$-tree, are only up to three times larger than the simple unauthenticated B+-tree. In terms of query cost, our techniques perform reasonably well. The authentication overhead in number of I/Os per query is about three times the unauthenticated case. Note that the CPU cost of our method is negligible due to the use of hash functions that are very efficient to compute. Finally, the update overhead depends on the number and type of updates. However, still they incur a small additional overhead, about twice the unauthenticated case.

A simple optimization can decrease the query time overhead without increasing the space overhead much. The idea is to build first a single B+-tree without any authentication information. Then, for each node of the tree, we compute the MHT of its entries and we attach (sequentially) additional pages to accommodate the hash values. Assuming that the original B+-tree has page utilization close to 70%, then we just need two more additional

pages for storing the hashes. Therefore, the space of the authenticated structure is three times the original structure. The advantage of this organization is that when we access a index node we perform a single random I/O and the additional pages that store the hash values are accessed sequentially. Since a sequential I/O is typically about 10 time faster than a random I/O, the overhead of accessing the authentication information is small. Furthermore, given that the hash computations required to construct the $\mathcal{VO}$ and do the verification are very cheap, the overall overhead of authentication in terms of time becomes very small.

7.1.4 Discussion

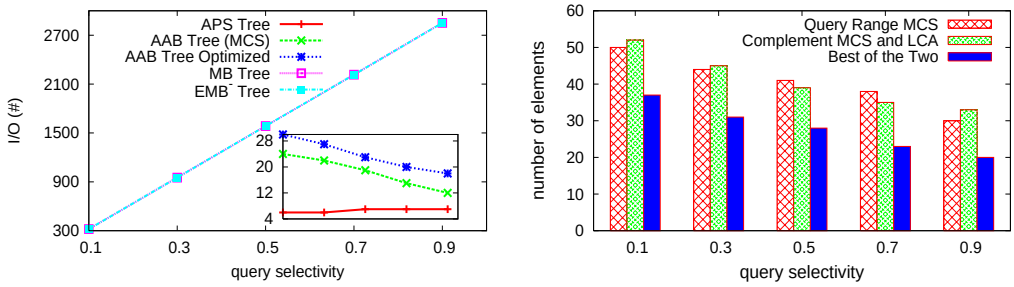
The experimental results clearly show that the novel authenticated structures proposed in this work perform better than the state-of-the-art with respect to all metrics except the $\mathcal{VO}$ size. Still, our optimizations reduced the size to four times the size of the $\mathcal{VO}$ of the ASB-tree. Overall, the EMB^--tree gives the best trade-off between all performance metrics, and it should be the preferred technique in the general case. By adjusting the fanout of the embedded trees, we obtain a nice trade-off between query (VO) size, verification time, construction (update) time and storage overhead.

7.2 Performance Evaluation for Authenticated Aggregation Index Structures

In this section we evaluate the performance of the aggregate structures with respect to query, authentication, storage, and update cost. We implemented the APS-tree, AAB-tree and AAR-tree. We also evaluated the only current alternative solution which is based on authentication structures for selection queries (see the beginning of section 4.3).

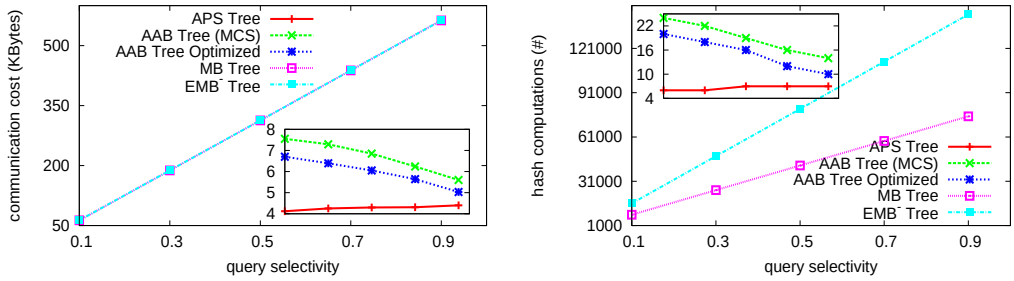
7.2.1 Setup

We use synthetic datasets for our experiments. We generate uniformly random d-dimensional tuples, with multiple sizes for the attribute domains D_i (the distribution of the data does not affect our authentication techniques). We vary the density δ of the data, where



(a) I/O. (b) $\mathcal{MCS}$ size.

Figure 7·8: One-dimensional queries. Server-side cost.



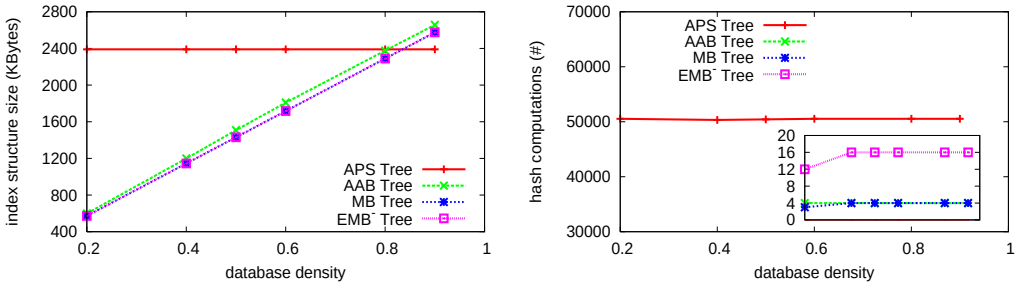
(a) Communication cost. (b) Verification cost.

Figure 7·9: One-dimensional queries. Communication and verification cost.

$\delta = \frac{N}{\prod_{i=1}^{d} M_i}$, and N the number of tuples. We also generate synthetic query workloads with one aggregate attribute A_q and up to three selection attributes. Every workload contains 100 queries, and we report averages in the plots. All experiments are performed on a Linux box with a 2.8GHz Intel Pentium4 CPU. The page size of the structures is set to be 1KByte. We use the OpenSSL [OpenSSL, 2005] and Crypto++[Crypto++ Library, 2005] libraries for hashing, verification and signing operations (SHA1 and RSA, respectively).

7.2.2 One-dimensional Queries

First, we evaluate the structures for one-dimensional queries. Candidates are the APS-tree, the AAB-tree, and the structures for selection queries, the MB-tree and the EMB$^-$-tree. For the naive approaches in order to authenticate a query, first we answer the range query and report all values to the client, which then reconstructs the result. For the

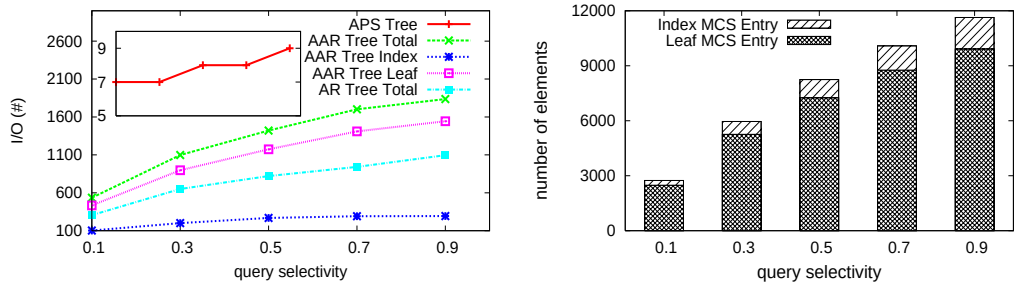


(a) Storage cost (b) Update cost

Figure 7·10: One-dimensional queries. Storage and update cost.

AAB-tree we evaluate both the structure based on $\mathcal{MCS}$ and the optimization based on $\mathcal{LCA}$. We generate a database with domain size of $M = 100,000$, $N = \delta M$ and vary the density of the database $\delta \in [0.1, 0.9]$, as well as the query selectivity $\rho \in [0.1, 0.9]$. Figure 7.8(a) shows the I/O cost at the server side. The best structure overall is the APS-tree, since it only needs to verify two PS entries, with a cost proportional to the height of the tree. The naive approaches are one to two orders of magnitude more expensive than the other techniques. For the AAB-trees it is interesting to note that the optimized version has slightly higher query I/O, due to the extra point queries that are needed for retrieving the query range boundaries. In addition, for both AAB-tree approaches the cost decreases as queries become less selective, since the $\mathcal{MCS}(Q)$ (and its complement range's $\mathcal{MCS}(\overline{Q}) \cup \mathcal{LCA}(Q)$) become smaller due to larger aggregation opportunities. This is clearly illustrated in Figure 7.8(b) that shows the actual sizes of $\mathcal{MCS}(Q)$, $\mathcal{MCS}(\overline{Q}) \cup \mathcal{LCA}(Q)$, and the best of the two for the average case over 100 queries. Figures 7.9(a) and 7.9(b) show the communication cost and verification cost at the client side. For the communication cost we observe similar trends with the $\mathcal{MCS}$ size, since the size of the $\mathcal{VO}$ is directly related to the size of $\mathcal{MCS}$. Notice also that the optimized version has smaller communication cost. The same is true for the verification cost for the APS-tree and AAB-tree. For the naive approaches, the communication and verification costs increase linearly with the query selectivity, since they have to return ρN number of aggregate values and keys.

The query efficiency of the APS-tree is achieved with the penalty of high storage and



(a) I/O (b) $\mathcal{MCS}$ size

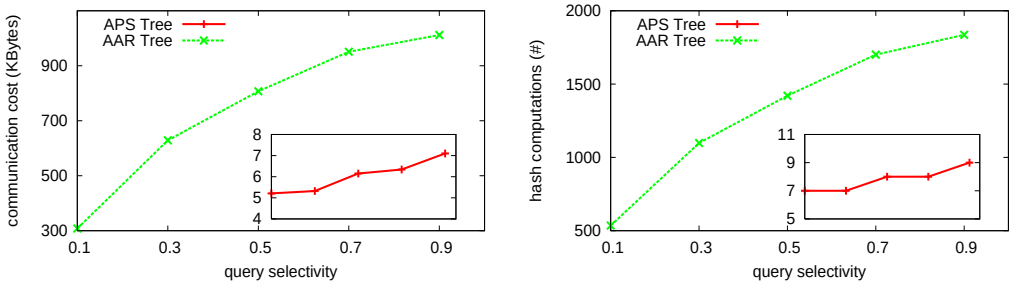
Figure 7·11: 3-dimensional queries. Server-side cost.

update cost. This is indicated in Figures 7.10(a) and 7.10(b). For this set of experiments we vary the database density δ. The storage cost of the APS-tree depends only on the domain sizes and is not affected by δ. The other approaches have storage that increases linearly with δ. Notice that, as expected, for very dense datasets all the trees have comparable storage cost. For the update experiment, we generate uniformly at random 100 updates and report the average. The update cost is measured in number of hash computations required. The APS-tree has to update half of the data entries on average. For the AAB-tree and MB-tree the update cost is bounded by the height of the tree. The EMB$^-$-tree has slightly increased cost due to the embedded trees stored in every node. Not surprisingly, the AAB-tree is orders of magnitude better than the APS-tree in terms of update cost.

7.2.3 Multi-dimensional Queries

In this section we compare the APS-tree and the AAR-tree approaches for 3-dimensional queries. To obtain similar database sizes with the 1-dimensional experimental evaluation, we generate synthetic datasets with maximum unique elements of $\prod_{i=1}^{3} M_i = 125,000$ tuples, query workloads with 100 queries with varying values of ρ, and database density $\delta = 0.8$. The storage and update costs are studied for databases with varying density values.

The I/O cost of the structures as a function of query selectivity is reported in Figure 7.11(a). The APS-tree once again has the best performance overall, since it only needs

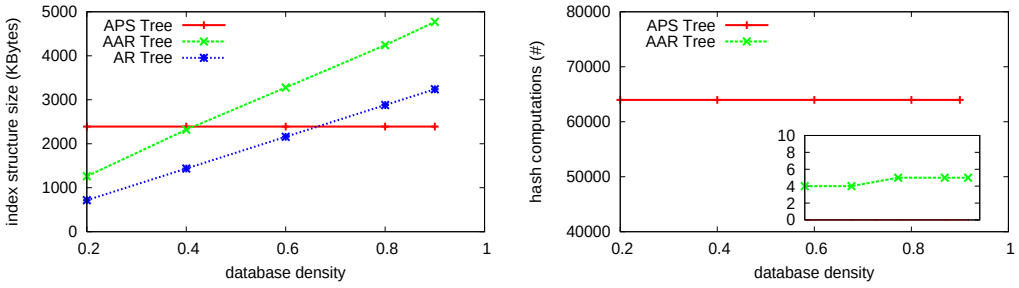


(a) Communication cost (b) Verification cost

Figure 7·12: 3-dimensional queries. Communication and verification cost.

to authenticate eight PS elements. The AAR-tree has much higher I/O since it needs to construct the $\mathcal{MCS}$ which requires traversing many different paths of the R-tree structure, due to multiple MBR overlaps at every level of the tree. Notice also that the I/O of the AAR-tree increases as queries become less selective, since larger query rectangles result into a larger number of leaf node MBRs included in the $\mathcal{MCS}$ as indicated in Figure 7.11(b). This becomes clear in Figure 7.11(a) which shows that most I/O operations come from the leaf level of the tree. The authentication and verification costs are shown in Figure 7·12. The APS-tree has small communication cost and verification cost, both bounded by eight times the height of the tree; notice that our algorithm by merging the common paths has reduced the costs from this worst case bound. The AAR-tree has much larger communication cost due to larger $\mathcal{MCS}$ sizes and because it has to return the MBRs of all $\mathcal{MCS}$ entries. The verification costs follow similar trends, since the number of hash computations is directly related to the number of entries in the $\mathcal{MCS}$.

Figure 7.13(a) shows the storage cost as a function of δ, for the APS-tree and AAR-tree. In multiple dimensions the AAR-tree consumes more space when the database becomes relatively dense, in contrast to the one-dimensional case. This is due to the fact that the AAR-tree has to store the 3-dimensional MBRs for all nodes. In our experiments we use 4-byte floating point numbers and a small page size, but the trend is indicative. Figure 7.13(b) plots the update cost as a function of δ. The superior query cost of the APS-tree is offset by its extremely high update cost. For 100 updates generated uniformly at random,



(a) Storage cost (b) Update cost

Figure 7·13: 3-dimensional queries. Storage and update cost.

regardless of database density the APS-tree has to update half of the data entries on average. In contrast, the AAR-tree inherits the good update characteristics of the R-tree.

7.2.4 Authentication Overhead

Again, we compare the cost of the authenticated version of the aggregation methods against the non-authenticated ones. As we can see, the additional storage overhead of the AAR-tree is small compared to the AR-tree. In Figure 7.13(a) we see that the extra storage overhead due to authentication is half the storage cost of the AR-tree. Furthermore, the query cost of the AAR-tree is less than twice the cost of the AR-tree. Note that the APS-tree has smaller space overhead than the AR-tree for dense datasets. As the dimensionality increases, we expect that the extra cost of authentication will become even smaller. Overall, the authentication overhead for the aggregation case is proportionally smaller than the case of selection and projection queries that we discussed before.

7.2.5 Discussion

In general, our proposed methods have multiple orders of magnitude smaller query cost with almost the same storage and update cost as the existing approaches. Among the new techniques proposed, the APS-tree has very small (constant) query cost but expensive updates, and considerable space overhead in the case of sparse datasets. However, it works the best for non-sparse datasets and naturally supports both one-dimensional and multi-

dimensional queries. In addition, it works with encrypted data as shown in Section 4.5. The AAB-tree and AAR-tree have higher query cost but better space usage, especially for sparse datasets, and superior update cost.

7.3 Experiments Evaluation for Authenticating Sliding Window Queries on Data Streams

We implemented the proposed techniques and evaluated their performance over two real data streams [Arlitt and Jin, 1998, AT&T, 2005]. The following cost metrics are considered: 1. the amortized update cost per tuple for the data owner; 2. the storage cost of the authentication structure for the server; 3. the query cost; 4. the $\mathcal{VO}$ size; and 5. the verification cost for the client.

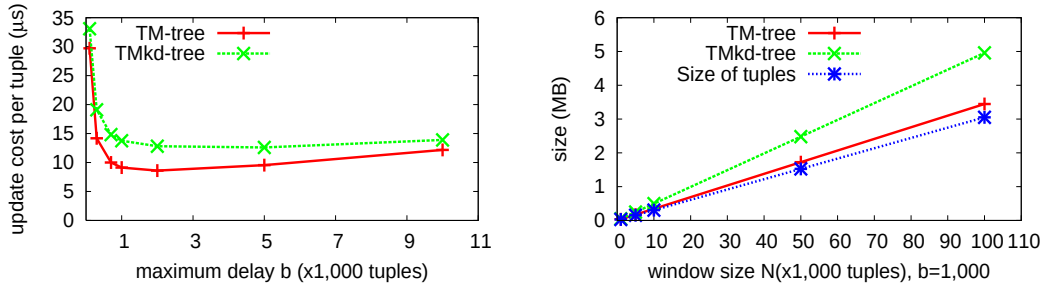
All algorithms are implemented using GNU C++. Cryptographic functions are provided by [Crypto++ Library, 2005, OpenSSL, 2005]. Two real data streams have been tested. The *World Cup (WC)* data stream [Arlitt and Jin, 1998] consists of web server request traces for the 1998 Soccer World Cup. Each request contains attributes such as a timestamp, a client id, a requested object id, a response size, etc. We used the request streams of days 46 and 47 that have about 100 millions records. The *IP traces (IPs)* data stream [AT&T, 2005] is collected over the AT&T backbone; each tuple is a TCP/IP packet header. Since similar patterns have been observed for all cost metrics for both data streams, we present results from the WC data only. We use tuples consisting of attributes response size, object id, and client id, and a unique timestamp. Experiments were run on a Linux box with an Intel Pentium 2.8GHz CPU. The SHA1 hash function takes about $1 \sim 2\mu s$ (for input size up to 500 bytes); 128-byte RSA has a signing cost of about $2ms$ and verifying cost of 120μs. Each hash value produced by SHA1 is 20 bytes and a signature from RSA is 128 bytes.

7.3.1 Sliding window queries.

The TM-tree and the TMkd-tree are the two candidates for authenticating sliding window queries. We compare them using one-dimensional queries on the "response size" attribute (as the TM-tree does not support multi-dimensional queries). Since kd-tree requires its indexed data having distinct values (see Section 5.3.1), we perturb the attribute "response size" so that all tuples have unique values. In addition, in order to easily generate a set of random queries with a fixed query selectivity, tuples are perturbed so that they have uniformly distributed values in "response size". Our performance study is not affected by this, as the query cost is solely determined by the query selectivity.

Update cost per tuple. The data owner has to maintain the authenticated data structures as tuples are being produced. The data streaming setting mandates this cost to be small. Figure 7.14(a) shows the amortized update cost per tuple for both trees over different values of b (the maximum delay, which is determined by the owner). We notice that for small b both trees have an excessive update cost; as b increases, the cost drops quickly. After some point both start to grow slowly, due to the fact that the cost consists of an $O(1/b)$ signing and an $O(\log b)$ update cost (see Table 5.1). For small b, signing is the dominant cost; when b exceeds a certain threshold, the latter begins to dominate. Figure 7.14(a) reveals that for both trees $b = 1,000$ is a sweet point. For smaller b's the update cost is too high to cope with bursty streams, while larger b's introduce longer response delays for the clients without further reducing the data owner's cost significantly. With $b = 1,000$ the amortized update cost per tuple is only $10 \sim 15\mu s$, i.e., both the owner and the server could handle 10^5 tuples per second.

Structure size. Our analysis has pointed out that both TM-tree and TMkd-tree use linear space given a window size N. It is still interesting to investigate the constant factors associated with this cost. Figure 7.14(b) plots the results. It should be noted that the storage cost does not depend on b. The size of the raw data (32 bytes per tuple) is also provided in Figure 7.14(b) as a baseline for comparison. Both trees have very



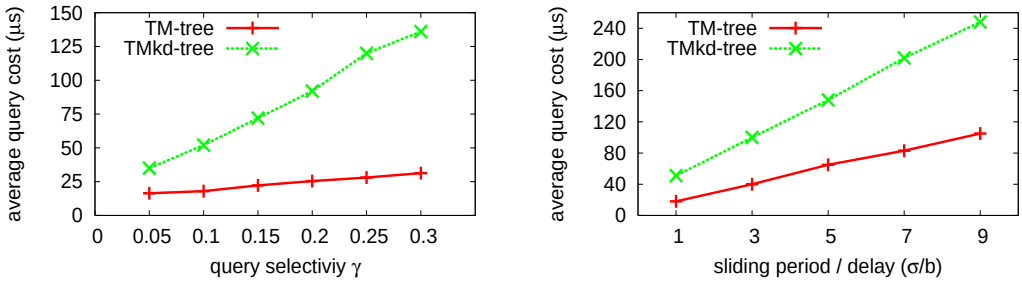
(a) Update cost (b) Tree size

Figure 7·14: Tumbling trees: update cost and size.

good scalability and introduce very small overhead in size (only 5 MB for 100,000 tuples). Another interesting fact to highlight is that at any time instance, the provider only needs space large enough to store one Merkle tree (or Mkd-tree), built for the latest b tuples. This has size roughly equal to 40 KB for $b = 1,000$.

Query cost. We turn our attention to the per period update cost for sliding window queries. The performance of one-shot window queries (or equivalently the initialization cost for a new sliding window query) will be studied in Section 7.3.2. The query cost is measured for two different workloads of $1,000$ randomly generated queries: 1. fixed sliding period σ but varying query selectivity γ (Figure 7.15(a)); and 2. fixed query selectivity γ but varying sliding period σ (Figure 7.15(b)). We set b to $1,000$ as suggested before and report the average cost for one query. Note that γ is essentially equal to the selectivity on the query attribute dimension, since most tuples in a window, except those in the boundary trees when window slides, are indexed by trees that are fully covered in the time dimension by the query window.

With the sliding period $\sigma = b = 1,000$, four boundary trees will be queried to report the new and expiring tuples. From the results we observe that both TM-tree and TMkd-tree have roughly linearly increasing costs w.r.t query selectivity. TM-tree does have a lower query cost even though theoretically it may access more nodes than TM-kd tree due to the fact that it incurs false positives. This is explained by the fact that a balanced binary search tree is used as the underlying structure for a single tree in TM-tree, which means

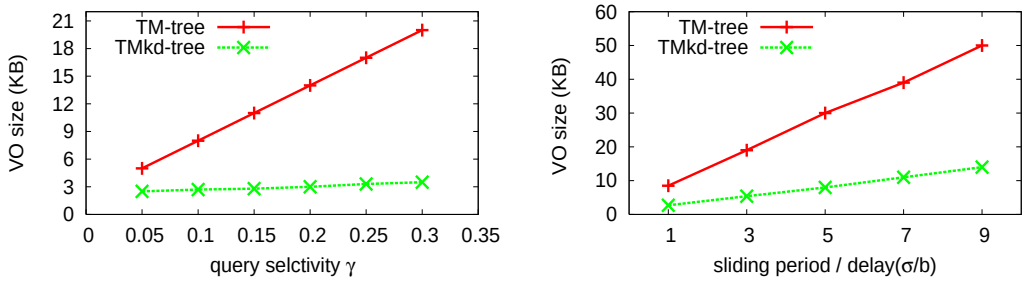


(a) $\sigma = 1,000$ (b) $\gamma = 0.1$

Figure 7·15: $b = 1,000$, Query cost per period.

that all tuples are retrieved in a sequential scan in the leaf level. In addition, since trees are bulk-loaded there will be strong locality among these leaf nodes in the main memory. Adjacent nodes are loaded into the cache in one memory access and this dramatically reduces the number of memory access. On the other hand, though TMkd-tree accesses less number of nodes theoretically by avoiding any false positives, but they are all random access which in practice leads to more nodes access and slower performance. When the sliding period σ is larger than b, around $2\frac{\sigma}{b} + 2$ trees will be queried. Hence, the query cost is roughly linear in σ as we have observed in Figure 7.15(b). Finally, since we only retrieve the new and expiring tuples as the window slides, the sliding window query cost does not depend on the window size n.

$\mathcal{VO}$ size. The $\mathcal{VO}$ size is the determining factor for the communication overhead between the server and the client. In Figure 7·16 we plot the $\mathcal{VO}$ size using the same queries as in Figure 7.15(a) and 7.15(b). Figure 7.16(a) reveals that the TM-tree has a much higher $\mathcal{VO}$ size than the TMkd-tree, as it will incur roughly $4\gamma b$ false positives in this case (see Figure 5·1). Recall that when $\sigma = b$, four boundary trees will be queried. On the other hand, the TMkd-tree can avoid false positives as it indexes and stores authentication information for both the selection attribute and the time axis. The difference can be order of magnitude as the query selectivity increases. This is due to the fact that the TM-tree generates false positives, which are part of the $\mathcal{VO}$ and each false positive is a tuple (32 bytes in our experiment). Similarly, the linear trend w.r.t σ in Figure 7.16(b) for both trees is explained



(a) $\sigma = 1,000$ (b) $\gamma = 0.1$

Figure 7·16: $b = 1,000$, $\mathcal{VO}$ size per period.

by the same reason as in Figure 7.15(b). Again, the sliding window size n does not affect the results.

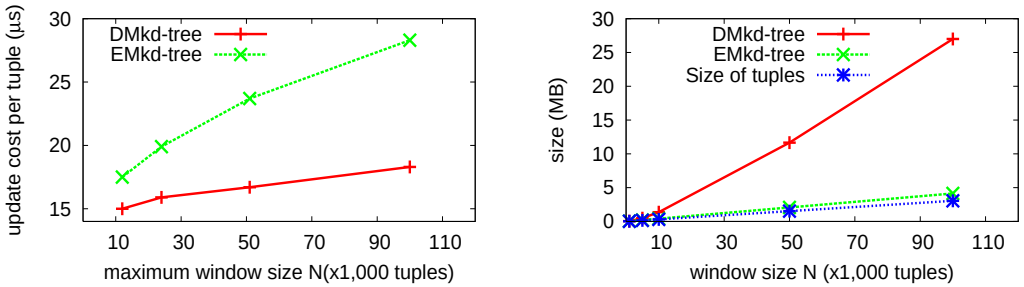
Verification cost. The verification cost at the client is a mirror of the query process performed by the server, except for the additional hashing operations to reconstruct the hashes of the roots, and the verification of the digital signatures. Since these costs are common to both the TM-tree and the TMkd-tree, similar trends as in Figure 7·15 have been observed for the verification cost. We omit the details.

Experimental conclusion. For sliding window queries, TM-tree has better query cost and TMkd-tree outperforms TM-tree with respect to the $\mathcal{VO}$ size. Nevertheless, TMkd-tree naturally supports multi-dimensional queries while TM-tree can not handle those cases.

7.3.2 One-shot queries

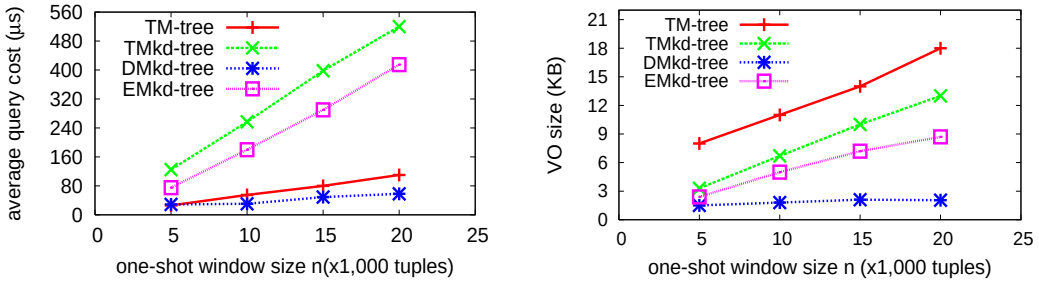
Both TM-tree and TMkd-tree have linear cost in n (Table 5.1) for one-shot queries. In this section, we study the performance of the DMkd-tree and the EMkd-tree.

Amortized update cost and structure size. First we study the amortized per-tuple update cost. The result is shown in Figure 7.17(a) with varying maximum window sizes N. Both trees have higher update cost compared to the TM-tree and TMkd-tree (see Figure 7.14(a)) due to the $O(\log N)$ dependence (comparing to the $O(\log b)$ cost for tumbling trees). In addition, the EMkd-tree is more expensive than the DMkd-tree as it has



(a) Update cost (b) Tree size

Figure 7·17: Update cost and size for DMkd-tree and EMkd-tree, $b = 1,000$.



(a) Query cost (b) $\mathcal{VO}$ size

Figure 7·18: One-shot window query $b = 1,000$, $\gamma = 0.1$ and $N = 20,000$.

an additional factor of $O(\log \frac{N}{b})$. Nevertheless, for fairly large N (up to 100,000 in our experiments), both DMkd-tree and EMkd-tree achieve update costs of less than 30μs per tuple. The smaller update cost of DMkd-tree is not for free. The tree occupies more space, as shown in Figure 7.17(b), by an $O(\log \frac{N}{b})$ factor compared to the EMkd-tree. EMkd-tree utilizes linear space that is almost equal to the raw data size. It should be highlighted though that the DMkd-tree still has a reasonable main memory size. As Figure 7.17(b) has suggested, it takes less than 30 MB of memory for the maximum window of 100,000 tuples.

Query cost and $\mathcal{VO}$ size. Our theoretical analysis shows that the DMkd-tree and EMkd-tree should perform better than the TM-tree and TMkd-tree respectively, especially for large window sizes. This is confirmed by our findings from Figure 7.18(a) (where N is set to 20,000 and $\gamma = 0.1$). The results are obtained by averaging over a workload of 1,000

randomly generated queries. Clearly, the DMkd-tree and EMkd-tree outperforms their counterparts TM-tree and TMkd-tree. The gaps between them are increasing with the larger value for n. This saving is critical when there are multiple clients registering many queries in the system. Lastly, we study the $\mathcal{VO}$ size for various trees under one-shot window queries and present the results in Figure 7.18(b). The TM-tree has the worst performance since it has to include roughly $2\gamma b$ tuples as false positives in the two boundary trees. Both the DMkd-tree and EMkd-tree have less $\mathcal{VO}$ size than the TM-kd tree.

Verification cost. Following the discussion in Section 7.3.1 for the verification cost, similar trends as those reflected in the query cost for one-shot window queries have been observed. The results are omitted for the brevity.

Experimental conclusion. Our results reveal that the DMkd-tree and EMkd-tree are good candidates for answering one-shot window queries. For one dimensional query, the data owner and the server could combine either the DMkd-tree, if reducing update and query cost is a higher priority, or the EMkd-tree, if reducing the space usage is a higher priority, together with the TM-tree, if the efficient query cost for maintaining the updates to sliding window queries is a higher priority, or the TMkd-tree if low $\mathcal{VO}$ size for maintaining the updates to sliding window queries is a higher priority. This gives the data owner and the server flexible choices to answer all queries efficiently in different settings.

7.3.3 Aggregation and multi-dimensional queries

All of the proposed structures in this thesis for authenticating sliding window queries on data streams can support authentication of aggregation queries as we have discussed. The detailed evaluation of the performance under different cost metrics is rather involved and they are not discussed here. However, as Table 5.1 has suggested, the cost analysis remains almost the same for aggregation queries. We would like to highlight that all of our index structures could be made to support the authentication of selection and aggregation queries at the same time, as the trees for authenticating aggregation queries trivially support

answering and authenticating selection queries.

Finally, the TMkd-tree and EMkd-tree support multi-dimensional queries. The combination of these two structures provide a nice treatment for a highly dynamic environment where multiple clients register various sliding window queries at any time instance, possibly with different dimensionality, window sizes and sliding periods.

”

7.4 Empirical Evaluation for Query Execution Assurance on Data Streams

In this section we evaluate the performance of the proposed synopses over two real data streams [Arlitt and Jin, 1998, AT&T, 2005]. The experimental study demonstrates that our synopses: 1. use very small space; 2. support fast updates; 3. have very high accuracy; 4. support multiple queries; and 5. are easy to implement.

7.4.1 Setup

Our synopses are implemented using GNU C++ and the GNU GMP extension which provides arbitrary precision arithmetic, useful for operating on numbers longer than 32 bits. The experiments were run on an Intel Pentium 2.8GHz CPU with 512KB L2 cache and 512MB of main memory. Our results show that using our techniques, even a low-end client machine can efficiently verify online queries with millions of groups on real data streams.

The *World Cup (WC)* data stream [Arlitt and Jin, 1998] consists of web server logs for the 1998 Soccer World Cup. Each record in the log contains several attributes such as a timestamp, a client id, a requested object id, a response size, etc. We used the request streams of days 46 and 47 that have about 100 millions records. The *IP traces (IPs)* data stream [AT&T, 2005] is collected over the AT&T backbone network; each tuple is a TCP/IP packet header. Here, we are interested in analyzing the source IP/port, destination IP/port, and packet size header fields. The data set consists of a segment of one day traffic and has 100 million packets. Without loss of generality, unless otherwise

	WC	IPs
Count	0.98 μs	0.98 μs
Sum	8.01 μs	6.69 μs

Table 7.1: Average update time per tuple.

stated, we perform the following default query: 1. Count or Sum (on response size) query group-by client id/object id for the WC data set; 2. Count or Sum (on packet size) query group-by source IP/destination IP for the IPs data set. Each client id, object id, IP address, the response size, or the packet size is a 32-bit integer. Thus, the group id is 64-bit long (by concatenating the two grouping attributes), meaning a potential group space of $n = 2^{64}$. The number of nonzero groups is of course far lower than n: WC has a total of 50 million nonzero groups and IPs has 7 million nonzero groups.

7.4.2 PIRS

A very conservative upper bound for the total response size and packet size is $m = 10^{10} \ll n \approx 2 \times 10^{19}$ for all cases in our experiments. So from our analysis in Section 6.3, PIRS-1 is clearly the better choice, and is thus used in our experiments. We precomputed p as the smallest prime above 2^{64} and used the same p throughout this section. Thus, each word (storing p, α, and $\mathcal{X}(\mathbf{v})$) occupies 9 bytes.

Space usage. As our analysis has pointed out, PIRS uses only 3 words, or 27 bytes for our queries. This is in contrast to the naïve solution of keeping the exact value for each nonzero group, which would require 600MB and 84MB of memory, respectively.

Update cost. PIRS has excellent update cost which is crucial to the streaming algorithm. The average per-tuple update cost is shown in Table 7.1 for Count and Sum queries on both WC and IPs. The update time for the two count queries stays the same regardless of the data set, since an update always incurs one addition, one multiplication, and one modulo. The update cost for sum queries is higher, since we need $O(\log u)$ time for exponentiation. The cost on WC is slightly larger as its average u is larger than that of IPs. Nevertheless,

PIRS is still extremely fast in all cases, and is able to process more than 10^5 tuples (10^6 tuples for count queries) per second.

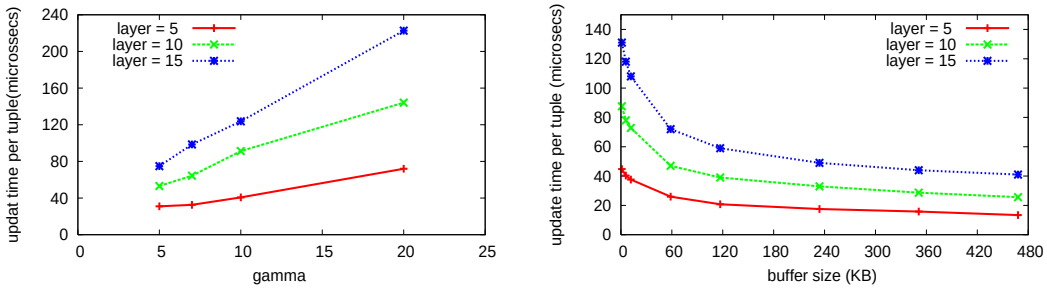
Detection accuracy. As guaranteed by the theoretical analysis, the probability of failure of PIRS-1 is $\delta \leq m/p$, which is at most 0.5×10^{-9}. Note that our estimate of m is very conservative; the actual δ is much smaller. We generated $100,000$ random attacks and, not surprisingly, PIRS identified all of them.

7.4.3 PIRS$^{\gamma}$ and PIRS$^{\pm\gamma}$

For the rest of the experiments, we focus on the Count query on the WC data set. Similar patterns have been observed on the IPs data set.

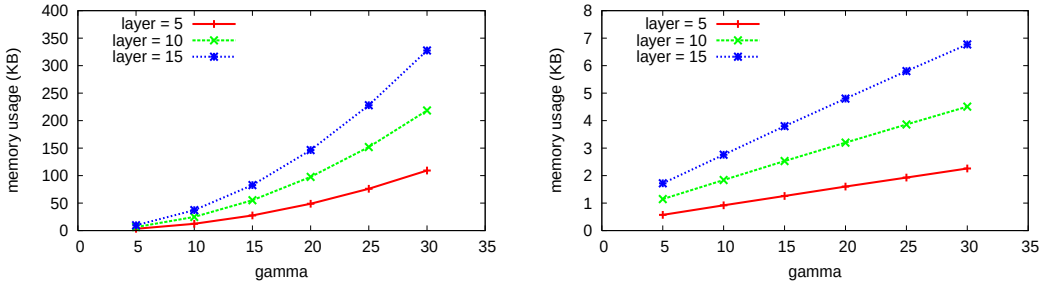
Update cost. In this set of experiments we study the performance of PIRS$^{\gamma}$ and PIRS$^{\pm\gamma}$. Clearly, PIRS$^{\gamma}$ has linear update cost w.r.t the number of layers and γ (the number of inconsistent groups to detect), as confirmed in Figure 7.19(a). It is not hard to see that PIRS$^{\gamma}$ and PIRS$^{\pm\gamma}$ have almost the same update cost if they are configured with the same number of layers. Essentially, each one has to generate the γ-wise (or γ^+-wise) independent random numbers on-the-fly and update one PIRS synopsis at each layer. Hence, we only show the cost for PIRS$^{\gamma}$. However, the space cost of the two synopses is different. PIRS$^{\gamma}$, as an exact solution for CQV$^{\gamma}$, is expected to use much larger space than its counterpart PIRS$^{\pm\gamma}$, which gives approximate solutions. This is demonstrated in Figure 7·20. By construction, at each layer PIRS$^{\gamma}$ has $O(\gamma^2)$ and PIRS$^{\pm\gamma}$ $O(\frac{\gamma}{\ln \gamma})$ buckets, which is easily observed in Figure 7.20(a) and 7.20(b) respectively.

Space/Time Trade-offs. If the client can afford to allocate some extra space, but still cannot store the entire vector $\mathbf{v}$, as discussed in Section 6.6, it is possible to exploit the locality in the input data streams to reduce the amortized update cost. A simple LRU buffer has been added to PIRS$^{\gamma}$ and PIRS$^{\pm\gamma}$ and its effect on update cost is reported in Figure 7.19(b) with $\gamma = 10$. Again, both synopses exhibit very similar behavior. As the figure indicates, a very small buffer (up to 500 KB) that fits into the cache is able to



(a) Update cost per tuple. (b) Space time tradeoff.

Figure 7·19: PIRS$^{\gamma}$,PIRS$^{\pm\gamma}$: running time.

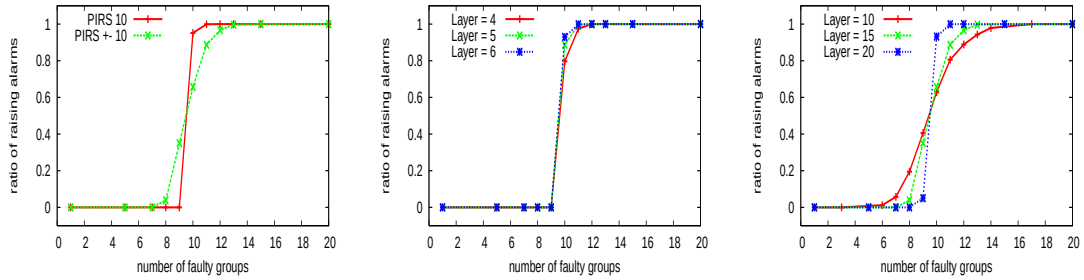


(a) PIRS$^{\gamma}$. (b) PIRS$^{\pm\gamma}$.

Figure 7·20: PIRS$^{\gamma}$, PIRS$^{\pm\gamma}$: memory usage.

reduce the update cost by an order of magnitude. The improvement on the update cost of this buffering technique depends on the degree of locality in the data stream. Note that if this simple buffering technique is still insufficient for achieving the desired update speed (e.g., when there is very little locality in the data stream or γ is large) we could use the technique of [Siegel, 1989] to reduce the cost to $O(\log \frac{1}{\delta})$ (independent of γ) by using extra space, but generating random numbers much faster.

Detection accuracy. We observed that both of our synopses can achieve excellent detection accuracy as the theoretical analysis suggests. All results reported here are the ratios obtained from 100, 000 rounds. Since the detection mechanism of the synopses does not depend on the data characteristics, both data sets give similar results. Figure 7.21(a) shows the ratios of raising alarms versus the number of actual inconsistent groups, with $\gamma = 10$ and 10 layers. As expected, PIRS$^{\gamma}$ has no false positives and almost no false negatives; only very few false negatives are observed with 10 and 11 actual inconsistent groups. On



(a) $\gamma = 10, \ell = 15$. (b) PIRS$^{\gamma}$, $\gamma = 10$. (c) PIRS$^{\pm\gamma}$, $\gamma = 10$.

Figure 7·21: Detection with tolerance for limited number of errors.

# queries	5	10	15	20
update time (μs)	5.0	9.9	14.9	19.8
memory usage (bytes)	27	27	27	27

Table 7.2: Update time and memory usage of PIRS for multiple queries.

the other hand, PIRS$^{\pm\gamma}$ has a transition region around γ and it does have false positives. Nevertheless, the transition region is sharp and once the actual number of inconsistent groups is slightly off γ, both false positives and negatives reduce to zero. We have also studied the impact of the number of layers on the detection accuracy. Our theoretical analysis gives provable bounds. For example with PIRS$^{\gamma}$ the probability of missing an alarm is at most $1/2^{\ell}$ (for ℓ layers). In practice, the probability is expected to be even smaller. We repeated the same experiments using different layers, and Figure 7.21(b) reports the result for PIRS$^{\gamma}$. With less layers (4–6) it still achieves excellent detection accuracy. Only when the number of actual inconsistent groups is close to γ, a small drop in the detection ratio is observed. Figure 7.21(c) reports the same experiment for PIRS$^{\pm\gamma}$ with layers from 10 to 20. Smaller number of layers enlarges the transition region and larger number of layers sharpens it. Outside this region, 100% detection ratio is always guaranteed. Finally, experiments have been performed over different values of γ's and similar behavior has been observed.

7.4.4 Multiple Queries

Our final set of experiments investigates the effect of multiple, simultaneous queries. Without loss of generality, we simply execute the same query a number of times. Note that the same grouping attributes with different query ids are considered as different groups. We tested with 5, 10, 15, and 20 queries in the experiments. Note that on the WC data set, the exact solution would use 600MB for each query, hence 12GB if there are 20 queries. Following the analysis in Section 6.6, our synopses naturally support multiple queries and still have the same memory usage as if there were only one query. Nevertheless, the update costs of all synopses increase linearly with the number of queries. In Table 7.2 we report the update time and memory usage for PIRS; similar results were observed for PIRS$^{\gamma}$ and PIRS$^{\pm\gamma}$.

Chapter 8

Conclusion

Query authentication is a fundamental problem in the data publishing model. The essential challenge is to introduce security guarantees while preserving the query efficiency, especially, for large scale datasets. This dissertation has made significant progress towards that goal. The key idea throughout this thesis is to adapt advanced techniques from other fields, such as cryptography and fingerprinting, into the process of data management without dramatically increasing the system cost and sacrificing the query performance. We realize this idea by designing algorithms that work extremely well in practice and bounding their performance with rigorous theoretical analysis.

More specifically, our study has revealed novel structures, featuring with low maintainable cost and minimal query overhead, for authenticating range and aggregation queries for both relational and streaming databases. This dissertation also fills in the gap for an important variation of the query authentication problem, namely query execution assurance on data streams. Rather than relying on random sampling techniques, as previous work has advocated for off-line, relational databases, which evidently bounds the success rate to the sampling ratio, this thesis introduces a complete different approach based on fingerprinting techniques. This technique achieves success rates that are almost always equal to 1. Furthermore, we prove the optimality of the fingerprinting technique in terms of space and accuracy.

All these are possible while consuming very limited amount of system resources in terms of both memory usage and CPU computation. Another notable contribution of this thesis is the extensive experimental evaluation for all proposed techniques. We have implemented a working prototype for empirical evaluations and our implementation has demonstrated

the superior practicality of the proposed solutions.

Looking beyond the scope of this thesis, the present work introduced several important new problems, such as, query authentication and query execution assurance over data streams. Our discussion has been concentrated on range and aggregation queries. However, our long term goal is to efficiently proof-infuse a database for all possible queries. Therefore, an important open problem is how to design and extend existing solutions for authenticating complex relational and continuous queries with minimum overhead. The queries can consist multiple multi-way or binary joins, projections, selection and group by operations with aggregation. As a start we can investigate simple join queries that have not been studied in depth yet and OLAP (online analytical processing) operations in data warehouses.

Another interesting problem that remains open is to authenticate only a subset of the returned query results. This problem has many compiling reasons to be an ideal candidate for future research. For example, it will enable clients to utilize results from a query workload when a subset of the queries or a portion of results for one query has been contaminated. The current methods do not achieve that; when the authentication of a query result fails, the complete answer is discarded.

List of Abbreviations for Conferences, Journals and Workshops

ASIAN	.....	Asian Computing Science Conference
CACM	.....	Communication of ACM
CCS	.....	Conference on Computer and Communication Security
CIDR	.....	Conference on Innovative Data Systems Research
CIKM	.....	Conference on Information and Knowledge Management
CRYPTO	.....	International Cryptology Conference
CT-RSA	.....	The Cryptographers' Track within the RSA Conference
DBSec	.....	Conference on Data and Applications Security
ESORICS	.....	European Symposium on Research in Computer Security
FOCS	.....	Symposium on Foundations of Computer Science
ICDE	.....	International Conference on Data Engineering
IH	.....	International Workshop on Information Hiding
ISC	.....	Information Security Conference
JACM	.....	Journal of ACM
NDSS	.....	Network and Distributed System Security Symposium
OSDI	.....	Operating Systems Design and Implementation
PODS	.....	Principal of Database Systems
RANDOM	.....	Workshop on Randomization and Approximation Techniques
SIGMOD	.....	International Conference on Management of Data
SODA	.....	ACM-SIAM Symposium on Discrete Algorithms

SSDBM	.....	Statistical and Scientific Database Management
STOC	.....	Symposium on Theory of Computing
TKDE	.....	Transaction of Knowledge and Data Engineering
TODS	.....	Transaction of Database Systems
VLDB	.....	Very Large Databases Conference
VLDBJ	.....	International Journal on Very Large Databases

References

[Abadi et al., 2005] Abadi, D., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The Design of the Borealis Stream Processing Engine. In *CIDR*.

[Agrawal et al., 2004] Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y. (2004). Order preserving encryption for numeric data. In *SIGMOD*, pages 563–574.

[Agrawal and Srikant, 2000] Agrawal, R. and Srikant, R. (2000). Privacy-preserving data mining. In *SIGMOD*, pages 439–450.

[Agrawal et al., 2005] Agrawal, R., Srikant, R., and Thomas, D. (2005). Privacy preserving OLAP. In *SIGMOD*, pages 251–262.

[Alon et al., 1996] Alon, N., Matias, Y., and Szegedy, M. (1996). The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29.

[Anagnostopoulos et al., 2001] Anagnostopoulos, A., Goodrich, M., and Tamassia, R. (2001). Persistent authenticated dictionaries and their applications. In *Information Security Conference (ISC)*, pages 379–393.

[Arasu et al., 2003] Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J. (2003). Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1):19–26.

[Arasu and et al., 2003] Arasu, A. and et al. (2003). Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1).

[Arasu and Manku, 2004] Arasu, A. and Manku, G. S. (2004). Approximate counts and quantiles over sliding windows. In *PODS*, pages 286–296.

[Arlitt and Jin, 1998] Arlitt, M. and Jin, T. (1998). http://www.acm.org/sigcomm/ITA/. ITA, 1998 World Cup Web Site Access Logs.

[AT&T, 2005] AT&T (2005). AT&T network traffic streams. AT&T Labs.

[Babcock et al., 2002] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *PODS*.

[Babcock et al., 2004] Babcock, B., Datar, M., and Motwani, R. (2004). Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361.

[Babcock et al., 2003] Babcock, B., M.Datar, Motwani, R., and O'Callaghan, L. (2003). Maintaining variance and k-medians over data stream windows. In *PODS*, pages 234–243.

[Bar-Yossef et al., 2002] Bar-Yossef, Z., Jayram, T. S., Kumar, R., Sivakumar, D., and Trevisan, L. (2002). Counting distinct elements in a data stream. In *RANDOM*, pages 1–10.

[Bellare et al., 1994] Bellare, M., Goldreich, O., and Goldwasser, S. (1994). Incremental cryptography: The case of hashing and signing. In *CRYPTO*, pages 216–233.

[Bellare et al., 1995] Bellare, M., Guerin, R., and Rogaway, P. (1995). Xor macs: New methods for message authentication using finite pseudorandom functions. In *CRYPTO*, pages 15–28.

[Bentley, 1975] Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517.

[Bertino et al., 2004] Bertino, E., Carminati, B., Ferrari, E., Thuraisingham, B., and Gupta, A. (2004). Selective and authentic third-party distribution of XML documents. *TKDE*, 16(10):1263–1278.

[Blum and Kannan, 1995] Blum, M. and Kannan, S. (1995). Designing programs that check their work. *Journal of the ACM*, 42(1):269–291.

[Bouganim et al., 2005] Bouganim, L., Cremarenco, C., Ngoc, F. D., Dieu, N., and Pucheral, P. (2005). Safe data sharing and data dissemination on smart devices. In *SIGMOD*, pages 888–890.

[Bouganim et al., 2003] Bouganim, L., Ngoc, F. D., Pucheral, P., and Wu, L. (2003). Chip-secured data access: Reconciling access rights with data encryption. In *VLDB*, pages 1133–1136.

[Bureau, 2006] Bureau, U. C. (2006). Quarterly retail e-commerce sales. http:// www.census.gov/ mrts/ www/ ecomm.html.

[Carney et al., 2003] Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. B. (2003). Monitoring streams – a new class of data management applications. In *VLDB*, pages 215–226.

[Chan et al., 2006] Chan, H., Perrig, A., and Song, D. (2006). Secure hierarchical in-network aggregation in sensor networks. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 278–287.

[Chandrasekaran et al., 2003] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. A. (2003). Telegraphcq: continuous dataflow processing for an uncertain world. In *CIDR*.

[Chandrasekaran and et al., 2003] Chandrasekaran, S. and et al. (2003). Telegraphcq: continuous dataflow processing for an uncertain world. In *CIDR*.

[Chen et al., 2002] Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., and Jakubowski, M. H. (2002). Oblivious hashing: A stealthy software integrity verification primitive. In *Proc. of the International Workshop on Information Hiding (IH)*, pages 400–414.

[Cheng et al., 2006] Cheng, W., Pang, H., and Tan, K. (2006). Authenticating multi-dimensional query results in data publishing. In *DBSec*.

[Comer, 1979] Comer, D. (1979). The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137.

[Cormode and Muthukrishnan, 2003] Cormode, G. and Muthukrishnan, S. (2003). What's hot and what's not: tracking most frequent items dynamically. In *PODS*, pages 296–306.

[Cormode and Muthukrishnan, 2005] Cormode, G. and Muthukrishnan, S. (2005). An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75.

[Cranor et al., 2003a] Cranor, C., Johnson, T., Spatscheck, O., and Shkapenyuk, V. (2003a). Gigascope: A stream database for internet databases. In *SIGMOD*.

[Cranor et al., 2003b] Cranor, C., Johnson, T., Spatscheck, O., and Shkapenyuk, V. (2003b). Gigascope: A stream database for internet databases. In *SIGMOD*, pages 647–651.

[Crypto++ Library, 2005] Crypto++ Library (2005). http://www. eskimo. com/ ∼weidai/ cryptlib.html.

[Datar et al., 2002] Datar, M., Gionis, A., Indyk, P., and Motwani, R. (2002). Maintaining stream statistics over sliding windows. In *SODA*, pages 635–644.

[de Berg et al., 2000] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. (2000). *Computational Geometry: Algorithms and Applications*. Springer.

[Devanbu et al., 2003] Devanbu, P., Gertz, M., Martel, C., and Stubblebine, S. (2003). Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291–314.

[Devanbu et al., 2000] Devanbu, P., Gertz, M., Martel, C., and Stubblebine, S. G. (2000). Authentic third-party data publication. In *IFIP Workshop on Database Security (DBSec)*, pages 101–112.

[Evfimievski et al., 2003] Evfimievski, A., Gehrke, J., and Srikant, R. (2003). Limiting privacy breaches in privacy preserving data mining. In *PODS*, pages 211–222.

[Flajolet and Martin, 1985] Flajolet, P. and Martin, G. N. (1985). Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209.

[Freivalds, 1979] Freivalds, R. (1979). Fast probabilistic algorithms. In *Proc. of International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 57–69.

[Fu et al., 2002] Fu, K., Kaashoek, M. F., and Mazieres, D. (2002). Fast and secure distributed read-only file system. *ACM Trans. Comput. Syst.*, 20(1):1–24.

[Ganguly et al., 2004] Ganguly, S., Garofalakis, M., and Rastogi, R. (2004). Tracking set-expression cardinalities over continuous update streams. *The VLDB Journal*, 13(4):354–369.

[Garofalakis et al., 2007] Garofalakis, M., Hellerstein, J. M., and Maniatis, P. (2007). Proof sketches: Verifiable in-network aggregation. In *ICDE*.

[Ge and Zdonik, 2007] Ge, T. and Zdonik, S. (2007). Fast, secure encryption for indexing in a column-oriented dbms. In *ICDE*.

[Gilbert et al., 2002] Gilbert, A. C., Kotidis, Y., Muthukrishnan, S., and Strauss, M. (2002). How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB*, pages 454–465.

[Golab, 2006] Golab, L. (2006). *Sliding Window Query Processing over Data Streams*. PhD thesis, University of Waterloo, Canada.

[Goldwasser et al., 1988] Goldwasser, S., Micali, S., and Rivest, R. L. (1988). A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):96–99.

[Goodrich et al., 2003] Goodrich, M. T., Tamassia, R., Triandopoulos, N., and Cohen, R. (2003). Authenticated data structures for graph and geometric searching. In *CT-RSA*, pages 295–313.

[Guttman, 1984] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57.

[Hacigümüs et al., 2002a] Hacigümüs, H., Iyer, B. R., Li, C., and Mehrotra, S. (2002a). Executing SQL over encrypted data in the database service provider model. In *SIGMOD*, pages 216–227.

[Hacigümüs et al., 2002b] Hacigümüs, H., Iyer, B. R., and Mehrotra, S. (2002b). Providing database as a service. In *ICDE*, pages 29–40.

[Hammad and et al., 2004] Hammad, M. and et al. (2004). Nile: A query processing engine for data streams. In *ICDE*.

[Hammad et al., 2004] Hammad, M. A., Mokbel, M. F., Ali, M. H., Aref, W. G., Catlin, A. C., Elmagarmid, A. K., Eltabakh, M., Elfeky, M. G., Ghanem, T. M., Gwadera, R., Ilyas, I. F., Marzouk, M., and Xiong, X. (2004). Nile: A query processing engine for data streams. In *ICDE*, page 851.

[Ho et al., 1997] Ho, C.-T., Agrawal, R., Megiddo, N., and Srikant, R. (1997). Range queries in OLAP data cubes. In *SIGMOD*, pages 73–88.

[Hore et al., 2004] Hore, B., Mehrotra, S., and Tsudik, G. (2004). A privacy-preserving index for range queries. In *VLDB*, pages 720–731.

[Huebsch et al., 2003] Huebsch, R., Hellerstein, J. M., Lanham, N., Loo, B. T., Shenker, S., and Stoica, I. (2003). Querying the internet with pier. In *VLDB*.

[Jürgens and Lenz, 1999] Jürgens, M. and Lenz, H. (1999). Pisa: Performance models for index structures with and without aggregated data. In *SSDBM*, pages 78–87.

[Kamel and Faloutsos, 1993] Kamel, I. and Faloutsos, C. (1993). On packing R-Trees. In *CIKM*, pages 490–499.

[Knuth, 1997] Knuth, D. E. (1997). *The Art of Computer Programming*. Addison-Wesley.

[Lazaridis and Mehrotra, 2001] Lazaridis, I. and Mehrotra, S. (2001). Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412.

[Li et al., 2006a] Li, F., Hadjieleftheriou, M., Kollios, G., and Reyzin, L. (2006a). Authenticated index sturctures for aggregation queries in outsourced databases. Technical report, CS Dept., Boston University.

[Li et al., 2006b] Li, F., Hadjieleftheriou, M., Kollios, G., and Reyzin, L. (2006b). Dynamic authenticated index structures for outsourced databases. In *SIGMOD*.

[Li et al., 2007] Li, F., Yi, K., Hadjieleftheriou, M., and Kollios, G. (2007). Proof-infused streams: Enabling authentication of sliding window queries on streams. In *VLDB*.

[Manku and Motwani, 2002] Manku, G. S. and Motwani, R. (2002). Approximate Frequency Counts over Data Streams. In *VLDB*, pages 346–357.

[Martel et al., 2004] Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., and Stubblebine, S. (2004). A general model for authenticated data structures. *Algorithmica*, 39(1):21–41.

[McCurley, 1990] McCurley, K. (1990). The discrete logarithm problem. In *Proc. of the Symposium in Applied Mathematics*, pages 49–74. American Mathematical Society.

[Merkle, 1978] Merkle, R. C. (1978). Secure communications over insecure channels. *CACM*, 21(4):294–299.

[Merkle, 1989] Merkle, R. C. (1989). A certified digital signature. In *CRYPTO*, pages 218–238.

[Micali, 1996] Micali, S. (1996). Efficient certificate revocation. Technical Report MIT/LCS/TM-542b, Massachusetts Institute of Technology, Cambridge, MA.

[Miklau, 2005] Miklau, G. (2005). *Confidentiality and Integrity in Distributed Data Exchange*. PhD thesis, University of Washington.

[Miklau and Suciu, 2003] Miklau, G. and Suciu, D. (2003). Controlling access to published data using cryptography. In *VLDB*, pages 898–909.

[Miklau and Suciu, 2005] Miklau, G. and Suciu, D. (2005). Implementing a tamper-evident database system. In *ASIAN*.

[Mitra et al., 2006] Mitra, S., Hsu, W., and Winslett, M. (2006). Trustworthy keyword search for regulatory-compliant record retention. In *VLDB*, pages 1001–1012.

[Motwani and Raghavan, 1995] Motwani, R. and Raghavan, P. (1995). *Randomized Algorithms*. Cambridge University Press.

[Muthukrishnan, 2003] Muthukrishnan, S. (2003). Data streams: algorithms and applications.

[Mykletun et al., 2004a] Mykletun, E., Narasimha, M., and Tsudik, G. (2004a). Authentication and integrity in outsourced databases. In *NDSS*.

[Mykletun et al., 2004b] Mykletun, E., Narasimha, M., and Tsudik, G. (2004b). Signature bouquets: Immutability for aggregated/condensed signatures. In *ESORICS*, pages 160–176.

[Mykletun and Tsudik, 2006] Mykletun, E. and Tsudik, G. (2006). Aggregation queries in the database-as-a-service model. In *DBSec*.

[Nagell, 1981] Nagell, T. (1981). *Introduction to Number Theory*. Chelsea Publishing Company, second edition.

[Naor and Nissim, 1998] Naor, M. and Nissim, K. (1998). Certificate revocation and certificate update. In *Proceedings of the USENIX Security Symposium*.

[Narasimha and Tsudik, 2005] Narasimha, M. and Tsudik, G. (2005). Dsac: Integrity of outsourced databases with signature aggregation and chaining. In *CIKM*, pages 235–236.

[National Institute of Standards and Technology, 1995] National Institute of Standards and Technology (1995). *FIPS PUB 180-1: Secure Hash Standard*. National Institute of Standards and Technology.

[OpenSSL, 2005] OpenSSL (2005). http://www.openssl.org.

[Pang et al., 2005] Pang, H., Jain, A., Ramamritham, K., and Tan, K.-L. (2005). Verifying completeness of relational query results in data publishing. In *SIGMOD*, pages 407–418.

[Pang and Tan, 2004] Pang, H. and Tan, K.-L. (2004). Authenticating query results in edge computing. In *ICDE*, pages 560–571.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *CACM*, 21(2):120–126.

[Rizvi et al., 2004] Rizvi, S., Mendelzon, A., Sudarshan, S., and Roy, P. (2004). Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, pages 551–562.

[Rusu and Dobra, 2007] Rusu, F. and Dobra, A. (2007). Pseudo-random number generation for sketch-based estimations. *TODS*, 32(2).

[Saroiu et al., 2002] Saroiu, S., Gummadi, P. K., Dunn, R., Gribble, S., and Levy, H. (2002). An analysis of internet content delivery systems. In *OSDI*.

[Schwartz, 1980] Schwartz, J. T. (1980). Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717.

[Siegel, 1989] Siegel, A. (1989). On universal classes of fast high performance hash functions, their time-space tradeoff, and their applications. In *FOCS*, pages 20–25.

[Sion, 2004] Sion, R. (2004). Proving ownership over categorical data. In *ICDE*.

[Sion, 2005] Sion, R. (2005). Query execution assurance for outsourced databases. In *VLDB*, pages 601–612.

[Sion et al., 2003] Sion, R., Atallah, M., and Prabhakar, S. (2003). Rights protection for relational data. In *SIGMOD*.

[Sion et al., 2004] Sion, R., Atallah, M., and Prabhakar, S. (2004). Protecting rights over relational data using watermarking. *IEEE TKDE*, 16(12):1509–1525.

[Sion et al., 2006] Sion, R., Atallah, M., and Prabhakar, S. (2006). Rights protection for discrete numeric streams. *TKDE*, 18(5):699–714.

[Tamassia and Triandopoulos, 2005a] Tamassia, R. and Triandopoulos, N. (2005a). Computational bounds on hierarchical data processing with applications to information security. In *Proc. of the International Colloquim on Automata, Languages and Programming (ICALP)*, pages 153–165.

[Tamassia and Triandopoulos, 2005b] Tamassia, R. and Triandopoulos, N. (2005b). Efficient content authentication over distributed hash tables. Technical report, Brown University.

[Tao and Papadias, 2004a] Tao, Y. and Papadias, D. (2004a). Performance analysis of R*-Trees with arbitrary node extents. *TKDE*, 16(6):653–668.

[Tao and Papadias, 2004b] Tao, Y. and Papadias, D. (2004b). Range aggregate processing in spatial databases. *TKDE*, 16(12):1555–1570.

[Tatbul et al., 2003] Tatbul, N., Cetintemel, U., Zdonik, S., Cherniack, M., and Stonebraker, M. (2003). Load shedding in a data stream manager. In *VLDB*, pages 309–320.

[Tatbul and Zdonik, 2006] Tatbul, N. and Zdonik, S. (2006). Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810.

[Theodoridis and Sellis, 1996] Theodoridis, Y. and Sellis, T. K. (1996). A model for the prediction of r-tree performance. In *PODS*, pages 161–171.

[Wasserman and Blum, 1997] Wasserman, H. and Blum, M. (1997). Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849.

[Wegman and Carter, 1981] Wegman, M. N. and Carter, J. L. (1981). New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3).

[Yang et al., 2005] Yang, S., Butt, A. R., Hu, Y. C., and Midkiff, S. P. (2005). Trust but verify: monitoring remotely executing programs for progress and correctness. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 196–205.

[Yi et al., 2007] Yi, K., Li, F., Hadjieleftheriou, M., Srivastava, D., and Kollios, G. (2007). Randomized synopses for query assurance on data streams. Technical report, AT&T Labs Inc.

[Zhang et al., 2005] Zhang, R., Koudas, N., Ooi, B. C., and Srivastava, D. (2005). Multiple aggregations over data streams. In *SIGMOD*, pages 299–310.

[Zhu and Hsu, 2005] Zhu, Q. and Hsu, W. (2005). Fossilized index: The linchpin of trustworthy non-alterable electronic records. In *SIGMOD*, pages 395–406.

CURRICULUM VITAE

Feifei Li

RM138, 111 Cummington St.
Department of Computer Science
Boston University
Boston, MA 02215 USA

Voice: (617) 353-5227
Fax: (617) 353-6457
E-mail: lifeifei@cs.bu.edu
Webpage: http://cs-people.bu.edu/lifeifei/

Research Interests

Security and privacy in both streaming and relational database systems, management and indexing of large databases, spatio-temporal database applications, stream and sensor databases.

Education

September 2002 - July 2007 (Expected) Ph.D. in Computer Science, Computer Science Department, Boston University. Advisor: Prof. George Kollios.

September 1998 - Jan 2002 B.S. in Computer Engineering, School of Computer Engineering, Nanyang Technological University, Singapore.

September 1997 - December 1997 Study in Department of Electrical Engineering, Tsinhua University, Beijing, China.

September 1994 - June 1997 Study in Tsinghua High School attached to Tsinhua University, Beijing, China.

Awards

- Best Presenter Award, in research summer interns Seminar@Luch Series, IBM T.J. Watson Research Center, August, 2006.
- Best Paper Award, in 20th International Conference on Data Engineering(ICDE), March, 2004.

Publications

Refereed Conferences

1. Proof-Infused Streams: Enabling Authentication of Sliding Window Queries On Streams, by **F. Li**, K. Yi, M. Hadjieleftheriou, G. Kollios, (To Appear) In Proceedings of the 33rd International Conference on Very Large Databases(**VLDB 2007**), Vienna, Austria, September 2007.

2. Time Series Compressibility and Privacy, by S. Papadimitriou, **F. Li**, G. Kollios, P. S. Yu, (To Appear) In Proceedings of the 33rd International Conference on Very Large Databases (**VLDB 2007**), Vienna, Austria, September 2007.

3. Hiding in the Crowd: Privacy Preservation on Evolving Streams through Correlation Tracking, by **F. Li**, J. Sun, S. Papadimitriou, G. Mihaila, and I. Stanoi. In Proceedings of the 23nd IEEE International Conference on Data Engineering (**ICDE 2007**), Istanbul, Turkey, April 2007.

4. Dynamic Authenticated Index Structures for Outsourced Databases, by **F. Li**, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. In Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (**SIGMOD 2006**), Chicago, Illinois, USA, June 2006.

5. Characterizing and Exploiting Reference Locality in Data Stream Applications, by **F. Li**, C. Chang, G. Kollios, and A. Bestavros. In Proceedings of the 22nd IEEE International Conference on Data Engineering (**ICDE 2006**), Atlanta, Georgia, USA, April 2006.

6. On Trip Planning Queries in Spatial Databases, by **F. Li**, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. In Proceedings of the 9th International Symposium on Spatial and Temporal Databases (**SSTD 2005**), Angra dos Reis, Brazil, August, 2005.

7. Approximate Aggregation Techniques for Sensor Databases, by J. Considine, **F. Li**, G. Kollios, and J. Byers. In Proceedings of the 20th IEEE International Conference on Data Engineering (**ICDE 2004**), Boston, MA, March 30 - April 2, 2004. **Best Paper Award**

8. Spatio-Temporal Aggregation Using Sketches, by Y. Tao, G. Kollios, J. Considine, **F. Li**, and D. Papadias. In Proceedings of the 20th IEEE International Conference on Data Engineering (**ICDE 2004**), Boston, MA, March 30 - April 2, 2004.

9. WICCAP Data Model: Mapping Physical Websites to Logical Views, by Z. Liu, **F. Li**, and W. Ng. In Proceedings of the 21st International Conference on Conceptual Modeling (**ER 2002**), Tampere, Finland, October 8-10, 2002. Springer.

10. Web Information Collection, Collaging and Programming, by **F. Li**, Z. Liu, Y. Huang, and W. Ng. In Proceedings of the 3rd IEEE International Conference on Information, Communications & Signal Processing (**ICICS 2001**), Singapore, October, 2001.

Refereed Journals

1. Robust Aggregation in Sensor Networks, by G. Kollios, J. Byers, J. Considine, M. Hadjieleftheriou, and **F. Li**. IEEE Data Engineering Bulletin, Vol. 28, No. 1, March 2005.

2. Towards Building Logical Views of Websites, by Z. Liu, W. Ng, E. Lim, **F. Li**. Data & Knowledge Engineering, Vol. 49, No 2, May 2004, Elsevier. (Invited by WIDM 2002).

Refereed Workshops

1. An Information Concierge for the Web, by **F. Li**, Z. Liu, Y. Huang, and W. Ng. In Proceedings of the First International Workshop on Internet Bots: Systems and Applications (**INBOSA 2001**), in conjunction with the 12th International Conference on Database and Expert System Applications (**DEXA 2001**), Munich, Germany, September 3-8, 2001.

Technical Reports

1. Efficient Processing of Top-k Queries on Uncertain Databases, by K. Yi, **F. Li**, D. Srivastava and G. Kollios, AT&T Technical Report, June, 2007.

2. Randomized Synopsis for Query Assurance on Data Streams, by K. Yi, **F. Li**, M. Hadjieleftheriou, D. Srivastava and G. Kollios, AT&T Technical Report, June, 2007.

3. Authenticated Index Structures for Aggregation Queries in Outsourced Databases, by **F. Li**, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, BUCS-TR-2006-011, May, 2006.

4. GreedyDual-Join: Locality-Aware Buffer Management for Approximate Join Processing Over Data Streams, by **F. Li**, C. Cheng, A. Bestavros , and G. Kollios, BUCS-TR-2004-028, April, 2004.

Professional Experience

- **Database Research Group, Microsoft Research**, Redmond, WA, USA

  Research Summer Intern **May, 2007 - August, 2007**

- **Database Management Research Department, AT&T Shannon Research Lab**, Florham Park, NJ, USA

  Research Consultant on Data, Network Security and Privacy **Since Sep, 2006**

- **Database Research Group, IBM T.J.Watson Research Center**, Hawthorne, NY, USA

  Research Summer Intern **June, 2006 - September, 2006**

- **SQL Server Group, Microsoft**, Redmond, WA, USA

  Summer Intern **June, 2005 - September, 2005**

Professional Service

- External Reviewer for the following journals: TKDE, VLDBJ.
- External Reviewer for the following conferences: VLDB 2007, ICDE 2007, VLDB 2006, ICDE 2006, VLDB 2005.
- Conference Volunteer for the following conferences: ICDE 2004, SSDBM 2003.

Teaching Experience

- Teaching Assistant, Computer Science Department, Boston University, Fall, 2002. Course: Introduction to Computer Science.
- Teaching Assistant, Computer Science Department, Boston University, Spring, 2003. Course: Introduction to C++.
- Teaching Assistant, Computer Science Department, Boston University, Fall, 2003. Course: Advanced Database Systems.
- Teaching Assistant, Computer Science Department, Boston University, Spring, 2004. Course: Introduction to Database Systems.

Patents

- Co-Inventor, Method of Privacy-Preserving Data Stream Publishing using Dynamic Correlations, US patent YOR9-2006-0786
- Co-Inventor, Method of Privacy-Preserving Data Stream Publishing using Dynamic Autocorrelation, US patent YOR9-2006-0787