TOWARDS SCALABLE, CLOUD-BASED, CONFIDENTIAL DATA STREAM PROCESSING

by

Cory Thoma

B.A. in Mathematics and Information Technology Leadership,

Washington & Jefferson College, 2012

Submitted to the Graduate Faculty of

the School of Computing and Information in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2019

UNIVERSITY OF PITTSBURGH

SCHOOL OF COMPUTING AND INFORMATION

This dissertation was presented

by

Cory Thoma

It was defended on

August 12, 2019

and approved by

Adam J. Lee, Associate Professor, University of Pittsburgh

Alexandros Labrinidis, Professor, University of Pittsburgh

Panos K. Chrysanthis, Professor, University of Pittsburgh

Florian Kerschbaum, Associate Professor, University of Waterloo

Dissertation Advisors: Adam J. Lee, Associate Professor, University of Pittsburgh,

Alexandros Labrinidis, Professor, University of Pittsburgh

TOWARDS SCALABLE, CLOUD-BASED, CONFIDENTIAL DATA STREAM PROCESSING

Cory Thoma, PhD

University of Pittsburgh, 2019

Increasing data availability, velocity, variability, and size have lead to the development of new data processing paradigms that offer users different ways to process and manage data specific to their needs. One such paradigm is data stream processing, as managed by Data Stream Processing Systems (DSPS). In contrast to traditional database management systems wherein data is stationary and queries are transient, in stream processing systems, data is transient and queries are stationary (that is, continuous and long running). In such systems, users are expecting to process temporal data, where data is only considered for some period of time, and discarded after. Often, as with many other software applications, those who employ such systems will outsource computation to third party computation platforms such as Amazon, IBM, or Google. The use of third parties not only outsources computation, but it outsources hardware and software maintenance costs as well, relieving the user from having to incur these costs themselves. Moreover, when a user outsources their DSPS, they often have some service level agreement that places guarantees on service availability and uptime.

Given the above benefits to outsourcing computation, it is clearly desirable for a user to outsource their DSPS computation. Such outsourcing, however, may violate the privacy constraints of the those who provide the data stream. Specifically, they may not wish to share their plaintext data with a third-party that they may not trust. This leads to an interesting dichotomy between the desire of the user to outsource as much of their computation as possible and the desire of the data stream providers to keep their data private and avoid

leaking data to a third-party system. Current work that explores linking the two poles of this dichotomy either limits the expressiveness of supported queries, requires the data provider to trust the third-party systems, or incurs computational or monetary overheads prohibitive for the querier.

In this dissertation, we explore the methods for shrinking the gap between the poles of this dichotomy and overcome the limitation of the state-of-the art systems by providing data providers and queriers with efficient access control enforcement on untrusted third party systems over encrypted data. Specifically, we introduce our system PolyStream for executing queries on encrypted data using computation-enabling encryption, with an online key management system. We further introduce Sanctuary to provide computation on any data on third-party systems using trusted hardware. Finally we introduce Shoal, our query optimizer that considers the heterogeneous nature of streaming systems at optimization time to improve query performance when access controls are enforced on the streaming data. Through the union of the contributions of this dissertation, we show that considering access controls at optimization time can lead to better utilization, performance, and protection for streaming data.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

1.1 MOTIVATION

Increasingly, consumers are employing third-party computation platform service providers to reliably, efficiently, and cost-effectively process and store their data for them. Such outsourcing provides consumers with the benefits of popular and widely used software (e.g., databases, message hubs, data processing engines, and other data transformation systems) without the overheads associated with having to maintain their own hardware and software infrastructure. Moreover, popular cloud applications often come with high availability and high uptime guarantees, allowing users to safely rely on the services that are being provided.

This increase in the use of third-party computational platforms not only stems from the desire to reduce or eliminate costs in maintaining and operating expensive servers, but also from the increase in the amount of data available. Cloud platforms can often offer scaled or tiered pricing based on use, allowing consumers to dynamically allocate more resources with the push of a button. Cloud platforms also offer a variety of techniques and systems for processing large quantities of data. Specifically, in a system where data is transient or where the aggregate is more important than individual data points, consumers may choose to process data using a Data Stream Processing System (DSPS). In a DSPS, *data is transient* (i.e., short-lived and not intended to be stored in persistent data storage, e.g., sensor or driving data) and *queries are persistent* and process data in batches. As such, data consumers can employ DSPSs to filter, aggregate, or map data from multiple sources to reduce the amount of data that needs to be processed, allowing the user to store or further process only the data that they care about, rather than *all* of the data available to them. Data consumers would choose a DSPS over a traditional database management system in

scenarios where live data points are the most important and queries on past data are less relevant (e.g., monitoring applications or trend analysis). Stream processing is a paradigm that can be leveraged for any time-sensitive queries (i.e., where the timeliness of results is the most important) or queries that need to process large amounts of potentially needless data to help pinpoint data of significance for further processing.

Outsourcing DSPS computation comes with many direct benefits for the data consumer. When an individual contracts a third-party service to process data streams on their behalf, they do not directly incur the costs associated with owning and maintaining the hardware necessary for computing. They further relieve themselves of the concerns associated with maintaining the infrastructure (e.g., updates, upgrades, security patches, etc.) as they are all handled by the third-party vendor. Moreover, if there is a spike in data availability or volume, a third-party cloud provider can scale on demand. Finally, cloud computing platforms often provide rolling upgrades to software, which allows for high uptime guarantees, and can replicate data across data centers, allowing for high availability guarantees.

A data consumer has much to gain when outsourcing their DSPS processing to a third-party cloud service provider. However, the data providers that stream data to the data consumers may experience some unintended violations of their privacy or access controls when their data streams are processed by someone other than the intended data consumer(s). When a data provider emits a transient data point (i.e., a tuple), they lose the ability to dictate or control who can access that tuple. Depending upon the perceived sensitivity of the data, the provider may be less willing to share if there is a possibility that their data may be shared with unknown or unintended entities (e.g., a third-party cloud computing platform). With this potential uncertainty over who has access to their data, a provider may alter their access controls to disallow third-party access, or to remove data consumers altogether. Alternatively, the data provider may choose to protect their data via encryption, thus removing a data consumers ability to outsource computation on data streams. This, of course, would require a key sharing framework that has the capability of revocation when the data provider decides to alter their access controls.

Given the marked benefits of outsourcing DSPS computation to a third-party system, a data consumer is likely to favor the use of such a system. However, given the potential

violation of a data provider's privacy and access controls, a data consumer is likely to favor a more limited use of third-party computation platforms. Given this dichotomy of interests in third-party computation, there exists an interesting and novel research space in the tradeoff of balancing the data consumer's interests with the data provider's interests.

1.2 EXAMPLE SCENARIO

To help illustrate the issues that arise when exploring the tradeoff between a data consumer's desire to outsource their data processing to third-party systems and a data provider's desire to maintain or ensure their data privacy and access controls, we consider the following scenario.

Mark is diabetic who is interested in sharing his fitness and health data with his doctor, to better manage his condition,. He uses connected devices that can automatically share data with his doctor such as a smart watch, a smart insulin pump, and a smart glucose meter. Mark's doctor, Robert, plans to use this data to help Mark control his diabetes and live an uninterrupted life. Dr. Robert, however, does not maintain the systems used to collect and process Mark's health data. Instead, Dr. Robert relies on a service offered by Mark's insurance provider, InsuranceCo, which provides an interface for Dr. Robert to access Marks data. Dr. Robert simply enters queries or custom scripts into the system, and the system does the rest (i.e., subscribes to the relevant streams, executes the queries or scripts on the streaming data, and returns the results to Dr. Robert).

Mark is uncomfortable with InsuranceCo having such granular access to his private health data, which he originally intended for only the eyes of Dr. Robert. Given his hesitation with sharing his private data with InsuranceCo, Mark has the choice of two extremes with respect to protecting his data: namely 1.) provide no protection and allow InsuranceCo to process and view his private data, or 2.) do not share his data at all with Dr. Robert. Both scenarios are undesireable for Mark, as he either must share his private data with a party he did not intend to, or he does not get to benefit from sharing his health data with his doctor. This implies that Dr. Robert would have to maintain some infrastructure himself in order to

process data from patients like Mark, which he may not elect to do.

The middle ground between Mark sharing everything with InsuranceCo and Mark sharing nothing with his doctor offers an interesting space to explore with respect to streaming data management systems. Specifically, research into schemes that allow Mark to share information subject to his access controls, while also allowing Dr. Robert to utilize third-party systems are of particular interest as they balance the concerns of the data provider and the data consumer in some novel way.

There are several approaches a data provider or data consumer could take to help protect sensitive information from unwanted eyes while allowing third-party processing by the data consumer. We can classify these paradigms by their enforcement location (in-network or off-network) and the data visibility (network vs. consumer). Specifically, these build up into the following combinations of *enforcement location* and *data visibility*:

- *In-network, Visible*: Access controls are enforced by third-party systems on plaintext data. Systems like FENCE [1] and Borealis [2] operate in this realm.
- *In-network, Not Visible*: Again, access controls are enforced by third-party systems, but data is not transfered in plaintext, and therefore not visible to the third-party system. The work presented in this dissertation examines this realm, where access controls are enforced in network and data is not visible to any party within the network short of the data consumers. Systems like PolyStream [3], Streamforce [4] and CryptDB [5] (for traditional relational databases) work in this space now.
- *Off-network, Visible*: Access controls are enforced by the data consumer over plaintext data. This case is undesirable for the data provider as their data is transmitted in plaintext. Worse yet, this case requires the querier to do all of the computation locally, eliminating any outsourced computation.
- *Off-network, Not Visible*: Data producers send encrypted data *directly* to data consumers, bypassing third-party systems. This is the degenerate case, and undesirable for the data consumer as they have to process all data themselves on their own hardware.

To help understand these different location and data-visibility paradigms, consider a simple query like the following:

```
SELECT a.location, b.steps
FROM phone AS a, health AS b
WHERE a.ID = b.ID;
```

This query simply filters and joins data from two streams so that people who are in a certain region can be marked as active. A simple app that pairs active runners in each region could outsource such queries to a third-party data stream processing system. Each data provider may be uncomfortable with their location and health data being made visible. In the *In-network, Visible* paradigm, their location and health data is visible to a third-party system, which itself may be undesirable. Further, if a data provider did want to enforce access controls, they would have to trust a third-party system to do so, which again may be undesirable. This scenario, however, is desirable for the data consumer as they can fully utilize third-party systems.

The *Off-network, Visible* option is similar in that their location and health data are transmitted in plaintext. Further, access control enforcement resides with the data provider, which implies that if the data provider doesn't want plaintext location or health data flowing through a third-party system, they will have to stream their data directly to the data consumer. This is undesirable for the data consumer as well as they will need to process data themselves, without the aid of a third-party system.

The *In-network, Not Visible* paradigm provides the most interesting case as it allows a data provider to keep their data secure (not transmitted in plaintext) and further allows them to enforce access controls in the network. This in-network access-control enforcement means that a data provider can author access controls and some component in the network will enforce them. For our query above, a data provider can author an access control (e.g., an Access Control List, or any sort of Role or Attribute Based Access Control) policy that will ensure only the intended data consumers will have access. Further, their location and health data can be kept secure with a set of encryption techniques (e.g., Randomized AES encryption, some computation enabling encryption, etc.). This approach, however, may be undesirable for a data consumer since secure data would imply that they lose their ability to

process on third-party systems. The focus of this dissertation, however, is the exploration of this paradigm wherein data providers can keep data secure, while data consumers can utilize the third-party system.

1.3 PROBLEM STATEMENT

In certain scenarios, a data provider can enforce access to their data by opting into third-party protection services. For instance, simple access control lists on a third-party cloud provider may be sufficient for a data provider (such as adding or removing users who had been previously given access on AWS, or creating groups of users for Google offerings)([6, 7, 8]). Such systems require that the data provider implicitly trusts the third-party system. However, when the data consumer is the entity contracting the third-party service, as is often the case (i.e., the data consumer purchases the resources for computing, not the data provider), the data provider has little input into who or what third-party has access to their data. Given this lack of input from the data provider as to who can access their data (without fully trusting the third-party system), we get the main limitation of these types of systems; *data providers may not trust third-party service providers to enforce access controls on their behalf.*

To help illustrate this, consider the following simple continuous query:

```
SELECT name, age, salary, average(travel)
FROM stream
GROUP BY name EVERY 10 minutes UPDATE 5 minutes;
```

This simple query gets a person's average travel distance over the last 10 minutes, and reports it to the user every 5 minutes, along with their name, age, and salary. A person who is providing this data is divulging what could very well be considered sensitive and private data: their salary and age. As such, they may not be comfortable sharing this data with anyone whom they have not explicitly granted access. If this data provider was presented with the option described above (i.e., trust the third-party with enforcing access to your

plaintext age and salary), they may choose to not provide data.

When a data provider wishes to confidently and unconditionally control access to their data, they can choose some type of encryption scheme. In such schemes, the data provider will share encryption keys with designated, pre-determined data consumers. If, however, the data provider elects to use a fully random encryption scheme (e.g., AES in CBC mode) the data consumer cannot employ a third-party computation system without releasing the key to the system (which would likely constitute a violation of the trust between the data provider and the data consumer), decrypting and forwarding data (which would likely also be a violation of the trust between the data provider and the data consumer), or to do the computation on their own trusted infrastructure (an undesirable option for the data consumer, who wants to push computation to a third-party service).

As a compromise, the data consumer and data provider can agree to use computation-enabling encryption [4, 3, 5] as a middle ground between enforcing access control and allowing for third-party computation. Computation-enabling encryption techniques are those that allow for some mathematical operation to be executed over the *ciphertext* of a given plaintext. Such techniques preserve some property of the plaintext within the ciphertext (i.e., a deterministic scheme would preserve the property that that if $x = y$ then $e(x) = e(y)$ for some encryption function e). Given the nature of the current state of computation-enabling encryption, however, data providers would have to accept some level of information leakage, and data consumers would have to accept some level of degradation in the expressiveness or coverage of their query language (i.e., some operations are not possible). This brings us to the major drawbacks of using encryption to enforce access; *either data consumers must execute all queries on their own hardware, or data providers must be willing to accept some level of information leakage.*

Consider again our example query above. Put simply, if the data provider elected for some fully random encryption scheme, the entire query would have to execute on the data provider's trusted hardware. This would, of course, be undesirable as the data consumer wants to utilize third-party systems as often as possible. If the data provider is willing to use some sort of computation-enabling encryption scheme, the data consumer (i.e., the query author) may utilize some third-party services, but the data provider may suffer some

data loss. For instance, if the data provider decides to deterministically encrypt their age, a curious adversary could gather all of the ages produced using the same key and develop a distribution of all of the ages in this stream. They can further compare this distribution to common age distributions to potentially reveal the actual age. Both scenarios (data consumer processing all data vs. some information being leaked via computation enabling encryption schemes) have some level of undesirability for the data consumer and data provider, an area that is further studied in this dissertation.

A final approach that a data consumer can take to enforce access control is via query rewriting [2, 9](moving physical operators, switching the order of operations, providing views or aggregate data, etc.), or via special operations that get inserted and direct data to different locations (which may require trusting the third-party) [1, 6]. These approaches require that the data consumer keeps some information about the privacy or access control preferences for each of their data providers (so that they can re-write or place operators appropriately) or it requires them to limit the types of operations that can be performed. This implies that *a data consumer may be limited in their query expressiveness or free use of the data for which they have been granted access, as dictated by the enforcement mechanism put in place.*

Consider again our example query above. We will add an assumption that, based on the data provider not wanting to reveal plaintext to the third-party system, the querier is only given aggregate level access to the stream via a pre-defined view such that they can only see an age range and a salary range for each data point. Given this view, at best the querier can change this query to just select the average age and salary combination in terms of distance traveled, but it will be coarse-grained data and may not represent the intention of the original query. This limiting factor has reduced the expressiveness of their query from a fine-grained, per age average, to a more coarse-grained, less precise result, which leads to a less satisfied querier.

This current state of distributed data stream processing leads to users having to make the choice between privacy (keeping all of their data secure and accessible by only permitted data consumers) and functionality (having the ability to compute any streaming operation in a cloud environment) rather than finding an acceptable middle ground. This gap naturally leads to the main problem space explored in this dissertation:

When providing data in a streaming fashion, a data provider loses their ability to enforce access controls to their data. Moreover, enforcing access controls in a cloud-based, distributed, data stream processing system without trusting a third-party to do so can result in large computational overheads (due to encryption or the limitations of potential query plans) and network overheads (due to larger data tuples and additional control tuples). Additionally, heterogeneity inherent to data and physical systems can affect and are affected by access control enforcement decisions.

In this dissertation, we explore the combination of computation-enabling encryption, specialized hardware, and user-specified preferences to help bridge the gap between privacy and functionality in data stream processing. Specifically, we introduce PolyStream, which allows data providers to specify differing levels of access (e.g., fully encrypted vs. some computation enabling encryption technique). These levels are created from different computation-enabling encryption schemes, where each different level allows *some* level of computation to be executed directly on the encrypted data. Such an approach, however, limits query expressiveness and has potential data leakage (where an adversary can learn something from the ciphertext depending on the chosen encryption technique). Moreover, we introduce Sanctuary as an augmentation or extension of PolyStream that allows for specialized hardware to fully outsource a data consumer's query on an untrusted cloud system *regardless* of the encryption used. We further augment PolyStream and Sanctuary by adding heterogeneity-aware query optimization with Shoal to help reduce the cost incurred by using computation-enabling-encryption and specialize hardware.

1.4 IMPROVING PROTECTION, UTILIZATION, AND PERFORMANCE TO BRIDGE THE GAP BETWEEN CONFIDENTIALITY AND FUNCTIONALITY

It is the hypothesis of this dissertation that designing new access control enforcement techniques for cloud-based distributed data stream processing systems and considering them at optimization time can lead to better protection, utilization, and performance.

In this dissertation, we present an approach that empowers data providers with the ability to protect their data via cryptography and access control frameworks while providing data consumers with the ability to utilize third-party computation platforms, all while minimizing the cost to both parties. Our approach helps bridge the gap between confidentiality and functionality by allowing both the data provider and data consumer to have a choice in how to best protect and process their data. We offer better protection via encryption and trusted hardware, utilization via computation-enabling encryption and specialized hardware, and performance via query optimization and different levels of computation-enabling encryption.

1.5 CONTRIBUTIONS

To support the hypothesis above, we specifically make the following contributions:

- In Chapter 3 we present our PolyStream system, which allows a data provider to use Attribute Based Access Controls (ABAC) (Section 3.3) to specify controls on their encrypted streaming data via Security Punctuations (Section 3.3). We then define a set of computation-enabling encryption techniques (Section 3.3) that a user can choose from to enable third-party computation on encrypted data. We finally evaluate PolyStream versus other access control systems (Section 3.4) and find that it outperforms the stat-of-the-art system up to 14x.
- In Chapter 4 we present our Sanctuary system that employs Intel's Software Guard Extensions (SGX) to allow for arbitrary computation on an untrusted third-party system (not possible with comparable state-of-the-art systems). We present memory-limited data stream processing operator algorithms that work within a memory-limited SGX environment (Section 4.4) that allow for arbitrary computation and provide a security analysis (Section 4.5) of those algorithms. We further evaluate Sanctuary for microbenchmarks and system level benchmarks (Section 4.6) and find that it provides greater privacy guarantees while having comparable performance.
- In Chapter 5 we present our Shoal optimizer for producing efficient query plans for multiple streaming queries. Shoal considers different implementations of the same streaming

operation when placing them in a distributed stream processing system. Specifically, Shoal uses a dynamic programming algorithm (Section 5.3) that optimally places each operation with respect to a distributed cost function (Section 5.2). We show that Shoal can produce better plans that the state-of-the-art (Section 5.4) with reasonable execution time overheads, all while considering specialized operations or specialized hardware to enhance performance in an access-controlled environment.

Polystream allows a data provider to enforce their access control policies while still allowing a data consumer to better utilize the third-party enforcement techniques, and to gain performance over the state-of-the-art system. To build further on top of Polystream, Sanctuary utilizes specialized hardware to, again, better utilize third-party systems while enhancing protection. Shoal empowers a data consumer to make full use of the operators presented in Sanctuary, and the techniques presented in Polystream, to maximize their query performance and system utilization at optimization time.

In Chapter 2 we present the background information and related work required for understanding the remainder of this dissertation. In Chapter 3, we present our PolyStream system for enforcing access control in a streaming system via Attribute Based Access Controls and Computation-Enabling Encryption. In Chapter 4 we present Sanctuary, our system for using trusted hardware to execute streaming operators on encrypted data. In Chapter 5, we present our optimizer that considers operators from PolyStream and Sanctuary with traditional streaming operators, at optimization time, to create more efficient query plans. Finally in Chapter 6 we summarize the contributions of this dissertation.

2.0 BACKGROUND

In this chapter, we overview relevant background knowledge on streaming data and computation enabling encryption assumed throughout the remainder of this dissertation. Specifically, we overview traditional data stream processing via the Continuous Query Language [10] and how it differs from traditional database languages such as SQL. We further detail the concept of computation-enabling encryption and the four main encryption techniques used throughout this dissertation. We overview Intel's Software Guard Extensions (SGX) within the context of this dissertation. Finally, we provide the base threat model assumed through this dissertation.

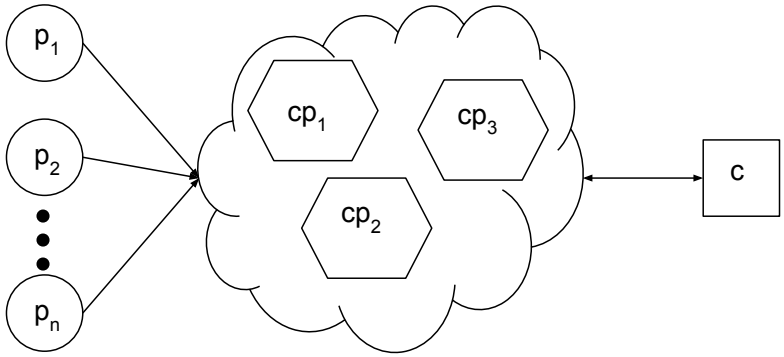


Figure 1: Our assumed Data Stream Processing System model.

2.1 CONTINUOUS QUERY PROCESSING

Traditional relational database models (such as those that use SQL) execute on data-at-rest, often stored on hard-disk or contained within in-memory data-stores. When interested

parties wish to query the data, they execute queries on the stored data. Those queries, transient in nature, undergo optimizations designed to reduce the cost of accessing stored data in some facet, depending upon the access type and data. Continuous query processing differs in that it switches the state of the data with the query. In a Data Stream Processing System (DSPS), data becomes transient and queries become long-running, often stored as routines or functions. Typical DSPSs, like those represented in Figure 1, are comprised of three main components; 1.) data providers, 2.) computing nodes, and 3.) data consumers.

- Data providers stream data tuples as well as author access controls over their data tuples. Data providers can author changes to access controls at any time, as well as change the frequency or selectivity of the data they produce.
- Computing Nodes are responsible for the execution of the operations comprising continuous streaming queries provided by the Data Consumers on the data streams provided by the Data Providers. Computing Nodes are simply hardware made available as a place for the Data Consumer to submit their streaming queries. Computing Nodes can either be hardware owned by the Data Consumer (e.g., a private server or personal computer), or, as we explore in this dissertation, hardware owned by a Cloud Computing Provider (represented by the cloud in Figure 1, where each internal component is a separate piece of hardware). A data consumer will sign a service contract with each Cloud Computing Provider to ensure queries are processed correctly. This is important as we will assume an honest-but-curious adversary [11] who will honor the contracts that they sign, but will try to gain knowledge about the data they process by reading it and making inferences. Honest-but-curious adversaries will not alter, drop, or add data that they serve. Rather, they will only try to learn something from the underlying data.
- Data Consumers are responsible for authoring and submitting streaming queries (further discussed in the next section) on data streamed by the data providers. Data consumers are further responsible for optimizing and placing queries given the accesses granted and the encryption used.

Data flows from left to right in Figure 1, from each of the Data Providers, through the Cloud Computing Provider based Computing Nodes, and into the Data Consumers.

Continuous queries get submitted from the Data Consumers back to the Cloud Computing Providers. Throughout the remainder of this dissertation, we will be augmenting Figure 1 with further details on how we alter data and queries in the system.

2.2 COMPUTATION-ENABLING ENCRYPTION

Throughout this dissertation, we will be referencing several computation-enabling encryption schemes to enhance operators to execute *directly* on encrypted data. To gain a sufficient understanding of the cryptographic primitives involved in computation-enabling encryption, we first overview AES and Public Key cryptography.

The Advanced Encryption Standard [12] (AES, NIST standard since 2001) is a widely used encryption standard for web processing and data transmission (among many other uses). AES is associated with several block cipher modes (where a block represents segments of the plaintext or consecutive plaintexts) and extensions. We highlight some below for context:

- Electronic Codebook (ECB) Mode: Identical plaintext are encrypted to identical ciphertext. Patterns can be discerned with this mode as an adversary can learn such patterns when observing traffic. This mode is not recommended for general use.
- Cipher Block Chaining (CBC) Mode: Using a random Initialization Vector (IV) for each encryption chain, and further exclusive or's each ciphertext block into the next plaintext block. Messages are padded to a multiple of the size of the block, and using the same IV will create a deterministic output (so, ideally, IVs should be different for each encryption chain).
- Propagating Cipher Block Chaining (PCBC) Mode: Similar to CBC mode, but adds an additional plaintext exclusive or before encrypting. This implies that changes to a ciphertext or a plaintext are propogated to each following block (since they are now dependent via the exclusive or).
- CBC-Mask-CBC(CMC) Mode [13]: As the name implies, this mode will take the result of one CBC *encryption* iteration (i.e., when one block has been executed), pass that through a mask, and into an iteration of CBC *decryption* mode. The mask here is

simply an exclusive or with a chosen mask string. The benefits gained over CBC mode from masking in such a manner is you can change the mask to get a different ciphertext for the same IV, key, and plaintext.

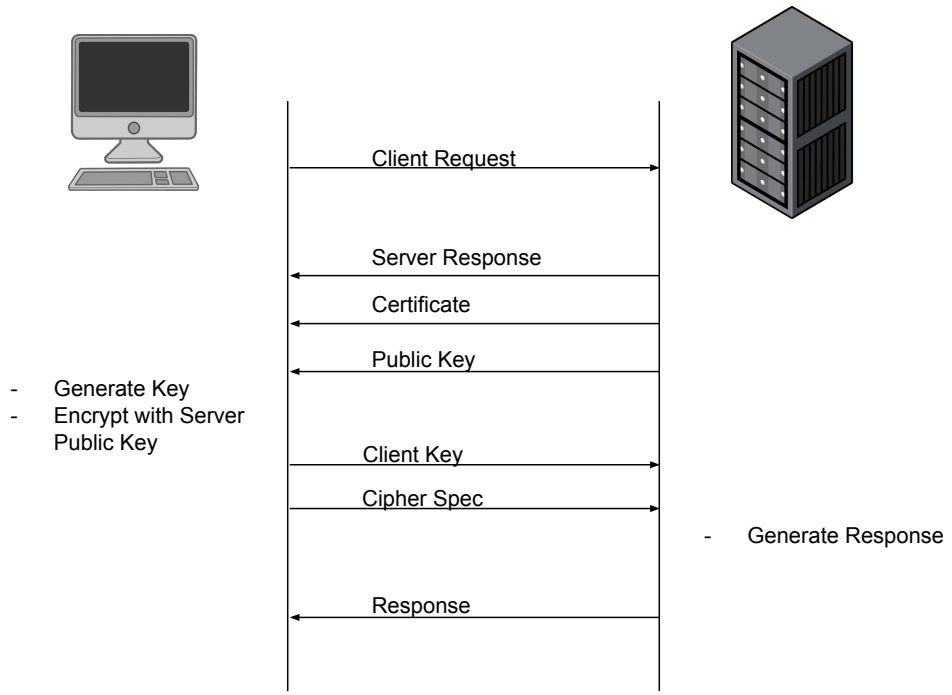


Figure 2: The TLS protocol.

Block ciphers will be used in multiple places in this paper, and the above description should serve as background for why we choose each as needed. In addition to AES and block ciphers, utilize Public Key Encryption schemes. Public key cryptography makes use of both a public and a private key. Depending on the scheme, a pair of keys are generates such that any value encrypted with a public key can be only be decrypted with the corresponding private key. If Alice wishes to send a message to Bob, she finds Bob's public key, encrypts the message, and sends it to Bob. Upon receiving it, Bob decrypts the message with his private key to generate the plaintext.

An implementation of a public key cryptosystem is the TLS protocol [14]. As depicted in Figure 2, the TLS protocol is a simple way for a client and a web server to establish a trusted line of communication. When the client makes a request, the server responds

with a certificate and a public key (that is paired with its private key). The certificate is one generated from a trusted certificate authority (e.g., GoDaddy or Entrust) to help prove the server's identity to the client. The public key is used by the client to encrypt their resulting response message. The client generates it's own encryption key for some other *cipher specification* and encrypts this with the server's public key. The server then decrypts this message, reads the cipher and key, and responds. This establishes a secure connection wherein every message can be encrypted with the specified cipher and key.

Throughout this dissertation, we use the following encryption techniques:

- *Random Encryption* (RND) enforces that $\mathsf{E}_{\mathsf{RND}}(k,\ x) \neq \mathsf{E}_{\mathsf{RND}}(k,\ x)$ or $\mathsf{E}_{\mathsf{RND}}(k,\ x)$!= $\mathsf{E}_{\mathsf{RND}}(k,\ y)$ for $x! = y$. RND uses a block cipher (e.g., AES in CBC mode, with a random initialization vector) to encrypt fields so that no two fields are encrypted to the same value, and does not leak information regarding the correspondence of actual values. Ciphertexts are different even when RND is given the same input for any given value (i.e., multiple AES CBC encryptions of the same value result in different ciphertexts). Given this probabilistic nature, RND is IND-CPA (indistinguishable under chosen plaintext attack) secure.
- *Deterministic Encryption* (DET) enforces the relationship that $x = y$ iff $\mathsf{E}_{\mathsf{DET}}(k,\ x) = \mathsf{E}_{\mathsf{DET}}(k,\ y)$. DET is implemented using a standard cipher (e.g., AES in CBC mode with fixed Initialization Vector (described with AES above)) with some small alterations. Values less than 64 bits are padded, and any value greater than 128 bits is encrypted in CMC mode [13] and sent as 128 bit ciphertexts. Either CMC or CBC mode can be used, but you can use CMC mode for both RND and DET since CMC mode will not leak prefix equalities. This enables equality checking over encrypted values. As shown in [15], DET can be shown to be IND-CCA (Indistinguishable under Chosen Cipher Text Attack), but is not for a chosen plaintext attack as the values would be the same.
- *Order-Preserving Encryption*(OPE) enforces the relationship that $x < y$ iff $\mathsf{E}_{\mathsf{OPE}}(k,\ x) < \mathsf{E}_{\mathsf{OPE}}(k,\ y)$. The OPE scheme used in our system is adapted from Boldyreva et al. [16], where the authors present Order-Preserving Symmetric Encryption. This enables range queries over encrypted data, but only has IND-OCPA (indistinguishability under ordered chosen-plaintext attack) security and therefore can leak the ordering of tuples [17].

- *Homomorphic Encryption* (HOM) enforces the relationship that $\mathsf{D}_{\mathsf{HOM}}(\mathsf{E}_{\mathsf{HOM}}(k,\ x) * \mathsf{E}_{\mathsf{HOM}}(k,\ y)) = x + y$. This allows the execution of summation (and by extension average) queries on untrusted servers without leaking field data values or the summation value. PolyStream uses the Paillier [18] encryption scheme. This enables in-network aggregation of encrypted data without leaking individual data values, but comes at the cost of increasing the computational load of this aggregation. An adversary does learn a relationship for the sliding window, since the encrypted sum for the sliding windows worth of tuples is revealed. Note that a sliding window is simply the range of tuples used to generate a result (e.g., 3 minutes, 100 tuples) over the life of the stream.

2.2.1 Intel's Software Guard Extensions

Intel's Software Guard Extensions (SGX) [19] are a set of architectural enhancements to recent Intel processors that provide developers with the ability to create a trusted environment within an untrusted machine. Specifically, a user can create a measurable and verifiable *enclave* that is executes in a segregated manner from other applications on they system. When an enclave is initiated, it is set up in user space and is separated from other processes and the OS by executing in its own, dedicated, CPU context. All necessary data and the actual code to be segregated needed are stored in protected memory. This protected memory can store data tuples from a stream while they are being processed, and since the memory is protected, the data provider can be assured of its confidentiality, and the data consumer can be assured of its availability. Intel's SGX enclaves provide three key properties:

1. *Isolated Execution*: When an enclave is created, the SGX-enabled CPU creates a hash of the code and data associated with the enclave (to be verified when required). The actual data and code of the enclave is then copied into the Enclave Page Cache (EPC), which is an area of memory that is isolated from untrusted applications or even the OS. This prevents an adversary from accessing any enclave data or enclave code without being the owner of the enclave. Once a user decides to execute the enclave, an Enclave Entry call is made that will activate the enclave and execute the code from a predefined entry point. Once inside of an enclave, the application can make arbitrary calls to untrusted

memory (which allows the application to expand the amount of memory available at execution time to include the size of the available untrusted memory). Trusted memory is limited to just 128MB, making the calls to untrusted memory critical for larger applications (e.g., streaming applications that require large amounts of short-term memory to store transient data). Enclaves, however, are limited to just a small set of trusted calls otherwise.

2. *Sealed Storage*: Inevitably, the enclave will either be switched out of the CPU via context switching, or will finish execution. In the times between executions, SGX provides sealed storage wherein data written to untrusted memory is encrypted with a *sealing key*. This key is generated by this specific CPU for this an individual enclave (to imply that no two enclaves on the same CPU have the same key, and the same enclave on different CPUs will not have the same key) and is inaccessible outside of the enclave context. Therefore, data written to untrusted storage is only recoverable from within the enclave that encrypts it. This would allow a user to have a persistent computation (e.g., an entire or parts of a streaming query) execute with an enclave over a longer period of time as no other process can access any state data. Parts of a streaming query, metadata, or long term streaming data can be kept within an enclave and sealed between executions.
3. *Remote Attestation*: Remote attestation allows a user to validate the identity and authenticity of an enclave and its host. In SGX, this process relies on a special Intel provided *Quoting Enclave*. This enclave uses a key, embedded in the CPU, to generate a *quote*. A quote is simply a cryptographic signature of an enclave that can be verified with an Intel provided public certificate. So long as the signature is valid, the user can trust that their enclave is unaltered. This would allow a querier to verify that their query (or partial query) is executing as expected at any point in time.

There are a couple of notable limitations to SGX enclaves. The first is in the size of the protected memory at 128MB. This small amount of trusted memory forces a user to either process data at a slower rate or to use untrusted memory with some sort of encryption. This is one of the limitations explored in this dissertation. The second notable limitation is the required context switching between enclave and non-enclave processes. When a user enters the enclave, there is no guarantee that the enclave will finish processing before the CPU

must preform another task. This will cause delays in processing, and is explored further in this dissertation. In this dissertation, our use of Intel's SGX focuses on the implementation of streaming queries and their components (operators, described in Section 2.2.2). We will introduce algorithms and structures that operate within the main limitation of Intel's SGX, its limited memory capacity of 128MB. We further show how such implementations can be optimized within a streaming query.

2.2.2 Continuous Queries

Traditional, more ubiquitous database technologies rely on some declarative algebra or relation query language such as the Structured Query Language (SQL [20]) for data processing. Such languages allow the data consumer to query data in a manner that correlates data across sources and data-stores while allowing for the underlying system to optimize the query to realize the best performance possible. While such languages are desirable for streaming systems as well, a direct implementation of any one of them would be insufficient as they do not account for the transient nature of data (i.e., there is no concept of time relative to data availability). Given the transient nature of the *data*, requiring a query to execute on *all of the data collected so far* may require substantial resources and storage. Instead, stream processing adds the concept of a *window* and *slide* to a traditional query language.

A window is simply a slice of time (or a count of tuples) for which the data consumer would like their query to return results. For instance, if the data consumer would like the average stock price for a given stock, they very likely wouldn't want the average stock price over the total life of the stock, but rather the last n minutes (or m tuples). They would therefore submit a relational query that would return results for the last n minutes worth of time, rather than all-time.

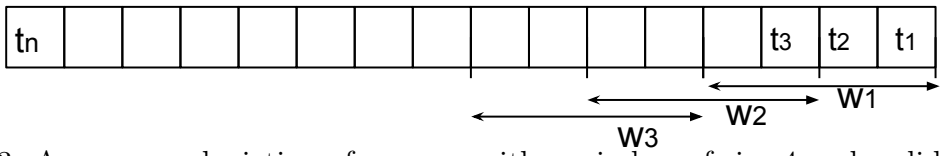


Figure 3: A common depiction of a query with a window of size 4 and a slide of size 2.

Similarly, a slide also represents a slice of time, but dictates how often the data consumer

would like to receive results for *window's* worth of time. For instance, a data consumer could want the average stock price of a certain stock for the last n minutes, but want this value to be updated every m minutes. More concretely, a stock monitoring continuous query could ask for average Google stock for the last four minutes (a window of four) every two minutes (a slide of two), resulting in twenty updates an hour. Figure 3 depicts a given continuous query where a data consumer has submitted a query with a window of size four, and a slide of size two. Notice that the data consumer will get results every two minutes, with each result being the cumulation of the previous four minutes.

Name	Operator Type	Description
`seq-window`	stream-to-relation	Implements time-based, tuple-based, and partitioned windows
`select`	relation-to-relation	Filters tuples based on predicate(s)
`project`	relation-to-relation	Duplicate-preserving projection
`binary-join`	relation-to-relation	Joins two input relations
`mjoin`	relation-to-relation	Multiway join from [VNB03]
`union`	relation-to-relation	Bag union
`except`	relation-to-relation	Bag difference
`intersect`	relation-to-relation	Bag intersection
`antisemijoin`	relation-to-relation	Antisemijoin of two input relations
`aggregate`	relation-to-relation	Performs grouping and aggregation
`duplicate-eliminate`	relation-to-relation	Performs duplicate elimination
`i-stream`	relation-to-stream	Implements `Istream` semantics
`d-stream`	relation-to-stream	Implements `Dstream` semantics
`r-stream`	relation-to-stream	Implements `Rstream` semantics
`stream-shepherd`	system operator	Handles input streams arriving over the network
`stream-sample`	system operator	Samples specified fraction of tuples
`stream-glue`	system operator	Adapter for merging a stream-producing view into a plan
`rel-glue`	system operator	Adapter for merging a relation-producing view into a plan
`shared-rel-op`	system operator	Materializes a relation for sharing
`output`	system operator	Sends results to remote clients

Figure 4: Operators comprising the Continuous Query Language. These dictate the operators that should be implemented by any DSPS, and further describe the relationship between system and stream. More information can be found in [10].

The use of a window and slide allows the DSMS to keep only as much state as is required, and to remove tuples that become too old to remain in the window. This results in a decrease in storage cost, while still offering the data consumer with the results they desire. Any

language that would support streaming data would therefore need to include primitives for defining windows and slides. In this dissertation, we will assume the use of the Continuous Query Language (CQL [10]), a SQL-like language that provides syntax for continuous queries using windows and slides. We will use CQL for any queries presented for optimization or for use in any examples presented.

2.2.2.1 Operators Continuous queries, much like their relational counterparts, are constructed of individual *operators* arranged in a *query network*. Common operators are aggregations, filters, joins, union, exist, and projection. Operators are simply the smallest *logical* working unit for a query. Operators are usually given one task to transform or filter data in some way, with the results being transmitted to the next operator in the query network. Each part of a continuous query can be represented with one or more logical operations.

A logical operator can be implemented using one or more *physical* operators. For example, consider that a data provider has give access to a data consumer that uses the order preserving encryption from Section 2.1. A range query asking for an a $age > 30$ can be implemented by either computing directly on the encrypted data on a third-party system (as we see in PolyStream in Chapter 3), by the data consumer locally on their own machine, or using an SGX enclave (as we see in Sanctuary in Chapter 4). The physical implementation of a logical operator is responsible for accessing raw data, and can vary depending on the algorithm used, the resources available, and the configuration being used. To see how a query can be broken into logical operators, consider the following query from Figure 5. This query wants the average stock price over the last 5 minutes (the window) every three minutes (the slide) reported with some expected price and the id. This relatively simple query can be broken down into the following query network:

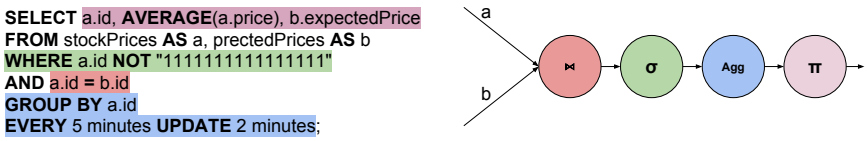


Figure 5: A simple query network representing Query 1.

Here, the query is broken down into a join ($\bowtie$) (colored red in Figure 5), an aggregation

(agg, colored green), a filter (θ) (colored blue), and a projection operation (π, colored pink, see (CQL [10]) for more details of each operator type). They are arranged so that the results of the join are passed to the aggregation, which are passed through the filter and then projected to give the final resulting tuple. Each of these logical operators can be implemented by any number of physical operators. For instance, the join can be implemented using a costly-but-secure nested loop technique (i.e., every tuple from one stream must be joined with every tuple from another stream), or can be implemented using a more efficient hashing technique (i.e., only tuples with matching hash values are joined). Moreover, if the data provider decides to use a deterministic encryption technique, the data consumer can choose to execute directly on this encrypted data. To further expand the pool of possible physical operators, any one of the join, aggregation, or filter can be done using an SGX enclave. The enumeration of such techniques is outside the scope of this dissertation, however, part of this dissertation is exploration of different algorithms and access control techniques that can be used to produce varying physical operators under certain restrictions or assumptions.

2.3 GENERAL THREAT MODEL

Throughout this dissertation, we assume that the untrusted third-party is *honest-but-curious* (an assumption also made in [3], [4], and [5]). An honest-but-curious adversary is one that will not maliciously alter, drop, or add data but will rather try to learn information about the victim by reading and understanding their data. Our general threat model focuses on three main entities:

1. *Data Provider*: Data providers provide streaming data to the system. In addition to streaming data, the data provider authors fine-grained access controls over data (controlling each field). Access controls can be enforced via cryptography and some key management system. Data providers are assumed to trust themselves and their hardware, as well as some subset of data consumers. We assume that data providers do not trust third-party systems employed by data consumers.
2. *Data Consumer*: The primary responsibility of a data consumer is to author continuous

streaming queries that are submitted to third-party data processors. We assume that the data provider will generate access controls that dictate accesses for data consumers, meaning that the data consumer will have a heterogeneous set of access controls.We further assume that contracts with third-party services that dictate data availability standards so that the third-party services are incentivized to maintain data integrity and not drop or alter any client data. Data consumers are further able to provision enclaves securely on third-party systems.

3. *Third-party Computational Systems*: Third-party computational systems process and execute continuous queries from data consumers on data provider's streaming data. These providers are contractually bound to correctly process data and ensure its availability for data consumers. Third-party computational systems can, however, process and view any data passing through their system. While the third-party system will not alter this data in any way, they may try to gain some information about the data that could help them increase profits. For instance, the third-party system could learn spending habits on streams coming into the system and tey to advertise to those end users based on where they are spending their money.

Given the above entities, the adversary of focus in this dissertation is the third-party systems that processes continuous queries for a data consumer on streams generated from a data provider. This adversary is focused on gathering information about the data providers for the goal of increasing profit, but will not alter or drop any data so as to honor contracts signed with data consumers.

3.0 POLYSTREAM

In our exploration of the hidden data, in-network access control enforcement research space, we start by attempting to execute queries directly on the encrypted (and therefore hidden) data with PolyStream [3] [1]. In this chapter, we explore the use of computation-enabling encryption on a data stream processing system. We use the concept of security punctuations along with attribute based access controls (enforced by attribute based encryption) to allow for an online key exchange without altering the underlying stream management system. We further show that we can increase performance and allow for a wider range of queries as compared to the current state-of-the-art through PolyStream.

3.1 INTRODUCTION

Over time, a data provider in a distributed data stream management system may wish to change their access control policies to match changes in the system or their personal preferences. When enforcing access controls over streaming data, modern systems often make use of outsourced third-party systems to cheaply and easily manage continuous queries. Adding access controls and encryption should not limit a data consumer's ability to outsource computation or author meaningful and useful queries over data for which they have been granted access, nor should it greatly impact the performance of these queries either when outsourced or executed locally.

The current state-of-the-art system to solve this problem is Streamforce [4] and although

[1] PolyStream was published in the Symposium on Access Control Models and Technologies (SACMAT) in 2016, Shanghai, China [3]

it addresses many of the issues in enforcing access controls (e.g., updates to access controls on live streams, in network access control enforcement, data this is not visible), it incurs prohibitive overheads, and limits the types of queries that can be issued to the system. A system like CryptDB [5] addresses a similar problem for outsourced databases but does not provide the dynamic online access control and key management protocol required for an ever-changing streaming environment.

We have designed the Polystream framework, which considers data confidentiality and access controls as first-class citizens in distributed DSMSs (DDSMS) while supporting a wide range of query processing primitives and flexible key distribution and policy management. Unlike previous work in this space, Polystream runs on top of an unmodified DDSMS platform; supports a wide range of attribute-based, user-specified, cryptographic access controls; allows dynamic policy updates and online, in-stream key management; and enables queriers to submit arbitrary queries using a wide range of in-network processing options. More precisely, in developing Polystream, we make the following *contributions*:

- In Polystream, access control is based upon a data consumer's cryptographically-certified attributes. Polystream supports Attribute-Based Access Controls (specifically, a large fragment of ABAC_α [21]) and Attribute-Based Encryption (ABE, described in Section 3.2) to enable data providers to write and enforce flexible access control policies over data at the column, tuple, or stream levels.
- Polystream utilizes a modified version of Security Punctuations [22] (SPs) to enforce ABAC policies. SPs are typically used to allow data providers to communicate access control policies to the trusted servers on which users run queries over their streaming data. Prior work in the cryptographic DDSMS space has largely ignored the subject of key management and changes to policy by relying on separate offline systems to handle key and policy distribution. By contrast, Polystream uses SPs to both communicate the policies protecting the contents of a given stream, as well as to provide a key distribution channel for decryption keys that are protected by Attribute-Based Encryption enforcement of ABAC policies. This enables a flexible, online key management and policy update infrastructure, even for stateful continuous queries.
- To the best of our knowledge, no streaming system has allowed in-network processing

of arbitrary queries over protected data streams handled by an untrusted infrastructure. In systems supporting user-specified queries, the data processing servers are typically assumed to be trusted [22, 1, 9, 6, 23]. In systems processing data over untrusted infrastructure, cryptographic protections are enforced such that data consumers have only limited query processing abilities [4]. By contrast, Polystream's key management infrastructure allows untrusted compute nodes to process equality, range, and aggregate queries, and also has limited support for in-network joins.

- Polystream is not, itself, a DDSMS. Rather, it provides an access control service layer on top of another DDSMS. SPs are processed by Polystream and are obtained via long-running selection queries on the underlying DDSMS. Queries are submitted via Polystream, rewritten, and deployed using operations already available from the underlying DDSMS, thereby requiring no changes to the system.

Polystream provides an API that sits between end users (i.e., data providers and data consumers) and an underlying Distributed Data Stream Management System (DDSMS). No changes to the underlying DDSMS are required for Polystream to work. Instead, Polystream makes uses of common functions provided by all DDSMSs. Specifically, DDSMSs provide the user with an *optimizeQuery* function that takes a CQL query, parses it into operators, optimizes it, and returns a query plan, and a function *submitQuery* that places the query. DDSMSs also provide a *results* function that yields the result of a query.

For Polystream, we augment our System and Threat models from Chapter 2 to add an Attribute Authority. *Attribute Authorities* (AAs) are trusted to correctly issue attribute credentials to entities within the system. There may be many AAs in the system, which can vary in the scope of attributes that they will certify. AAs are the master secret key holders of the ABE system and, as such, are responsible for creating ABE decryption keys for the entities whose attributes they certify. AAs are also responsible for the revocation of attributes once a user's attributes have changed, which is outside the scope of this dissertation. The literature has explored attribute revocation [24, 25, 26] and interested readers are encouraged to explore further. The attribute authorities verify and certify the attributes of system components. The scope of an AA may vary: while one AA may exist to certify the job titles or roles of employees within a company, others may certify attributes that cross-cut many

organizations (e.g., ABET accredits many universities). One system can have many AAs, and individuals may choose which AAs they trust.

Section 3.3 describes the design and implementation of Polystream, which is then experimentally analyzed in Section 3.4. Finally, we present our conclusions and directions for future work in Section 3.4.

3.2 RELATED AND PRELIMINARY WORK

This section outlines the related work as well as the primitives necessary for understanding Polystream.

3.2.1 Related Work

FENCE [1] is a streaming access control system that trusts third parties to enforce access controls. Nehme et al. introduced the concept of a *Security Punctuation* for enforcing access control in steaming environments [22]. A Security Punctuation (SP) is a tuple inserted directly into a data stream that allows a data provider to send access control policies and updates to the stream processing server(s) where access controls are to be enforced.

Carminati et al. provide access control via enforcing Role Based Access Control (RBAC) and secure operators [9, 6, 7]. Operators are replaced with secure versions which determine whether a client can access a stream by referencing an RBAC policy. Their work assumes a trusted and honest server that enforces their access control policies. In [6], the authors extend this work to interface with any stream processing system through the use of query rewriting and middleware, as well as a wrapper to translate their queries into any language accepted by a DSMS.

Ng et al. [27] allow the data provider to author policies over their data. The system uses the principles of limited disclosure and limited collection to limit who can access and operate on data streams, requiring queries to be rewritten to match the level at which they can access the data. Their system requires changing the underlying DSMS and therefore is

not globally applicable, and it also requires a trusted server to rewrite the queries.

Lindner and Meier [2] focus on securing the Borealis Stream Engine [28]. They introduce a version of RBAC called owner-extended RBAC, or OxRBAC which operates over different levels of stream objects. OxRBAC allows for each object to have an owner, as well as allowing for rules and permissions. Owners are allowed to set RBAC policies over their objects, which the system will enforce. Objects include schemas, streams, queries, or systems. Users are limited to RBAC policies and must trust the server to enforce their policies as well as see their data in plain text.

Unlike the aforementioned work, Streamforce [4] does not trust the stream processing infrastructure to enforce access control and instead relies on cryptography. Streamforce assumes an untrusted, honest-but-curious DDSMS and utilizes Attribute-Based Encryption (ABE) to enforce access control. The data provider will encrypt their data based on what attributes they desire a potential data consumer to possess. Streamforce is able to enforce access control over encrypted data through the use of their main access structure, *views*. Views are submitted by the *data provider* to the (untrusted) server as a query and only those results are returned to the data consumer. The use of views in this system requires the data provider to be directly involved in the querying process, which has the consequence of limiting what a querier can do with the permissions they were given. Streamforce's use of ABE results in large decryption times depending on the number of attributes. In order to reduce the cost on the data consumer's end, Streamforce outsources decryption to the server [29, 30]. However, even with outsourced decryption, Streamforce reports up to 4,000x slowdown compared to an unmodified system due to their extensive use of ABE. Streamforce also requires the *data provider* to execute all aggregates locally, which may not be feasible since the provider may be a system of sensors, or simply a publish/subscribe system. Finally, Streamforce requires an offline key management solution which makes it hard to reason about key revocation and policy updates.

CryptDB [5] allows computation over encrypted data on an untrusted honest-but-curious relational DBMS. CryptDB's primary goal is not access control, but rather allowing computation over encrypted data stored on an untrusted third-party database system. Essentially, CryptDB offers protection from honest-but-curious database administrators through the use

of encryption, but does not offer fine-grained access controls over the data stored on the system, nor does it offer a key management mechanism since the data owner is in direct control of who can access their data and can change keys at will. CryptDB utilizes specialized encryption techniques for allowing queries to operate on untrusted servers over encrypted data. Specifically, CryptDB employs Deterministic, Order-Preserving, Homomorphic, Specialty Search, Random, and Join encryption techniques to enable many different queries to operate. CryptDB uses *onion* structures to store data, in which data is encrypted under multiple keys: the outer layer of the onion is the most secure, and successive layers provide more functionality (i.e., allow for queries to be executed), but may leak some data. The use of onions as tuples in a streaming system would lead to unnecessary encryptions and decryptions as not all encryption levels are required (cf. Section 3.4). MONOMI [31] extends CryptDB to allow the querier to also processes queries to provide a broader range of queries to the users.

3.2.2 Cryptographic Primitives

We now overview the basic encryption techniques that will be used in the coming sections. We use two main types of encryption: computation-enabling and attribute-based encryption.

3.2.2.1 Attribute-Based Encryption Attribute-Based Encryption (ABE) is used to encrypt data such that only entities with the proper certified attributes can decrypt a given ciphertext. In an ABE system, an Attribute Authority (AA) holds a master key that can be used to generate decryption keys tied to an individual's attributes (e.g., *Professor* or *Orthopedist*). Encryption requires only public parameters released by the AA and a logical policy p in addition to the data to be encrypted, while decryption requires attribute-based decryption keys provided by the AA. The following functions comprise an ABE system:

- **GenABEMasterKey()**: Generates a master key MK.
- **GenABEPublicParamaters(MK)**: Generates the public parameters pa_p needed for encryption.

- **GenABEDecryptionKey**(UA_{user}, MK): generates a decryption key k_d for $user$ based on their set of attributes UA_{user}.

- $\mathbf{Enc}_{ABE}(pa_p, p, d)$: generates an ABE encrypted ciphertext c with the public parameters, a logical policy p, and the data d.

- $\mathbf{Dec}_{ABE}(p, k_d, c)$: recovers the data d using the ABE decryption key k_d, the policy p, and the ciphertext c.

Note that the first three functions are executed by the AA. Meanwhile, $\mathbf{Enc}_{ABE}(pa_p, p, d)$ can be executed by any entity, as it relies only on public information, while $\mathbf{Dec}_{ABE}(p, k_d, c)$ can be executed by any entity with an ABE decryption key k_d.

Recall from Chapter 2 the four different encryption techniques used in this dissertation. Polystream combines those techniques with Attribute Based Encryption to generate the following functions:

- $\mathbf{GenKey}_{DET,OPE,RND}()$: Generates a symmetric key k corresponding to the technique used.

- $\mathbf{Enc}_{DET,OPE,RND}(k, d)$: Encrypts data d with key k.

- $\mathbf{Dec}_{DET,OPE,RND}(k, c)$: Decrypts ciphertext c with key k.

The Pallier homomorphic cryptosystem does not rely on a single key, but rather a pair of (public) encryption and (private) decryption parameters. For the purposes of this paper, we represent this functions parameterizing this cryptosystem as follows, and refer the reader to [32] for more information:

- $\mathbf{GenKey}_{HOM}()$: Generates a encryption parameter pa_{HOM} and a private parameter pp_{HOM}.

- $\mathbf{Enc}_{HOM}(pa_{HOM}, d)$: Encrypts data d with key pa_{HOM}.

- $\mathbf{Dec}_{HOM}(pp_{HOM}, c)$: Decrypts ciphertext c with key pp_{HOM}.

3.3 POLYSTREAM

We now overview the Polystream system. First, we introduce the access control framework that a data provider can use to describe and author policies. We then detail the online policy distribution and cryptographic key management channel used to communicate and enforce the access control policies. We also detail how data consumers' queries are handled in the Polystream system.

3.3.1 Access Control Model and Mechanism

Given the dynamic nature of real-time data stream processing systems, data providers, data consumers, and compute and routing nodes are likely to join and leave the system over time. This inhibits a data provider's ability to have a full understanding of every entity acting in the system. As such, Polystream makes use of attribute-based policies to help data providers protect their sensitive data in a more generalizable manner.

Access Control Model Attribute-based access controls allow a data provider to *describe* authorized consumers of their data, rather than listing them explicitly. Polystream makes use of a large fragment of the ABAC_α [21] model. An ABAC_α system is comprised of the following state elements:

- Sets U, S, and O of users, subjects, and objects
- Sets UA, SA, and OA of user attributes, subject attributes, and object attributes

Furthermore, ABAC_α makes use of the following grammar for specifying policies:

$$
\begin{aligned}
p ::= \quad & p \wedge p \quad | \quad p \vee p \quad | \quad (p) \quad | \quad \neg p \quad | \\
& \text{set} \quad \textit{setcompare} \quad \text{set} \quad | \quad \text{atomic} \in \text{set} \quad | \\
& \text{atomic} \quad \textit{automiccompare} \quad \text{atomic} \\
\textit{set} ::= \quad & \textit{set}_{sa} \subseteq SA | \textit{set}_{oa} \subseteq OA | \textit{set}_{ua} \subseteq UA \\
\textit{setcompare} ::= \quad & \subset \mid \subseteq \mid \not\subset \\
\textit{atomic} ::= \quad & \textit{attribute} \in SA | \textit{attribute} \in OA | \textit{attribute} \in UA \\
\textit{atomiccompare} ::= \quad & < \mid = \mid \leq
\end{aligned}
$$

$$attribute ::= \quad < string >$$

In Polystream, the set U is comprised of all entities acting in the system (i.e., data providers, consumers, and third-party providers). The set O contains pairs (t, ℓ) containing all tuples t being processed by the underlying DDSMS (i.e., data fields or streams), and the access level ℓ at which they should be protected. Polystream supports four such access levels, corresponding to the type of in-network processing that will be allowed: NONE (no in-network access), SJ (in-network selection and join), RNG (in-network range queries), and AGG (in-network aggregation). While there are no explicit subjects in Polystream, queries issued by a data consumer can be given access to a limited set of the issuing user's attributes. As such, S is comprised of the long-running queries submitted by data consumers.

Data producers use the ABAC_α policy grammar to author protections over the data that they supply to the DDSMS. In PolyStream, we will use the shorthand $(q \wedge r) \vee s$ to express a policy of the form $(q \in UA \wedge r \in UA) \vee s \in UA)$, since all policies are written as constraints over the set UA of user attributes that must be possessed by an authorized data consumer (and thus by the query subject operating on their behalf). Note also that Polystream does not make use of the atomic operators for $<$ and $\leq$, since our underlying ABE library supports only string attributes.

Enforcement Mechanism Unlike most stream processing systems, in Polystream, third-party systems are not trusted to correctly enforce data provider access controls. As such, we enforce ABAC_α policies cryptographically by encrypting data prior to introducing it to the DDSMS. Recall that Polystream supports four access permissions: NONE, SJ, RNG, and AGG. We now describe each in more details, and discuss how cryptography can assist in the enforcement of these permissions.

- **NONE**. This permission prevents all in-network processing. To enforce the NONE permission for a tuple t, we simply encrypt t using a randomized cryptosystem (e.g., AES in CBC mode) prior to transmission. That is, given a session key k, we transmit ciphertext $c = \mathsf{E}_{\mathsf{RND}}(k,\, t)$ to the DDSMS. Intermediate third-party systems cannot glean any

information about the contents of this ciphertext, but authorized consumers can decrypt it upon receipt.

- **SJ**. This permission allows in-network selection and joins of streams sent by the same data producer. To enforce the SJ permission for a tuple t, we encrypt t using a deterministic cryptosystem (e.g., AES in CMC mode) prior to transmission. Given a session key k, we transmit the ciphertext $c = \mathsf{E}_{\mathsf{DET}}(k, t)$ to the DDSMS. Since the same plaintext value will always encrypt to the same ciphertext value, untrusted third-party systems can carry out selection on static values or join two streams whose join attributes are encrypted under the same key.
- **RNG**. This permission allows in-network processing of range queries. To enforce the RNG permission for a tuple t, we use an order-preserving encryption scheme (e.g., [16]) and a session key k to transmit the ciphertext $c = \mathsf{E}_{\mathsf{OPE}}(k,\ t)$ to the DDSMS. Given (encrypted) range bounds $l = \mathsf{E}_{\mathsf{OPE}}(k,\ v_1)$ and $h = \mathsf{E}_{\mathsf{OPE}}(k,\ v_2)$, an untrusted CRN can check whether $l \leq c \leq h$ without learning v_1, v_2, or t.
- **AGG**. This permission allows in-network processing of aggregate queries. Enforcement of the AGG permission uses an additively homomorphic cryptosystem (e.g., Pallier [32]) to enable in-network aggregation. Given tuples $t_1, t_2, \ldots, t_n$ and a public/private key pair $\langle k, k^{-1} \rangle$, we compute and transmit
  $c_1 = i\ \mathsf{E}_{\mathsf{HOM}}(k,\ t_1)$, $c_2\ \mathsf{E}_{\mathsf{HOM}}(k,\ t_2)$, $\ldots, c_n = \mathsf{E}_{\mathsf{HOM}}(k,\ t_n)$ to the DDSMS. An untrusted CRN can then compute $c_1 \times c_2 \times \ldots \times c_n = \mathsf{E}_{\mathsf{HOM}}(k,\ s = t_1 + t_2 + \ldots + t_n)$ without learning s or any t_i.

Table 1: Summary of what types of queries and operators are supported by each encryption scheme, as well as what each scheme could reveal to a potential adversary.

Perm	Scheme	Type of Queries	Supported operators	Information Gained by Adversary
NONE	RND	None	None	Nothing
SJ	DET	Equality	Equality Select, Project, Join, Count, Group By, Order by	Equality of attributes
RNG	OPE	Range	Equality Select, Range Select, Join, Count	A partial to full order of tuples
AGG	HOM	Summations	Aggregates over summations	Encrypted Sum for sliding window

Finally, a data tuple d is encrypted with either any of the computation enabling encryption techniqyes and transmitted to the data consumer where it is decrypted using the corresponding decryption key.s

Table 1 summarizes how each permission can be enforced cryptographically, as well as the DDSMS operations enabled by the permission. Note that Polystream only supports encrypted joins on streams that are DET-encrypted under the same key. In principle, this likely means that joins are only possible over streams published by the same data producer. Supporting a richer variety of joins is covered in Chapter 4.

Although the above constructions enable in-network processing, they do not enable attribute-based control of these access permissions. To cryptographically enforce ABAC_α policies over objects in Polystream, we make use of attribute-based encryption to ensure that the session keys used above can only be recovered by authorized data consumers. In particular, consider an ABAC_α policy p authored over attributes issued by some authority AA_i whose public parameters are pa_i, and a session key k used to enforce one of the above four access permissions over some data tuple. In this case, the data producer can transmit $\mathbf{Enc}_{ABE}(pa_i, p, k)$ to authorized data consumers. Authorized consumers can then decrypt the session key k, which can be used to access protected data tuples. The exact mechanics of this policy distribution and key management process will be discussed next.

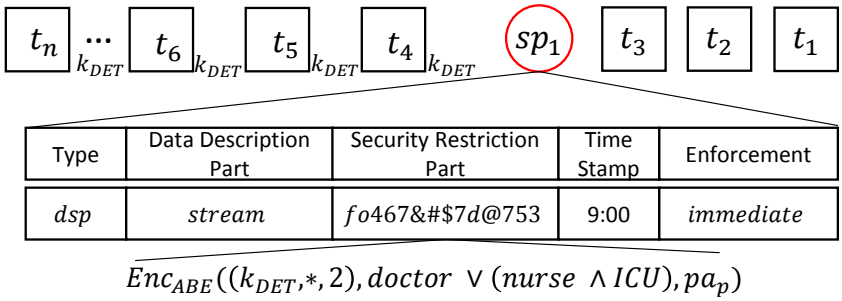


Figure 6: A typical Security Punctuation with an example use case.

3.3.2 Policy Distribution

In a DDSMS, data providers do not control the paths taken by their data. As such, distributing, updating, and enforcing policies protecting that data take some effort, particularly if the infrastructure itself is only semi-trusted. Security Punctuations (SP) [22] address this issue by providing a mechanism for distributing policy along *with* data. A SP is simply a tuple

injected into a provider's data stream (represented as a circle in Figure 6) that describes an access control policy over some set of protected data. For Polystream, a SP dictates the ABE-enforced ABAC policy or policies protecting a stream to potential consumers. SPs are comprised of the five fields below (and the top box in Figure 6):

- *Type*: Indicates that the SP originated from a data provider.
- *Data Description Part*: Indicates the schema fields (e.g., "heart rate") within a tuple that are protected by this policy. This may be as broad as an entire stream, or as specific as an individual field.
- *Security Restriction Part*: Describes the policy being enforced.
- *Timestamp*: The time at which the tuple was generated.
- *Enforcement*: Either *immediate* or *deferred*. Immediate enforcement applies the new policy to tuples in buffers, whereas deferred enforcement applies the new policy only to tuples timestamped after the SP.

While prior work has used SPs to distribute plaintext policies for enforcement by a trusted DDSMS, Polystream makes use of SPs as a policy and key distribution mechanism, but relies on cryptography for policy enforcement. This means that while the *type*, *data description part*, *timestamp*, and *enforcement* fields are straightforward, the structure of the *security restriction part* (SRP) requires greater explanation. Polystream uses the SRP field to transmit a tuple $\langle c, p \rangle$ where p is the ABAC_α policy protecting access to the fields listed in the data description part, and c is an ABE ciphertext generated by encrypting the following three pieces of information:

- *Access Type:* The type of in-network permission (i.e., NONE, SJ, RNG, or AGG) allowed by this policy
- *Index:* The position(s) of the data field(s) being protected by the policy, listed in the DDP.
- *Decryption Key:* The symmetric key k used to recover data protected at the NONE, SJ, or RNG levels, or the private key k^{-1} used to recover data protected at the AGG level.

Note that the above index information is needed due to the fact that a given stream may include several copies of a given schema field. For instance, if one policy on a stream grants

AGG access to "heart rate" to some individuals while providing other individuals with SJ access, two copies of the "heart rate" field will be transmitted: one encrypted using Pallier (for AGG access) and one encrypted with AES in CMC mode (for SJ access).

Given an SP with an SRP containing the pair $\langle c, p \rangle$, a data consumer can inspect p to determine whether they possess the attributes needed to decrypt c. If so, decrypting c provides the data consumer with a description of the in-network processing allowed by the policy, the indexes upon which this processing can occur, and the (symmetric or private) key needed to decrypt result tuples. This is enough information to facilitate query planning (*Which queries can I run?*), operator placement (*How can I place physical operators for these queries in the CRN network?*), and results analysis (*How can I decrypt the results that I receive?*).

Key revocation in PolyStream is as simple as updating the access control policy (even the same one again) so that a new key is generated. A key is therefore revoked when a user no longer possesses the proper attributes to satisfy the ABAC policy to get the new key. Data providers can develop their own policy for updating and refreshing keys to satisfy their own needs. Key revocation does not include the revocation of attributes. Attribute Authorities (AA) are responsible for the revocation of attributes so when a user loses possession of an attribute, a new ABE decryption key is issued. This could lead to a time where data consumers can have unauthorized accesses due to a loss of an attribute but still have the key from the last Security Punctuation. This can be protected against by data providers periodically updating the keys that they use to protect their streams, via the Security Punctuation mechanism described previously.

Algorithm 1 SubmitQuery

```
1: Submit query q to DSMS query Optimizer for Plan p
2: for Operation o in qurery q do
3:   if no entry in schemaTable then
4:     Return permission denied
5:   else
6:     retrieve Schema Key k from schemaTable
7:     Encrypt Attribute with k
8:     if o is Filter or Count then
9:       if Filtering on Equality AND
         permissionTable contains "SJ" then
10:        Encrypt value in o with key in permissionTable
11:      else if Filter on range AND
         permissionTable contains "RNG" then
12:        Encrypt value in o with key in permissionTable
13:      else if permissionTable contains "NONE" then
14:        Operator Executes Locally, exit
15:    if o is Sum then
16:      if permissionTable contains "AGG" then
17:        Change o to Multiplication
18:      else
19:        Operator Executes Locally, exit
20:    if o is Average then
21:      if permissionTable contains "AGG" then
22:        Create operations Count, SUM
23:        Create local operation Division for sum/count
24:      else
25:        Operator Executes Locally, exit
26:    if o is Join then
27:      if permissionTable contains "SJ" or "RNG"
         for the same provider then
28:        Encrypt value in o with key in permissionTable
29:      else
30:        Operator Executes Locally, exit
31:    if Other operator then
32:      Operator Executes Locally, exit
33: submit(Q)
```

Algorithm 2 handleSecurityPunctuation

```
1: SRP = Security Restriction Part of SP
2: DDP = Data Description Part of SP
3: Have pa_pr ABE Decryption Key from AA
4: if pa_pr decrypts SRP then
5:   for Fields f in DDP do
6:     Associate f with permission p at index i from SRP
```

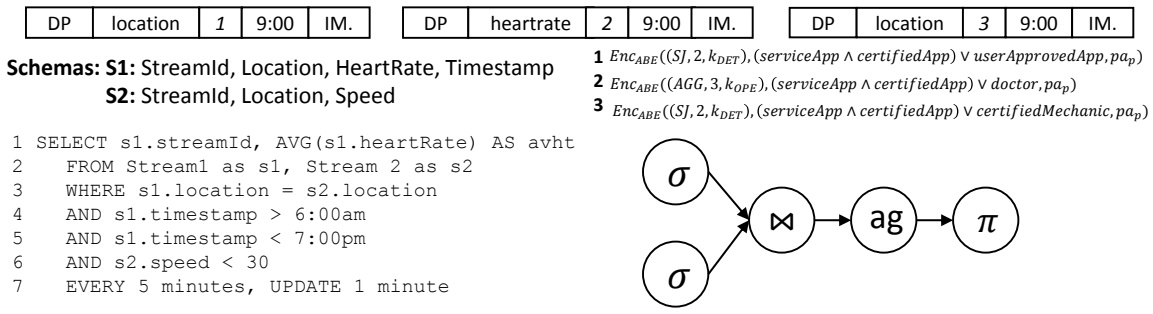


Figure 7: Motivating Example - RoadRageReducer App.

3.3.3 Query Processing

This section overviews how a data consumer can submit and change queries based on policy updates from the data providers they are are interested in. When a data consumer authors a query, they submit it to Polystream, which follows the steps outlined in Algorithm 1. Recall that Polystream sits between an unmodified DDSMS and the data providers/consumers. First, Polystream submits the query to the underlying DDSMS's query optimizer using the DDSMS's own *optimizeQuery* function, and receives back the generated query plan (in the form of a physical operator graph). Using this plan, Polystream iterates through each operation of the plan starting at the data source nodes in the graph (line 2) and determines where the operation must be placed.

Using data extracted from the SRP field of SPs received by the consumer from the data providers, Polystream iteratively checks each operation to see if in-network processing has been enabled by the data provider (lines 8,15,20,26,31) and if access has been granted to the data consumer (lines 9,11,13,16,21,27). If so, this operation can be submitted to a CRN for in-network processing. If in-network processing is not possible, but the consumer has access to the field(s) being operated upon, the operator executes locally on the data consumer's device, where local decryption is possible (lines 14,19,25,30,32). We note that once *any* operator is placed on the data consumer's machine, all subsequent operations are placed on the data consumer's machine as well to avoid unnecessary network round trips. Once all operator placement decisions have been made, the query is submitted using the *submitQuery*

function provided by the DDSMS and results are processed using the *results* function.

Polystream provides a large number of operations that can be executed by third-party providers over encrypted data, including operations that require multiple encrypted streams. For instance, a data consumer who is interested in aggregates over multiple encrypted streams can simply execute an aggregate separately over each encrypted stream and combine the results on their trusted machine once they are decrypted. A data consumer can also preform a join on two encrypted streams so long as they are encrypted with the same DET or OPE key. When the streams are encrypted under different keys, or processing on the CRN is otherwise not possible, Polystream provides functionality similar to MONOMI [31] in that operations are executed on the data consumer's trusted machine after data is decrypted, so long as the data consumer possesses the proper decryption keys. Ultimately, this allows the consumer to issue *any* query for which they at least have decryption capabilities.

There exist alternative approaches to executing a multi-provider joins on the data consumer's trusted machine. One approach is for the data consumer to deploy a trusted node in the CRN network that simply decrypts multiple streams that are to be joined and re-encrypts each using a single symmetric key. This will allow later nodes in the CRN network to handle in-network joins, while minimizing the computational impact on the data consumer. Another approach is to use proxy re-encryption to compute on data even when it is encrypted with different keys [33]. Proxy re-encryption will enable one stream to be joined with another simply by re-encrypting one (still in its encrypted form) so that it is encrypted form matches the other. These techniques are being considered in our ongoing work

Example Consider the scenario presented in Figure 7 with a single data provider, a city commuter, who is producing two data streams. The first stream contains health and location data being produced by a fitness watch linked to a phone, while the second contains location and travel data from her car's on-board computer. Stream 1 is protected by SP_1, which enables in-network SJ processing on the Location field for entities satisfying the policy $p_1 = (serviceApp \wedge certifiedApp) \vee userApprovedApp$ to recover the resulting data. Stream 1 is also protected by SP_2 enabling in-network AGG processing on the HeartRate field for anyone satisfying $p_2 = (serviceApp \wedge certifiedApp) \vee doctor$. Stream 2 is protected by SP_3, which enables in-network SJ processing on the Location field for entities satisfying the policy

$p_3 = (serviceApp \wedge certifiedApp) \vee certifiedMechanic$ to recover the resulting data.

A data consumer, a mobile app called RoadRageReducer (a certified service app), wishes to execute the query shown in Figure 7. This query determines if the commuter has road rage by checking whether they are in their car while their average heart rate is elevated. To reduce the overall workload of the query, only high traffic driving times at low speeds are considered. Optimizing this query using the underlying DDSMS produces the operator graph shown in Figure 7. Given the information recovered from SP_1, SP_2 and SP_3, each of these operators can be placed on the CRN network since (i) the initial selection operates over unprotected fields (Speed and Timestamp), (ii) the join combines both streams using the SJ protected Location field, (iii) the averaging operator aggregates over the AGG protected HeartRate field, and (iv) the only input to the projection operator is a field index. Once the query is processed and a result is returned, the RoadRageReducer app can then use its Paillier decryption key to decrypt the resulting average over the HeartRate field.

3.4 EXPERIMENTAL EVALUATION

Like many other confidentiality enforcement systems, Polystream exposes a tradeoff between performance and confidentiality. To better understand this tradeoff, we examined many different configurations/workloads on an experimental system comprised of a cluster of 10 small instances on Amazon EC2, which implements Polystream as described above. All network communications occur over SSL/TLS tunnels. We also compared Polystream with the current state-of-the-art Streamforce [4].

3.4.1 Experimental Setup and Platform

Our system is built on top of the Storm distributed computing platform [34], as is the case for many other distributed DSMS prototypes/evaluations [35, 36]. Given that we do not use any functionality unique to Storm, we fully expect that Polystream could be trivially ported to other distributed computing platforms like Spark Streaming [37] and Twitter Heron [38].

Storm provides a communication layer that guarantees tuple delivery. Storm accepts user-defined topologies that direct how components are networked. The main components of Storm are *spouts* and *bolts*. Spouts provide data to the system and therefore assume the role of data provider, and bolts compute on the data and take the role of data consumer or third-party system. To better control experiments, a special scheduler was implemented to dictate which machines handled which components.

Tests were run on Amazon EC2 using small instances. All components were programmed in Java and packaged as JAR files. Each data consumer was assigned a set of attributes from a bolt *Central Authority*. One EC2 instance was devoted to controlling Storm's required libraries as well as assigning tasks and was not used in experimentation; leaving nine that were used as third-party systems with data consumers on them. Data providers were generated from outside machines and fed into the cluster so that data generation would not alter the state and load of each machine. Tests involved between one and eight data providers; 1,000 and 8,000 tuples per second input rates; two and 20 data consumers; and two and eight third-party systems. All CP-ABE functionality was provided by the Advanced Crypto Software Collection library [39], and the HOM key size was 1024 bytes.

3.4.2 Workload Description

For our experiments, we used simulated Twitter-like data from a workload generator that provided control over distribution and frequency of keywords as input data. This generator is capable of forming both text and numerical data. Values can be controlled in either a fine-grained or coarse-grained fashion. Fine-grained control allows us to define a small dictionary and assign a distribution over the occurrence of each value in the dictionary. Coarse-grained support simply sets a desired amount of data and desired selectivities (as to control selectivity for windowed and one-shot queries). We chose not to use the Linear Road [40] benchmark for two main reasons. First, adding encryption and policy changes to arbitrary values adds overheads to the actual benchmark and requires altering it, which could undermine the intentions behind the data and queries. Secondly, Linear Road requires compatibility with a traditional database system. In the Polystream model, the database

system may reside on the server (colocated with the data) which can leak data since it would be required to remain in plaintext. A system like CryptDB could be used in this regard, but that says nothing for the methods of Polystream which focuses on data streams.

3.4.3 Overhead for Computation Functionality

To better understand the effect that each operator has on the overall throughput, we compared *unencrypted* versus *encrypted* processing using one encryption type. We also included a *Strawman* approach where all data is routed to the data consumer for processing under the RND encryption scheme.

Configuration One data provider with one stream distributed the data to a single third-party system with a single data consumer. This data consumer posed one query to the stream corresponding to the given encryption scheme (e.g., DET encryption matched to equality select and OPE mapped to range queries). One field was encrypted for each operator. For equality queries, Range, and summation queries DET, OPE, and HOM were used, respectively.

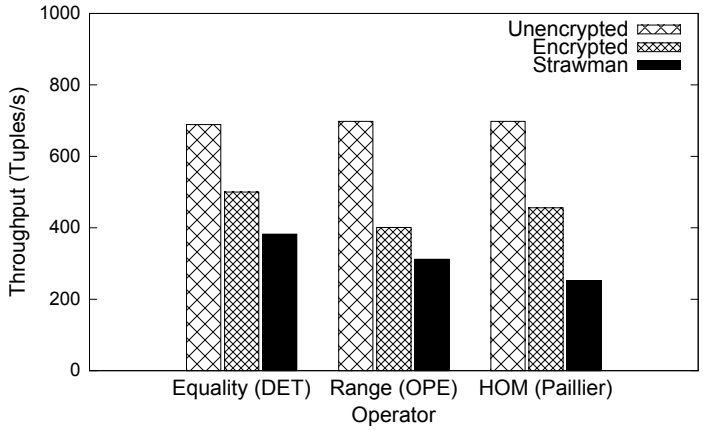


Figure 8: Throughput for each of the different operations supported for both unencrypted and encrypted streams.

Results (Figure 8 and Table 2) On average, deterministic encryption only incurs 12% overhead, whereas HOM decreases throughput by 49% on average. This large difference in HOM is attributed to its use of Homomorphic encryption, which involves costly homomorphic additions running on each third-party system. We also evaluated a more secure scheme

(implemented on the same system) in which RND encryption is used and all tuples are sent back to the data consumer for processing. This requires *every* tuple to be decrypted and the operation computed over the plaintext value. Since every tuple is encrypted, the overall cost of execution is hindered by the cost of decrypting each tuple before processing. The overhead incurred by each encryption scheme originates either from the encryption or the decryption phase of the algorithm. Table 2 shows exactly how much time is spent during each phase of encryption. Note that a summation for the HOM scheme itself takes on average .015ms, and the key size (modulus size) for HOM plays a significant role in its encryption and decryption time. It is also important to note that the system will always pay the encryption cost for every tuple, but may not pay the decryption cost for each tuple depending on the selectivities.

Takeaway Compared to an unmodified DSMS, Polystream's overhead is a modest 28% in supporting access control on honest-but-curious third-party systems. In contrast, the overhead of the state-of-the-art Streamforce is 4,000x, according to the authors [4].

Table 2: The encryption and decryption times (in ms) for each of the schemes used by our system (H-xxxx = HOM at that key size).

Mode	RND	OPE	DET	H-1024	H-2048	H-4096
Encrypt	8.2	13.1	12.5	18.1	70.2	151.8
Decrypt	8.2	13.2	12.3	12.9	21.6	36.5

3.4.4 Effects on Latency per Encryption Type

To explore the perceived effect on waiting for a result based on an incoming tuple, this experiment compared the latency of Polystream to that of the baseline Storm-based DDSMS without any encryption.

Table 3: The latency (ms) of each encryption when used in a query.

System	RND	OPE	DET	HOM
Polystream	425	413	326	1,144
Baseline	356	357	308	485

Table 4: The percentage of system time spent on a task based on the input rate. CP-ABE represents the time spent passing keys and managing attribute-based encryptions.

Tuples/second	*Encrypt*	*Decrypt*	*CP-ABE*	*Compute*	*Transmit*	*Idle*
2,000	3.8	4.0	6.0	**41.2**	9.5	35.6
4,000	3.9	5.3	6.2	**61.7**	10.3	12.6
6,000	3.4	7.2	6.3	**69.0**	12.2	1.6
8,000	3.4	8.6	5.8	**76.2**	16.0	0.0

Configuration This experiment used only one EC2 small instance. One query was used to test each encryption type, and each query was simply a selection (i.e., on comparison) or addition wherein one addition or one comparison needed to be made. The input rate of tuples remained constant. Each query was tested five times with the average reported for 1,000 tuples per trial. Finally, experiments were carried out in succession with the same system setup and background. Results are reported in milliseconds (ms).

Results (Table 3) Table 3 shows the latencies for each type of permission. The main differences between Polystream and the baseline DSMS is in the decryption time on the data consumer. The actual computation on the third-party system is roughly the same (with the exception of HOM) since the operators are only comparing larger integers or strings. HOM, however, takes longer to compute since the integers are larger and require multiplication as opposed to simple summations. Note that the HOM latency is calculated as the arrival of the first tuple in a window until the time the resulting summation is outputted. For this experiment, the window size was five tuples.

Takeaway Summation or averaging queries incur larger delays due to the need for multiplying larger numbers (a costlier operation) to homomorphically sum tuples using the Paillier [32] scheme.

3.4.5 Total System Overview

Given that each encryption scheme yields an overhead, it is worth exploring exactly what percentage of system time is devoted to doing a given task. We consider six main tasks when examining where the system spends its time: encrypting, decrypting, attribute alterations (CP-ABE), computing, transmitting, and waiting.

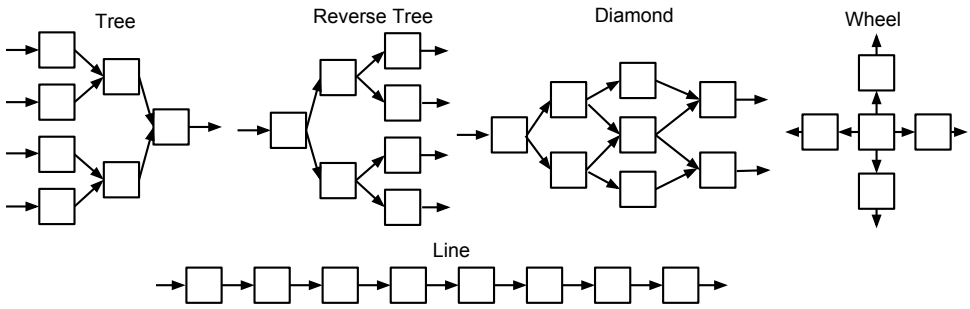


Figure 9: Configurations used to test how network topology affects Polystream.

Configuration The results are based on an hour-long simulation where over 40,000 tweets were generated, and 600 changes in policy were assigned. The worker nodes were in a wheel configuration (Figure 9) with each leaf sending data to a sink (a bolt which receives and deletes data) to emulate retransmission. We used a mixed query workload, consisting of equality, range, and summation queries (33% for each type). Since the workload depends largely on the input rate, results are given for different input rates. In the event that the machines became overwhelmed, a typical simple load shedding technique was used [41].

Results (Table 4) For all experiments, over 70% of the time was spent on computation or idling if the workload was light. The time spent on attribute alterations and the time spent encrypting stays relatively constant throughout the simulation. The system spends more time decrypting as the workload increases since more tuples are sent. The wait time of 0.0%, for the 8,000 tuples/sec case, indicated a system saturated with tuples and, as such, some tuples were dropped (4.9% of tuples).

Takeaway Polystream spends on average 15-17% of the total time in encryption, decryption, and key management.

3.4.6 SP Frequency vs. Throughput

A change in policy can occur at any time. Next, we evaluate how the frequency of policy changes effects the overall latency.

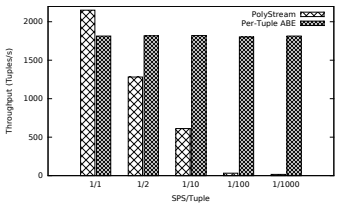
Configuration We used two machines, each with two data consumers. The frequency of policy changes is determined by the frequency of inserting SPs into the stream. We compare against using ABE encryption for all tuples, similar to the state-of-the-art [4]. The number of attributes was fixed at five, with a mixed query workload of equality, range, and summation queries (33% each).

Results (Figure 10a) Results are depicted in Figure 10a. Note that the per-tuple ABE uses outsourced decryption. These experiments show that Polystream was better than per-tuple ABE for all cases except the degenerate case of one SP per data tuple. Polystream performed well when changes in policy are infrequent. It is clear that Polystream outperforms the state-of-the-art [4] in even the simple case of one policy update for every two tuples, while providing more flexibility in submitting queries.

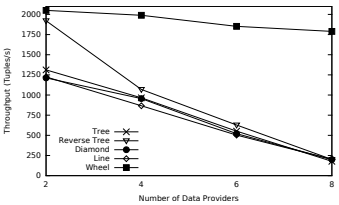
Takeaway Given a ratio of 1/100 (data tuples to SPs, likely higher than in practice), Polystream outperforms Streamforce by over 40x.

3.4.7 Tuple vs. Punctuation Level ABE

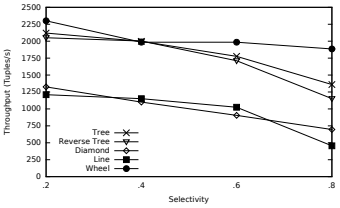
The implementation of the current state-of-the-art, Streamforce [4], uses decryption outsourcing techniques from Green et al. [29] to outsource decryption of Attribute-Based Encryptions to the cloud. Through the use of a transformation key, the server (third-party system) is able to aid in decryption by doing most of the decryption, leaving only a small decryption operation to the data consumer. For *every tuple* selected by the system, however, a full attribute-based decryption must be done, which is costly regardless of whether or not it is done on a server. This means the number of ABE decryptions in Streamforce is large when compared to Polystream which only uses ABE for policy updates (SPs). To test the effects of outsourcing attribute-based decryption to the cloud, we implemented the scheme used by Streamforce to compare our key distribution approach with their attribute-based approach.



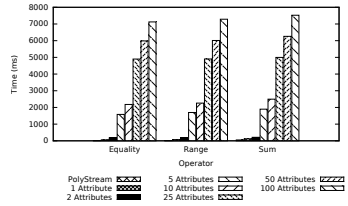
(a) Effect on average throughput by altering the frequency of Security Punctuations.



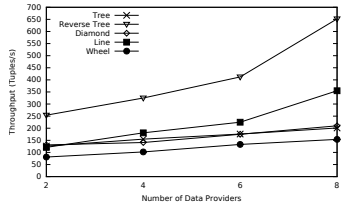
(b) Total throughput for increased load with selectivity .8, two clients per CRNs, and all encryption types used.



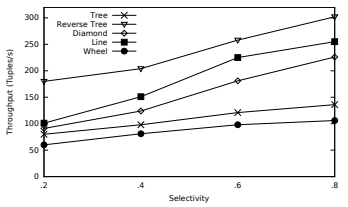
(c) Total throughput for increasing selectivity where there is one data source, two clients per CRN, with only DET used over selection queries.



(d) Outsourced ABE decryption for different operators with different numbers of attributes. Note that Polystream decryption are included as the first column per set.



(e) Total latency for increased load with selectivity .8, two clients per CRNs, and all encryption types used.



(f) Total latency for increasing selectivity where there is one data source, two clients per CRN, with only DET used over selection queries.

Figure 10: Network Configuration, SPS frequencey, and Encryption Technique Effects on Throughput and Latency

Configuration Streamforce used four different queries ranging from simple selections to summations. To compare their results with ours, the total decryption time is taken as the transformation time plus the decryption time performed on the data consumer. The total decryption time for Polystream is simply the faster cryptographic scheme decryption time, which averages to 13.2 seconds, as mentioned above. Since Polystream only uses ABE to share keys (i.e., only when a Security Punctuation is issued and processed), it does not pay the cost of ABE decryption on every tuple; instead, it only pays the cost once per SP, as

described above. One query was used, along with one stream on one machine. Note that the ABE decryption time depends on the number of attributes, so results are given for different numbers of attributes. Also, note that in this experiment the only comparison drawn between Streamforce and our work is Streamforce's use of ABE for each tuple. Streamforce relies on the data provider to do aggregates rather than the server, and the deterministic encryption and summations are the same as the ones used in Polystream, so they were excluded.

Results (Figure 10d) Even with the smallest number of attributes, outsourced ABE is 4x slower than the Polystream approach, and at one point it is nearly 550x slower depending on the number of attributes. These results are in line with initial results from Green et al. [29], which were on similar, yet better, hardware.

Takeaway By using ABE only for key management (i.e., not for every tuple), Polystream incurs up to 550x less overhead per tuple than Streamforce.

3.4.8 Network Effect on Throughput & Latency

Configuration (Figure 9) Storm enables the user to describe the configuration of the network interconnecting the worker nodes. To better see how network connections affect the system, we tested five configurations with different input rates, data consumers, selectivities, and third-party systems. These five configurations consisted of a tree, a reverse tree, a line, a diamond, and a wheel.

Throughput Results (Figures 10b, 10c) The first network experiment measured the throughput with respect to the workload. Each configuration had all of the worker nodes running. Figure 10b depicts the results. As the workload increases for each configuration, there is a corresponding drop in throughput. The wheel configuration is less affected as there is no single bottleneck whereas each other configuration has at least one bottleneck where multiple streams meet at a third-party system. The throughput is not just a factor of the workload, it is also a factor of selectivity and the number of worker nodes. Only deterministic selection queries were used in this experiment. Figure 10c shows the effects of selectivity on throughput for each configuration. The results are similar to the increase in workload, but the trees have a higher throughput since they reduce the number of tuples at

each stage due to changes in selectivity.

Latency Results (Figures 10e, 10f) Figure 10e shows that the reverse tree incurs the highest latency. Again, the output node becomes the bottleneck, causing delays to compound as the number of tuples increases. The wheel configuration preforms the best since there is no delay getting data consumers. Figure 10f shows the effects on latency when the selectivity of operators increases. Networks that reduce the number of third-party systems as data flows tend to do worse as the workload increases. This verifies that Polystream does not incur unnecessary overheads that would not appear otherwise.

Takeaway Network configurations have an impact on latency and throughput since delays compound depending on the encryption types and selectivities.

3.4.9 Overhead of Analytical Queries

Analytical queries can be more costly than regular queries when summation is involved. We explore these next.

Configuration For analytical queries, we used an equal mix of range and summation queries for a total of 100. Range queries had a selectivity of 0.5. Ten queries were registered to each of ten data consumers who were assigned two per machine in a wheel pattern (see Figure 9). The same data was used for the non-analytical queries, but all query types were included to show how throughput was affected. Analytical queries were simply summations over a fixed window and filters over a fixed window, whereas non-analytical queries were equality filters and plain-text joins.

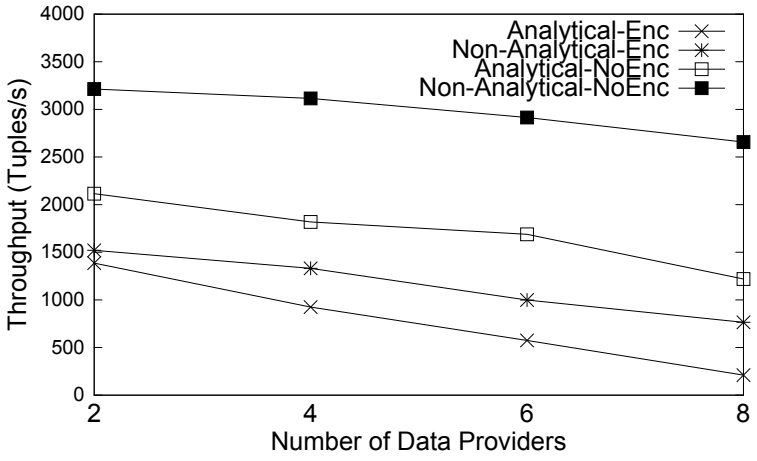


Figure 11: Effects of encrypted analytical workloads versus encrypted non-analytical.

Results (Figure 11) Figure 11 shows the throughput for an analytical query-heavy workload and a non-analytical-query-heavy workload. Analytical queries must use the Paillier [32] encryption scheme, which requires large integer computations to be done on the server, resulting in the slowdown depicted in Figure 11.

Takeaway Analytical queries require multiplication of large numbers and will incur larger overheads than simpler queries.

3.4.10 Encryption Overhead Comparisons

Configuration Here, we introduce CryptDB [5] as adapted for a streaming environment. CryptDB [5] and Polystream utilize many of the same tools to accomplish their goals, although they were designed for very different system needs: CryptDB operates on traditional Database Management Systems, whereas Polystream operates on DDSMSs. CryptDB's primary goal is not access control for all parties, but rather eliminating unwanted access by third-party storage systems by allowing computation over encrypted data on the untrusted third-party database. CryptDB utilizes specialized encryption techniques for allowing queries to operate on untrusted servers over encrypted data. Specifically, CryptDB employs Deterministic, Order-Preserving, Homomorphic, Specialty Search, Random, and Join encryption techniques to enable many different queries. Each technique leaks a different level of in-

formation (discussed in Section 3.3.3) but allows for different levels of functionality. These different techniques are structured in "Onions" in which the outer layer contains the most secure encryption technique. Removal of layers allows more functionality (i.e., going from RND to DET), but leaks some sensitive data.

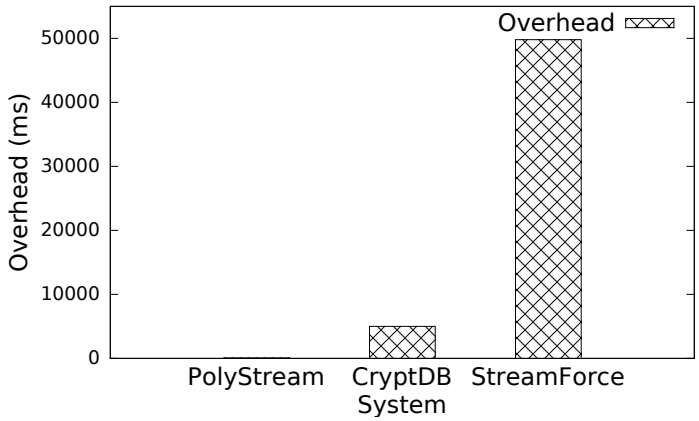


Figure 12: Comparisons between Polystream, CryptDB (in a streaming environment), and Streamforce.

When considered for use in a DDSMS, CryptDB encounters a few limitations. First, the data consumer no longer has control of the data source, meaning they do not control the encryption being used, or the accesses being given (including whether they themselves have access). This requires an online key management system as well as knowledge of what types of encryption are required for each potential data consumer, and an access control mechanism for different end users. In the system model described above, one data provider can have many data consumers digesting their data. Each data consumer may require a different level of encryption for processing.

We implemented a micro-benchmark to show the average overhead incurred by using onions in a streaming environment. This benchmark consisted of three onions (all those from CryptDB minus searches and joins) for a simple schema of four fields: *Name*, *HeartRate*, *StepsTaken*, and *Glucose*. Each field was onion-encrypted, resulting in 12 fields. Between 2 and 12 fields were chosen at random to be decrypted to a random level, for 10,000 tuples.

Results (Figure 12) The average overhead from decryption was 51.5ms per tuple for the

stream adaptation of CryptDB. This means a DSMS that could handle 10,000 tuples per second would be reduced to 194 tuples per second, hindering the useful work being done by 98%, and causing an increase in encryption overhead of nearly 5,000%. These overheads and the need for an access control element limit the use of CryptDB in a streaming environment. Polystream avoids these overheads by simply encrypting data at one level and by avoiding re-encryption. Also note that using CryptDB for a streaming application would cause greater overheads, due to a large number of insertions into the database and frequent query re-execution to get up to the date results. Both of these overheads are not explored here.

In addition to these overheads, recall that the encryption overhead for Streamforce causes a 4,000x slowdown on an unaltered system (as claimed in [4]). Our experiments from Section 3.4.6 show that a workload with just 5 attributes would incur at least 49,000% overhead for every tuple. Note also that from Section 3.4.6, Polystream with a relatively low policy update rate can incur as little as 12% overhead attributed to encryption, but will average roughly 56%. These overheads are displayed in Figure 12.

Takeaway Polystream incurs very little overhead versus the closest related work.

3.5 POLYSTREAM DISCUSSION

PolyStream is designed to allow *in-network* access control enforcement through computation-enabling encryption, security punctuations, and Attribute Based Access Controls. Since data is always encrypted at the discretion of the data provider, data is *not-visible* when flowing through PolyStream. Since each data provider dictates their own level of accesses granted to end users (through the different levels of computation enabling encryption), PolyStream offers the data provider greater protection. Further, since data consumers have the ability to utilize some computation-enabling encryption technique, PolyStream offers better utilization versus the state-of-the-art systems.

We show that PolyStream out-preforms the state-of-the-art system (550x performance gains) in almost all cases by utilizing faster encryption techniques and by not requiring any changes to the underlying streaming system. Similarly, we provide greater functionality

versus other streaming access control systems by having an online key management system that requires no changes to the underlying system as well through the use of Security Punctuations. In Chapter 4, we address the main limitation of PolyStream, namely reducing information that can be gained by an adversary when certain encryption techniques are used (see Section 3.3.3) by allowing an extension of the operators that can be used.

4.0 SANCTUARY

In this chapter we introduce Sanctuary [42] [1], our system for processing encrypted streaming data in an Intel Software Guard Extension enclave. As noted in Chapter 3, PolyStream may leak some information to an adversary given the computation-enabling encryption scheme used. In this chapter, we aim to augment or extend PolyStream to overcome this limitation. We implement streaming operators using secure *enclaves* that provide a private, trusted, secure processing base on an otherwise untrusted system. We show that any streaming operator can execute in such an enclave and provide algorithms that overcome the memory limitations inherent to an SGX enclave, overcoming one of the major flaws in both PolyStream and Streamforce as seen in Chapter 3. We further show trough evaluation that such operations have limited overheads when used in a streaming system (1.8-2x latency, on the order of milliseconds).

4.1 INTRODUCTION

In this chapter, we present Sanctuary, our system for using specialized hardware to enforce access control on untrusted third-party systems. To extend our work presented in Polystream (Chapter 3), Sanctuary allows a user to specify specialized operators that can be used at optimization time to both increase efficiency and system utilization by allowing for greater use of untrusted third-party systems. Sanctuary employs Intel's Software Guard Extensions (SGX) to generate secure and protected *enclaves* to execute streaming operators in a private

[1] Sanctuary was presented in the Conference on Data and Application Security and Privacy (CODASPY) in 2019, Dallas, Texas [42].

and secure context. These enclaves execute trusted and verifiable code on the untrusted third-party systems. The use of such enclaves further overcomes the limitations of Polystream in terms of expressibility and coverage of the query language.

Recall that Polystream was limited in several ways with respect to being able to execute *arbitrary* queries on an untrusted third-party system. For instance, Polystream could not execute an arbitrary join on two different streams since the underlying encryption key for each stream would likely be different. Furthermore, Polystream uses computation-enabling encryption which may leak metadata or other information about a user's private data (as detailed in Sectio 4.5). Sanctuary addresses these limitations by using a secure enclave to execute any streaming operation on any streaming data for which the data consumer has sufficient access. Specifically, we contribute the following:

- A Data Stream Processing System that utilizes SGX enclaves to support the execution of arbitrary streaming relational operations over sensitive data on untrusted third-party infrastructure.
- SGX enclaves have access to a limited memory (128 MB). Often, stateful operators will require more memory than what an enclave can provide. Sanctuary includes algorithms for stateful relational operators that are designed for the memory-limited enclave environment.
- Sanctuary can achieve a greater level of data protection when compared to state-of-the-art cryptographically-enforced access controls, and further show Sanctuary to be near ideal in terms of information leakage when compared to a baseline system.
- Finally, we carry out an in-depth evaluation of each relational streaming operation in Sanctuary and compare it to similar relational streaming operations for both unprotected (i.e., plaintext) data, and data protexted using different computation-enabling encryption techniques [3, 4]. We further include enclave-enabled operators as part of larger query networks and evaluate the overheads associated with their use.

Sanctuary is introduced with relevant related work in Section 4.2, our stateless and stateful operators in Section 4.3 and Section 4.4, and a detailed security analysis in Section 4.5 and a detailed performance analysis in Section 4.6.

4.2 SYSTEM MODEL AND ASSUMPTIONS

In Chapter 2 we detailed our threat model and system models. Here we augment those with further assumptions and details. Specifically, we augment our threat assumptions by further assuming that the SGX hardware itself remains uncompromised; i.e., it is patched against side-channel attacks such as Spectre [43] and Foreshadow [44]. As we later detail, Sanctuary is data oblivious with respect to its use of cryptographic keys, which are unlikely to be leaked via side-channel attacks. We further detail how Sanctuary is *not* data oblivious in terms of the data being processed in Section 4.5.

Sanctuary aims to prevent third-party service providers and adversaries observing the network from being able to obtain or infer the underlying plaintext data produced and transmitted by a data provider. Moreover, Sanctuary allows for data to be encrypted in any manner during transit to prevent inference of the underlying plaintext values. Finally, Sanctuary aims at limiting the leakage of ancillary data (e.g., tuple value distributions, orderings, etc.) to the service provider and third-parties observing the system.

4.2.1 Closely Related Work

SGX for Data Protection: Current work in SGX-enabled computation has focused on many different areas, from securing ZooKeeper data [45]; to managing transactions, or enterprise rights management privately [46]; to segregating linux containers [47]. SecureStream [8] is a system that explores the use of Intel's SGX as a way to execute Map-Reduce streaming applications. Further, VC3 [48] builds a Map-Reduce engine on top of SGX hardware that allows for attestation of code and data on a powerful adversary. In Sanctuary we focus on streaming relational data operators and the challenges associated with windowed operators and memory limitations.

Opaque [49] augments SQL operators so that their memory accesses are hidden and operate within an SGX enclave. Similarly, EnclaveDB [50] provides SQL operators that execute with an SGX enclave. Finally, SGX-BigMatrix [51] provides a high level language that can be used to provide secure, enclave-enabled computations. Sanctuary is complementary to these

previous works, as it is designed to enable enclave-protected, real-time relational stream processing with the goal of overcoming the limitations of state-of-the-art cryptographic stream processing systems while also providing enhanced security (cf. Section 4.5).

4.2.2 Deployment Model

Processing a query in Sanctuary involves three main steps: ensuring that the data consumer has the necessary cryptographic keys to decrypt all relevant streams; generating and deploying standard or enclave-based data stream operators; and executing the running query. We will describe each of these phases using Figure 13 as a simple example. This query involves two streams: an unencrypted stream originated by $DP1$, and an encrypted stream originated by $DP2$. In this query, a standard selection operator is first applied to the unencrypted data stream. The resulting stream is then joined to the encrypted stream using SGX-enabled operator.

Key acquisition. In order for a data consumer to access an encrypted data stream, they must have access to the cryptographic key (or keys) used to encrypt the stream. For simplicity, in Figure 13, we assume that a single key, k_p, is used to encrypt the stream originated by $DP2$. In Sanctuary, key management is handled either in an offline manner, or online using a mechanism such as Fence [1] or Polystream [3]. Once a data consumer is able to decrypt streaming data, they are in a position to leverage the ability of Sanctuary to deploy SGX-enabled query operators.

Query deployment. Sanctuary is developed on top of the Apache Storm [34] infrastructure (cf. Section 4.6 for details). Plaintext relational operators are deployed as Storm bolts in the typical manner. The deployment of SGX-enabled operators is a multi-step process, as shown in Figure 13. First, Sanctuary will create an SGX enclave capable of executing the desired streaming operator (cf. Sections 4.3 and 4.4 for details). Next, this enclave is deployed to the Storm infrastructure (arrow 1). SGX remote attestation is then used to ensure the integrity of this operator as it is instantiated within Storm (bidirectional arrow 2). This process results in the derivation of a session key k_s that can be used to communicate securely between the data consumer and the enclave. Finally, k_s is used to encrypt and transmit the

data stream key k_p to the operator enclave (arrow 3). At this point, the query network is ready to receive input tuples.

Query execution. In steady state, unencrypted tuples from $DP1$ and processed by the selection operator as in a standard DSMS. Encrypted tuples flowing from $DP2$ into the enclave-based join operator are decrypted using k_p and joined with the output of the selection operator. All result tuples are encrypted with k_s and forwarded to the data consumer.

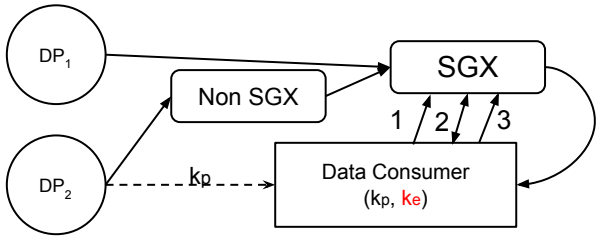


Figure 13: An example scenario of Sanctuary query deployment.

4.3 STATELESS OPERATORS

In this section we briefly describe common DSPS stateless operators supported by Sanctuary and discuss how these operators must be altered to execute within an SGX enclave.

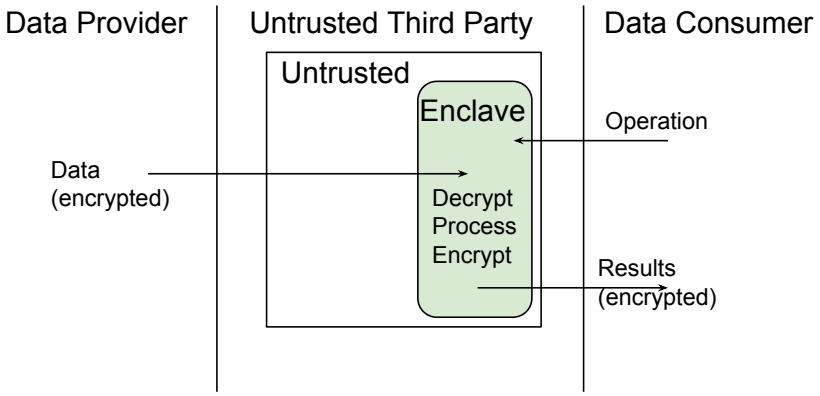


Figure 14: Stateless operator interaction with the enclave.

4.3.1 Stateless Operator Overview

Stateless operators interact with the enclave in a manner depicted in Figure 14. Operators are sent to the untrusted third-party by the data consumer. As data arrives from the data providers, it is decrypted, processed, re-encrypted, and finally sent to the data consumer. This decryption key is provided at the initialization of the enclave as part of the operation itself. There are three common stateless operators used in DSPSs: Filter, Projection, and Map. Given the straightforward nature of these operators, the only changes needed for execution within an enclave are the decryption of input tuples and the (potential) re-encryption of output tuples with a constant overhead (explored further in Section 4.6).

Filter: The filter operation simply verifies that a particular field matches a desired expression for range or equality (e.g., $x = y$, $x > y$, etc.). A filter must store *the predicate* (i.e., the constant comparator as described by the data consumer's query), the field to compare the predicate too (i.e., the "name" field, or a field identified by its placement in the tuple), and the operator code itself within the enclave, which reduces the overall enclave capacity. Processing a tuple will likely require a decryption (of at least the required field) and may require a re-encryption, depending upon whether the result must be transmitted in ciphertext or plaintext.

Projection: The projection operation reduces the size of a tuple by filtering out a specific set of fields to be passed along. The only information required to be saved in the enclave for a filter operation is the set of field identifiers (e.g., field name or placement within the tuple) to be preserved in the resulting tuple. It is likely that the tuple will not need to be decrypted, as no value is being checked. However, if fields are identified by a key-value type of system (i.e., fields may appear in any order in a given tuple), then a decryption of the entire tuple is required.

Map: Similar to the projection operation, a mapping operation reduces the size of a tuple by performing a function on several fields. For instance, a mapping operation may take fields for *revenue* and *expenses* and produce one field called $profit$. A mapping operator will have to decrypt the desired fields required to preform its function, and may need to encrypt the resulting field if the result needs to passed further down the operator network.

4.3.2 Enclave vs. Non-Enclave CPU Contention

Recall from Section 4.2 that when a program requests enclave operations, the CPU will halt all other processes and load the enclave-enabled program, as well as any data that is associated with the program itself. Similarly, whenever the enclave-enabled program completes its task within the enclave, it must write back any instance data, its code, CPU memory, and its metadata back to encrypted memory so that the CPU can preform other tasks and maintain the proper separation between enclave and non-enclave processes. This context-switching adds a processing overhead to the overall operator execution for every switch that is required. Specifically, entering an enclave will have a cost c_{enter} and exiting has a cost of c_{exit}. Each cost is dependent on the machine, workload, and other processes on the CPU and can vary with every entry and exit. The best way to mitigate this context-switching cost is to reduce the number of entries into and exits from the enclave.

Some context switching is unavoidable, but a streaming operation can mitigate the negative impact of context switching by simply reducing the number of calls into the enclave by batching data tuples. Rather than calling the enclave and paying c_{enter} and c_{exit} for *every* data tuple received, data tuples can be batched so that a single c_{enter} and c_{exit} is incurred and amortized over all n tuples in a batch. In a data stream, batching may add considerable latency to a query since results are delayed until a batch has been filled. In Section 4.6 we explore the benefits of batching.

4.4 STATEFUL OPERATIONS

When an operation is required to consider multiple tuples in order to execute a query, it is considered to be stateful. In this section, we overview the common stateful operators used by Sanctuary, and detail the challenges associated with implementing this class of operations within an enclave. We further propose three algorithms for executing stateful operations inside an SGX enclave.

4.4.1 Operators

There are two main types of stateful operations in a DSPS: joins an aggregations. This section overviews those operations and classifies the different types of each.

Joins: In a streaming system, a join operation compares two or more different streams in a given window and returns a set of data tuples comprised of data from each stream based upon some join condition. The specified period for comparing each stream is called the *window* which can be expressed either in time or in number of data tuples. A join must keep state on all data tuples that are within the current window for all streams in the join. For example, if a data consumer requests "all tuples where `streamA.id = streamB.id` for the last 10 minutes", all tuples in `streamA` and `streamB` that were timestamped within the last 10 minutes must be stored to compare with new tuples within the window.

Streaming join algorithms can be designed to either 1.) consider all possible pairs of join tuples, or 2.) consider a smaller set of tuples in each stream by using some auxiliary data structure. The *nested loop join* (NLJ) is a join algorithm that must consider all of the tuples in each stream's state by attempting to join every data tuple in one stream with every data tuple in the other. In practice, such a join algorithm is undesirable because of the overheads incurred. However, looping over all tuples avoids the leakage of positional information regarding the specific tuples being joined. *Hash joins* use an auxiliary hashing structure to reduce the number of tuples that need to be compared, reducing the overhead for the join algorithm, but potentially leak information about underlying data.

Distributive and Algebraic Aggregation: An aggregation is *distributive* if the input can be distributed to many partitions where a partial aggregation is processed, followed by a final aggregation of the partials (e.g., a sum can be broken down into smaller sums, with a final sum of the partials generating the overall result). *Algebraic* aggregations are those that can be represented as an algebraic function of two or more distributive aggregations (e.g., average can be calculated by a summation and a count). Distributive and algebraic aggregates have a constant memory overhead.

Holistic Aggregation: An aggregation is holistic if there is no constant bound on the memory required for partial or final aggregation. For instance, the *median* operation is

holistic because there is no way to determine the size of the resulting set of median values. A window in a holistic aggregate is treated in the same manner as in the distributive or algebraic aggregations.

4.4.2 Issues with Stateful operators and Enclaves

Stateful operators cannot be directly implemented in an enclave environment without alteration. Specifically, there are two main concerns when implementing a stateful operation on an enclave: *memory limitations* of the enclave and *update costs* associated with auxiliary data structures that may be required.

Memory Limitations: Recall from Section 4.2 that an enclave is limited to just 128MB of memory, which is further reduced by the need to store operator code o and meta-data m within the enclave. Stateful operations must make use of this limited memory to store each operation's windows of tuples. Windows can be of a nondeterministic size w (e.g., the last 10 minutes saw 10k tuples, but the next 10 minutes may see 13k tuples) and may not fit into enclave memory. Any stateful algorithm will therefore have to consider swapping between (encrypted) non-enclave memory and enclave memory.

In addition to the window itself, some operations (e.g., hash joins) maintain auxiliary structures. Such structures will vary in size s across operators and will likely need to be kept (at least in part) in enclave memory, further reducing the available memory. Therefore, the total capacity for storing data tuples in an enclave with $enclaveSize = 128MB$ and n operators each with w_s windows (that can vary in size, 0 for stateless operations) is fixed at:

$$capacity = enclaveSize - (m + (\Sigma_0^n(o + s + \Sigma_0^{w_s} w))). \tag{4.1}$$

Update Cost Overhead: In a traditional DSPS, removing tuples from or adding tuples to a window in relatively straightforward: one simply checks the timestamp for each tuple as new tuples are added, and removes or dereferences those that have expired. The use of an SGX enclave presents a new challenge with regards to updating the state of a stateful operator. Specifically, when the state of an operator is encrypted in non-enclave memory, the timestamp and most recent tuple information may not be available without either trusting

the third-party service to remove expired tuples or leaking some temporal information about the tuples. We can calculate the cost c_{up} of an update by simply multiplying the number of tuples being updated n_{up} by the time it takes to execute that update l_{up}. This cost is added to the overall latency for stateful operators.

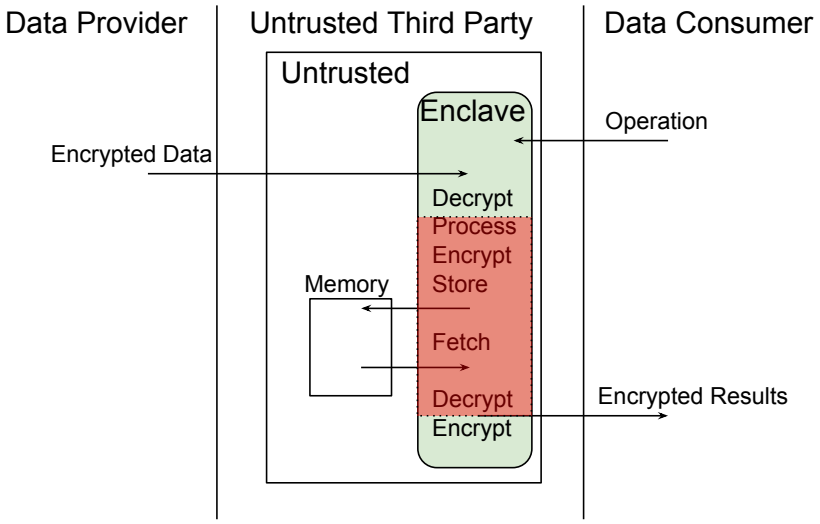


Figure 15: Stateful operator interaction with the enclave.

4.4.3 Enclave-Enabled Stateful Operators

We now introduce three algorithms for stateful streaming operations that can execute within an SGX enclave: Nested Loop Join (NLJ), Hash Join (HJ), and generic aggregation (AGG). All algorithms follow the structure depicted in Figure 15 (where red signifies the actual operation on decrypted data, and green is common across all operators). Data consumers use SGX's remote attestation capabilities to ensure that their stateful operator enclaves have been appropriately provisioned to the remote infrastructure. As (encrypted) data is received by the enclave from a data provider, it is decrypted and processed. Once processed, it is either discarded (aggregation) or stored in untrusted, encrypted memory. Tuples can be brought back into enclave memory as needed (e.g., to be joined with new tuples) or the partial aggregate to which it contributed can be brought into memory (e.g., the sum for the slides affected by the tuple is brought in to sum new tuples to). This process of fetching and storing continues until all new data is processed. Note that these alrorithms are designed to

Algorithm 3 *enclaveNestedLoopJoin*(*Array*{*tuple*}*batch*)

```
1:  Array[tuple] batchSide                    ▷ Stored tuples from the same stream as the tuples in batch
2:  Array[tuple] otherBuffer                  ▷ Stored tuples from the other stream
3:  int maxJoinSetSize                        ▷ Available in-enclave memory.
4:  Object metadata                           ▷ Storage describing the operator.
5:  for t ∈ batch do
6:    batchSide.add(t)
7:    enclave.decrypt(t)
8:  for i = 0; i ≤ ⌈otherBuffer/maxJoinSetSize⌉; i++ do
9:    if i! = ⌈otherBuffer[i]/maxJoinSetSize⌉ then
10:     segment_tuple = memGet(i ∗ maxJoinSetSize, (i ∗ maxJoinSetSize) + maxJoinSetSize)
11:   else
12:     segment = memGet(i ∗ maxJoinSetSize, otherBuffer.size)
13:   for l ∈ segment do
14:     if l.timeStamp < currentTime − window then
15:       evict(otherBuffer.getiAllMatchingValues(l.value))
16:     enclave.decrypt(l)
17:     for s ∈ batch do
18:       if l[metadata.joinField] ⋈ s[metadata.joinField] then
19:         emit(enclave.encrypt(join(l, s)))
```

work in *any* memory limited environment, but are more well suited for the SGX use-case as they aim at avoiding costs associated with CPU context switching.

4.4.3.1 Join Algorithms

Nested Loop Join: We first overview our Nested Loop Join (NLJ) in Algorithm 3. The enclave sets up two spaces in non-enclave memory to represent the windows for each stream. When new data for a stream enters the enclave, the window for the other stream is loaded into enclave memory. If the entire window does not fit, it is segmented (lines 8–12). The *memGet* function simply takes two indices, maps them to registers in memory, and fetches the values. Each segment is then compared and joined to the new tuples being processed (lines 13–19), any joined results are emitted to the next operation or data consumer. In addition to being compared to new tuples, a tuple being brought in from non-enclave memory is also evicted if its timestamp no longer fits within the window (line 14). Finally, all new tuples are added to the end of their window without bringing that state into memory. To implement such an operator, Sanctuary need only the field names from each stream, as well as the slide and window. Sanctuary simply submits the operator with this metadata to a remote system as a function and then verifies it via remote attestation. Whenever the query is ready to be executed, Sanctuary simply executes the function.

A Nested Loop Join is not generally desirable, given that it must compare every tuple

Algorithm 4 *hashJoin*(*Array*{*tuple*}*batch*)

```
1:  Map[String, Array[tuple]] batchSideHash
2:  Map[String, Array[tuple]] otherHash
3:  Array[tuple] batchSideBuffer          ▷ Stored tuples from batch's corresponding stream.
4:  Array[tuple] otherBuffer              ▷ Stored tuples from the other stream.
5:  int maxJoinSetSize                    ▷ Available in-enclave memory.
6:  Object metadata                       ▷ Storage describing the operator.
7:  for t ∈ batch_tuple do
8:    enclave.decrypt(t)
9:    if otherHash.get(t.value)! = null then
10:     int jSize = otherBuffer(otherHash.get(t.value)).size
11:     for i = 0; i ≤ ⌈jSize/maxJoinSetSize⌉; i++ do
12:       if i! = ⌈jSize/maxJoinSetSize⌉ then
13:         matchSet = memGet(otherBuffer, i ∗ maxJoinSetSize, (i ∗ maxJoinSetSize) + maxJoinSetSize)
14:       else
15:         matchSet = memGet(otherBuffer, i ∗ maxJoinSetSize, otherBuffer.size)
16:       for r ∈ matchSet do
17:         enclave.decrypt(r)
18:         if r.timeStamp < currentTime − window then
19:           evict(otherHash.get(r.value).get(r))
20:         else if t[metadata.joinField] ⋈ r[metadata.joinField] then
21:           emit(enclave.encrypt(join(t, r)))
22:   if t.value ∈ batchSideHash then
23:     batchSideBuffer[batchSideHash.get(t.value)].add(t)
24:   else
25:     batchSideHash.put(t)
26:     batchSideBuffer.extendByOne()
27:     batchSideHash.get(t) = batchSideBuffer.size − 1
28:     batchSideBuffer[batchSideBuffer.size − 1] = newArray()
29:     batchSideBuffer[batchSideBuffer.size − 1].add(t)
```

in one window with every tuple in the other (or at the very least compare new tuples from each window with the other one). This does, however, offer a nice confidentiality guarantee in an enclave setting, as it does not reveal the relationship between any *specific* tuples in non-enclave memory with new tuples being processed (discussed further in the next section), since every tuple is compared against all buffered tuples.

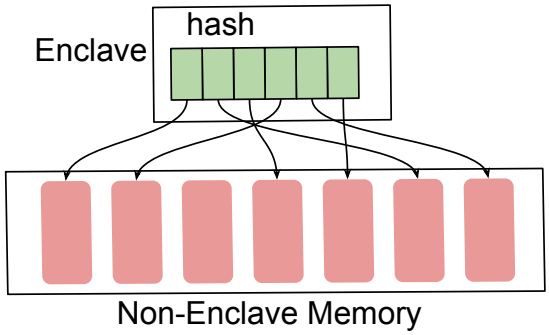


Figure 16: Use hashing to split a window in non-enclave memory.

Hash Join: Algorithm 4 details our Hash Join (HJ) algorithm. Within the enclave

memory, a hash structure it maintained for each stream in the join. For every unique key in the join predicate (e.g., each name in a "name" field that has been processed), an entry is made into the hash structure where the key is the predicate, and the value is a space in non-enclave memory where the actual tuples are held, as pictured in Figure 16.

The process of operating on a new tuple is simple. It is first decrypted (Line 8) and then the tuple's join predicate value is hashed to see if there exists at least one match in the other stream (Line 9). If so, the entire set of tuples in the hash entry, or the *matching set* (i.e., the segment of all tuples with the same hash key) is brought into the enclave by enclave-memory sized segments (Lines 10–15). Once all matching tuples are joined to the new tuple, the new tuple is added to its originating stream's hash with its corresponding entry. If the tuple's predicate did not exist in the stream's hash, a new entry is created and the tuple is added to the endpoint for the pointer stored in the hash (Lines 22–29). Note that if the internal hash structures(s) run out of memory, a secondary hash is created in non-enclave memory that will absorb some hash values (i.e., all non-matching predicates are sent to a second hash to be checked). This adds one extra memory operation and one extra hash operation to fetch the desired hash-value. This is not included in Algorithm 4 for simplicity. Similar to the requirements for Nested Loop Join, Sanctuary only requires the fields needed to preform the join, as well as the window and slide. The user may also specify some memory specifications for how they prefer to handle allocation of enclave memory.

Updates in the HJ algorithm are a bit more complicated. A tuple is only removed from a set referred to by a hash entry if that entry is brought into enclave memory. The benefit of this approach is rather straightforward in that updates are handled in an ad-hoc, on-the-fly manner without requiring any extra loads from memory. The obvious drawback is that some data may linger around for a while if its predicate is not matched by the other stream. We leave garbage collecting expired tuples to future work.

4.4.3.2 Generic Aggregation

Our generic aggregation algorithm (AGG) can handle distributive, algebraic, and holistic aggregations. There are two main memory structures for an aggregation: storage for internal partial aggregations, and the final aggregation step. For distributive and algebraic operations, the state needed for storing the partial results

during a window is deterministic and can be provisioned accordingly. To accommodate for holistic operations, we assume that the size of the returned result is non-deterministic for all aggregations. In a memory limited environment, this means that results for each window of the aggregation may need to be stored in non-enclave memory. Further, in a holistic aggregation, these results may vary in size depending on the data and the window.

To accommodate for all three types of aggregation operations, we adopt an approach similar to the hash join approach. Each slide (or window in the case where there is no slide) gets an entry in an in enclave memory array. This entry (potentially) points to a hash table kept in non-enclave memory. For distributive and algebraic aggregation, this hash table may contain only one entry and may fit in enclave memory. For holistic aggregations, this hash table may contain many entries that need updating (e.g., the total sales for each company in a given stream for a given window).

Algorithm 5 details our aggregation algorithm. Each slide is given an index in an array that is stored in enclave memory. Every time a new tuple is received (or a batch of tuples), the algorithm loops through this array. If an entry has expired (based on checking the time inside a designated hash entry (Line 7) it is brought into memory (Lines 8–18) where each entry is emitted (Line 15) and cleared (Line 16) so that the hash may be reused.

For non-expired slides, they are similarly brought into memory, but instead of being emitted, they are aggregated with each tuple in the batch (Line 24). If the entry already exists in the hash for the batched value, it is aggregated to the matching value (Lines 25- 26). The function *genAgMemHelp* takes the array that the hash refers to, the current index, and the maximum size of the buffer, and fills that buffer with values from the encrypted memory by converting the index into a starting and ending register. Once all of the hash has been brought into enclave memory, if any new tuples remain (e.g., a new company has entered the stream that was not yet encountered during this window), they are simply added to the hash as the first entry for that value (Line 28- 30).

Again, to make use of this operator, Sanctuary simply needs the field to aggregate, the type of aggregation, and the window and slide information. The user may also specify the allocation of memory here as well, but Sanctuary handles the bulk of the load by allowing a user to just specify the most basic of information and doing the submission, attestation,

and execution for them.

State updates in AGG are simple in that they only require that a slide expire for state to be reset. An adversary can only gain information on how many entries are in a hash table and how many slides are in a window based on what is stored in non-enclave memory. This is no different than in distributive or algebraic operations directly on encrypted data, as the length of the slide can be determined by the rate at which results are produced, and the size of the result set is equivalent to the size of the hash table in AGG.

Algorithm 5 *aggregate*(*Array*{*tuple*}*newBatch*, *int window*, *intslide*, *int maxBufferSize*)

```
1:  Takes: aggregate(Array[tuple]newBatch, int window, int maxBufferSize)
2:  Array[Array[tuple]] slideArray                          ▷ Stores current slides in the window
3:  Map[String, Array[tuple]] hash
4:  int maxBufferSize                                       ▷ Available in-enclave memory.
5:  for hash ∈ slideArray do
6:                          ▷ If the hashed aggregates are now greater than the window length, emit them as results.
7:    if earliestTime(newBatch) ≤ (hash.get(startTime)) + slide then
8:      for i = 0; i < ⌈hash.size/maxBufferSize⌉; i++ do
9:        Map[String, Array[tuple]] currentHash
10:       if i < ⌈hash.size/maxBufferSize⌉ then
11:         currentHash = genAgMemHelp(slideArray.indexOf(hash), i, maxBufferSize)
12:       else
13:         currentHash = genAgMemHelp(slideArray.indexOf(hash), i, hash.size)
14:       for j = 0; j < currentHash.size; j++ do
15:         emit(currentHash.get(j))
16:         currentHash.get(j).clear
17:                                  ▷ Otherwise aggregate the new batch into each slide.
18:   else
19:     for i = 0; i < ⌈hash.size/maxBufferSize⌉; i++ do
20:       if i < ⌈hash.size/maxBufferSize⌉ then
21:         currentHash = genAgMemHelp(slideArray.indexOf(hash), i, maxBufferSize)
22:       else
23:         currentHash = genAgMemHelp(slideArray.indexOf(hash), i, hash.size)
24:       for t ∈ newBatch do
25:         if t.group ∈ keys(currentHash) then
26:           aggregate(t.value, currentHash.get(t.group))
27:           newBatch.remove(t)
28:       if i = ⌈hash.size/maxBufferSize⌉ && newBatch.size > 0 then
29:         for t ∈ newBatch do
30:           currentHash.put(t.group) = t.value
31:       memOut(hashArray.indexOf(hash), i, currentHash)
```

4.5 SECURITY ANALYSIS

We now detail the information that an adversary can learn by observing the execution of queries within Sanctuary. To contextualize this analysis, we compare directly to two alternative DSMS approaches.

4.5.1 Comparison Framework

Below are the three system models (including Sanctuary) within which we will compare information leakage:

- **Sanctuary**: In this system model (cf. Chapter 2) we assume that our adversary is the third-party computational infrastructure hosting a query comprised of the SGX-enabled streaming operators described with Sanctuary. As such, the adversary can observe all (encrypted) traffic flowing between operators, as well the encrypted traffic flowing between the enclave and non-enclave portions of an individual operator. To upper-bound information leakage, we assume one operator per enclave.
- **Cryptographic**: Similar to PolyStream, in this system model, we assume that our adversary is a third-party computational platform hosting a query comprised of cryptographic streaming operators. I.e., data streams are encrypted using computation-enabling encryption as in [3, 4, 5]. As such, the adversary can observe all (encrypted) traffic flowing in and between operators.
- **Trusted Infrastructure**: As a baseline for comparison, we consider a trusted third-party computational platform capable of processing standard streaming operators over plaintext tuples (e.g., [1, 22]). This is effectively the *optimal* approach in terms of minimizing leakages with our current threat model. We assume that all streaming tuples are encrypted (e.g., using TLS) while in the network. Our adversary is *not* the computational infrastructure, but rather an entity capable of monitoring all communications between nodes in the system. To upper-bound information leakage, we assume one operator per node.

The Sanctuary and cryptographic models are meant to provide a level playing field for comparing the approach presented with Sanctuary with the current state-of-the-art by considering streaming computations that execute on an untrusted infrastructure. The latter Trusted Infrastructure model serves as a basis of comparison for considering what information can be learned by an outside observer who is watching data being processed on a trusted platform. We now examine the types of leakages exhibited by each type of operator considered with Sanctuary, within each of the above system models.

Table 5: Level of leakage for the various approaches (**S** = Selectivity, **TM** = tuple matching, **VO** = tuple ordering, **VD** = value distribution, **SM** = segment matching, **W** = Window)

Operator		Sanctuary	Cryptographic Data	Trusted Infrastructure
Filter	Equality	S	S, TM, VD	S
	Range	S	S, TM, VO, VD	S
Project		∅	∅	∅
Join		S, SM, W	*Not Supported*	S
Aggregation		W	W	W

.

4.5.2 Properties

To understand the leakage of information in various DDSMS deployment models, we first identify types of leakage. In this section, we use the notation $\mathsf{E}_{\mathsf{DET}}(k,\, v_i)$ (resp. $\mathsf{E}_{\mathsf{OPE}}(k,\, v_i)$ or $\mathsf{E}_{\mathsf{RND}}(k,\, v_i)$) to denote the deterministic (resp. order-preserving or CCA-secure) encryption of a tuple v_i using the key k. We use the notation $\mathsf{E}(k,\, v_i)$ in situations where the specific type of encryption used is immaterial.

- **Tuple Matching (TM)**: Given a set of input tuples $S^{in} = \{t_1^{in}, \ldots, t_i^{in}\}$ and a set of output tuples $S^{out} = \{t_1^{out}, \ldots, t_j^{out}\}$, compute the matching $M = \{\forall t^{out} \in S^{out} : (t^{in}, t^{out}) \,|t^{in} = \mathsf{E}(k,\, v^{in}) \wedge t^{out} = \mathsf{E}(k,\, v^{out}) \wedge v^{in} = v^{out}\}$.
- **Value Ordering (VO)**: Given a set of tuples $S = \{t_1 = \mathsf{E}(k,\, v_1),\, t_2 = \mathsf{E}(k,\, v_2)\, \ldots,\, t_i = \mathsf{E}(k,\, v_i)\}$, compute an ordering $t'_1, t'_2, \ldots, t'_i$ such that $v'_1 \leq v'_2 \leq \ldots \leq v'_i$.
- **Value Distribution (VD)**: Given a set of tuples $S = \{t_1 = \mathsf{E}(k,\, v_1),\, t_2 = \mathsf{E}(k,\, v_2)\, \ldots,\, t_i = \mathsf{E}(k,\, v_i)\}$, compute the frequency distribution $\hat{v_1} = \mathsf{count}\,(v_1, S)\,, \hat{v_2} = \mathsf{count}\,(v_2, S)\,, \ldots, \hat{v_i} = \mathsf{count}\,(v_i, S)$. Note $\hat{v_i}$ does not necessarily reveal the value v_i.
- **Segment Match (SM)**: Given an input tuple t and a segmented window $w = \{ms_1, \ldots, ms_k\}$, identify the matching segments ms_i within which t can complete a join.

In addition, we will explore whether the selectivity (S) of a given predicate or the window size (W) of an operation can be inferred.

4.5.3 Leakage Comparison

We now examine the information that can be inferred by an adversary when observing the execution of each of the above systems. We consider the streaming relational operators described with Sanctuary, and summarize our results in Table 5.

Select/Filter. Each selection operator takes as input a stream $S^{in} = t_1^{in}, \ldots, t_i^{in}$, applies a filter f, and produces as output a stream $S^{out} = t_1^{out}, \ldots, t_j^{out}$. In each system considered, the adversary can compute the selectivity of f by comparing the cardinality of S^{in} and S^{out} over some window. In both the Sanctuary and Trusted Infrastructure models, all tuples are encrypted using CCA-secure cryptography (i.e., $t_i = \mathsf{E}_{\mathsf{RND}}(k, v_i)$ (random encryption)). As such, the values v_i comprising the input and output streams are protected.

In the Cryptographic model, equality filtering is enabled by the use of deterministic encryption (i.e., $t_i = \mathsf{E}_{\mathsf{DET}}(k,\ v_i)$). As a result, $\mathsf{E}_{\mathsf{DET}}(k,\ v_i) = \mathsf{E}_{\mathsf{DET}}(k,\ v_j)$ if and only if $v_i = v_j$. This enables the adversary to infer a distribution of values over the encrypted tuples in S^{in} irrespective of the filter f. Further, the adversary can infer exactly which tuples t_i^{in} match the predicate f, as these tuples appear unmodified in S^{out}. For range filtering, order-preserving encryption must be employed (i.e., $t_i = \mathsf{E}_{\mathsf{OPE}}(k,\ v_i)$) so that $\mathsf{E}_{\mathsf{OPE}}(k,\ v_i) \leq \mathsf{E}_{\mathsf{OPE}}(k,\ v_j)$ if and only $v_i \leq v_j$. This enables the adversary to determine an ordering over tuples appearing in S^{in} and S^{out}.

Projection. For all three frameworks, a projection simply removes fields from *every* tuple and therefore always has a 100% selectivity. Since input and output values are encrypted, tuple values, distributions, and orderings remain hidden in all cases.

Join. A join operator takes as input two streams $S_1^{in} = t_{11}^{in}, \ldots, t_{i1}^{in}$ and $S_1^{in} = t_{12}^{in}, \ldots, t_{j2}^{in}$, applies a join predicate p, and produces an output stream $S^{out} = t_1^{out}, \ldots, t_k^{out}$ that joins S_1^{in} and S_2^{in} using p over some (time- or tuple-based) window W. Given that S_1^{in} and S_2^{in} are typically encrypted with different keys, no existing cryptographic DDSMS supports join operations over streaming data.

In both the Sanctuary and Trusted Infrastructure models, input tuples are encrypted using randomized encryption (i.e., $t_{ij}^{in} = \mathsf{E}_{\mathsf{RND}}(k_j,\ v_i)$), thereby preventing the inference of

tuple values, distributions, and orderings. In both cases, the adversary can easily compute the selectivity of p by comparing the cardinality of S_1^{in}, S_2^{in}, and S^{out} over some time window. In Sanctuary, the adversary has the ability to monitor the enclave's accesses to non-enclave memory. Recall that given enclave memory limitations, a join window $w_i^1 \in S_1^{in}$ is managed as a set of matching segments $w_i^1 = \{ms_{i1}^1, \ldots, ms_{ik}^1\}$ each of which fits into enclave memory. By monitoring the eviction rate from these segments, the adversary can infer the window size used by the join. Further, as new tuples arrive, auxiliary data structures are used to retrieve only the segments that will join with the incoming tuples, thereby leaking the segments that incoming tuples match to.

Aggregation. During aggregation, a sliding window of tuples is combined to produce a single output tuple. In the Cryptographic and Trusted Infrastructure models, the adversary can infer the window size, W, used for the aggregation operation by counting the number of tuples $\mathsf{E}_{\mathsf{HOM}}(k, v_i)$ consumed by the operator prior to emitting each output tuple. Likewise, in Sanctuary, the adversary can infer the W by watching the transition of encrypted tuples between enclave and non-enclave memory. In all cases, the use of randomized encryption prevents the inference of tuple values, ordering, and distribution.

Summary: Sanctuary leaks the same minimal information as the Trusted Infrastructure system for all operations excepting the join, which can be mitigated if the join state fits in enclave memory. Further, Sanctuary not only leaks the same or less information as cryptographic DDSMS systems, but also enables arbitrary join operations, the lack of which is a severe limitation of existing cryptographic systems.

4.6 EVALUATION

In this section, we explore the efficiency of Sanctuary operations in a real streaming system. Specifically, we benchmark each operation in comparison to Trusted Infrastructure approaches under different experimental conditions. We then use SGX-enabled operations within the context of deployed streaming queries to evaluate the impact of enclave-based operations on query performance.

4.6.1 Configuration

Our evaluation framework builds upon the Apache Storm infrastructure [34] to manage the network topology and deliver tuples. Our work can be trivially deployed over any DSPS. Storm uses two main computational components called *spouts* (provide data) and *bolts* (execute on it). For this work, we use bolts to emulate a node. This implies that a single bolt has access to one SGX-enabled CPU. We use Storm to emulate the temporal aspect of real-time data stream processing since it can be configured to guarantee in-order tuple delivery, and spouts can emit tuples at a given timestamp (to better control input rate). All experiments were executed on a machine using the Windows 10 operating system with a dual core Intel i5-6200 CPU (2.30 GHz) and 4 GB of ram.

Datasets: To better explore the tradeoffs in size, speed, and selectivity of data, we will use two different types of data-sources. The first type is synthetically-generated data that is purposefully created to test the boundaries for each SGX-enabled operation as well as its alternatives. Each evaluation that alters this data will describe, in detail, how it is generated and used. The second set of data is the 2015 DEBS Grand Challenge dataset [52] consisting of tuples that describe an instance of a taxi ride (i.e., the start time, taxi driver ID, cab ID, end time, fare, distance, etc.).

Comparison Approaches: For each operator or algorithm we test, we will compare it to the same operation executed over *plaintext* and also in a DSPS using *computation-enabled encryption*. We will use three different encryption techniques for comparison: 1.) Deterministic Encryption (DET, which uses AES in CBC mode with padding, see [3] and [5]) for equality operations, 2.) Order Preserving Encryption (OPE, which uses the Boldyreva et. al [16] technique) for range operations, and 3.) Homomorphic Encryption (HOM, using the Paillier technique [18]) for aggregate operations.

4.6.2 Micro Benchmark: Stateless Operations

For stateless operators, there is no state to keep track of, tuples are simply fed directly to the enclave and processed in *batches*, with batched results being returned. All results are based on the average processing time for 10,000 tuples. For batch execution, the results are

based on the average of 10 runs for each batch size.

4.6.2.1 Filter Configuration: Given that the time to decrypt a field depends on the size of the field, we evaluate the processing time for different sized fields. We further evaluate enclave batching by including five different batch sizes (1, 1K, 10K, 100K, 1M) and their overheads. We compare Sanctuary to a plaintext system as well as one that uses order-preserving encryption.

Results: Sanctuary filters will incur roughly 2x-4x overall execution time versus plaintext approaches (but leak less information), and 1.5x-3x overall execution time versus CPE techniques. Note that most of the overhead is due to context switching between trusted and untrusted code in the CPU. To this end, using larger batch sizes reduces this overhead, and most execution times are under 10ms overall. Moreover, Sanctuary operations are similarly impacted by the size of the underlying ciphertext.

4.6.2.2 In-Memory Aggregation Here, we evaluate Sanctuary summation operations against a HOM computation-enabling cryptographic operation. Since a HOM summation is done through multiplication, processing on encrypted data requires greater overhead.

Configuration: We use a summation operation to evaluate an in-memory aggregation within the enclave. For simplicity, we assume no window semantics (meaning the final aggregation is simply inclusive of all tuples) to better understand the underlying operation. We will use a windowed query in Section 4.6.4 to evaluate performance on an actual streaming query. Again, we include five batching sizes and compare to a plaintext non-enclave summation, and we use a HOM computation-enabled encryption scheme.

Results (Figure 17): Batching for an in-memory aggregation algorithm has a similar impact as the filter operation. However, when using the Paillier homomorphic encryption scheme, computing a sum requires multiplication which comes at a greater cost. This is evident from the HOM line in Figure 17 being the most costly line. Even in the case where there is a batch size of one, the enclave-enabled in-memory aggregation operation can outperform its encrypted counterpart. Since both the HOM-encrypted processing and the Sanctuary enabled operation provide an adversary with the same level of information, Sanctuary becomes a desirable choice when considered in conjunction with the performance advantages.

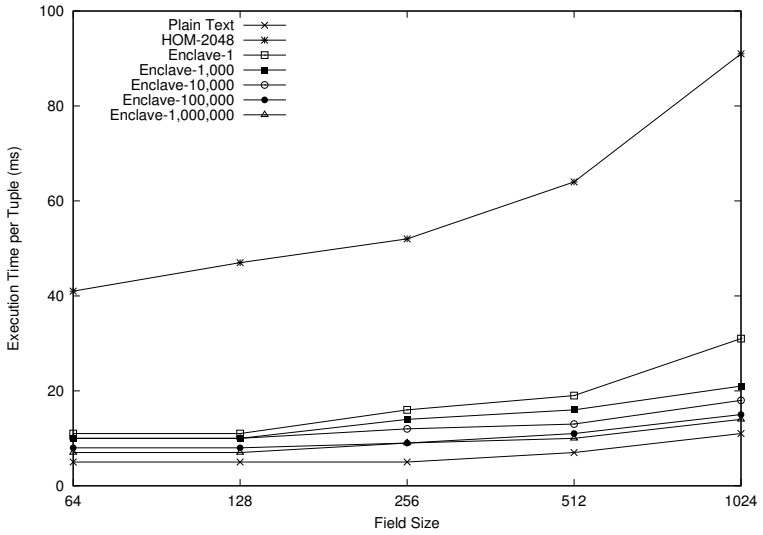


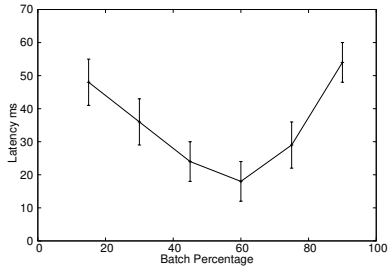
Figure 17: Execution time for in memory aggregation for non-enclave and enclave-enabled operations.

4.6.3 Micro Benchmark: Stateful Operations

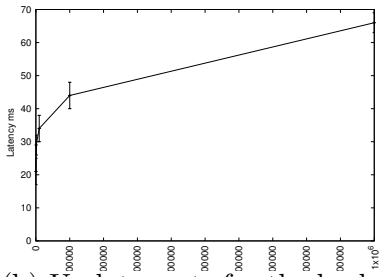
This section explores the overheads required for the security advantages of the memory-limited algorithms presented in Section 4.4. Recall from Section 4.4 that there are five factors that can influence the overhead of a memory-limited operation; 1.) Batch Size, 2.) Enclave Memory Structure Size 3.) Operator State Size, 4.) Window Size, and 5.) Update Cost. We reduce all arbitrary units of size measurement (e.g., tuple, enclave memory) to megabytes for simplicity when making comparisons. We evaluate each of the four algorithms presented in Section 4.4 based on the tradeoffs below:

- ***Batch Size vs. Enclave Memory Size***: Given the limited memory of an enclave, there exists an inherent tradeoff between the batch size of incoming tuples and the enclave memory available for bringing operator state into the enclave for processing.
- ***Operator State Size vs. Enclave Memory Size***: Stateful operations require a comparison or computation over some amount of internal state. Given the limitation on the internal enclave memory available for operator state, there is a tradeoff that can affect the execution time (or the latency) of the operation.

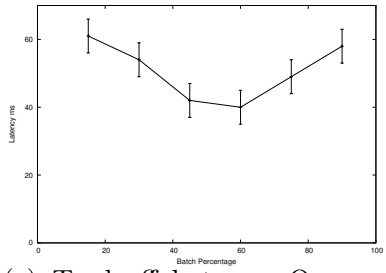
- ***Window Size vs. Update Cost***: When a tuple expires or is introduced to the state of the operation, there is an associated update cost. For larger windows (either by definition or through high-bandwidth streams), these updates can come with a greater overhead, affecting the overall performance of an operation.



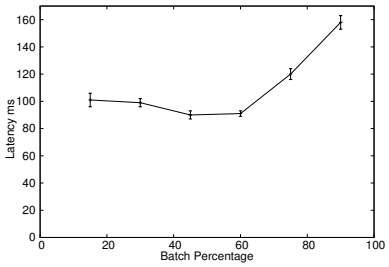
(a) Hash Join batch size vs enclave memory size.



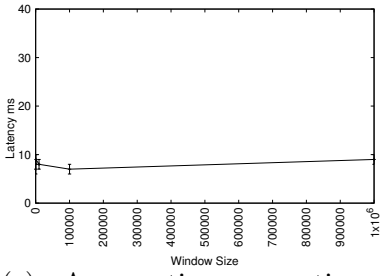
(b) Update costs for the hash join algorithm to update each hash for each window.



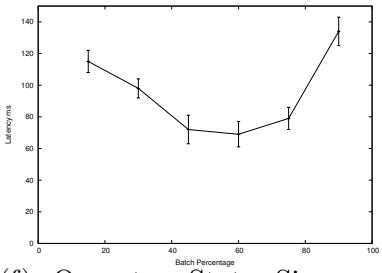
(c) Tradeoff between Operator State Size and Enclave Memory Size for the Hash Join.



(d) Aggregation batch size vs enclave memory size sows the impact of batching on aggregation operations.



(e) Aggregation operations require very little for updates by nature, as shown by this window size versus update cost evaluation.



(f) Operator State Size vs Enclave Memory Size for the General Aggregation.

Figure 18: Results for the the common evaluation for the three stateful operations from Section 4.4

Each experiment depicts the average of seven runs with the lowest and highest removed to ensure that background tasks are weighted similarly as the CPU is shared by other processes.

4.6.3.1 Symmetric Hash Join Configuration: Each tuple contains two fields: a *comparator* (8 Bytes) and a *payload* (92 Bytes). Joined tuples are combined into 184 Byte

payloads. Comparator fields are uniformly selected from a range of integers from 1-256. After metadata, the internal enclave memory is roughly 122 MB, of which all is available for buffers and state comparisons.

Results (Figure 18a) Batch Size vs Enclave Memory Size: For this experiment, we evaluate the batch size versus the available enclave memory. Specifically, we hold the window size constant at 100 MB with an input rate of 10 MB/s. We reserve 30% of the enclave memory to be given to the internal hash for tracking non-enclave memory. We see the tradeoff between batching and freed enclave memory maximize at roughly 60%.

Results (Figure 18b) Window Size vs. Update Cost: For this experiment, we fix the batching to enclave memory ratio at 50% each. We adjust the window size from 10MB to 100 MB and hold an input of 10 MB/s. We again allocate 30% of enclave memory to the internal hash. Recall that an update to the external structure to expire a tuple will only occur when that value is joined (Lines 18- 19 in Algorithm 4). Note here that each data tuple must be hashed, brought into memory, compared, then evicted, which all adds to a higher latency compared to a plaintext system.

Results (Figure 18c) Operator State Size vs. Enclave Memory Size: Here, we fix the batch size at 30% of enclave memory. We then trade off internal state size with free memory size. We hold the input rate at 10 MB/s as usual with a window size of 100 MB. This tradeoff exhibits similar behavior to all of the others. Giving more memory to the hash for each stream yields the best results with a 60% ratio. It is important to note, however, that for a join that has a very low selectivity (i.e., one in which the two streams rarely join), the enclave memory will not be filled since there is insufficient data with which it can be filled. This means an enclave will only employ external memory when the join is highly selective.

4.6.3.2 General Windowed Aggregation

We evaluate the general windowed aggregate operation using the four costs from the previous section and the update cost. We also evaluate the impact on an operation's overhead based on the size and number of slides.

Configuration: We use the same data as in Section 4.6.3.1, but focus on a portion of the payload for aggregation operations (16 B integers) that is symmetrically encrypted. The

16 B integers are uniformly generated from 1-30,000 and used in a summation aggregation operation across 250 different groups. We use a window that can be divided into 10 slides for all experiments.

Results (Figure 18d) Batch Size vs Enclave Memory Size: This experiment is set up exactly like the SHJ Batch Size vs Enclave Memory Size experiment, with 30% allocated to a possible internal hash structured. Note that we no longer see a profound benefit from greater batching (gaining only 10ms), and we see a far greater increase in latency after about 40% to 60% batch allocation due to the consistent size of the hash structures storing each slide. No matter what we batch, every tuple must be aggregated to the same segmentation size of non-enclave memory, meaning that the relative effects of batching are reduced, since the cost of bringing non-enclave memory into the enclave is normalized.

Results (Figure 18e) Window Size vs. Update Cost: This experiment is similarly set up the same as SHJ Window Size vs. Update Cost, but with slides dividing the window size by 1 t Larger windows do not cause significant increases in update cost as one might expect with larger slides due to the consistent slide size for each hash brought into memory. More specifically, this consistency can be attributed to the uniform group size of 250. If we remove this uniformity and increase the number of groups (not included as a figure), we see that the greater the number of groups, the larger the update cost, especially with a greatly segmented hash table.

Results (Figure 18f) Operator State Size vs. Enclave Memory Size: This experiment is configured exactly like the SHJ Operator State Size vs. Enclave Memory Size experiment with the internal operator structure representing a cache of some hash tables. Similar to what we saw in the Batch Size vs Enclave Memory vs. Operator State Size evaluation for this aggregation, giving more memory to caching is advantageous up to about 60%.

4.6.4 Macro Benchmark

Here, we chose a query to execute on streaming data with varying conditions. We break the query evaluation into two different environments. The first environment illustrates the

effectiveness of an enclave-enabled operation when all of the state fits into enclave memory. The second uses operations where state does not fit in memory. For each query, we compare to 1.) a plaintext version (i.e., no enclave) and 2.) a version where operators are compared with a computation-enabling encryption version. We further test by altering the input rate, selectivity, and size of the data tuples.

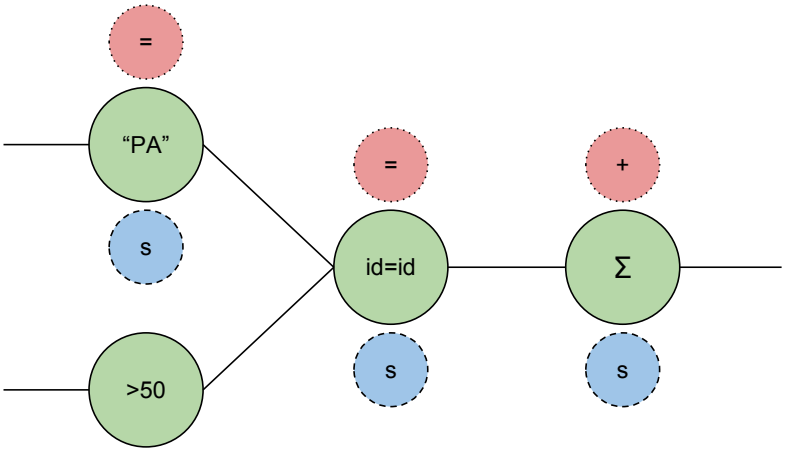


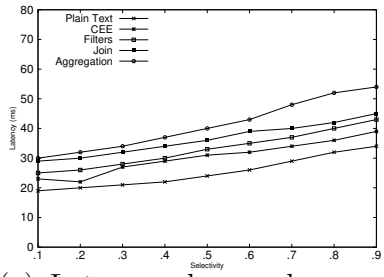
Figure 19: Example continuous query. The operations in the red dotted circles execute on computation-enabling encrypted tuples ("=" for deterministic, "+" for homomorphic), the blue dashed circles represent non memory-limited enclave-enabled operations, and the green solid circles represent plaintext operations.

4.6.4.1 Non Memory-Limited Query The query we use to evaluate the operations that fit entirely in enclave memory is intended to test each of the operation types (i.e., join, aggregation, and a stateless operation) on a real system. The query (below) aims to get the total profit for all companies whose profit margin is more than $500. The query is presented below in SQL and graphically in Figure 19:

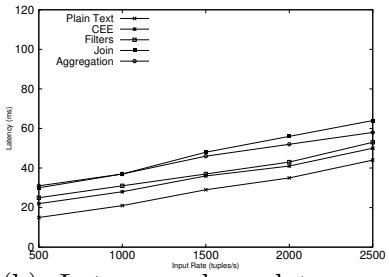
```
SELECT SUM(t.sales) FROM t JOIN o
WHERE t.id = o.id AND t.state = "PA"
  AND o.profitMargin > 500
GROUP BY t.company EVERY 30s UPDATE 5s
```

There are two filter operations (the "=" in Figure 19), a join (the "*id*=*id*" in Figure 19), and an aggregate-group by operation (the "+" in Figure 19). The encrypted versions of the

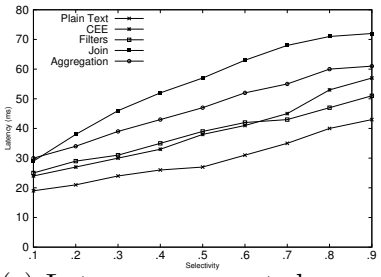
operations are presented in dotted red circles in Figure 19.



(a) Latency when each operation's data can fit in enclave memory; changing selectivity.



(b) Latency when data can fit in enclave memory; with changing input rate.



(c) Latency as executed on an SGX enclave with changing selectivity.

Figure 20: Changes in latency as changing input selectivity and input rate.

Results: (Figure 20) In this experiment, we wish to evaluate the overhead of introducing Sanctuary and SGX-enabled operators into the normal query processing pipeline. When evaluating a streaming query, one must consider the effects of input rate, selectivity, and tuple size on the responsiveness of the system. In this section, we explore changes in input rate and selectivity since tuple size was explored through micro-benchmarking. For each result, we evaluate a completely plaintext set of operations, a set of operations using only computation-enabling encryption, and then queries where each of the operator types (the summation, the join, and a filter) are all placed on an enclave using one of the algorithms from Section 4.4.

Selectivity (Figure 20a): Here, we change the overall selectivity and inspect the result. To change the selectivity, we simply increase the selectivities of each of the filters and the join incrementally to reach the desired selectivity (.1-.9) while maintaining an input rate of 1,000 tuples/s. As you can see from Figure 20a, a decrease in selectivity generally causes an increase in individual tuple latency, regardless of the operator type. Note that latency here includes the window time for each tuple. An increase in latency signifies that operators have difficulty processing data within a given window. Given that all of the operator state fits into memory, the effects of increasing selectivity was equally beneficial for all operators.

Input Rate (Figure 20b): We evaluate the input rate by increasing from 500 to 2,500

tuples/s (kept sufficiently slow so that we can evaluate scenarios wherein the input rate does not cause the memory capacity to be exceeded) and inspecting the latency. Similarly to the selectivity result, we notice from Figure 20b no significant difference between each operator type with increased throughput. We also notice that latency was within 15% of plaintext, and 7% of computation-enabling cryptosystems for SGX-enabled operators.

4.6.4.2 Memory-Limited Query For our memory-limited query, we simply re-use the query above, but with a drastically increased workload to force the utilization of non-enclave memory.

Selectivity (Figure 20c): For this experiment, we increase the input rate to 20,000 tuples/s (where only roughly 10,000 tuples can fit into memory. We otherwise keep the same configuration as the non-memory-limited version above. Notice from Figure 20c that selectivity has a greater impact on latency in this scenario. Specifically, when more results are filtered, less state is kept, and therefore SGX operators benefit (especially the join operation) since less state is needed to compute the final result, with fewer iterations to non-enclave memory. Specifically, a join performs up to 2.5x faster on data with very low selectivity versus very high, and an aggregation can perform up to 2x faster.

Input Rate (Figure 21): We generate input rates from 10,000 to 50,000 tuples/s to explore the impact on throughput of SGX-enabled operations. We can immediately see from Figure 21 that higher input rates negatively affect latency across the board, but more noticeably for SGX operations. This is expected since it will increase the state being stored in non-enclave memory, and therefore increase the overall processing time per tuple. In some instances we see the join operation increasing latency as much as 78% for large input rates, and as much as 31% for aggregations versus plaintext, and 58% for joins and 2% for aggregations versus a computation-enabling encryption operation. Note that this increase in throughput allows a user to have near-minimal leakage when compared to the encrypted version, and also allows for third-party joins to be executed.

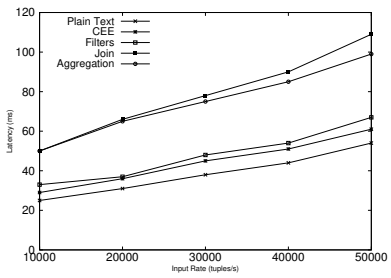


Figure 21: Latency of each operation type as executed on an SGX enclave with changing input rate.

4.6.5 Summary

As discussed in motivating this thesis, the current state-of-the-art access control enforcement systems for Data Stream Management Systems either rely on trusted third-party systems to enforce controls, or use some form of computation-enabling encryption that limits query expressiveness, increases data transmission overheads, and can leak information about underlying plaintext values. In this chapter, we detailed Sanctuary to implement and evaluate and a method for using Intel's SGX as a trusted computing base for executing streaming operations on untrusted cloud providers. In doing so, we are able to overcome the limitations of the state-of-the-art computation-enabling systems by allowing for *any* query to be executed on an untrusted machine while maintaining *near-ideal* level information leakage. Moreover, we discuss and resolve issues related to enclave memory size limitations by introducing memory-aware, stateful streaming operators. Finally, we demonstrate that the use of enclave-based processing in a streaming environment incurs only modest overheads when compared to the state-of-the-art systems. The use of a Sanctuary algorithm in a streaming operator allows for better *utilization* of third-party systems as well as better *protection* for a data consumer's data (given that it is fully encrypted), and can even lead to better *performance* in certain situations. When combined with our optimizer Shoal (Chapter 5), these operators can even be considered at *optimization time* to produce better and more efficient query plans through better operator selection and placement.

5.0 SHOAL

In this chapter, we present Shoal [53][1], our optimizer that considers the heterogeneous nature of Data Stream Processing Systems at optimization time. In the previous chapters, we have simply created new operators or altered the way data was presented. Here, we aim to improve query performance by considering these operators or access control changes at optimization time. We show that such considerations increase query plan quality versus the state of the art (by up to 7-8x) and can do so in both offline and online fashion.

5.1 INTRODUCTIONS

Given the transient nature of streaming data, a DDSPS will need to adapt and reconfigure streaming computations over time. Fluctuations in network utilization or data frequency can change the overall system load, which could result in over-loaded or underloaded physical systems. Similarly, the underlying physical systems are not homogeneous and provide potential users with a choice of computational elements that vary in capacity, memory, network availability, and even geographic location. Fluctuations and differences in physical systems and network utilization can greatly impact the performance of a streaming computation, whether it is by overloading a network connection, sending data across a geographically distant network, or simply overloading a single machine, and should therefore be considered when deciding how to order and place individual pieces of the streaming computation.

In addition to the physical fluctuations and differences in a DDSPS, there are also fluctu-

[1]Shoal appears in the 33rd Annual Conference on Data and Applications Security and Privacy (DBSec) in 2019, Charleston, South Carolina [53].

ations in the underlying streaming data that should be considered when placing and ordering individual pieces of a streaming computation. A data stream can change in both speed and data variability over time. For instance, a weather monitoring stream can produce different rates of data during a severe weather event versus a normal day. The weather stream can also alter its underlying data characteristics in such events as well (e.g., wind speed reading could be higher and therefore produce more results for a streaming computation that executes with higher wind speeds, increasing its selectivity). Finally, a data provider can alter their access control policies to allow for differing levels of access to their data stream which can then lead to different potential utilizations of the underlying system. Furthermore, a data consumer can take advantage of potential computation-enabling cryptographic techniques that allow for execution directly on encrypted data, resulting in further potential utilizations of the underlying system (e.g., decrypt-process-encrypt vs. a computation-enabling cryptographic scheme).

When a data consumer authors a streaming computation to be optimized for execution on a DDSMS, an ideal streaming computation optimizer would consider the underlying physical system as well as data stream characteristics when optimizing and placing the streaming computation. Such an optimizer could leverage heterogeneity (as described above) in the data streams and physical systems to produce an optimization and placement plan which better utilizes the underlying system. Furthermore, an ideal optimizer should provide some guarantees as to the quality of the resulting optimization and placement plan.

Thus far, DDSPS optimizers and related work have explored DDSPSs optimization in a limited scope. Some have simply focused on better utilization of the underlying computation hardware alone [54], while others have focused on the underlying network alone [55, 56, 57, 58]. Several optimizers and systems have focused on the impact of data variability on the system in the presence of access controls [3, 4] and in the impact of data stream rates and selectivities [28, 59]. No single system strives for a general optimization approach that can encompass a variety of heterogeneous characteristics simultaneously at runtime. Furthermore, most of these data streaming optimizers use an *optimize-then-place* approach where a user's query is first optimized for non-distributed execution and then post-processed for placement on distributed resources. *This optimize-then-place approach can result in sub-*

optimal plans since network and system loads may be adversely affected by certain orderings of operations (as we show in Section 5.2). To better optimize streaming continuous queries for heterogeneous, we introduce Shoal, which makes the following contributions:

- We show that the optimize-then-place approach is a sub-optimal approach for computing operator placement in a DDSPS.
- We introduce the first cost model for distributed streaming computations that leverages parallelism inherent to the DDSPS and accounts for key sources of heterogeneity such as fluctuations and changes in the underlying data streams and the underlying system and network.
- We detail an offline algorithm for the optimization and placement of operators in a DDSPS using dynamic programing. This approach is guaranteed to be optimal while considering key sources of heterogeneity at optimization time with reasonable overheads for moderate-sized streaming queries.
- We detail an online re-optimization algorithm that can execute when a change in the system is detected. This algorithm only optimizes the parts of the query that are potentially affected by a change in the system or data streams. By only considering the parts of the query that are affected by system changes, this online algorithm is able to quickly re-optimize and recover while maintaining an optimal solution.
- We run an extensive evaluation of our algorithms and compare to several baseline, state-of-the-art, and optimize-then-place algorithms. We show that our proposed framework can produce higher quality optimization and placement plans (up to 120% better) with reasonably low overheads, and further show that these plans are of a higher quality when compared to related work.

5.2 PROBLEM DESCRIPTION

In this section, we detail the exact optimization problem addressed by our framework and the different kinds of heterogeneity inherent in a Distributed Data Stream Processing System

(DDSPS). We offer a description of our optimization approach, as well as show the optimize-then-place approach to be suboptimal.

5.2.1 Problem Description

In order to properly define the problem being addressed here, we must first formalize the required components. There are three main components to optimizing and placing a data consumer's computation: *third-party systems*, *operations queries*, and *query networks*.

Definition 5.2.1. ***Third-party Systems****: As introduced in Section 2, a third-party system s executes operators. Interconnections between third-party systems have* bandwidth *(in bits/s, where tuples can be of varying size) and* latency*, (in ms) characteristics that we represent as $b(s_1, s_2)$ and $l(s_1, s_2)$ respectively. Third-party systems are associated with the following properties:*

- *$s.cap$ is the third-party system's processing capacity in (in cycles translated to tuples/s.*
- *$s.name$ is the third-party system's name used for unique identification purposes.*
- *$s.per$ is a set of permissions $\{< o, f >|$ third-party system s can execute a physical operator o on field $f\}$.*

Definition 5.2.2. ***Operation:*** *An* operation *op is a set of operators that execute the same task via different physical implementations.*

- *$op.type$ represents the action to be preformed on a data stream (e.g., filter, projection, summation, top-k, etc.).*
- *$op.args$ includes metadata about the operations such as the join condition or selection criteria.*
- *$op.input$ represent the set of fields required for this operation to execute.*
- *$op.output$ represents the set of fields in the output of this operation.*
- *$op.id$ a unique identifier for the operation.*

A typical operation can be a filter over someone's age, a join to match two streams, or an aggregation to find the maximum profit in a given window of time. Operations can be implemented in many different ways using different techniques, represented as operators.

Definition 5.2.3. ***Operator:*** *The basic computational unit used in Shoal is an* operator *o, which has the following properties (for brevity, we refer to third-party systems as a* third-party system*):*

- *o.s represents the expected or actual* selectivity *of the operation. Selectivities can be derived either by estimation, measurements during a warm-up period, or historical selectivity data.*
- *o.c represents the* cost *of the operation in terms of the latency for computing on one tuple. It can be calculated in a manner similar to the selectivities.*
- *o.tps represents the* third-party system *an operator has been assigned to.*
- *o.opId represents the ID of the operation (that is to say, the logical operator that this physical operator represents) that this operator implements.*
- *o.window represents the window for a stateful operator either in tuples or ms, with a default of 0.*

Operators allow a data consumer to have flexibility when implementing an operation. Consider the potential difference between a hash-join implementation of a join operation versus a merge-join implementation. Given the input rate and selectivity of each stream, it is highly likely that one join would outperform the other in terms of overall latency. An operation would have the merge-join and hash-join as potential operators, and each operator would have a cost that can be used to better optimize the query network.

Definition 5.2.4. ***Query:*** *A* query *is represented as a set q of operations that describe the query or task a data provider wishes to execute over a set of data streams.*

- *leaves(q) returns the set of operations that operate on a raw data stream (i.e., do not require the output of another operation to execute).*

Definition 5.2.5. ***Query Network:*** *A* query network *is represented as a set qn of queries that will execute within the DSMS.*

- *sinks(qn) returns the set of operations that return a result to a querier (i.e., the last part of any one query).*

Using a query, permissions, and a set of available third-party systems as input, Shoal produces a *plan* as output:

Definition 5.2.6. ***Plan:*** *A* plan $p = (V_o, E_o)$ *where* V_o *is a set of physical operators and* $E_o \subset V_o \times V_o$ *is the edge set linking the outputs of one operator to the inputs of adjacent operators.*

Definition 5.2.7. ***Satisfiability:*** *A plan* p satisfies *a query network* qn *if:*

- $\forall op \in qn, \exists!\ o \in V_o$ *s.t.* $o.opId = op.id$
- $\forall o \in V_o, \exists!\ op \in qn$ *s.t.* $op.id = o.opId \wedge < o, o.metadata > \in o.tps.per$
- $\forall o \in p, o.input \subseteq \bigcup_{o' | <o', o> \in E_o} o'.output$

That is to say that the each operation in each query that comprises the query network has a unique operator in the plan and that each operator in the plan is the implementation of one operation in the query, and that each operation in the query is represented in the resulting network. Additionally, each operator's input must be part of the output of its immediate predecessor, and each operator must be permitted to execute as implemented on its third-party system.

To determine the relative quality of a given plan p, we use the following cost model

Definition 5.2.8. ***Cost:*** *For a plan* p *of a Query Network* qn*, the* cost *of* p*, starting at the leaf node(s), is determined by:*

$$\max_{path \in \mathsf{Paths}(p)} \mathsf{pathCost}(path) \tag{5.1}$$

where Paths is the set of paths from leaf nodes to sink nodes. The expected Input Rate for each operator (starting from the initial input rate of the leaf node from the source stream) is:

$$\mathsf{ir}(o_i) = IR_{qn} * \prod_{op_j \in \mathsf{pathUpTo}(op_i)} op_j.s \tag{5.2}$$

The function pathUpTo(o) for an operator o is the ordered subset of operators that precede o in the plan (as part of the same query). IR_{qn} is the maximum input rate of any leaf operator on the current path. The cost of a path is:

$$\mathsf{pathCost}(path) = \sum_{op_i \in path} \max(op_i.c, op_i.window) + \mathsf{Latency}(op_i, path) * \mathsf{Penalty}(o_i) \quad (5.3)$$

The penalty is defined as:

$$\mathsf{Penalty}(o) = \begin{cases} 1, & \text{if } \mathrm{pr}(o, o.tps) > \mathrm{ir}(o_i) \\ \frac{\mathrm{ir}(o)}{\mathrm{pr}(o, o.tps)}, & \text{otherwise} \end{cases} \quad (5.4)$$

The function $\mathsf{pr}(operator, tps)$ determines what the processing rate of a third-party system would be with the operator o assigned to it. If this processing rate is greater than the input rate, then there is no penalty. If the processing rate is insufficient to handle the input rate, the plan is penalized by the input rate over the processing rate. Latency is computed as:

$$Latency(o, path) = \begin{cases} 0, & \text{for } o \in leaves(qn) \\ \mathrm{l}(op, op_{i-1}), & \text{for } op_{i>1} \end{cases} \quad (5.5)$$

Constrained by $s.cap$ and $bandwidth(op_i, op_{i-1})$.

Given the above components, Shoal solves the following problem:

Definition 5.2.9. ***Problem:*** *Given a query network qn of queries, a set of third-party systems s, and a set of access control permissions perms, produce a plan p that satisfies qn such that Equation 5.1 is minimized.*

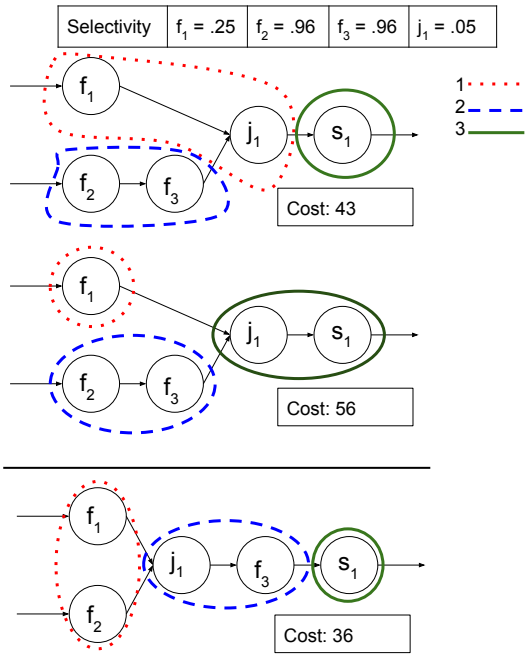


Figure 22: Simple continuous query.

5.2.2 Optimize-then-place Approach

A reasonable first step solution to this optimization problem would be to separate optimization from the actual operator placement. This would allow a placement algorithm to simply use an existing optimizer (e.g., a continuous query optimizer or a map-reduce scheduler) and post-process the result for placement. This approach, however, can lead to a sub-optimal plan even if the query itself is fully optimized. Consider the following scenario for a continuous query optimizer, which we will use throughout the remainder of this chapter to illustrate Shoal:

A simple query contains three filters (f_1,f_2,f_3), a join (j_1), and a projection (s_1) which are arranged in the fashion depicted in the top of Figure 22 as the result of an optimization step. Next consider the three third-party systems available for placement, and assume the network cost is uniform. Each third-party system has a capacity of 10 (unit-less for simplicity). The cost of each filter is 4, of the join 6, and of the project 2; each have selectivities shown in Figure 22. Given these costs, either the join must be co-located with f_1 (The top of Figure 22)

or separated from all of f_1, f_2 and f_3, (the middle of Figure 22). However, notice that the selectivity of f_1 combined with f_2 is .92, meaning that a tremendous amount of data is being sent over the network to the join. The selectivity of the join, however, is far lower at .05, meaning that a smaller amount of data is being produced. If the query was instead optimized so that f_3 were to follow the join, the overall network cost would be substantially reduced, resulting in a higher quality plan (78.8ms vs. 67.8ms with Equation 5.1). This illustrates the need for the optimization and placement steps to be considered simultaneously.

5.2.3 Addressing Heterogeneity

During the optimization step, Shoal simultaneously considers four key causes of heterogeneity in the DDSPS, listed below:

Hardware: In a distributed streaming system, a data consumer can take advantage of third-party providers to augment their own computing capabilities, and each of these providers can offer different types of physical systems. In Shoal, each new *third-party system* can represent different physical third-party systems that can be controlled by different providers. Shoal considers all of the different assignments to determine which third-party systems to use, and can therefore take advantage of a heterogeneous physical system.

Network: The physical location of each operator can greatly impact the effect on the quality of the plan generated by Shoal. If operations that share an edge in the plan are physically and geographically separated, the overall plan latency could be higher than if the operations were not physically and geographically separated. To compensate for the effects on latency due to the physical location of each operation, Equation 5.1 considers the latency between two operators as part of the overall quality of the plan which allows plans with lower physical separation to be favored over others, allowing Shoal to reduce the impact of network heterogeneity.

Access Controls: A data provider can enforce many types of access controls impacting the level and type of access a data consumer can have to their data. Depending on the accesses granted and the type of enforcement being used (e.g., computation-enabling encryption or third-party enforcement), a single streaming operation can be physically im-

plemented in different ways. Shoal considers all of the available alternatives and selects the best use of the permissions based on which permissions produce the best quality plans.

Data Variability: In a streaming system, the underlying characteristics of a stream can change over time. These characteristics include the selectivity, speed, and size of the data. To accommodate different selectivities, Equation 5.1 ensures that there is sufficient capacity on a third-party system to handle the data being processed at that third-party system. To accommodate for different data rates, Equation 5.1 calculates the input rate of a given operation to correctly calculate the work that may need to be done. Finally, the size of the data is taken into account by considering a maximum capacity for the network bandwidth, and by limiting the memory used on a device.

5.3 THE SHOAL OPTIMIZER

In this section, we introduce our online and offline optimization and placement algorithms.

5.3.1 Offline Placement

Before detailing our offline algorithm, let us first present the intuition behind it. Traditional database management systems and distributed database management systems typically make use of *dynamic programming* for query or multi-query optimization. Dynamic programming algorithms are guaranteed to find the most optimal query plan by building upon smaller optimal plans [60].

For a DDSPS to make use of a dynamic programming optimizer, each dynamic step of the algorithm should consider both the order of the operations that make up the streaming queries, as well as the third-party system on which the operation can execute. Dynamic programming optimizers used in traditional single-machine database management system need not worry about operator placement since it is assumed that a queries will execute on a single third-party system. Shoal adapts traditional dynamic programming approaches so that operator placement is included when deciding on an order as well. Furthermore,

Shoal considers potential different physical implementations for each operation given different possible algorithms or data types for executing the same task (e.g., merge-join vs. hash-join, plaintext vs. computation-enabling encryption, etc.).

To leverage the benefits of dynamic programming, we use Algorithm 6. Much like traditional dynamic programming approaches, Algorithm 6 starts with the operations that access the raw streams (Line 2). An optimal placement is found for each of these operations one by one (Lines 5–9). From this point, we consider each operation in turn, both for placement and ordering, even if they originate from different queries. Plans are dynamically built so that the addition of new operations are direct extensions of the previous plans (Line 20). Each of these operations are then considered on each third-party system and for each permission level that the corresponding data consumer has been given (Lines 15–19 and Lines 22–26). Algorithm 6 also verifies that the current third-party system has enough anticipated capacity and permissions to support the operations for a given permission (Lines 7, 17). The function $operators(o)$ enumerates all of the possible physical operator implementations for the operation o. This enumeration includes all of the different algorithms (e.g., hash-join vs. merge-join) and all of the different data access types (e.g., deterministic encryption vs. homomorphic encryption). Once an operation is optimized, plans are pruned using $prunePlans()$ so that the new plans that are kept are optimal for each available permission on each third-party system (Lines 10, 27). $prunePlans()$ ensures that each operation's physical operator is kept on its best available third-party system so that future plans may better utilize potential physical implementations to reduce latency. Once all operations are optimized, the new optimization and placement plan is returned. If, however, an operation remained unplaced due to size limitations and/or insufficient privileges, no plan is generated for that iteration.

So long as there is sufficient capacity, Algorithm 6 will return the optimal resulting plan for each operation given the permissions granted to the data consumer. It is important to note that Shoal works with the physical *operator* placement and not an *operation* as presented in Section 5.2, since an operation can be physically implemented as many different operators. Shoal considers all permissible physical implementations (Lines 7, 17) of an operation and selects the most optimal choice at each step.

Algorithm 6 DynamicProgramming

```
1: DynamicProgramming(sc, perms, tpss)
2: optPlace= new Array(ArrayList(plan))
3: for leaf ∈ leaves(sc) do
4:   optPlace[0] = ∅                                          ▷ Initialize Empty list at level 0
5:   for s ∈ tpss do
6:     for l ∈ operators(leaf) do
7:       if s.cap ≥ l.c && (< l, l.metadata.field >∈ s.perms) then
8:         l.tps = s
9:         optPlace[0].add(new plan(l))                      ▷ Capacity kept per plan
10:  prunePlans(optPlace[0])
11: for lv = 1...|sc| do
12:  optPlace[lv] = ∅                                        ▷ Initialize Empty list at level lv
13:  for operation ∈ sc | operation.type = join do
14:    for plan1, plan2 ∈ optPlace[lv − 1] | (operation.input ⊆ (plan1.output ∩ plan2.output)) ∧ operation.opId ∉
       plan1.opIds ∧ operation.opId ∉ plan2.opIds do
15:      for s ∈ tpss do
16:        for join ∈ operators(operation) do
17:          if (updateCapacity(plan1, plan2, s) ≥ join.c) ∧ (< join, join.metadata.field >∈ s.perms) then
18:            join.tps = s
19:            optPlace[lv].add(joinPlans(plan1, plan2, join))
20:  for plan ∈ optPlace[lv − 1] do
21:    for operation ∈ sc | (operation.type ≠ join)∧(operation.opId ∉ plan.opIds) ∧ (operation.input ⊆ plan.output) do
22:      for s ∈ tpss do
23:        for o ∈ operators(operation) do
24:          if plan.s.cap ≥ o.c∧ < o, o.metadata.field >∈ s.perms then
25:            o.tps = s
26:            optPlace[lv].add(combine(plan, o))
27:  prunePlans(optPlace[lv])
```

5.3.2 Online Update Placement

Given the long-running nature of continuous queries, there is a high chance (essentially a certainty) that the system will change over time, requiring re-optimization of the network of streaming queries currently deployed. This re-optimization can be triggered in several ways. Our framework triggers re-optimization for three different scenarios: 1.) access control updates, 2.) changes in query result's latency or throughput, or 3.) the addition of new hardware that can impact the query (faster, higher capacity, not just *more*). To accommodate for these changes, there are two possible approaches: *stop-the-world* and *on-the-fly*. The stop-the-world approach simply halts query execution and uses Algorithm 6 to re-optimize the query. This approach, however, can lead to large re-optimization times for larger query networks, and can end up doing repetitive work when a relatively small set of operations are affected by the change.

Instead, we introduce an *on-the-fly* approach to mitigate optimization overheads. The principle behind our on-the-fly approach is to execute Algorithm 6 from the operator that is

Algorithm 7 Access Control first-impacted identifier.

```
1: ACUpdate(Update u, Plan p)
2: cld = sc.leaf                                   ▷ Operations not processed
3: for o ∈ cld do
4:   if o.input | u.protectedFields then
5:     return p.levelOf(o)
6:   else
7:     cld.add(p.childrenOf(o)
8:   cld.remove(o)
```

first affected by the update relative to the data providers of the overall query network, which we will call the *first-impacted*. This requires the ability to determine the first-impacted, which depends upon what type of event had triggered the update. For the three scenarios covered by Shoal, two of them can benefit from locating and re-optimizing from the first-impacted operator. The addition of new hardware should require an optimization of the whole plan since it can not be easily determined which operators would benefit from new hardware without first testing it there.

Access Control Algorithm 7 determines which operators are first-impacted by a change in access controls. The algorithm starts by adding operations that directly access raw data streams on Line 2 to the current query network, *cld*. These operators are then looped through on Line 3, and Line 4 determines if that operator accesses the data being protected by the new access control update. If so, this operation is the first-impacted and the algorithm determines its level by asking the plan for the level. If the operation does not access the protected data, its children are added to the *cld* set, and it removes itself from this set. This continues until the first-impacted is found. Once the first-impacted operators have been identified, Algorithm 6 will execute on the *descendant* children of the first-impacted, as well as all operations at the same level and their descendants. This leaves the already optimized operations and their ancestor operations intact from the previous plan, and re-optimizes the operations at and after the first-impacted's level, leading to less optimization time. The only alteration required for Algorithm 6 is the inclusion of the current plan from which to start, which is simply placed in the *optimalPlans* set and the Algorithm starts from Line 11 where the level is determined by traversing back to the leaf nodes.

System Changes The second trigger requiring re-optimization is changes in latency or

Algorithm 8 System Changes first impacted identifier.

```
1: SystemChanges(Plan p)
2: tpss = p.getTpss()
3: ea = ∅                                                    ▷ All first-impacted
4: for s ∈ tpss do
5:   if s.latency > lt ∨ s.throughput < tt then
6:     ea.add(p.firstOperatorLevel(s))
7: Return minimum(ea)
```

throughput. If latency exceeds a user-defined lt threshold or if throughput falls below a user-defined threshold tt, an update is triggered. Unlike updates to the access control policies, throughput and latency fluctuations are the result of system fluctuations, and therefore require the examination of third-party systems rather than operators to determine which operators need to be re-optimized. Algorithm 8 details how the first-impacted are determined in this scenario. Put simply, the first-impacted in this scenario are the first operators on third-party systems that are currently under the most load. These are identified in Line 5, and each third-party system's first operation's level (i.e., the level of the plan at which the operation resides determined by the function $firstOperatorLevel$) is added to the first-impacted set (Line 6). This is done for all third-party systems, and Algorithm 6 begins similarly to the Access Control first-impacted in that it starts with the first-impacted operation's level (as determined by the minimum number found in the first-impacted set).

Additional Hardware The final trigger that can require a re-optimization is the addition of new hardware. In this case, there is no first-impacted operator. Instead, the entire query should be re-optimized to take advantage of any hardware gains. Such hardware may contain an Intel SGX-enabled processor that could allow an operation to be implemented with an enclave, which could benefit the user, or simply be faster hardware with more capacity.

It is important to note that monitoring fluctuations in latency and throughput encompasses heterogeneity in the network and data variability. Network congestion and changes in bandwidth will alter the throughput and latency of an executing operator, which could trigger an update given their severity. Additionally, if underlying selectivities are altered, an operator may require more resources than available on its current third-party system,

causing an increase in latency and/or a decrease in throughput and a potential update.

Discussion While finding the first-affected in any of the above cases, Shoal can guarantee optimality for resulting plans when the only changes to the system are those that brought about the re-optimization in the first place. Shoal can not guarantee optimality in the online, on-the-fly case when *other* system properties have changed in addition to the one causing the re-optimization. For instance, if a data provider issues a change in access controls, Shoal can use Algorithm 7 to detect the first affected operator and reoptimize from there. If this change in access control causes larger packets to be sent between operators on different sites, it may have been better to take an operator that was originally downstream and now move it to before the first-affected operator to reduce the overall network latency. Moreover, if system characteristics have changed but not up to the threshold considered for re-optimization, an entire re-run of Shoal could result in a different plan. Consider the simple case query below and depicted in Figure 23.

```
SELECT s1.Name, s2.ID, AVG(s3.heartrate)
FROM s1, s2, s3
WHERE s1.id=s2.id
AND s2.id=s3.id
EVERY 5 minutes UPDATE 2 minutes;
```

Here, we see that the updated first join operator (consequence of an access control update) was past the system of joins that precede it. Notice, however, that the update that simply changes the operator but keeps the same plan at that point in time (the upper right of the figure) results in query plan that has a higher cost that the previous plan, even though this section remained unaffected by the update. This is caused by the steady change in selectivities over time. We can see that the selectivity of the first Join has dropped, making it send more data to the second join. The bottom network in Figure 23 shows the result of a reoptimization from scratch. Notice here that the plan has actually changed to allow a more selective operator to go ahead of a less selective operator, resulting in a marginally better query plan. If a query network were sufficiently large, these small deltas in plan cost could add up to substantial cost (i.e., latency) savings.

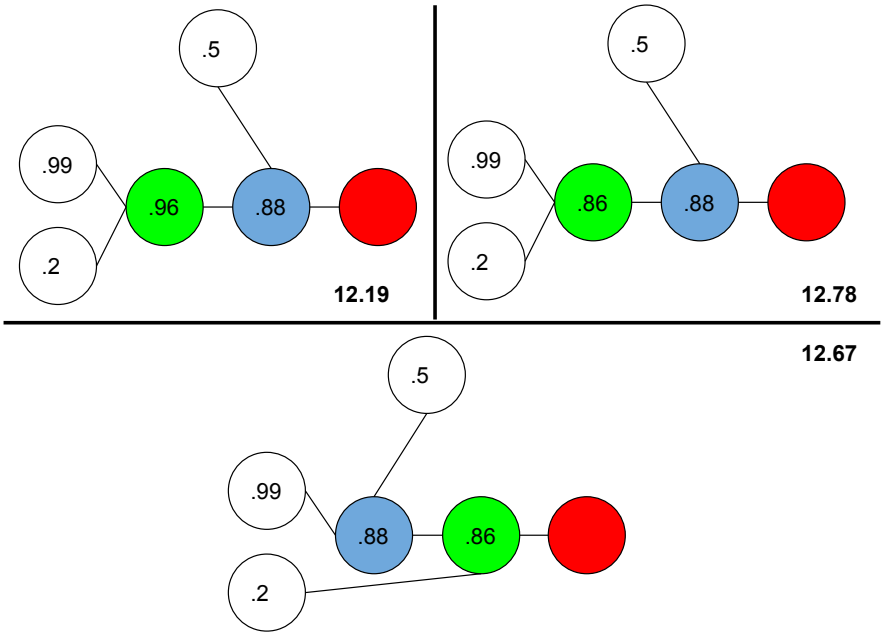


Figure 23: Simple query plan showing that running Shoal from the first-affected may be sub-optimal.

Although Shoal does process changes in selectivities, input rates, and hardware, there must be a certain threshold for which each type of change must adapt. If the selectivity threshold were set too low, the query would constantly be reoptimized. If the threshold is too high, the query may not be optimized frequently enough to mitigate increasing costs. However, this threshold setting is outside of the scope of this dissertation and would make for interesting future work. Given an unchanged state of operators leading up to the first-affected operator, Shoal can guarantee an optimal updated plan since Shoal would have produced the same plan given the same input. Otherwise to guarantee optimality, Shoal would need to be executed from raw input.

5.3.3 Greedy & Hybrid Approaches

As with traditional dynamic programming optimizers, our dynamic programming algorithm begins to suffer from prohibitively large execution times for large or complicated query networks (explored further in Section 5.4). When query networks become too large or complex,

we defer to a greedy approach. This approach simply considers one operator at a time and optimally places it. In the offline case, the user defines a time threshold $t_{offline}$ for their optimization step. If the dynamic programming approach is expected to exceed $t_{offline}$, then the greedy approach is used. The online approach poses a different problem because there may be uncertainty in how costly an update may be to the system (i.e., the number of operators that need to be re-optimized).

The larger the number of operators that need to be considered, the greater the number of operators requiring re-optimization, and therefore the greater the cost of the update. In a system operating at or near capacity, online updates may end up hindering the quality of the result as some information may be lost during optimization, especially for costly updates on large overall query networks. To combat this problem, we use the greedy approach when updates are too costly relative to the system load. The greedy approach simply re-optimizes, placing each operator in the most optimal location, in a quick but likely non-optimal fashion.

The greedy approach lends itself nicely to distributed systems with heavy load where re-optimization needs to be quick to avoid losing data, but it will not produce plans of the same quality as the dynamic programming approach. To help a data consumer determine which to use, we propose a *hybrid* solution which automatically determines which approach to use given the current system state. The determination is based on three factors relative to the overall streaming query network submitted by the data provider: 1.) buffer capacity, 2.) processing time of a single streaming tuple (end-to-end), and 3.) the input rate. When an update is deemed necessary, its cost c in seconds is determined by multiplying the number of operations needing to be re-optimized by the average amount of time to optimize one operator (based on the offline execution time, or a running average). Then, the following equation is used to determine which algorithm to use:

$$uc = (\sum_{i}^{o \in p}(b_i * t_i)) + ir_o * c \tag{5.6}$$

where o is the operator in the plan p, b_i is the utilized buffer size of o, t_i is the processing time of o, ir is the input rate. If $c < uc$ then the dynamic programming approach is used, otherwise the greedy approach is used to minimize data loss.

5.3.4 Example

To help illustrate how Shoal optimizes a set of streaming queries, consider the following continuous query on a data stream that contains tuples with a *timestamp*, *companyName*, *companyId*, and the company *profit*, as illustrated in Figure 24:

```
SELECT max(avg_profit), companyName
FROM (SELECT companyName, AVG(profit) as avg_profit
    FROM profitStream
    GROUP BY companyName
    EVERY 1m UPDATE 15s;)
GROUP BY companyName
EVERY 1m UPDATE 15s;
```

This query requires five operations; a max (m), a projection (p), an average (a), and two group-by operations (g_1 and g_2) represented by circles in Figure 24. Assume that the profit field is protected by a Homomorphic encryption and the others are plaintext. Further assume (for simplicity) that there are two third-party systems A and B with capacities 10 and 10 respectfully, and a latency of 10ms between them (squares in Figure 24).

Shoal starts with the operation a as it is the operation that accesses raw data. Since a homomorphic option exists for the aggregation, an operator executing a homomorphic scheme is put onto each third-party system and the next round of dynamic programming is initiated. Further, plans are also added for random encryption and trusted machine processing. This aggregation requires a group-by operation, which can execute on the plaintext column for "*company name*". This operation is placed with the aggregation on each third-party system, making each third-party system's best plan having a cost of 8 which, along with other plans with varying physical operators, are kept for each third-party system. Shoal then tests the remaining operations and determines that the projection p can be added to each third-party system's best plan for a cost of 9. Plans are now kept for each third-party system and for each physical operator, but the minimum plan score is 9. Note that this choice reduces the overall network load by eliminating all columns except the company name and the average. Shoal continues and determines that the maximum operation along with its group-by can

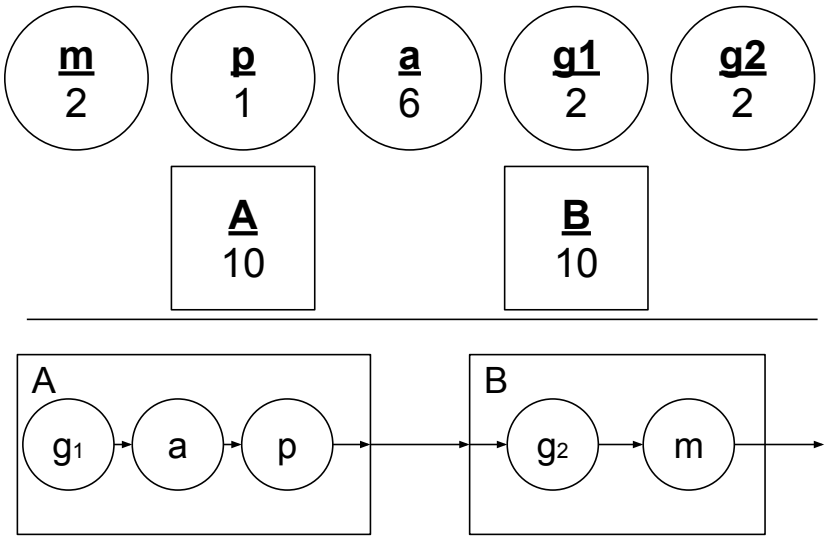


Figure 24: Given a the set of operations and the third-party systems A and B, Shoal optimizes and places the operations so that the first aggregation is placed on A with the projection reducing network load to the second aggregation operation placed on B.

not fit on either third-party system and chooses them to operate on third-party system B with third-party system A keeping its previous plan. With all operators placed, the new plan resembles the one in the lower half of Figure 24.

5.4 EVALUATION

To evaluate our optimizers, we decided to use relational continuous queries for the bulk of our experimentation.

5.4.1 Experimental Setup

Setup: For out evaluation, we limit Shoal to be used in a simple streaming system with a data provider, data consumer, and data processing components. For our simulation, data is streamed from a laptop into Amazon AWS EC2 instances where data processing occurs. Once data is processed, it is passed back to the laptop to act as the data consumer. We implement

the streaming layer on the Apache Storm [34] framework. To keep the streaming layer simple, we use the most basic functionality of Storm where our data provider implements a *spout* and our data processing nodes implement *bolts* with no multi-threating or replication (i.e., a bolt just mimics a machine for our purposes). We use storm *only* for the transport layer as it guarantees delivery and provides positive and negative acknowledgments. To simulate real world streams, each stream is imposed with an artificial latency of 0-30ms to emulate them being geographically separated.

Datasets: We use queries from the TPC-H [**?**] workload and modify them for use in a streaming system (e.g., aggregations use windows). We will explicitly call out any changes to the query we made, or if we use more than one query as part of the query network. We further segment data based on a timestamp so that it is streamed into the system (in a pre-processing step) so that days are equivalent to minutes. In a addition to the TPC-H workload, we use the taxi dataset from the 2015 DEBS Grand Challenge dataset [52] consisting of tuples that describe an instance of a taxi ride (i.e., the start time, taxi driver ID, cab ID, end time, fare, distance, etc.).

Baseline Algorithms: In addition to our online, hybrid, and offline dynamic programming algorithms, we chose three additional baselines for comparison: (1) *all-on-client*, where all of the operations run on one machine, (2) *first third-party system*, where each operation is placed on the first third-party system available, and switched to the next when either the third-party system is at capacity or there is a conflict with access controls, and (3) *greedy*, where a plan is generated by greedily assigning each operation based on the best score.

5.4.2 Offline Optimizer

We first compare our offline optimizer to other baseline approaches.

5.4.2.1 Optimizer Execution Time, Single Query This experiment evaluates the runtime of Shoal for different-sized queries, as well as comparing to baseline approaches.

Setup: We choose TPC-H queries with varying numbers of underlying operators and

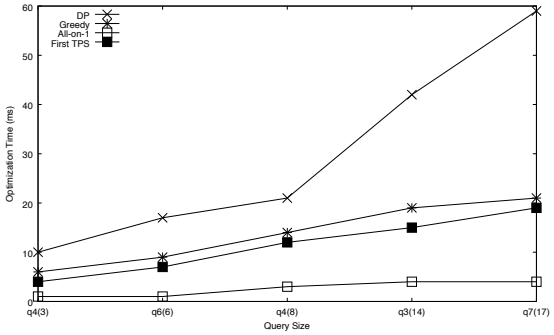


Figure 25: Optimization time (in ms) for the offline dynamic programming algorithm and baseline algorithms.

evaluate them in increasing order. We choose queries 4, 6, 3 and 7 with 8, 6, 14, and 17 operators respectively. We further break the sub-query in query 4 into its own query to test a small operator network for a total of five query sizes. All aggregation and join operations are given windows of 5 days (to directly use the date field in each relevant tuple).

Results (Figure 25): As expected, our dynamic programming approach has the longest optimization execution time amongst all of the baselines. Since it must compare far more plans than the baseline approaches as it optimizes each operation, it is expected that it will take longer to do so.

Takeaway: Although being the longest optimization time, the execution time of Shoal is still small with a highest of 59ms.

5.4.2.2 Optimizer Execution Time, Multiple Query

This experiment evaluates the runtime of Shoal for different-sized query networks.

Setup: We combine queries in increasing size order (i.e., 1 query, 2 queries, 3 queries, up to 4 queries, or 8, 14, 28, and 45 operators). This provides four data points with an increasing size and number of sinks. All aggregation and join operations are given windows

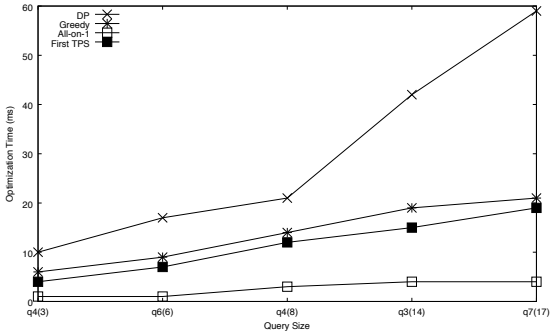


Figure 26: Optimization time (in ms) for the offline dynamic programming algorithm and baseline algorithms on multiple queries.

of 5 days (to directly use the date field in each relevant tuple).

Results (Figure 26): As in the single query case, our dynamic programming approach has the longest optimization execution time amongst all of the baselines. As above, however, we can see Shoal having execution times less than 100ms.

Takeaway: Although being the longest optimization time, the execution time of Shoal is still small with a highest of 78ms for 45 operators.

5.4.2.3 Optimizer Plan Quality, Single Query

This experiment evaluates the overall plan quality of each approach in terms of latency (ms). We present both the expected latency, as well as the actual latency.

Setup: This experiment has the same configuration as Experiment 5.4.2.1 and presents the expected and actual latency for each plan generated in Experiment 5.4.2.1 for the same queries. Once the query had been optimized, each was executed for 5 minutes with the average latency reported below.

Results (Figures 27 and 28): Our dynamic programming optimizer produces the lowest overall actual latency for each query, and does so while maintaining an average of a 3% difference in the actual versus expected latency for each query. Note that other techniques require a larger latency to produce the same result, and the All-on-1 baseline eventually overwhelmed its machine and produced no results (without load shedding).

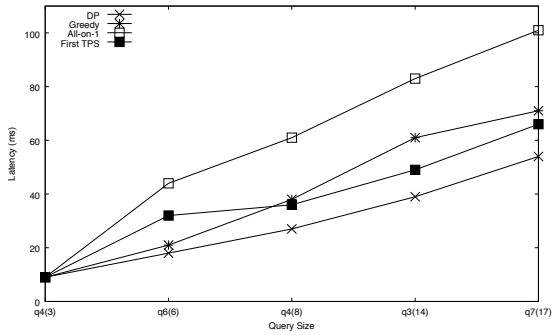


Figure 27: Expected latency in milleseconds for each query size as generated by each optimizer.

Takeaway: Our dynamic programming optimizer offers the best overall actual latency.

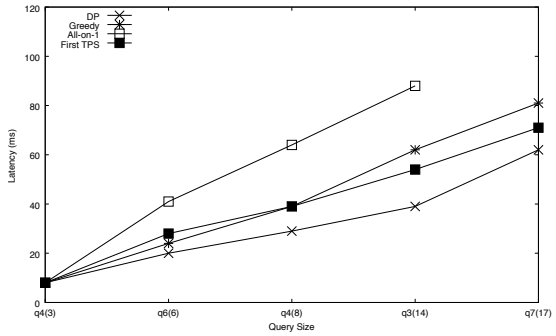


Figure 28: Actual latency in milleseconds for each query size over an execution time of five minutes.

5.4.2.4 Optimizer Plan Quality, Multiple Query This experiment evaluates the overall plan quality of each approach in terms of latency (ms). We present both the expected latency, as well as the actual latency for the multiple query scenarios presented above.

Setup: This experiment has the same configuration as Experiment 5.4.2.2 and presents the expected and actual latency for each plan generated in Experiment 5.4.2.2 for the same query networks. Once the query had been optimized, each was executed for 10 minutes with the

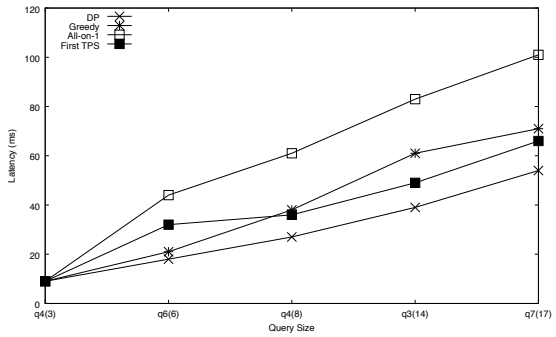


Figure 29: Expected latency in milleseconds for each query size as generated by each optimizer for a query network.

average latency reported below.

Results (Figures 29 and 30): Our dynamic programming optimizer produces the lowest overall actual latency for each query, and does so while maintaining an average of a 4.8% difference in the actual versus expected latency for each query. Note that other techniques require a larger latency to produce the same result, and the All-on-1 baseline eventually overwhelmed its machine and produced no results (without load shedding).

Takeaway: Our dynamic programming optimizer offers the best overall actual latency.

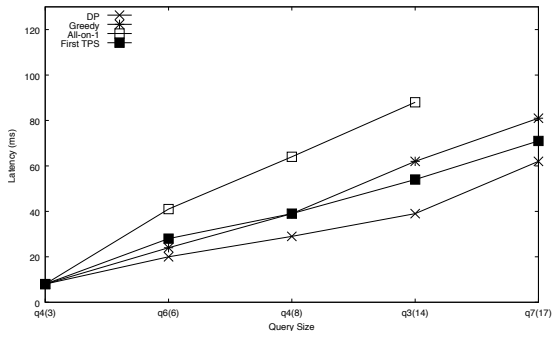


Figure 30: Actual latency in milleseconds for each query size over an execution time of five minutes for a query network.

5.4.3 Online Optimizer

This section evaluates our online optimizer as compared to other baseline approaches. Recall that the cost of an update is based on how many operations in the query network are affected by the update, so we omit cases where the entire network was updated since it would degenerate to the offline case, which we have previously evaluated.

5.4.3.1 Optimization Time Given the cost of an update, this experiment determines the average optimizer execution time for the online dynamic programming approach as well as the baseline approaches.

Setup: Using the same queries as in Experiment 5.4.2.2, we trigger updates so that only a certain number of operators in each query are affected by the update. Each optimizer is then used to order and place the subsequent operations.

Results (Figure 31): Much like with the offline optimizers, the online dynamic programming approach is the slowest. This optimizer execution time included the time to determine the first-impacted for each approach, for each query, which manifests itself as the slight difference in execution time between similarly sized queries in the offline approach.

Takeaway: Although Shoal has the highest optimization time, it is still relatively low, especially when executing on a long-running continuous query in a network where the resulting plan quality is much more important.

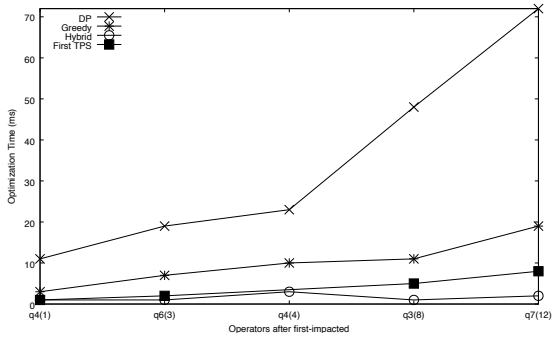


Figure 31: Optimizer execution time for the online algorithm approaches.

5.4.3.2 Plan Quality This experiment evaluates the overall plan quality of each approach in terms of latency (ms) for each updated plans. Again, we present both the expected latency, as well as the actual latency. Here, we include the hybrid approach to show when it may switch optimizers to reduce the overall impact of an update.

Setup: Similar to Experiment 5.4.2.4, queries are executed for 10 minutes in total. There is a two minute window for the initial query, after which an update is presented. The query is then updated and the remainder of the time is spent monitoring the updated query. The results presented below are the quality (latency in ms) of the updated query network, as presented by the number of operators updated in the largest network.

Results (Figures 32 and 33): Our dynamic programming optimizer produces the best overall latencies for both expected and actual evaluations for the query network. The difference between the expected and the actual is roughly 10.2%, which indicates that our online approach can produce results that are close to the actual values. Notice that the hybrid approach chose to switch to the Greedy optimizer in the last update to the query network. This is due to the system being near capacity when the update occurred (roughly 2,500 tuples/s with a processing rate of roughly 2,615 tuples/s), and in the time to process a new query, the system could have lost data, so the hybrid algorithm chose the greedy optimizer.

Takeaway: Our online optimizer produces higher quality plans when compared to the baseline algorithms.

5.4.3.3 Recovery Time When an update occurs, the system must determine what to update, how to update, and then put the new update in place. This process takes time, and while it is processing, the query will still need to be executing. The time between the start of an update optimization and the normal execution of the resulting plan is the time it takes the system to recover from an update. In this experiment, we evaluate this *recovery time* for updates that are generated due to three different reasons: 1.) more system load, 2.) more selective data, and 3.) access control updates.

Setup: For each experiment, the same randomly generated 128-operator query was used with the same offline optimization. Operations were selected from a random distribution of

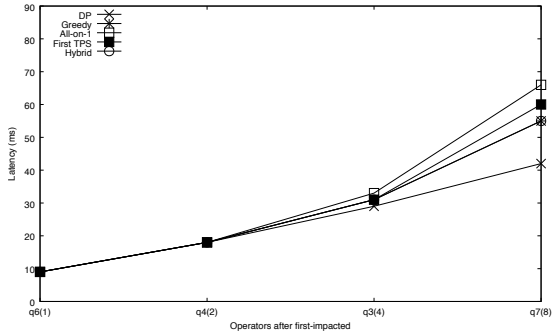


Figure 32: Expected latency in milliseconds for each updated query network size as generated by each optimizer.

operations which included two-way joins, filters, summations, averages, projections, and decrypt-process-encrypt operations as before.

When adjusting selectivities, data was generated so that the overall query's selectivity increased from .5 to .9 by increasing the selectivity of operations after the desired first-impacted. This process was done in a greedy manner, with each subsequent desired update cost building on the previous. All operations processed on plaintext data with a constant input rate.

To adjust the input rate to a specified cost, spouts increased their rates at a specified time by either double, four times, or seven times their base rate which in turn was reflected at the first-impacted. It is somewhat unlikely that the highest of the rate changes will occur in real life, but we used the input rates here for consistency across this evaluation.

Access control updates occur by specifying a specific change in access controls that target a specific operator. These operators are picked so that the update cost remains consistent across the evaluation, and each update causes an increase in latency and a decrease in throughput (i.e., switch from plaintext to encrypted). A query is considered *recovered* once the the latency has normalized back to a steady value that may differ than the beginning value.

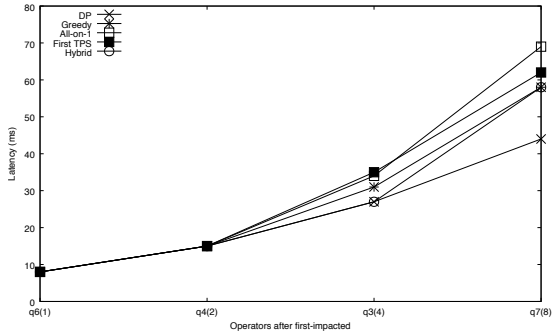
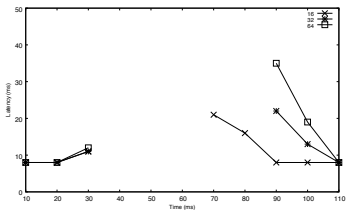
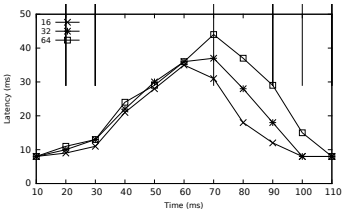


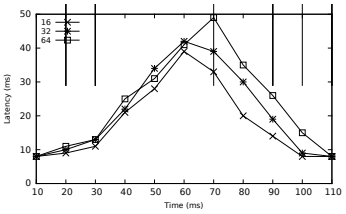
Figure 33: Actual latency in milliseconds for each query network updated size over an execution time of five minutes.



(a) Recovery time caused by an access control update for different costs.



(b) Recovery time caused by a system overload for different costs.



(c) Recovery time caused by more selective data for different costs.

Figure 34: Recovery time for various update costs and causes.

Access Control Results (Figure 34a): When an update occurs to an access control policy (Figure 34a), the data consumer may lose access as indicated by the unreported latency values. Once the query has been resumed, the larger updates cause a large spike in latency that takes more time to recover from, as expected. Note the processing time of each update also increases, but the recovery time is more-or-less the same (20-30ms). This shows that the new queries can handle the increased workload to make-up for the lost work and then maintain a new latency rather quickly.

System Load Results (Figure 34b): When updates occur due to an increase in the

workload (i.e., an increase in the input rate), the system does not stop processing, but instead continues until the new query is in place. Here, inputs increase at the 10ms mark with the system requesting an update when the latency doubles (roughly 20 ms). When each update peaks (maximum latency), the new query is placed and has begun processing. Once the system has processed all the buffered data, the latency normalizes.

Selectivity Results (Figure 34c): When an update occurs due to higher selectivity, the system does not stop processing. Data updates happen at 10ms and take roughly 15 ms to enter the system. Once the latency has doubled, the system requests an update. Updates for selectivity can recover faster than updates based on the system load simply because the amount of data is unchanged, but a single operator or set of operators may have some excess tuples in their buffers. Overall, larger updates take longer to process and require a larger recovery time than updates with a lesser cost. This time is comprised of finding the first-impacted as well as in optimizer execution time and with processing excess tuples.

Takeaway: More costly updates require greater recovery time regardless of the reason for the update.

5.4.4 Comparison to the State-of-the-Art

This section evaluates the quality of the plans produced by Shoal versus other operator placement approaches. We specifically focus on the works of Pietzuch et al. [61] and Srivastava et al. [55]. For these experiments, we evaluate each approach for a number of queries for both the optimized expected latency and the actual latency realized by the query.

Algorithms: Pietzuch et al. [61] propose a solution that focuses on placing operators in a large-scale distributed network using a latency metric. Their optimizer takes a query plan and places it using a two-step algorithm: first a Virtual Operator Placement step and then a Physical Operator Placement step. The virtual operator placement step considers all operators in a query and places them based on a cost space. This cost space consists of a decentralized view of the network from a single node's perspective and focuses on the latencies between potential third-party systems. There is also a load dimension that can ensure that a single third-party system does not become overwhelmed. Their approach allows for online

updates by allowing operators to migrate between third-party systems. To compare to our work, we fix the cost space by artificially creating latencies and data rates between potential third-party systems (i.e., the assumed information gathered by the DDSMS in their work) and then allow it to adjust over time. The main optimization function used in their work is to minimize the following formula:

$$\sum_{l \in L} DR(l) * LAT(l) \tag{5.7}$$

Where l is the link between two nodes, $DR(l)$ is the data rate of that link, and $LAT(l)$ is the latency of that link.

The other approach we focus on is that of Srivastava et al. [55]. Srivastava et al. also reduce data transmission, but do so for localized networks. Their work focuses on using parts of the query itself, as well as the machines available for placement, to make a placement decision. Specifically, they focus on the selectivity of filtering operations, the cost associated with each operation, and the cost associated with sending a tuple through the network. In addition to the above costs, a join's cost is calculated using its selectivity and the cost per unit time for processing one tuple. The cost of a placement plan is therefore the sum of all of the nodes where the selectivities of upstream filters is multiplied by the cost of the current filter. Some filters are correlated and some are not, so the ordering decision comes from the commutative aspect and the overall cost comes from minimizing the cost of the filter and join orderings. To compare with our work, we again assume an artificially created latency and use the same operators' costs and latencies'across all approaches. For more information and for the exact cost model used in this comparison, the reader is highly encouraged to read [55].

Setup: For our comparison, we use multiple queries over a fixed number of third-party systems. We demonstrate each approach over an example query in Section 5.4.5. We use 5 third-party systems, each connected to each other with an initial latency randomly selected from a range of 5-500 ms. Each query is comprised of between 4 and 128 operations selected as either filters (selection operations) or joins, with all data being in plaintext. Since [61] requires an initial query plan, we use Shoal with a single third-party system and unlimited (i.e., more than is needed) capacity to generate a non-distributed query plan. Finally, each

filter is given a selectivity randomly selected from the set {.1,.2,...,.9}. To gather information on actual latencies, each query was executed for a total of five minutes for each approach.

Results (Figures 35& 36): As depicted in Figures 35 and 36 for all cases, Shoal produces better latencies both in the expected and actual cases. As before, the expected and actual are within an average of 8%, however the Pietzuch et al. approach is more predictable since its expected is on average only 4% different from the actual value. Shoal is able to outperform the other approaches because it attempts to find an optimal solution that takes into account the parallelism inherent to a Distributed system by preferring plans that allow work to be done on multiple devices simultaneously. The Pietzuch et al. approach relies on an optimize-then-place approach and missed better filter orderings, which becomes more apparent as queries grow larger. The Srivastava et al. approach does consider ordering, but does not consider the parallelism inherent to a distributed system and would often serialize sets of operations that could have otherwise been done in parallel.

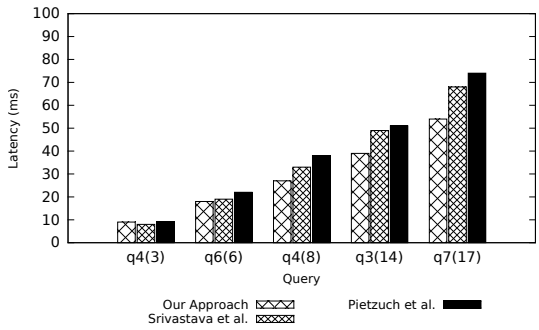
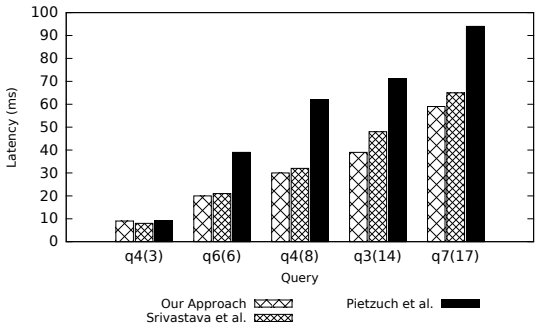


Figure 35: Expected latency for each optimizer for synthetic data on a random query.



8

Figure 36: Actual latency for each optimizer for synthetic data on a random query.

Takeaway: By considering ordering and placement at optimization time, as well as taking advantage of parallelization inherent to the distributed system, Shoal can out-preform other state-of-the-art optimizers in terms of end-to-end latency.

5.4.5 Taxi Cab Data

To evaluate our optimizer on real data, we implement the following query on the 2015 DEBS Grand Challenge taxi dataset [52].

```
SELECT TOP profit, companyName
FROM (SELECT SUM(fare) as profit, companyName
      FROM taxiStream
      EVERY 5m UPDATE 1m
      GROUP BY companyName;)
EVERY 10m UPDATE 10m
GROUP BY companyName
LIMIT 5;
```

The purpose of this query is to get the top grossing taxi cab companies for the last 10 minutes. For this query, we modified the data so that the token id of the taxi driver was associated with a company (a simple modulus). We further altered the data so that the fare data was encrypted with a homomorphic encryption.

We optimize the query through our offline approach and produce the query plan in Figure 37(a). Note that all of the operations were able to fit on one Amazon EC2 t2.xlarge general purpose machine with an average latency of 51ms. When given a much smaller set of machines, specifically six t2.micro machines, the plan produced in Figure 37(b) was produced with a latency of 61ms. When switching to a t2.nano machine, the middle plan was also produced, with a similar 62ms latency. However, during certain times of day for the taxi data (morning, lunch time, and all night) it switched to the plan in Figure 37(c) with a much higher average latency of 99ms.

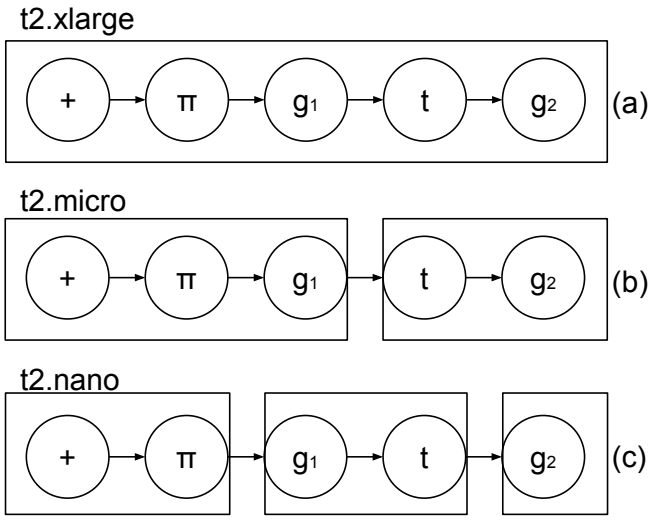


Figure 37: Example simple query with taxi cab data with different sizes of EC2 machines.

We further compare Shoal with the optimizer of Pietzuch et al. [61] and Srivastava et al. [55] for this query. The results are in Table 6 using the t2.macro machines.

Table 6: Expected or actual latencies for each approach.

Approach	Expected Latency (ms)	Actual Latency (ms)
Shoal	**58**	**61**
Pietzuch et al.	62	65
Srivastava et al.	65	70

Given the small nature of this query, there is not much difference between each approach other than ordering, which has a seemingly negligible impact.

5.4.6 Related Work

Table 7: Literature survey for DDSPS optimizers and systems as it relates to heterogeneous characteristics inherent to the DDSPS.

Citation	Hardware		Network			Data			Optimizer	
	Mem	CPU	Bandwidth	Latency	Distributed	ACs	Sel	Rate	Optimal	UDF
Borealis [28]	✓	✓	✓	✓	✓					✓
Cardellini et. al. [62]			✓	✓	✓				Some	
Pietzuch et. al. [61]			✓	✓	✓			✓		
Huang et. al. [54]		✓		✓	✓			✓		
Thoma et. al. [63]			✓							
Backman et. al. [56]				✓	✓					
Chatzistergiou et. al. [57]		✓		✓	Clusters					
Rizou et. al. [58]			✓	✓						
Streamforce [4]						✓				
PolyStream [3]						✓				
Shoal		✓	✓	✓	✓	✓	✓	✓	Some	✓

Operator placement has been studied in the literature for Distributed Database Systems (DDBMSs) and Distributed Data Stream Management Systems (DDSMS). For traditional database applications, the focus for operator placement in distributed database systems usually focuses on replication, sharding, or scalability [64, 65, 66, 67]. The PAQO [68] optimizer focuses on placing operators in a distributed database system so that one entity does not learn the underlying intension of the query.

For data steaming systems, operator placement is of a larger concern since queries are long-running and operators are expected to consume resources for long periods of time while possibly fluctuating in their required resource utilization. The contributions in [62] explore the general problem of operator placement on heterogeneous computational platforms for DDSMs, and propose a linear programming model to place operators. Their approach processes placement in a separate step from optimization, and as shown in Section 5.2, this may lead to suboptimal results.

Huang et al. [54] fit operators onto potential third-party systems by calculating the execution time of an operation and place operations based on the capacity of each third-party system, using end-to-end delay and throughput as the metrics.

Pietzuch et al. [61] explore wide area network implications of operator placement and propose methods which minimize overall network delay. Similarly, Srivastava et al. [55] look into operator placement on different third-party systems with different bandwidths allocated to each third-party system, but assumes third-party systems are closely located.

Thoma et al. [63] place operators in a DDSMS where queriers have the ability to control where operators are placed via a set of constraints. These constraints generally cover all aspects of the placement, but do not consider the access control policies of a data consumer.

Operators placement using heuristics to optimize for end-to-end latency and network traffic have also been explored [56, 57, 58]. Additionally, some related work has focused on network cost models [69] that do not include CPU or access controls.

Finally, some related work has focused on the impact of enforcing access controls in a DDSPS. Enforcement systems such as FENCE [1, 22] include the enforcement overheads in the optimization step by adding streaming operations that can be handled like any other operation, but do so without considering operator placement. Other systems will rewrite queries or alter streaming operators [9, 6, 7, 27], while others focus on protecting a single system, such as Borealis [2]. These systems simply explore the overheads associated with access control enforcement and do not consider them at optimization time or during operator placement. Furthermore, these systems do not explore the tradeoff between different types of access control enforcement during optimization time, which is provided in Shoal. Systems like our own, PolyStream [3], and Streamforce [4], CryptDB [5] consider such tradeoffs, but do either do not operate in a distributed fashion (CryptDB), or do not consider them at optimization time.

Thus far current optimizers and systems have focused on a limited scope of heterogeneous characteristics within a DDSPS. Either they do not consider optimization and placement simultaneously, or they limit their approach to optimize solely for network, hardware, or access control alone. Shoal provides a general cost model and dynamic programming algorithm that accounts for heterogeneity in the networking, hardware, and access control enforcement of a streaming system. Table 7 details the forms of heterogeneity addressed in the literature and in Shoal. Chatzistergiou et. al. [57] does not focus on large scale geographically distant networks, but rather on clusters of shared resources. Both Shoal and Cardellini et. al. [62]

are not guaranteed to be optimal, either because the query is too large (Shoal) or because it makes use of the optimize-then-place approach (Cardellini et. al. [62]).

5.5 SUMMARY

Shoal allows a user to realize the benefits gained in PolyStream and Sanctuary *at optimization time*. Whether it is a permission granted that allows the data consumer to use a deterministic or order preserving encryption to improve their query performance, or hardware available that enables an SGX enclave operator to be used, Shoal can build an efficient query plan. Given the different physical operators for each operation realized through Sanctuary and PolyStream, or any other streaming operator, Shoal allows for better utilization of third-party systems by considering the different physical operators for each optimization time.

Shoal further leads to better performance, as depicted in the experimental evaluation in Section 5.4, by producing the optimal query plan using a dynamic programming optimizer. Further, Shoal is able to implement any operator, allowing for full query expressiveness. Finally, by considering access controls at optimization time, Shoal is able to provide protection at optimization time, helping to reduce the cost of protecting a data provider's data.

6.0 CONCLUSION AND FUTURE WORK

In this dissertation, we addressed the following hypothesis:

Designing new access control enforcement techniques for cloud-based Distributed Data Stream Processing Systems and considering them at optimization time can lead to better protection, utilization, and performance.

To support this hypothesis, we first showed that any satisfactory solution would have to reside in the space where access controls were enforced in-network and plaintext data would not be visible to untrusted, third-party systems. We further show throughout this dissertation that normal, straw-man optimize-then-place approaches are costly in terms of computational overhead (e.g., increased latency, decreased expressiveness, etc.) or are lacking in privacy guarantees. Therefore, for better protection in this space, we had to design systems that could enforce access controls on encrypted data. For utilization, we had to design systems that could effectively make use of third-party systems while processing protected data. And further, every system had to maintain some level of optimal or near optimal performance when compared to a plaintext system and other state-of-the-art systems. There were three main contributions that support our hypothesis:

PolyStream (Chapter 3) illustrated that better protection and utilization can be achieved by combining online key management with computation-enabling encryption. Through the use of four main computation-enabling encryption schemes, queriers could directly execute queries over encrypted data using untrusted third-party infrastructure. Such computation allowed for the querier to better utilize third-party systems when data is being protected by a data provider. Moreover, the use of such encryption schemes does not require any changes to the underlying stream management system for the data consumer.

We showed that data providers can author Attribute-Based Access Controls to protect

different parts of their data stream in varying levels of granularity. By using Attribute Based Encryption (ABE) in concert with Attribute Based Access Controls, data providers can protect data for only those consumers who have proper access as assured by a trusted Attribute Authority. We further showed that using ABE to protect keys for other computation-enabling encryption schemes can improve utilization and performance by allowing data providers to transmit data using CPE rather than ABE. Keys can be shared in an online manner using the security punctuation framework to send access control updates (i.e., new encryption keys) protected with ABE to queriers. Through a thorough evaluation, we see better performance versus the state-of-the-art system Streamforce [4], and often performance nearly as good as a plaintext system.

Sanctuary (Chapter 4) aims at overcoming the limitations in systems like PolyStream to allow for better utilization of the untrusted third-party systems. Prior approaches limit utilization of third-party systems since not all queries are possible. In Sanctuary, we take advantage of Intel's Software Guard Extensions (SGX) to implement *any* streaming operator on an untrusted third-party system. In Sanctuary, we show that full expressiveness of any query language is possible when using an SGX enclave. We further showed that the main limitations of an enclave (processor contention and limited trusted memory) can be mitigated through batching and specialized operators. We introduce several such operators that work in memory-limited environments by writing to a higher level of memory and encrypting the data stored there.

Sanctuary yield the best protection for a data provider as data can be encrypted in the most secure way possible, so long as the key is shared with the querier so that it can be used in the SGX enclave. Sanctuary, therefore, offers full third-party utilization to data consumers, and full protection to data providers. Furthermore, as shown in Section 4.6, the performance of operators executing in an SGX enclave is on par with other plaintext or baseline systems in most cases.

Shoal (Chapter 5) is an optimizer that is designed to consider the heterogeneous nature of streaming systems. Specifically, Shoal is a continuous query optimizer and operator placement algorithm that considers different physical implementations of streaming operators at optimization time to ensure the best performance. Shoal uses dynamic programming to select

operators and third-party systems to build an optimal query network. When constructing the network, Shoal considers the different physical operators possible for each streaming operation, meaning that it can choose a plaintext operator, one that uses CPE (i.e., an operator that PolyStream would use), one that uses an SGX enclave (i.e., a Sanctuary provisioned operator), or other variations of the same operator (e.g., more traditional alternatives like a hash join versus a nested-loop join). By considering all of these alternative operator implementations, Shoal can guarantee the best query plan.

In addition to physical operator implementation, Shoal also considers changes to the system itself, whether in hardware, data volume, data selectivity, or even access control policy changes. Shoal provides an online, real-time re-optimization step that considers the operations that have been affected by changes in the system status. A data provider's protection is ensured since any operation that may leak data would not be considered given the proper access controls. Moreover, the best utilization is also ensured since any operator that executes on a third-party system will be placed there so long as the cost is lowest. Finally, as optimal performance is the goal of any optimizer, we show that Shoal produces query plans that out-preform other continuous query optimizers. We further show that optimization time is on par with similar optimizers, and can be reduced to a greedy approach if the time is too long to recover from.

Shoal, in using PolyStream for efficient query processing on encrypted data, and in using Sanctuary for full utilization of a third-party system, shows that considering access control enforcement techniques *at optimization time* can lead to better performance. Sanctuary can guarantee full utilization of a third-party-system, and full expressiveness of the continuous query language. PolyStream can yield better performance and allow users to share keys in an online manner. Together, these systems support our hypothesis.

6.1 FUTURE WORK

There are many directions for future work derived from this dissertation. Each system can be expanded and improved upon to further discover novel approaches in this space.

6.1.1 Memory Protection

While the operators introduced in Sanctuary are very near to a baseline, ideal system in terms of privacy leakage, they are not perfect. Specifically, some of the stateful algorithms have the potential to leak information about the memory being accessed, regardless of whether or not it is encrypted. This is undesirable and can be improved to further increase protection, for example, using algorithms can make use of Oblivious RAM (ORAM) techniques to hide RAM accesses. ORAM is a technique that transforms memory accesses to randomly read and write to memory, even if it is the same data being accessed. ORAM also allows the user to make calls to memory without alterations.

Each operator introduced in Sanctuary could be augmented to make memory calls through ORAM. While this may ensure better data protection, further investigation would be needed to ensure that performance is not severely impacted. Moreover, the ORAM algorithm may have to run as part of the enclave, which would reduce available memory within the enclave.

6.1.2 Additional Heterogeneity

While Shoal considers many different forms of heterogeneity, there are more that can be considered. For instance, changes in monetary price over time may affect decisions on operator placement. Rather than adding more and more forms of heterogeneity to a pre-built optimizer, a general amendable optimizer can be created. Users can then specify a cost model of their choosing to get results that are important to them. Further, users can pick and choose which underlying system characteristics should be considered at runtime, and potentially add their own. Such an optimizer can have protections built in by reading a list of access controls provided by the querier. Building off of a system such as Sanctuary as a default access control and key management, the optimizer can consider protections as a first-class citizen, with all other sources of heterogeneity being added by the user.

6.1.3 Optimization Constraints

Users may be able to augment their queries with a set of constraints that enable them to give preferential treatment to some third-party providers, or black-list others. Queriers may also want to dictate where results can be sent from one operation to another. In our work [63], we show how such constraints can be formalized and even satisfied through different techniques. These techniques, however, should become part of our dynamic programming approach in Shoal so that they are considered at optimization time, rather than in a post optimization phase.

6.1.4 Protection of non-Relational Streams

This dissertation focused on relational data using a continuous query model. This model, however, is not the only stream-processing paradigm. Map-reduce is a popular stream-processing paradigm wherein users specify a set of mappers and reducers that continually execute on data. Exploring the protection, utilization, and performance of access control enforcement in map-reduce systems would be an interesting challenge, as explored in VC3 [48]. Would a system similar to Shoal be as performant or as near an ideal system in terms of privacy? Could computation-enabling encryption be used for such systems? And what differs when trying to optimize a map-reduce framework? These questions are interesting and could lead to novel research contributions.

6.2 IMPACT OF THIS DISSERTATION

This dissertation's impact can be broken down into two fields: technological and societal.

6.2.1 Technological Impact

This dissertation explores a novel area in which a user wishes to enforce access controls on their data in a distributed stream processing system while still permitting a querier to

query their access-controlled data. We propose and implement three systems comprised of novel approaches and algorithms. PolyStream combines computation-enabling encryption with an online key management system via security punctuations to provide a first-of-a-kind streaming system in which queriers execute directly on encrypted data with changing access controls. Polystream is an improvement over the closest state-of-the-art system for streaming systems by providing more functionality and more performant queries. Polystream was the first system to include an online key management system that allowed for updates to access controls and processing on encrypted data.

Similarly, Sanctuary introduces a system for using Intel's SGX on stream processing systems. Sanctuary implements novel memory-aware stateful streaming operators inside of an SGX enclave that allow for arbitrary streaming operators to be implemented, regardless of their memory requirements. Sanctuary further improves on the closest related work by allowing for full query expressiveness and near ideal baseline protection. Sanctuary is the first system to explore the use of Intel's SGX on a streaming system.

Shoal is a first-of-its-kind streaming optimizer that considers many different forms of heterogeneity inherent to a streaming system at optimization time. Specifically, Shoal considers hardware, data selectivity, data rate, third-party system capacity, and access controls when optimizing and placing relational continuous queries on a network of third-party resources. Shoal introduces an online and offline algorithm for query optimization and will re-optimize when the underlying system characteristics or access controls change over time. The Shoal optimizer is, to the best of our knowledge, the first streaming optimizer to consider access controls at optimization and re-optimization time, let alone the many different forms of heterogeneity. Further, Shoal is shown to produce higher quality query plans versus other state-of-the-art systems.

6.2.2 Societal Impact

This dissertation addresses a problem that will become more prevalent in the coming years. As motivated throughout this dissertation, data is becoming more and more available, with breaches or unintentional sharing happening more frequently. As technology delves deeper

into our personal matters with examples like the Apple watch being a sufficient substitute for pulse and heart monitoring devices [70], or glucose meters sending data to cloud providers to work with your smart phone [71], or even the Owlet infant monitoring socks shipping information about the infant to a cloud server to be processed and viewed anywhere in the world [72], there are greater chances that private or personal data can be consumed by unintended third-party resources. The work presented in this dissertation acts as a preventative measure for any one person to maintain a desired level of protection in a streaming system.

While adoption of these systems would require buy-in from queriers as well as other users, the systems in this dissertation should act as an avenue of exploration for any protection-minded queriers in the future. Additionally, one of the strengths of the work presented in this dissertation is its relative portability given the underlying stream management system. We aspired to make our contributions as agnostic of the streaming system as possible, but rather focus on being additive to existing systems. Other than the requirement of having an SGX-enabled CPU available for Shoal, we have accomplished this with our systems. A traditional relation streaming system can be optimized with Shoal or protected with Sanctuary. This is important for any future adoption as it shouldn't require any major refactoring or re-branding to realize the benefits of any one system. It is our hope that the findings, algorighms, and systems used in this dissertation can be used in future streaming systems to improve a person's sense of protection on their personal data, while also maintaining some level of utilization and performance for the queriers.

I would like to acknowledge that the work in this dissertation was supported in part by NSF (CNS-1228697, CAREER CNS-1253204, CAREER IIS-0746696, OIA-1028162, CNS-1704139).

BIBLIOGRAPHY

[1] R. V. Nehme, H.-S. Lim, and E. Bertino, "Fence: Continuous access control enforcement in dynamic data stream environments," in *Proceedings of the third ACM conference on Data and application security and privacy*, pp. 243–254, 2013.

[2] W. Lindner and J. Meier, "Securing the Borealis data stream engine," in *Database Engineering and Applications Symposium, 2006. IDEAS'06. 10th International*, pp. 137–147, IEEE, 2006.

[3] C. Thoma, A. J. Lee, and A. Labrinidis, "Polystream: Cryptographically enforced access controls for outsourced data stream processing," in *Symposium on Access Control Models and Technologies (SACMAT)*, vol. 21, p. 12, 2016.

[4] D. T. T. Anh and A. Datta, "Streamforce: outsourcing access control enforcement for stream data to the clouds," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, pp. 13–24, 2014.

[5] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 85–100, 2011.

[6] B. Carminati, E. Ferrari, J. Cao, and K. L. Tan, "A framework to enforce access control over data streams," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 3, p. 28, 2010.

[7] B. Carminati, E. Ferrari, and K. L. Tan, "Specifying access control policies on data streams," in *Advances in Databases: Concepts, Systems and Applications*, pp. 410–421, Springer, 2007.

[8] A. Havet, R. Pires, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "Securestreams: A reactive middleware framework for secure data stream processing," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pp. 124–133, ACM, 2017.

[9] B. Carminati, E. Ferrari, and K. L. Tan, "Enforcing access control over data streams," in *Proceedings of the 12th ACM symposium on Access control models and technologies*, pp. 21–30, 2007.

[10] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 15, no. 2, pp. 121–142, 2006.

[11] A. Paverd, A. Martin, and I. Brown, "Modelling and automatically analysing privacy properties for honest-but-curious adversaries," *Tech. Rep.*, 2014.

[12] S. Heron, "Advanced encryption standard (AES)," *Network Security*, vol. 2009, no. 12, pp. 8–12, 2009.

[13] S. Halevi and P. Rogaway, "A tweakable enciphering mode," in *CRYPTO 2003*, pp. 482–499, Springer, 2003.

[14] T. Dierks and C. Allen, "The TLS protocol version 1.0," tech. rep., 1998.

[15] A. Boldyreva, S. Fehr, and A. ONeill, "On notions of security for deterministic encryption, and efficient constructions without random oracles," in *Annual International Cryptology Conference*, pp. 335–359, Springer, 2008.

[16] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill, "Order-preserving symmetric encryption," in *Eurocrypt*, pp. 224–241, Springer, 2009.

[17] A. Boldyreva, N. Chenette, and A. O'Neill, "Order-preserving encryption revisited: Improved security analysis and alternative solutions," in *Advances in Cryptology–CRYPTO 2011*, pp. 578–595, Springer, 2011.

[18] P. Paillier, "Public key cryptosystems based on composite degree residuosity classes," *Advances in Cryptography - EURPCRYPT'99*, vol. 1562, 1999.

[19] V. Costan and S. Devadas, "Intel SGX explained.," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[20] J. Melton, "Database language SQL," in *Handbook on Architectures of Information Systems*, pp. 105–132, Springer, 1998.

[21] X. Jin, R. Krishnan, and R. Sandhu, "A unified attribute-based access control model covering DAC, MAC and RBAC," in *Data and applications security and privacy XXVI*, pp. 41–55, Springer, 2012.

[22] R. Nehme, E. A. Rundensteiner, and E. Bertino, "A security punctuation framework for enforcing access control on streaming data," in *IEEE 24th International Conference on Data Engineering (ICDE)*, pp. 406–415, 2008.

[23] R. Adaikkalavan and T. Perez, "Secure shared continuous query processing," in *ACM SAC*, pp. 1000–1005, 2011.

[24] J. Hur and D. K. Noh, "Attribute-based access control with efficient revocation in data outsourcing systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 7, pp. 1214–1221, 2011.

[25] S. Yu, C. Wang, K. Ren, and W. Lou, "Attribute based data sharing with attribute revocation," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pp. 261–270, ACM, 2010.

[26] S. Jahid, P. Mittal, and N. Borisov, "Easier: Encryption-based access control in social networks with efficient revocation," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 411–415, ACM, 2011.

[27] W. S. Ng, H. Wu, W. Wu, S. Xiang, and K.-L. Tan, "Privacy preservation in streaming data collection," in *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pp. 810–815, 2012.

[28] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, *et al.*, "The design of the borealis stream processing engine.," in *Cidr*, vol. 5, pp. 277–289, 2005.

[29] M. Green, S. Hohenberger, and B. Waters, "Outsourcing the decryption of abe ciphertexts.," in *USENIX Security Symposium*, 2011.

[30] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-based encryption for fine-grained access control of encrypted data," in *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 89–98, 2006.

[31] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich, "Processing analytical queries over encrypted data," in *Proceedings of the 39th international conference on Very Large Data Bases*, pp. 289–300, 2013.

[32] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. of Eurocrypt*, pp. 223–238, 1999.

[33] B. Wang, M. Li, S. S. Chow, and H. Li, "A tale of two clouds: Computing on data encrypted under multiple keys," in *Communications and Network Security (CNS), 2014 IEEE Conference on*, pp. 337–345, IEEE, 2014.

[34] StormProject, "Storm: Distributed and fault-tolerant realtime computation." http://storm.incubator.apache.org/documentation/Home.html, 2014.

[35] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[36] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM DEBS*, pp. 207–218, ACM, 2013.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, p. 10, 2010.

[38] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 239–250, ACM, 2015.

[39] J. Benthencourt, A. Sahai, and B. Waters, "Advanced crypto software collection: Ciphertext-policy attribute-based encryption," 2011.

[40] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 480–491, VLDB Endowment, 2004.

[41] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," in *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pp. 309–320, 2003.

[42] C. Thoma, A. J. Lee, and A. Labrinidis, "Behind enemy lines: Exploring trusted data stream processing on untrusted systems," in *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, pp. 243–254, ACM, 2019.

[43] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[44] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX) Security 18)*, pp. 991–1008, 2018.

[45] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Proceedings of the 16th Annual Middleware Conference (Middleware)*, 2016.

[46] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions.," in *HASP@ ISCA*, p. 11, 2013.

[47] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OKeeffe, M. L. Stillwell, *et al.*, "SCONE: Secure linux containers with Intel SGX," in *12th USENIX Symp. Operating Systems Design and Implementation*, 2016.

[48] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *2015 IEEE Symposium on Security and Privacy*, pp. 38–54, IEEE, 2015.

[49] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 283–298, 2017.

[50] C. Priebe, K. Vaswani, and M. Costa, "EnclaveDB: A secure database using SGX," in *EnclaveDB: A Secure Database using SGX*, p. 0, IEEE, 2018.

[51] F. Shaon, M. Kantarcioglu, *et al.*, "Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors," in *SIGSAC*, pp. 1211–1228, ACM, 2017.

[52] D. G. Challenge, "Debs grand challenge." http://dl.acm.org/citation.cfm?id=2772598, 2014.

[53] C. Thoma, A. J. Lee, and A. Labrinidis, "Shoal: Query optimization and operator placement for access controlled stream processing systems," in *IFIP Annual Conference on Data and Applications Security and Privacy*, pp. 261–280, Springer, 2019.

[54] Y. Huang, Z. Luan, R. He, and D. Qian, "Operator placement with QoS constraints for distributed stream processing," in *2011 7th International Conference on Network and Service Management*, pp. 1–7, IEEE, 2011.

[55] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 250–258, ACM, 2005.

[56] N. Backman, R. Fonseca, and U. Çetintemel, "Managing parallelism for stream processing in the cloud," in *HOTCDP*, pp. 1–5, ACM, 2012.

[57] A. Chatzistergiou and S. D. Viglas, "Fast heuristics for near-optimal task allocation in data stream processing over clusters," in *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pp. 1579–1588, ACM, 2014.

[58] S. Rizou, F. Durr, and K. Rothermel, "Solving the multi-operator placement problem in large-scale operator networks," in *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pp. 1–6, IEEE, 2010.

[59] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," in *Data Stream Management*, pp. 317–336, Springer, 2016.

[60] A. Lew and H. Mauch, *Dynamic programming: A computational tool*, vol. 38. Springer, 2006.

[61] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *22nd International Conference on Data Engineering (ICDE'06)*, pp. 49–49, IEEE, 2006.

[62] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 69–80, ACM, 2016.

[63] C. Thoma, A. Labrinidis, and A. J. Lee, "Automated operator placement in distributed data stream management systems subject to user constraints," in *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*, pp. 310–316, IEEE, 2014.

[64] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et al.*, "Spanner: Googles globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[65] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, *et al.*, "F1: A distributed SQL database that scales," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1068–1079, 2013.

[66] C. Curino *et al.*, "Relational cloud: A database-as-a-service for the cloud," *CIDR*, 2011.

[67] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[68] N. Farnan, A. Lee, P. Chrysanthis, and T. Yu, "PAQO: Preference-aware query optimization for decentralized database systems," in *ICDE*, 2014.

[69] L. Golab and M. T. Özsu, "Issues in data stream management," *ACM Sigmod Record*, vol. 32, no. 2, pp. 5–14, 2003.

[70] A. Capritto, "The FDA just cleared an iPhone ECG sensor that beats the apple watch," Apr 2019.

[71] "Diabetes management: Glucose monitors that connect to your smart phone," Jun 2018.

[72] I. Thomson, "Wi-Fi baby heart monitor may have the worst iot security of 2016," Oct 2016.