ABSTRACT
We describe PECCARY, a system that allows confidentiality-preserving computation of big data queries in public cloud infrastructures. It does so by encrypting all sensitive data residing in the cloud, and employing practical partially homomorphic encryption (PHE) schemes to perform operations securely.

In contrast to previous approaches, PECCARY achieves both practical performance and expressiveness by abandoning full transparency. That is, PECCARY proposes the abstraction of secure data types and corresponding annotations for programmers to conveniently denote constraints relevant to security. Performance is achieved by PECCARY’s compiler applying several novel compilation techniques leveraging these abstractions and intrinsic knowledge of crypto systems, as well as a re-encryption service running in the client domain for dealing with limitations of PHE, if any, remaining after applying our compilation techniques.

We have implemented a prototype of PECCARY on top of Apache Pig which ultimately translates to a set of MapReduce jobs that can leverage the parallelism of the underlying cloud infrastructure. PECCARY can securely execute all queries in standard benchmarks such as TPC-H and PigMix2 with an average overhead of 2.10x and 3x respectively compared to an un-encrypted solution which reveals all queries in standard benchmarks such as TPC-H and PigMix2 with an average overhead of 2.10x and 3x respectively compared to an un-encrypted solution which reveals all data.

1. INTRODUCTION
Over the past few years, compute clouds have emerged as cost-efficient big data analysis platforms for corporations and governments. Yet, many organizations still decline to widely adopt cloud services due to: (i) severe confidentiality and privacy concerns (e.g., [7, 27]); and (ii) explicit regulations in certain sectors (e.g., healthcare and finance [5]).

One promising avenue to address the above issues is to use homomorphic encryption: data is stored in encrypted form in the cloud, and computations are performed on such ciphertexts. Over the past few years, many researchers have vigorously investigated fully homomorphic encryption (FHE) schemes [19]. These support arbitrary computations on ciphertexts, but still incur around $10^6 \times$ slowdowns [20] compared to executing computations on plaintext, thus nullifying the benefits of computing in a cloud. As a consequence they have not gained a great momentum for secure cloud computing yet.

While several systems with practical performance have been proposed based on partially homomorphic encryption (PHE), these systems are not specifically targeted at typical “data flow”-style big data analytics jobs (cf. FlumeJava [12], PigLatin [24], Dryad [23]), and thus are limited in terms of expressiveness in that many relevant queries cannot be computed in the cloud (e.g., [21, 26, 33, 32, 31, 28]).

The fundamental problem stems from the fact that unlike FHE, PHE schemes by definition do not support arbitrary computations, and applying an operation to two operands requires both operands to be encrypted under the same scheme. For instance, to compute $(x + y) \times z$ on ciphertexts, $x$ and $y$ need to be encrypted under additive homomorphic encryption; this would carry over to the result of their addition, rendering a subsequent multiplication by $z$ impossible as that requires both its operands to be available under multiplicative homomorphic encryption. Thus the big challenge is for a system to perform operations on “unencrypted” encryption schemes. There are three ways this challenge can be tackled. (a) No support: a system can reject queries that hit limitations such as the above untreated (e.g., [31]). (b) Client-side completion: in this approach, and as soon as data is not available under the required scheme, the computation returns back to the client (or trusted servers considered as part of the client’s domain) with intermediate data and continues there until completion (e.g., [21, 26, 33, 28]). (c) Client-side re-encryption: in this approach, the encrypted data is sent to the client, gets re-encrypted under the desired scheme, and is sent back to the cloud to continue computation (e.g., [31]). Clearly, the first two approaches limit expressiveness, in the sense that many queries can not be executed in the cloud at all or only to a limited degree, which in turn hampers performance. Furthermore, both approaches — in addition to client-side re-encryption if done straightforwardly — hamper performance with typ-
ical big data workloads if client resources are limited, which is often the motivation for relying on cloud resources in the first place.

It seems thus that the two desirable properties of performance and expressiveness are at odds with each other in the context of cloud-based big data analytics. However, we observe that the existing approaches suffer from attempting to be fully transparent to users. In the case of FHE, the goal is for arbitrary programs to be executable on ciphertexts at the flip of a switch. Similarly, in the case of PHE, existing systems promote unmodified query languages and computational models (e.g., MapReduce [32], SQL [21, 26], PigLatin [30]), and their runtime systems execute queries close to line-by-line and treat individual encryption texts at the flip of a switch. Similarly, in the case of PHE, the goal is for arbitrary programs to be executable on ciphertexts.

In this paper, we introduce PECCARY, a PHE-based system which performs analytical queries over encrypted big data in public clouds, achieving both practical performance and expressiveness. To this end, PECCARY employs specifically tailored programming abstractions along with a smart compiler and runtime planner engine. More specifically, PECCARY makes the following contributions:

**Programming abstractions:** PECCARY introduces secure (abstract) data types (SDTs) to express data flow style big data computations. SDTs are based on well-known data types but capture simple but fine-grained information relevant to security and encryption such as sensitivity levels and ranges for data types.

**Compiler:** Our compiler leverages the above data types to accelerate queries substantially. For instance, the PECCARY compiler incorporates a set of compilation techniques aiming to reduce the times and extent of client (or trusted) side involvement, for instance by reducing the number of required re-encryptions. In addition, it exploits intrinsic properties of crypto systems such as secondary homomorphic properties. Several of our query rewriting techniques go against traditional techniques in order to address the specific constraints of PHE.

**Planner engine:** based on the observation that client-side query completion (b) really can be seen as a special case of re-encryption (c), our planner engine automatically decides when to return to the client, or back to the cloud to achieve the best performance.

The rest of this paper is structured as follows. Section 2 provides an overview of PECCARY and background information. Section 3 introduces SDTs. Section 4 presents our novel compilation techniques. Section 5 describes our heuristics for re-encryption. Section 6 discusses the implementation of PECCARY. Section 7 empirically evaluates PECCARY. Section 8 contrasts PECCARY with related work. Section 9 concludes with final remarks.

## 2. OVERVIEW

In this section we give an overview of PECCARY. We start by explaining the cryptographic primitives that we use to perform computations over encrypted data. We then present the threat model and security guarantees that PECCARY provides. Finally, we explain the high level design of our system.

### 2.1 Cryptographic Primitives

<table>
<thead>
<tr>
<th>Crypto system</th>
<th>Property</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-RND</td>
<td>RND</td>
<td>–</td>
</tr>
<tr>
<td>FNR [14], AES-DET</td>
<td>DET</td>
<td>=, GROUP, JOIN</td>
</tr>
<tr>
<td>Boldyreva et al. [10]</td>
<td>OPE</td>
<td>&lt;, &gt;, ORDER, MIN</td>
</tr>
<tr>
<td>Song et al. [29]</td>
<td>SRCH</td>
<td>MATCHES pattern</td>
</tr>
<tr>
<td>Paillier [25]</td>
<td>AHE</td>
<td>+, SUM</td>
</tr>
<tr>
<td>EIGamal [17]</td>
<td>MHE</td>
<td>×</td>
</tr>
</tbody>
</table>

Each of the crypto systems used by PECCARY to achieve confidentiality inside an untrusted cloud allows computations over encrypted data with respect to some operations. We summarize the set of crypto systems used by PECCARY per homomorphic property — RND: random, DET: deterministic, OPE: order-preserving encryption, SRCH: secure search, AHE: additive homomorphic encryption, MHE: multiplicative homomorphic encryption — and the corresponding operations they support in Table 1 and give some additional details leveraged by PECCARY below.

**Secondary secure operation.** The operation supported by each crypto system as shown in [Table 1] requires that its operands are both encrypted under the same crypto system. In addition to this, some crypto systems support a secondary operation as long as one of the operands is available in plaintext form (i.e., it holds no sensitive information). For example, the Paillier crypto system [25] is an AHE crypto system which means it supports addition between two encrypted values (i.e., there exists some known operation ⊕ s.t. \( x + y = \text{Dec} (\text{Enc}(x) \otimes \text{Enc}(y)) \)). Furthermore, if one of the two operands, say \( y \) is in plaintext form, Paillier can also perform multiplication between the two operands (i.e., there exists some known operation ∗ s.t. \( x \times y = \text{Dec} (\text{Enc}(x) \otimes \text{Enc}(y)) \)). Similarly, the EIGamal crypto system [17] supports multiplication between two encrypted operands and exponentiation between an encrypted and a plaintext operand.

**Secure search constructions.** PECCARY uses Song et al.’s crypto system [29] to perform word search over encrypted text. We implemented this crypto system to accept two boolean parameters. The first parameter (\( \text{NoDup} \)) indicates that the resulting ciphertext does not contain duplicate words and the second parameter (\( \text{Rand} \)) indicates that the order of the words is randomized in the ciphertext. These parameters lead to three different search constructions as shown in [Table 2]. The most secure construction is A which removes duplicate words and randomizes the order of the remaining words before encrypting. This construction offers near random security (leaks only the number of encrypted words within the text) but only allows to check whether a word exists (\( \text{matchExists} \)) in the encrypted text. Search construction B randomizes the order of the words but does not remove duplicate words which reveals the number of instances of each word within the encrypted text and thus enables the use of \( \text{matchCount} \). Lastly, construction C neither removes duplicate words nor randomizes their order, allowing matching on any pattern (\( \text{matchPattern} \)) including any number of words and involving the regex operators ".*".
Table 2: Search constructions. NoDup: ciphertext does not contain duplicate words. Rand: word order is randomized

<table>
<thead>
<tr>
<th>Construction</th>
<th>Parameters</th>
<th>match</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NoDup Rand Exists Count Pattern</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>T T ✓ ✓ X</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>F T ✓ ✓ X</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>F F ✓ ✓ ✓</td>
<td></td>
</tr>
</tbody>
</table>

(any text) and “|” (or). PECCARY automatically selects the most secure search construction that can satisfy the required computation.

2.2 Threat Model

PECCARY provides strong data confidentiality guarantees against a powerful adversarial model, assumed to have full access to machines in the cloud. The adversary can have root access to cloud servers, view the data and the query code uploaded and observe the entire query execution. PECCARY’s goal is to preserve confidentiality, but not integrity or availability. Consequently, the adversary is assumed to be passive (honest-but-curious) and cannot make changes in the queries, results or data stored in the cloud.

2.3 System Design

Figure 1 shows the high level architecture of PECCARY. PECCARY ensures the confidentiality of computations of submitted queries by transforming them into semantically equivalent queries that operate over encrypted data. When a user submits a query, the compiler transforms it into a remote query and a local query. The remote query which operates on encrypted data is deployed on the untrusted cloud. The cloud runs an unmodified Apache Pig service [18] and uses cryptographic user defined functions (UDFs) to perform secure operations. The local query then decrypts the results of the remote query and performs any remaining computations on plaintext data.

Table 3: Overview of techniques

<table>
<thead>
<tr>
<th>Technique category</th>
<th>G1</th>
<th>G2</th>
<th>G3</th>
<th>G4</th>
<th>G5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression rewriting</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selective encryption</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subexpression elimination</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Efficient encryption</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Caching &amp; speculative re-encryption</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

overheads — PECCARY’s compiler uses a set of compilation techniques. Table 3 summarizes how these techniques contribute to our high level goals, namely by (G1) enabling queries not previously executable in a public cloud without sacrificing confidentiality, (G2) reducing the amount of re-encryption, (G3) accelerating re-encryption, (G4) reducing the amount of computation, and (G5) reducing the amount of encrypted data. Below, we give an intuition of the categories of techniques, and in later sections describe how they are used in PECCARY.

Expression rewriting. PECCARY rewrites expressions in queries in encryption-sensitive ways that reduce execution latency. For example, if it is known that variables $x$ and $y$ are positive integers, we can rewrite $x + y > 0$ to $x > 0 \land y > 0$. The changed expression has the potential to eliminate expensive additive homomorphic encryption for $x$ and $y$ and the need for re-encryption.

Selective encryption. PECCARY allows fields that do not contain sensitive information to exist in plaintext in the cloud, thus supporting more homomorphisms.

Subexpression elimination. Since PECCARY executes expressions over encrypted data, the cost of recomputing common subexpressions are compounded. We eliminate such subexpressions in our compilation phase.

Efficient encryption. PECCARY reduces the amount and size of encrypted data by identifying situations where one field is involved in multiple operations that can be supported by the same crypto system or packing together multiple values into a single encrypted value.

Caching and speculative re-encryption. PECCARY also caches and pre-computes most frequently used encrypted values to reduce the cost of re-encryption.

3. Secure Data Types

PECCARY utilizes a set of compilation techniques to improve the expressiveness and performance of queries executed on encrypted data. A cornerstone of these techniques consists in PECCARY’s ability to define input data in ways not typically allowed by conventional query languages.

In particular, PECCARY allows the definition of: (i) sensitivity levels, (ii) ranges for data values, (iii) enumerations, and (iv) composition of data types. These are captured by our secure data types (SDTs) and leveraged by the PECCARY compiler to substantially optimize analytical queries. Table 4 cross-references our SDTs with the compilation techniques that leverage them.

3.1 Sensitivity Level

Different crypto systems offer different security guarantees. PECCARY captures this difference by categorizing crypto
Positive or negative integer values

Table 4: Secure data type usage in compilation techniques

<table>
<thead>
<tr>
<th>Secure data type</th>
<th>Sensitivity level</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data range and precision</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Enumerated type</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Composite type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Sensitivity levels. DET* denotes deterministic encryption on fields with unique values

<table>
<thead>
<tr>
<th>Level</th>
<th>Encryption scheme</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>HIGH</td>
<td>RND, AHE, MHE, SRCH, DET*</td>
<td>uid:long&lt;HIGH&gt;</td>
</tr>
<tr>
<td>LOW</td>
<td>DET, OPE</td>
<td>uid:long&lt;LOW&gt;</td>
</tr>
<tr>
<td>NONE</td>
<td></td>
<td>uid:long&lt;NONE&gt;</td>
</tr>
</tbody>
</table>

Table 6: Data range specification

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[+</td>
<td>-]</td>
</tr>
<tr>
<td>[range(&lt;numA&gt;-&lt;numB&gt;)]</td>
<td>Values between &lt;numA&gt; and &lt;numB&gt;</td>
</tr>
<tr>
<td>[&lt;num&gt;]</td>
<td>&lt;num&gt; decimal point(s) precision</td>
</tr>
<tr>
<td>[unique]</td>
<td>No duplicate values</td>
</tr>
<tr>
<td>[delimiter(&lt;char&gt;)]</td>
<td>Tokens separated by &lt;char&gt;</td>
</tr>
</tbody>
</table>

systems under the categories HIGH and LOW. Category HIGH captures crypto systems that are probabilistic or have no leakage except for the length of the plaintext. We also allow deterministic schemes to be under this category, but only in situations where the values of a field to be encrypted are unique and hence deterministic does not reveal duplicate entries. Category LOW captures encryption schemes such as DET and OPE which reveal duplicates or order. PECCARY allows each field of a table to be optionally annotated with the sensitivity level HIGH, LOW or NONE as shown in Table 5.

The HIGH level of sensitivity is intended for fields that hold highly sensitive values. These fields are only allowed to exist in the untrusted cloud encrypted under a crypto system that belongs in category HIGH. The LOW sensitivity level is intended for fields that are less sensitive such as fields with high entropy like timestamps. These fields can be encrypted under crypto systems of the LOW category. Finally, a field can be annotated with NONE to indicate that it is a non-sensitive field and can hence reside in the cloud in its plaintext form. This distinction of sensitivity level allows PECCARY to be more expressive and achieve better performance without compromising security by enabling the selective encryption and efficient encryption techniques described in Section 4.

We note that the programmer does not have to specify a sensitivity level for each field. If no sensitivity level is explicitly given, the default behavior is to use an appropriate crypto system that supports the operation the field is involved in. In practice, we found that it is sufficient to specify: (i) the fields with high sensitivity so that PECCARY never allows those fields to exist under a crypto system that does not offer high security guarantees, and (ii) the fields that do not hold sensitive information, allowing PECCARY to leave those fields in plaintext which can lead to generating queries with improved expressiveness and performance.

3.2 Data Range and Precision

Apart from the sensitivity level, programmers can optionally provide information about data ranges and precision of types, helping PECCARY generate more expressive and efficient queries. For example, information about the range of values of a field allows us to pre-determine the maximum number of groupings possible if that field is used in a GROUP operation, or the maximum and minimum values if two fields with known ranges are to be added together.

Table 6 summarizes this information and shows examples of its use. For numerical types, the programmer can specify the sign of the values within a field. For example, int[+] declares a field that holds only positive integers and similarly int[-] declares a field that holds only negative integers. For more constrained numerical values the programmer can specify the range of values. For instance int[range(100-200)] indicates a field that holds integer values within the range 100 to 200. For floating point numbers, the programmer can specify the number of decimal points that need to be preserved when encrypting values. For example double[2] indicates that 2 decimal points need to be preserved. If the number of decimal points to preserve is not specified, PECCARY truncates all decimal points of the floating number making it a whole number. This is necessary since crypto system used for arithmetic operations do not support floating point numbers by default. PECCARY uses the precision information to multiply the values of the field appropriately, truncating the remaining decimal points, before encrypting the values.

Apart from numerical types, the programmer can declare any data type as being unique to indicate that the field does not contain duplicates (e.g. chararray[unique]) which allows PECCARY to encrypt that field under a DET crypto system preserving high security guarantees. Lastly, for fields that hold string values the programmer can specify a delimiter that divides individual words, which is particularly important when encrypting under a SRCH crypto system, since the latter must tokenize the string to individual words before encrypting.

3.3 Enumerated Types

Fields containing a small and fixed set of values can be represented as an enumerated type in PECCARY. This is done by declaring a field as an enum and listing the possible values of that enum. For example, enum {EUROPE, AMERICA, ASIA, AFRICA} declares a field as an enum that can take one of the four values. PECCARY parses the given enum type into a key-value map, by assigning a unique key to each of the given values, and stores the map with an associated label that can be used internally to refer to this map. The enum enables PECCARY to use the efficient encryption technique to reduce data size overhead that leads to substantial reduction in the numbers of re-encryptions required in a query.

3.4 Composite Types

Oftentimes computation is applied only to “subparts” of a value, e.g., a query may extract only the month of a date
type value and use it in subsequent computations. Usually, crypto systems do not allow performing such operations on subparts of an encrypted value, which can limit the query expressiveness. To allow such computations to be performed, **Peccary** introduces composite types. Composite types are specified according to the following BNF syntax:

```
c := composite{ t s t }
t := (num : α) | (num : α) s t
s := - | / | , | ...
num := 1 | 2 | 3 | ...
```

where α refers to a type such as int, long, etc. with optional annotation as shown in Table 9. These types specify that values of a field are composed of one or more annotation types, allowing **Peccary** to reason about individual parts. For example, a string of the format YYYY-MM-DD representing a date, where YYYY is a 4 digit number for the year and MM for the month, can be declared in **Peccary** as `:composite[(4:int[*])-2:int[range(1-12)]]`. By doing so, **Peccary** can split the input into its parts and encrypt them individually, which allows for more expressive queries to be generated. Similar to Data range, this information is used to rewrite expressions in queries.

### 3.5 Example

Listing 1 shows the table definition for a subset of fields from the Linelen table used in TPC-H benchmark, defined using **Peccary** secure data types. Keywords in red are specific to **Peccary**, while blue represents standard primitive types and keywords inherited from the SQL language supported by **Peccary**, presented in Section 4. Line 1 declares field `orderkey` that should support addition, and line 2 declares field `linenumber` specifying that each value will be unique within the field. Line 4 declares tax to be of type `double` with two digits of precision, and specifies that this tax rate does not need to be encrypted. Line 5 declares `shipdate` field as a `composite` type in the format YYYY-MM-DD. Line 6 declares the field to be of `enum` type with four possible values `IN_PERSON`, `COLLECT_COD`, `RETURN`, `NONE`. Finally line 7 declares `comment` to be of type `chararray` or string.

### 4. COMPILATION

When a user submits a query for execution, **Peccary** compiles it into a transformed query that operates over encrypted data. In this section, we first give a brief overview of Pig Latin, a simple and widespread data analysis language that the **Peccary** compiler targets. Then we describe the set of compilation techniques that **Peccary** uses to support more expressive and efficient queries.

---

1. `DEFINE Lineitem AS {`;
2. `orderkey : long[*],`;
3. `linenumber : long[*], unique,`;
5. `shipdate : composite[(4:int[*])-2:int[range(1-12)]],`;
6. `shipinstruct : enum(IN_PERSON, COLLECT_COD, RETURN, NONE),`;
7. `comment : chararray);`

Listing 1: **Peccary** table definition with secure data types.

---

### 4.1 Background: Pig Latin

Apache Pig [2] is a data analysis platform which includes the Pig runtime system for the high-level data flow language Pig Latin [18]. Pig Latin expresses data analysis jobs as sequences of data transformations, and is compiled to MapReduce [15] tasks by Pig which are then executed by Hadoop [1]. Pig allows data analysts to query big data without the complexity of writing MapReduce programs. The wide adoption of Pig Latin [18] prompted us to select it as the basis for the data flow language of **Peccary**. We give a short overview of Pig Latin here and refer the reader to [13] for more details.

#### Data types and statements

Pig Latin includes simple types (e.g., `int`, `long`), and complex types (e.g., `bag`, `tuple`, `map`). Furthermore, a `field` is a data item which can be a `bag`, `tuple`, or `map`. Pig Latin statements work with `relations`; a relation is simply a (outermost) `bag` of `tuples`. Relations are referred to by program variables (aliases). Pig Latin supports assignments to variables.

#### Operators and expressions

Relations are created by loading an input file or by applying relational operators to other relations. Examples of relational operators are `JOIN`, `GROUP..BY`, and `FOREACH..GENERATE`. Operators in Pig Latin can also include arithmetic operators (e.g., `+`, `-`, `\*`, `*`), comparisons, casts, as well as `STORE` and `LOAD` operators.

#### Functions

Functions in Pig Latin include built-in functions (e.g., `ABS`, `COS`, `AVG`) and user defined functions (UDFs) if needed. For the purpose of this work, we assume that **Peccary** is aware of the encryption type required for the correct operation of all functions that are part of a Pig Latin script. We enforce this by pre-registering a set of functions with **Peccary**.

**Example.** Listing 2 shows a sample query written in **Peccary** that extends Pig Latin syntax. Line 1 loads the input table `table1` into the program variable input. The program then filters out all lines greater than 10 (line 2) and then groups the filtered input based on field `x` (line 3). Line 4 finds the `SUM`, `COUNT` and `AVG` for each group and saves it into the output file `outfile`.

---

1. `input = LOAD 'table1' AS (x, y, z);`;
2. `inputfiltered = FILTER input BY (x <= 10);`;
3. `inputclassgroup = GROUP filtered BY x;`;
4. `output = FOREACH inputclassgroup GENERATE
SUM(y), COUNT(y), AVG(z);`;
5. `STORE output INTO 'outfile';`

Listing 2: **Peccary** program to find sum count and average in a group.

#### 4.2 Expression Rewriting

The first compilation technique that **Peccary** employs is to rewrite expressions in queries into simpler but semantically equivalent expressions. Being PHE-centered, our rewriting

---

1. According to IBM [16], “Yahoo estimates that between 40% and 60% of its Hadoop workloads are generated from Pig [...] scripts. With 100,000 CPUs at Yahoo and roughly 50% running Hadoop, that’s a lot[...].”
techniques are however different from standard compiler optimizations, and in the light of re-encryption may even seem counter-intuitive. The benefits of rewriting expressions is many-fold. Firstly, PECCARY makes use of the Secure Data Types information on composite types to replace expressions that cannot be executed in the cloud over encrypted data hence improving expressiveness of queries. As an example, the expression $\text{SUBSTRING(\text{shipdate}, 0, 4) == '1994'}$ which compares the first four digits of a date to a constant cannot be performed in the cloud since there is no crypto system that supports substring queries. PECCARY replaces the substring operation with the encrypted field that contains only the year of the date. Secondly, PECCARY uses information it has on the range of values to generate more efficient expressions. For instance, knowing that both $x$ and $y$ are non-negative numbers allows PECCARY to replace the expression $x+y>100$ with the expression $x>50 	ext{ AND } y>50$. Similarly if $x$ has range 50-200 and $y$ has range 60-100, the expression $x+y>100$ can be removed entirely since it always evaluates to true. Lastly, PECCARY can rewrite expressions in a way that reduces the number of re-encryptions the query includes. For example, expression $(x+y)*2$ can be replaced, unconventionally, by the equivalent $x*2 + y*2$. Even though the new expression is semantically equivalent it can lead to fewer re-encryptions as shown in Listing 6 and Listing 7. In Listing 7 both $x$ and $y$ need to be re-encrypted into an AHE scheme before they can be added together since they are initially encrypted under an MHE scheme (to perform the previous multiplication). Once they are added together, the result must also be re-encrypted back to an MHE scheme so that the subsequent multiplication can be performed. This gives a total of three re-encryptions. After the expression is replaced as shown in Listing 8 only two re-encryptions are required to convert the result of $x*2$ and $y*2$ into AHE so that the two can be added together.

4.3 Selective Encryption

The second compilation technique PECCARY employs uses the sensitivity level of each field along with the operations that the field is involved in to infer what encryption schemes it should be encrypted under. If a field is marked as non-sensitive (NONE), PECCARY can allow that field to exist in the cloud in its plaintext form. This allows for the secondary secure operation (see subsection 2.1) to become applicable which provides increased expressiveness and efficiency in the resulting queries. For example, consider the expression $(a+b)*c$. Since AHE and MHE are incompatible with each other, after the addition is performed on encrypted data, the result must be re-encrypted under MHE before the multiplication can be performed. Instead, if field $c$ is a non-sensitive field and it is available in the cloud as plaintext, the multiplication can be directly performed using Paillier’s secondary secure operation. We demonstrate this example in Listings 6 and 7 where we show part of the resulting transformed query with and without selective encryption applied. Line 2 of Listing 6 uses the ENCA dd UDF to perform addition on encrypted data. Here a_ahe, b_ahe and the resulting sum_ahe indicate that fields a, b, and sum are encrypted under an AHE scheme. Similarly in line 3 we use the REENCRYPT UDF to re-encrypt sum_ahe from the AHE scheme to the MHE scheme (sum_ahe). Finally, at line 4 we use the ENCMULT UDF to perform the multiplication. Note that field c is loaded as a plaintext (c_plain) in Listing 7. This allows us to perform a multiplication even though sum is encrypted under an AHE scheme, by using the ENCMULT_PLAIN UDF which uses the secondary secure operation of the AHE scheme to perform multiplication with the plaintext value.

4.4 Subexpression Elimination

Computing over encrypted data is more expensive than working with plaintext data. Hence it is critical that any unnecessary or repeated computation is avoided. PECCARY performs common subexpression elimination within each relation to identify and eliminate unneeded subexpressions. For example, Listing 7 shows part of query Q01 of the TPC-H benchmark which contains the expression $\text{price}*(1-\text{discount})$ followed by the longer expression $\text{price}*(1-\text{discount})+(1+\text{tax})$. To avoid unnecessary computations, PECCARY extracts the first of the two expressions and refers to it instead of recomputing it, as shown in Listing 8. In the latter code snippet, line 2 introduces a new relation that is used to compute the common subexpression and stores it in a field named disc_price. The field is then referenced twice in relation B in line 4 of the listing. In addition to subexpressions, PECCARY removes unneeded computation that can occur across different aggregation functions. Specifically, query Q01 of TPC-H computes all three of the aggregation functions AVG, SUM, and COUNT on multiple different fields. Since AVG can be calculated by SUM/COUNT we can remove the AVG.

4.5 Efficient Encryption

One of the sources of overhead when computing over encrypted data stems from the increased data size overhead. To mitigate that, PECCARY employs a set of techniques that aim to reduce the size of encrypted data. Firstly, PECCARY identifies situations where one field is involved in multiple operations. To accommodate this need, PECCARY will have to encrypt the field under multiple crypto systems. When doing so, PECCARY minimizes the number of crypto systems a field is encrypted under by recognizing that some crypto systems can accommodate multiple operations. For example, the order preserving encryption (OPE) scheme that we use is also deterministic, which means that a field that is involved in both equality and order operations needs only be encrypted under the OPE scheme and can avoid having a separate deterministic encryption in the cloud. There are other crypto systems that support multiple operations (albeit not currently used by PECCARY) such as RSA which, unpadded, supports both multiplications and equality comparisons, or the Boneh-Dau-Goh crypto system which allows multiple additions followed by a single multiplication.
Another way PECCARY reduces encrypted data overhead is through the use of enumerated types described in Section 3.3. In situations where an enumerated type is involved in deterministic or ordering operations, instead of encrypting the input value itself which could lead to a longer ciphertext, PECCARY assigns a secret integer key to it and keeps the mapping of the value-key pairs locally so that when decrypting results the original value can be retrieved. If the relative order of the values is not needed (i.e., when the field is only involved in equality operations) the key sequence is randomized to avoid giving out order information.

PECCARY uses FNR [14] which is a format preserving encryption which allows PECCARY to encrypt data under a DET scheme with no size overhead, as long as the data fits to its 128 block size. For longer values, we use AES in CMC mode as used in CryptDB [26]. Lastly, PECCARY reduces the size of data encrypted under AHE and MHE schemes by packing together multiple values into a single encrypted value as used in previous work [26][22].

5. PLANNER ENGINE

When an operation cannot be executed over encrypted data, we can either use re-encryption and continue computation in the cloud or continue computation on the client-side over plaintext data. In this section we first give an intuition of where re-encryption can lead to better performance than client-side completion in the case of such unsupported operations, and then describe the heuristic the PECCARY planner engine uses to identify these situations. Finally, we describe an optimization that reduces the cost of re-encryption.

5.1 Re-encryption vs Client-side Completion

To identify which of the two options will lead to better performance we start observing that the following hold oftentimes for analytical queries:

a. Queries start with a large volume of input data but the results of the query are much smaller in size.

b. Queries over encrypted data include unsupported operations involving small amounts of data, the results of which are then used in computations with larger amounts of data.

c. Incurring a higher cost for re-encryption (decrypted and encrypting afresh) compared to client-side completion can be favorable because it can reduce the amount of data that needs to be shipped to the client and subsequently decrypted with the latter option.

Consider for example the query given below which returns the number of employees with a salary above the average salary.

```sql
A = LOAD salary;
B = GROUP A BY ALL;
C = FOREACH B GENERATE AVG(salary) AS average_salary;
D = FILTER A BY salary > C.average_salary
E = GROUP D BY ALL;
F = FOREACH E GENERATE COUNT(E);
```

In this example we first perform a GROUP operation on the entire table to compute the average salary of all employees (average_salary) and then filter based on the condition salary > average_salary. To perform this computation over encrypted data, the SUM and COUNT of aggregation functions must first be computed which must then be divided to get the average and converted to an OPE scheme to perform the subsequent inequality comparison. Since division is not supported over encrypted data, one option would be to simply continue computation on the client which requires shipping the entire table containing all salaries to the client. Instead, by realizing that the unsupported operation involves only a single tuple of data (containing the SUM and COUNT of all salaries) we can instead send that tuple to the re-encryption service, decrypt SUM to perform the division to get the average, encrypt the average under an OPE scheme, and send the value back to the cloud. Thus the rest of the query can be executed entirely in the cloud, utilizing the large amounts of computational resources and not overburdening the client with large amounts of data.

As another example, consider the following query which is part of the TPC-H Q10 benchmark query.

```sql
A = LOAD c1, c2, c3, c4, c5, c6, c7;
B = GROUP A BY c1, c2, c3, c4, c5, c6, c7;
C = FOREACH B GENERATE SUM(c1) AS total, c2, c3, c4, c5, c6, c7;
D = ORDER B BY total;
E = LIMIT D 10;
```

In this example we first perform a GROUP operation on a number of fields and then generate the sum of field c1 for each group and store it as total. Then we need to order the results based on field total which we cannot do over encrypted data since total is encrypted under an AHE scheme.
but ORDER requires an OPE scheme. Like before, we have
the option of continuing computation in the client by send-
ing fields total and fields c2 to c7 to the client. Instead
we can re-encrypt total under an OPE scheme which allows
us to perform the rest of the query in the cloud including
the LIMIT operation at the end which keeps only the first
10 tuples of the relation. Even though we initially incur a
slow down in computation to perform the re-encryption, the
overall performance of the query will be better because the
final amount of data send to the client is much less after
the Limit 10 operation is performed.

5.2 Re-encryption Heuristic

PECCARY uses an oracle which stores profiles for previously
executed queries that capture critical performance character-
stics, similarly to the approach used by Verma et al. [34].
In particular, the oracle stores the execution time and selec-
tivity (ratio of input-to-output size) at various stages of a
query (we give more insight on how these stages are defined
in the algorithm below). This information is used when a
new query is submitted to generate the following estimates
each stage of the query denoted by i:

\[ Q^R_i \] - execution time estimate for executing stage i over en-
crypted data.

\[ Q^P_i \] - execution time estimate for executing the rest of the
query starting from position i over plaintext data.

\[ V_{i,f} \] - size estimate (bytes) of field f at stage i.

\[ E_f \] - time estimate to encrypt 1 byte of field f.

\[ D_f \] - time estimate to decrypt 1 byte of field f.

Steps. Once the directed acyclic graph (DAG) that cap-
tures the query computation is generated, PECCARY greedily
considers each data flow branch of the DAG and identifies
all operations that are not supported in the cloud i = 1...N
where operation N is the first unsupported operation and oper-
ation 1 is the last unsupported operation in the query, which
in turn defines stage i to be the portion of the query
between unsupported operation i and i − 1.

1. For each unsupported operation i, generate two sets: Re-
coveSet represents all fields that need to be sent
to the re-encryption service in case of re-encryption,
ClientSet, all fields that need to be sent to the client
in case of client side completion.

2. For each operation i estimate the cost of re-encryption
step Ri and client-side completion step Si as:

\[ R_i = \sum_{f=1}^{||\text{ReencSet}_i||} V_{i,f}(D_f+E_f), \quad S_i = \sum_{f=1}^{||\text{ClientSet}_i||} V_{i,f}D_f \]

3. For each unsupported operation i, decide whether to
use re-encryption or client-side computation based on:

\[ C_i = \min (C^R_i, C^C_i) \]

where

\[ C^R_i = R_i + Q^P_i + C_{i-1}, \quad C^C_i = S_i + Q^P_i \]

and \( C_0 = S_0 + Q^P_0 \) is the cost of sending the final
results to the client and decrypting them.

\( C^R_i \) is the overall cost of performing a re-encryption at
position i and is calculated as the cost of the re-encryption
step Ri added to the cost of executing stage i over encrypted
data and the cost of the previous position of \( C_{i-1} \). Similarly,
\( C^C_i \) is the cost of using client-side completion at position i,
which includes the client-side completion step Si added to
\( Q^P_i \) which is the cost of performing the rest of the query
over plaintext data at the client. The algorithm recursively
estimates the cost of the query at each stage \((C_i)\), starting
from the end of the query and moving backwards. For each unsupported
operation i, PECCARY chooses re-encryption if
\( C^R_i < C^C_i \), otherwise it uses client-side completion.

5.3 Caching and Speculative Re-encryption

Since encrypting and decrypting data can be costly op-
erations, PECCARY uses two optimizations aiming to reduce
the overall cost of re-encryption. After a value is decrypted
from a deterministic scheme, the ciphertext-plaintext pair
can be cached. The same can be done when encrypting to
a deterministic scheme. Subsequent requests to encrypt or
decrypt a cached value are then completed much faster by
simply referring to the cached data. In addition, PECCARY
uses secure data type range information to predict the val-
ues that need to be re-encrypted. When the range of values
to be re-encrypted is bounded, PECCARY generates a map of
encrypted values that are likely to be needed. This computa-
tion takes place during times when the re-encryption service
is idle, to avoid slowing down other re-encryption requests.
Lastly, some crypto systems use large random numbers to
encrypt data (e.g., Paillier [25] and ElGamal [17]). These
random numbers do not depend on the data to be encrypted
and hence can be precomputed.

6. IMPLEMENTATION

PECCARY is made up of several components as shown in
Figure 1. The PECCARY compiler is implemented in 5500
lines of Java code and it includes a parser, a transformation
module and a metadata module. The parser parses queries
written in Pig Latin that optionally may contain Secure
Data Types information following the syntax described in
section 3. The transformation module is responsible for gen-
erating the remote query that operates over encrypted data
and the local query that will be executed on the client side
to generate the final results. The metadata module stores
information about the state of the encrypted database (e.g.,
encrypted table schemas, encryption key ids, enum maps
etc...) in the form of XML files, which is then used by the
transformation module when transforming queries.

The cloud service provider is an unmodified apache pig
service. Operations on encrypted data are implemented
through the use of pig UDFs in 2000 lines of Java code. All
UDFs extend the EvalFunc class and in addition, UDFs of
aggregation functions such as SUM also implement the Alge-
braic interface. This allows the pig engine to deploy Hadoop
jobs that make use of the Hadoop combiner when comput-
ing these UDFs to do partial aggregations before computing
the final value.

The re-encryption service is implemented as a Java server
using Java sockets. Requests for re-encryption are made
through special UDFs that first establish a connection with
the re-encryption service and then send data for re-encryption.
Each container in a Hadoop job uses a dedicated connection
with the server and in turn, the re-encryption server uses multiple threads to handle these connections.

Finally, the local query executor is implemented as a pig service running in local mode that executes the local query. Similar to the cloud service, the local query executor has access to a set of UDFs that it uses to decrypt the results of the remote query which act as the input for the local query.

7. EVALUATION

In this section, we empirically assess the benefits of our proposed abstractions and techniques. We evaluate PECCARY on four different aspects. We first evaluate how valuable our compilation techniques are in improving expressiveness and efficiency by examining how frequently they apply to analytical queries. We then evaluate the performance of PECCARY by comparing its execution time to executions on plaintext in Pig and to the closest related system for computing on encrypted data. We also examine how effective the proposed re-encryption heuristic is in executing queries more efficiently. Finally we evaluate scalability of PECCARY when running queries on large volumes of data.

7.1 Experimental Setup

To evaluate PECCARY we use the following two standard industry adopted-benchmarks:

1. TPC-H is a decision support benchmark comprised of a set of 22 queries. These queries are designed to have broad industry-wide relevance and be representative of realistic decision support query with a high degree of complexity that give answers to critical business questions.

2. PigMix2 [3] is a set of queries used by Apache to test performance of the Pig runtime. PigMix2 queries measure latency of various features like grouping, ordering, projecting, different types of joins, or aggregation operations. Data for PigMix2 is generated by an associated data generator tool which produces data with a Zipfian distribution for grouping and join keys. Other fields are generated using uniform data distribution.

We performed all our experiments using Amazon EC2 instances. We used m4.large instances which have 2 virtual CPUs and 8 GB of memory or m4.xlarge instances which have 4 virtual CPUs and 16GB of memory to represent the untrusted cloud. We also used an EC2 VM for the trusted client machine, where the re-encryption service is run and the final results are sent. While we retain the default EC2 network throughput for all nodes within the cloud (450Mbps for m4.large instances and 750Mbps for m4.xlarge instances) but cap the network bandwidth between cloud and that client machine to 10Mbps.

To provide an insight on how the compilation techniques of PECCARY improve performance over the state-of-the-art approaches that perform computations over encrypted data, we compare PECCARY to Crypsis [31]. Crypsis is a modified Pig Latin runtime that performs data flow analysis and program transformations for Pig Latin scripts automatically and transparently to enable their execution on encrypted data. Unlike PECCARY, Crypsis employs the greedy re-encryption technique to handle operations that cannot execute in the cloud. It also does not support all computations involved in our used benchmarks (e.g., substrings, string patterns, floating point numbers). Thus we extend Crypsis to handle such computations in order to enable a comparison, and call this system Crypsis*. We note that Crypsis* does not include any of our compilation techniques that lead to more optimized execution.

Similar to other PHE approaches [26, 33, 31], we assume that data is already encrypted and securely stored in the cloud, and hence do not include encryption latency in our evaluations. We use HDFS as our storage medium with a replication factor of 3. All reported execution times in the experiments that follow are the average of 3 runs.

7.2 Compilation Techniques Applicability

We first analyze how applicable our proposed compilation techniques are in TPC-H and PigMix2 benchmarks. Table 7 shows the number of scripts each compilation technique was applicable for both benchmarks.

<table>
<thead>
<tr>
<th># of scripts</th>
<th>PigMix2</th>
<th>TPC-H</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression Rewriting</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Selective Encryption</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>Efficient Encryption Strategy</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Caching and Speculative Re-encryption</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>Subexpression Elimination</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

To evaluate the effectiveness of our compilation techniques in reducing the number of re-encryptions required to execute queries, we use TPC-H to compare the number of queries that require re-encryption to complete in Crypsis* and in PECCARY, and compare the total number of re-encryptions involved too. Table 8 shows the results of this comparison. In Crypsis*, 18 out of 22 TPC-H queries require re-encryption to complete with a total of 51 re-encryptions involved in all 22 queries. In comparison, PECCARY compilation techniques allow PECCARY to execute TPC-H with only 6 queries requiring re-encryption for a total of 7 re-encryptions in all 22 queries.

<table>
<thead>
<tr>
<th># of queries with re-encryption</th>
<th>Crypsis*</th>
<th>PECCARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total # of re-encryptions</td>
<td>51</td>
<td>7</td>
</tr>
</tbody>
</table>

7.3 Execution Time

Executing queries over encrypted data provides confidentiality in an untrusted cloud but due to the size overhead of the encrypted data and the higher cost of applying operations on encrypted data, execution times are impacted. To evaluate this overhead, we compare the execution time of PECCARY with the plaintext execution and Crypsis* in order to assess the performance gains due to our compilation techniques and the re-encryption heuristic on the TPC-H and PigMix2 benchmarks.

TPC-H. We run TPC-H at scale factor 10, i.e., 10GB of plaintext data. TPC-H data is divided across 8 tables with a total of 61 fields. We selected 4 of these fields and
marked them as non-sensitive (security level NONE), namely the discount, tax and quantity fields in the lineitem table and the available-quantity in the partsupp table. In addition we represent 5 of these fields as enum types (ship-instruction, nation-name, region-name, order-status and ship-mode). We used a cluster of 10 m4.xlarge nodes as the untrusted cloud and 1 m4.large node as the trusted client where we run the re-encryption service too. We capped the execution time to 3 hours (only relevant to Crypsis∗) and marked queries not completed in that time as timed out.

Figure 2 shows the execution time increase of all 22 TPC-H queries in Crypsis∗ and PECCARY compared with the plaintext execution time. Crypsis∗ times out in 6 of the queries, and has an average overhead of over 17× in the remaining queries with a minimum overhead of 2.3× and a maximum of 43.5×. In comparison, PECCARY achieves much lower overheads by not only being able to execute all 22 queries with no timeouts, but also having an average overhead of only 2.10× (ranging from 1.07× to 4.49×). The queries with highest overhead are Q1, Q11 and Q15 with overheads of 4.49, 4.04 and 3.62 respectively. The overhead of Q11 is due to the substantial amount of computation that needs to be performed in the client which has less resources. The overhead of Q1 and Q15 is mainly due to executing the expressions extendedprice * (1 - discount) and extendedprice*(1-discount) ×(1+tax) on the lineitem table which is the largest table of the TPC-H benchmark. These expressions are executed entirely in the cloud by utilizing the secondary secure operation of the AHE scheme (Paillier) to perform the multiplications after realizing that the tax and discount fields are marked as non-sensitive. We note that despite the ~4× overhead of these two queries, to the best of our knowledge, PECCARY is the only system that can execute these kinds of expressions entirely in the cloud. Other approaches either use re-encryption to handle these expressions hence incurring a higher overhead (e.g. Crypsis∗) or pre-compute the entire expressions into a separate field and only refer to the result during the query executions (e.g., Monomi [33]).

**PigMix2.** PigMix2 queries were designed to evaluate the performance of Pig Latin operators and to not to semantically represent common queries used in applications. As such, the input schema of PigMix2 does not give us an intuitive understanding of the fields that need to be sensitive. Hence, we arbitrarily picked two fields to be non-sensitive and 1 field to be represented as an enum type, out of a total of 521 fields. For this experiment, we used 10 m4.xlarge computing nodes as the untrusted cloud and 1 m4.large node as the client. No re-encryption was required in any of the queries. Using the PigMix2 data generator tool we generated a total of 26 GB of input. This translates to 5 GB of input for queries L01 to L16 and 15 GB of input for query L17. Figure 3 shows the results of the 7 most affected queries out of a total of 17. No performance penalties were incurred in the remaining 10 queries, but since no significant gains were achieved either, we elide the corresponding results. Even though the queries of the PigMix2 benchmark are poor on expressions leaving less opportunities for optimization, even with only 2 out of a total of 521 fields chosen to be non-sensitive, we can already see some improvements in performance. On average, PECCARY incurs 3× overhead compared with the plaintext version. Compared to Crypsis∗ which has an overhead of 4×, PECCARY is approximately 30% faster. In the best case, for query L15, PECCARY reduces the overhead to 3.8× with respect to plaintext from 6.3× overhead occurred in Crypsis∗.

Figure 3: PigMix2 latency. Plaintext execution times (seconds) are given in parentheses.

### 7.4 Re-encryption Heuristic

In this section, we evaluate the effectiveness of our re-encryption heuristic by running the queries in TPC-H that include operations that execute entirely in the public cloud without utilizing trusted resources. In Figure 4 we compare
our heuristic to the following two extreme cases of utilizing trusted resources (client) for performing queries:

(a) Client-side completion: query execution continues on the client immediately after the first occurrence of an operation that cannot execute in the cloud. (b) Greedy re-encryption: re-encryption is applied greedily to handle all operations that cannot execute in the cloud; return to the client happens only after query completion with the final results. (c) PECCARY: the PECCARY re-encryption heuristic is used to decide at what point of a query to stop using re-encryption and complete the remaining computation at the client side, as described in Section 5.

After we apply our compilation techniques, TPC-H has 6 queries that require utilizing trusted resources to complete. Five of these queries include only 1 such operation and 1 query, namely Q11 includes two such operations. Depending on the location of each such operation in the query, and the amount of data it involves, it might be more efficient to perform re-encryption and continue computation in the cloud or continue computation of the remaining part of the query in the client. As shown in Figure 4, Q3 performs better if re-encryption is used to deal with unsupported operations. In contrast, Q5 performs better if the client-side completion is used to deal with unsupported operations. PECCARY’s re-encryption heuristic is able to choose whether to use re-encryption or client-side completion to gain the best execution performance. More interestingly, the heuristic makes a decision on each such operation and in queries that include multiple unsupported operations such as Q11. In this case, PECCARY is able to apply re-encryption to a number of unsupported operations before using client-side completion.

### 7.5 Scalability: Top-k

We used PECCARY to find the top 10 Wikipedia projects viewed by users on 150 GB of Wikipedia traffic statistics data. Each row of the data set has 4 fields: (i) a project-code which is the Wikipedia project name, (ii) the title of the page retrieved (pagename), (iii) the number of requests for that page (pageviews), and (iv) a bytes field representing the size of content returned. The PECCARY query loads the encrypted data, groups it by project code, and sums the number of pageviews for each project code. Finally the query orders the results, and retrieves the top-10 results.

PECCARY executes this query over encrypted data by encrypting project-code under a DET scheme, and pageviews under an AHE scheme. It then performs a re-encryption per group to convert the result of sum per group to OPE scheme to support the order by operation. The results are summarized in Figure 5. We can see that PECCARY scales well with the number of nodes, reducing latency by a factor of 1.7 when the number of nodes go up from 20 to 40. Also, this latency reduction factor of 1.7 in PECCARY matches with the 1.6 reduction factor in plaintext, showing that there are no scaling bottlenecks because of PECCARY.

### 8. RELATED WORK

#### Computing over encrypted data.
CryptDB [26] is a system that provides confidentiality for applications using an SQL database backend. It works by executing SQL queries over encrypted data using a collection of encryption schemes. CryptDB also uses a trusted proxy to analyze queries. Monomi [33] builds off of CryptDB’s design and introduces split client/server query execution which can support more queries. Monomi also proposes techniques that improve performance for different workloads and adds a designer to automatically choose an efficient design suitable for each workload. Talos [28] encrypts and stores data specific to the Internet of Things in a cloud database while still allowing query processing. Unlike PECCARY these systems do not allow MapReduce-style parallelization of queries. Further, they depend on data definitions provided by MySQL or Postgres databases which makes it difficult to leverage optimizations that PECCARY uses that rely on specifying data ranges, precisions, or sensitivity levels.

As explained in the previous section, Cryptsys [31] is a modified Pig Latin runtime that performs data flow analysis and program transformations for Pig Latin scripts, automatically and transparently enabling their execution on encrypted data. Cryptsys focuses on the program transformation itself and does not propose novel abstractions or compile-time and runtime optimization techniques such as expression rewriting, caching/speculative re-encryption, etc. The performance improvements we have over Cryptsys because of these techniques are evident from Section 7.

#### Confidentiality using trusted hardware.
Haven [9] offers another way of protecting data confidentiality using specialized hardware that provides a trusted computing base. Haven relies on hardware encrypted and integrity protected physical memory, which requires a specialized CPU. CipherBase [6] provides an FPGA-based implementation of a trusted hardware that can be used to run a commercial SQL database system without sacrificing data confidentiality. TrustedDB [8] preserves confidentiality by using a server-hosted, tamper-proof trusted hardware in critical query processing stages. These approaches are orthogonal to the approach of PECCARY and could be used to extend the
design of peccary by allowing secure computations or subcomputations to be performed on a trusted hardware in an untrusted cloud, thus reducing the amount of computation that needs to be performed in the client.

9. CONCLUSIONS

Partially homomorphic encryption seems appealing to support big data analytics in the public cloud while preserving data confidentiality and with practical performance. Existing works that try to leverage PHE to achieve data confidentiality either cannot achieve expressiveness or sacrifice practical performance to do so.

This paper presents peccary, the first practical system for confidentiality-preserving big data analytics in the cloud which achieves both expressiveness and practical performance. To this end, peccary leverages three key concepts: the novel abstraction of secure data types, a set of compilation techniques, and a planner engine for efficient execution. We have demonstrated how these contributions in combination allow for practical performance, and in particular, substantial performance improvements over related approaches.

10. REFERENCES