Privacy-aware Distributed Incremental Computation

M. Köhler
TU Darmstadt
Germany
kohler.mirko@gmail.com

P. Haller
KTH Royal Institute of Technology
Sweden
phaller@kth.se

S. Erdweg
TU Delft
Netherlands
S.T.Erdweg@tudelft.nl

M. Mezini
TU Darmstadt
Germany
mezini@st.informatik.tu-darmstadt.de

G. Salvaneschi
TU Darmstadt
Germany
salvaneschi@st.informatik.tu-darmstadt.de

Abstract

Distributed incremental processing is an effective solution for processing large amounts of data in an efficient way. In this setting, algorithms for *operator placement* automatically distribute data queries the the available processing units. However, current algorithms for operator placement focus on performance and ignore privacy concerns that arise when handling sensitive data. We present ongoing research on a new methodology for privacy-aware operator placement that both prevents leakage of sensitive information *and* improves performance. We implement a working prototype based on previous work on (local) incremental computation.

Keywords Data Privacy, SQL, Information-Flow Type System, Operator Placement, Scala

ACM Reference format:
M. Köhler, P. Haller, S. Erdweg, M. Mezini, and G. Salvaneschi. 2017. Privacy-aware Distributed Incremental Computation. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, Barcelona, Spain, June 18–23, 2017 (PLDI'17),* 2 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

One of the major challenges for achieving high performance in a distributed system is utilizing all available processing units to their full extent, taking load balancing, latency, and bandwidth into account. In particular, systems that solve the *operator placement problem* translate high-level descriptions of sequential computations into efficient distributed computations. These systems provide a high-level language-based interface to distributed systems. In this work, we consider the problem of placement from the perspective of *data privacy* in a distributed system for incremental computation where intermediate operators propagate incremental changes that contribute to the final result of a query.

While it is relatively easy to secure communication channels, it is virtually impossible to protect sensitive data unless one controls the machine and can prevent a concurrent process, the OS, or the hardware from leaking information. We tackle this problem by introducing a privacy-aware placement strategy that does not distribute sensitive data to machines that are deemed to be insecure.

We proposes a two-phase algorithm for automated privacy-aware operator placement of SQL-like, – incremental and distributed – queries over relational data. The first phase generates a set of deployment candidates that do not violate privacy constraints. The second phase finds the best placement based on a cost model. Our prototype, SecQL , is an extension to i3QL [2], an existing framework for incremental computation.

2 Overview

Th running example of a Hospital information system introduces our approach. The hospital's clinical database *PatientDB* stores information about the current patients. The *KnowledgeDB* database contains data of case reports (symptoms and corresponding diagnosis etc). The *PersonDB* database provides general information about citizens. Their information can be combined to find diagnoses and suggestions for the current patients, e.g., a user can request the name of all patients that have symptoms that could point to allergy.

```
1 val result = SELECT (*)
2  FROM (personDB, patientDB, knowledgeDB)
3  WHERE ( (person, patient, knowledge) =>
4    person.id == patient.id AND patient.symps == knowledge.symps
5    AND knowledge.diagnosis == "Allergy" )
```

For simplicity, we assume that each database contains a single table (e.g., the *PersonDB* contains a *PersonDB* table). The query can be represented as a tree of operators, shown in figure 1. In the tree, the condition from the WHERE-clause has been split into three operators: The selection know.diagnosis == "Allergy" (close to the Knowledge source), the join pers.id == pati.id, and the join pati.symps == know.symps.

When a new element is added to one of the databases, the change is propagated through each operator in the tree. Finally the result of the query is incrementally updated.

PLDI'17, Barcelona, Spain
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

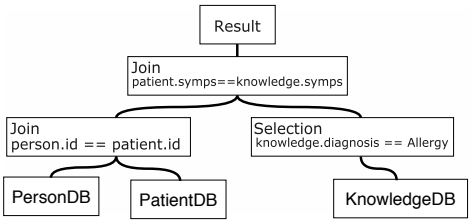

Figure 1. Local query.

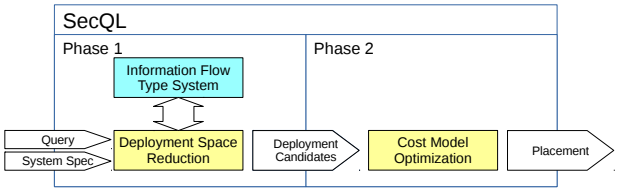

Figure 2. Overview of SecQL .

Distribution and Privacy Using the tree representation, SecQL automatically finds optimal deployments in a distributed system which satisfy user-defined privacy requirements. An overview of the approach is in figure 2. The input is, additionally to a query tree, the system specification about the distributed environment, i.e. available hosts and access control specifications. The query is processed in two phases.

The first phase, *deployment space reduction*, generates all possible deployments that do not violate privacy constraints. The constraints are derived from (1) the sensitivity of the source relations, where we allow each column to declare a different sensitivity, for example, to differentiate between a person's name, home address, and diagnosis; (2) the privilege of the hosts involved, which determines what data may be forwarded where; and (3) the information flow of the query, because privacy concerns only arise where sensitive data can flow to a non-privileged processing unit. An **information flow type system** (not discussed here, for brevity), provides a static taint analysis that tracks the taint of each column individually. A placement algorithm for the incremental operators is expressed as a **code transformation** that introduces remote connections into the queries. The transformation can be proven correct with respect to the type system to ensure that all operators are deployed on hosts with sufficient permissions.

The second phase, *cost model optimization*, computes the optimal deployment based on performance metrics among all candidates. For the cost of a deployment, we consider bandwidth and CPU load, but the approach can be easily extended to other metrics such as latency. The goal of this phase is to find the deployment with the minimal cost to achieve the best performance for the distributed incremental computation.

Our approach derives an operator placement that is provably correct: It preserves the sequential behavior and never leaks sensitive data.

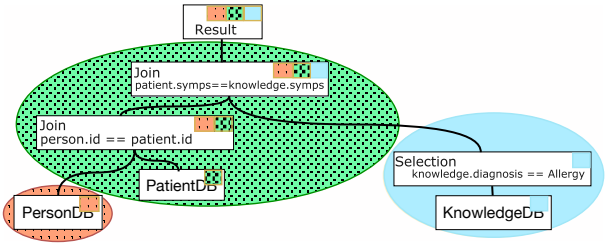

Figure 3. Distribution of the query.

Example Figure 3 shows a possible distribution for the query above. We assume that all databases run on different hosts – the colored background circles. The following privacy constraints hold. The information in *PatientDB* is private: Only the client and the host of *PatientDB* can access the data. *PersonDB* can also be accessed by the host of *PatientDB*. The *KnowledgeDB* database, on the other hand, contains public data. Data is sent to the host of *PatientDB*, it is aggregated there, and the result is sent to the client. This is correct, because the host of *PatientDB* has permission to access all data. The knowledge database performs the selection before sending the data to minimize the network load. In summary, this placement reduces the amount of computation performed on the client and the amount of data transferred over the network.

3 Implementation

We implemented SecQL as a Scala DSL extending i3QL [2], which provides SQL-like queries, local incremental data processing and relational algebra optimizations. We use light weight modular staging (LMS) [3] to inspect and edit functions as well as performing common subexpression elimination. We use the relational algebra trees generated by i3QL and distribute the operators, thus gaining benefits from all optimizations.

For the distribution, we compile SecQL queries to Akka actors [1]. Each operator in the tree is executed inside an actor and deltas among operators are transmitted as asynchronous actor messages. Also, we extended i3QL to support the additional syntax for privacy.

References

[1] Akka. 2017. Akka toolkit and runtime. (2017). http://akka.io.

[2] Ralf Mitschke, Sebastian Erdweg, Mirko Köhler, Mira Mezini, and Guido Salvaneschi. 2014. I3QL: Language-integrated Live Data Views. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 417–432. DOI: http://dx.doi.org/10.1145/2660193.2660242

[3] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *SIGPLAN Not.* 46, 2 (Oct. 2010), 127–136. DOI: http://dx.doi.org/10.1145/1942788.1868314