Safe Query Processing for Pairwise Authorizations in Coalition Networks

Qiang Zeng, Jorge Lobo, Peng Liu, Seraphin Calo, Poonam Yadav

Penn State Univ., IBM Watson

ACITA
September 2012
Example scenario (1/2)

- Information is shared among servers of multi-parties
- A distributed DB system is established by the servers
- Top concerns: Safety, flexibility and efficiency.
Example Scenario (2/2)

- Say, for some specific data, its owner Party V1 only wants to share with V2 and V3
- For some other data, V1 only wants to expose it to V2 and V4
- How to achieve such information sharing autonomy?
- Goal: A safe and efficient solution to autonomous information sharing in a multi-party distributed system.
Requirements for access control

- R1: each party has its own view over the database.
- R2: each party can independently determine which portion of its data is shared and with whom.
- R3: tuple-granularity access control.
- Last but not least, low communication cost
Existing work

- None has addressed $R1-R3$ simultaneously.
- Federated database systems: all parties share a uniform view over the database [Bocca et al., VLDB’94], [Vimercati, JCS’97], which violates $R1$.
- [Vimercati JCS’11] requires different parties to define policies collaboratively and cannot provide tuple-granularity access control, which violates $R2$ and $R3$. 
A policy is defined as a triple \(<Vi, Vj, tuple_set>\), where \(tuple_set\) defines a set of tuples owned by \(Vi\) and accessible by \(Vj\), that is, \(Vi\) is the data owner party, while \(Vj\) is the consumer.

**Key uniqueness:** (1) the data consumer is a specific party (instead of the whole federation) \((R1)\); (2) the policy definer is the data owner (instead of some supervisor) \((R2)\).

So, a safe query processing has to consider the view disparity between parties, when data is transmitted among servers.
Semi-join [Bernstein et al., 1981] breaks down a join query into two sub-joins to save communication cost.

However, it assumes the view equality between parties.

We propose split-join, which splits a join to three sub-joins to save communication cost and is compliant with the view disparity between parties:

$$A \text{ join } B = A \text{ join } (B_1 \cup B_2)$$

$$= (A \text{ join } B_1) \cup (A_1 \text{ join } B_2) \cup (A_2 \text{ join } B_2)$$
Split-join (2/2)

$A \text{ join } B = (A \text{ join } B_1) // \text{ step 2, 5}$

$U (A_1 \text{ join } B_2) // \text{ step 1, 6}$

$U (A_2 \text{ join } B_2) // \text{ step 3, 4}$

- Given a medium join selectivity factor, we can expect
  $|A_1 \text{ join } B_2| < |A_1|$ and
  $|A \text{ join } B_1| < |B_1|$ So, the total communication cost may be much lower than that of a straightforward and safe strategy by sending $A$ and $B$ to the destination directly.

The consolidator is $S_b$
The master is $S_1$

Steps: (1) $<S_1, S_2, A_1>$,
(2) $<S_2, S_1, B_1>$,
(3) $<S_1, S_b, A_2>$,
(4) $<S_2, S_b, B_2>$,
(5) $<S_1, S_b, A \bowtie B_1>$,
(6) $<S_2, S_b, A_1 \bowtie B_2>$
In each join, a buddy can act as a broker.

(a) Semi-join

(b) Peer-join

(c) Broker-join

The **consolidator** is $S_1$

*Steps:* (1) $<S_1, S_2, \pi_{\text{district}}(A)>$

(2) $<S_2, S_1, \pi_{\text{district}}(A) \bowtie B>$

The **consolidator** is $S_2$

*Steps:* (1) $<S_1, S_2, A>$

The **consolidator** is $S_b$

*Steps:* (1) $<S_1, S_b, A>$,

(2) $<S_2, S_b, B>$
The most efficient join method for “A join B” is not necessarily the best in “A join B join C”, considering, e.g., the server that obtains “A join B” may vary for different join methods.

An algorithm that achieves the best overall efficiency for any given query is proposed.
It takes a poster-order walk over the query tree to accumulate candidate query strategies and finally annotates the tree with the best strategy.
We have proved the algorithm

- Correct: always generate correct query results
- Safe: compliant with all policies

We also proved a desirable property of the algorithm: *Authorization Confidentiality*, i.e., the policy definition doesn’t need to be leaked for executing the query.
Experiments

- The experiments compare the costs of following cases:
  - Case 1: all related tables are sent to $Sq$
    --- baseline
  - Case 2: buddy servers are explored
    --- save 42% communication cost
  - Case 3: split-join is applied
    --- save 39%
  - Case 4: both buddies and split-joins are used
    --- save 60%
Conclusion

- Identified essential information sharing needs:
  - R1: per-party view
  - R2: data owner has the information sharing autonomy
  - R3: fine-granularity access control
- Formalized the authorization policies defined in terms of parties and tuple set.
- Proposed a novel join method (split-join) and an algorithm that generates efficient query strategies.