Distributed Differential Datalog

1 Model description

We assume a network of computation nodes, each running a version of Differential Datalog [1]. Every node has its own Datalog program, i.e. set of Datalog rules, and its own relations which are stored locally. Nodes communicate by using the output stream of one node as input stream for another node. In addition to inputs resulting from computations in other nodes, nodes might receive inputs from external sources.

2 Example

Consider for example a computer network as depicted in Figure 1 with a hierarchy of switches that perform packet filtering. The switches might use DDDlog in order to populate and maintain tables containing information about the participating hosts and whether they are allowed to send packets. The switches which are closest to the end hosts, here switches $S1$ and $S2$, gather information about which hosts are currently connected. Higher level switches, here switch $S3$, on the other hand, decide whether certain hosts are allowed to send packets. For example, there might be a policy to temporarily drop all packets from hosts that behave badly, e.g. by flooding the network with too many packets. Using observations of the hosts' behavior, higher level switches construct blacklists of hosts whose packets should be dropped. As the filter should become effective as close to the bad hosts as possible, the blacklists are then shared with the lower level switches.

The Datalog program that is run in switch $S1$ is displayed in Listing 1. The rules for switch $S2$ are symmetrical. Whenever a host joins or leaves the network there will be an event in the `host` input stream. The switch than stores those hosts in its local host relation that are connected via one of its ports. The local blacklist table is updated on changes in $S3$'s blacklist table to one of the local hosts.

```
input relation host(hostID: int, switchID: int)
input relation S3.blacklist(hostID: int)

output relation S1.host(hostID: int)
output relation S1.blacklist(hostID: int)

S1.host(id) :- host(id, 1).

S1.blacklist(id) :- S3.blacklist(id, 1).
```

Listing 1: Rules for switch $S1$

The Datalog program that is run in switch $S3$ is displayed in Listing 2. Switch $S3$ takes as input the host relations of the other switches in order to collect all hosts that are currently connected. Furthermore, it receives an external input stating which hosts are supposed to be blacklisted.

In the example four hosts $H1$ - $H4$ are connected to the bottom level switches. The first events in the system will be some external inputs that lead the hosts to be added to the `host` tables of $S1$ and $S2$. These additions will be propagated and populate the `host` table of $S3$. Then assume at some point host $H3$ starts flooding the network. It will be added to the $S3.blacklist$ and consequently to $S2.blacklist$. When at some point the host stops flooding the network and is allowed to send packets again, it will be removed from all blacklist tables.

3 Eventual Consistency

In order to show eventual consistency of a DDDlog system, we want to show that if external inputs stop at some point, eventually

1. the system stabilizes, i.e. no new outputs are generated and
2. some application specific property P, describing the resulting local relations holds.

In the example from the previous section the properties one wants to check might include $S3.host = S1.hosts \cup S2.hosts$, $S1.blacklist \subset S3.blacklist$ and $S2.blacklist \subset S3.blacklist$.

```
input relation S1.host(hostID: int)
input relation S2.host(hostID: int)
input relation blacklist(hostID: int)

output relation S3.host(hostID: int, switchID: int)
output relation S3.blacklist(hostID: int,
        switchID: int)

S3.host(id, 1) :- S1.host(id).
S3.host(id, 2) :- S2.host(id).

S3.blacklist(hostID, switchID) :-
        blacklist(hostID),
        S3.host(hostID, switchID).
```

Listing 2: Rules for switch $S3$

Conjecture

If there is no distributed recursion, both requirements stated in the definition for eventual consistency follow from the corresponding properties in a non distributed DDlog program constructed by taking the composition of the DDlog programs running on all nodes.

3.1 Programs without distributed recursion

Definition 1. A *DDlog program* (I, O, R) consists of a finite set I of input relations, a finite set O of output relations and and a finite set R of rules. An input relation r_i is declared by `input relation` $r_i(a_1, ..., a_m)$, an output relation r_j by `output relation` $r_j(a_1, ..., a_m)$. Rules have the form

$$r_1(u_1) : -r_2(u_2), ..., r_n(u_n).$$

where the head relation r_1 is an output relation and the body relations $r_2, ..., r_n$ are either input or output relations.

Definition 2. The *dependency graph* of a DDlog program (I, O, R) is a graph $G_d = (V_d, E_d)$ where $V_d = O \cup I$ and for $r_i, r_j \in V_d$ there is an edge

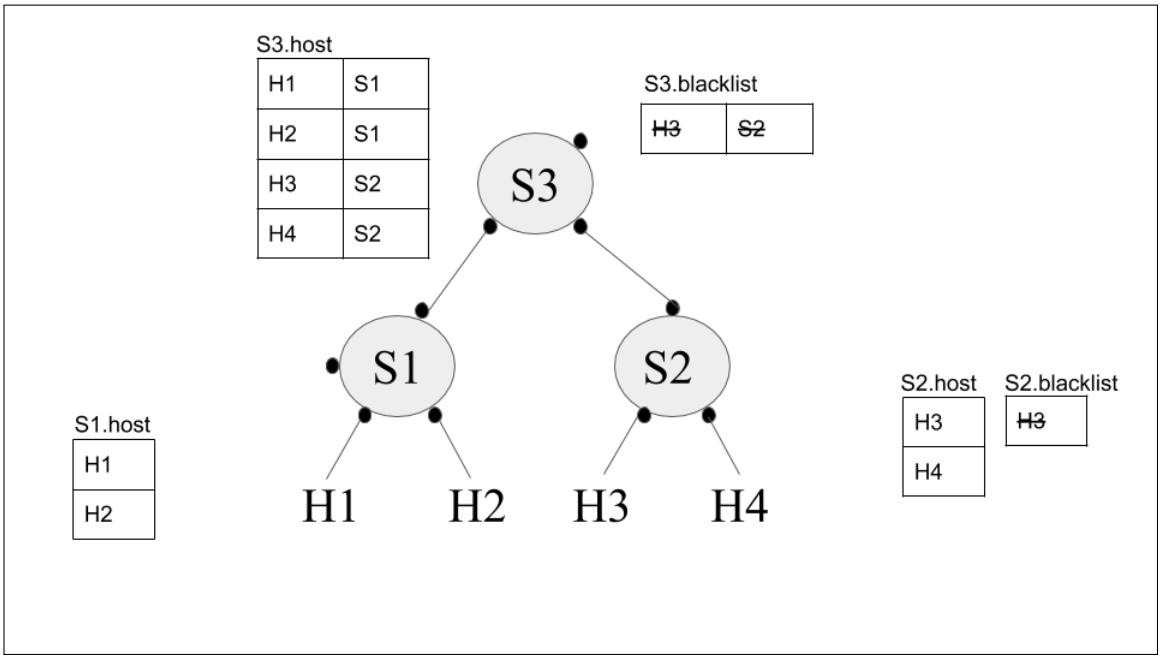

Figure 1: Computation of blacklists based on DDDlog

$(r_i, r_j) \in E_d$ if and only if there is a rule in R with head relation r_i and a body relation r_j.

Remark: The following definition is a rather informal description of DDlog semantics. Having a more formal definition (e.g. some kind of operational semantics) might make the proof cleaner.

Definition 3. *Semantics of DDlog*: DDlog maps a stream of changes in the input relations to a stream of changes in the output relations. If the stream of input changes is finite, the stream of output changes is finite. In particular, if there are infinitely many changes to some output relation there must have been infinitely many changes to an input relation that the output relation depends on, i.e. an input relation that is a successor in the dependency graph.
For finite streams of input changes, the *accumulated input* In of a DDlog program is the state of its input relations after applying all input changes. The *accumulated output* is the state of the output relations after applying all output changes.
The accumulated output of a DDlog program equals the least fixed point of the *immediate consequence operator* applied to the accumulated input. The immediate consequence operator for a set of rules R and a set of tuples T is

defined as

$$\Gamma_R(T) = T \cup \{t | t :- t_1, ..., t_n \text{ is a valid instantiation of a rule in } R \text{ with each } t_i \in T\}.$$

Definition 4. A *distributed DDlog program* is a tuple (V, E, O, I, R).

- $V = \{v_1, ..., v_n\}$ is a finite set of DDlog programs where $v_i = (I_i, O_i, R_i)$ for all $1 \leq i \leq n$.
- $O = \bigcup_{i=1}^{n} O_i$ is a set of output relations, with $O_i \cap O_j = \emptyset$ for all $1 \leq i, j \leq n$.
- $I = \bigcup_{i=1}^{n} I_i$ is a set of input relations with $I_i \cap I_j = \emptyset$ for all $1 \leq i, j \leq n$. I is partitioned into a set of external input relations $I_{ext} = I - O$ and a set of internal input relations $I_{int} = I - I_{ext}$.
- $R = \bigcup_{i=1}^{n} R_i$ is a set of DDlog rules.
- $E \subseteq V \times O \times V$ is the edge relation. For an edge (v_1, o, v_2) between nodes $v_1 = (I_1, O_1, R_1)$ and $v_2 = (I_2, O_2, R_2)$ it must hold that $o \in O_1$ and $o \in I_2$. Furthermore, edges form reliable links between nodes. For any edge (v_1, o, v_2) if there is an output change on relation o in v_1, the same change will eventually appear at the corresponding input relation of v_2. If there are only finitely many output changes to o, the accumulated output of relation o at v_1 equals the accumulated input of o at v_2.

Definition 5. The dependency graph of a distributed DDlog program (V, E, O, I, R) is a directed graph $G_d = (V_d, E_d)$ where $V_d = O \cup I$ and for $r_i, r_j \in V_d$ there is an edge $(r_i, r_j) \in E_d$ if and only if there is rule in R with head r_i and a body containing r_j.

Two relations recursively depend on each other if they occur on a cycle in the dependency graph. Hence, it is useful to keep track of recursive dependencies by considering the strongly connected components (SCCs) of the dependency graph or equivalently nodes in the condensation of the dependency graph.

Definition 6. Let G be a directed graph. The *condensation* of G is the directed acyclic graph (DAG) G' containing one node per SCC in the original graph where an edge in G' is present if and only if there exists an edge between the nodes of the corresponding SCCs.

Definition 7. A distributed DDlog program has *distributed recursion* if two nodes from the same SCC in the dependency graph belong to two different nodes in the distributed DDlog program.

Definition 8. Let (V, E, O, I, R) be a distributed DDlog program. Its *composition* is the DDlog program (I_C, O_C, R_C) where $I_C = I_{ext}$, $O_C = O$ and $R_C = R$.

Theorem 3.1. Let D be a distributed DDlog program without distributed recursion. If at some point there are no more changes to the external inputs of D

1. it will eventually stop producing output changes and
2. a fact is in the accumulated output of D if and only if it is in the accumulated output of its composition run on the same inputs.

Proof. Let $D = (V, E, O, I, R)$ be a distributed DDlog program without distributed recursion.
1. Termination: By contradiction. Assume there are only finitely many changes to the external input but there are infinitely many changes to the output of D. This means that at least one node $v \in V$ generates infinitely many outputs. Then, by Definition 3, v must have received an infinite amount of inputs for a relation that the corresponding output relation depends on. Given that external inputs are finite, there must be an internal input relation to v with infinite changes. Since D does not have distributed recursion and the relation belongs to another node, it must belong to another SCC in the dependency graph. In particular, the input relation must belong to a successor SCC in the condensation of the dependency graph. Repeatedly applying the same argument yields an infinite path in the condensation of the dependency graph which is a contradiction to the fact that the condensation is a DAG.

2. Correctness: Let $C = (I_C, O_C, R_C)$ be the composition of D. Let f be a fact in the accumulated output of C. Let In denote the accumulated input facts. As the accumulated output of C is the least fixed point of the immediate consequence operator applied to the accumulated inputs, it must hold that $f \in \Gamma^n(In)$ for some $n \in \mathbb{N}$. We prove that f is in the output of some node $v \in V$ by induction over n. For the base case assume $f \in \Gamma(In)$. Hence, there is a rule $r \in R_C$ that says $f :- f_1, ..., f_k$ where $f_1, ..., f_2$ are input facts. By definition of C, there is a node $v_i \in V$, $v_i = (I_i, O_i, R_i)$ such that $r \in R_i$. As v_i runs a valid DDlog program, the relations of $f_1, ..., f_k$

are in I_i. As D – and hence v_i – receives the same accumulated inputs as C for those relations and the accumulated output of v_i is the fixed point of Γ_{R_i} over its accumulated inputs, it will generate output f. For the inductive hypothesis assume that for all $m < n$ some node in V generates fact f' if $f' \in \Gamma^m(In)$. Let $f \in \Gamma^n(In)$. Hence, there is a rule $r \in R_C$ that says $f :- f_1, ..., f_k$ where for all $1 \leq j \leq k$ $f_j \in \Gamma^m(In)$ for some $m < n$. By definition of C, there is a node $v_i \in V$, $v_i = (I_i, O_i, R_i)$ such that $r \in R_i$. As v_i runs a valid DDlog program, it takes the relations of $f_1, ..., f_k$ as input. The relations of $f_1, ..., f_k$ are either external inputs or in the accumulated output of some node $v' \in V$. Let $1 \leq l \leq k$ and let $f_l = R_l(a_l)$. If R_l is an external input relation, v_i receives the same inputs as C by assumption. If R_l is an internal input relation, by Definition 4, the accumulated input to R_l of v_i equals the the accumulated output of some v' for R_l. By the inductive hypothesis, the outputs of R_l are correctly generated at v'. Given that v_i correctly implements fixed point semantics, f is in the accumulated output of v_i.

The second direction is shown by contradiction. Let f' be a fact in the accumulated output of D that is not in the accumulated output of C. Let $v \in V$ denote the node that generated the invalid output. As C per definition contains all rules belonging to v, v must have received different inputs than C. As C by assumption gets all the external inputs that v gets, the difference in the input must be in some internal input relation, i.e. was generated as output by another node $v' \in V$ computing the relations of a successor SCC in the dependency graph. Furthermore, the difference in the output was not derived in C either (otherwise f' could have been derived in C). Applying the same argument repeatedly yields an infinite path in the condensation of the dependency graph which is a contradiction to the fact that the condensation is a DAG.

□

References

[1] Ryzhyk, L., and Budiu, M. Differential datalog. In *Datalog 2.0* (2019), pp. 56–67.