Disclosure Analysis of SQL Workflows

Marlon Dumas[1], Luciano García-Bañuelos[1], and Peeter Laud[2]

[1] University of Tartu, Estonia
{marlon.dumas,luciano.garcia}@ut.ee
[2] Cybernetica, Estonia
peeter.laud@cyber.ee

Abstract. In the context of business process management, the implementation of data minimization requirements requires that analysts are able to assert what private data each worker is able to access, not only directly via the inputs of the tasks they perform in a business process, but also indirectly via the chain of tasks that lead to the production of these inputs. In this setting, this paper presents a technique which, given a workflow that transforms a set of input tables into a set of output tables via a set of inter-related SQL statements, determines what information from each input table is disclosed by each output table, and under what conditions this disclosure occurs. The result of this disclosure analysis is a summary representation of the possible computations leading from the inputs of the workflow to a given output thereof.

1 Introduction

Data minimization is one of the principles underpinning the European General Data Protection Regulation (GDPR) as well as previous privacy frameworks and standards such as ISO 29100 [3]. In the context of Business Process Management (BPM) this principle entails that workers, contractors, and other stakeholders involved in the execution of a business process, should only have access to private data to the extent it is required to perform the tasks for which they are responsible. In order to verify compliance vis-a-vis of this requirement, analysts need to have a fine-grained understanding of what private data each worker is able to access, not only directly via the inputs of the tasks they perform, but also indirectly via the chain of tasks that lead to the production of these inputs.

Previous work on business process privacy analysis [1] has led to techniques for boolean ("yes-no") disclosure analysis. These techniques allow an analyst to determine whether or not a given stakeholder has access to a data object or data collection (e.g. a document or a database table). However, it does not allow analysts to determine what part of the data collection (e.g. what attributes) are accessible to each stakeholder and under which conditions.

This paper proposes a finer-grained disclosure analysis technique which characterizes how the contents of the database on top of which a business process is executed, affects each output of the process, specifically, which columns of which tables become part of an output, in which manner, and under which conditions. The proposed technique tasks as input a *SQL workflow*, which we define as a process model in the standard BPMN notation[3] in which each task corresponds to a SQL statement executed against a database. Each SQL statement in the workflow queries a set of input tables from the database and produces new tables, which can be later used by subsequent tasks in the workflow. The table (or set of tables) that are taken as input by the first SQL statements in the workflow are called the inputs. Conversely, the tables produced by the last SQL statements in the workflow are called the final output(s), while the tables produced by intermediate tasks in the workflow are called intermediate outputs.

As a running example, Figure 1 presents an example SQL workflow from an Aid Distribution process, in which a country facing a catastrophe, requests aid from the international community. The situation requires distributing goods to the population via maritime transportation. Henceforth, a SQL workflow is executed to identify ships in nearby locations and to allocate berths to ships, such that ships can move people and goods from/to the requesting country.

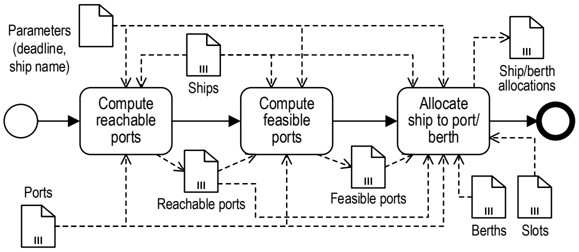

Fig. 1. Conceptual model of the *Aid distribution* scenario

Some of the inputs used in this workflow are confidentail (e.g. ship location and capacities) and the countries involved in the process seek to minimize their exposure to different stakeholders. Accordingly, an analyst needs to determine: (i) who gets access to which input tables during the performance of the process? (ii) what information (e.g. table columns or functions over columns) are disclosed? and (iii) under what conditions this disclosure occurs? The disclosure analysis technique proposed in this paper supports this task by determining what information is disclosed via each intermediate and final output of the workflow, and under which conditions (i.e. for which table rows) this disclosure occurs.

The rest of the paper is structured as follows. Section 2 formalizes the notion of SQL workflow. Section 3 presents the disclosure analysis technique, while Section 4 presents how the output of this technique can be simplified and visually presented. Section 5 discusses related work, while Section 6 draws conclusions.

[3] http://www.bpmn.org/

2 SQL Workflows

For the disclosure analysis, we assume that the overall computation is specified as a set of inter-related SQL statements over a database. Each step takes some input tables and derives new information that is stored in output tables which might be used by subsequent steps. Each task in the SQL workflow is associated with a SQL statement. Listing 1.1 presents the script associated with task "Compute reachable ports" from the running example.

Listing 1.1. SQL script associated with task "Compute reachable ports"

```
1  create function earliest_arrival(
2    ship_latitude double, ship_longitude double,
3    port_latitude double, port_longitude double,
4    max_speed bigint) returns bigint as
5  $$
6  select ceil((point(ship_latitude, ship_longitude)
7              <@> point(port_latitude, port_longitude)) / max_speed)::bigint
8  $$
9  language SQL immutable returns null on null input;
10
11 select port.port_id as port_id,
12   earliest_arrival(ship.longitude, ship.latitude, port.longitude,
13                    port.latitude, ship.maxspeed) as arrival
14 into reachable_ports
15 from ports as port, ships as ship, parameters as p
16 where earliest_arrival(ship.longitude, ship.latitude,
17           port.longitude, port.latitude, ship.maxspeed) <= p.deadline
18   and ship.name = p.shipname
19   and port.port_id = port.port_id;
```

The syntax used in the script is that of PostgreSQL and, as it can be seen, the underlying query is not trivial. In this example, the script includes a user defined function (i.e. earliest_arrival) that computes the time for a ship to reach a port given their coordinates and the ship's speed. Each task can be associated with any number of user-defined functions and at least one *select-into* statement that would store the outcome of the computation on a (temporary) table, to be consistent with the intent specified in the conceptual model. In Listing 1.1, such *select-into* statement is defined in lines 11-19. Moreover, it can be seen that such statement takes as input tables ports, ships and parameters (highlighted in line 15) and stores the result in table reachable_ports (line 14), consistent with the model. The select statement, in turn, calls the function earliest_arrival in lines 12 and 16, which is defined in lines 1-9.

SQL workflows may include sophisticated constructs to captures conditional branching and concurrency, as per the BPMN standard. For simplicity, the disclosure analysis is performed not on the whole SQL workflow but on the set of the runs of it. A run is the set nodes and edges that are visited on a SQL workflow to track one possible execution of the workflow.

To illustrate the concept of run, consider the sample workflow in Figure 2, which contains AND gateways (cf. diamonds decorated with +) and XOR gateways (cf. decorated with ×). The semantics of such gateways, as defined in the BPMN specification [8], is the following. An AND gateway activates all the elements on the outgoing paths and synchronizes the completion of all the elements

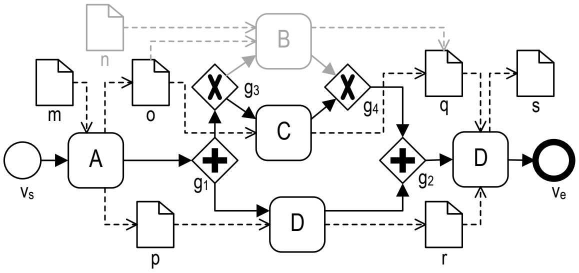

Fig. 2. Sample SQL workflow and its runs

on the incoming paths. Conversely, a XOR gateway activates only one outgoing path (cf. based on a predicated associated with the edge) or waits for the completion of one of the incoming paths. Henceforth, a run of this workflow is a connected subgraph of the process model that contains the start (entry) and the end (exit) node, and such that at most one outgoing edge of each XOR-split is represented. In the example shown in Figure 2, there are exactly two runs, one of which is highlighted.

Before describing the method, we need to introduce some notation. Conceptually, a SQL workflow can be represented as a directed graph, formally defined as tuple $\mathcal{W} = (\mathcal{V}, \mathcal{E}, \mathcal{P}, \mathcal{O}, \mathcal{F}, \mathcal{V}^{\times}, \mathcal{V}^{+}, \mathcal{A}^{D}, \mathcal{A}^{Q})$. There, $\mathcal{V}$ is the set of nodes and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ the set of control flow edges. For convenience, we assume that $\mathcal{V} = \mathcal{P} \cup \mathcal{V}^{\times} \cup \mathcal{V}^{+} \cup \{v_s\} \cup \mathcal{V}_e$, where $\mathcal{P}$ denotes the set of data processing nodes, $\mathcal{G}^{\times}$ the set of AND gateways, $\mathcal{V}^{+}$ XOR gateways, v_s is the start node, and $\mathcal{V}_e$ is a non-empty set of end nodes (e.g. v_e). A SQL workflow must have exactly one start node and at least one end node. $\mathcal{O}$ is the set of data objects and $\mathcal{F} \subseteq (\mathcal{O} \times \mathcal{P}) \cup (\mathcal{P} \times \mathcal{O})$ the set of dataflow edges. Similarly, $\mathcal{A}^{D} : \mathcal{O} \rightarrow SQL$ is a mapping that associates data objects with SQL data definition statements, and $\mathcal{A}^{Q} : \mathcal{P} \rightarrow SQL$ a mapping that associates processing nodes with SQL data manipulation statements. Finally, we will write $\bullet v = \{v' \in \mathcal{V} | (v', v) \in \mathcal{E}\}$ to denote the set of predecessors of node v and $v \bullet = \{v' \in \mathcal{V} | (v, v') \in \mathcal{E}\}$ to refer to the set of successors of v.

Consider the workflow $\mathcal{W}$. We write $\mathrm{P}(\mathcal{W})$ to denote the set of all runs of $\mathcal{W}$, iif for every $\rho \in \mathrm{P}(\mathcal{W})$ with $\rho = (\mathcal{V}', \mathcal{E}')$, all the following conditions hold: (i) ρ is subgraph of $\mathcal{W}$, i.e. $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{E}' = \mathcal{E} \cap (\mathcal{V}' \times \mathcal{V}')$, (ii) ρ includes start/end nodes of $\mathcal{W}$, i.e. $v_s, v_e \in \mathcal{V}'$, (iii) ρ includes exactly one path incoming/outgoing XOR gateways, i.e. $\forall g \in \mathcal{V}' \cap \mathcal{V}^{\times} : | \bullet g \cap \mathcal{V}'| = |g \bullet \cap \mathcal{V}'| = 1$, (iv) ρ includes all paths incoming/outgoing nodes other than XOR gateways, i.e. $\forall g \in \mathcal{V}' \setminus \mathcal{V}^{\times} : \bullet g \subset \mathcal{V}' \wedge g \bullet \subset \mathcal{V}'$, and (v) all the nodes in ρ are in a path from the start node and finishes in at least one of end node v_e, i.e. $\forall v \in \mathcal{V}', \exists v_e \in \mathcal{V}_e : (v_s, v), (v, v_e) \in \mathcal{E}'^{*}$. The set $\mathrm{P}(\mathcal{W})$ can be trivially computed if $\mathcal{W}$ is acyclic, using a tailored depth-first search traversal as presented in [9]. When the input SQL workflow contains loops, it is possible to unroll all the loops one iteration. The method described here can then be applied on the resulting acyclic workflow.

Note that the notion of a run is defined over the control flow edges of the input workflow. However, the data dependencies can be trivially derived for each run ρ by computing the subgraph over dataflow nodes (i.e. data objects) and edges induced by the set of data processing nodes of ρ. Formally, $\mathcal{O}(\rho) \subset \mathcal{O}$ denotes the set of data objects associated with run ρ and is defined as $\mathcal{O}(\rho) = \{o \in \mathcal{O} | \exists v \in \mathcal{V}' : (v, d) \in \mathcal{E}' \vee (d, v) \in \mathcal{E}'\}$.

Finally, given a run ρ of a SQL workflow $\mathcal{W}$, we would need to derive the SQL script that the run would execute. Such a script can be derived from the SQL statements associated with data objects and data processing nodes, by concatenating them following a topological order of the nodes in the run. It is this script that serves as input to the disclosure analysis described below.

3 Disclosure analysis

3.1 Databases, schemas, and queries

Workflow runs (cf. previous section) can be straightforwardly turned to *relational algebra workflows*. These workflows carry the same information, without the syntactic baggage of SQL. These workflows are defined in Fig. 3, also depending on the definitions of relation and database schemas, as stated below.

A *relation schema* is $r(a_1 : D_1, \ldots, a_n : D_n; \mathsf{Dis}_r)$, where r is relation name, $a_1, \ldots, a_n$ are attribute names, $D_1, \ldots, D_n$ are sets, and Dis_r is a set of subsets of the set of attributes $\{a_1, \ldots, a_n\}$. The last component indicates, which attributes or sets of them must be unique in a relation satisfying this schema. An element of Dis_r describes a possible index for a table satisfying the relation schema r. In our analysis, we require Dis_r to contain at least one set of attributes.

Let $D[r]$ denote the set $D_1 \times \cdots \times D_n$. A relation R over the schema r is a subset of $D[r]$, such that for each $\{a_{i_1}, \ldots, a_{i_k}\} \in \mathsf{Dis}_r$ and each $(x_{i_1}, \ldots, x_{i_k}) \in D_{i_1} \times \cdots \times D_{i_k}$ there is at most one $(y_1, \ldots, y_n) \in R$ satisfying $y_{i_1} = x_{i_1}, \ldots, y_{i_k} = x_{i_k}$. Let $\mathcal{X}_r$ denote the set of all relations over the schema r. For $x \in D[r]$, let $x[a_i]$ denote the value of attribute a_i on x.

A *database schema* is $dbs = (t_1 : r_1, \ldots, t_m : r_m)$, where $t_1, \ldots, t_m$ are *table names* and $r_1, \ldots, r_m$ are relation schemas. A database over the schema is a tuple of relations $\mathbf{D} = (R_1, \ldots, R_m)$, where R_i is over r_i. For a fixed *dbs*, let $\mathcal{Y}$ denote the set of all databases over the schema *dbs*, and let $D[t_i]$ denote the set $D[r_i]$. For a database $Y \in \mathcal{Y}$, let $Y.t_i \subseteq D[r]$ denote its table t_i.

Suppose that we have selected the primary keys for each table in the database. That means, for each $t : r$ in the database schema, we have selected $\mathsf{index}_r \in \mathsf{Dis}_r$. We can then think of a relation R over the schema $r(a_1 : D_1, \ldots, a_n : D_n; \mathsf{Dis}_r)$ as a set of partial functions $\mathsf{f}^r_1, \ldots, \mathsf{f}^r_n$ from the cartesian product $\prod_{a_i \in \mathsf{index}_r} D_i$ to each of the sets $D_1, \ldots, D_n$. All these partial functions are defined on the same domain. If $a_i \in \mathsf{index}_r$, then the function f^r_i must be a partial projection.

The syntax for workflows of simple database queries is given in Fig. 3. The workflow is executed against a database with a certain schema *dbs*. The meaning of the syntax for queries Q is the following.

$$
\begin{array}{lllll}
Q ::= t & \mid Q_1 \times \cdots \times Q_k & \mid [Q]_{a \to a'} & & \mid \sigma(Q; e) \\
\quad \mid \ \pi_{a_1,\ldots,a_k}(Q) & \mid \mathsf{col}_{a \leftarrow e}(Q) & \mid \mathsf{let}\ t = Q_1\ \mathsf{in}\ Q_2 & & \mid Q_1 \cup Q_2 \\
\quad \mid \ Q_1 \cap Q_2 & \mid Q_1 \ltimes_e Q_2 & \mid \mathsf{group}^{a_1,\ldots,a_k}_{(a'_1 \bigotimes_1),\ldots,(a'_l,\bigotimes_l)}(Q) & & \\
e ::= a & \mid \otimes(e_1,\ldots,e_k) & & &
\end{array}
$$

Fig. 3. Syntax of queries

- The query t returns the table t. This table must exist in the current database.
- The query $Q_1 \times \cdots \times Q_k$ returns the cartesian product of the results of queries $Q_1, \ldots, Q_k$. We require that the names of the attributes in $Q_1 \times \cdots \times Q_k$ are unique, i.e. the queries $Q_1, \ldots, Q_k$ result in datasets which have non-intersecting sets of attributes.
- $[Q]_{a \to a'}$ executes the query Q. Its result is a relation with a certain schema; this schema must contain attribute a, which is then renamed to a'.
- $\sigma(Q; e)$ filters the result of the query Q with the expression e. The expression e, which must return a Boolean value, is built up from attributes and arithmetic / relational / logical etc. operations $\otimes$. We expect the expressions e to be well-typed, but will not discuss this here any more.
- $\pi_{a_1,\ldots,a_k}(Q)$ projects the result of Q onto attributes $a_1, \ldots, a_k$. The dataset returned by Q must have these attributes in its schema.
- $\mathsf{col}_{a \leftarrow e}(Q)$ runs Q and then adds a new column (a new attribute) to the result. The name of the attribute is a. Its value for each row is computed from the existing attributes of this row according to the expression e.
- $\mathsf{let}\ t = Q_1\ \mathsf{in}\ Q_2$ is used to build workflows. It executes the query Q_1 against the current and gives the resulting dataset the name t. It will then execute the query Q_2 against the database the contains the current database, as well as the the table t.
- $Q_1 \cup Q_2$ and $Q_1 \cap Q_2$ return the union and the intersection of the results of Q_1 and Q_2, which must have the same schema.
- $Q_1 \ltimes_e Q_2$ returns all such rows $\mathbf{r}_1$ from the result of Q_1, such that there exists no row $\mathbf{r}_2$ in the result of Q_2, such that the boolean expression e holds. This construction is used to build outer joins.
- $\mathsf{group}^{a_1,\ldots,a_k}_{(a'_1 \bigotimes_1),\ldots,(a'_l,\bigotimes_l)}(Q)$ expresses grouping and aggregation of the result of Q. The resulting dataset will have attributes $a_1, \ldots, a_k, a'_1, \ldots, a'_l$, with $\{a_1, \ldots, a_k\}$ forming the index. There will be a row with particular values of $a_1, \ldots, a_k$ if the result of Q had at least one row with these values. The attribute a'_i in the query result will be the aggregation by $\bigotimes_i$ of the attributes a'_i in all these rows in the result of Q.

Fig. 3 gives us a rich language for expressing SQL workflows, allowing the use of various types of filters, joins, and projections. Note that the `ORDER BY` component of a SQL statement does not change the resulting relation, hence sorting does not appear among our relational algebra operations. However, sorting may be combined with the `row_number()` function that exists in some SQL dialects. More generally, the row number generation can be done after the dataset has been partitioned according to the values of some other column(s). The row numbers of sorted datasets have been used in the last step of the scenario depicted in Fig. 1. Such use of ordering and row numbers can be modelled with the help of grouping and aggregation.

3.2 Dependency graphs and summaries

A *dependency graph* (DG) is a directed graph $G = (V, E, \mathsf{s}, \mathsf{t}, \ldots)$, where $\mathsf{s}, \mathsf{t} : E \to V$ give the source and the target nodes of arcs. The DG also has the following additional components:

- There are subsets of nodes $I, O \subseteq V$. The in-degree of any node in I and the out-degree of any node in O is 0. The in-degree of any node in O is 1. These nodes represent the inputs coming to, and the outputs produced by the DG.
- There is a set $\mathbf{Op}$ of possible operations. Each *internal* node v (i.e. $v \in V \backslash (I \cup O)$) has a label $\lambda(v) \in \mathbf{Op}$.
- For each internal node v, its incoming arcs are linearly ordered; let $<_v$ denote the ordering relation. The number of incoming arcs of an internal node v is equal to the number of operands that the operation $\lambda(v)$ expects.

Let $\mathbf{V}$ be a set of values; the operations in $\mathbf{Op}$ consume and produce values. Given the semantics $[\![\otimes]\!] : \mathbf{V}^* \to \mathbf{V}$ of each operation $\otimes \in \mathbf{Op}$, the dependency graph G defines a mapping $[\![G]\!] : \mathbf{V}^I \to \mathbf{V}^O$. If G has no directed cycles, then this mapping is defined by assigning a value to each node of G, with the values for input nodes given by the input to $[\![G]\!]$; the values of intermediate nodes v computed by applying $\lambda(v)$ to the values of direct ancestors of v; and the values of output nodes being equal to the values of their direct ancestors. For dependency graphs with directed cycles, the semantics can be defined using a fix-point construction [11], if there is a partial order on $\mathbf{V}$ with the least element $\perp$, and if the operations are monotonic. In this deliverable, we do not have cyclic dependency graphs, hence we will not discuss this any more.

A dependency graph may be infinite, with infinitely many inputs and outputs, as well as with nodes having an infinite number of incoming edges. In the latter case, the operation in the node must make sense for infinite number of inputs (e.g. it may be conjunction or disjunction of booleans). If G is infinite then $[\![G]\!]$ is still well-defined as long as for each output node v_O there is a bound B_O, such that any path in the graph ending in v_O has length at most B_O.

The computations of an SQL workflow can naturally be expressed as infinite dependency graphs. Given a table t with the schema $r(a_1 : D_1, \ldots, a_n : D_n)$ and its index index_r, we express its use in a workflow by the input nodes $v^t_{i,K}$ for each attribute a_i and each possible value K of the index attributes of t. Additionally, the use of the table t is expressed by the input nodes $v^t_{\exists,K}$, denoting whether the row with the index value K is present in the database. As the index attributes typically come from infinite sets (e.g. integers), there are infinitely many possible values K. The input nodes $v^t_{i,K}$ and $v^t_{\exists,K}$ are followed by computation nodes for the expressions e occurring in the workflow. Again, these are replicated as many times as there are possible values for index attributes in the relations that they work on. We end up with a graph with output nodes $w_{j,K'}$ and $w_{\exists,K'}$ for each possible value K' of the index of the resulting dataset. The attributes of the index of the resulting dataset, and hence also the set from which the values K' come from, can be computed from the query as shown in Fig. 4.

Q	index_Q
t	$\prod_{a_i \in \mathsf{index}_r} D_i$, where $r(a_1 : D_1, \ldots, a_n : D_n)$ is the schema of t
$Q_1 \times \cdots \times Q_k$	$\mathsf{index}_{Q_1} \times \cdots \times \mathsf{index}_{Q_k}$
$[Q]_{a \to a'}$	index_Q
$\sigma(Q; e)$	index_Q
$\pi_{a_1,\ldots,a_k}(Q)$	index_Q
$\mathsf{col}_{a \leftarrow e}(Q)$	index_Q
$\mathsf{let}\ t = Q_1\ \mathsf{in}\ Q_2$	index_{Q_2}, where $\mathsf{index}_t \leftarrow \mathsf{index}_{Q_1}$

Fig. 4. Computing the index set of the query

We represent the infinite dependency graphs as finite summaries. The summary dependency graph (SDG) has the same components $(V, E, I, O, \lambda, <)$ as a DG. However, there is additional structure for the nodes and the edges.

- There is a set of possible index sets $\mathcal{S}$. The elements of $\mathcal{S}$ are typically the set of integers, the set of strings, the unit set (a set with a single element). For handling a particular database schema, $\mathcal{S}$ must contain all sets D_i that are associated to some attribute in the index of some table in this schema.
- Each node $v \in V$ has the *dimension* $\mathsf{dim}(v)$ and *input dimension* $\overrightarrow{\mathsf{dim}}(v)$. They are both sets.
  - In our representation, both $\mathsf{dim}(v)$ and $\overrightarrow{\mathsf{dim}}(v)$ are sets that can be expressed as *polynomials* over $\mathcal{S}$. A polynomial over a set of sets $\mathcal{X}$ is a set of the form $\sum_{i=1}^{n} \prod_{j=1}^{m_i} X_{ij}$, where $X_{ij} \in \mathcal{X}$, and $\sum$ denotes the non-intersecting union (or: sum) of sets. Hence there is a finite representation for $\mathsf{dim}(v)$ and $\overrightarrow{\mathsf{dim}}(v)$.
- Each node v has a mapping $\delta(v)$ from $\overrightarrow{\mathsf{dim}}(v)$ to $\mathsf{dim}(v)$.
  - In our representation, the mapping $\delta(v)$ is a *canonical polynomial map*. Let $\overrightarrow{\mathsf{dim}}(v) = \sum_{i=1}^{n} \prod_{j=1}^{m_i} X_{ij}$ and $\mathsf{dim}(v) = \sum_{i=1}^{s} \prod_{j=1}^{t_i} Y_{ij}$. A canonical polynomial map is built up from identity mappings between X_{ij} and $Y_{i'j'}$ (which must be the same set) as follows:
    - ∗ A canonical mapping $c : \prod_{j=1}^{m} X_j \to \prod_{j=1}^{t} Y_j$ is defined by an injective mapping $\gamma : \{1, \ldots, t\} \to \{1, \ldots, m\}$ satisfying $X_{\gamma(j)} = Y_j$ for all $j \in \{1, \ldots, t\}$. The mapping c is given by

$$c((x_1, \ldots, x_m)) = (x_{\gamma^{-1}(1)}, \ldots, x_{\gamma^{-1}(t)}) \enspace .$$

    - ∗ A canonical mapping from $\prod_{j=1}^{m} X_j$ to $\sum_{i=1}^{s} \prod_{j=1}^{t_i} Y_{ij}$ consists of an index $q \in \{1, \ldots, s\}$ and a canonical mapping of the previous kind from $\prod_{j=1}^{m} X_j$ to $\prod_{j=1}^{t_q} Y_{qj}$.

        * A canonical mapping from $\sum_{i=1}^{n}\prod_{j=1}^{m_i} X_{ij}$ to $\sum_{i=1}^{s}\prod_{j=1}^{t_i} Y_{ij}$ consists of n canonical mappings of the previous kind.
    - If $\delta(v)$ is not the identity mapping, then the node v must have exactly one incoming arc.
- Each arc $\alpha \in E$ still has a single target node $\mathsf{t}(\alpha)$. But an arc may have several source nodes, i.e. $\mathsf{s}(\alpha) \subseteq V$.
- Each arc $\alpha \in E$ has a mapping $\overline{\delta}(\alpha)$ from $\overrightarrow{\mathsf{dim}}(\mathsf{t}(\alpha))$ to $\sum_{v\in\mathsf{s}(\alpha)} \mathsf{dim}(v)$.
    - Mapping $\overline{\delta}(\alpha)$ is again a canonical polynomial map.

A summary dependency graph G_{sum} is expanded to a potentially infinite dependency graph $G = \mathsf{expand}(G_{\mathrm{sum}})$ in the following manner:

- For each node v in the summary dependency graph, there are nodes $\{(v,x)\,|\,x \in \mathsf{dim}(v)\}$ in the actual dependency graph, which have the same operation $\lambda(v)$.
    - We call the node (v,x) in the actual dependency graph the *instance* x of the node v in the SDG.
- For each arc α going to a vertex v in the summary dependency graph, and for each element $x \in \overrightarrow{\mathsf{dim}}(v)$, there is an edge from the node $\overline{\delta}(\alpha)(x)$ to the node $\delta(v)(x)$. Note that the output of $\overline{\delta}(\alpha)(x)$ is a pair of some node $w \in \mathsf{s}(\alpha)$ and a value $y \in \mathsf{dim}(w)$.
    - Let $x \in \mathsf{dim}(v)$. If $\delta(v)$ is the identity mapping and thus $v \in G_{\mathrm{sum}}$ may have several input arcs, the ordering $<_{(v,x)}$ of the inputs of the vertex $(v,x) \in G$ is inherited from v. The vertex (v,x) has the same number of input arcs as the vertex v does.
    - Otherwise, the vertices $(v,x) \in G$ may have any number of inputs, perhaps an infinite number. In this case, $\lambda(v)$ must be an associative and commutative operation, and make sense for infinite number of inputs.

In our analysis, we translate an SQL workflow into a summary dependency graph. The semantics of a summary dependency graph is the same as the semantics of the dependency graph resulting from its expansion. This semantics can be related to the semantics of the SQL workflow in a manner that shows their equivalence. We simplify the summary dependency graph, removing spurious dependencies, while changing the semantics of the graph only in a manner that still relates it to the SQL workflow. From the resulting graph, we can read out the actual dependencies of each output, including the actual computation, as well as the conditions of outputting them.

The translation of a query Q to a summary dependency graph works in syntax-directed manner. We first translate the database schema, resulting in a Partial Summary Dependency Graph (PSDG) consisting of only input nodes. Beside the PSDG, we also get a mapping from the attributes of tables to the nodes. This PSDG is given as the input to the translation of Q. The result is another PSDG, which is post-processed to add the output nodes. The translation is given in App. A. Fig. 5 shows the result of translating the workflow consisting of Listing 1.1, followed by the query

```
select rport.port_id, port.name,
  earliest_arrival(ship.longitude, ship.latitude, port.longitude,
                   port.latitude, ship.maxspeed) as arrival
from reachable_ports as rport, port, ship, parameters as p
where port.port_id = rport.port_id
  and ship.name = p.shipname
```

into a SDG. We have removed dead nodes, and identity nodes from this figure. In this figure, the rectangles with sharp corners denote the nodes of SDG. In the top row, it lists the name of the operation and the ID of the node v. The following rows list the components of $\mathsf{dim}(v)$, these components are elements of $\mathcal{S}$. An arc α, where $\overline{\delta}(\alpha)$ is the identity mapping, is depicted as line ending in an arrow, possibly with a short label in the middle, indicating the position of the value flowing along this arc in the operation at $\mathsf{t}(\alpha)$. If $\overline{\delta}(\alpha)$ is not identity, then it is depicted inside a rectangle with rounded corners. At the top of this rectangle is the label of the arc (if any), and other rows show, which dimension components of the target node correspond to which dimension components of the source node.

4 Simplifications and output presentation

4.1 Simplifying the SDG

We have implemented a number of simplifications of SDG, both structural and semantical. Below we discuss these simplifications on the basis of the full scenario depicted in Fig. 1. A simplification operation, applied to a certain node or a group of nodes, checks whether the local context of these nodes matches some pattern. If it does, then these nodes are replaced with some other nodes that have the same effect semantically (or an effect that is similar in the view of our task to find which inputs end up where, how, and when), but have simpler structure.

The SDG is a very helpful data structure in determining the applicability of simplifications. The applicability of many simplifications can be determined locally, i.e. by considering a subgraph of bounded diameter. Also, more complex structural transformations have applicability checks which consist of simple traversals of the graph. Hence the current set of simplifications may be easily extended, depending on the needs of analysed scenarios.

One simplification may enable others. We thus run the simplifications in the order that seems to make the most sense; some simplifications (e.g. the removal of dead nodes) are run many times. In the following, we will describe some simplifications that our analyzer currently runs.

Removal of dead nodes. A node that has no descendants may be removed, unless it is an output node. Running this removal many times, we will remove all nodes that are not backwards reachable from any output node.

Folding of identity operations. An ID node (the node whose operation is identity; our translation from relational algebra expressions, given in App. A, produces many such nodes) can be cut out of paths: if v is an ID node and

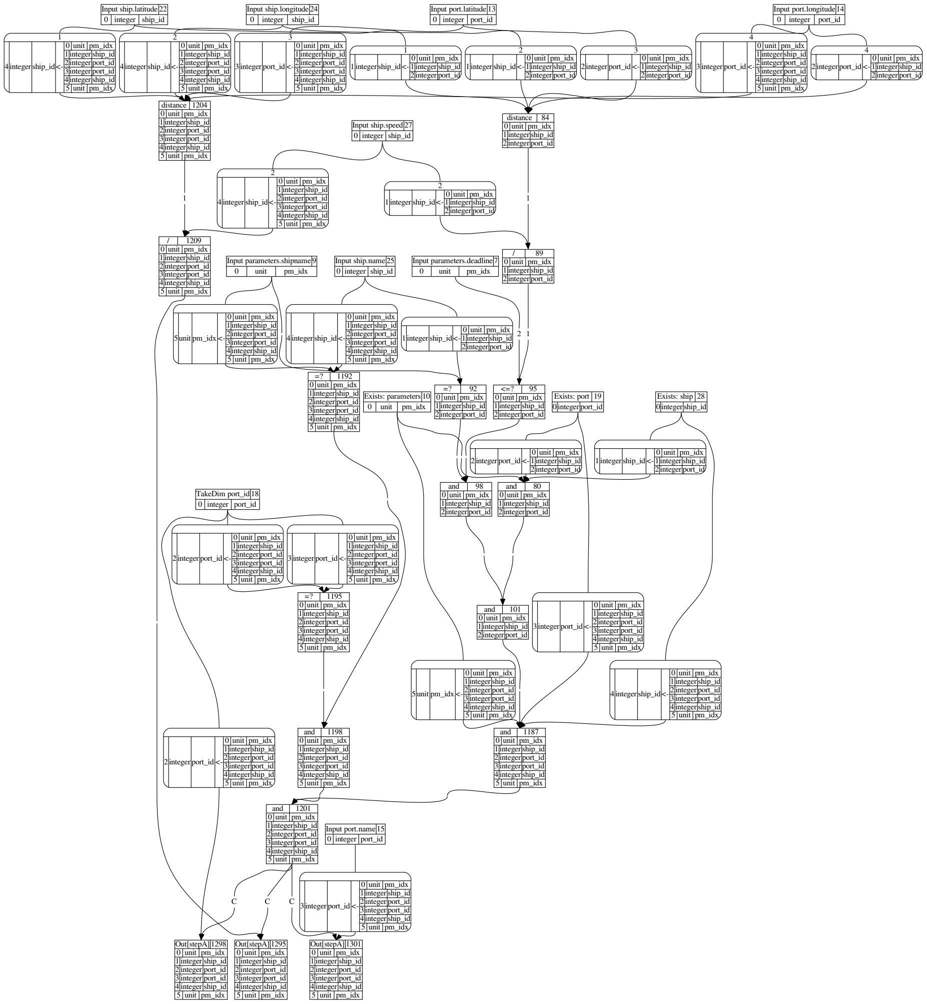

Fig. 5. Initial SDG

α is the arc leading to it, and β is any arc with the source v, then β may be replaced with the arc $\beta \circ \alpha$: we define $\mathsf{s}(\beta \circ \alpha) = \mathsf{s}(\alpha)$, $\mathsf{t}(\beta \circ \alpha) = \mathsf{t}(\beta)$ and $\overline{\delta}(\beta \circ \alpha) = \overline{\delta}(\alpha) \circ \overline{\delta}(\beta)$, assuming that $\delta(v)$ is the identity mapping (which is always the case in the SDGs that we construct). After all arcs leaving v have been replaced, v is dead and can be removed by the previous simplification.

Splitting nodes with sum dimensions. A node v with $\mathsf{dim}(v) = \sum_{i=1}^{n} \prod_{j=1}^{m_i} X_{ij}$, where $n > 1$, is replaced with n nodes having the same operation, each corresponding to one component of $\mathsf{dim}(v)$. This transformation makes subsequent structural simplifications easier to apply.

Folding the "&"-nodes. If v and v' are both computing boolean conjunctions, and there is an arc α from v' to v, then we add arcs from all predecessors of v' to v (with the correct $\overline{\delta}(\cdot)$-mapping) and remove the arc α. If there were no other arcs leaving v', then it is dead.

Joining nodes with identical computation. If two nodes have the same operation and the same inputs, they can be turned to a single node. In our SDGs, the recognition of these nodes is complicated by the need to determine if a suitable isomorphism between their dimensions exists.

Reducing the dimension of a node. In our SDG-s, the dimensions of nodes are products of elements of $\mathcal{S}$. If for some node v in SDG, the predecessors of the nodes corresponding to v in the infinite dependency graph do not depend on some component of the elements in $\mathsf{dim}(v)$, then this component may be removed from $\mathsf{dim}(v)$.

Joining components of dimensions. Let v be a node that computes a boolean result, and let $\mathsf{dim}(v) = \prod_{i=1}^{n} X_i$. Suppose that we have deduced that there are indices $i, j \in \{1, \ldots, n\}$, such that a node $(v, (x_1, \ldots, x_n))$ in the expanded dependency graph may be true only if $x_i = x_j$. This may happen in a workflow that creates complex joins of tables, joining the same table many times while requiring the primary keys to be equal; we use these equality checks to deduce that v implies $x_i = x_j$. It may also happen due to uniqueness constraints on attributes, when conjunctions of several comparisons involving these attributes are formed. If we have identified that the i-th and the j-th component of $\mathsf{dim}(v)$ have to be equal for v to be true, and when v being false only implies that certain outputs are not made, then we can identify these components and thereby reduce the dimension of v. This reduction works differently from the previous simplification, and has to be propagated along the SDG in both directions.

Arithmetic simplifications. A conjunction with a single input, or a sum with a single input can be turned to an ID node. A conjunction with a FALSE-input can be turned to FALSE-node (with no inputs). A `COALESCE`-operation can also be simplified if we know that some of its arguments certainly are, or certainly are not NULL.

Fig. 6 depicts the results of the simplifications applied to the SDG in Fig. 5.

4.2 Presenting the result of the analysis

The dependencies and conditions are depicted in our final, simplified SDG, but they are not given in terms of certain rows existing or not existing in the tables of the database. To present the outcome, we have to map from the product of elements of $\mathcal{S}$ back into tables. Let $v_\bullet$ be a particular output node, for which we are interested in the computation of the value it outputs, as well as in the condition that must be satisfied for the output to take place. We perform the following steps for obtaining the description of the outputs from $v_\bullet$.

– First, we remove all output nodes except $v_\bullet$ from the SDG, and remove all dead nodes from it. After that, we will transform the directed acyclic SDG

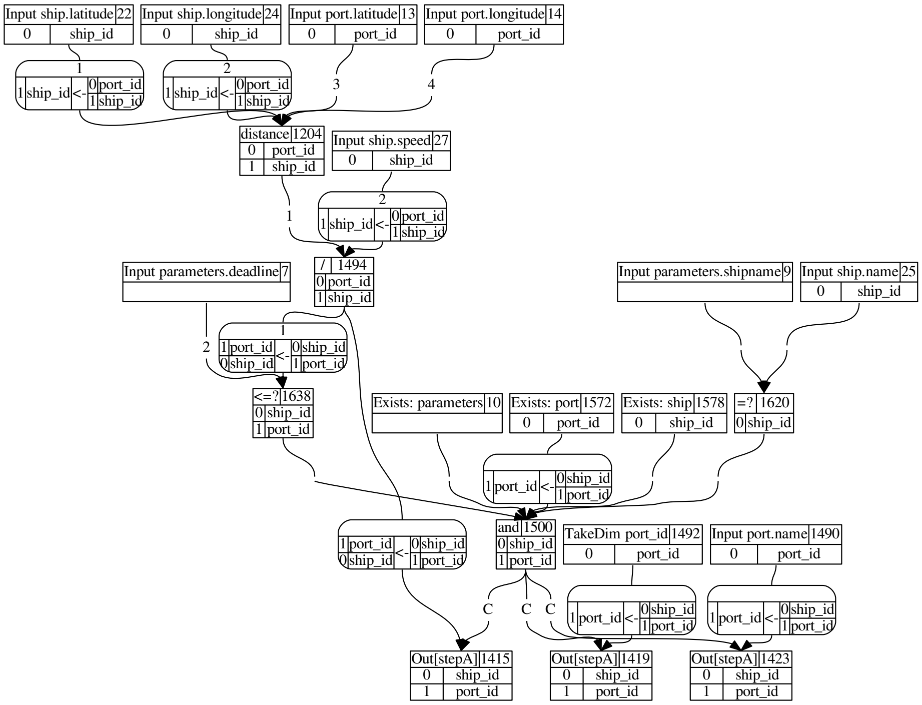

Fig. 6. Final SDG

into a tree T, by duplicating nodes with several outgoing arcs. The root of T is $v_\bullet$. The leaves of T are the input nodes, referring to a particular attribute in a particular table.

- Let $\mathcal{PC}$ be the set of all *dimension components* (i.e. the elements of $\mathcal{S}$) of all nodes in T, formally

$$\mathcal{PC} = \bigcup_{v \in V(T)} \left(\{(v, i, X_i) \mid \mathsf{dim}(v) = \prod_{i=1}^{n} X_i\} \cup \{(v, -i, X_i) \mid \overrightarrow{\mathsf{dim}}(v) = \prod_{i=1}^{n} X_i\} \right) .$$

  Let $\mathcal{C}$ be the set $\mathcal{PC}$ factored by an equivalence relation generated by the $\delta(\cdot)$-mappings of all vertices and the $\overline{\delta}(\cdot)$-mappings of all arcs in T. The set $\mathcal{C}$ is the inventory of all *different* dimension components that occur in T.
- Each input node refers to a table, and its dimension refers to some elements of $\mathcal{C}$. The inputs nodes with the same table and the same elements of $\mathcal{C}$ correspond to the same row of the table. We replace the input nodes, and forget the dimensions and their maps of the internal nodes and arcs. For input nodes with partially overlapping sets of elements of $\mathcal{C}$, we introduce the equality checks of the respective components of the table rows, which must be satisfied for the node $v_\bullet$ to output anything.

The result, when $v_\bullet$ is the node with ID 1415 in Fig. 6, is depicted in Fig. 7.

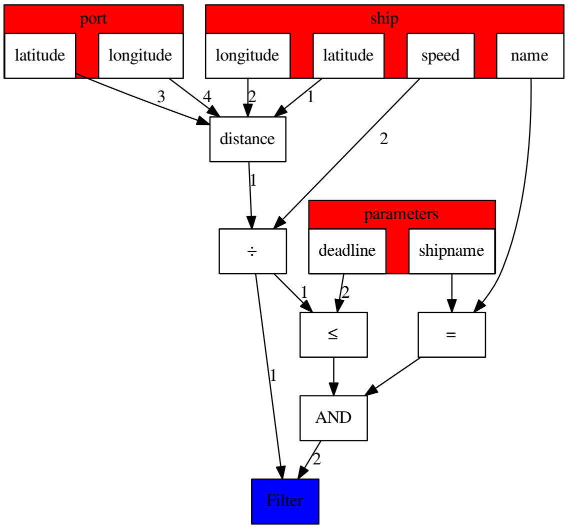

Fig. 7. Representation of the computations

5 Related work

One of the most prominent examples of methods to quantify the potential disclosure of information is that of differential privacy, which has been widely studied in the context of program analysis, using e.g. types [5] or theorem proving [2]. These techniques allow one to reason about the theoretical bounds of the amount information revealed by a program on its output relative to its input. In a similar vein, techniques have been proposed to analyze sensitivity and differential privacy for database queries expressed in SQL [6], and other SQL-like languages (e.g. PINQ) [7]. Here, the reasoning on sensitivity is formulated in terms of individual database queries and the effects on the output of those queries with respect to variations on the input tables. In recent work [4, 10], we have extended the results on differential privacy to reason about not only one single computation step, but to to assess the overall differential privacy of a data processing workflows, which require the aggregation of the sensitivity of the steps in a workflow that can be observed by one stakeholder. The goal of works on differential privacy is the to derive theoretical bounds of the amount of information that a stakeholder can infer from the outputs of programs or steps on a workflow. Conversely, in this work we look not at quantifying the disclosure of information but rather providing an insight on what is disclosed and on the conditions that must hold for that disclosure to happen.

Also close to our setting is the work on information leak detection on business process models reported in [1]. Such method takes as input workflows on which tasks have been classified in levels of confidentiality, which can be either high or low. By using a reachability analysis, the method is capable of identifying structures on the workflow (e.g. sequencing of tasks, mutual exclusion, etc) where information may be leaked to stakeholders, when changes between domains of confidentiality are not properly guarded. In contrast, our method considers the underlying computation (e.g. SQL code) and identifies what information as well as the conditions that will be revealed after having executing a SQL workflow.

6 Conclusions and Future Work

The paper presented an analysis technique to determine what information from each input table is disclosed by each output table of a SQL Workflow, and under what conditions this disclosure occurs. The proposed technique has been implemented on top of the Pleak open-source business process privacy analysis toolset. The source code of the toolset is available at https://github.com/pleak-tools while a demonstrator is available at http://pleak.io/.

The current technique operates over unprotected workflows, meaning workflows that do not make use of any Privacy-Enhancing Technologies (PETs) such as multi-party computation, encryption, or differential privacy. In future work, we plan to extend the technique to take as input workflows where some of the tasks have PETs attached to them. This extension would allow analysts to perform "what-if" privacy analysis. Concretely, an analyst would be able to see how the addition, removal, or modification of a PET in a workflow affects the information that is disclosed to different parties.

Another extension is the ability to compare a disclosure report against a privacy policy. This capability would allow an analyst to determine what additional PETs could be added to a given process in order to fulfill a privacy policy.

Acknowledgments. This research was funded by the Air Force Research laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under contract FA8750-16-C-0011. The views expressed are those of the author(s) and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

1. Rafael Accorsi, Andreas Lehmann, and Niels Lohmann. Information leak detection in business process models: Theory, application, and tool support. *Inf. Syst.*, 47:244–257, 2015.
2. Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. *ACM Trans. Program. Lang. Syst.*, 35(3):9, 2013.
3. Michael Colesky, Jaap-Henk Hoepman, and Christiaan Hillen. A critical analysis of privacy design strategies. In *IEEE Security and Privacy Workshops (SP)*, pages 33–40. IEEE Computer Society, 2016.
4. Marlon Dumas, Luciano García-Bañuelos, and Peeter Laud. Differential privacy analysis of data processing workflows. In *Proc. of GraMSec 2016*, volume 9987 of *LNCS*, pages 62–79. Springer, 2016.

5. Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *Proc. of POPL 2013*, pages 357–370. ACM, 2013.
6. Noah Johnson, Joseph P. Near, and Dawn Song. Towards practical differential privacy for sql queries. *Proc. VLDB Endow.*, 11(5):526–539, January 2018.
7. Frank McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proc. of SIGMOD 2009*, pages 19–30. ACM, 2009.
8. OMG. Business Process Model and Notation (BPMN), Version 2.0. Technical report, Object Management Group, January 2011.
9. Sinnakkrishnan Perumal and Ambuj Mahanti. A graph-search based algorithm for verifying workflow graphs. In *Proc. of DEXA'05*, pages 992–996. IEEE Computer Society, 2005.
10. Martin Pettai and Peeter Laud. Combining differential privacy and mutual information for analyzing leakages in workflows. In *Proc. of ETAPS 2017*, volume 10204 of *LNCS*, pages 298–319. Springer, 2017.
11. Ilja Tšahhirov and Peeter Laud. Application of Dependency Graphs to Security Protocol Analysis. In *Proc. of TGC 2007*, volume 4912 of *LNCS*, pages 294–311. Springer, 2008.

A Translating SQL workflows to internal representation

The translation of a query Q to a summary dependency graph (SDG) proceeds by first translating the database schema, then performing the syntax-directed translation of the actual query Q, followed by the addition of output nodes. We call the intermediate graphs *Partial Summary Dependency Graphs* (PDSG), where the partiality indicates the lack of output nodes.

Let G be a PSDG and consider a relation schema r with attributes $a_1, \ldots, a_n$. A *representation of* r *in* G is a mapping $R : \{\exists, a_1, \ldots, a_n\} \rightarrow V(G)$, such that $\mathsf{dim}(R(\exists)) = \mathsf{dim}(R(a_1)) = \cdots = \mathsf{dim}(R(a_n))$, the output type of each $R(a_i)$ matches with the type of a_i, and the output type of $R(\exists)$ is boolean. We write $\mathsf{dim}(R)$ for $\mathsf{dim}(R(\exists))$. A *representation* of a database schema *dbs in* G is a mapping from the contained relations into their representations in G.

Translating a database schema. The translation of a database schema *dbs* returns a PSDG G_{dbs}, as well as a representation R_{dbs} of *dbs* in it. These are the following:

- Let $t : r$ be a table declaration in *dbs*, where r is the relation schema $r(a_1 : D_1, \ldots, a_n : D_n; \mathsf{index}_r)$, with certain attributes belonging to the index. W.l.o.g. let $a_1, \ldots, a_h$ be the index attributes. The graph G will contain nodes $v^t_\exists$ and v^t_i for $1 \leq i \leq n$. The input dimension and the dimension of all nodes is $\mathcal{I} = \prod_{i=1}^{h} D_i$. All nodes are input nodes. During the execution, the instance $(x_1, \ldots, x_h)$ of the node v^t_i is supposed to carry the value of the attribute a_i in the row of the table t that corresponds to the index value $(a_1 = x_1, \ldots, a_h = x_h)$. The instance $(x_1, \ldots, x_h)$ of the node $v^t_\exists$ carries the value true iff the table t has a row with index value $(a_1 = x_1, \ldots, a_h = x_h)$.
- The representation R_{dbs} maps each table t to the mapping $\{\exists \mapsto v^t_\exists\} \cup \{a_i \mapsto v^t_i \mid 1 \leq i \leq |t|\}$.

Translating the query. The translation $\mathcal{G}$ of a query Q against a database with schema *dbs* takes as input a PSDG $G_\circ$ and a representation R_{dbs} of *dbs* in it. It returns a new PSDG $G_\bullet$ (which is obtained from $G_\circ$ by adding zero or more nodes to it) and a representation of $\mathbf{attr}(Q)$ in $G_\bullet$, where $\mathbf{attr}(Q)$ is the schema of the output relation of Q.

The translation $\mathcal{G}$ may call the translation $\mathcal{E}$ for expressions e. It takes as input a PSDG $G_\circ$ and a representation R of a relation schema in $G_\circ$. This relation schema must contain all attributes used by e. The translation $\mathcal{E}$ returns a new PSDG $G_\bullet$ and a node $v_e \in V(G_\bullet)$. The translation $\mathcal{E}$ works as follows.

- $\mathcal{E}[\![a]\!](G_\circ, R)$ returns $G_\circ$ and $R(a)$.
- $\mathcal{E}[\![\otimes(e_1, \ldots, e_k)]\!](G_\circ, R)$ calls $\mathcal{E}[\![e_1]\!], \ldots, \mathcal{E}[\![e_k]\!]$ one after another. Let the output of $\mathcal{E}[\![e_i]\!]$ be G_i and v_i. Then the inputs to $\mathcal{E}[\![e_i]\!]$ are G_{i-1} (with $G_0 \equiv G_\circ$) and R. After obtaining G_k, add a new node v to the graph. Its label is $\otimes$, and its dimension and input dimension are both $\mathsf{dim}(R)$. Also add arcs $\alpha_1, \ldots, \alpha_k$ to the graph, going from nodes $v_1, \ldots, v_k$ to the node v. For all i, the mapping $\overline{\delta}(\alpha_i)$ is equal to the identity map on $\mathsf{dim}(R)$. Return the modified graph G_k and the vertex v.

The translation $\mathcal{G}$ works as follows.

- $\mathcal{G}[\![t]\!](G_\circ, R_{dbs})$ returns $G_\circ$ and $R_{dbs}(t)$.
- $\mathcal{G}[\![Q_1 \times \cdots \times Q_k]\!](G_\circ, R_{dbs})$ calls $\mathcal{G}[\![Q_1]\!], \ldots, \mathcal{G}[\![Q_k]\!]$ one after another. Let the output of $\mathcal{G}[\![Q_i]\!]$ be G_i and R_i^Q. Then the inputs to $\mathcal{G}[\![Q_i]\!]$ are G_{i-1} (with $G_0 \equiv G_\circ$) and R_{dbs}. After obtaining G_k and $R_1^Q, \ldots, R_k^Q$, we add the following nodes and arcs to G_k:
  - Let $\mathcal{I} = \prod_{i=1}^{k} \mathsf{dim}(R_i^Q)$.
  - Add a node $v_\exists$. The label of this node is "&" (boolean conjunction). Its dimension and input dimension are both $\mathcal{I}$.
  - For each $i \in \{1, \ldots, k\}$ add an arc $\alpha_{\exists,i}$ from the node $R_i^Q(\exists)$ to $v_\exists$. The mapping $\overline{\delta}(\alpha_{\exists,i})$ is the canonical projection from $\mathcal{I}$ to its i-th component $\mathsf{dim}(R_i^Q)$.
  - For each $i \in \{1, \ldots, k\}$ and each attribute $a_j \in \mathbf{attr}(Q_i)$ add a node $v_{i,j}$. The label of this node is "ID" (the identity mapping). Its dimension and input dimension are both $\mathcal{I}$.
  - Also, add an arc $\alpha_{i,j}$ from $R_i^Q(a_j)$ to $v_{i,j}$. The mapping $\overline{\delta}(\alpha_{i,j})$ is the canonical projection from $\mathcal{I}$ to its i-th component $\mathsf{dim}(R_i^Q)$.

  Let the output PSDG $G_\bullet$ be the modified graph G_k. The output representation R maps $\exists$ to $v_\exists$ and the attribute a_j in $\mathbf{attr}(Q_i)$ to $v_{i,j}$.
- $\mathcal{G}[\![[Q]_{a \to a'}]\!](G_\circ, R_{dbs})$ runs $(G_\bullet, R) = \mathcal{G}[\![Q]\!](G_\circ, R_{dbs})$. It returns $G_\bullet$ and $R[a' \mapsto R(a)]$.
- $\mathcal{G}[\![\sigma(Q; e)]\!](G_\circ, R_{dbs})$ runs $(G', R) = \mathcal{G}[\![Q]\!](G_\circ, R_{dbs})$ and $(G'', v_?) = \mathcal{E}[\![e]\!](G', R)$. It adds a node $v_\exists$ to G''. The label of this node is "&" and both its dimension and input dimension are $\mathsf{dim}(R)$. The node $v_\exists$ has two inputs, from $R(\exists)$ and from $v_?$. The $\overline{\delta}(\cdot)$-mappings of both respective arcs are the identity mappings over $\mathsf{dim}(R)$. Let $G_\bullet$ be the modified graph G''. The translation returns $G_\bullet$ and $R[\exists \mapsto v_\exists]$.
- $\mathcal{G}[\![\pi_{a_1,\ldots,a_k}(Q)]\!](G_\circ, R_{dbs})$ runs $(G_\bullet, R) = \mathcal{G}[\![Q]\!](G_\circ, R_{dbs})$. It returns $G_\bullet$ and R restricted to $\{\exists, a_1, \ldots, a_k\}$.
- $\mathcal{G}[\![\mathsf{col}_{a \leftarrow e}(Q)]\!](G_\circ, R_{dbs})$ runs $(G', R) = \mathcal{G}[\![Q]\!](G_\circ, R_{dbs})$ and $(G_\bullet, v_e) = \mathcal{E}[\![e]\!](G', R)$. It returns $G_\bullet$ and $R[a \mapsto v_e]$.
- $\mathcal{G}[\![\mathsf{let}\ t = Q_1\ \mathsf{in}\ Q_2]\!](G_\circ, R_{dbs})$ runs $(G', R_0) = \mathcal{G}[\![Q_1]\!](G_\circ, R_{dbs})$, followed by $(G_\bullet, R) = \mathcal{G}[\![Q_2]\!](G', R_{dbs}[t \mapsto R_0])$. It returns $G_\bullet$ and R.
- $\mathcal{G}[\![Q_1 \cup Q_2]\!](G_\circ, R_{dbs})$ runs

  $$\begin{aligned}(G', R') &= \mathcal{G}[\![Q_1]\!](G_\circ, R_{dbs}) \\ (G'', R'') &= \mathcal{G}[\![Q_2]\!](G', R_{dbs})\ .\end{aligned}$$

  For each attribute $a \in \mathbf{attr}(Q_1) = \mathbf{attr}(Q_2)$ it will then add a node v_a to G'', with the operation "ID" and its dimension and input dimension both being equal to $\mathsf{dim}(R') + \mathsf{dim}(R'')$. The mapping $\delta(v_a)$ is the identity mapping. The node v_a has a single incoming arc α_a, which has *two* sources — $R'(a)$ and $R''(a)$. The mapping $\overline{\delta}(\alpha_a)$ is the identity mapping from $\overrightarrow{\mathsf{dim}}(v_a)$ to $\mathsf{dim}(R'(a)) + \mathsf{dim}(R''(a))$.

We also add a node $v_\exists$ to the graph G'' with the same dimension, input dimension and $\delta(\cdot)$-mapping as described in the previous paragraph. The operation in this node is again "ID" (boolean disjunction), and it again has a single incoming arc $\alpha_\exists$ with two sources: $R'(\exists)$ and $R''(\exists)$, with the mapping $\overline{\delta}(()\alpha_\exists)$ again being the identity map.

Let the output PDSG $G_\bullet$ be the graph G'' with the added nodes and arcs. The output representation R maps $\exists$ to $v_\exists$ and each attribute a to v_a.

- $\mathcal{G}[\![Q_1 \cap Q_2]\!](G_\circ, R_{dbs})$ runs

  $$(G', R') = \mathcal{G}[\![\sigma(Q_1 \times [Q_2]_{a:\mathbf{attr}(Q_2)\to a'}; \bigwedge_{a\in\mathbf{attr}(Q_1)} a = a')]\!](G_\circ, R_{dbs})$$

  first, while also keeping the representation R_1 that was produced while $\mathcal{G}[\![Q_1]\!](G_\circ, R_{dbs})$ was run as a subroutine. Here the write-up $[Q_2]_{a:\mathbf{attr}(Q_2)\to a'}$ denotes that we have renamed all attributes a of Q_2 into their primed versions.

  We add to G' a node $v_\exists$ with the operation "$\bigvee$" (boolean disjunction). We let $\mathsf{dim}(v_\exists) = \mathsf{dim}(R_1)$ and $\overrightarrow{\mathsf{dim}}(v_\exists) = \mathsf{dim}(R')$. Recall that $\mathsf{dim}(R')$ is equal to the Cartesian product of $\mathsf{dim}(R_1)$ and the dimension of the nodes resulting from the translation of the query Q_2. The mapping $\delta(v_\exists)$ is the natural projection to the first component of this product.

  As $\mathsf{dim}(v_\exists) \neq \overrightarrow{\mathsf{dim}}(v_\exists)$, this node may have a single incoming arc. This arc comes from the node $R'(\exists)$, its $\overline{\delta}(\cdot)$-mapping is the identity mapping.

  We return the graph G' with the extra node and arc. As the output representation, we return $R_1[\exists \mapsto v_\exists]$.
- $\mathcal{G}[\![Q_1 \ltimes_e Q_2]\!](G_\circ, R_{dbs})$ runs

  $$(G', R_2) = \mathcal{G}[\![Q_1 \times Q_2]\!](G_\circ, R_{dbs})$$
  $$(G'', v_e) = \mathcal{E}[\![e]\!](G', R_2)\ \ .$$

  We also keep the representation R_1 that was produced when $\mathcal{G}[\![Q_1]\!](G_\circ, R_{dbs})$ was run as a subroutine. After that, we add the following nodes and arcs to G''.
  - Node v_1, operation "&", with dimension and input dimension equal to $\mathsf{dim}(R_2)$. Its inputs are v_e and $R_2(\exists)$.
  - Node v_2, operation "$\bigvee$". Its dimension is equal to $\mathsf{dim}(R_1)$ and its input dimension to $\mathsf{dim}(R_2)$. The mapping $\delta(v_2)$ is the natural projection from the second to the first. The input to v_2 is the node v_1.
  - Node v_3, operation "NOT". Its dimension and input dimension are equal to $\mathsf{dim}(R_1)$. Its input is the node v_2.
  - Node v_4, operation "&". Its inputs are v_3 and $R_1(\exists)$.

  For all arcs described above, their $\overline{\delta}(\cdot)$-mapping is the identity mapping. The translation returns the PSDG G'' together with added nodes and arcs. As the output representation, it returns $R_1[\exists \mapsto v_4]$.
- $\mathcal{G}[\![\mathsf{group}^{a_1,\ldots,a_k}_{(a'_1\bigotimes_1),\ldots,(a'_l,\bigotimes_l)}(Q)]\!](G_\circ, R_{dbs})$ first runs $(G', R') = \mathcal{G}[\![Q]\!](G_\circ, R_{dbs})$. It will determine the types $D_1,\ldots,D_k$ of the attributes $a_1,\ldots,a_k$ of Q. These types must be elements of $\mathcal{S}$. The following nodes and arcs are then added to G':

- Nodes $v_1^{\mathrm{TD}},\dots,v_k^{\mathrm{TD}}$. These are input nodes of the SDG. The dimension of v_i^{TD} is D_i. In the infinite dependency graph, a node v corresponding to the value $x \in D_i$ and the node v_i^{TD}, is expected to carry the value x. Let $\mathcal{I} = D_1 \times \cdots \times D_k$.
- Nodes $v_1^{=},\dots,v_k^{=}$. The operation of these nodes is "=" (equality check). The dimension and input dimension of these nodes is $\mathsf{dim}(R') \times \mathcal{I}$. The node $v_i^{=}$ has two inputs: v_i^{TD} and $R'(a_i)$. The $\overline{\delta}(\cdot)$-mappings for the arcs connecting these nodes are the natural projections.
- Node $v^{=}$. The operation of this node is "&". Its dimension and input dimension are both $\mathsf{dim}(R') \times \mathcal{I}$. Its inputs are the nodes $v_1^{=},\dots,v_k^{=}$.
- Node $v_\exists$. The operation of this node is "$\bigvee$". Its dimension is $\mathcal{I}$ and its input dimension is $\mathsf{dim}(R') \times \mathcal{I}$. The mapping $\delta(w_\exists)$ is the natural projection. Node $v_\exists$ receives its input from $v^{=}$.
- Nodes $v_1^f,\dots,v_l^f$. The operation of these nodes is "Output"; this operation takes two arguments and returns the first one only if the second one is true. Their dimension and input dimension are $\mathsf{dim}(R') \times \mathcal{I}$. The inputs of the node v_j^f are $v^{=}$ (for the first, "conditioning" argument) and $R'(a'_j)$ (for the second, "value" argument). The $\overline{\delta}(\cdot)$-mapping for the arc connecting to the first input is the identity mapping, while for the arc connecting to the second input is the natural projection from $\mathsf{dim}(R') \times \mathcal{I}$ to $\mathsf{dim}(R')$.
- Nodes $v_1^{\otimes},\dots,v_l^{\otimes}$. The operation of the node $v_j^{\otimes}$ is "$\bigotimes_j$". The dimension of $v_j^{\otimes}$ is $\mathcal{I}$, while its input dimension is $\mathsf{dim}(R') \times \mathcal{I}$. The mapping $\delta(v_j^{\otimes})$ is the natural projection. The input to the node $v_j^{\otimes}$ is the node v_j^f.

We see that the expansions of the nodes $v_j^{\otimes}$ in the infinite dependency graph perform the actual aggregations of the values of the dataset resulting from the query Q. We have implicitly assumed that the NULL-values among the inputs of the operations $\bigotimes_j$ do not change their output value.

The translation returns the graph G' together with the added nodes and arcs. The output representation R is the following:

- $R(\exists) = w_\exists$;
- $R(a_i) = v_i^{\mathrm{TD}}$ for the attributes $a_1,\dots,a_k$;
- $R(a'_j) = v_j^{\otimes}$ for the attributes $a'_1,\dots,a'_l$.

Adding output nodes. Let the query Q be translated by calling $\mathcal{G}[\![Q]\!]$ on the translation of the database schema. The result of $\mathcal{G}[\![Q]\!]$ is a PSDG G and a representation R of $\mathbf{attr}(Q)$ in G. We add the following nodes and arcs to G:

- For each $a_i \in \mathbf{attr}(Q)$, add nodes v_i and v_i^O. For both of them, their dimension and input dimension are equal to $\mathsf{dim}(R)$. Node v_i is an internal node, while v_i^O is an output node. There is an arc from v_i to v_i^O; its $\overline{\delta}(\cdot)$-mapping is the identity mapping on $\mathsf{dim}(R)$. There are two arcs into v_i, first from $R(\exists)$ and second from $R(a_i)$. Their $\overline{\delta}(\cdot)$-mappings are also the identity mappings on $\mathsf{dim}(R)$. The operation of v_i is named "Output". The semantics of an "Output" operation is to return the second argument, if the first argument is true, and to return NULL otherwise.