The Vadalog Parallel System: Distributed Reasoning with Datalog+/-

Luigi Bellomarini
Banca d'Italia
luigi.bellomarini@bancaditalia.it

Davide Benedetto
Prometheux & Università Roma Tre
davben@prometheux.co.uk

Matteo Brandetti
TU Wien
matteo.brandetti@gmail.com

Emanuel Sallinger
TU Wien & University of Oxford
sallinger@dbai.tuwien.ac.at

Adriano Vlad
Prometheux, TU Wien &
University of Oxford
adriano@prometheux.co.uk

ABSTRACT

Over the past years, there has been a growing demand for ontological reasoning systems based on languages of the Datalog+/- family, such as Vadalog, for their ability to effectively model a wide range of real-world problems with powerful features such as existential quantification. As the scale and complexity of data analysis tasks continue to grow, the ability to distribute the computational workload across multiple non-communicating processors has become vital for these systems to achieve scalable performance.

The joint presence of existential quantification and recursion poses new challenges, currently unsolved by existing distributed systems, which only concentrate on Datalog and are therefore unsuitable for ontological reasoning. When working across multiple processors, generating all the facts to answer a specific reasoning query, avoiding duplication, and guaranteeing termination are non-trivial tasks as infinitely many new symbols and facts can be generated by existential quantification and recursion.

In this paper, we address such challenges and introduce the first distributed framework in the Datalog+/- space. We propose the condition of homomorphic decomposability, which identifies sets of Datalog+/- rules with good distribution properties. We put homomorphic decomposability into action with a distributed reasoning algorithm for Warded Datalog+/-, the core of Vadalog. We implement Vadalog Parallel, a distributed reasoner for Vadalog and provide experimental evaluation against state-of-the-art systems.

PVLDB Reference Format:
Luigi Bellomarini, Davide Benedetto, Matteo Brandetti, Emanuel Sallinger, and Adriano Vlad. The Vadalog Parallel System: Distributed Reasoning with Datalog+/-. PVLDB, 17(13): 4614 - 4626, 2024.
doi:10.14778/3704965.3704970

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/prometheuxresearch/VadalogParallel-Experiments.

Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704970

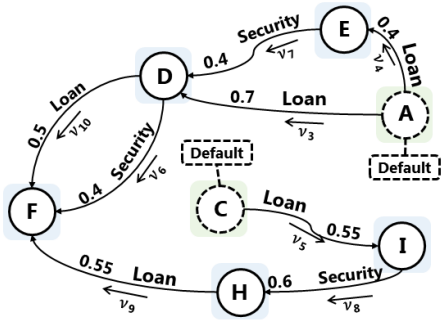

Figure 1: The exposure network of Example 1.1. The nodes are financial entities and the edges represent issued loans or security ownership relationships.

1 INTRODUCTION

Recent years have witnessed a rising interest in the adoption of reasoning frameworks based on languages of the *Datalog*$^{\pm}$ family [16], such as Vadalog [13], thanks to their *reasoning features*, such as existential quantification and recursion. These features allow them to support both triple-based models and multi-attributed graphs, to efficiently encode graph traversals, to capture SPARQL under the OWL 2 QL entailment regime for querying the semantic web [28].

We observe the use of Datalog$^{\pm}$ in finance (e.g., banking supervision, creditworthiness, anti-money laundering, fraud detection), medicine and biology (e.g., virology, patient pathways), enterprise resource planning (e.g., supply chain management), and more domains, as also confirmed by a flourishing of industrial applications [3, 5, 10, 12, 15–18, 23, 33, 44, 56] and dedicated venues [4].

The Need for Distribution. With the undebatable data growth trend and the complexity of analytical tasks, the ability to scale out and distribute the reasoning workload across multiple workers is a vital feature of modern reasoning engines. While state-of-the-art studies show the effectiveness of parallel evaluation techniques primarily based on the shared memory and the message passing paradigms with pure Datalog [20, 21, 37, 50, 57, 60, 61, 63], existential quantification and the need for ontological reasoning with Datalog$^{\pm}$ present complex challenges, unaddressed by current literature and existing techniques. This work deals with them.

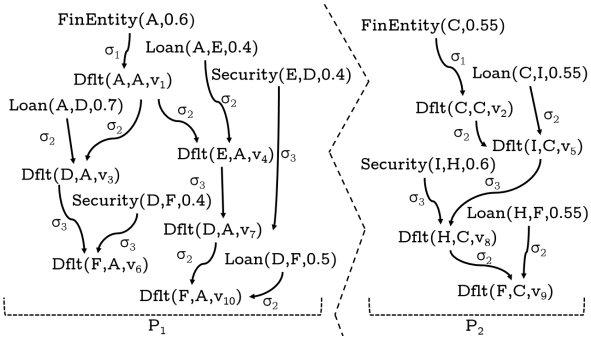

Figure 2: A portion of the chase for Example 1.1.

Motivating Example. To enable ontological reasoning, the *Datalog*$^{\pm}$ languages augment the expressivity of Datalog by adding advanced features, such as existential quantification, while introducing syntactic limitations to guarantee tractability [28]. Intuitively, ontological reasoning consists in answering a *conjunctive query* (CQ) Q over a database D, augmented with new tuples derived by the application of a set of Datalog$^{\pm}$ rules Σ. Consider an example:

Example 1.1. The database D models an "exposure network" (Figure 1), where financial entities x: (1) are represented as nodes and have a default probability p (FinEntity(x, p)); (2) receive loans from financial entities y with a credit risk lgd (Loan(x, y, lgd)); (3) receive securities of percentage w, issued by financial entities y (Security(x, y, w)).

D={$Loan(A, D, 0.7)$, $Loan(A, E, 0.4)$, $Loan(H, F, 0.55)$, $Loan(C, I, 0.55)$, $Loan(H, F, 0.55)$, $Loan(D, F, 0.5)$, $FinEntity(A, 0.6)$, $FinEntity(C, 0.55)$, $Security(E, D, 0.4)$, $Security(D, F, 0.4)$, $Security(I, H, 0.6)$}.

Entities are affected by default events d triggered by other entities z (Dflt(x, z, d)). D is augmented with a set Σ of rules, which encodes the conditions to propagate the default events throughout the network.

$$FinEntity(a, p), p > 0.5 \rightarrow \exists d\ Dflt(a, a, d) \quad (\sigma_1)$$
$$Dflt(b, a, d_1), Loan(b, c, lgd), lgd \geq 0.5 \rightarrow \exists d_2\ Dflt(c, a, d_2) \quad (\sigma_2)$$
$$Dflt(b, a, d_1), Security(b, c, s), s \geq 0.3 \rightarrow \exists d_2\ Dflt(c, a, d_2) \quad (\sigma_3)$$

We assume that an entity a with a default probability (p) greater than 50% initiates a default event d (σ_1). Similarly, if c granted a loan to a defaulting entity b with a high credit risk lgd, then it defaults on its debt (σ_2). Finally, c defaults on its debt, if it owns more than 30% of the securities issued by a defaulting entity b (σ_3).

Rules in Σ are function-free Horn clauses, potentially including existential quantification. They are known as *Tuple-Generating Dependencies* (TGDs) and are of the form $\forall \mathbf{x}\, \boldsymbol{\phi}(\mathbf{x}) \rightarrow \exists \mathbf{z}\, \boldsymbol{\psi}(\mathbf{y}, \mathbf{z})$, where $\boldsymbol{\phi}(\mathbf{x})$ (the *body*) and $\boldsymbol{\psi}(\mathbf{y}, \mathbf{z})$ (the *head*) are conjunctions of atoms over a relational schema $\mathbf{S}$. The semantics of TGDs is defined with an algorithmic tool known as the CHASE procedure [41]. Intuitively speaking, the chase expands D with new facts by applying the TGDs in Σ until a fixpoint is reached, introducing freshly generated *labelled nulls* that act as placeholders for existential quantification.

Let us now analyze our stress test via an ontological reasoning task. For example, we are interested in extracting the entities whose default has been initiated by the default of at least two other entities. In other terms, given D and Σ, we want to answer the CQ Q : $q(a) \leftarrow Dflt(a, b, d_1), Dflt(a, c, d_2), b \neq c$.

The chase of D under Σ is illustrated in Figure 2, where the entailed the facts $Dflt(F, A, v_6)$ and $Dflt(F, C, v_9)$ witness that $a = F$ is the desired defaulting entity, as an answer to Q.

As a base question, we wonder: can the computational workload to build the chase be split amongst two independent processors P_1 and P_2? If the relations *Loan* and *Security* are replicated in P_1 and P_2, then we could differentiate computation of facts for *Dflt* by filtering the applications of σ_1, for instance, as follows.

Example 1.2. The next distribution plan computes the chase for the involved financial entities: processor P_1 handles the larger entities (e.g., A in Figure 2) and P_2 the smaller ones (e.g., C).

$$FinEntity(a, p), Large(a), p > 0.5 \rightarrow \exists d\ Dflt(a, a, d) \quad (\rightarrow P_1)$$
$$FinEntity(a, p), Small(a), p > 0.5 \rightarrow \exists d\ Dflt(a, a, d) \quad (\rightarrow P_2)$$

It is immediate to observe that when the scale of our stress test grows, and so does the cardinality of *FinEntity*, such a parallel execution has an incremental performance gain over the serial execution. Moreover, adopting a finer-grained distribution would shorten execution times by achieving a uniform workload balance among workers, for instance using a hashing criterion based on the values of a to distribute facts for $FinEntity(a, p)$.

Distribution Properties. A distribution plan is hinged on *partitioning properties*, which play a crucial role in both the efficiency of the plan itself and the correctness of the computation.

First, we need a form of *non-redundancy in the computation*. For instance, the distribution plan in Example 1.2 shows that the two processors P_1 and P_2 produce distinct facts, i.e., non-redundant for answering Q. For the TGDs in Example 1.1, all facts involving a specific entity are always produced by the same processor. Such property holds for Σ independently of the underlying database instance. For instance, the large entity A occurs only in the facts generated by processor P_1. Instead, the small financial entity C occurs exclusively in the facts generated by processor P_2.

Second, we wish for a form of *completeness of the parallel computation*. For example, no TGD in Σ should be triggered by conjunctions of facts in different partitions. In our example, the facts can be derived only by joining *Dflt* with *Loan* or *Security*, which are replicated in all processors, or from *FinEntity* itself. This implies that our distribution plan produces two "independent" subsets of facts, in parallel. When merged, this set is equivalent to the set of facts produced by a serial execution, thus upholding correctness.

Conversely, an improper partitioning of the data would result in either an unbalanced workload or an unbalanced/incorrect result.

Challenges. Ensuring non-redundancy is challenging with existential quantification. Labelled nulls are unknown values generated at runtime, which makes it hard to define partitioning strategies based on constant values known beforehand. For instance, in Example 1.1, if we partition by the third term of *Dflt*, then $Dflt$(d,a,v_3) and $Dflt$(d,a,v_7) will be redundantly produced by P_1 and P_2. Ensuring termination is also challenging. In the presence of TGDs with recursion and existential quantification, the chase can produce infinitely many labelled nulls, such as in Example 1.1. Based on the specific Datalog$^{\pm}$ language (e.g., *Warded* [28], *Shy* [39], and *Guarded* [15]), several terminating variants of the chase have been proposed. They

feature specific *applicability conditions*, which control when a chase step, namely, the "firing" of a rule, can take place so as to avoid non-termination. The applicability condition checks whether a fact will be generated in the chase for example by preempting the creation of homomorphic facts—i.e., there exists a constant-preserving mapping between the terms of the two facts exists. This task is affected by distribution since homomorphic facts can be produced in different processors. For instance, in Example 1.1, the fact $Dflt(F, A, v_{10})$ is homomorphic to $Dflt(F, A, v_6)$.

Existing Parallel Paradigms for Datalog. Numerous studies delve into the parallel evaluation of simple Datalog rules [49, 58–60]. In this context, several parallel Datalog frameworks have emerged (e.g., decomposability, load sharing, etc.), to guarantee different forms of non-redundancy and completeness properties. However, they are ineffective in the context of Datalog$^{\pm}$ due to the presence of existential quantification in TGDs. In fact, these frameworks are based on the *Set Semantics*, namely, they consider two facts equivalent (and thus, one is superfluous/duplicated) if they refer to the same predicate and share the same constants in the same positions. As evident, the Set Semantics ignores the labelled nulls within the facts and thus, violates the TGD applicability conditions governing chase steps – a fact is generated even if it is homomorphic to another one in D – hampering the chase termination. This cannot work in general in the context of TGDs; the facts containing labelled nulls in the chase are never equivalent, yet they are redundant according to chase step applicability.

Homomorphically Decomposable TGDs. In this paper, we identify the new class of homomorphically decomposable TGDs. As a key feature, homomorphically decomposable TGDs are such that there always exists a partitioning criterion for D such that two processors never generate two homomorphic facts. This makes our class particularly suitable for distributed reasoning. This is the case of Example 1.2, where the applicable chase steps in each processor only depend on chase steps applied by the same processor. In other terms, all pairs of facts for *Dflt* generated in different processors are not homomorphic. This guarantees the generation of independent and non-overlapping chase instances while preserving the correctness of the serial chase execution. Each instance can be generated by a single processor from a subset of facts in D and the processors can perform local termination checks without any communications.

Building on this notion, the paper offers several **contributions**:

- The characterization of **homomorphically decomposable** TGDs.
- A **sufficient condition** for homomorphic decomposability as well as a **partitioning strategy** of the input database.
- A concrete **application** of our techniques to *Warded Datalog*$^{\pm}$ [28], an expressive and tractable language of the Datalog$^{\pm}$ family.
- The **implementation** of such notions in *Vadalog Parallel*, a new system for distributed ontological reasoning adopting the Vadalog language, an extension of Warded Datalog$^{\pm}$ with features of practical utility.
- A full-scale **experimental evaluation** of Vadalog Parallel in a variety of real-world and synthetic scenarios.

Overview. The remainder of this paper is organized as follows. In Section 2, we provide the background. In Section 3, we discuss the related work. In Section 4, we introduce homomorphic decomposability. In Section 5, we illustrate the architecture of Vadalog Parallel. Section 6 covers experimental evaluation. Section 7 concludes the paper. Further examples and proofs are in the Appendix [11].

2 BACKGROUND

Let us start by laying out the preliminary notions.

Relational Foundations and Homomorphisms. Let $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$ be disjoint countably infinite sets of *constants*, *(labelled) nulls* and *variables*, respectively. They are known also as *terms*. A *(relational) schema* $\mathbf{S}$ is a finite set of predicates with associated arities. An *atom* is an expression $R(\bar{v})$, where $R \in \mathbf{S}$ is of arity $n \geq 0$ and $\bar{v}$ is an n-tuple of terms. A *database (instance)* D over $\mathbf{S}$ associates to each relation symbol in $\mathbf{S}$ a relation of the respective arity over the domain of constants and nulls. We denote as $dom(D)$ the set of constants in D. Relation members are called *tuples* or *facts*.

Given two atoms a_1 and a_2, we define a *homomorphism* from a_1 to a_2, a constant preserving mapping h such that $h(a_1) = a_2$. If h is a bijection, a_1 and a_2 are isomorphic. This can be extended to sets (or conjunctions) of atoms and facts.

CQs and Dependencies. A *conjunctive query* (CQ) Q over a schema $\mathbf{S}$ is an implication $q(\mathbf{x}) \leftarrow \boldsymbol{\phi}(\mathbf{x}, \mathbf{y})$, where $\boldsymbol{\phi}(\mathbf{x}, \mathbf{y})$ is a conjunction of atoms over $\mathbf{S}$, $q(\mathbf{x})$ is an n-ary predicate that does not occur in $\mathbf{S}$, and $\mathbf{x}$ and $\mathbf{y}$ are vectors of terms. A *Boolean conjunctive query* (BCQ) is a CQ of arity zero. A CQ or a BCQ Q is satisfied in D if there exists a homomorphism h from the atoms in $\boldsymbol{\phi}(\mathbf{x}, \mathbf{y})$ to the facts in D, i.e., $h(\boldsymbol{\phi}(\mathbf{x}, \mathbf{y})) \subseteq D$. A set of Datalog$^{\pm}$ rules Σ is a set of tuple-generating dependencies (TGDs). A TGD is a first-order implication $\forall \mathbf{x}\ \boldsymbol{\phi}(\mathbf{x}) \rightarrow \exists \mathbf{z}\ \boldsymbol{\psi}(\mathbf{y}, \mathbf{z})$, where $\boldsymbol{\phi}(\mathbf{x})$ (the *body*) and $\boldsymbol{\psi}(\mathbf{y}, \mathbf{z})$ (the *head*) are conjunctions of atoms over $\mathbf{S}$ and boldface variables denote vectors of variables, with $\mathbf{y} \subseteq \mathbf{x}$. We write these existential rules as $\boldsymbol{\phi}(\mathbf{x}) \rightarrow \exists \mathbf{z}\ \boldsymbol{\psi}(\mathbf{y}, \mathbf{z})$, using commas to denote conjunction of atoms in $\boldsymbol{\phi}(\mathbf{x})$ and $\boldsymbol{\psi}(\mathbf{y}, \mathbf{z})$. We denote the set of body and head variables of σ as $body(\sigma)$ and $head(\sigma)$ and as $pred(\Sigma)$ the set of all predicates in Σ.

The Chase. The chase [41] expands D with facts inferred by applying a set of TGDs Σ to D into a database $chase(D, \Sigma)$, possibly containing labelled nulls. A chase execution builds the *universal model* for D and Σ, i.e., for every database B that is a model for D and Σ, there is a homomorphism mapping $chase(D, \Sigma)$ to B. Given a database D, a TGD $\sigma : \boldsymbol{\phi}(\mathbf{x}) \rightarrow \exists \mathbf{z}\ \boldsymbol{\psi}(\mathbf{y}, \mathbf{z})$ is applicable to D if there exists a homomorphism θ such that $\theta(\boldsymbol{\phi}(\mathbf{x})) \subseteq D$. Then, the TGD chase step adds the fact $\theta'(\boldsymbol{\psi}(\mathbf{y}, \mathbf{z}))$ to D, if not already in or already added to D, where $\theta' \supseteq \theta$ is a homomorphism that extends θ by mapping the variables of $\mathbf{z}$ (if non-empty) to newly created labelled nulls. The chase iteratively applies the TGDs chase steps until a fixpoint is reached, which may lead to an infinite sequence of chase step applications. The chase graph $\mathcal{G}(D, \Sigma)$ is a directed graph having nodes labelled after facts from $chase(D, \Sigma)$ and having an edge from a node $\mathbf{a}$ to $\mathbf{b}$ if $\mathbf{b}$ derives from $\mathbf{a}$ by the application of a chase step (e.g., Figure 2 for Example 1.1). The chase tree $\mathcal{T}(f, D, \Sigma)$ of a fact f is the subgraph of $\mathcal{G}$ containing all the nodes and edges from which f can be reached, excluding f itself and its incoming edges.

Warded Datalog$^{\pm}$ and Vadalog. We define as $p[i]$ be the term in the i-th position of a predicate p and refer to it as *position* and as $exist(\sigma)$ the set of existentially quantified variables of σ. A position

$p[i]$ is *affected* if: (i) the variable $v \in exist(\sigma)$ and v appears in position $p[i]$; (ii) the variable $v \in body(\sigma) \cap head(\sigma)$, v appears only in affected positions in $body(\sigma)$ (i.e., it is *harmful*) and in position $p[i]$ in $head(\sigma)$ (i.e., it is *dangerous*). A TGD $\sigma \in \Sigma$ is *warded* if all the dangerous variables $v \in body(\sigma)$ appear in a single body atom, the *ward*, which shares only non-harmful variables with other body atoms. A set of TGDs Σ is Warded if all the TGDs in Σ are warded. After normalization steps [9], query answering on D under Σ can be performed over a finite variant of the chase that produces the same facts as the infinite chase [13].

Vadalog extends Warded Datalog$^{\pm}$ with features of practical utility [13]. For the experiments dealt with in this paper, we are interested in *monotonic aggregations* [51]. Rules including aggregations have the following form (and for simplicity, we only consider a single aggregate, but the presence of several ones could be easily accommodated): $\boldsymbol{\phi}(\mathbf{x}), v = aggr(q) \rightarrow \exists \mathbf{z}\, \boldsymbol{\psi}(\mathbf{y}, \mathbf{z}, v)$. Intuitively, aggregations operate as stateful record-level operators that keep an updated version and return the current aggregate at each invocation. The value of the variable v is computed by incrementally aggregating the values of q over the distinct groups identified by the values of $\mathbf{x}$. Note that q is an algebraic expression with variables from $\mathbf{x}$ and constants as arguments.

3 RELATED WORK

To the best of our knowledge, our research is the first to investigate parallel algorithms for ontological reasoning with Datalog$^{\pm}$. The previous literature only focused on the parallel evaluation of pure Datalog [2, 26, 37, 58–60, 63], i.e., without existential quantification. Some initial work studied highly parallelizable Datalog fragments, NC in data-*complexity* [21, 36]. The notion of decomposability was initially introduced and defined for Datalog *sirups*, programs with a single intensional predicate [60]. A study showcased the undecidability of determining whether a Datalog program is decomposable [59] and proposed a general approach for parallel Datalog evaluation based on the idea of *data reduction*, which involves creating copies of rules to enable parallel evaluation by multiple processors. A related concept, the *load sharing scheme* [58], offers a relaxed version of decomposability and is applicable to a broader range of Datalog programs. Under this scheme, even non-decomposable Datalog programs can be evaluated in parallel without requiring inter-processor communication. Another study defined a sufficient syntactic condition to find decomposable Datalog programs based on the notion of *generalized pivoting* [49]. In addition to decomposability, in the literature, several parallel frameworks have been defined for Datalog, which allow for different forms of inter-processor communication, via either shared-nothing or shared-memory paradigms [26, 59]. A recent theoretical framework [37] for Datalog implicitly defines the evaluation strategy in terms of the policies adopted by the processors to consume facts or produce new ones. The related parallel Datalog systems are analyzed in detail in Section 6.

4 VADALOG PARALLEL

In this section, we provide the theoretical foundations and algorithms that underpin our approach to distributed reasoning.

Technique Overview. Given a set of TGDs Σ, a BCQ Q, and a database D, our approach operates as follows: initially, we extract k "disjoint" sets of TGDs $\Sigma_0, \cdots, \Sigma_{k-1}$ from Σ; namely, each TGD set contains the same number of rules and structure as Σ, and constructs the chase starting from a partition of facts in D. These sets exhibit the following favorable distribution properties.

(1) The value of k can be arbitrarily chosen independently of D.
(2) The chase constructed by each Σ_i from D only contains facts that cannot be derived by the chase of any other Σ_j, with $i \neq j$ (*non-redundancy*).
(3) No fact derivation is missing compared to the standard chase execution of Σ over D, i.e.., the union of all chase instances coincides with the standard chase (*completeness*).

We then construct the chase of each Σ_i in parallel from D and combine the results to answer Q. We demonstrate that this technique yields identical results to directly evaluating Q and Σ over D.

4.1 Restricted Sets of TGDs

We name as *evaluable atom* a comparison condition with standard built-in comparison operators (e.g., $=, \neq, <, >, \leq, \geq$), including usual algebraic expressions defined over the body variables of a TGD in the left- and right-hand side. Typically, a real number is used in the right-hand side. For instance, $p > 0.5$, $lgd \geq 0.5$ and $s \geq 0.3$ in Example 1.1 are evaluable atoms. We name *restricted TGD* a TGD that contains an evaluable atom in the body. Given a TGD σ, the TGD obtained by adding an evaluable atom to it is a *restricted copy* of σ. Thanks to evaluable atoms, we can limit the applicability of TGDs. Given a database D, a restricted TGD $\sigma : \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\xi}(\mathbf{w}) \rightarrow \exists \mathbf{z}\, \boldsymbol{\psi}(\mathbf{y}, \mathbf{z})$, where $\boldsymbol{\xi}(\mathbf{w})$ (with $\mathbf{w} \subseteq \mathbf{x}$) is the evaluable atom defined over a set of symbols $\mathbf{T}$ (with $\mathbf{S} \cap \mathbf{T} = \emptyset$), is applicable to D if there exists a homomorphism θ from $body(\sigma)$ to D such that $\theta(\boldsymbol{\phi}(\mathbf{x})) \subseteq D$ and $\theta(\boldsymbol{\xi}(\mathbf{w}))$ is satisfied. In practice, given k processors, to define evaluable atoms, we will simply use hash functions $\Pi(\mathbf{w}) = hash(\mathbf{w})\ mod\ k$. Then, for a TGD σ and a set of processors k, the restricted copies of σ can be defined straightforwardly as in the following example.

$$FinEntity(a, p), \Pi(a) = 0 \rightarrow \exists d\ Dflt(a, a, d) \qquad (\sigma_0)$$
$$\cdots$$
$$FinEntity(a, p), \Pi(a) = k - 1 \rightarrow \exists d\ Dflt(a, a, d) \qquad (\sigma_{k-1})$$

We can compactly represent a set of restricted TGDs whose evaluable atoms have the same left-hand side and real numbers corresponding to processors i in the right-hand side (with $0 \leq i < k$) as a single *parametric restricted TGD*, by adding a conjunct $Partition(\mathbf{w})$, defined as follows in a piecewise fashion, and depending on the executing processor i.

$$Partition(\mathbf{w}) = \begin{cases} \text{TRUE} & \text{if } \Pi(\mathbf{w}) = i, \text{ with } i < k. \\ \text{FALSE} & \text{otherwise.} \end{cases} \qquad (1)$$

In our example, we would have $FinEntity(a, p), Partition(a)$. The set Σ_i obtained by replacing in Σ all the restricted TGDs with their restricted copy for processor i is named *restricted set*.

4.2 Homomorphically Decomposable TGDs

In general, a restricted set such that the induced chase instances are non-redundant and complete does not always exist.

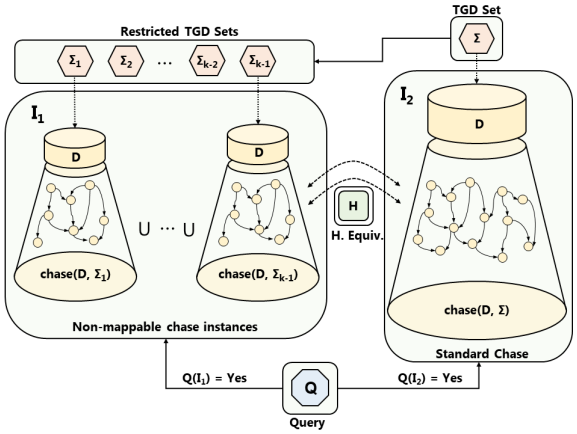

Figure 3: Homomorphically Decomposable TGDs.

Example 4.1. Two banks (Bank) in the same country (SameCountry) are supervised by the same National Central Bank (NCB).

D={$Bank(A)$, $Bank(B)$, $SameCountry(A, C)$, $SameCountry(B, C)$}

$$Bank(x), Partition(x) \rightarrow \exists cb\ NCB(cb, x) \qquad (\sigma_1)$$

$$NCB(cb, x), SameCountry(x, y) \rightarrow NCB(cb, y) \qquad (\sigma_2)$$

By letting $k = 2$, we have that processor $i = 0$ generates the first chase instance with the facts $NCB(A, v_1)$ by applying σ_1^0 and $NCB(C, v_2)$ by σ_2; and processor $i = 1$ creates the second chase instance containing the facts $NCB(B, v_3)$ by applying σ_1^1 and $NCB(C, v_4)$ by σ_2.

In this case, the distribution induced by $Partition(x)$ is redundant, in fact, it is sufficient that two different banks ($Bank(A)$ and $Bank(B)$) share the country with the same bank ($Bank(C)$) to generate homomorphically equivalent facts in different processors (e.g., $NCB(C, v_2)$ and $NCB(C, v_4)$ with $v_3 \rightarrow v_4$ and vice-versa).

Intuitively, we say that a set of TGDs Σ is *homomorphically decomposable* if there exist $k > 1$ restricted sets of Σ that give rise to n chase instances that are non-overlapping and complete.

Definition 4.2 (Homomorphically Decomposable TGDs). A set of TGDs Σ is homomorphically decomposable if there exist $k > 1$ restricted sets of TGDs $\Sigma_0, \ldots, \Sigma_{k-1}$ of Σ such that all the following conditions are satisfied.

- *Non-triviality. There exists at least a database D such that for each $i \in \{0, \ldots, k-1\}$, we have $chase(D, \Sigma_i) \neq \emptyset$.*
- *Non-mappability (i.e., non-redundancy). For every database D and for every pair $i, j \in \{0, \ldots, k-1\}$ with $i \neq j$, there does not exist a homomorphism θ mapping a fact $f_1 \in chase(D, \Sigma_i)$ to $f_2 \in chase(D, \Sigma_j)$, i.e., $\theta(f_1) = f_2$ with $f_1 \notin D$ and $f_2 \notin D$.*
- *Homomorphic Equivalence (i.e., completeness). For every database D and for all $i \in \{0, \ldots, k-1\}$, $chase(D, \Sigma)$ and $\bigcup_{i=1}^{n} chase(D, \Sigma_i)$ are homomorphically equivalent.*

The TGDs of Example 4.1 are not homomorphically decomposable: the non-mappability condition does not hold as $NCB(C, v_3)$ maps to $NCB(C, v_4)$. The TGDs of Example 1.1 are homomorphically decomposable, shown by the independent chase instances in Figure 2.

Partitioning the Chase. We now study important chase partitioning properties that directly descend from the notion of homomorphic decomposability. Given D, Σ and a finite set of k processors P, a *chase partitioning* is a total surjective function $\Omega : chase(D, \Sigma) \rightarrow P$ that maps each $f \in chase(D, \Sigma)$ to a processor of P.

We argue that for a set of homomorphically decomposable TGDs Σ, it is always possible to define an *eligible chase partitioning*, that is, a partitioning such that each fact f and all its chase predecessors, except for those in D, are assigned to the same processor. It is the case of Figure 2, where the chase tree of $Dflt(F, C, v_9)$ contains nodes for the facts $Dflt(H, C, v_8)$, $Dflt(I, C, v_5)$, and $Dflt(C, C, v_2)$.

Definition 4.3 (Eligible Chase Partitioning). Consider a database D, a set of TGDs Σ and a set of processors $P = \{0, \ldots, k-1\}$ with $k \geq 2$. An eligible partitioning is a chase partitioning Ω such that if $\Omega(f) = i$ with $f \in chase(D, \Sigma)$ and $i \in P$, then, for each $f_i \in \mathcal{T}(f, D, \Sigma)/D$ we have that $\Omega(f_i) = i$.

We capture this property in the following theorem that descends from the definition of homomorphically decomposable TGDs.

Theorem 4.4. *Consider a set of TGDs Σ, a non-empty D and set of processors $P = \{0, \ldots, k-1\}$ with $k \geq 2$, if Σ is homomorphically decomposable then there exists at least an eligible partitioning Ω of $chase(D, \Sigma)$ into P.*

Note that the existence of an eligible partitioning is a necessary but not sufficient condition for homomorphic decomposability. For instance, in Example 4.1, although there is an eligible chase partitioning for Σ, facts belonging to different chase instances are homomorphically mappable, thus not satisfying Definition 4.2.

4.3 Characterization of Homomorphically Decomposable TGDs

While the problem of determining if a set of TGDs Σ is homomorphically decomposable turns out to be undecidable, we introduce a sufficient syntactic condition, which moves from the observation that not all the sets of homomorphically decomposable TGDs are suitable for a distributed evaluation. In fact, in some cases, the partitions induced by the restricted sets Σ_i are unbalanced and some processors take most of the workload. In other cases, the partitions are fixed and processors cannot be scaled up.

Example 4.5. Consider the following set Σ of homomorphically decomposable TGDs, modeling the interactions between methods of different classes (Calls) in a Java project.

$$Call(x, y), Call(y, x) \rightarrow \exists d\ Dependency(d, x, y) \qquad (\sigma_1)$$

$$Dependency(d, x, x) \rightarrow \exists r\ SimpleRecursion(x, d, r) \qquad (\sigma_2)$$

Two methods of different classes x and y have a cyclic dependency (Dependency) if they invoke (Call) each other (σ_1). We detect simple recursions (r), i.e., a method invokes itself via self-dependencies (σ_2).

We can construct two restricted sets by adding the evaluable atoms $x = y$ and $x \neq y$ to the body of σ_1. In this case, the number of partitions is fixed and cannot be scaled up when the volume of D grows. The facts derived by σ_2 are generated only by the restricted set containing $x = y$ in the body of σ_1, while the remaining facts D, having $x \neq y$ will never trigger σ_2.

Balancing the Workload. We can define a subset of homomorphically decomposable TGDs whose evaluation can be always scaled up with an increasing size of D (and $dom(D)$). For such TGD sets, we can identify an arbitrary number of restricted sets—depending on the number of processors available—that induce an eligible partitioning of the chase with k, evenly distributed, chase instances.

Example 4.6. Consider a network modeling academic collaboration among researchers (Res), connected by collaborative relationships (CR). We aim to identify influential researchers who exert influence over collaborations between institutions.

$$Res(x), CR(x, y), Partition(x, y) \rightarrow \exists kr\ Influence(kr, x, x, y) \quad (\sigma_1)$$
$$Influence(kr, x, y, z), CR(z, y) \rightarrow Influence(kr, x, z, y) \quad (\sigma_2)$$

If researcher x (Res) has a collaborative relationship with researcher y, then y is influenced (Influence) by a key researcher kr via x (σ_1). If researcher z is influenced by the key researcher kr of x via a collaborative relationship with y, and z has a collaborative relationship with researcher y, then also y is influenced by kr via z (σ_2).

From the partitioning in Example 4.6, we observe that the k restricted sets of Σ defined by *Partition* give rise to k chase instances, each handling exactly a fixed set of constants determined by the initial partitioning of $CR(x, y)$ in D. Actually, constants never propagate across instances. The number of k processors can be scaled up with the growing size of $dom(D)$, balancing the workload.

Eligible Propagation: a Sufficient Condition. We observe that the favourable partitioning conditions of Example 4.6, are always satisfied whenever there is a set of variables that never bind to labelled nulls, namely, are *harmless* and propagate from the body to the head of all TGDs of Σ via the same set of non-affected positions. This condition implies that all the facts generated by the chase sequences induced by the k restricted TGDs copies differ for at least a constant, and no homomorphism can be found between the facts generated by different sequences. The harmlessness property of the propagated variable is crucial as the presence of a labelled null bound to the propagated variable may invalidate non-mappability.

In Example 4.6, in the TGD σ_1, the variable x is propagated from *Researcher*[0] and *CR*[0] to *Influence*[1] and *Influence*[2], while y is propagated from *CR*[1] to *Influence*[3]. On the contrary, in σ_2 the variable x is propagated from *Influence*[1] to *Influence*[1], y from *Influence*[2] to *Influence*[3] and z vice-versa.

To formalize our sufficient condition, we define an *eligible propagation position* for a harmless variable $v \in body(\sigma) \cap head(\sigma)$, as a non-affected position where v appears in a predicate of σ.

Theorem 4.7 (Eligible Propagation). *Given a set of TGDs Σ over a schema $\mathbf{S}$, we have that Σ is homomorphically decomposable if for each predicate ρ of Σ, there exists at least one eligible propagation position $\rho[i]$ shared by all the occurrences of ρ-atoms in every $\sigma \in \Sigma$.*

The homomorphically decomposable TGDs in Example 1.1 satisfy Theorem 4.7 with the shared eligible propagation position *Dflt*[0] as well as the TGDs in Example 4.6, with positions *Influence*[1], *Influence*[2], and *Influence*[3]. On the contrary, there is no shared eligible propagation position for the TGDs in Example 4.5.

Algorithm 1 summarizes a technique that uses Theorem 4.7 to decide whether a set of TGDs Σ is homomorphically decomposable. We use two different dictionary structures: (i) *Pos* maps a TGD $\sigma \in \Sigma$ and an atom α in σ to a set of positions (line 2); (ii) Θ maps the predicates in $pred(\Sigma)/\mathbf{S}$ to the set of eligible propagation positions in Σ (line 10). The algorithm exploits also three predefined functions: (i) HARMFUL(σ, Σ) to derive the harmful variables in σ (line 7); (ii) POSITION(x, α) and (iii) VARS(x) to extract the positions of a variable x in an atom α and the set of variables appearing in α, respectively (line 8,9 and 7).

Algorithm 1 Algorithm for Homomorphic Decomposability.

```
1:  function EligiblePropagation(Σ)
2:      Pos = ∅                                          ▷ dictionary of TGDs and atoms
3:      for all σ ∈ Σ do
4:          φ is the head atom of σ
5:          for all atoms α in the body of σ do
6:              for all x ∈ vars(α) do                   ▷ x is harmless and propagated
7:                  if x ∉ harmful(σ, Σ) ∧ x ∈ vars(φ) then
8:                      Pos[(σ, α)] = positions(x, α)    ▷ x positions in α
9:                      Pos[(σ, φ)] = positions(x, φ)    ▷ x positions in φ
10:     Θ = ∅                                            ▷ dictionary of predicates and positions
11:     for all σ ∈ Σ do
12:         for all ρ ∈ pred(Σ) in σ do
13:             for all atoms α in σ s.t. the predicate of α is ρ do
14:                 if ρ ∉ Θ.keys then                   ▷ if predicate ρ not yet found
15:                     Θ[ρ] = Pos[(σ, α)]
16:                 else
17:                     Θ[ρ] = Θ[ρ] ∩ Pos[(σ, α)]        ▷ shared positions
18:     if ∃ρ s.t. ρ ∈ pred(Σ)/S and Θ[ρ] is empty then
19:         return false
20:     return true
```

After iterating over the TGDs Σ to obtain all the eligible propagation positions of each atom in the body or in the head (line 3-10), the algorithm checks if, for each predicate $\rho \in pred(\Sigma)/S$ in σ (i.e., intensional predicate), there exists at least one shared position between all the atoms referring to ρ (line 10-20). If there exists an atom α in the body of σ that does not propagate any harmless variable, $Pos[(\sigma, \alpha)]$ is empty. The algorithm complexity is PTIME, by varying $|\Sigma|$ and the arities of the predicates in $pred(\Sigma)$.

Partitioning the Database. Our sufficient condition suggests a partitioning strategy for D. Given a set of TGDs Σ over a schema $\mathbf{S}$, we define as *body-ground TGDs* the TGDs in $\Sigma' \subseteq \Sigma$ whose body is composed only of atoms referring to predicates in S (i.e., *extensional* atoms). Note that (i) for each set of TGDs Σ, there exists at least one body-ground TGD, and (ii) not having dependencies on other rules, body-ground TGDs are evaluated first in the chase.

From the syntactical structure of Σ' we can define the partitioning of D across the processors as follows: we consider the extensional atoms in each $\sigma \in \Sigma'$; we construct an evaluable atom *Partition*$(\mathbf{x})$ on the set of variables $\mathbf{x}$ appearing in shared eligible propagation positions in $head(\sigma')$, i.e., the positions satisfying Theorem 4.7 for Σ. This implies that the positions $p[i]$ of every extensional predicate ρ in the body of σ' where the variables in $\mathbf{x}$ appear can be used as a partitioning criterion to initially distribute the facts of D across all the processors. For instance, in the body-ground TGD σ_1 of Example 4.6, variables x and y in the eligible propagation positions *Influence*[1], *Influence*[2] and *Influence*[3] appear also in *Researcher*[0], *CR*[0] and *CR*[1]. The vector of constants in these positions (i.e., *partitioning positions*) can be used to define a distribution key of the facts in D.

4.4 Homomorphic Decomposability and Warded TGDs

Homomorphic decomposability sustains efficient distribution techniques and therefore scalability. Nevertheless, for an arbitrary set of TGDs, homomorphic decomposability does not imply decidability or tractability of the query answering task. We adopt a practical approach and concentrate on a specific TGD fragment, namely Warded Datalog$^{\pm}$, a language exhibiting tractable query answering and very high expressive power, being then suitable for a variety of applications. In this section, we explore the case of Warded TGDs that are also homomorphically decomposable, which, in our experience, covers many practical scenarios. However, in Section 4.5, we discuss a fallback technique we apply for a distributed evaluation of Warded TGDs that are not homomorphically decomposable.

We have seen in Section 2 that with Warded TGDs, query answering on D under Σ can be equivalently performed over a finite chase. We name such variant $chase^{w}$ (Algorithm 2). Operationally, an instance of $chase^{w}$ is initialized with D and augmented with new facts obtained by activating TGDs of Σ only if such facts are not isomorphic to others already in $chase^{w}$.

Algorithm 2 Isomorphic Chase for Warded TGDs.

```
1: function CHASE^W(D, Σ)
2:     I = D
3:     for all σ ∈ Σ and x ∈ I to which σ applies do        ▷ for all TGDs
4:         if CHECK_ISOMORPHISM(I, σ(x)) then
5:             I = I ∪ {σ(x)}                              ▷ new fact found
6:     return I
```

In our distributed framework we consider sets of Warded TGDs for which the homomorphic decomposability property holds. Our results directly apply: within each processor, we execute $chase^{w}$ and thus do not generate isomorphic facts, ensuring a form of local termination. Thanks to homomorphic decomposability, no pairs of facts in different partitions are homomorphic and thus they are not isomorphic, guaranteeing the correctness of query answering thanks to the properties of Warded TGDs.

Algorithm 3 provides the full procedure (Parallel-Evaluate) to perform ontological reasoning with a set of Warded TGDs Σ enjoying homomorphic decomposability. It adopts the dictionary structure computed in Algorithm 1, mapping predicates to their corresponding partitioning positions. An empty database D_i is initialized in each processor (line 5). The facts referring to extensional predicates $\rho(D)$ appearing in non-body ground TGDs are replicated for every D_i in P with the function Replicate (line 6-7). The function DistributedByKey assigns the facts referring to extensional predicates in body-ground TGDs based on the values in the positions in $\Theta(\rho)$ (line 8-9). Each processor executes Algorithm 2 for Σ (i.e., $chase^{w}$) starting from its assigned D_i and performing local isomorphic checks. The instances I_i are then merged into a single one to answer Q (lines 10-11).

Theorem 4.8 (Correctness). *Consider a set of Warded TGDs Σ, a database D and set of processors $P = \{0, \ldots, k-1\}$, if Σ is homomorphically decomposable, then, for every BCQ Q, it holds that* Parallel-Evaluate(D, Σ, Q)=**true** *iff* $chase(D, \Sigma) \models Q$

Algorithm 3 Distributed evaluation of H.D. Warded TGDs.

```
1: function PARALLEL-EVALUATE(D, Σ, Q)
2:     Let Σ' be the body-ground TGDs of Σ
3:     Let P = {0, ..., k − 1} be the set of processors
4:     for all i ∈ P run in parallel do
5:         D_i = ∅
6:         for all ρ ∈ S appearing in Σ/Σ' bodies do
7:             D_i = D_i ∪ REPLICATE(ρ(D), i)
8:         for all ρ ∈ S appearing in Σ' bodies do
9:             D_i = D_i ∪ DISTRIBUTEBYKEY(ρ(D), Θ(ρ), i)
10:        I = I ∪ CHASE^W(D_i, Σ)
11:    return Q(I)                                          ▷ If Q is satisfied in I
```

4.5 Non-homomorphically Decomposable Warded TGDs

While homomorphic decomposability offers an intuitive and efficient partitioning strategy, as we shall see also in experimental settings, to guarantee the broadest applicability, we complement our approach with a technique to support the distributed evaluation of any Warded set of TGDs, that is, also when homomorphic decomposability does not apply. To this end, we introduce *Distributed Warded Seminaive Evaluation (DW-SNE)*, a Map-Reduce evaluation strategy conceived as a variant of *seminaive evaluation* (SNE) [1], that supports $chase^{w}$ and is tailored for distributed settings.

Our distributed variant prevents the generation of isomorphic facts by considering, at each iteration, only the set of facts that are not isomorphic to facts already produced by any processor in previous iterations. When no processor produces new facts, the algorithm terminates by producing $chase^{w}(D, \Sigma)$ as the union of the facts generated by each processor.

The DW-SNE splits the construction of $chase^{w}(D, \Sigma)$ into a pipeline of Map-Reduce steps which takes D in input. At the beginning of every step, facts are moved to the same processor according to the partitioning criteria foreseen by the mentioned operation. When Σ is recursive, a subset of the steps of the pipeline is repeated considering the delta facts until no new isomorphic facts are generated. To perform the labelled null creation, deduplication and difference in a parallel fashion, each fact stored in a distributed dataset, contains two additional meta-fields: (1) A *reasoning key* representing the fact-identity. Isomorphic facts have equivalent keys to perform deduplication and different operations in the DW-SNE. The keys are used to move facts into the same processor and can be created on the fly by renaming the labelled nulls contained in the fact. (2) The *provenance* encoding a fact rooted chase tree. The provenance value is unique in the chase and can be used to efficiently generate fresh symbols on the fly to support existential quantification without coordinating with other processors.

5 SYSTEM ARCHITECTURE

We introduce the novel system Vadalog Parallel, implementing homomorphic decomposability and DW-SNE. For the specification of TGDs, the system adopts the Vadalog language [13].

Architecture Description. Given a CQ $Q : q(\mathbf{x}) \leftarrow \boldsymbol{\phi}(\mathbf{x}, \mathbf{y})$, a set of TGDs Σ and a database D, the process of evaluating Q over D under Σ is composed of 4 phases, each managed by a dedicated functional module (represented in Figure 4). In terms of distribution, phases

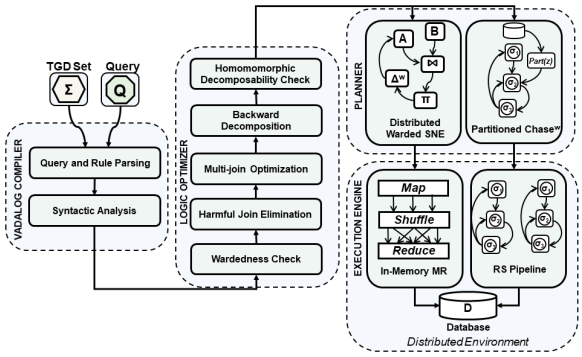

Figure 4: Architecture of the Vadalog Parallel System.

(1)-(3) are database-independent and executed by the cluster master node. The execution phase (4) is performed by all the processors.

(1) The *compiler* performs the syntactic checks on Q and Σ to verify if they are compliant with the Vadalog grammar.
(2) The *logic optimizer* performs normalization steps (e.g., splitting n-ary joins into sequences of binary joins) and checks that Σ is Warded and homomorphically decomposable, by Algorithm 1.
(3) The *planner* creates the execution plan for Σ and Q: *Replicated Streaming Pipeline*, if Σ is homomorphically decomposable, DW-SNE otherwise. Both are explained next in this section.
(4) The *execution engine* evaluates Q based on the execution plan built by the previous module and writes the output facts satisfying Q into external sources (e.g., HDFS).

Replicated Streaming Pipeline (RSP). This distributed evaluation model applies when Σ is homomorphically decomposable and is based on Algorithm 3. To encode its chase instance, each processor builds and executes an independent execution *streaming pipeline* from the dependency graph of the predicates of Σ. In each pipeline, the atoms correspond to filters, connected by pipes that denote the input-output transformations applied by the TGDs. Data flow from source filters, the extensional atoms, to the target, the atoms in the query body $\boldsymbol{\phi}$, while undergoing the transformations performed by the TGDs (e.g., selections, projections, joins, value inventions). The termination checks are performed via local hash tables implementing a form of *isomorphism check* for the generated facts. This strategy allows for activating only the chase steps required to answer Q from a single restricted copy of Σ. The labelled null generation process is performed by adopting different fresh symbols in each processor exploiting the constants occupying the partitioning positions concatenated with a processor-local incremental index (Section 4.4). For the extensional predicates in the body of non-exit (join) TGDs, replicated in each processor, we build a hash index on the corresponding join keys. When all the processors reach a fixpoint, the output facts are collected by the master processor to compute the final answer to Q.

DW-SNE. This model applies when Σ is not homomorphically decomposable. Our DW-SNE implementation follows the *template method* pattern [38]: we create abstract interfaces offering SQL-like operations to manipulate distributed data structures (i.e., Spark `Dataset`) [47], where each record is associated with a fact plus its metadata (e.g., reasoning key and provenance). The main algorithm consists of a possibly iterative sequence of such operations composed via interface procedures. Under the hood, the operations offered by the interfaces exploit Spark `Dataset`, i.e., a shared-memory abstraction implementing in-memory Map-Reduce data transformations. Several *recursion-aware optimizations* are incorporated to limit redundant computations, minimize the memory footprint, and reduce the exchange of data across processors. The execution exploits both the logical and physical optimization performed by Catalyst, a state-of-art optimizer embedded in Spark. The memory footprint benefits from the serialization and encoding mechanism of Spark's data structures. We enabled an efficient auto-clearing caching mechanism to break and store intermediate results.

6 EXPERIMENTS

We validate homomorphic decomposability and the architecture of our system by showing that it exhibits high scalability and outperforms the existing parallel Datalog-based systems in both synthetic and real-world scenarios. Our experiments leverage a variety of ontologies and datasets, both synthetic and real-world, to comprehensively evaluate various dimensions of the system performance.

Benchmarks and Hardware Setup. In Section 6.1, we evaluate the impact of specific properties of wardedness on the performance of our execution models RSP and DW-SNE (Section 5). In Section 6.2, we compare our system with other distributed or parallel Datalog-based systems on various graph traversal problems. In Section 6.3, we compare with other TGD-based systems. In Section 6.4, we stress the scalability of our system and show that by resource scale-up, it outperforms ad-hoc implementations in data-intensive problems.

For *shared-memory* systems and implementations, we used a c5d.metal AWS instance with Ubuntu v20, having 96 cores, 192GB RAM, and 900GB SSD NVMe. For *shared-nothing* systems and implementations, we used an AWS cluster with six nodes. Each node is a c5d.4xlarge instance with Ubuntu v20 with 16 cores, 32GB RAM, and 400GB SSD NVMe. For comparative experiments, we ensured equivalence in resources of the two configurations.

6.1 iWarded: Synthetic Scenarios

We examine the performance impact of wardedness and the input database D with homomorphic decomposability. Our experiments show that, in the average case when the constants in $dom(D)$ are evenly distributed among the facts in D, the speed-up of an evaluation model based on non-communicating processors (i.e., RSP) is considerable. Instead, when a set of constants is highly concentrated in the partitioning positions of body atoms in body-ground TGDs, an evaluation strategy based on inter-processors communication (i.e., DW-SNE) is preferable to mitigate computational bottlenecks. To generate the testing scenarios, we used iWarded [7], a generator of Warded benchmarks.

Scenarios and Datasets. We created 8 scenarios, each composed of a database and 20 TGDs, with different parameters for linear/join TGDs, recursive TGDs, existentials, and eligible propagation positions (parameters in Figure 5). The input databases contain 100k facts each, constructed with different selectivity values (**Sel**) for the partitioning positions of the body atoms of body-ground TGDs

(**Pos**). This measure highly affects the load balance factor of the partitioning and greater selectivity implies more balanced partitions. We also considered 10 CQs generated with iWarded (Figure 5).

Scen.	L/⋈ TGDs	L/⋈ rec	rec len	∃ TGDs	Pos	Sel
SynthA	10/10	3/3	3	10	1	0.90
SynthB	10/10	3/3	3	10	1	0.01
SynthC	10/10	3/3	3	10	1	0.3
SynthD	0/20	0/10	2	0	5	0.95
SynthE	0/20	0/10	2	10	5	0.95
SynthF	20/0	10/0	5	10	3	0.85
SynthG	10/10	5/5	2	10	10	0.99
SynthH	10/10	5/5	10	10	10	0.99

Figure 5: iWarded parameters of the synthetic scenarios.

Results. In Figure 6, we report the execution times for each scenario, considering RSP and DW-SNE. For *SynthA*, *SynthB*, and *SynthC*, the impact of selectivity on RSP performance becomes evident: RSP exhibits significant slowdowns with low selectivity values, where the facts repeatedly have the same constant, which unbalances the workload. In contrast, the uniform execution times for DW-SNE in *SynthA*, *SynthB*, and *SynthC* show that selectivity does not influence its performance. This depends on the absence of pre-defined partitioning in the input instance. For the remaining scenarios, we observe a consistent trend for both RSP and DW-SNE. In particular, RSP consistently outperforms DW-SNE, being from 2x to 10x faster. We observe the shortest times with both RSP and DW-SNE in *SynthF*, which has linear TGDs, hence highlighting the impact of join operations. Remarkably, despite having a selectivity of 0.85, RSP surpasses the speed of scenarios with higher selectivity, primarily due to the absence of join TGDs. This trend is further supported by the performance of *SynthD* and *SynthE*, which feature join TGDs and no linear rules and are the slowest scenarios after *SynthF*. The suboptimal performance of both execution models on *SynthF* can be attributed to the recursive length value, where all the recursive TGDs are entwined in the same dependency cycle. *SynthG* has the same parameters as *SynthF*, except for a recursive length of 2, which has a notable impact on the execution dynamics and makes Vadalog Parallel on *SynthG* much faster.

6.2 Comparison with Parallel Datalog Systems

The majority of the existing parallel Datalog-based systems in the literature support pure Datalog enriched with standard and monotonic aggregations, but none of them supports ontological query answering tasks under Datalog$^{\pm}$. Thus, to compare our system with other existing parallel tools, we only focus on pure Datalog and monotonic aggregation scenarios.

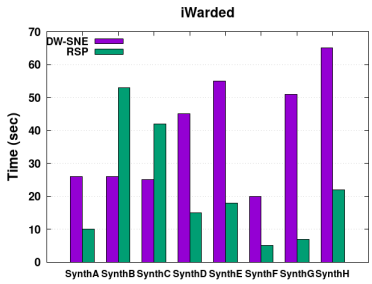

Figure 6: Experiment results for iWarded Scenarios.

Systems Tested. We selected both *shared-nothing* systems such as BigDatalog [50] and Myria [32], and *shared-memory* ones, such as Soufflè [48] and RecStep [25]. We excluded from the comparison Cog [34], DCDatalog [61], DeALS-MC [62], and LogicBlox [29] due to unavailability for direct benchmarking.

BigDatalog is a distributed system based on the parallel SNE. It uses ad-hoc implementations of Spark distributed datasets, and new operators designed and optimized for recursive Datalog queries (also known as `SetRDD`). Myria is another shared-nothing distributed system supporting iterative queries with aggregates; it features incremental computations with both synchronous and asynchronous query evaluation. Soufflè is a high-performance Datalog engine for large-scale program analysis that employs several optimization techniques such as efficient program synthesis, specialized parallel data structures for indexing and compression, and automatic index selection. RecStep is a general-purpose Datalog engine built on top of *QuickStep* [46], an efficient in-memory parallel RDBMS. It supports Datalog with both stratified negation and aggregation, namely, a language fragment that can express a wide variety of data processing tasks.

Scenarios and Datasets. We selected five relevant ontologies modeling graph-related problems described in Figure 7-(1-5).

(1) *Same Generation (SG)* computes all the node pairs (x, y) in a directed graph, such that there exist a source node p and two distinct paths connecting p to x and p to y, respectively.
(2) *Transitive Closure (TC)* computes all the node pairs (x, y) in a graph such that y is reachable from x via at least one path.
(3) *Triangle Counting (Tri-C)* computes the number of distinct "triangle" components in an undirected graph, i.e., given three distinct nodes x, y and z, there exists a path of length 3 from x to itself, traversing the nodes y and z (and vice-versa).
(4) *All Shortest Paths (ASP)* computes all the shortest paths from a node x to a node y in a directed weighted graph. We consider a random weight of the edges ranging from 1 to 8.
(5) *Non-2-Colorability (N2C)* is a mutual recursive set of TGDs that computes all the node pairs (x, y) that are connected via odd (*Odd*) and even (*Even*) length paths in a graph. By the BCQ $Q \leftarrow Odd(x, x)$ we check whether the graph is not 2-colorable.

We evaluate such ontologies on synthetic and real-world graphs (Figure 8). The synthetic scenarios consist of many graph topologies, generated with *networkX* [31]: *Tree17* is a tree of height 17, where the degree of each non-leaf node is randomly between 2 and 4; *Grid150* and *Grid250* are grid graphs of 150×150 and 250×250 size, respectively. The *Gn-p* graphs are n-vertex graphs generated by randomly connecting pairs of nodes with a 0.002 probability. As real-world datasets, we selected four graphs of scientific collaboration networks [52] and *LiveJournal* [53]

Although the input graphs are not considerable, the size of the chase can be quadratic in the number of nodes.

Results. The outcomes are in Figure 9. In homomorphically decomposable scenarios, such as TC, ASP, and N2C, Vadalog Parallel consistently outperforms its counterparts, exhibiting a performance gain of up to 10000x. The highest speedup is observed when

Ontology Name	TGD Sets	Query	H. Decom.
(1) Same Generation (SG)	$Arc(p,x), Arc(p,y), x \neq y \rightarrow SG(x,y)$ (σ_1) $Arc(a,x), SG(a,b), Arc(b,y) \rightarrow SG(x,y)$ (σ_2)	$Q(x,y) \leftarrow SG(x,y)$	N/A
(2) Transitive Closure (TC)	$Arc(x,y) \rightarrow TC(x,y)$ (σ_1) $TC(x,y), Arc(y,z) \rightarrow TC(x,z)$ (σ_2)	$Q(x,y) \leftarrow TC(x,y)$	$Arc[0]$
(3) Triangle Counting (Tri-C)	$Arc(x,y), Arc(y,z), Arc(z,x), x < y, y < z \rightarrow T(x,y,z)$ (σ_1) $T(x,y,z), c = mcount(1) \rightarrow CountT(c)$ (σ_2)	$Q(x) \leftarrow CountT(x)$	N/A
(4) All Shortest Paths (ASP)	$Arc(x,y,d), dm = mmin(d) \rightarrow ASP(x,y,dm)$ (σ_1) $ASP(x,y,d1), Arc(y,z,d2), dm = mmin(d1+d2) \rightarrow ASP(x,z,dm)$ (σ_2)	$Q(x,z,w) \leftarrow ASP(x,z,w)$	$Arc[0]$
(5) Non-2-Colorability (N2C)	$Edge(x,y) \rightarrow Odd(x,y)$ (σ_1) $Odd(x,y), Edge(y,z) \rightarrow Even(x,z)$ (σ_2) $Even(x,y), Edge(y,z) \rightarrow Odd(x,z)$ (σ_3)	$Q \leftarrow SG(x,x)$	$Edge[0]$
(6) Close Links (CL)	$Own(x,y,w), tw = msum(w) \rightarrow MCL(x,y,tw)$ (σ_1) $MCL(x,y,w_1), Own(y,z,w_2), tw = msum(w_1 \cdot w_2) \rightarrow MCL(x,z,tw)$ (σ_2) $MCL(x,y,tw), tw \geq 0.2 \rightarrow CL(x,y)$ (σ_3)	$Q(x,y) \leftarrow CL(x,y)$	$Own[0]$
(7) Company Control (CCTR)	$Own(x,y,w), x \neq y \rightarrow ControlledShares(x,y,y,w)$ (σ_1) $Control(x,y), Own(y,z,w), x \neq z \rightarrow ControlledShares(x,z,y,w)$ (σ_2) $ControlledShares(x,z,y,w), tw = msum(w) \rightarrow TControlledShares(x,z,tw)$ (σ_3) $TControlledShares(x,z,tw), tw > 0.5 \rightarrow Control(x,z)$ (σ_4)	$Q(x,y) \leftarrow Control(x,y)$	$Own[0]$
(8) Person with Significant Control (PSC)	$KeyPerson(x,p), Person(p) \rightarrow PSC(x,x,p)$ (σ_1) $Company(x) \rightarrow \exists p\ PSC(x,x,p)$ (σ_2) $Control(y,x), PSC(y,z,p) \rightarrow PSC(x,z,p)$ (σ_3)	$Q(x,z,p) \leftarrow PSC(x,z,p)$	$KeyPerson[0]$ $Company[0]$
(9) Strong Links (SL)	$KeyPerson(x,p), Person(p) \rightarrow PSC(x,x,p)$ (σ_1) $Company(x) \rightarrow \exists p\ PSC(x,x,p)$ (σ_2) $Control(y,x), PSC(y,z,p) \rightarrow PSC(x,z,p)$ (σ_3) $PSC(x,z,p), PSC(y,z,p), x \neq y, w = mcount(1), w > 3 \rightarrow SL(x,y,w,z)$ (σ_4)	$Q(x,y,w,z) \leftarrow SL(x,y,w,z)$	$KeyPerson[0]$ $Company[0]$

Figure 7: Benchmark Ontologies for the Experimental Evaluation. The column H. Decom. shows the partitioning positions of body atoms in body-ground TGDs of Σ. If Σ is not homomorphically decomposable we write N/A.

Scenarios	Nodes	Edges	Type
Tree17	1,631,318	1,631,319	Synthetic
Grid150	22,500	44,700	Synthetic
Grid250	62,500	124,500	Synthetic
G5K	5000	24,978	Synthetic
G10K	10,000	50,057	Synthetic
G20K	20,000	200,229	Synthetic
G40K	40,000	798,979	Synthetic
Hep-Ph	12,008	237,010	Real-world
Hep-Th	9,877	51,971	Real-world
Cond-Mat	23,133	186,936	Real-world
Astro-Ph	18,772	396,160	Real-world
LiveJournal	4,847,572	68,993,773	Real-world

Figure 8: Synthetic and Real-world graph parameters.

comparing Vadalog Parallel with other shared-nothing systems, including BigDatalog and Myria. BigDatalog is optimized for Datalog decomposable programs, leveraging `SetRDD` in the SNE for set-difference and deduplication operations through *zipPartitions* functions. This operation involves merging the facts of corresponding partitions on the same processor. However, the performance drawback of this mechanism lies in the necessity for processors to await the completion of a single Spark transformation in each recursion round. On the other hand, Vadalog Parallel executes different streaming pipelines in parallel, where all the operations executed by distinct processors are completely independent. A pipeline execution is embedded into a single Spark *mapPartition* operation. The performance gap between Vadalog and shared-memory systems is reduced compared to shared-nothing systems, particularly with Soufflè, which exhibits slightly superior performance in non-homomorphically decomposable scenarios, such as SG and Tri-C. However, in scenarios like TC, ASP, and N2C, Vadalog Parallel is 2x to 100x faster than Soufflè. This enhanced performance wrt shared-nothing systems is due to the optimizations employed in systems like Soufflè, which uses a specialized data structure called *Brie* [35], known for its effective compression capabilities for high-density relations. Nevertheless, the cost of maintaining these indexes becomes relevant with higher chase cardinalities ($\approx$ 1000 mln), as observed in N2C on Grid150 and G40K, TC on Grid250, where Vadalog Parallel outperforms Soufflè and the other systems.

6.3 Related TGD-based Tools

This section delves into the benefits of parallel evaluation to perform ontological reasoning tasks with TGDs.

Reasoners Tested. We look at the scenarios and reasoners of ChaseBench [14], a comprehensive benchmark for Datalog$^{\pm}$ systems implementing the chase. We select the top-performing ones.

RDFox [43] is a high-performance RAM-based Datalog engine, implementing a parallel, non-distributed variant of the seminaive chase. It only supports existentials under w-acyclicity condition [24].

LLunatic [27] is a Datalog-based system that can handle data exchange tasks. It supports certain query answering under weak acyclic TGDs and runs on top of PostgreSQL.

DLV$^{\exists}$ [40] is a disjunctive RAM-based Datalog system supporting CQ-answering under Shy TGDs with the *parsimoniuos* chase. It also employs the SNE to materialize the chase and answer the CQ.

Scenarios and Datasets. We select scenarios from ChaseBench with many existential quantifications and join TGDs. The TGD sets in our scenarios are weakly acyclic, shy, and also warded. *STB-128* and *ONT-256* are data exchange scenarios generated with iBench [6], a popular tool for benchmarking TGDs. *STB-128* is a data mapping scenario composed of 167 TGDs with 150k source instances; *ONT-256* is composed of 529 TGDs with 1m source instances. We considered 20 CQs. *Doctors* and *DoctorsFD* is a non-recursive data integration task from the schema mapping literature. We used source instances of 10K, 100K, 500K, and 1M facts and ran 9 CQs. *LUBM* [30] is a widely adopted benchmark from the university domain. We used source instances of 90K, 1M, 12M, and 120M facts. We ran 14 queries and averaged the answering time. The execution times comprise input loading, chase, result export, and queries. We run Vadalog Parallel with a different number of

Ontology	Dataset	Chase Size	Vadalog Par.	BigDatalog	Myria	Soufflè	RecStep
(1) SG	*Hep-Th*	74.618.689	71s	911s	1378s	55s	1345s
	Grid150	2.295.050	36s	2530s	1600s	16s	61s
	G5K	10.427.944	34s	195s	1386s	15s	65s
	G20K	279.694.744	153s	6873s	7700s	130s	16149s
(2) TC	*Tree17*	23.381.118	5s	150s	36s	14s	55s
	Grid250	984.328.125	41s	4244s	2162s	638s	2820s
	G40K	529.405.185	92s	5865	7316s	240s	5620s
	Astro-Ph	320.520.848	73s	3748s	4320s	160s	6270s
(3) Tri-C	*Cond-Mat*	173.361	18s	26s	120s	3s	TOE
	Hep-Ph	3.358.499	20s	30s	146s	6s	TOE
	Astro-Ph	1.351.441	24s	27s	153s	5s	TOE
	LiveJournal	112.319.229	115s	150s	7253s	135s	TOE
(4) ASP	*G5K*	1.880.768	2s	260s	ONW	12s	ONW
	G10K	5.496.016	1s	289s	ONW	13s	ONW
	G20K	80.765.694	9s	8780s	ONW	105s	ONW
	Grid150	131.675.775	6s	25140s	ONW	202s	ONW
(5) N2C	*Tree17*	23.381.118	11s	104s	132s	13s	53s
	Grid250	984.328.125	40s	9693s	9743s	700s	2288s
	G40K	1.013.868.830	203s	53150s	53650s	480s	9699s
	Hep-Th	149.238.388	13s	1688s	1831s	93s	537s

Figure 9: Execution time comparison with other Datalog systems (TOE: "Time out exceeded"; ONW: "Ontology not working").

Scenario	Cores	Speed-up
Doctors (1 mln)	64	x10
DoctorsFD (1 mln)	64	x10
STB-128 (150k)	64	x9
Ontology-256 (1 mln)	64	x4
LUBM (120 mln)	64	x10

Ont.	Dataset	CPU	GPU
(1) SG	*Hep-Th*	101s	35s
	Grid150	70s	27s
	G5K	66s	25s
	G20K	210s	78s
(3) Tri-C	*Cond-Mat*	51s	6s
	Hep-Ph	55s	9s
	Astro-Ph	61s	11s
	LiveJournal	163s	20s

Figure 10: Vadalog Parallel speed-up (top table) and CPUs vs GPUs comparison (bottom table).

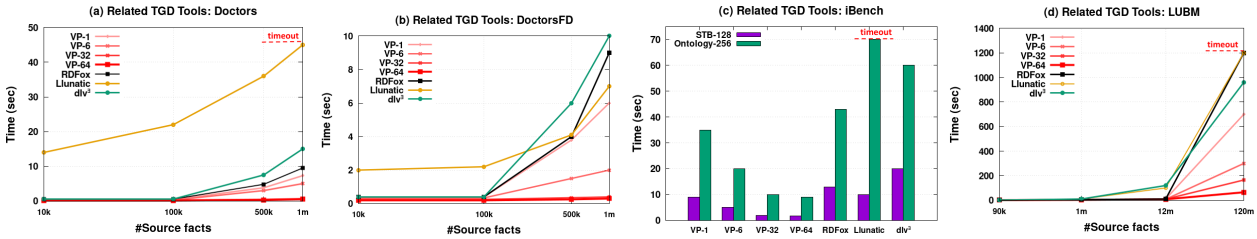

Figure 11: Experiment results for TGD-based tools.

processors (in Figure 11 VP-n stands for Vadalog Parallel with n processors) and RDFox with 32 cores.

Results. Vadalog Parallel efficiently distributes the workload among all processors, demonstrating a noteworthy performance enhancement from 1 processor (VP-1) to 64 processors (VP-64) in all scenarios and outperforming the other systems. In Figure 10 (top table) we report the speed-up of Vadalog Parallel, namely, the ratio between the execution times with 1 processor and 64 processors. LLunatic exhibits the poorest performance in all the scenarios except for DoctorsFDs and STB-128. The performance gap between LLunatic and the other systems is evident in Doctors and LUBM, where it exceeds the 1200 seconds timeout. This can be attributed to the cost of continuous disk accesses in contrast to RAM-based chase implementations. DLV$^{\exists}$ cannot scale the workload for larger inputs as it is based on a centralized SNE and exhibits the worst performance on DoctorsFD (1 mln) and LUBM (120 mln). RDFox uses a parallel model [42] in which each processor autonomously consumes facts from D. It employs dynamic scheduling of the TGD applications, assigned to a processor as soon as it becomes available. The generated facts are then stored in a centralized indexed RDF storage. This evaluation demonstrates notable efficiency gains in the Doctors and STB-128 scenarios. However, RDFox struggles in DoctorsFD and LUBM, particularly when dealing with larger input sizes. We attribute these inefficiencies to the concurrent update process of the RDF storage, which becomes susceptible to race conditions (lock) as the size of D and the number of TGDs substantially grow.

6.4 Validation on Production Scenarios

In this section, we experiment Vadalog Parallel to solve complex real-world problems from industrial settings of our partners. We show that thanks to homomorphic decomposability, our system outperforms other parallel and distributed implementations by 10x or 100x in terms of execution time and memory footprint.

Scenario 1: Close Links (CL). Given an ownership graph, this ontology (Figure 7-(6)) models the existence of a direct or indirect link between companies, based on a high overlap of shares tw.

Scenario 2: Company Control (CCTR). This scenario (Figure 7-(7)) consists in determining who takes decisions in a company, that is, who controls the majority of its votes.

Scenario 3: PSC and Strong Links (SL). The ontology *Persons with Significant Control (PSC)* (Figure 7-(8,9)) identifies the set of the persons that directly or indirectly have some control over a company. We then compute the *Strong Links (SL)*, i.e., two companies that share more than N PSC of the same company z.

Scenario 4: Propagation of Defaults (DP). This ontology (Example 1.1) simulates the consequences of defaults given the network of financial exposures in which the companies are involved.

Parallel Baselines and Datasets. We compare our system with ad-hoc efficient implementations of the problems in scenarios: (1) Spark [47] denotes the SparkSQL implementations featuring caching of the delta relations produced in each iteration, storing intermediate checkpoints and forcing the broadcast joins, replicating the extensional datasets in every processor. (2) Flink [19] is a Flink-based implementation adopting the ad-hoc Flink's `DataSet`

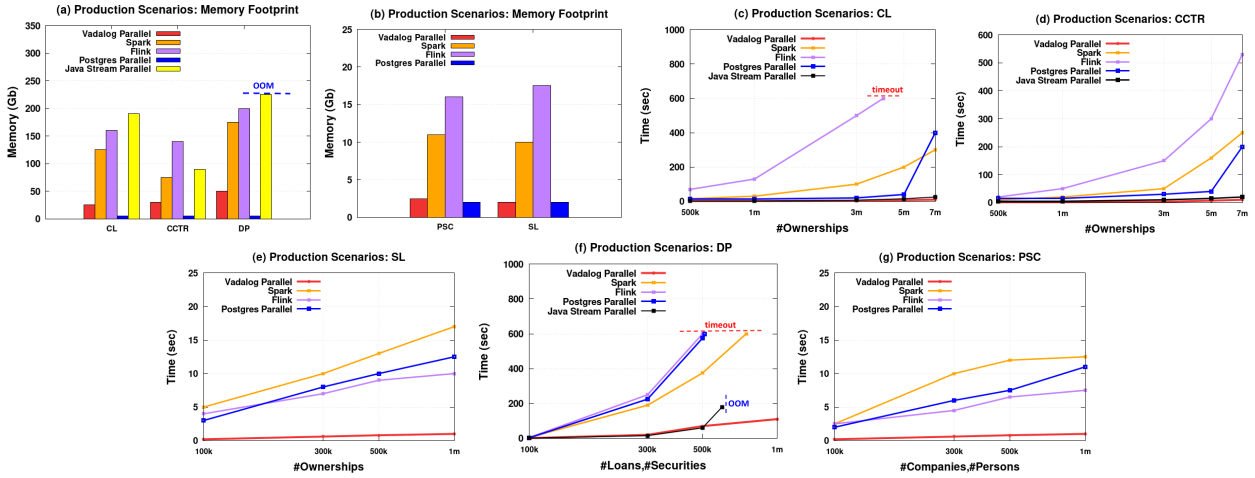

Figure 12: Experiment results for Production Scenarios.

construct `DeltaIteration`, that efficiently supports iterative incremental computations. It maintains a single incremental `DataSet` across the iterations and we use it only for scenarios 3 and 4. (3) Postgres Parallel is SQL-based and parallel implementation in PostgreSQL [54] adopting parallel SNE with materialized views and implementing labelled nulls and the isomorphic chase. (4) Parallel Java Stream is a high-performance graph-based implementation using parallel Java streams with the `ForkJoinPool` framework, it constructs the query answer exploiting parallel graph visits.

For CL, CC we selected as input the knowledge graph of the Italian companies [8] having person-company and company-company shareholding relationships, counting 7 mln nodes and 6.8 mln edges. For PSC and SL we use real-world data extracted from DBpedia [22] about companies (100k), persons (1 mln), company-control relationships (50k) and company key persons (10k). For DP we constructed an artificial weighted network of financial entities interacting via loans or security relationships (2 mln). We also vary the input sizes.

Results. Vadalog Parallel outperforms all the ad-hoc implementations in all scenarios up to 100x in terms of execution time (less than 20 seconds for CL, CCTR, PSC and SL, and less than 150 seconds for DP on all input sizes), as shown in Figure 12-(c-g). The only solution with comparable performance is Java Stream Parallel as it is based on a parallel algorithm which does not require any communication exchanged among the threads. Java Streams adopts the work-stealing technique. This is possible only for multi-threaded implementations in the same JVM and cannot be used in a distributed environment. Despite its low overhead, Java Stream Parallel is inefficient in terms of memory footprint compared to Vadalog Parallel and goes out of memory (OOM) when the generated facts are considerable, as in the case of DP, with about 1000 mln of output facts and more than 250 GB of memory used (Figure 12-(a,f)). Vadalog Parallel adopts the efficient serialization mechanism of Spark SQL and uses, as processor-local data structures, the optimized collections from the *fast-util* library [55]. Flink exhibits the slowest performance for scenarios not implemented with `DeltaIteration` and experiences timeouts in both the CL and DP scenarios. Although it is marginally faster than Spark and Postgres Parallel in PSC and SL, scenarios with smaller query answer sizes, Flink lags behind other solutions in DP, the most computationally intensive task, despite benefiting from the recursion optimization of `DeltaIteration`. On the other hand, Spark outperforms all other implementations in the most data-intensive scenarios (except for Vadalog Parallel and Java Stream Parallel), such as DP and CL. However, its performance is hindered by the continuous communication overhead required in each recursion round for the deduplication operations. Postgres Parallel outperforms Flink and Spark on smaller datasets for efficient updates of indexed tables, but its performance degrades on larger datasets due to the demanding task of continuously updating concurrent indexes for recursive and aggregation predicates.
This implementation only uses secondary memory and stands out for its efficiency in terms of memory footprint (Figure 12-(a,b)).

Scaling up with GPUs. For non-homomorphically decomposable TGD sets evaluated via DW-SNE, requiring fact exchanges between processors, our system leverages GPUs through Spark-Rapids [45], NVIDIA's GPU-accelerated Spark core, optimizing shuffle efficiency. We benchmarked using Spark-RAPIDS 24.02.0 and CUDA 12.0 on an Amazon EC2 AMI p3.16xlarge machine with 64 vCPUs, 8 NVIDIA V100 GPUs (16 GB memory each), and 488 GB RAM. Results in Figure 10 (bottom table) show up to a 6x speedup over CPU execution.

7 CONCLUSION

In this paper, we introduced Vadalog Parallel, a novel framework designed for distributed reasoning with Datalog$^{\pm}$. Looking at extensions, we aim to expand Vadalog Parallel in extending its expressive power by capturing other advanced reasoning features, such as the distributed evaluation of Equality-generating Dependencies (EGDs) and distributed temporal reasoning.

ACKNOWLEDGMENTS

This work has been supported by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG18013, 10.47379/NXT22018, 10.47379/ICT2201]. The Vadalog Parallel system as presented here is the intellectual property of Prometheux Limited.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[2] Foto N. Afrati and Jeffrey D. Ullman. 2012. Transitive Closure and Recursive Datalog Implemented on Clusters. In *EDBT*.

[3] Tommaso Alfonsi, Luigi Bellomarini, Anna Bernasconi, and Stefano Ceri. 2022. Expressing Biological Problems with Logical Reasoning Languages. In *RuleML+RR*.

[4] Mario Alviano and Andreas Pieris (Eds.). 2022. *4th International Workshop on the Resurgence of Datalog in Academia and Industry*.

[5] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*.

[6] Patricia C. Arocena, Boris Glavic, Radu Ciucanu, and Renée J. Miller. 2015. The iBench Integration Metadata Generator. In *PVLDB*.

[7] Paolo Atzeni, Teodoro Baldazzi, Luigi Bellomarini, and Emanuel Sallinger. 2022. iWarded: A Versatile Generator to Benchmark Warded Datalog+/−Reasoning. In *International Joint Conference on Rules and Reasoning*.

[8] Paolo Atzeni, Luigi Bellomarini, Michela Iezzi, Emanuel Sallinger, and Adriano Vlad. 2020. Augmenting Logic-based Knowledge Graphs: The Case of Company Graphs.. In *KR4L@ ECAI*.

[9] Teodoro Baldazzi, Luigi Bellomarini, Emanuel Sallinger, and Paolo Atzeni. 2021. Eliminating Harmful Joins in Warded Datalog+/-. In *International Joint Conference on Rules and Reasoning*.

[10] Pablo Barceló and Reinhard Pichler (Eds.). 2012. *Datalog in Academia and Ind*.

[11] Luigi Bellomarini, Davide Benedetto, Matteo Brandetti, Emanuel Sallinger, and Adriano Vlad. 2024. Appendix. https://drive.google.com/file/d/1ZSMFUrEMmDrFYHR7C_RiQJoZ29gn2G_L/view?usp=sharing [Online; July-2024].

[12] Luigi Bellomarini, Daniele Fakhoury, Georg Gottlob, and Emanuel Sallinger. 2019. Knowledge Graphs and Enterprise AI: The Promise of an Enabling Technology. In *ICDE*.

[13] Luigi Bellomarini, Emanuel Sallinger, and Georg Gottlob. 2018. The Vadalog System: Datalog-based Reasoning for Knowledge Graphs. *VLDB*.

[14] Michael Benedikt, George Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. 2017. Benchmarking the chase. In *SIGMOD*.

[15] Andrea Calì, Georg Gottlob, and Michael Kifer. 2013. Taming the Infinite Chase: Query Answering under Expressive Relational Constraints. *Journal of Artificial Intelligence Research*.

[16] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. 2009. A general datalog-based framework for tractable query answering over ontologies. In *PODS*.

[17] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. 2010. Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *LICS*.

[18] Andrea Calì, G. Gottlob, and A. Pieris. 2012. Towards more expressive ontology languages: The query answering problem. *Journal of Artificial Intelligence*.

[19] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*.

[20] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). In *IEEE Transactions on Knowledge and Data Engineering*.

[21] Stavros S. Cosmadakis and Paris C. Kanellakis. 1986. Parallel Evaluation of Recursive Rule Queries. In *SIGMOD*.

[22] DBpedia. 2023. DBpedia tables. http://wiki.dbpedia.org/services-resources/downloads/dbpedia-tables. [Online; 27-Dec-2023].

[23] Owen P Dwyer, Teodoro Baldazzi, Jim Davies, Emanuel Sallinger, and Adriano Vlad. 2023. Reasoning over Health Records with Vadalog: a Rule-based Approach to Patient Pathways. In *RuleML+RR*.

[24] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theoretical Computer Science* (2005).

[25] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-up in-Memory Datalog Processing: Observations and Techniques. In *VLDB*.

[26] Sumit Ganguly, Avi Silberschatz, and Shalom Tsur. 1990. A Framework for the Parallel Processing of Datalog Queries. In *SIGMOD*.

[27] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2014. That's All Folks! LLUNATIC Goes Open Source. In *PVLDB*.

[28] Georg Gottlob and Andreas Pieris. 2015. Beyond SPARQL under OWL 2 QL Entailment Regime: Rules to the Rescue. In *IJCAI*.

[29] Todd J Green, Dan Olteanu, and Geoffrey Washburn. 2015. Live programming in the LogicBlox system: A MetaLogiQL approach. In *VLDB*.

[30] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. 2005. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*.

[31] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *7th Python in Science Conference*.

[32] Daniel Halperin, Victor Teixeira de Almeida, et al. 2014. Demonstration of the Myria big data management service. In *SIGMOD*.

[33] Aidan Hogan et al. 2022. Knowledge Graphs. In *ACM Computing Surveys*.

[34] Muhammad Imran, Gábor E Gévay, and Volker Markl. 2020. Distributed graph analytics with datalog queries in flink. In *Software Foundations for Data Interoperability and Large Scale Graph Data Analytics*.

[35] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard Scholz. 2019. Brie: A specialized trie for concurrent datalog. In *10th International Workshop on Programming Models and Applications for Multicores and Manycores*.

[36] Paris C. Kanellakis. 1986. Logic programming and parallel complexity. In *ICDT*.

[37] Bas Ketsman, Aws Albarghouthi, and Paraschos Koutris. 2020. Distribution Policies for Datalog. In *Theory of Computing Systems*.

[38] Craig Larman et al. 1998. *Applying UML and patterns*. Prentice Hall Upper Saddle River.

[39] Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. 2012. Efficiently Computable Datalog∃ Programs. In *Principles of Knowledge Representation and Reasoning*.

[40] Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. 2019. Fast Query Answering over Existential Rules. In *ACM Transaction on Computational Logic*.

[41] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. 1979. Testing Implications of Data Dependencies. In *ACM Transactions on Database Systems*.

[42] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2015. Incremental update of datalog materialisation: the backward/forward algorithm. In *AAAI Conference on Artificial Intelligence*.

[43] Boris Motik, Yavor Nenov, Robert Piro, Ian Horrocks, and Dan Olteanu. 2014. Parallel materialisation of datalog programs in centralised, main-memory RDF systems. In *AAAI Conference on Artificial Intelligence*.

[44] Walaa Eldin Moustafa, Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. 2016. Datalography: Scaling datalog graph analytics on graph processing systems. In *IEEE*.

[45] Nvidia. 2023. NVIDIA RAPIDS Accelerator for Apache Spark. https://resources.nvidia.com/en-us-spark. [Online; 25-May-2024].

[46] Jignesh M Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: A data platform based on the scaling-up approach. In *VLDB*.

[47] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics*.

[48] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On fast large-scale program analysis in datalog. In *25th International Conference on Compiler Construction*.

[49] Jürgen Seib and Georg Lausen. 1991. Parallelizing Datalog Programs by Generalized Pivoting. In *PODS*.

[50] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *SIGMOD*.

[51] Alexander Shkapsky, Mohan Yang, and Carlo Zaniolo. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *International Conference on Data Engineering*.

[52] Stanford University. 2007. Stanford Large Network Dataset Collection: Collaboration Network. https://snap.stanford.edu/data/#canets. [Online; Dec-2023].

[53] Stanford University. 2008. Stanford Large Network Dataset Collection: LiveJournal. https://snap.stanford.edu/data/soc-LiveJournal1.html. [Online; Dec-2023].

[54] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. In *ACM Sigmod Record*.

[55] Unimi. 2023. Fastutil. http://fastutil.di.unimi.it/. [Online; 27-Dec-2023].

[56] Victor Vianu. 2021. Datalog Unchained. In *PODS*.

[57] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing recursive queries with progam synthesis. In *International Conference on Management of Data*.

[58] O. Wolfson. 2000. Sharing the Load of Logic-Program Evaluation. In *First International Symposium on Databases in Parallel and Distributed Systems*.

[59] Ouri Wolfson and Aya Ozeri. 1990. A New Paradigm for Parallel and Distributed Rule-Processing. In *International Conference on Management of Data*.

[60] Ouri Wolfson and Avi Silberschatz. 1988. Distributed Processing of Logic Programs. In *SIGMOD*.

[61] Jiacheng Wu, Jin Wang, and Carlo Zaniolo. 2022. Optimizing Parallel Recursive Datalog Evaluation on Multicore Machines. In *SIGMOD*.

[62] Mohan Yang and Carlo Zaniolo. 2014. Main memory evaluation of recursive queries on multicore machines. In *International Conference on Big Data*.

[63] Weining Zhang, Ke Wang, and Siu-Cheung Chau. 1995. Data Partition and Parallel Evaluation of Datalog Programs. In *IEEE Trans. Knowl. Data Eng*.