The Vadalog Parallel System: Distributed Reasoning with Datalog+/- (extended abstract)

Luigi Bellomarini[1], Davide Benedetto[2,3], Matteo Brandetti[1,4,*], Emanuel Sallinger[4,5] and Adriano Vlad[2,4,5]

[1]*Banca d'Italia, Italy*
[2]*Prometheux, London, United Kingdom*
[3]*Università Roma Tre, Department of Computer Science and Engineering, Rome, Italy*
[4]*TU Wien, Faculty of Informatics, Vienna, Austria*
[5]*University of Oxford, Department of Computer Science, Oxford, UK*

Abstract

Over the past years, ontological reasoning systems based on Datalog+/- languages, such as Vadalog, have gained popularity for their effectiveness in modeling real-world problems. As data analytics tasks grow in scale and complexity, distributing computation across multiple non-communicating processors has become essential for scalability. However, avoiding duplication and guaranteeing termination in a distributed setting is challenging due to the infinite generation of new symbols and facts from existential quantification and recursion. In this paper, we tackle these challenges by introducing the first distributed framework for Datalog+/- based on homomorphic decomposability, a novel condition that ensures favorable distribution properties. We implement this in Vadalog Parallel, a distributed reasoner for Vadalog, and provide experimental evaluation against state-of-the-art systems.

1. Introduction

Recent years have witnessed increasing interest in logic-based ontological reasoning, driven by the resurgence of reasoning frameworks based on *Datalog*$^{\pm}$ [1], such as Vadalog [2]. These frameworks leverage key *reasoning features*, including existential quantification and recursion, enabling efficient graph traversals and supporting SPARQL queries under the OWL 2 QL entailment regime for the semantic web [3]. Datalog$^{\pm}$ has been applied in diverse domains such as finance, medicine, biology, and enterprise resource planning [1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14], with emerging dedicated venues [15].

As data scales and analytical tasks grow in complexity, distributing reasoning workloads across multiple processors becomes essential. While parallel techniques using shared memory and message passing have been effective for pure Datalog [16, 17, 18, 19, 20, 21, 22, 23], the challenges posed by existential quantification and ontological reasoning in Datalog$^{\pm}$ remain largely unexplored. Datalog$^{\pm}$ extends the expressiveness of Datalog by introducing existential quantification to support ontological reasoning. The reasoning process consists in answering a

XXX

*Corresponding author.
luigi.bellomarini@bancaditalia.it (L. Bellomarini); davben@prometheux.co.uk (D. Benedetto); matteo.brandetti@bancaditalia.it (M. Brandetti); sallinger@dbai.tuwien.ac.at (E. Sallinger); adriano@prometheux.co.uk (A. Vlad)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-03-13

conjunctive query (CQ) Q over a database D, where D is enriched with new facts generated through the application of a set of Datalog$^{\pm}$ rules Σ.

Example 1. *Consider a database D modeling an "exposure network," where financial entities are nodes with associated default probabilities p (FinEntity(a, p)), and entities are linked by loans (Loan) and securities (Security) issued by other financial entities. Entities in D may be affected by default events d initiated by other entities (Dflt), with the following rules Σ defining how these default events propagate.*

$$\textit{FinEntity}(a, p), p > 0.5 \rightarrow \exists d\ \textit{Dflt}(a, a, d) \qquad (\sigma_1)$$

$$\textit{Dflt}(b, a, d_1), \textit{Loan}(b, c) \rightarrow \exists d_2\ \textit{Dflt}(c, a, d_2) \qquad (\sigma_2)$$

$$\textit{Dflt}(b, a, d_1), \textit{Security}(b, c) \rightarrow \exists d_2\ \textit{Dflt}(c, a, d_2). \qquad (\sigma_3)$$

These rules state that an entity with a default probability greater than 50% initiates a default event, and entities that have granted loans or hold securities from defaulting entities will also default.

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 [24]. 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.

In the context of a parallel execution, we explore how the evaluation can be distributed across two processors. For example, consider the distribution plan where a processor P_1 handles larger entities (e.g., A): $\textit{FinEntity}(a, p), \textit{Large}(a), p > 0.5 \rightarrow \exists d\ \textit{Dflt}(a, a, d)$; and a processor P_2 handles smaller ones (e.g., C): $\textit{FinEntity}(a, p), \textit{Small}(a), p > 0.5 \rightarrow \exists d\ \textit{Dflt}(a, a, d)$. This partitioning enables non-redundant computation by ensuring that each processor handles distinct facts; it also ensures that no facts are redundantly computed across processors.

Distribution Properties. A distribution plan relies on *partitioning properties*, which ensure both the efficiency and correctness of the computation. First, *non-redundancy in the computation* implies that each processor should produce distinct facts. Second, *completeness of the parallel computation* guarantees that no rule should be triggered by facts that are distributed across different partitions. This ensures that when the facts from all partitions are merged, they produce the same result as a serial execution, preserving correctness. An improper partitioning can lead to incorrect results.

Challenges. Ensuring non-redundancy is difficult with existential quantification because labelled nulls, which are unknown values generated at runtime, complicate partitioning strategies based on constant values. Termination is also challenging, especially in the presence of recursive TGDs with existential quantification, where the chase can generate infinitely many labelled nulls. Various terminating variants of the chase have been proposed in specific Datalog$^{\pm}$ languages; such variants feature *applicability conditions* that preempt the creation of homomorphic facts during the chase—i.e., there exists a constant-preserving mapping between the terms of a new fact to the ones of an existing one. The applicability conditions are affected by distribution since homomorphic facts can be produced in different processors.

Homomorphically Decomposable TGDs. We identify the new class of homomorphically decomposable TGDs that guarantees the mentioned distribution properties, i.e., the existence of a partitioning criterion for D such that two processors never generate two homomorphic facts, making this class particularly suitable for distributed reasoning. This paper offers several **contributions**:

- The characterization of **homomorphically decomposable** TGDs; we identify a **sufficient condition** as well as a **partitioning strategy** of the input database for the parallel evaluation.
- A concrete **application** of our techniques to *Warded Datalog*$^{\pm}$ [3], an expressive and tractable language of the Datalog$^{\pm}$ family modeling a variety of real-world problems.
- The **implementation** of such underpinnings in *Vadalog Parallel*, a new system for distributed ontological reasoning adopting the VADALOG language, and a full-scale **experimental evaluation** of Vadalog Parallel in a variety of real-world and synthetic scenarios.

Related Work. Our research is the first to explore parallel algorithms for ontological reasoning with Datalog$^{\pm}$. Previous work primarily focuses on the parallel evaluation of pure Datalog, without existential quantification, such as studies on distribution policies, decomposability, and parallel frameworks for Datalog [16, 21, 25, 26, 20, 27, 28]. Additionally, the load sharing scheme [25] provides a relaxed decomposability concept applicable to broader Datalog programs. A recent theoretical framework for Datalog defines evaluation strategies based on processor policies [16].

Overview. This work is the extended abstract of a paper published at VLDB [29], where we present the key ideas of *Vadalog Parallel*, a distributed reasoning system for Datalog$^{\pm}$, with a focus on its theoretical foundations and experimental evaluation. This paper is organized as follows. In Section 2, we provide the background. In Section 3, we discuss homomorphic decomposability. In Section 4 we illustrate the implementation of Vadalog Parallel and experiments. Further content is available in the Appendix [30].

2. Background

Let $\mathbf{C}$, $\mathbf{N}$, and $\mathbf{V}$ represent disjoint countably infinite sets of constants, labelled nulls, and variables, respectively. A relational schema $\mathbf{S}$ is a set of predicates with associated arities. An atom is an expression $R(\bar{v})$, where $R \in \mathbf{S}$ is a predicate and $\bar{v}$ is an n-tuple of terms. A database (instance) D associates a relation to each predicate in $\mathbf{S}$ over the domain of constants and nulls, denoted as $dom(D)$. A homomorphism is a constant-preserving mapping h from atom a_1 to atom a_2 if $h(a_1) = a_2$. Two atoms are isomorphic if h is a bijection. A conjunctive query (CQ) Q over a schema $\mathbf{S}$ is an implication $q(\mathbf{x}) \leftarrow \phi(\mathbf{x}, \mathbf{y})$, where $\phi(\mathbf{x}, \mathbf{y})$ is a conjunction of atoms, and $q(\mathbf{x})$ is an n-ary predicate not in $\mathbf{S}$. A CQ Q is satisfied in D if there exists a homomorphism h from the atoms in $\phi(\mathbf{x}, \mathbf{y})$ to the facts in D. A set of Datalog$^{\pm}$ rules Σ includes tuple-generating dependencies (TGDs), which are first-order implications of the form $\forall \mathbf{x} \phi(\mathbf{x}) \rightarrow \exists \mathbf{z} \psi(\mathbf{y}, \mathbf{z})$. The chase is a procedure that expands a database D by iteratively applying TGDs to infer new facts until a fixpoint is reached, generating $chase(D, \Sigma)$.

3. Vadalog Parallel

This section outlines the theoretical foundations and algorithms supporting our reasoning approach.

Homomorphic Decomposability. An *evaluable atom* is a comparison condition using standard operators (e.g., $=, \neq, <, >, \leq, \geq$), including algebraic expressions over TGD body variables. For example $p > 0.5$ in Example 1. A *restricted TGD* contains an evaluable atom in its body. A *restricted copy* of a TGD σ is obtained by adding an evaluable atom in its body. Evaluable atoms allow us to limit the applicability of TGDs. In fact, given a database D, a restricted TGD $\sigma : \boldsymbol{\phi}(\mathbf{x}), \boldsymbol{\xi}(\mathbf{w}) \rightarrow \exists \mathbf{z}\, \boldsymbol{\psi}(\mathbf{y}, \mathbf{z})$ 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 $\Pi(\mathbf{w}) = hash(\mathbf{w})\ mod\ k$. The restricted copies of σ are given by:

$$FinEntity(a, p), \Pi(a) = i \rightarrow \exists d\ Dflt(a, a, d)\ \ \forall i \in \{0, \ldots, k-1\} \qquad (\sigma_i)$$

We can represent a set of restricted TGDs with the same left-hand side and real numbers for processors as a *parametric restricted TGD*, adding a conjunct $Partition(\mathbf{w})$ in the body of the TGD, defined piecewise depending on the processor i: $Partition(\mathbf{w})$ = TRUE if $\Pi(\mathbf{w}) = i$, with $i < k$; or FALSE otherwise. Thus, for our example, we would have $FinEntity(a, p), Partition(a)$. The set Σ_i obtained by replacing restricted TGDs in Σ with their restricted copy for processor i is called the *restricted set*. A restricted set that ensures non-redundant and complete chase instances does not always exist.

Example 2. *Two banks in the same country are supervised by the same National Central Bank (NCB).*

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

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

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

For $k = 2$, processor $i = 0$ generates $NCB(A, \nu_1)$ via σ_1^0 and $NCB(C, \nu_2)$ via σ_2, while $i = 1$ generates $NCB(B, \nu_3)$ via σ_1^1 and $NCB(C, \nu_4)$ via σ_2. This distribution is redundant as different banks sharing a country lead to homomorphically equivalent facts across processors (e.g., $NCB(C, \nu_2)$ and $NCB(C, \nu_4)$).

A set of TGDs Σ is *homomorphically decomposable* if there exist $k > 1$ restricted sets generating non-overlapping and complete chase instances.

Definition 1 (Homomorphically Decomposable TGDs). *A set of TGDs Σ is homomorphically decomposable if there exist $k > 1$ restricted sets $\Sigma_0, \ldots, \Sigma_{k-1}$ satisfying:*

- *NON-TRIVIALITY. $\exists D$ such that $\forall i$, $chase(D, \Sigma_i) \neq \emptyset$.*
- *NON-MAPPABILITY. $\forall D, i \neq j, \nexists \theta$ mapping $f_1 \in chase(D, \Sigma_i)$ to $f_2 \in chase(D, \Sigma_j)$ with $f_1, f_2 \notin D$.*

- *Homomorphic Equivalence. chase(D, Σ) and $\bigcup_{i=1}^{n}$ chase(D, Σ_i) are homomorphically equivalent.*

In Example 2, the non-mappability condition fails as *NCB*(C, ν_3) maps to *NCB*(C, ν_4), making the TGDs non-homomorphically decomposable; TGDs in Example 1 are instead decomposable.

Since determining whether a set of TGDs Σ is homomorphically decomposable is undecidable, we introduce a sufficient syntactic condition which also allows to identify a chase partitioning that does not depend on Σ, but only on the input D. In fact, some partitions induced by restricted sets Σ_i may be fixed and unbalanced, leading to inefficient processing like in the following example.

Example 3. *Consider a set Σ modeling interactions (dependencies) between methods in a Java project and detecting simple recursions (r) i.e., when a method invokes itself via self-dependencies (σ_2).*

$$\textit{Call}(x, y), \textit{Call}(y, x) \rightarrow \exists d\ \textit{Dependency}(d, x, y) \qquad (\sigma_1)$$
$$\textit{Dependency}(d, x, x) \rightarrow \exists r\ \textit{SimpleRecursion}(x, d, r) \qquad (\sigma_2)$$

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.

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 a partitioning of the chase facts with k, evenly distributed.

Example 4. *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.*

$$\textit{Researcher}(x), \textit{CR}(x, y), \textit{Partition}(x, y) \rightarrow \exists kr\ \textit{Influence}(kr, x, x, y) \qquad (\sigma_1)$$
$$\textit{Influence}(kr, x, y, z), \textit{CR}(z, y) \rightarrow \textit{Influence}(kr, x, z, y) \qquad (\sigma_2)$$

From the partitioning in Example 4, 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. In the TGDs of Example 4 there exist a set of *harmless* variables that never bind to labelled nulls and consistently propagate through non-affected positions in all TGDs of Σ. This ensures that the facts generated by different chase sequences remain distinct, preventing homomorphisms between them. The harmlessness property is crucial, as binding to a labelled null could invalidate non-mappability. 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 $v \in \mathit{body}(\sigma) \cap \mathit{head}(\sigma)$, as a non-affected position where v appears.

Theorem 3.1 (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 satisfy Theorem 3.1 with the shared eligible propagation position *Dflt*[0], as well as the TGDs in Example 4, with positions *Influence*[1], *Influence*[2], and *Influence*[3]. There is no shared eligible propagation position for the TGDs in Example 3.

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 $\mathit{Partition}(\mathbf{x})$ on the set of variables $\mathbf{x}$ appearing in shared eligible propagation positions in $\mathit{head}(\sigma')$, i.e., the positions satisfying Theorem 3.1 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. In the body-ground TGD σ_1 of Example 4, 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 constants in these positions (i.e., *partitioning positions*) can be used to define a distribution key of D.

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 our experience, many practical scenarios can be modeled with Warded TGDs that are also homomorphically decomposable. With Warded TGDs, query answering on D under Σ can be equivalently performed over a finite chase which we name chase^W, that is obtained by activating TGDs of Σ only if they generate facts are not isomorphic to others already in chase^W.

Algorithm 1 provides the full procedure (Parallel-Evaluate) to perform ontological reasoning with a set of Warded TGDs Σ enjoying homomorphic decomposability. Θ is a dictionary of predicates 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 chase^W for Σ

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).

Algorithm 1 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
```

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) [31], that supports *chase*^{W} and 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.

4. Implementation and Experiments

We introduce the novel system Vadalog Parallel, implementing homomorphic decomposability and DW-SNE. For the specification of TGDs, the system adopts the Vadalog language [2].

Architecture Description. Given a CQ $Q : q(\mathbf{x}) \leftarrow \phi(\mathbf{x}, \mathbf{y})$, a set of TGDs Σ, and a database D, query evaluation under Σ consists of four phases, each managed by a dedicated module.

1. The *compiler* checks Q and Σ for compliance with the Vadalog grammar.
2. The *logic optimizer* normalizes Σ and verifies if it is Warded and homomorphically decomposable.
3. The *planner* generates an execution plan: *Replicated Streaming Pipeline*, if Σ is homomorphically decomposable or DW-SNE otherwise (both explained in the next sections).
4. The *execution engine* evaluates Q using the execution plan and writes output facts to external sources.

Replicated Streaming Pipeline (RSP). Applied when Σ is homomorphically decomposable, this model (Algorithm 1) constructs independent *streaming pipelines* from the predicate dependency graph of Σ. Atoms serve as filters, connected by pipes representing TGD transformations. Data flow from extensional atoms to query body atoms ϕ through selections, projections, joins, and value inventions. Termination checks use local hash tables for *isomorphism checks*, ensuring only necessary chase steps are activated. Labelled nulls are generated using fresh symbols derived from partitioning constants and processor-local indices. Extensional predicates in

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 1: 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.

non-ground TGDs are indexed on join keys. Once all processors reach a fixpoint, the master processor collects output facts to compute the final answer.

DW-SNE. If Σ is not homomorphically decomposable, DW-SNE is used. Implemented via the *template method* pattern [32], it provides SQL-like interfaces for manipulating distributed data structures (i.e., Spark `Dataset`) [33]. The algorithm iteratively applies DW-SNE operations composed via interface procedures, leveraging Spark `Dataset` for in-memory Map-Reduce transformations.

Benchmarks. We first compare our system with other distributed or parallel systems on various graph traversal problems. Secondly, we consider other TGD-based systems, then, we stress the scalability of our system and show that outperforms ad-hoc implementations in data-intensive problems.

Comparison with Parallel Datalog Systems. Existing parallel Datalog systems primarily support pure Datalog with standard and monotonic aggregations but lack ontological query answering under Datalog$^{\pm}$. Thus, our comparison focuses on pure Datalog and monotonic aggregation scenarios. We evaluate both *shared-nothing* systems (BigDatalog [23], Myria [34]) and *shared-memory* systems (Soufflè [35], RecStep [36]). BigDatalog, based on parallel SNE, optimizes recursive Datalog queries via Spark's `SetRDD`. Myria supports iterative queries with aggregates and incremental computations. Soufflè is optimized for large-scale program analysis, using specialized parallel data structures for indexing and compression. RecStep, built on *QuickStep* [37], supports Datalog with stratified negation and aggregation. We evaluate five ontologies modeling graph-related problems (Figure 1):

- *Same Generation (SG)*: Finds node pairs (x, y) sharing a common ancestor via distinct paths.
- *Transitive Closure (TC)*: computes reachability between nodes.

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	5865s	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

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	5,000	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 2: Comparison with other Datalog systems (TOE: "Time out exceeded"; ONW: "Ontology not working") (left) and synthetic and real-world graph parameters (right).

- *Triangle Counting (Tri-C)*: counts unique triangle structures in an undirected graph.
- *All Shortest Paths (ASP)*: computes shortest paths in a weighted directed graph.
- *Non-2-Colorability (N2C)*: checks whether a graph is not 2-colorable via odd/even path computations.

We test them on both synthetic and real-world graphs of different topologies and sizes (Figure 2, right). Although input graphs vary in size, chase expansion can be quadratic in the number of nodes.

Figure 2 summarizes the performance. In homomorphically decomposable cases (TC, ASP, N2C), Vadalog Parallel achieves up to 10000x speedup over alternatives, particularly outperforming shared-nothing systems like BigDatalog and Myria. BigDatalog relies on SetRDD, requiring processors to synchronize Spark transformations at each recursion step, causing delays. Vadalog Parallel, in contrast, executes independent streaming pipelines, embedding computation within a single Spark *mapPartition* operation. Against shared-memory systems, Soufflè shows competitive performance, occasionally outperforming Vadalog Parallel in non-homomorphically decomposable cases (SG, Tri-C). However, in TC, ASP, and N2C, Vadalog Parallel remains 2x to 100x faster. Soufflè's optimizations, including the *Brie* [38] data structure for high-density relation compression, improve efficiency but become costly at high chase cardinalities ($\approx$ 1000 million facts). In cases like N2C on Grid150 and G40K, and TC on Grid250, Vadalog Parallel surpasses all competitors due to reduced indexing overhead.

Related TGD-based Tools. We analyze the advantages of parallel evaluation for ontological reasoning with TGDs by comparing Vadalog Parallel to top-performing reasoners from ChaseBench [39]. RDFox [40] is a high-performance RAM-based Datalog engine implementing a parallel, non-distributed seminaive chase, supporting existentials only under w-acyclicity [41]. LLunatic [42] is a PostgreSQL-based system for data exchange, supporting CQ-answering under weakly acyclic TGDs. DLV$^{\exists}$ [43] is a RAM-based Datalog system supporting CQ-answering under Shy TGDs using the *parsimonious* chase and SNE-based materialization. We evaluate weakly acyclic, shy, and warded TGDs from ChaseBench:

- *STB-128* (167 TGDs, 150K instances) and *ONT-256* (529 TGDs, 1M instances) from iBench [44].
- *Doctors* and *DoctorsFD*, non-recursive data integration tasks, tested with up to 1M facts and

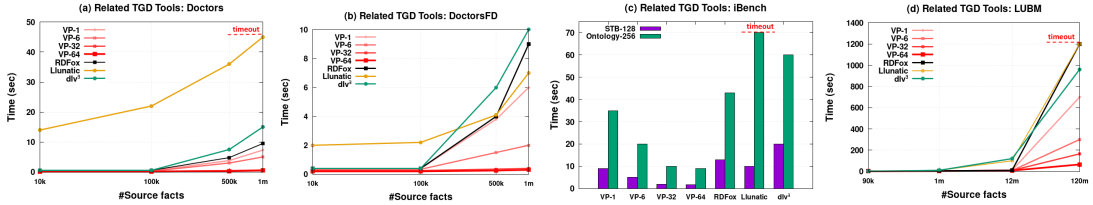

Figure 3: Experiment results for TGD-based tools.

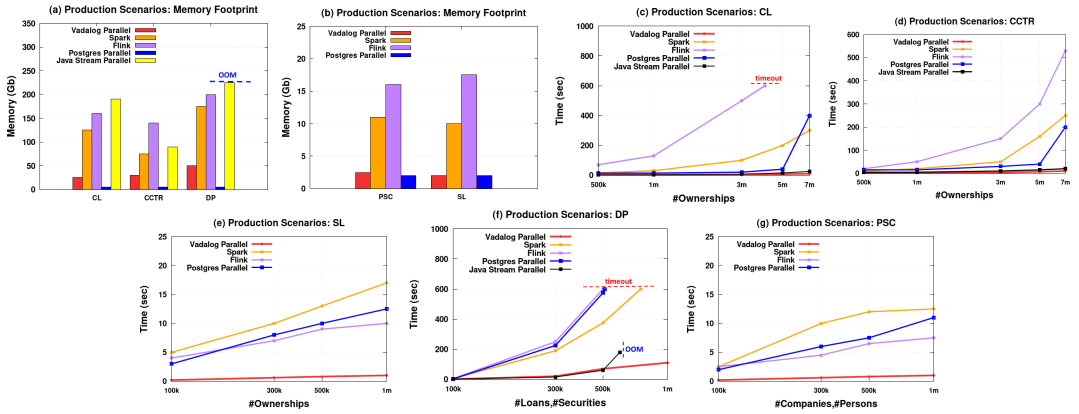

Figure 4: Experiment results for Production Scenarios.

9 CQs.

- *LUBM* [45], a university domain benchmark, tested with up to 120M facts and 14 CQs.

Execution times include input loading, chase computation, result export, and query answering. Vadalog Parallel is tested with varying processor counts (*VP-n* in Figure 3), while RDFox runs on 32 cores.

Vadalog Parallel efficiently distributes workload across processors, achieving significant speedups from 1 processor (VP-1) to 64 (VP-64) and outperforming all other systems. LLUNATIC has the worst performance in most cases, timing out (>1200s) on Doctors and LUBM due to frequent disk accesses. It performs slightly better on DoctorsFD and STB-128. DLV$^{\exists}$ struggles with scalability due to its centralized SNE, showing the worst results on DoctorsFD (1M) and LUBM (120M). RDFox employs a parallel model [46] where processors independently consume facts from D with dynamic TGD scheduling. It performs well on Doctors and STB-128 but struggles with DoctorsFD and LUBM due to race conditions in its centralized RDF storage when handling large inputs and complex TGDs.

Validation on Production Scenarios. We evaluate Vadalog Parallel on real-world industrial problems, demonstrating its superior execution time and memory efficiency (10x-100x improvement) due to homomorphic decomposability.

- *Scenario 1: Close Links (CL).* Models direct and indirect links between companies in an ownership graph based on significant shareholding overlap (Figure 1-(6)).
- *Scenario 2: Company Control (CCTR).* Determines decision-making authority within a company by analyzing majority vote ownership (Figure 1-(7)).
- *Scenario 3: Persons with Significant Control (PSC)* identifies individuals who directly or indirectly control a company; *Strong Links (SL)* detects companies sharing more than N PSCs (Figure 1-(8,9)).
- *Scenario 4: Propagation of Defaults (DP).* Discussed in Example 1.

We benchmark Vadalog Parallel against the following hoc-implementations: (1) Spark, an iterative SparkSQL [33] job that uses caching, intermediate checkpoints, and broadcast joins; (2) Flink, a job that uses the Flink [47] construct `DeltaIteration` for incremental computation (applied to PSC and DP); (3) Postgres Parallel [48], a SQL-based parallel implementation using materialized views and parallel SNE; (4) Parallel Java Stream, a graph-based parallel approach using Java's `ForkJoinPool` [49].

For CL and CCTR, we use the Italian company knowledge graph [50] (7M nodes, 6.8M edges). For PSC and SL, we extract company data from DBpedia [51] (100K companies, 1M persons, 50K company-control relationships). For DP, we generate an artificial financial network (2M entities).

Vadalog Parallel outperforms all alternatives, achieving execution times below 20s for CL, CCTR, PSC, and SL, and under 150s for DP across all input sizes (Figure 4-(c-g)). Java Stream Parallel shows comparable speed due to its thread-local execution, but it suffers from high memory usage, leading to out-of-memory (OOM) errors in DP (1000M output facts, 250GB RAM, Figure 4-(a,f)). Vadalog Parallel avoids this issue by leveraging Spark SQL's efficient serialization and optimized *fast-util* collections [52]. Flink exhibits the slowest performance in CL and DP, timing out due to inefficient recursion handling. It slightly outperforms Spark and Postgres Parallel in smaller scenarios (PSC, SL) but lags in DP, despite `DeltaIteration` optimizations. Spark excels in data-intensive tasks (DP, CL) but suffers from communication overhead during recursive deduplication. Postgres Parallel performs well on small datasets due to efficient indexed table updates but struggles with larger inputs, as frequent index updates slow recursive computations.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] A. Calì, G. Gottlob, T. Lukasiewicz, A general datalog-based framework for tractable query answering over ontologies, in: PODS, 2009.

[2] L. Bellomarini, E. Sallinger, G. Gottlob, The vadalog system: Datalog-based reasoning for knowledge graphs, 2018.
[3] G. Gottlob, A. Pieris, Beyond sparql under owl 2 ql entailment regime: Rules to the rescue, in: IJCAI, 2015.
[4] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, G. Washburn, Design and implementation of the LogicBlox system, in: SIGMOD, 2015.
[5] P. Barceló, R. Pichler (Eds.), Datalog in Academia and Ind., LNCS, 2012.
[6] A. Calì, G. Gottlob, M. Kifer, Taming the infinite chase: Query answering under expressive relational constraints, 2013.
[7] A. Calì, G. Gottlob, T. Lukasiewicz, B. Marnette, A. Pieris, Datalog+/-: A family of logical knowledge representation and query languages for new applications, in: LICS, 2010.
[8] A. Calì, G. Gottlob, A. Pieris, Towards more expressive ontology languages: The query answering problem, 2012.
[9] V. Vianu, Datalog unchained, in: PODS, 2021.
[10] W. E. Moustafa, V. Papavasileiou, K. Yocum, A. Deutsch, Datalography: Scaling datalog graph analytics on graph processing systems, in: IEEE, 2016.
[11] A. Hogan, et al., Knowledge graphs, in: ACM Computing Surveys, 2022.
[12] L. Bellomarini, D. Fakhoury, G. Gottlob, E. Sallinger, Knowledge graphs and enterprise AI: the promise of an enabling technology, in: ICDE, 2019.
[13] T. Alfonsi, L. Bellomarini, A. Bernasconi, S. Ceri, Expressing biological problems with logical reasoning languages, in: RuleML+RR, 2022.
[14] O. P. Dwyer, T. Baldazzi, J. Davies, E. Sallinger, A. Vlad, Reasoning over health records with vadalog: a rule-based approach to patient pathways, in: RuleML+RR, 2023.
[15] M. Alviano, A. Pieris (Eds.), 4th International Workshop on the Resurgence of Datalog in Academia and Industry, 2022.
[16] B. Ketsman, A. Albarghouthi, P. Koutris, Distribution policies for datalog, in: Theory of Computing Systems, 2020.
[17] Y. R. Wang, M. Abo Khamis, H. Q. Ngo, R. Pichler, D. Suciu, Optimizing recursive queries with progam synthesis, in: International Conference on Management of Data, 2022.
[18] S. Ceri, G. Gottlob, L. Tanca, What you always wanted to know about datalog (and never dared to ask), in: IEEE Transactions on Knowledge and Data Engineering, 1989.
[19] J. Wu, J. Wang, C. Zaniolo, Optimizing parallel recursive datalog evaluation on multicore machines, in: SIGMOD, 2022.
[20] W. Zhang, K. Wang, S. Chau, Data partition and parallel evaluation of datalog programs, in: IEEE Trans. Knowl. Data Eng., 1995.
[21] O. Wolfson, A. Silberschatz, Distributed processing of logic programs, in: SIGMOD, 1988.
[22] S. S. Cosmadakis, P. C. Kanellakis, Parallel evaluation of recursive rule queries, in: SIGMOD, 1986.
[23] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, C. Zaniolo, Big data analytics with datalog queries on spark, in: SIGMOD, 2016.
[24] D. Maier, A. O. Mendelzon, Y. Sagiv, Testing implications of data dependencies, in: ACM Transactions on Database Systems, 1979.
[25] O. Wolfson, Sharing the load of logic-program evaluation, in: First International Symposium on Databases in Parallel and Distributed Systems, 2000.

[26] O. Wolfson, A. Ozeri, A new paradigm for parallel and distributed rule-processing, in: International Conference on Management of Data, 1990.
[27] F. N. Afrati, J. D. Ullman, Transitive closure and recursive datalog implemented on clusters, in: EDBT, 2012.
[28] S. Ganguly, A. Silberschatz, S. Tsur, A framework for the parallel processing of datalog queries, in: SIGMOD, 1990.
[29] L. Bellomarini, D. Benedetto, M. Brandetti, E. Sallinger, A. Vlad, The vadalog parallel system: Distributed reasoning with datalog+/-, Proc. VLDB Endow. 17 (2025) 4614–4626. URL: https://doi.org/10.14778/3704965.3704970. doi:10.14778/3704965.3704970.
[30] L. Bellomarini, D. Benedetto, M. Brandetti, E. Sallinger, A. Vlad, Appendix, 2024. URL: https://drive.google.com/file/d/1ZSMFUrEMmDrFYHR7C_RiQJoZ29gn2G_L/view?usp=sharing, [Online; accessed 16-Mar-2025].
[31] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison-Wesley, 1995.
[32] C. Larman, et al., Applying UML and patterns, Prentice Hall Upper Saddle River, 1998.
[33] S. Salloum, R. Dautov, X. Chen, P. X. Peng, J. Z. Huang, Big data analytics on apache spark, 2016.
[34] D. Halperin, V. Teixeira de Almeida, et al., Demonstration of the myria big data management service, in: SIGMOD, 2014.
[35] B. Scholz, H. Jordan, P. Subotić, T. Westmann, On fast large-scale program analysis in datalog, in: 25th International Conference on Compiler Construction, 2016.
[36] Z. Fan, J. Zhu, Z. Zhang, A. Albarghouthi, P. Koutris, J. M. Patel, Scaling-up in-memory datalog processing: Observations and techniques, in: VLDB, 2019.
[37] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, S. Saurabh, Quickstep: A data platform based on the scaling-up approach, in: VLDB, 2018.
[38] H. Jordan, P. Subotić, D. Zhao, B. Scholz, Brie: A specialized trie for concurrent datalog, in: 10th International Workshop on Programming Models and Applications for Multicores and Manycores, 2019.
[39] M. Benedikt, G. Konstantinidis, G. Mecca, B. Motik, P. Papotti, D. Santoro, E. Tsamoura, Benchmarking the chase, in: SIGMOD, 2017.
[40] B. Motik, Y. Nenov, R. Piro, I. Horrocks, D. Olteanu, Parallel materialisation of datalog programs in centralised, main-memory rdf systems, in: AAAI Conference on Artificial Intelligence, 2014.
[41] R. Fagin, P. G. Kolaitis, R. J. Miller, L. Popa, Data exchange: semantics and query answering, Theoretical Computer Science (2005).
[42] F. Geerts, G. Mecca, P. Papotti, D. Santoro, That's all folks! LLUNATIC goes open source, in: PVLDB, 2014.
[43] N. Leone, M. Manna, G. Terracina, P. Veltri, Fast query answering over existential rules, in: ACM Transaction on Computational Logic, 2019.
[44] P. C. Arocena, B. Glavic, R. Ciucanu, R. J. Miller, The ibench integration metadata generator, in: PVLDB, 2015.
[45] Y. Guo, Z. Pan, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, 2005.
[46] B. Motik, Y. Nenov, R. Piro, I. Horrocks, Incremental update of datalog materialisation: the backward/forward algorithm, in: AAAI Conference on Artificial Intelligence, 2015.
[47] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink:

Stream and batch processing in a single engine, 2015.
[48] M. Stonebraker, L. A. Rowe, The design of postgres, in: ACM Sigmod Record, 1986.
[49] Oracle Corporation, ForkJoinPool Documentation, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html, 2014. [Online; accessed 16-Mar-2025].
[50] P. Atzeni, L. Bellomarini, M. Iezzi, E. Sallinger, A. Vlad, Augmenting logic-based knowledge graphs: The case of company graphs., in: KR4L@ ECAI, 2020.
[51] DBpedia, DBpedia tables, http://wiki.dbpedia.org/services-resources/downloads/dbpedia-tables, 2023. [Online; accessed 16-Mar-2025].
[52] Unimi, Fastutil, http://fastutil.di.unimi.it/, 2023. [Online; accessed 16-Mar-2025].