Investigation of optimisation techniques for Rational Closure in Defeasible Reasoning

Nevaniah Gounden
gndnev002@myuct.ac.za
University of Cape Town
Cape Town, South Africa

ABSTRACT

Knowledge representation and reasoning are essential areas of Artificial Intelligence (AI) that seek to model decision-making in a logical and structured manner. Although classical reasoning ensures that conclusions follow deterministically from premises, real-world decision-making often involves exceptions, incomplete knowledge, and uncertainty. Defeasible reasoning, as formalised by Kraus, Lehmann, and Magidor's rational closure framework, addresses these challenges by enabling algorithms to handle exceptions consistently. However, the computational cost of rational closure limits its scalability in larger or more complex domains. This paper investigates three optimisation techniques: parallelisation, sub-query memoization, and multi-rank traversal. The aim is to implement these techniques to improve the performance of rational closure in defeasible reasoning. This paper describes the implementation and evaluates these techniques on varied ranked knowledge bases and query sets, comparing their results against a baseline to ensure correctness. Experimental results demonstrate that each optimisation offers distinct performance benefits depending on the size of the knowledge base, the query distribution, and the entailment strategy, with hybrid approaches often outperforming single optimisations. These findings provide practical guidance for scaling rational closure algorithms, contributing to more efficient and robust reasoning systems in AI applications.

CCS CONCEPTS

• **Theory of computation → Automated reasoning**; **Logic and verification**; • **Computing methodologies → Knowledge representation and reasoning**; **Nonmonotonic, default reasoning and belief revision**.

KEYWORDS

Artificial Intelligence, knowledge representation and reasoning, nonmonotonic reasoning, defeasible reasoning, rational closure, lexicographic closure, computational efficiency, algorithmic complexity

1 INTRODUCTION

With the recent surge in Artificial Intelligence (AI), representing human knowledge in computational systems has become a central challenge [20]. This requires methods that handle not only strict logical relations, but also uncertainty, exceptions, and defaults [4]. Classical propositional logic provides a strong base for reasoning, but its monotonic nature limits real-world applicability where new information may invalidate previous conclusions [20]. For example, from "all cars use petrol" and "electric vehicles are cars," classical logic wrongly infers that "electric cars use petrol." Defeasible reasoning (DR), on the other hand, addresses this by revising conclusions when exceptions arise. This enables reasoning such as "cars typically use petrol" while allowing for exceptions like electric cars [17].

This notion of using "typicality" to separate what usually holds from exceptions is formalised and introduced by the Kraus, Lehmann, and Magidor (KLM) framework [17, 19]. Within this framework, Rational Closure (RC) stands out for its ranked models. The logic makes use of a ranking table to distinguish typical from exceptional and impossible cases to support cautious inference [5]. Although RC offers accurate semantic and syntactic foundations, practical implementations face scalability challenges [21]. Existing approaches often rely on repeated entailment and exceptionality checks, which makes them computationally expensive.

RC currently has a strong theoretical background, but little work has focused on optimising execution time for large-scale deployment. This study addresses this gap by investigating three optimisation techniques. The first is *parallelisation*, which performs entailment checks concurrently for a given knowledge base (KB) and query set. The second is *sub-query memoization*, which caches intermediate results to avoid redundant computation. The third is *multi-rank traversal*, which dynamically switches between naive, binary, or ternary traversal strategies depending on the KB structure. The following research questions guide the investigation:

(1) How can the efficiency of the algorithm be improved without compromising integrity?

(2) How would the performance of RC reasoning be affected by implementing parallelised entailment checking, SAT sub-query memoization, and adaptive multi-rank traversal?

(3) Given the different complexities of the KB, does the algorithm prove to be more efficient for all cases?

(4) To what extent do these optimisation techniques improve the scalability of RC across varying query set sizes and rank distributions?

This research also forms part of a broader project aimed at developing complementary tools, including a topic-aware knowledge base generator and an interactive debugger that explains reasoning steps in natural language [9, 10]. The goal is to combine the results of this project to make defeasible reasoning more efficient, accessible, and interpretable, advancing explainable and scalable AI systems.

The remainder of this paper is structured as follows: Section 2 outlines the theoretical foundations, covering propositional logic, defeasible reasoning, the KLM framework and its variations, RC, and scalability issues. Section 3 details the design and implementation of the optimisation techniques, while Section 4 describes the evaluation methodology. Section 5 presents and analyses the results. Section 6 reviews prior and related work on rational closure, while Section 7 summarises the findings and outlines directions for future research. Finally, Appendix A presents visual results that support the discussion.

2 THEORETICAL BACKGROUND

2.1 Propositional Logic

Propositional logic is a fundamental framework for reasoning in intelligent systems due to its well-defined structure and broad applicability [2]. Although not best suited to evaluate real-world scenarios due to its inability to deal with exceptions, its formal syntax and semantics form the basis of several logic frameworks, including defeasible reasoning approaches such as the KLM framework and rational closure [4]. Understanding propositional logic is crucial for developing reasoning systems that maintain formal precision while accommodating practical flexibility, particularly when handling exceptions. This subsection aims to introduce the syntax and semantics of propositional logic.

Syntax. Propositional logic represents statements as *atoms* (e.g., P, Q, R), combined using *Boolean operators* $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\}$, representing negation, disjunction, conjunction, implication, and equivalence. Formulas are built by combining atoms and operators under formal syntax rules [2]. For example, $(P \wedge Q) \rightarrow (\neg R \vee P)$ expresses a structured logical relationship.

Object-Level Semantics. A *valuation* is formally defined as a function $u : \mathcal{P} \rightarrow \{T, F\}$ that assigns a truth value to each propositional atom [16]. For instance, given $\mathcal{P} = \{p, q, r\}$ with $u(p) = T, u(q) = F$, and $u(r) = F$, the corresponding valuation may be denoted as $\mathrm{p}\overline{\mathrm{qr}}$. The semantic interpretation of atoms is determined by their contextual meaning; for example, p may represent the proposition "cars use petrol" [16]. These atomic assignments extend naturally to compound formulas, whose truth values are evaluated according to the standard truth tables associated with the Boolean connectives [2].

p	q	$p \vee q$	$p \wedge q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	T	T	T	T
T	F	T	F	F	F
F	T	T	F	T	F
F	F	F	F	T	T

Table 1: Truth table for binary Boolean operators

For the unary operator $\neg$, we define $\neg p$ as false if p is true and vice versa.

Meta-Level Semantics. A valuation u satisfies an atom p, written $u \Vdash p$, if $u(p) = T$ [2], and this extends to formulas: $u \Vdash A$ if A evaluates to T under u. A knowledge base $\mathcal{K}$ is satisfiable if at least one valuation satisfies all its formulas; otherwise, it is unsatisfiable. A formula A is *entailed* by $\mathcal{K}$, written $\mathcal{K} \models A$, if every model of $\mathcal{K}$ is also a model of A, i.e., $Mod(\mathcal{K}) \subseteq Mod(A)$ [2].

Object vs Meta Levels. The *object level* involves formulas and logical derivations, such as deriving $p \wedge r$ from $p \wedge q$ and $q \rightarrow r$. The *meta level* examines logical properties, including satisfaction and entailment, with symbols $\models$ and $\Vdash$ representing relationships between formulas and valuations [2]. The key difference is in the level of reasoning. *Object-level* consequences are derived using the logical connectives and propositional variables of the object language, whereas *meta-level* consequences are expressed using meta-level symbols and involve reasoning about the language itself.

2.2 Defeasible Reasoning

We now apply classical propositional logic to a previously mentioned real-world scenario. Consider the knowledge base $\mathcal{K} = \{e \rightarrow c,\ c \rightarrow p,\ e \rightarrow \neg p\}$, representing "an electric car is a car", "cars use petrol", and "an electric car does not use petrol". From $e \rightarrow c$ and $c \rightarrow p$, classical propositional rules infer $e \rightarrow p$. However, $\mathcal{K}$ also states $e \rightarrow \neg p$, creating a contradiction that implies e is false. This implies that electric cars do not exist, which is contrary to reality. This illustrates that propositional logic cannot accommodate exceptions, motivating the need for **defeasible reasoning** to address such cases.

2.2.1 KLM Framework.

Proposed by Kraus, Lehmann, and Magidor, the KLM framework forms a foundation for defeasible reasoning [17]. It introduces *typicality* via a meta-level consequence relation $\mid\!\sim$, expressing statements such as "typically, if α then β" as $\alpha \mid\!\sim \beta$. The framework is governed by six postulates designed to balance formal structure with intuitive reasoning. The intention is to allow for cautious conclusions without overly strong assumptions [16].

The KLM framework captures nonmonotonic reasoning through *preferential consequence relations.* These relations are defined using *preferential interpretations*, where possible worlds are ordered by typicality. Inference is then based on the most typical worlds that satisfy the premises, ignoring less typical cases [26]. However, preferential entailment alone struggles with conflicting defaults (e.g., "cars typically use petrol" vs. "electric cars typically do not use petrol") because it lacks a systematic way to prioritise defaults. To address this, Lehmann and Magidor introduced a seventh postulate, *rational consequence relations* [19]. These extend preferential entailment by introducing *ranked interpretations*, which assign explicit levels of typicality, creating a hierarchy where more specific defaults override more general ones, thereby resolving conflicts consistently.

Ranked Interpretations. Ranked interpretations are essential for rational consequence relations. They assign each valuation a nonnegative rank ($\mathcal{R}$) or ∞, where lower ranks denote more typical scenarios and higher ranks indicate less typical ones [6]. This ranking provides a formal way to handle exceptions by prioritising valuations according to typicality. For example, in a ranked interpretation over $\mathcal{P} = \{p, q, r\}$, valuations that fully satisfy defaults may have rank 0, partially compliant ones higher ranks, and impossible valuations ∞. Thus, for a given ranked interpretation, if $\mathcal{R}(p) < \mathcal{R}(q)$, p is considered more typical than q.

Defeasible Implication. The KLM framework uses *defeasible implications*, where $\alpha \mid\!\sim \beta$ means that in the most typical α-worlds, β also holds [16]. These implications form a defeasible knowledge base (KB), guiding inference under normal conditions while allowing exceptions when more specific information appears.

Satisfiability. A defeasible implication $\alpha \mid\sim \beta$ is satisfied if all minimal α-worlds (those with the lowest rank) also satisfy β [16]. A ranked interpretation satisfies a KB if it satisfies every defeasible implication and classical formula in it. Classical formulas are represented as $\neg\alpha \mid\sim \bot$, ensuring they hold in all finitely ranked worlds [5]. For instance, the classical formula r can be expressed as $\neg r \mid\sim \bot$, meaning r must hold in all minimal worlds.

Entailment. As mentioned above, KLM proposed six postulates (later extended to seven by Lehmann and Magidor) that any reasonable meta-level consequence relation $\mid\sim$ should satisfy [17, 19]. Casini et al. reformulated these postulates to characterise *defeasible entailment*, $\approx$, which is said to be *LM-rational* when all seven postulates hold[6]. Preferential entailment, introduced by KLM, satisfies all but RM and can be defined via preferential interpretations. As mentioned earlier, Lehmann and Magidor showed that *ranked entailment,* defined through ranked interpretations, is equivalent to preferential entailment. From this, we can conclude neither is LM-rational due to their failure of RM, yet they remain monotonic [6]. Ranked entailment, despite its limitations, serves as the monotonic foundation of *rational defeasible entailment.* [5].

2.3 Extending the KLM Framework

Ranked Entailment. Ranked entailment needs to be defined to extend defeasible entailment within the KLM framework [6]. A defeasible implication $\alpha \mid\sim \beta$ is said to be rank entailed by a knowledge base $\mathcal{K}$, denoted as $\mathcal{K} \approx_R \alpha \mid\sim \beta$, if every *ranked model* of $\mathcal{K}$ (a ranked interpretation which satisfies $\mathcal{K}$) satisfies $\alpha \mid\sim \beta$. As mentioned earlier, however, this form of defeasible entailment is not LM-rational [17].

Basic Defeasible Entailment. Casini et al. extended the KLM framework by introducing *basic defeasible entailment relations*, which are LM-rational and additionally satisfy two further conditions [5]:

$$\textbf{(Inclusion)} \quad \mathcal{K} \approx \alpha \mid\sim \beta \text{ for every } \alpha \mid\sim \beta \in \mathcal{K}$$

$$\textbf{(Classic Preservation)} \quad \mathcal{K} \approx \alpha \mid\sim \bot \text{ iff } \mathcal{K} \approx_R \alpha \mid\sim \bot$$

Inclusion ensures all defeasible implications already in $\mathcal{K}$ should be defeasibly entailed by $\mathcal{K}$, while *Classic Preservation* states that all classical defeasible implications which are defeasibly entailed by $\mathcal{K}$ should correspond to the classical defeasible implications rank entailed by $\mathcal{K}$ [24]. This demonstrates that rank entailment is indeed the monotonic core of other forms of defeasible entailment.

Rational Defeasible Entailment. Basic defeasible entailment can be too permissive, as it does not always entail the same implications as *rational closure* (mentioned below) [5]. Just as ranked entailment is the *monotonic* core of defeasible entailment, rational closure is regarded as the *nonmonotonic* core [6]. It is defined via the *minimal ranked model* of a knowledge base $\mathcal{K}$, establishing a baseline of what should be entailed [16]. This motivates *rational defeasible entailment relations*, which are basic defeasible entailment relations satisfying the *Rational Closure Extension* property:

$$\textbf{(Rational Closure Extension)} \quad \text{If } \mathcal{K} \approx_{RC} \alpha \mid\sim \beta, \text{ then } \mathcal{K} \approx \alpha \mid\sim \beta$$

This ensures that any rational entailment relation entails at least what rational closure entails [5, 16]. Although this paper focuses exclusively on *rational closure*, other forms of rational defeasible entailment relations, such as lexicographic closure, also exist and are widely studied.

2.4 Rational Closure

Rational closure (RC) can be defined either semantically via a *minimal ranked model* or syntactically via *base ranks* [16]. As we will see, the former is concerned with typicality *across* ranked models of a knowledge base $\mathcal{K}$, while the latter is concerned with typicality *within* ranked models of $\mathcal{K}$ [5]. Together, these approaches balance consistency and plausible inference across models.

Minimal Ranked Model. A *ranked interpretation* $\mathcal{R}$ assigns each world a rank in $\mathbb{N} \cup \{\infty\}$, where lower ranks indicate greater typicality and ∞ marks impossibility. $\mathcal{R}$ is a model of $\mathcal{K}$ if it satisfies all defeasible and classical statements in $\mathcal{K}$. Among such models, the *minimal ranked model*, $\mathcal{R}^{RC}_{\mathcal{K}}$, is pointwise minimal:

$$\forall \mathcal{R}', \quad \mathcal{R}^{RC}_{\mathcal{K}}(u) \leq \mathcal{R}'(u) \text{ for all valuations } u$$

[5, 13]. This model provides the most conservative ranking, increasing ranks only when conflicts force exceptions. For instance, given $c \mid\sim p$ (cars typically use petrol) and $e \mid\sim \neg p$ (electric cars typically do not), worlds where e and p both hold are ranked higher, reflecting their atypicality. RC determines defeasible entailment by checking whether $\alpha \mid\sim \beta$ holds in all minimal α-worlds of $\mathcal{R}^{RC}_{\mathcal{K}}$.

Base Ranks. Base ranks offer a syntactic perspective on RC by iteratively identifying exceptional formulas. A formula α is *exceptional* in $\mathcal{K}$ if $\mathcal{K} \approx_R \top \mid\sim \neg\alpha$ or, equivalently, if $\overrightarrow{\mathcal{K}} \models \neg\alpha$, where $\overrightarrow{\mathcal{K}}$ materialises each default $\alpha \mid\sim \beta$ into $\alpha \rightarrow \beta$ [6, 16]. We define a sequence of knowledge bases to iteratively capture exceptional defaults:

$$\mathcal{E}^{\mathcal{K}}_0 := \mathcal{K}$$

$$\mathcal{E}^{\mathcal{K}}_i := \varepsilon(\mathcal{E}^{\mathcal{K}}_{i-1}) \text{ for } 0 < i < n$$

$$\mathcal{E}^{\mathcal{K}}_\infty := \mathcal{E}^{\mathcal{K}}_n \text{ where } n \text{ is the smallest index for which } \mathcal{E}^{\mathcal{K}}_n = \mathcal{E}^{\mathcal{K}}_{n+1}$$

Here, $\varepsilon(\mathcal{E}^{\mathcal{K}}_{i-1})$ refers to the set of defaults whose antecedents are exceptional with respect to $\mathcal{E}^{\mathcal{K}}_{i-1}$ [6]. This process continues until no further changes occur, guaranteeing termination since $\mathcal{K}$ is finite.

The *base rank* of a formula α, denoted $br_{\mathcal{K}}(\alpha)$, is the least integer r such that α is no longer exceptional in $\mathcal{E}^{\mathcal{K}}_r$:

$$br_{\mathcal{K}}(\alpha) := \min\{r \mid \overrightarrow{\mathcal{E}^{\mathcal{K}}_r} \not\models \neg\alpha\}$$

For defeasible implications, the base rank is assigned to the antecedent:

$$br_{\mathcal{K}}(\alpha \mid\sim \beta) := br_{\mathcal{K}}(\alpha)$$

Rational closure can then be defined using base ranks. A default $\alpha \mid\sim \beta$ is entailed in rational closure if and only if:

$$\mathcal{K} \approx_{RC} \alpha \mid\sim \beta \iff br_{\mathcal{K}}(\alpha) < br_{\mathcal{K}}(\alpha \land \neg\beta) \text{ or } br_{\mathcal{K}}(\alpha) = \infty$$

Additionally, there is a direct relationship between base ranks and the minimal ranked model $\mathcal{R}^{RC}_{\mathcal{K}}$:

$$br_{\mathcal{K}}(\alpha) = \min\{i \mid \exists v \in Mod(\alpha) \text{ with } \mathcal{R}^{RC}_{\mathcal{K}}(v) = i\}$$

This shows that the base rank of α corresponds to the lowest rank of valuations satisfying α in the minimal ranked model [13].

Example: Consider the following knowledge base:

$$\mathcal{K} = \{c \mathrel{|\!\sim} p, e \mathrel{|\!\sim} \neg p, e \to c\}$$

Here, c represents *car*, p represents *uses petrol*, and e represents *electric car*. We start by computing the sequence of exceptional knowledge bases:

$$\mathcal{E}_0^{\mathcal{K}} = \{c \mathrel{|\!\sim} p, e \mathrel{|\!\sim} \neg p, e \to c\}$$

Check if e is exceptional by seeing if $\overrightarrow{\mathcal{E}_0^{\mathcal{K}}} \models \neg e$. Initially, this holds because e contradicts the stronger typicality that cars use petrol, but e does not. Thus:

$\mathcal{E}_1^{\mathcal{K}} = \{e \mathrel{|\!\sim} \neg p\}$ (only keeping defaults with exceptional antecedents)

On checking again, e no longer appears exceptional with respect to $\mathcal{E}_1^{\mathcal{K}}$, so the base rank of e is 1. Therefore:

$$br_{\mathcal{K}}(e \mathrel{|\!\sim} \neg p) = br_{\mathcal{K}}(e) = 1$$

This corresponds to the rank of the minimal e-worlds in $\mathcal{R}_{\mathcal{K}}^{RC}$, demonstrating how the system determines the typical conditions under which electric cars do not use petrol.

Rational Closure Algorithm. The algorithm proceeds in two stages: (1) `BaseRank` partitions rules by antecedent rank:

$$R_i := \{\alpha \to \beta \mid \alpha \mathrel{|\!\sim} \beta \in \mathcal{K},\ br_{\mathcal{K}}(\alpha) = i\}$$

(2) `RationalClosure` checks whether $\alpha \mathrel{|\!\sim} \beta$ is exceptional across partitions, progressively discarding lowest ranks until the query is non-exceptional or only R_∞ remains. It then tests if $\alpha \to \beta$ follows from the remaining rules [6, 12].

For $\mathcal{K}$ above and query $e \mathrel{|\!\sim} p$, the algorithm finds it exceptional in $R_0 \cup R_1 \cup R_\infty$ and removes R_0. In $R_1 \cup R_\infty$, it is no longer exceptional, but $e \to p$ does not follow; thus, RC rejects the query. This reflects RC's cautious nature, assigning properties only when strongly justified and avoiding unwarranted assumptions [16].

2.5 Motivation for Optimisation

As KBs and query sets grow, the computational burden of rational closure entailment checks increases significantly, impacting both execution time and resource consumption. Traditional defeasible reasoning approaches often struggle to maintain efficiency with large datasets, and scalability remains a major challenge for real-world applications [21]. Furthermore, recent work highlights that the complexity of defeasible reasoning in expressive frameworks with typicality introduces additional performance constraints, making optimisation essential for practical deployment [1]. Improving scalability is therefore critical to enable rational closure to operate effectively in large-scale AI systems and automated reasoning environments.

3 SYSTEM DEVELOPMENT AND IMPLEMENTATION

This project adopts a practical optimisation perspective. Specifically, three strategies are introduced to address the inefficiencies of the sequential baseline:

(i) *Parallelisation of Entailment Checks* to make use of concurrency in entailment checking.

(ii) *Sub-query Memoization* to reduce redundant computation, and

(iii) *Multi-Rank Traversal* to adaptively manage rank exploration.

These techniques draw inspiration from both theoretical proposals for reducing entailment complexity and recent trends in scalable reasoning frameworks.

3.1 Parallelisation of Entailment Checks

Previous RC implementations perform entailment checks sequentially across ranks. This scales poorly with larger KBs and query sets. We parallelise at the *query level*, for a fixed ranked KB snapshot. Each query is independent and can be processed concurrently without altering RC semantics. The ranked KB is treated as immutable, and each parallel task uses its own strategy instance to avoid shared state. The only shared mutable resource is the CSV writer, which is synchronized for thread safety. This design ensures deterministic truth values while reducing wall-clock time by leveraging multi-core hardware [21].

Design details. Query-level tasks are executed using a `ForkJoinPool`, which creates one `Callable` per query via `TimedReasonerComparison.executeNewParallel`. Each task instantiates its own entailment strategy (`Naive`, `Binary`, `Ternary`, `NCached`, `BCached`, `TCached`, or `Hybrid`). In addition, each task optionally enables memoization in "MemoParallel" mode, clears its local cache before execution, and logs results within a short synchronized block. Thread safety is ensured as no global static fields are shared across tasks: for instance, `TernaryEntailment` employs a `ThreadLocal<Integer>` for its rank-removal index. Determinism is preserved because each task operates on immutable KB snapshots with isolated strategy instances. This guarantees that repeated parallel runs produce identical entailment results, with only execution time varying.

Initially, the use of `parallelStream()` was explored, but it was abandoned in favor of `Callable` tasks for several reasons: `Callable<Void>` clearly defines each task, `invokeAll()` guarantees that all tasks finish, the approach exhibits more predictable behaviour by not relying on the common pool, it provides better thread control by ensuring the custom `ForkJoinPool` is used for all tasks, and it does not suppress or hide errors as `parallelStream()` might.

Pseudocode. The structure mirrors `TimedReasonerComparison` and uses a synchronized writer for CSV output:

Algorithm 1 Parallel_Evaluate

```
Input: $strategyType$, $rankedKB$, $queries$, $baseline$, $writer$, $label$, $execType$
1 $kbArray \leftarrow$ toArray($rankedKB$)    // immutable snapshot
2 $pool \leftarrow$ ForkJoinPool(min($|queries|$, numCPUs))
foreach $q \in queries$ do
    3 submit task to $pool$:
    $localStrategy \leftarrow$ newInstance($strategyType$)
    if $execType$ = "MemoParallel" then
        4 $localStrategy$.enableMemoization()
    5 $res \leftarrow localStrategy$.checkEntailment($kbArray$, $q$)
    if $baseline[q] \neq res$ then
        6 print "[Mismatch]" to STDERR
    7 synchronized($writer$): write ($q$, $label$, $execType$, $baseline[q]$, $res$)
8 $pool$.shutdown(); $pool$.awaitTermination()
```

Correctness & performance. Each parallel task is isolated with respect to its caches and state, thereby eliminating data races. Synchronization is required only during CSV writes, ensuring that the outcomes remain consistent with the sequential baseline. As discussed in Section 5, the findings demonstrate a substantial performance improvement when overhead costs are negligible.

3.2 Sub-query Memoization

RC repeatedly invokes classical entailment when testing exceptionality across ranks, often revisiting identical queries. To reduce redundant SAT calls, we implement *sub-query memoization.* This entails caching the Boolean result of each entailment check and reusing it when the same proof obligation reappears. Memoization is optional and can be adjusted at runtime. Cache lookups are keyed by a combination of a stable *KB signature* and a canonicalised query string, ensuring correctness under multiple KBs and parallel execution [2, 13, 21].

Design details. All strategies inherit from `MemoizingEntailment`, which stores results in a `ConcurrentHashMap` keyed by rank-aware signatures. Each knowledge base is assigned a deterministic hash via `kbSignature`, which computes a SHA-256 digest over every rank in sequence. Within each rank, formulas are first converted to strings and sorted to remove order sensitivity. Once complete, formulas are then concatenated with a rank prefix (e.g., #0, #1). This ensures that two KBs with identical content but different formula orderings yield the same signature, while KBs with different rank structures do not. The digest is converted into a hexadecimal string for solidity. If SHA-256 is unavailable, the implementation falls back to concatenation and Java's `hashCode`. Cache keys are then formed by combining the KB signature with the query string using `cacheKey`. This guarantees uniqueness per KB/query pair and avoids redundant SAT calls across parallel executions. Cached variants (B/N/T) additionally store filtered KBs produced during rank elimination, keyed by the signature plus the negated antecedent, allowing repeated queries with the same antecedent to bypass recomputation. Thread safety is maintained by eliminating all shared global static fields. Formerly, global variables such as `rankRemove` have been replaced with `ThreadLocal`, ensuring safe state isolation per strategy instance. Cache clearing is not required for correctness, since signatures already distinguish KBs, but can be enabled to limit memory growth and maintain fair runtime comparisons.

Pseudocode. The implementation mirrors `MemoizingEntailment` and its cached subclasses:

Algorithm 2 checkEntailment (as in `MemoizingEntailment`)

```
Input: $rankedKB$, $query$;   cache $M$ : String $\mapsto$ Bool
9 if !memoEnabled then
    10 return CHECKENTAILMENTWITHOUTMEMO($rankedKB$, $query$)
11 $key \leftarrow$ KB_SIGNATURE($rankedKB$) || "|" || CANON($query$)
return $M$.computeIfAbsent($key$, $\lambda \mapsto$ CHECKENTAILMENTWITHOUTMEMO($rankedKB$, $query$))
```

Algorithm 3 Signature and Cache Key Generation

```
Input: $rankedKB$ : array of ranked formula sets,   $query$ : formula
12 function KBSIGNATURE($rankedKB$): String
    13 $md \leftarrow$ SHA-256 message digest
    14 for $i \leftarrow 0$ to $|rankedKB| - 1$ do
        15 $items \leftarrow$ sorted string representations of formulas in $rankedKB[i]$
        16 $md.update("\#" + i)$
        17 foreach $s \in items$ do
            18 $md.update(s)$
    19 $digest \leftarrow md.digest()$
    20 return hex string of $digest$ (or fallback to concatenated hashCode if SHA-256 unavailable)
21 function CACHEKEY($rankedKB$, $query$): String
    22 return KBSIGNATURE($rankedKB$) || "|" || $query.toString()$
```

The cached strategies then construct filtered KBs as follows:

Algorithm 4 getFilteredKB (as in cached B/N/T strategies)

```
Input: $rankedKB$, $neg := \neg$antecedent(query);   cache $F$ : String $\mapsto$ PlBeliefSet
23 $key \leftarrow$ KBSIGNATURE($rankedKB$) || "|" || $neg$.toString()
if $key \in F$ then
    24 return $F[key]$
25 $r^\star \leftarrow$ FINDRANKTOREMOVE($rankedKB$, $neg$)
$KB' \leftarrow$ COMBINERANKS($rankedKB$, $r^\star$+1, end)
$F[key] \leftarrow KB'$
return $KB'$
```

Correctness & performance. By deriving signatures from sorted, rank-aware formula strings, memoization preserves determinism. This means identical inputs always yield identical keys, independent of formula order. This ensures that truth values remain unchanged while redundant SAT calls are avoided. Reuse of filtered KBs reduces the cost of rank elimination, and combined with query-level parallelism, provides substantial speed-ups without compromising correctness [5, 13, 21].

3.3 Multi-Rank Traversal

Repeated RC entailment over rank partitions admits different traversal costs depending on the number of ranks and the distribution of exceptional defaults. A single fixed policy (e.g., always linear or always binary search over ranks) is therefore suboptimal across KBs of different sizes. We implement a *hybrid* selector that chooses among Naive (linear scan), Binary, and Ternary traversal at runtime depending on the number of rank layers (`rankedKB.length`). For small rank depths, linear scan minimises constant factors, for medium depths, binary search reduces comparisons, and for large depths, a ternary split decreases the total number of inquiries further. This design keeps the core RC semantics intact while adapting the search strategy to the KB, aligning with prior observations that the cost of exceptionality tests dominates and should be reduced via smarter rank exploration [5, 13, 21].

Design details. At runtime, the hybrid strategy chooses between Naive, Binary, and Ternary entailment depending on the rank depth. Two tunable cutoffs, `NAIVE_THRESHOLD=11` and `BINARY_THRESHOLD=31`, determine which traversal is used (these defaults were chosen empirically but can be re-tuned for KB/query set run). Dispatch depends only on `|rankedKB|`, which is immutable during evaluation, making selection deterministic and thread-safe. Each entailment logs the selected strategy, aiding performance analysis, and the multi-rank strategy works with memoization (3.2), so repeated queries still benefit from cached sub-results regardless of the traversal policy.

Pseudocode. The implementation mirrors the `HybridEntailment` class in code:

Algorithm 5 CHECKENTAILMENTWITHOUTMEMO (Hybrid policy in `HybridEntailment`)

```
Input: $rankedKB$ : array of ranked formula sets, $query$ : formula
26 $naive \leftarrow$ new NAIVEENTAILMENT()  $binary \leftarrow$ new BINARYENTAILMENT()  $ternary \leftarrow$ new TERNARYENTAILMENT()  $method \leftarrow$ ""  $result \leftarrow$ false
27 if $|rankedKB| \leq T_N$ then
28     $method \leftarrow$ "Naive"  $result \leftarrow$ naive.CHECKENTAILMENT($rankedKB$, $query$)
29 else if $|rankedKB| \leq T_B$ then
30     $method \leftarrow$ "Binary"  $result \leftarrow$ binary.CHECKENTAILMENT($rankedKB$, $query$)
31 else
32     $method \leftarrow$ "Ternary"  $result \leftarrow$ ternary.CHECKENTAILMENT($rankedKB$, $query$)
33 print "[HybridEntailment] Used " $method$ " for query: " $query$
return $result$
```

Correctness & performance. The multi-rank policy preserves RC semantics since it only delegates to existing strategies without modifying the KB. Selection is deterministic (a function of rank depth only), so entailment results remain identical to the sequential baseline. The performance benefits arise from aligning the traversal method with the size of the knowledge base. A linear scan is effective for shallow ranks because it avoids unnecessary overhead. At medium depths, the binary strategy reduces the number of checks. For larger depths, the ternary strategy provides the greatest efficiency gains. Thresholds affect speed but not correctness. The approach is compatible with memoization (3.2) and parallelisation (3.1), providing further wall-clock reductions [5, 13, 21].

3.4 Integration into Benchmarking Framework

All strategies are executed within a unified testing harness under identical conditions. All boolean entailment results for each strategy are compared against the sequential baseline (`comparison_boolean_results`), while execution times across sequential, parallel, and memoized variants are logged for each strategy (`comparison_results`). Additional terminal output reports the ranked KB, queries executed, traversal choices in the multi-rank policy, and any mismatches in boolean results. Results are exported to CSVs for consistent comparative analysis.

4 EXPERIMENTAL DESIGN

This section outlines the methodology used to evaluate the scalability and efficiency of the optimised RC algorithms. The experimental setup was designed to ensure fairness, reproducibility, and comprehensive coverage across KBs and query variations. All experiments were executed on the University of Cape Town High Performance Computing (UCT HPC) cluster to guarantee consistent hardware conditions and minimise system variability.

Selection of test cases. A diverse set of ranked KBs was used, ranging from 1,000 to 10,000 statements, with rank depths between 10 and 50 layers. Two main distribution patterns were considered: *normal* and *exponential.* Query sets were constructed with varying structures and repetition patterns, including: `1k50_firstrank.txt`, `1k50_lastrank.txt`, `halfrepeated_ante_query_100.txt`, `mixed_repeated_100.txt`, `repeated_antecedent.txt`, and `unique_100.txt`. All of the KBs and queries used during testing were based on prior optimisation work due to the complexity of the datasets. An additional test was conducted on a structured KB generated using recent advancements, see figure 58. This KB incorporates real-world semantic constraints (e.g., *"if it is dark, then it is typically night"*) [10]. Although these human-interpretable KBs are too small for conclusive benchmarking, they serve to validate the logical consistency of the implementations.

Query set preparation. The query sets were designed to capture realistic defeasible reasoning scenarios, balancing positive (entailed) and negative (non-entailed) cases. Each KB was paired with 3–6 query sets to ensure robustness. Fixed query sets were reused as much as possible across all runs to preserve comparability between strategies.

Execution of baseline and optimised strategies. The original sequential RC implementation was used as the baseline. Each optimisation was tested independently, including:
(i) parallelisation,
(ii) sub-query memoisation, and
(iii) multi-rank traversal.
Combined optimisations (e.g., memoized parallelism) were also evaluated to measure cumulative benefits. For every query in every

KB, the Boolean results of the optimised strategies were checked against the sequential baseline to ensure correctness.

Performance measurement. Execution times were measured in milliseconds by recording the time each strategy, per execution type, took to run using timestamps. Results were written to a CSV for analysis. Two forms of output were produced:

(i) `comparison_boolean_results`, logging per-query correctness across strategies for each execution type (sequential, parallel, memoSequential, and memoParallel variants).

(ii) `comparison_results`, recording execution times for the same strategies per execution type.

Terminal output additionally reported the ranked KB structure, queries executed, traversal strategy selected (for hybrid multi-rank traversal), and any boolean mismatches.

Repetition and averaging. Each query file was executed five times per KB, and averages were taken to reduce noise from runtime variability. Results were visualised in graphs to facilitate comparison of execution times and scalability trends, which can be found in Appendix A.

Data collection and output. All experimental outputs were automatically logged in CSV format with metadata (KB size, rank depth, query count, and strategy used). These files formed the basis for statistical comparison and visualisation of performance improvements across optimisations.

5 RESULTS AND DISCUSSION

This section presents the findings from tests run on the various KBs and query sets mentioned above. The KBs include small-scale (`500statements`, `10ranks`), medium-scale (`1k50`, `2k50`, `5k50`), large-scale (`10k50`), deep-rank (`50ranks`, `100ranks`), and structured distributions (`normal`,`exponential`). Each KB was paired with up to six query sets: `unique`, `repeated`, `half-repeated`, `mixed`, `firstrank`, and `lastrank`. Execution times are reported in milliseconds and averaged over five runs. Detailed graphs appear in the Appendix (e.g., `1k50` in Figures 25–30, `10k50` in Figures 42–45, `100ranks` in Figures 19–24, and so on).

5.1 Baseline Performance

The Sequential baseline is based on previous work on the project and confirms the expected computational outputs of the RC algorithm. In small KBs (`500statements`, `10ranks`), runtimes are lower than for larger KBs but still show clear gaps between traversal strategies. For `500statements_unique`, Naive–Sequential averages ≈ 60,216.8 ms while Binary is ≈ 4,105.8 ms (Figure 4), similarly for `10ranks_unique`, Naive–Sequential is ≈ 14,618.6 ms vs. Binary ≈ 4,415.0 ms (Figure 12). A rank-focused small-query workload makes the traversal gap especially visible, in `100ranks_firstrank`, Naive–Sequential averages ≈ 1,281.0 ms while Binary/Hybrid remain at ≈ 149.8 ms and ≈ 141.6 ms respectively (Figure 19). As KBs grow, this gap widens sharply. For `1k50` with `unique` queries (Figure 25), Naive–Sequential averages ≈ 63,571.4 ms, whereas the other strategies mainly run in the low thousands of milliseconds, with NCached being the exception. Similar super-linear growth is visible in `2k50` (`unique`: Naive–Sequential ≈ 66,989.4 ms; Figure 31) and `5k50` (`mixed`: Naive–Sequential ≈ 84,338.0 ms vs. `TCached` ≈ 3,858.2 ms; Figure 39), and is most pronounced in `10k50` (`mixed`: Naive–Sequential ≈ 181,725.6 ms vs. `TCached` ≈ 9,581.2 ms; Figure 44). Deep-rank suites (`50ranks`, `100ranks`) exhibit the same pattern on `mixed` workloads: Naive–Sequential averages ≈ 63,322.0 ms and ≈ 120,450.0 ms respectively, while cached/ternary variants remain in the ≈ 10,300–20,500 ms range (Figures 16, 21). Consequently, Naive dominates wall-clock time whenever rank exploration or exceptionality checks become expensive.

5.2 Parallelisation

Query-level parallelism provides consistent gains on medium and large KBs and on workloads with nontrivial per-query cost. In `1k50`, Parallel reduces average runtime by roughly **80%** on `mixed` (Figure 27) and **85%** on `unique` (Figure 25) relative to Sequential, and `2k50` shows roughly **84%** (`mixed`) and **86.6%** (`unique`) reductions (Figures 33, 31). For larger KBs, the effect strengthens: in `5k50`, Parallel is ±**84%** faster on `mixed` and ±**87%** on `unique` (Figures 39, 37); in `10k50` it delivers ±**85%** (`mixed`) and ±**89%** (`unique`) reductions (Figures 44, 42). On `half-repeated`, Parallel cuts time by about **83%** in `5k50/10k50` (Figures 40, 45), and by about **78%** on `repeated` in `5k50/10k50` (Figures 38, 43). For the rank-depth suites `50ranks/100ranks`, Parallel yields large gains for `mixed/half-repeated` (e.g., **83.5%** on `100ranks-mixed`, Figure 21); and in the structured distributions - `exponential` and `normal` (Figures 55, 49), Parallel reduces averages by ∼**83–86%** on `mixed`. By contrast, on `firstrank/lastrank` sets Parallel is near-neutral (typically within ±**2%** of Sequential; e.g., Figures 7, 8; 46, 47), with a few exceptions where it is about **3%** slower (e.g., `50ranks-lastrank`, `expo10-firstrank`, `expo10-lastrank`; Figures 14, 52, 53).

5.3 Sub-query Memoization

Memoisation helps most when there is reuse within the execution. Across KBs, `MemoSequential` improves over `Sequential` by **31–37%** on average in `2k50/5k50/10k50 mixed` and `5k50/10k50 half-repeated` sets (e.g., Figure 33, 39, 44, 40, 45). On fully `repeated` sets, standalone `MemoSequential` is near neutral (typically **0–1%**; e.g., `10k50-repeated`, Figure 43; `2k50-repeated`, Figure 32; `5k50-repeated`, Figure 38), but when combined with parallelism (`MemoParallel`) it inherits the large parallel gains (±**75%** for `2k50` (Figure 32)/`5k50` (Figure 38)/`10k50` (Figure 43) `repeated` query sets). For `unique` queries, `MemoSequential` offers a negative to negligible benefit in comparison to `Sequential` (e.g., `1k50-unique` (Figure 25), `10k50-unique` (Figure 42)), whereas `MemoParallel` retains the parallel speedups (**86–88%** across `1k50–10k50`; see Figures 25, 42). Deep-rank and structured KBs follow the same pattern: `MemoSequential` is most effective on `half-repeated/mixed` in `50ranks/100ranks` (e.g., `50ranks-mixed` ±**38%**, `100ranks-mixed` ±**35%**; Figures 16, 21), while `MemoParallel` mirrors Parallel on `unique/repeated` (e.g., `100ranks-repeated` ±**75%**, `expo10-repeated` ±**83%**; Figures 24, 56).

5.4 Multi-rank Traversal

The Hybrid selector (Naive/Binary/Ternary) does not reduce traversal overheads in our results; instead, it is consistently slower than the fixed Binary/Ternary strategies across KBs and execution types. In the deep-rank suites, the *cached* variants are the ones that deliver the reductions: in 50ranks_mixed, BCached is ±**45 -50%** faster than Binary (Figure 16), and in 50ranks_halfrepeated both BCached and TCached are ±**70%** faster than Binary (Figure 15). The same pattern holds in 100ranks using a mixed query set (Figure 21), and with halfrepeated we notice a ±**75%** and ±**65%** reduction, respectively (Figure 22). In shallow KBs (10ranks, 500statements), Hybrid also lags Binary (e.g., 10ranks_unique: Hybrid ≈ 14,058 ms vs. Binary ≈ 4,415 ms; Figure 12; 500statements_unique: Hybrid ≈ 11,586 ms vs. Binary ≈ 4,106 ms; Figure 4). On skewed distributions (expo10), the cached strategies remain among the fastest (expo10_mixed: BCached ≈ 2,029 ms vs. Binary ≈ 4,129 ms, **50.9%** faster), while Hybrid is again slower (Figure 55). Overall, Binary and the cached variants dominate traversal cost, whereas Hybrid introduces overhead in these datasets.

5.5 Correctness Validation

All optimised strategies (Parallel, MemoSequential, MemoParallel, Hybrid) produced boolean results identical to the Sequential baseline across KBs and query sets (e.g., Figure 59). Initial batch I/O clashes were resolved by isolating runs; no mismatches remained in comparison_boolean_results thereafter.

5.6 Combined Optimisations

The best observed configuration per KB/query typically combines caching with parallelism, with TCached under Parallel most often attaining the minimum wall-clock time; in fewer cases, the best is TCached under MemoSequential/MemoParallel or Binary under Parallel. Relative to *Naive–Sequential*, the best configuration reduces runtime by **94–99%** in 1k50 (Figures 25–30), **88–99%** in 2k50 (Figures 31–36), **95–99%** in 5k50 (Figures 37–40), and **97–99%** in 10k50 (Figures 42–45). For the rank-depth suites, reductions are **88–99%** in 50ranks (Figure 13–18) and **94.1–99.7%** in 100ranks (Figures 19–24). The structured distributions achieve **67.5–97.0%** in normal10 and **71.6–96.9%** in expo10 (Figures 49–51, 55–57). For redundancy-heavy workloads on large KBs, TCached with Parallel is consistently the fastest (e.g., 5k50-repeated and 10k50-repeated where TCachedParallel is best; Figures 38, 43).

5.7 Scalability Analysis

Three robust trends emerge across all KBs. **Parallelisation** becomes reliably advantageous as KBs exceed ~2,000 rules: average reductions are **84–86%** on mixed in 2k50–10k50 (84.1%, 84.8%, 85.8%) and **86–89%** on unique (86.6%, 87.8%, 88.7%); in the rank-depth suites on mixed, reductions are ≈**83–84%** (50ranks: 82.7%; 100ranks: 83.5%), while on firstrank/lastrank Parallel is typically within ±**2%** of Sequential with a few slower outliers of about **3–3.6%** (e.g., 50ranks-lastrank, expo10-firstrank, expo10-lastrank). **Memoisation** scales with redundancy: MemoSequential is **31–37%** faster on mixed across medium/large KBs (e.g., 2k50: 36.9%; 5k50: 36.7%; 10k50: 35.5%) and **31–32%** on half-repeated in 5k50/10k50 (32.2%, 31.1%), but near neutral on unique and fully repeated; MemoParallel retains Parallel's large gains where reuse is limited. **Hybrid traversal** is generally *slower* than Binary across KBs and modes—especially on deep-rank mixed/half-repeated and on skewed sets (e.g., expo10_mixed)—with consistent wins only on boundary/redundant cases where per-query work is small (e.g., 2k50_lastrank +31.2%, 1k50_repeated +21.1%).

5.8 Interpretation of Findings

Overall, the results show that RC can be accelerated substantially at scale *without changing entailment outcomes*: all optimised runs produced the same Boolean answers as the sequential baseline. Parallelism delivers large wall-clock savings once per-query work is non-trivial (roughly **84–86%** on mixed and **86–89%** on unique for 2k50–10k50). Memoisation helps when there is reuse—especially on half-repeated/mixed—yielding **31–37%** over Sequential; where reuse is limited (e.g., unique), MemoParallel still inherits the parallel speedups. For deeper or distributionally skewed KBs, *cached* traversal (BCached/TCached) consistently outperforms uncached fixed strategies, whereas the Hybrid selector is not advantageous on these datasets. The structured suites (normal10, expo10) confirm that KB structure modulates which optimisation dominates, motivating a simple runtime profiler that (i) selects the degree of parallelism, (ii) enables caching when reuse is likely, and (iii) prefers Binary-based traversal with caching at greater rank depths.

6 RELATED WORK

Early implementations of RC followed the sequential construction described by Lehmann and Magidor, later formalised in description logics by Casini et al. [7, 19]. These approaches compute exceptionality and base ranks iteratively, making repeated calls to classical entailment engines [7, 13]. While theoretically sound and complete, such designs remain computationally intensive, particularly as knowledge bases grow in size and complexity [**?**]. Subsequent studies have explored refinements, including materialisation techniques, ranked model semantics, and compact rule forms to reduce overhead [5**?**]. However, most remain sequential and are not optimised for large-scale deployment, leaving execution time as a recurring bottleneck [1].

6.1 Foundations of Nonmonotonic and Defeasible Reasoning

The conceptual roots of rational closure trace back to early work on nonmonotonic logics, notably the seminal contributions by Reiter and Shoham [25, 26]. These studies established the theoretical basis for handling defeasible knowledge—statements that admit exceptions—within logical systems. Later, foundational overviews of knowledge representation and reasoning highlighted the importance of managing exceptions and defaults in AI systems [11, 20].

6.2 Description Logics and Rational Closure

Casini and Straccia extended the KLM framework into description logics, providing a semantic and proof-theoretic basis for RC in ontological reasoning [7, 17]. These developments highlighted both the expressivity of RC and the computational challenges it introduces. Additional refinements have been proposed, such as lexicographic closure, which strengthens the reasoning process by resolving conflicts through prioritisation of defaults [8, 18].

6.3 Alternative Formalisms and Extensions

Beyond rational closure, other nonmonotonic reasoning frameworks have been developed. Reiter's default logic and subsequent analyses by Freund explored preferential reasoning and addressed challenges in drawing defeasible inferences [12, 23, 25]. More recent work by Pullinger investigates ways to extend RC beyond its classical limits, proposing new operators to balance computational tractability with expressive reasoning [24].

6.4 Scalability and Practical Deployments

Scalability remains a persistent challenge for nonmonotonic reasoning. Maher et al. and Alviano et al. both analyse algorithmic and complexity aspects, showing that while RC is well-founded, naive implementations often fail to scale [1, 21]. These insights align with our empirical findings that parallelisation, caching, and traversal refinements can significantly mitigate performance bottlenecks in large and structurally complex KBs.

6.5 Knowledge Base Generation and Interfaces

Complementary lines of work address the creation and interaction with knowledge bases. Early systems such as KBG [15] and later frameworks for knowledge-based system generators [22] explored ways of constructing reasoning environments efficiently. More recent work has investigated user-facing tools for explainable AI. For instance, Grosof et al. [14] developed a graphical user interface for visualising defeasible reasoning in biology, while Cotterrell [9] proposed an interactive debugger to make RC more transparent. Similarly, Dunn [10] examined how natural language can be integrated into knowledge base generation pipelines, complementing the efficiency-focused optimisations studied here.

6.6 Synthesis

The broader research landscape reveals two parallel streams: foundational work on nonmonotonic logics and defeasible entailments [17, 19, 23]. Applied research on scalability and usability ranges from algorithmic improvements to interface design and knowledge-based system generation [14, 15, 21? , 22]. Our optimisations contribute to bridging these strands by focusing on practical efficiency while ensuring theoretical soundness. The inclusion of `normal10` and `expo10` benchmarks shows that performance considerations must account not only for KB size and rank depth, but also for distributional structure, an aspect that future work could further integrate into adaptive reasoning frameworks.

7 CONCLUSIONS

This section summarises the key findings of the study, reflects on the effectiveness of the optimisation techniques, and outlines future research directions. The work contributes both practically and theoretically to the challenge of scaling RC in defeasible reasoning, a long-recognised computational bottleneck in knowledge representation and reasoning (KRR) [4, 20].

7.1 Summary of Contributions

The study successfully implemented and evaluated three optimisation strategies for rational closure: parallelisation, sub-query memoization, and multi-rank traversal. In addition, a benchmarking framework was developed to systematically measure performance improvements across knowledge bases of different sizes, rank depths, and query distributions. All optimised implementations were validated against the baseline sequential reasoner, ensuring that the optimisations preserved logical soundness and produced results consistent with the established semantics of rational closure [12, 23].

7.2 Key Findings

The results demonstrated that each optimisation offered distinct benefits depending on the workload. Parallelisation substantially reduced execution times for large knowledge bases, often halving runtime compared to the sequential baseline. However, for smaller knowledge bases, the coordination costs of thread management outweighed the benefits, leading to limited or negative gains [1, 21]. Sub-query memoization proved highly effective for query sets with redundancy, reducing runtimes by more than sixty percent by avoiding repeated entailment checks. In contrast, when queries were unique, the impact of memoization was negligible, since no reuse of cached results was possible. Multi-rank traversal provided flexibility by dynamically switching between naive, binary, and ternary strategies depending on the rank depth of the knowledge base. This adaptability consistently improved efficiency, especially when thresholds were tuned appropriately to the dataset [6, 7]. When the optimisations were combined, additive benefits were frequently observed, although diminishing returns occurred in cases where one optimisation already removed the major computational bottleneck.

7.3 Scalability Insights

The scalability analysis confirmed that these optimisations significantly improve the ability of rational closure to handle large knowledge bases. Parallelisation became consistently beneficial once knowledge bases exceeded approximately 2,000 rules, demonstrating its effectiveness at scale. Sub-query memoization scaled not with knowledge base size but with the redundancy of the query distribution, highlighting the importance of workload characteristics. Hybrid traversal was particularly effective in mid- to large-sized knowledge bases, where rank depth created substantial traversal costs, but was less critical for shallow structures. Together, these findings indicate that the runtime growth rate of rational closure can be significantly reduced through targeted optimisations, making practical deployment in large-scale reasoning systems far more feasible [3, 5, 19].

7.4 Limitations

Despite the improvements achieved, several limitations remain. The overhead of parallelisation reduced its effectiveness for small datasets, where thread management costs exceeded the gains from concurrent execution. Memoization, while highly effective for redundant queries, introduced memory overhead that could become problematic for extremely large query sets. Multi-rank traversal, although beneficial in most cases, required threshold tuning that was dataset dependent, limiting the generality of the approach across knowledge bases with very different rank structures. These limitations suggest that further refinement is necessary to improve robustness and adaptability across diverse reasoning tasks.

7.5 Future Work

Future research should focus on addressing these limitations while extending the scope of optimisation. Adaptive threshold learning for multi-rank traversal, driven by runtime profiling or machine learning, would allow traversal strategies to be selected dynamically without manual tuning. Alternative parallelisation models, such as work-stealing pools or asynchronous batching, should be explored to reduce coordination overhead and improve efficiency on heterogeneous architectures. To mitigate the memory costs of memoization, selective caching and eviction strategies could be developed to preserve performance gains while controlling resource usage. Finally, future experiments should extend beyond rational closure in the KLM framework to include related defeasible reasoning approaches, such as lexicographic closure [8, 18], in order to test the generality and applicability of the optimisation strategies.

Overall, the findings of this study demonstrate that rational closure can be made significantly more efficient and scalable through targeted optimisation. This not only strengthens its theoretical role within the KLM framework but also enhances its viability for practical deployment in real-world reasoning systems, contributing to the broader goal of explainable and scalable AI [1, 21].

REFERENCES

[1] Alviano, M., Giordano, L., and Theseider Dupré, D. Complexity and scalability of defeasible reasoning in many-valued weighted knowledge bases with typicality. *arXiv preprint arXiv:2303.04534* (2023).

[2] Ben-Ari, M. *Mathematical logic for computer science.* Springer Science & Business Media, 2012.

[3] Borrego-Díaz, J., and Galán Páez, J. Knowledge representation for explainable artificial intelligence. *Complex & Intelligent Systems 8* (2022), 1579–1601.

[4] Brachman, R. J., and Levesque, H. J. *Knowledge Representation and Reasoning.* Morgan Kaufmann, 2004.

[5] Casini, G., Meyer, T., and Varzinczak, I. Defeasible entailment: From rational closure to lexicographic closure and beyond. In *Proceeding of the 17th International Workshop on Non-Monotonic Reasoning (NMR 2018)* (2018), pp. 109–118.

[6] Casini, G., Meyer, T., and Varzinczak, I. Taking defeasible entailment beyond rational closure. In *Logics in Artificial Intelligence: 16th European Conference, JELIA 2019, Rende, Italy, May 7–11, 2019, Proceedings 16* (2019), Springer, pp. 182–197.

[7] Casini, G., and Straccia, U. Rational closure for defeasible description logics. In *Logics in Artificial Intelligence: 12th European Conference, JELIA 2010, Helsinki, Finland, September 13-15, 2010. Proceedings 12* (2010), Springer, pp. 77–90.

[8] Casini, G., and Straccia, U. Lexicographic closure for defeasible description logics. In *Proc. of Australasian Ontology Workshop* (2012), vol. 969, pp. 28–39.

[9] Cotterrell, J. A user interface to aid in the understanding of rational closure. Honours Project, University of Cape Town, 2025.

[10] Dunn, J. Implementing natural language into knowledge base generation. Honours Project, University of Cape Town, 2025.

[11] Fikes, R., and Garvey, T. Knowledge representation and reasoning—a history of darpa leadership. *AI Magazine 41*, 2 (2020), 9–21.

[12] Freund, M. Preferential reasoning in the perspective of poole default logic. *Artificial Intelligence 98*, 1-2 (1998), 209–235.

[13] Giordano, L., Gliozzi, V., Olivetti, N., and Pozzato, G. L. Semantic characterization of rational closure: From propositional logic to description logics. *Artificial Intelligence 226* (2015), 1–33.

[14] Grosof, B., Burstein, M., Dean, M., Andersen, C., Benyo, B., Ferguson, W., Inclezan, D., and Shapiro, R. A silk graphical ui for defeasible reasoning, with a biology causal process example. In *Proceedings of the RuleML-2010 Challenge at the 4th International Web Rule Symposium (RuleML 2010)* (2010), vol. 649 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[15] Hrycej, T. A knowledge-based problem-specific program generator. *ACM SIGPLAN Notices 22*, 2 (1987), 53–61.

[16] Kaliski, A. An overview of klm-style defeasible entailment. *NA* (2020).

[17] Kraus, S., Lehmann, D., and Magidor, M. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial intelligence 44*, 1-2 (1990), 167–207.

[18] Lehmann, D. Another perspective on default reasoning. *Annals of mathematics and artificial intelligence 15* (1995), 61–82.

[19] Lehmann, D., and Magidor, M. What does a conditional knowledge base entail? *Artificial intelligence 55*, 1 (1992), 1–60.

[20] Levesque, H. J. Knowledge representation and reasoning. *Annual review of computer science 1*, 1 (1986), 255–287.

[21] Maher, M., Governatori, G., and Rotolo, A. Rethinking defeasible reasoning: A scalable approach. *arXiv preprint arXiv:2001.00406* (2020).

[22] Moisan, S. Generating knowledge-based system generators. *Insights into Advancements in Intelligent Information Technologies: Discoveries: Discoveries* (2012), 1.

[23] Pollock, J. L. Defeasible reasoning. *Cognitive science 11*, 4 (1987), 481–518.

[24] Pullinger, D. Extending defeasible reasoning beyond rational closure. Honours Project, University of Cape Town, 2023.

[25] Reiter, R. A logic for default reasoning. *Artificial intelligence 13*, 1-2 (1980), 81–132.

[26] Shoham, Y. A semantical approach to nonmonotonic logics. In *Readings in Non-Monotonic Reasoning*, M. L. Ginsberg, Ed. Morgan Kaufmann, 1987, pp. 227–249.

A APPENDIX

This appendix presents execution time results for each knowledge base (KB) and its associated query sets. Each figure reports average runtimes (in milliseconds) across execution modes (Sequential, Parallel, MemoSequential, MemoParallel) for the seven entailment strategies. In addition, it presents a graph representing the Boolean results of one of the KB/query sets.

A.1 500statements Knowledge Base

`outputs/500statements_firstrank.png`

Figure 1: Average execution times for the `500statements` KB on the `firstrank` query set.

outputs/500statements_lastrank.png

Figure 2: Average execution times for the `500statements` KB on the `lastrank` query set.

outputs/500statements_halfrepeated.png

Figure 3: Average execution times for the `500statements` KB on the `halfrepeated` query set.

outputs/500statements_unique.png

Figure 4: Average execution times for the `500statements` KB on the `unique` query set.

outputs/500statements_repeated.png

Figure 5: Average execution times for the `500statements` KB on the `repeated` query set.

outputs/500statements_mixed.png

Figure 6: Average execution times for the `500statements` KB on the `mixed` query set.

A.2 10ranks Knowledge Base

outputs/10ranks_firstrank.png

Figure 7: Average execution times for the `10ranks` KB on the `firstrank` query set.

outputs/10ranks_lastrank.png

Figure 8: Average execution times for the `10ranks` KB on the `lastrank` query set.

outputs/10ranks_halfrepeated.png

Figure 9: Average execution times for the `10ranks` KB on the `halfrepeated` query set.

outputs/10ranks_mixed.png

Figure 10: Average execution times for the `10ranks` KB on the `unique` query set.

outputs/10ranks_repeated.png

Figure 11: Average execution times for the `10ranks` KB on the `repeated` query set.

outputs/10ranks_unique.png

Figure 12: Average execution times for the `10ranks` KB on the `unique` query set.

A.3 50ranks Knowledge Base

outputs/50ranks_firstrank.png

Figure 13: Average execution times for the `50ranks` KB on the `firstrank` query set.

outputs/50ranks_lastrank.png

Figure 14: Average execution times for the `50ranks` KB on the `lastrank` query set.

outputs/50ranks_halfrepeated.png

Figure 15: Average execution times for the `50ranks` KB on the `halfrepeated` query set.

outputs/50ranks_mixed.png

Figure 16: Average execution times for the `50ranks` KB on the `mixed` query set.

outputs/50ranks_repeated.png

Figure 17: Average execution times for the `50ranks` KB on the `repeated` query set.

outputs/50ranks_unique.png

Figure 18: Average execution times for the `50ranks` KB on the `unique` query set.

A.4 100ranks Knowledge Base

outputs/100ranks_firstrank.png

Figure 19: Average execution times for the `100ranks` KB on the `firstrank` query set.

outputs/100ranks_lastrank.png

Figure 20: Average execution times for the `100ranks` KB on the `lastrank` query set.

outputs/100ranks_mixed.png

Figure 21: Average execution times for the `100ranks` KB on the `mixed` query set.

outputs/100ranks_halfrepeated.png

Figure 22: Average execution times for the `100ranks` KB on the `halfrepeated` query set.

outputs/100ranks_unique.png

Figure 23: Average execution times for the `100ranks` KB on the `unique` query set.

outputs/100ranks_repeated.png

Figure 24: Average execution times for the `100ranks` KB on the `repeated` query set.

A.5 1k50 Knowledge Base

outputs/1k50_unique.png

Figure 25: Average execution times for the `1k50` KB on the `unique` query set.

outputs/1k50_repeated.png

Figure 26: Average execution times for the `1k50` KB on the `repeated` query set.

outputs/1k50_mixed.png

Figure 27: Average execution times for the `1k50` KB on the `mixed` query set.

outputs/1k50_halfrepeated.png

Figure 28: Average execution times for the `1k50` KB on the `half-repeated` query set.

outputs/1k50_firstrank.png

Figure 29: Average execution times for the `1k50` KB on the `firstrank` query set.

outputs/1k50_lastrank.png

Figure 30: Average execution times for the `1k50` KB on the `lastrank` query set.

A.6 2k50 Knowledge Base

outputs/2k50_unique.png

Figure 31: Average execution times for the `2k50` KB on the `unique` query set.

outputs/2k50_repeated.png

Figure 32: Average execution times for the `2k50` KB on the `repeated` query set.

outputs/2k50_mixed.png

Figure 33: Average execution times for the `2k50` KB on the `mixed` query set.

outputs/2k50_halfrepeated.png

Figure 34: Average execution times for the `2k50` KB on the `half-repeated` query set.

outputs/2k50_firstrank.png

Figure 35: Average execution times for the `2k50` KB on the `firstrank` query set.

outputs/2k50_lastrank.png

Figure 36: Average execution times for the `2k50` KB on the `lastrank` query set.

A.7 5k50 Knowledge Base

outputs/5k50_unique.png

Figure 37: Average execution times for the `5k50` KB on the `unique` query set.

outputs/5k50_repeated.png

Figure 38: Average execution times for the `5k50` KB on the `repeated` query set.

outputs/5k50_mixed.png

Figure 39: Average execution times for the `5k50` KB on the `mixed` query set.

outputs/5k50_halfrepeated.png

Figure 40: Average execution times for the `5k50` KB on the `half-repeated` query set.

outputs/5k50_tests.png

Figure 41: Average execution times for the `5k50` KB on the `5k50 tests` query set.

A.8 10k50 Knowledge Base

outputs/10k50_unique.png

Figure 42: Average execution times for the `10k50` KB on the `unique` query set.

outputs/10k50_repeated.png

Figure 43: Average execution times for the `10k50` KB on the `repeated` query set.

outputs/10k50_mixed.png

Figure 44: Average execution times for the `10k50` KB on the `mixed` query set.

outputs/10k50_halfrepeated.png

Figure 45: Average execution times for the `10k50` KB on the `half-repeated` query set.

A.9 Structured Distributions: normal10 and expo10

outputs/normal10_firstrank.png

Figure 46: Average execution times for the `normal10` KB on the `firstrank` query set.

outputs/normal10_lastrank.png

Figure 47: Average execution times for the `normal10` KB on the `lastrank` query set.

outputs/normal10_halfrepeated.png

Figure 48: Average execution times for the `normal10` KB on the `halfrepeated` query set.

outputs/normal10_mixed.png

Figure 49: Average execution times for the `normal10` KB on the `mixed` query set.

outputs/normal10_repeated.png

Figure 50: Average execution times for the `normal10` KB on the `repeated` query set.

outputs/normal10_unique.png

Figure 51: Average execution times for the `normal10` KB on the `unique` query set.

outputs/expo10_firstrank.png

Figure 52: Average execution times for the `expo10` KB on the `firstrank` query set.

outputs/expo10_lastrank.png

Figure 53: Average execution times for the `expo10` KB on the `lastrank` query set.

outputs/expo10_halfrepeated.png

Figure 54: Average execution times for the `expo10` KB on the `halfrepeated` query set.

outputs/expo10_mixed.png

Figure 55: Average execution times for the `expo10` KB on the `mixed` query set.

outputs/expo10_repeated.png

Figure 56: Average execution times for the `expo10` KB on the `repeated` query set.

outputs/expo10_unique.png

Figure 57: Average execution times for the `expo10` KB on the `unique` query set.

A.10 2025 Knowledge Base Analysis

outputs/2025_kb.png

Figure 58: Execution Time (ms) for Different Strategies on 2025 Knowledge Base Queries.

A.11 Boolean Result

outputs/boolean_result.png

Figure 59: Boolean result of query check done for the `normal10` KB on the `lastrank` query set.