UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Mattias Kimst

Implementing and Testing a Simple Algorithm for Consistent Query Answering

Bachelor's Thesis (9 ECTS)

Supervisor:
Miika Juhani Hannula, PhD

Tartu 2025

Implementing and Testing a Simple Algorithm for Consistent Query Answering

Abstract:

To address inconsistencies in databases, data cleaning is commonly used; however, it can be complex or may result in the loss of some data. As an alternative, the consistent query answering (CQA) paradigm has been developed to provide consistent query results without altering the database. In this thesis, an algorithm for CQA for primary key violations and conjunctive queries is implemented in the Java programming language and its performance is evaluated on synthetically generated data. Additionally, the database purification technique is tested as an option to enhance the algorithm's performance. The results indicate that purification significantly improves performance. However, regardless of the use of purification, the algorithm's runtime increases rapidly as the database size grows.

Keywords: database, query, inconsistency, primary key, logic

CERCS: P170 Computer science, numerical analysis, systems, control

Lihtsa andmebaasipäringutele kooskõlaliste vastuste leidmise algoritmi teostamine ja testimine

Lühikokkuvõte:

Tavaliselt, et tulla toime ebakõladega andmebaasis, see puhastatakse, mis aga võib olla keeruline või võib osa infot kaotsi minna. Alternatiivina on välja pakutud meetod ebakõladeta päringuvastuste leidmiseks andmebaasi muutmata. Selles töös teostatakse taoline algoritm keeles Java ja hinnatakse algoritmi jõudlust sünteetiliselt genereeritud andmetel. Algoritm on kasutatav primaarvõtme kitsenduse rikkumiste ja konjunktiivsete päringute korral. Lisaks proovitakse andmebaasist päringu suhtes ebaoluliste andmete eemaldamist jõudluse parandamiseks. Tulemused näitavad, et viimane vähendab oluliselt algortimi käitusaega, kuigi sõltumata ebaoluliste andmete eemaldamisest käitusaeg kasvab andmebaasi mahu suurenemisel kiiresti.

Võtmesõnad: andmebaas, päring, ebakõla, primaarvõti, loogika

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1. Introduction

In practical applications, databases often contain data that violates integrity constraints — a set of rules that define valid data values — leading to inconsistencies. Such inconsistencies may arise, for example, from data entry errors, integration from multiple sources, or evolving schema definitions. A key challenge in database management is ensuring that queries return certain results despite the presence of inconsistent data. A widespread solution in industry is to clean the databases, which typically involves removing inconsistent data — resulting in information loss — or correcting it, which often requires significant manual effort and domain knowledge. As an alternative, the *consistent query answering* (CQA) paradigm has been developed, allowing databases to remain in an inconsistent state while still providing certain query results.

Recently, a simple algorithm for CQA in the case of Boolean conjunctive queries and primary key integrity constraint violations was proposed by Figueira et al. [1]. The algorithm leverages mathematical logic to verify the certainty of query answers and has been proven to run in polynomial time with respect to the size of the database. In comparison, the naive approach to this problem may have exponential time complexity with respect to database size.

This thesis aims to implement the algorithm based on its theoretical description and to evaluate its performance and scalability using synthetically generated data and 14 example Boolean conjunctive queries. In addition, the study explores the impact of database purification — a preprocessing technique that removes facts irrelevant to the query and thereby reduces the size of the database to be processed — in order to enhance the algorithm's tractability. To the best of the author's knowledge, the mentioned CQA algorithm and purification technique have neither been implemented and explored in the published literature, nor have they been combined.

Chapter 2 provides an overview of the theoretical concepts related to CQA, including definitions of relational databases, database constraints, queries, consistency, the emergence of inconsistencies, and methods to address them — such as data cleaning, CQA, and the purification technique. Chapter 3 presents the background, description, and complexity estimation of the CQA algorithm proposed by Figueira et al. Chapter 4 outlines the implementation process, explaining technological choices, data representation, and the implementation of the algorithm and purification. Chapter 5 covers synthetic data generation, the testing process, and the collection of performance metrics. Chapter 6 presents and analyzes the experimental results and proposes directions for future research. Additional materials, including the queries used in the experiments, the Java method implementing the algorithm, and detailed experimental results, are provided in the

appendices. The ChatGPT-4o[1] language model was used throughout the writing of this thesis to assist with phrasing, grammar, and language refinement.

[1] https://www.chatgpt.com

2. Background

This chapter provides an overview of the definitions of databases, database constraints, and queries. It also covers the definition of database inconsistencies, how they can arise, the methods available to address or prevent them, and the database purification technique.

2.1 Database

Generally, a database can be seen as an abstract, simplified description of some real-world phenomena [2]. There are several types of databases, with the relational database being one of the most commonly used and also the one referred to in this thesis. Various definitions exist for the relational database.

In this thesis, the following definition provided in a paper by Figueira and others is used [1]. A *relational signature* σ is a finite set of symbols, where each symbol R represents a specific relation and is associated with an *arity* $ar(R)$. A *fact* u is an R-fact of some relation R if $u = R(a_1, ..., a_n)$, where R is the symbol associated with the fact u and $a_1, ..., a_n$ is a tuple associated with u. The length n of the tuple $a_1, ..., a_n$ is equal to the arity $ar(R)$. Therefore, according to the source, the database D can be seen as a finite set of facts, and the number of facts in that set is the size of the database.

To illustrate this definition in practice, consider the database represented in Table 1.

Table 1. Sample database expressing students curriculum enrollments

Enrolled	
Student	**Curriculum**
Alice	Computer Science
Bob	Maths

Details		
Student	**Level**	**Year**
Alice	BSc	1
Bob	BSc	2
Bob	MSc	2

Table 1 models a database D_a defined by a relational signature σ_a containing two relation symbols: *Enrolled* with an arity of 2 and *Details* with an arity of 3. The relation *Enrolled*(D_a) is a set of facts representing students' enrollments in different curricula, where each fact is associated with a tuple containing two elements. The relation $Details(D_a)$ is a set of facts indicating students' year and level of enrollment, with each fact associated with a tuple containing

three elements. Specifically, the database consists of the following relations:

$$\textit{Enrolled}(D_a) = \{(\text{Alice}, \text{Computer Science}), (\text{Bob}, \text{Maths})\}$$
$$\textit{Details}(D_a) = \{(\text{Alice}, \text{BSc}, 1), (\text{Bob}, \text{BSc}, 2), (\text{Bob}, \text{MSc}, 2)\}.$$

Thus, the complete database is the set:

$$\begin{aligned} D_a = \{&\textit{Enrolled}(\text{Alice}, \text{Computer Science}), \textit{Enrolled}(\text{Bob}, \text{Maths}), \\ &\textit{Details}(\text{Alice}, \text{BSc}, 1), \textit{Details}(\text{Bob}, \text{BSc}, 2), \textit{Details}(\text{Bob}, \text{MSc}, 2)\}. \end{aligned}$$

2.2 Database queries

A database *query*, in abstract terms, can be viewed as a function that maps a database to a finite set of tuples (of the same length) contained within it. Such queries can be expressed using logic that defines the conditions the selected subset must satisfy.

In this paper, the focus is on *self-join-free Boolean conjunctive queries*. *A Boolean conjunctive query* q is said to be *self-join-free* if no relation name appears more than once in the query q [3].

The following definition of a Boolean conjunctive query is provided by Jef Wijsen [3]. A Boolean conjunctive query q is formally defined as a set of conditions, known as *atoms*:

$$q = \{R_1(\vec{x}_1), \ldots, R_n(\vec{x}_n)\}$$

In the query q each atom $R_i(\vec{x}_i)$ represents a relationship in the database. An atom over a signature σ is an expression of the form $R(\vec{x})$ where $R \in \sigma$ and $\vec{x}$ is a sequence of variables or constants. The length of this sequence matches the arity of R. This query can also be expressed as the logical statement:

$$\exists \vec{y}\, (R_1(\vec{x}_1) \wedge \cdots \wedge R_n(\vec{x}_n)),$$

where for each $i \in \{1, \ldots, n\}$, $R_i(\vec{x}_i)$ is a relational atom such that the variables and constants in $\vec{x}_i$ are all included in $\vec{y}$.

Finally, in the definition by Wijsen, a database D satisfies the query q, written as $D \models q$, if there exists an assignment μ that maps the variables or constants $\vec{x}_i$ in $\vec{y}$ to constants in D — where μ acts as the identity function on constants in $\vec{x}$ — such that $R_i(\mu(\vec{x}_i)) \in D$ for every atom $R_i(\vec{x}_i)$ in q.

Summarizing Wijsen's definition in simple terms, the query checks whether there is a way to assign values to its variables such that all specified relationships exist in the database.

In the sample database represented in Table 1, there can be defined a Boolean conjunctive query that checks if there exists a student who is enrolled in the 1st year BSc level in Computer Science curriculum. Formally, the query can be written as:

$$q_1 = \exists x \ (\textit{Enrolled}(x, \text{Computer Science}) \wedge \textit{Details}(x, \text{BSc}, 1)),$$

where x represents a variable corresponding to a student in the database. We can find a valuation μ_a that maps the variable x to an element in database D_a:

$$\mu_a(x) = \text{Alice}$$

For the constants *Computer Science, BSc, 1* valuation μ_a acts as an identity function:

$$\mu_a(\text{Computer Science}) = \text{Computer Science}$$
$$\mu_a(\text{BSc}) = \text{BSc}$$
$$\mu_a(1) = 1$$

This query does not return specific student identifiers but instead evaluates to a Boolean value: *TRUE* if such a student exists in the database and *FALSE* otherwise. Since there exists in the example the valuation μ_a, the query q_1 outputs *TRUE* on the database D_a. This query is self-join-free because neither the relation *Enrolled* nor *Details* appear in two or more atoms of the query.

2.3 Database constraints

In order to make a database more accurately and consistently express real-world phenomena, *constraints* are declared on a database specifying which data instances are considered to be correct and which are erroneous [4]. If there exists a subset of data in the database that fails to comply with integrity constraints, it is referred to as a *violation of constraints* [4]. This thesis specifically focuses on one type of integrity constraint: the *primary key constraint*. Henceforth, any mention of a constraint in this thesis refers to a primary key constraint. The set of primary key constraints is denoted by Σ.

According to Figueira and others [1], a primary key constraint over a relational signature σ defines, for each relation symbol R in σ, a specific subset of indices $\{i_a, .., i_b\}$ from the set of all indices $\{1, ..., ar(R)\}$ of a fact of R. These indices form a unique identifier — or key — for each fact within that relation. Each index refers to an element — called an *attribute* — in the tuple associated with a fact. A primary key constraint for relation R is denoted R:$\{i_a, ..., i_b\}$.

For a database to satisfy the key constraint according to the source, it must ensure that any two facts in a relation R that share identical values at the key indices are indeed the same facts — no two distinct facts may have the same values for the designated key attributes.

In the database expressed in Table 1, the attribute *Student* in relation *Enrolled* (element at the first index of tuples of relation *Enrolled*) might be assigned as the primary key, assuming that each student can be enrolled in only one curricula. This would make the *Student* column unique across entries. The primary key constraint is denoted as Enrolled:$\{1\}$. This constraint requires each enrollment to have a unique *Student* value, ensuring no two entries share the same *Student* value. If the same student is associated with two different curricula, it would violate the primary key constraint.

If the *Student* column is similarly assigned as the primary key in the *Details* relation in Table 1, a primary key violation occurs because two distinct rows share the same value, *Bob*, in the *Student* column.

In the example database D_a, if the column *Student* is defined as the primary key in both relations, the set of primary key constraints Σ_a is:

$$\Sigma_a = \{Enrolled{:}\{1\}, Details{:}\{1\}\}$$

2.4 Database consistency

The state of a database in which all its data adhere to the constraints defined on it is referred to as *database consistency* [2]. When a database is described as *inconsistent*, it means that it fails to meet the specified integrity constraints, thereby violating them [2].

2.4.1 Emergence of inconsistencies

Inconsistencies in databases can arise in numerous ways. Wijsen [5] notes that these often occur when multiple databases are integrated into one. While each database may be consistent on its own, conflicts may arise when they are combined. Another common cause of inconsistency is the declaration of a less restrictive constraint, which may allow incorrect data to be entered. Later, as Wijsen points out, if a stricter constraint is introduced after the issue is identified, the database might already be inconsistent.

To illustrate how inconsistent databases can emerge, consider the two separate single-relation databases represented in Table 2:

Table 2. Sample student enrollments databases before merging

Enrolled	
Student	**Curriculum**
Alice	Computer Science
Bob	Maths

Enrolled	
Student	**Curriculum**
Alice	Statistics
Charlie	Maths

Assuming that each student can be enrolled in only one curriculum, the column *Student* could be declared as the primary key $\Sigma_a = \{Enrolled:\{1\}\}$. In this case, if considered separately, the databases in Table 2 are consistent because there are no constraint violations; in both databases, each student is enrolled in only one curriculum. However, combining the two databases into one results in the database represented in Table 3.

Table 3. Sample student enrollments database after merging

Enrolled	
Student	**Curriculum**
Alice	Statistics
Charlie	Maths
Alice	Computer Science
Bob	Maths

In Table 3, the student named Alice is enrolled in two curricula, which violates the defined primary key constraint. As a result, an inconsistent database has emerged.

2.4.2 Addressing inconsistencies

Methods for addressing inconsistencies in databases can generally be divided into two categories according to Figueira and others [1]. The first category is said to involve modifying the database to remove information that violates the constraints, a process known as *data cleaning*. Once the database is cleaned, it can be used to query consistent data. The second approach retains the database in its original inconsistent state and employs various techniques to extract consistent information during queries. This method according to the source is referred to as *consistent query answering*.

2.4.3 Data cleaning

Addressing database integrity issues using the *data cleaning* method involves inserting, deleting, and modifying existing database records [6]. The removal of records is considered relatively simple in the source, but it may result in the loss of important and useful information. While inserting and modifying records does not result in data loss, selecting appropriate values for new entries can be challenging [6].

Assadi et al. [7] present two approaches for making appropriate corrections to eliminate database inconsistencies. The first is a faster and simpler method, which involves automatically applying rules that all database records must satisfy; any records that conflict with these rules are adjusted to comply. However, the authors note that this approach lacks precision in identifying actual errors. The second, more rigorous approach involves domain experts manually determining suitable modifications to meaningfully correct errors in the database. While this method can yield more accurate results, it is highly labor-intensive due to the potentially large volume of information and the number of corrections that may be required [7].

In the sample inconsistent database represented in Table 3, the simplest and most automated approach to cleaning would involve removing both records related to the student Alice. However, this results in the complete loss of information regarding Alice's enrollment.

If external knowledge were available about the correct curriculum for Alice, an expert could manually remove only one of the conflicting records. Alternatively, if one of the records involving Alice was incorrect due to a misidentified student, the student's name could be corrected — preserving more information while still resolving the inconsistency. However, both of these approaches rely on external knowledge, which may not be available or could be difficult and time-consuming to find.

2.4.4 Consistent query answering

As an alternative to database cleaning, the concept of *consistent query answering* (CQA) was first introduced in the article [8]. In this foundational work, a *consistent query* is defined as one that yields the same result across all potential *repairs* of the database.

Formally, CQA is defined as the following computational problem [9]:

$$\text{CERTAINTY}(q, \Sigma) = \{D \mid D \text{ is an inconsistent database such that every repair } D' \text{ of } D \text{ with respect to the integrity constraints } \Sigma \text{ satisfies the query } q\}.$$

The problem CERTAINTY(q, Σ) is generally classified as coNP-complete [10]. This complexity classification also holds for self-join-free conjunctive queries; however, some of these queries have been shown to be tractable [11].

The paper by Arenas and others [8] provides a formal definition of a database *repair*: a database D' is considered a repair of an inconsistent database D if D' satisfies all integrity constraints in Σ associated with D, and the symmetric difference between D and D' is minimal among all database instances that adhere to those constraints.

The concept of a database repair is illustrated in Table 4, which presents potential repairs of the database shown in Table 3.

Table 4. Merged enrollments database possible repairs

Enrolled	
Student	**Curriculum**
Alice	Computer Science
Bob	Maths
Charlie	Maths

Enrolled	
Student	**Curriculum**
Alice	Statistics
Bob	Maths
Charlie	Maths

In order to repair Table 3, at least one of the facts involving the student Alice must be removed to satisfy the integrity constraint that each student can only be enrolled in only one curriculum. To ensure the repair is minimal, no more than the necessary changes should be applied. While removing both facts related to Alice would result in a valid repair, it would not be a minimal repair, as it introduces more modifications than necessary to obtain a database free of primary key violations.

To illustrate how repairs can be used to find consistent query answers, consider the following sample query q_2, evaluated on the database repairs shown in Table 4:

$$q_2 = \exists x \ (\textit{Enrolled}(x, \text{Computer Science}))$$

The query q_2 evaluates as *TRUE* on the left database repair in Table 4 but as *FALSE* on the right database repair. Therefore, according to the concept of CQA, the query q_2 is not consistent.

To illustrate more specifically the definition of CERTAINTY(q, Σ), consider the following queries q_3 and q_4:

$$q_3 = \exists x, y \; Details(x, y, 2)$$
$$q_4 = \exists x \; Details(x, \text{BSc}, 2)$$

Since every possible repair of the database D_a, given the set of primary key constraints Σ_a, contains a second-year student but not a second-year BSc student, the following holds:

$$D_a \in \text{CERTAINTY}(q_3, \Sigma_a)$$
$$D_a \notin \text{CERTAINTY}(q_4, \Sigma_a)$$

2.4.5 Database purification technique

Evaluating the CERTAINTY(q, Σ) problem on the full database can be computationally intensive, as the entire database may be significantly larger than the actual set of facts influencing the outcome of the query q. Thus, finding ways to reduce the size of the input D can enhance the computational practicality of the problem.

Wijsen [9] suggests database *purification* as a method to enhance the tractability of the problem CERTAINTY(q, Σ). He defines database purification as follows: A database P is said to be *purified* in respect to Boolean conjunctive query q if, for every fact $a \in P$, there exists a valuation θ over all variables and constants $\vec{y}$ in q such that $a \in \theta(q) \subseteq P$. This means that every fact in the database is relevant to the query. Furthermore, he proves that it is possible to compute in polynomial time, a database P_0 that is purified in respect to q, such that:

$$P \in \text{CERTAINTY}(q, \Sigma) \iff P_0 \in \text{CERTAINTY}(q, \Sigma).$$

The process of purification can be performed in polynomial time. Initially, identifying relevant facts requires computing all possible answers to the query. This can be achieved by first enumerating all potential answers. For self-join-free Boolean conjunctive queries, the number of potential answers is given by $r_1 \cdot ... \cdot r_i \cdot ... \cdot r_k$, where r_i denotes the number of facts in the relation corresponding to the i-th atom in the query, and k is the total number of atoms. Since each $r_i \leq n$, where n represents the size of the database, the number of potential answers is bounded by n^k.

For each potential answer, the query condition can be evaluated in constant time. If the condition is satisfied, the associated facts are collected into the set of relevant facts. Subsequently, block

assignment is performed by iterating over the n facts and grouping them into blocks according to their primary key. Next, for each block, it is necessary to iterate over all its facts and verify whether each fact belongs to the set of relevant facts. Since there can be up to n relevant facts, this step has complexity $O(n \cdot n) = O(n^2)$. If all facts within a block are deemed relevant, they are included in the purified database.

Overall, the complexity is $n^k \cdot Const + n + n^2$. The purification procedure thus has a time complexity of either $O(n^k)$ or $O(n^2)$, depending on whether $k \leq 2$, which is polynomial in regards of the database size n.

3. The simple algorithm for consistent query answering

The main objective of this thesis is to implement, test, and evaluate the performance of a simple iterative algorithm proposed by Figueira and others [1] for finding certain answers to Boolean conjunctive queries over databases with primary key constraint violations. This chapter provides a brief overview of the algorithm.

3.1 Previous works on CQA algorithms

The most intuitive and naive approach to solving the CERTAINTY(q, Σ) problem is to simply find all possible database repairs D' and evaluate the query on each of them. However, it is easy to see that there can be exponentially many repairs for a given database and therefore, evaluating the query on all repairs would result in exponential time complexity [1]. Many researchers have investigated better worst-case-complexity solutions to this problem. One of the earliest methods, based on iterative query modification, was proposed by Arenas et al. in the same work where the CQA paradigm was initially introduced [8]. Over time, several other approaches have been developed, including techniques based on consistent first-order rewriting [12] and SAT-solvers [13].

3.2 Algorithm description

The algorithm focused on in this thesis has a short and simple description. It is also easy to see that the algorithm has polynomial time complexity, which will be demonstrated later in this chapter. However, the proof of correctness is non-trivial and extensive; therefore the details are beyond the scope of this thesis. The algorithm is originally described as follows [1:1]:

> *We propose a simple inflationary fixpoint algorithm for consistent query answering which, for a given database, naively computes a set Δ of subsets of facts of the database of size at most k, where k is the size of the query q. The algorithm runs in polynomial time and can be formally defined as:*
>
> 1. *Initialize Δ with all sets S of at most k facts such that $S \models q$.*
> 2. *Iteratively add any set S of at most k facts to Δ if there exists a block B (i.e., a maximal set of facts sharing the same key) such that for every fact $a \in B$, there is a set $S' \subseteq S \cup \{a\}$ where $S' \in \Delta$.*
>
> *For an input database D, the algorithm outputs "q is certain" if Δ eventually contains the empty set.*

The algorithm is proven in the same paper to correctly determine the certainty of self-join-free queries that are known to be solvable in polynomial time. For other self-join-free queries, a query is guaranteed to be certain if the algorithm outputs so. However, it is mentioned that in some cases, the algorithm might incorrectly output that a query is not certain when it actually is.

The algorithm could be implemented in pseudocode as follows:

```
def Cert_k(q, D):
    """
    Determines if the query q is certain on the database D.

    Arguments:
        q: The query.
        D: The database (a set of facts).

    k_sets(D): returns sets of facts in D up to size k,
            where k is the number of query atoms
    blocks(D): returns sets of facts in D that share the primary key.

    Returns:
        True if the query q is certain on database D; False otherwise.
    """
    delta = {S for S in k_sets(D) if S |= q}

    while any(
        S not in delta and
        any(
            all(
                any(
                    S_prime.is_subset(S.union({a})) for S_prime in delta
                ) for a in B
            ) for B in blocks(D)
        ) for S in delta):
            delta.add(S)

    if {} in delta:
        return True
    else:
        return False
```

This algorithm follows an inflationary fixpoint process [1]. It means that new sets S of up to k facts — also referred to as *k-sets* — are added to the set Δ and never removed until no further changes occur.

3.3 Time complexity

In the worst-case scenario, the algorithm must iterate over every candidate set S of maximum size of k facts. If a database contains n facts, then there are up to:

$$\sum_{i=0}^{k} \binom{n}{i}$$

possible sets S. To determine if to add a set S to the set Δ, the algorithm iterates over all blocks B. A block B is a maximal set of facts sharing a key. There is up to n blocks B (in the case where each block consists of a single fact). If a new set S is added to the set Δ, a previously checked candidate set S might start to satisfy condition for adding that set to Δ. Therefore, all candidate sets S must be checked again. This re-evaluation might occur up to as many times as there are possible candidate sets S, assuming that in each iteration exactly one new set is added to Δ. As a result, the total number of steps can be expressed as:

$$\sum_{i=0}^{k} \binom{n}{i} \cdot n \cdot \sum_{i=0}^{k} \binom{n}{i} = n \cdot \left(\sum_{i=0}^{k} \binom{n}{i} \right)^2 .$$

The summation can be expanded as:

$$\sum_{i=0}^{k} \binom{n}{i} = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{k}.$$

Since the binomial coefficient $\binom{n}{k}$ has time complexity $O(n^k)$ and dominates the binomial coefficients where $i < k$, the entire sum is in $O(n^k)$. Squaring this yields $O(n^{2k})$, and the additional multiplication by n results in a total time complexity of:

$$O(n^{2k+1}).$$

This complexity is polynomial with respect to the size of the database n, as k is constant.

4. Implementation

This chapter provides an overview of the technological choices made for the implementation, the representation of queries and data, the implementation of database purification and the CQA algorithm, and the integration of these components to compute consistent query answers.

4.1 Technological choices

The algorithm was implemented in the Java programming language due to the author's familiarity with the language and its object-oriented programming (OOP) capabilities. Java's OOP features enabled an intuitive representation of database relations as Java classes, with individual facts modeled as instances of these classes.

In an industrial setting, data would typically be stored in a relational database and managed using a database management system (DBMS). To facilitate object-oriented access, an object-relational mapping (ORM) tool would generally be employed to translate relational data into Java objects. However, it was decided that, within the scope of this thesis, introducing an additional transformation layer would not have provided significant benefits. Therefore, the database was generated directly as Java objects, allowing it to be processed by the algorithm written in Java.

No additional libraries other than Java core libraries[2] were used. Although an application-building framework like Spring Boot was considered to be used, it was deemed unnecessary, as it would not have provided enough value rather than resulting in application with more unused functionality.

The experiments were conducted on the following hardware: HP EliteBook 840 G8 notebook, equipped with 16GB of RAM and 11th Gen Intel® Core™ i5-1135G7 @ 2.40GHz processor, running the Ubuntu 24.04.1 LTS operating system.

The codebase[3] was stored and made publicly accessible via GitHub version control system.

4.2 Representation of database, schema and facts

In the code, the database was represented as a list of relations, with each relation implemented as a list of fact objects. Each fact object was an instance of a Java class implementing the

[2] https://docs.oracle.com/en/java/javase/22/core/java-core-libraries1.html
[3] https://github.com/MattiasKimst/CqaAlgorithm

`Fact` interface. These fact classes contained fields corresponding to the attributes of a relation. Primary key fields were annotated with a custom `@PrimaryKey` tag to mark primary keys for detection. Additionally, the fact classes included getter and setter methods for attribute access, a method for retrieving the primary key, and a `toString()` method for logging facts to the console.

The schema was represented as a list of `Class` objects, where each entry corresponded to a class implementing the `Fact` interface, conveying the structure of the database relations.

4.3 Implementing queries

The queries used in this study were taken from [13], with the full list provided in Appendix I. These are non-Boolean conjunctive queries that return a list of attribute values. To determine the certainty of an answer, the returned value was substituted into the query, ensuring that all variables in the query were either constants or quantified, thereby transforming it into a Boolean query. Consider the following query q_s:

$$q_s(z) := \exists x, y, v, w \, (R_1(x, y, z) \wedge R_2(y, v, w))$$

This is a non-Boolean query, as it returns values of z. To transform it into a Boolean query, a specific value $z = C$, returned by running query q_s, is substituted as a constant:

$$q_b := \exists x, y, v, w \, (R_1(x, y, C) \wedge R_2(y, v, w))$$

The modified query q_b evaluates to a Boolean value instead of returning specific values of z.

The CQA algorithm can then be applied, taking a Boolean conjunctive query as input and returning either `True` or `False`, depending on whether the query result is certain. To represent Boolean conjunctive queries in code, the queries were transformed into Java logical statements using `&&` operator and `String.equals()` checks for comparing attribute values. The queries were implemented in classes where there were separate Boolean conditions for `SELECT` and Boolean queries, methods to execute either of query types, method to find facts satisfying the query and a method for transforming input facts in a way that they satisfy the defined query for data generation purposes.

4.4 Implementing purification

To improve the performance of the algorithm, database purification described in Chapter 2.4.5 was employed to remove irrelevant facts from the database.

Finding sets of relevant facts in this implementation was achieved by iterating over all possible relevant fact combinations, i.e., combinations matching the relational structure of the query. If a combination of facts satisfied the query, those facts were added to the set of relevant facts.

It was expected that purification, without impacting the outcome of the algorithm, would reduce the number of steps in the iterative part of the algorithm. In comparison, running the algorithm on all facts was expected to significantly increase the number of candidate k-sets S and blocks B to be processed. By filtering out irrelevant facts beforehand, computational efficiency was expected to be significantly improved.

The purifier was implemented as follows: first, all combinations of facts that satisfied the query were identified and stored in separate lists for each relation. Then, purified relations were created by dividing the facts in the original relation into blocks. For each block, it was checked whether all of its facts were present in the corresponding list of facts that were part of any combination satisfying the query. If all facts in a block were found in this list, the block was added to the purified relation. Finally, the purified database was returned as a list of purified relations. The flow of the implemented purification is illustrated in Figure 1.

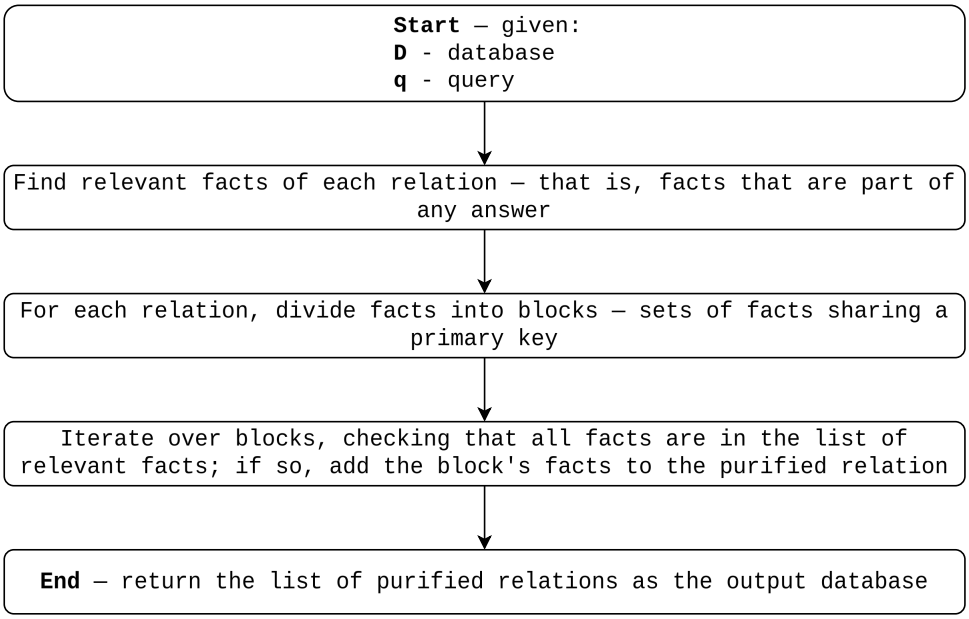

Figure 1. Flow of the implemented purification

The purifier was applied to the database twice — first for the original non-Boolean query and then for each answer substitution in relation to the Boolean query. This two-step purification process helped to narrow down the database by initially filtering it based on the original non-Boolean

query, and then applying a more specific purification step based on the transformed Boolean query to a relatively smaller subset of facts.

To compare the performance of the algorithm with and without purification, the purification step was optional and the same synthetically generated database instances were evaluated both with the purification step enabled and disabled. The purification was performed in the wrapper algorithm.

4.5 Implementing the wrapper algorithm

To integrate various implementation components responsible for finding certain answers for a given database and query — such as computing answers to initial non-Boolean queries, converting them into Boolean queries by substituting the answer into the original query as a constant, identifying certain answers, and performing database purification when the corresponding parameter is set to true — a wrapper algorithm was implemented. The algorithm's flow is illustrated in Figure 2.

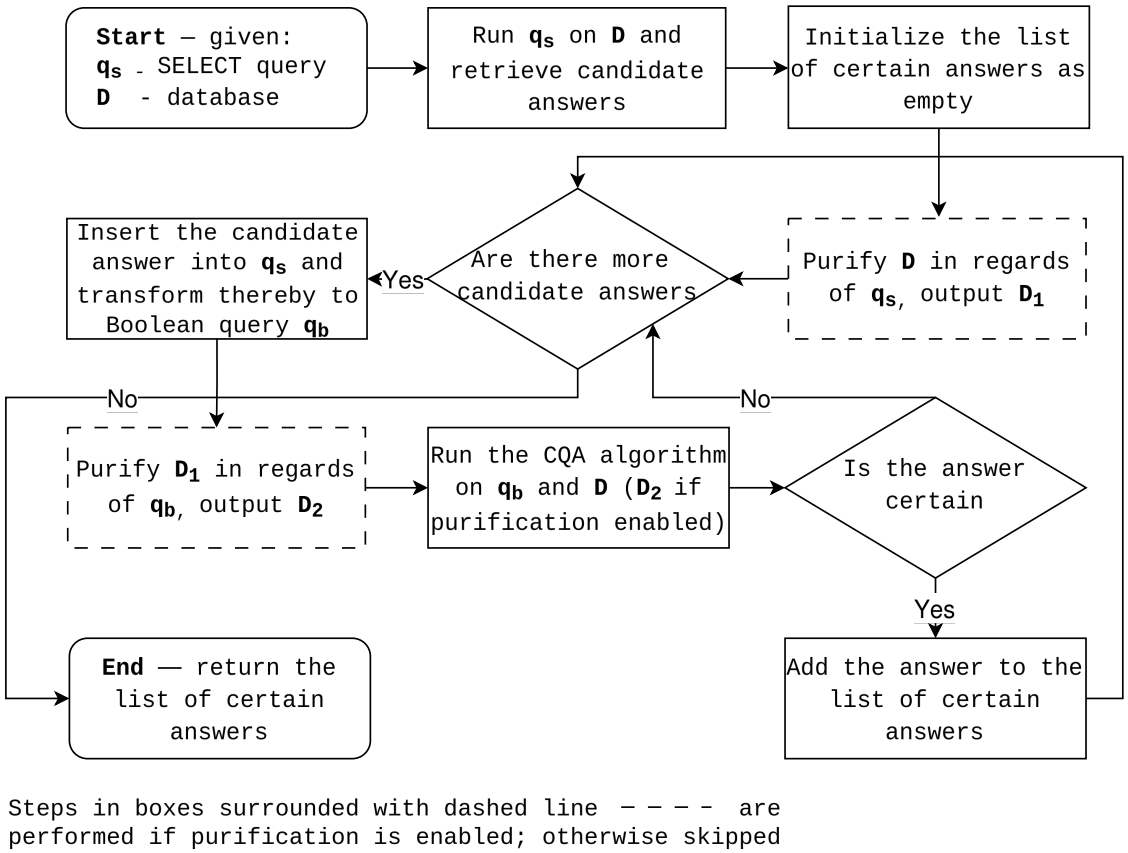

Figure 2. Flow of the implemented wrapper algorithm

First, the original query was evaluated by applying the query condition to all possible combinations of facts from relations that matched the required structure. This process produced an initial set of candidate answers.

Next, the certainty of each candidate answer was verified. This was achieved by substituting the answer as a constant into the original query, transforming it into a Boolean query, as outlined in Chapter 4.3. The resulting Boolean query was then processed using the CQA algorithm. If the algorithm returned `True`, indicating that the query is certain on the given database, the corresponding answer was included in the set of certain answers.

Finally, the complete set of certain answers was returned as the final output, representing the certain answers to the initial query.

4.6 Implementing the CQA algorithm

The algorithm described by Figueira et al. [1] was implemented in Java, following as closely as possible the original description provided in Chapter 3.2. A code snippet of the main Java method that applies the CQA algorithm, as implemented in the scope of this thesis, is included in Appendix II. The general flow of the implementation is shown in Figure 3.

The algorithm begins by initializing the set Δ using a method defined in the `Query` class to identify sets of facts from the database that satisfy the query. Next, the facts in the database are grouped into sets, called blocks, based on their primary keys.

In the main iterative phase, candidate k-sets are generated using an implementation of the `Iterator` interface. In an earlier version of the application, all possible k-sets S were generated at once and stored in a variable. However, this approach frequently led to `OutOfMemoryError` exceptions in Java due to the large number of possible k-sets, which exhausted the Java Virtual Machine (JVM) heap space and caused the application to crash.

For each candidate k-set S, the algorithm iterates over every block B. For each combination of S and B, every fact a in B is checked to determine whether there exists a set $S' \in \Delta$ such that $S' \subseteq S \cup \{a\}$. This is done by constructing the set $S \cup \{a\}$ and iterating over all sets S' in Δ to verify the condition.

If every fact a in a block B satisfies the condition $S' \subseteq S \cup a$, the set S is added to Δ. At this point, processing of the current k-set S is halted, as the condition for adding S to Δ has already been satisfied, and the iteration restarts with a new candidate k-set. Reiteration begins from the

start of the set of all k-sets, since adding a new set to Δ may cause already checked k-sets to satisfy the condition $S' \subseteq S \cup a$

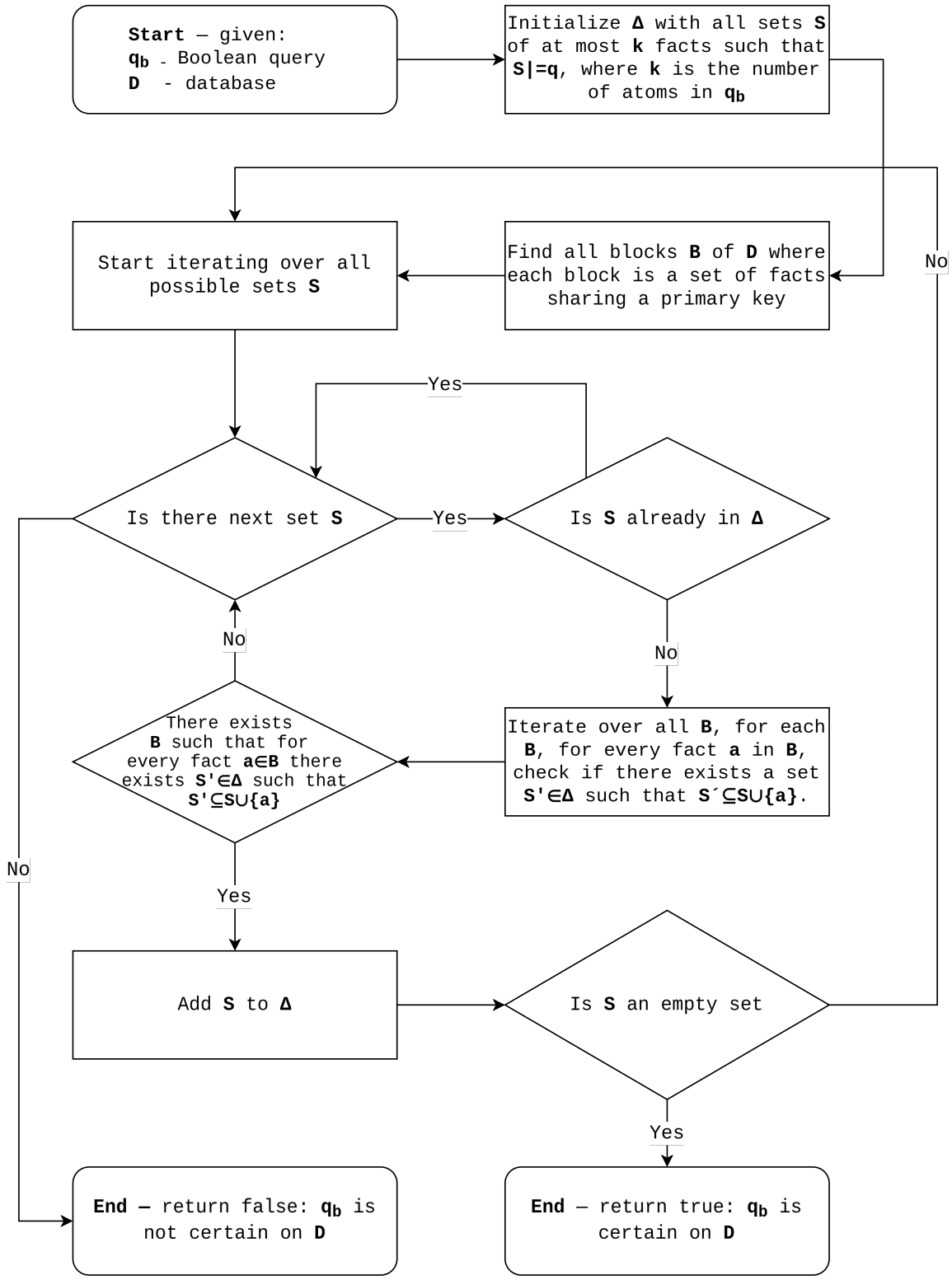

Figure 3. Flow of the implemented CQA algorithm

If, during the iteration over facts in block B, a fact is found that does not satisfy the query, further iteration over that block is terminated to optimize execution, as the block cannot satisfy the condition.

If an empty set S is added to Δ, the algorithm terminates and returns `True`, indicating that the query is certain for the given database D. If the algorithm completes without adding any empty sets to Δ, it returns `False`, signifying that the query was not confirmed to be certain (although it may still be certain, as noted in Chapter 3.2).

5. Testing

This chapter covers the testing process of the algorithm, including the generation of synthetic data and the rationale for its use, the procedures and parameters applied during algorithm execution, and the performance metrics used for evaluation.

5.1 Generating synthetic data

To conduct the experiments, synthetically generated data was used. Synthetic data allows greater control over parameters such as database size, degree of inconsistency, and number of query answers than a real-world database. The methodology for data generation was based on the approach described in a paper by Dixit and Kolaitis [13], where a similar experimental setup was used. The data generation process consisted of three main phases: (1) generating consistent facts, (2) inserting answers that satisfy the query, and (3) inserting inconsistencies. The steps of data generation are illustrated in Figure 4.

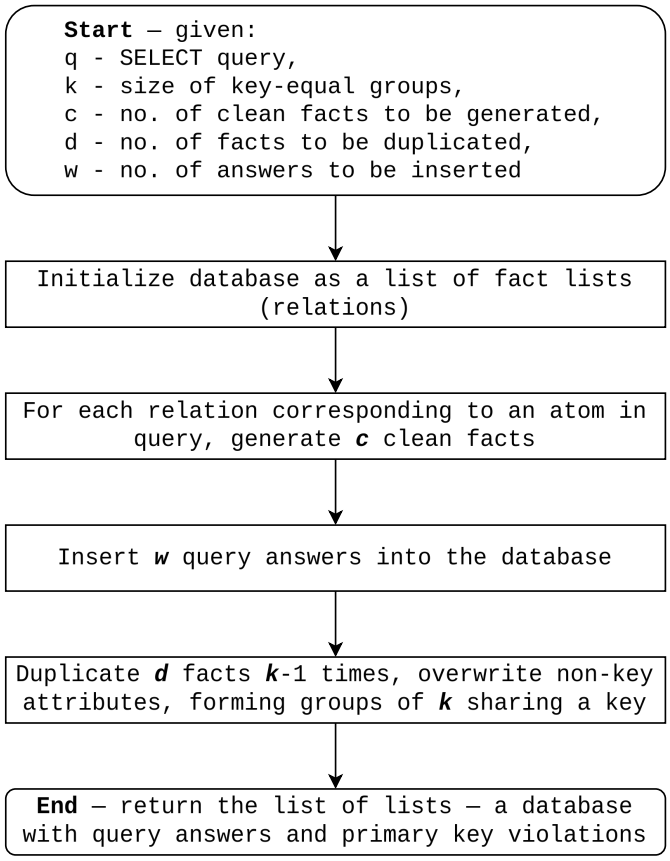

Figure 4. Steps of generating synthetic data

Several parameters were used to control the data generation process, including the number of clean facts generated per relation, the size of key-equal groups, the number of duplicated facts within each relation, and the number of answers to be inserted into the database.

5.1.1 Generating consistent data

To ensure the inclusion of consistent data in the database, first the facts that did not violate any primary key constraints were generated for each relation in the predefined schema. The number of facts per relation was controlled by a predefined parameter. Each fact was created by assigning random alphanumeric strings of length 10 to its attribute values. The same number of clean facts were generated for each relation.

5.1.2 Inserting answers

To ensure that specific queries would evaluate to `True` for a predefined number of fact sets, corresponding answers were explicitly inserted into the synthetic database. The number of inserted answers was controlled by a parameter, which was adjusted to ensure that 15–20% of the facts in the final database contributed to the query results. The insertion process involved selecting random combinations of facts from different relations and modifying their attribute values to satisfy the query conditions.

5.1.3 Inserting inconsistencies

To introduce inconsistencies into the database, a subset of clean facts was selected based on the *number of facts to be duplicated* parameter. For each selected fact, $k - 1$ duplicate facts were generated, preserving the same primary key values while assigning newly generated values to non-primary key attributes. This process resulted in the formation of key-equal groups of the specified size, thereby introducing controlled inconsistencies into the dataset.

5.2 Running the experiments

The experiments were executed under different parameter settings. Following the experimental methodology of Dixit and Kolaitis [13], three primary parameters were considered: relation size, key-equal group size, and the degree of inconsistency. The experiments were conducted using combinations of relation sizes of 10, 100, 1,000, and 10,000 facts; key-equal group sizes of 2, 3, 4, and 5; and degrees of inconsistency of 5%, 10%, and 15%. For each relation size and each combination of the other two parameters, the experiment was repeated 25 times. A timeout of one hour was imposed, and if the evaluation for a given query and parameter setting exceeded this limit, it was skipped due to the infeasibility of executing the algorithm at that scale. The

total database size was calculated as the product of the relation size and the number of atoms in the query, since each atom in the query corresponded to a relation in the database schema.

The actual parameters used in the implementation were not explicitly the relation size and the degree of inconsistency, but the number of clean facts to be generated and the number of facts to be duplicated. In addition, a parameter was defined for the size of key-equal groups. The combinations of these actual parameters were selected in a way that resulted in the targeted relation size, degree of inconsistency, and key-equal group size. This was done to simplify the logic, since calculating how many clean facts to generate and how many to duplicate based on the relation size and the required degree of inconsistency required rounding. In some cases, especially when the relation size was 10, this led to datasets that did not match the specified relation size or degree of inconsistency.

Where possible, operations were executed in parallel to distribute the workload across CPU cores. In earlier versions, computations were performed using a single-thread, single-core approach, which significantly increased the runtime. To utilize all available CPU resources and thereby speed up execution, Java's `parallelStream`[4] was employed in cases where parallel processing was feasible and free of side effects — such as when the next step in an iteration did not depend on the outcome of the previous step. For example, retrieving results for `SELECT` queries, purifying the database, and running the CQA algorithm on a list of `SELECT` query answers were executed in parallel.

5.3 Measuring the performance of the algorithm

The performance of the algorithm was evaluated by measuring the time taken in milliseconds to compute consistent answers from the input set. The purification step was included in the time measurement when it was enabled. For comparative purposes, the time taken to execute the initial `SELECT` query was also measured. In addition to execution time, several algorithm step counters were tracked to collect data for later analysis of the algorithm's computational behavior. These included:

- The number of times a block was checked.
- The number of times a potential k-set was checked.

[4] https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#parallelStream--

- The number of times a new k-set S was added to Δ.
- The number of times the condition $S' \subseteq S \cup \{a\}$ was evaluated

At the end of each query evaluation on the database, the values of these metrics, along with the input parameter values, were logged to a CSV file for later analysis.

6. Results

This chapter presents the results of running the implemented application on synthetic data, provides an analysis of the findings and conclusions along with perspectives for possible future developments. More detailed results are included in Appendices III and IV.

6.1 CQA algorithm running time

The most influential factors affecting query runtime were the number of atoms in the query and whether purification was used. Since queries with the same number of atoms and the same purification setting exhibited similar performance, the results are presented based on these two parameters, as shown in Figure 5.

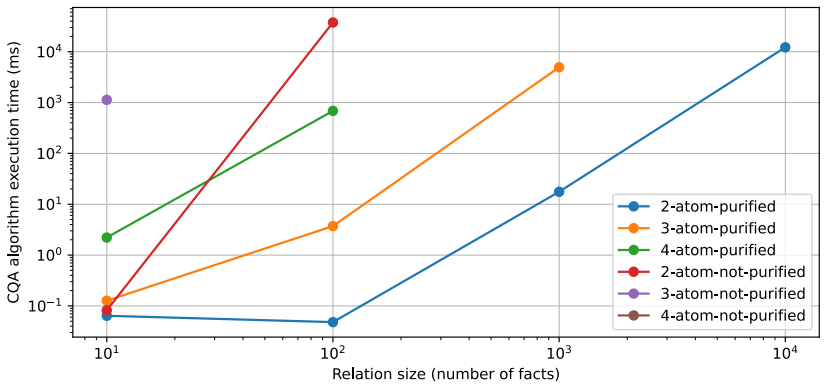

Figure 5. Average CQA algorithm running time per relation size and number of atoms in query

Purification had a significant impact on query execution time. Without purification, a 2-atom query exceeded the time limit with just 1,000 facts in the relation, a 3-atom query did so with 100 facts, and a 4-atom query with only 10 facts.

In contrast, purification enabled the processing of larger databases within the same time limit. The 2-atom query with purification never exceeded the time limit; however, when the number of facts increased tenfold from 1,000 to 10,000, the running time rose from approximately 10 to 10,000, reflecting a 1,000-fold increase. The 3-atom query with purification exceeded the time limit at a relation size of 10,000, while the 4-atom query did so at 1,000. It is also important to note that the time taken to purify the database was included in the runtime measurement when purification was enabled, whereas no such overhead was present in the unpurified cases.

Both the purified and unpurified results indicate poor scalability of query execution time. The computational cost increases rapidly, reaching infeasible levels even for moderately sized datasets. This suggests that the method may not be well-suited for processing large volumes of data.

6.2 Number of steps performed

In addition to running time, another insightful metric is the number of steps the algorithm performs. In this experiment, counters were introduced to record the number of times a block was checked (Figure 6); the number of times the condition $S' \subseteq S \cup \{a\}$ was evaluated (Figure 7); the number of times a candidate k-set was examined (Figure 8); and the number of times a new k-set was added to Δ (Figure 9).

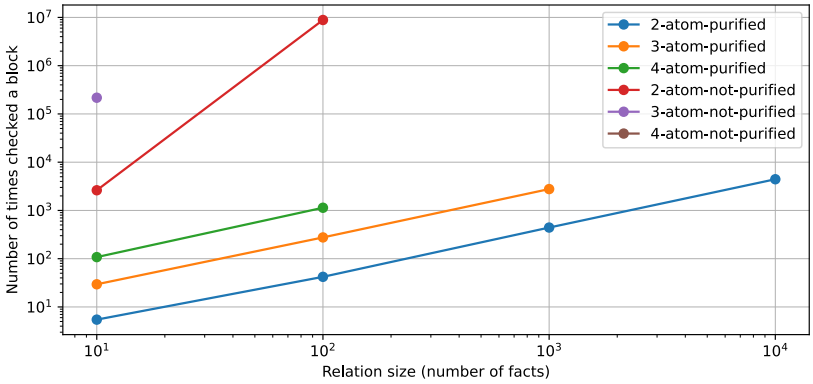

Figure 6. Average number of times a block was checked per relation size and number of atoms in query

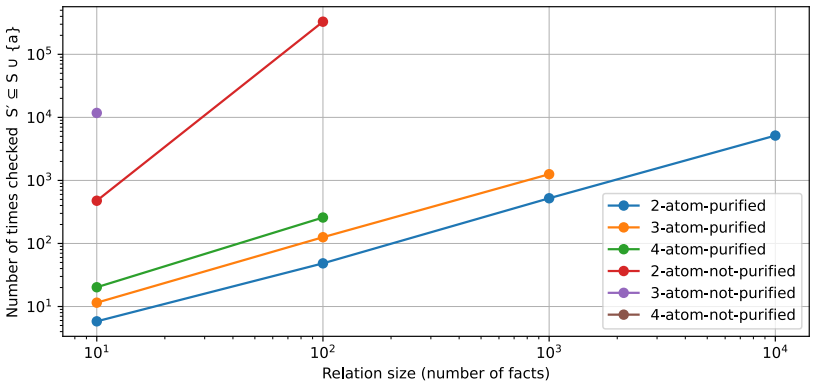

Figure 7. Average number of times the condition $S' \subseteq S \cup \{a\}$ was checked per relation size and number of atoms in query

The figures illustrate the reasons behind the low performance of executions without purification. With purification, the CQA algorithm performs significantly fewer steps. For instance, for a

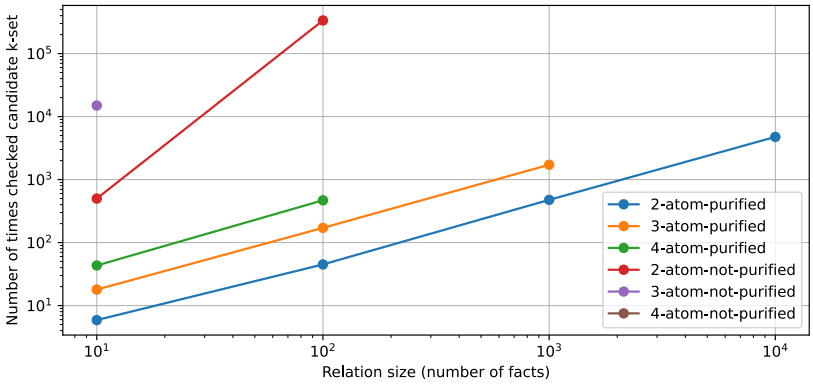

Figure 8. Average number of times a candidate k-set was checked per relation size and number of atoms in query

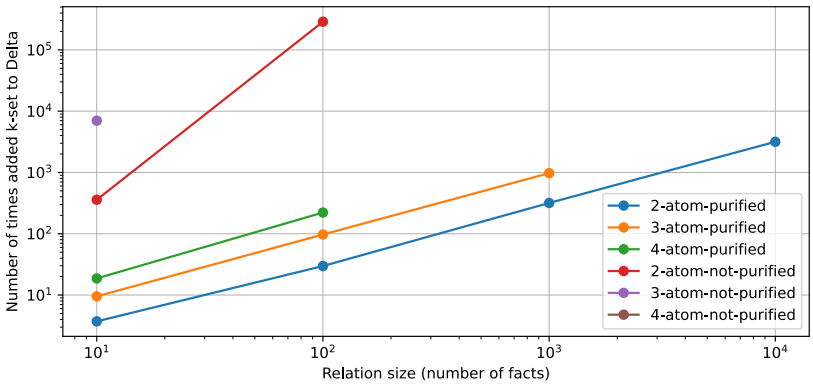

Figure 9. Average number of times a new k-set was added to Δ per relation size and number of atoms in query

2-atom query on a relation of size 100, the algorithm without purification checked a block over a million times, whereas with purification, it checked the same block only about 100 times on average. This discrepancy arises because, without purification, the algorithm must also iterate over many blocks in the database that are unrelated to the query. Nonetheless, the actual total number of steps with purification is higher, as it introduces additional operations that are not reflected in these metrics and were not counted in this experiment.

A similar relationship between purified and unpurified performance is observed in the results of the other step counters (Figures 7–9).

All results — including checking potential k-sets, adding a new k-set to Δ, and verifying the condition $S' \subseteq S \cup \{a\}$ — demonstrate a linear relationship between the number of facts in the relation and the number of steps performed. It is also evident that, for the same relation size,

queries with more atoms lead the algorithm to perform more steps, resulting in longer execution times.

6.3 Perspectives for future developments

Using purification to remove irrelevant facts often implies, especially as the database size grows, that the purification process itself becomes the most computationally expensive step, as it involves iterating over the entire database. In contrast, executing the CQA algorithm becomes less costly, since it operates on a relatively smaller set of facts that are present in any answer to the query found in the database. This raises the question of whether an advanced CQA algorithm is necessary at all, and whether it offers any advantage over the naive approach of simply identifying all repairs of the database, evaluating the query on each repair, and checking whether all answers are identical.

Although the naive approach may exhibit exponential time complexity, it may still be computationally less expensive in certain cases. The polynomial-time CQA algorithm implemented in this thesis may only outperform the naive approach when handling a large number of facts. Therefore, a comparative analysis between the naive algorithm and the implemented CQA algorithm is necessary to assess their relative performance.

Moreover, it was observed that purification may potentially serve as a standalone CQA algorithm. For certain queries examined in this thesis, a notable relationship emerges: answers to the query persist in the database after purification only if the query is certain. For example, consider the query q_1:

$$q_1 := \exists x, y, v, w \ (R_1(\underline{x}, y, C) \wedge R_2(\underline{y}, v, w))$$

If the query is not certain, there must exist facts of R_1 with distinct y values such that one fact contributes to an answer and the other does not. Such conflicting blocks of facts would be removed during purification. If no other blocks contribute to an answer, no answers would remain after purification, making it apparent that the query is uncertain; conversely, if any answers remain, the query is certain. (An analogous argument applies to R_2.)

Using purification as a CQA method for such queries would require a formal proof of correctness and, if necessary, modification to the approach to ensure coverage of all possible cases of input database. This leads to the question: for which instances of the CERTAINTY(q, Σ) problem does purification alone serve as a sound and complete CQA algorithm? To the best of the

author's knowledge, the use of purification as a standalone CQA algorithm has not been explored in the published literature.

In the experiments, all logic — including data handling — was implemented at the code level. To execute queries, the implementation sequentially iterated over all facts or possible combinations of facts. However, in a real-world setting, a database management system (DBMS) would likely be employed, leveraging optimizations such as indexing to accelerate query evaluation. Integrating the algorithm within a DBMS could significantly enhance its performance. For example, during the purification step, relevant facts could be efficiently retrieved using the DBMS, after which the CQA algorithm could be applied to this relatively smaller set of relevant facts.

Furthermore, additional optimizations could be explored in the current implementation. At present, the algorithm performs a relatively large number of steps, some of which could potentially be skipped without affecting the final outcome. Moreover, various parts of the implementation — such as helper methods for checking satisfaction conditions or methods used to verify the existence of certain elements — could possibly be rewritten to improve efficiency, thereby further enhancing the overall performance of the algorithm.

7. Summary

In this thesis, a Consistent Query Answering algorithm by Figueira et al. [1] was implemented, and its performance was evaluated. The algorithm was implemented in the Java programming language, selected for its object-oriented capabilities and the author's familiarity with it.

To conduct the experiments, synthetic data was generated in three phases: creating consistent data, inserting answers that satisfy specific queries, and introducing controlled inconsistencies. The use of synthetic data enabled control over both database size and inconsistency levels. Java objects were used to represent relations, with facts modeled as instances of these objects. No database management system was employed.

First, the answers to the non-Boolean conjunctive `SELECT` queries under evaluation were obtained. The algorithm then processed these queries by transforming them into Boolean queries, substituting the answers as constants into the original queries. The CQA algorithm was then applied to determine the certainty of the original answers, and as the final output, a list of certain answers was returned. Queries were represented in Java using logical expressions.

Database purification was employed to enhance performance and to enable a comparison with executions that did not use purification. Purification was implemented by filtering out irrelevant facts in two phases, corresponding to the original non-Boolean query and its transformed Boolean version, prior to executing the algorithm. This preprocessing step possibly reduces the number of computational steps required, resulting in more efficient query execution.

Performance was evaluated using the following metrics: execution time, the number of blocks checked, the number of candidate k-sets examined, and the number of k-sets added to Δ. The results indicated that purification significantly reduced both the running time and the number of steps performed by the algorithm. However, in both scenarios — whether purification was applied or not — the algorithm's runtime increased rapidly as the database size grew, indicating poor scalability.

Future developments could focus on optimizing the implementation of the algorithm presented in this thesis, comparing its performance with that of the naive approach to CQA, and integrating the algorithm with a database management system. Future work could also explore the potential of using purification as a standalone CQA algorithm. The current implementation provides a foundation for further research into the practical applications of this particular CQA algorithm.

References

[1] Figueira D., Padmanabha A., Segoufin L., and Sirangelo C. A Simple Algorithm for Consistent Query Answering under Primary Keys. *Logical Methods in Computer Science*, 2025, 21(1), pp. 18:1–18:43. https://doi.org/10.46298/lmcs-21(1:18)2025.

[2] Bertossi L. Database Repairing and Consistent Query Answering. Cham, Switzerland: Springer Nature Switzerland. 2011. https://doi.org/10.1007/978-3-031-01883-1_1.

[3] Wijsen J. Certain conjunctive query answering in first-order logic. *ACM Trans. Database Syst.*, 2012, 37(2), pp. 1–35. https://doi.org/10.1145/2188349.2188351.

[4] Ihab F. Ilyas X. C. Trends in Cleaning Relational Data: Consistency and Deduplication. *Foundations and Trends® in Databases*, 2015, 4(5), pp. 281–393. https://doi.org/10.1561/1900000045.

[5] Wijsen J. Making More Out of an Inconsistent Database. *Advances in Databases and Information Systems*. Berlin, Germany: Springer-Verlag, 2004, pp. 291–305. https://doi.org/10.1007/978-3-540-30204-9_20.

[6] Liu B. Inconsistent data repairs in database integrations. *2013 Ninth International Conference on Natural Computation (ICNC)*. Piscataway, USA: Institute of Electrical and Electronics Engineers (IEEE), 2013, pp. 1140–1144. https://doi.org/10.1109/ICNC.2013.6818149.

[7] Assadi A., Milo T., and Novgorodov S. Cleaning Data with Constraints and Experts. *WebDB'18: Proceedings of the 21st International Workshop on the Web and Databases*. New York, USA: Association for Computing Machinery, 2018, 1:1–1:6. https://doi.org/10.1145/3201463.3201464.

[8] Arenas M., Bertossi L., and Chomicki J. Consistent query answers in inconsistent databases. *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, USA: Association for Computing Machinery, 1999, pp. 68–79. https://doi.org/10.1145/303976.303983.

[9] Wijsen J. Charting the tractability frontier of certain conjunctive query answering. *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. New York, USA: Association for Computing Machinery, 2013, pp. 189–200. https://doi.org/10.1145/2463664.2463666.

[10] Calì A., Lembo D., and Rosati R. On the decidability and complexity of query answering over inconsistent and incomplete databases. *PODS '03*. San Diego, Califor-

nia: Association for Computing Machinery, 2003, pp. 260–271. https://doi.org/10.1145/773153.773179.

[11] Fuxman A. and Miller R. J. First-order query rewriting for inconsistent databases. *Journal of Computer and System Sciences*, 2007, 47(3), pp. 610–635. https://doi.org/10.1016/j.jcss.2006.10.013.

[12] Koutris P. and Wijsen J. Consistent Query Answering for Self-Join-Free Conjunctive Queries Under Primary Key Constraints. *ACM Trans. Database Syst.*, 2017, 2, pp. 9:1–9:45. https://doi.org/10.1145/3068334.

[13] Dixit A. A. and Kolaitis P. G. A SAT-Based System for Consistent Query Answering. *Theory and Applications of Satisfiability Testing – SAT 2019*. Cham, Switzerland: Springer Nature Switzerland, 2019, pp. 117–135. https://doi.org/10.1007/978-3-030-24258-9_8.

Appendices

I. Queries Used in Experiments

Note: *Primary key attributes are indicated by underlining within each relational atom.*

$$q_1(z) := \exists x, y, v, w\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, v, w))$$

$$q_2(z, w) := \exists x, y, v\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, v, w))$$

$$q_3(z) := \exists x, y, v, u, d\ (R_1(\underline{x}, y, z) \wedge R_3(\underline{y}, v) \wedge R_2(\underline{v}, u, d))$$

$$q_4(z, d) := \exists x, y, v, u\ (R_1(\underline{x}, y, z) \wedge R_3(\underline{y}, v) \wedge R_2(\underline{v}, u, d))$$

$$q_5(z) := \exists x, y, v, w\ (R_1(\underline{x}, y, z) \wedge R_4(\underline{y, v}, w))$$

$$q_6(z) := \exists x, y, x_0, w, d\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{x'}, y, w) \wedge R_5(\underline{x}, y, d))$$

$$q_7(z) := \exists x, y, w, d\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w) \wedge R_5(\underline{x}, y, d))$$

$$q_8(z, w) := \exists x, y\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w))$$

$$q_9(z) := \exists x, y, w, u, d\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w) \wedge R_4(\underline{y}, u, d))$$

$$q_{10}(z, w, d) := \exists x, y, u\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w) \wedge R_4(\underline{y}, u, d))$$

$$q_{11}(z) := \exists x, y, w\ (R_1(\underline{x}, y, z) \wedge R_2(\underline{y}, x, w))$$

$$q_{12}(v, d) := \exists x, y, z, u\ (R_3(\underline{x}, y) \wedge R_6(\underline{y}, z) \wedge R_1(\underline{z}, x, d) \wedge R_4(\underline{x, u}, v))$$

$$q_{13}(v) := \exists x, y, z, u\ (R_3(\underline{x}, y) \wedge R_6(\underline{y}, z) \wedge R_7(\underline{z}, x) \wedge R_4(\underline{x, u}, v))$$

$$q_{14}(d) := \exists x, y, z, u\ (R_3(\underline{x}, y) \wedge R_6(\underline{y}, z) \wedge R_1(\underline{z}, x, d) \wedge R_7(\underline{x}, u))$$

II. Implemented CQA algorithm Java method

```
public boolean isQueryCertainOnGivenDatabase(
        Database database, Query query) {

    int checkedSUnionA = 0;
    int checkedPotentialKSet = 0;
    int checkedBlock = 0;
    int addedNewKSetToDelta = 0;

    Delta delta = new Delta();
    Blocks blocks = new Blocks();
    KSets kSets = new KSets();

    delta.initialize(query, database);
    blocks.initialize(database);

    boolean thereMightBeMoreKSetsToAddToDelta;

    do {
        kSets.initialize(database.getDatabase(), query.getK());
        thereMightBeMoreKSetsToAddToDelta = false;

        while (kSets.hasNext()) {
            HashSet<Fact> S = kSets.next();
            if (delta.set.contains(S)) continue;

            for (HashSet<Fact> B : blocks.set) {
                boolean shouldAddSToDelta = true;

                for (Fact a : B) {
                    HashSet<Fact> SUnionA = SetUtils.union(S, a);

                    if (!thereExistsSPrimeThatIsSubsetOfSUnionA(
                            delta.set, SUnionA)) {
                        shouldAddSToDelta = false;
                        break;
                    }
                    checkedSUnionA++;
                }
```

```
39
40                  if (shouldAddSToDelta) {
41                      if (S.isEmpty()) {
42                          concludeResults(checkedSUnionA,
43                                  checkedPotentialKSet, checkedBlock,
44                                  addedNewKSetToDelta);
45                          return true;
46                      }
47
48                      delta.set.add(S);
49                      addedNewKSetToDelta++;
50                      thereMightBeMoreKSetsToAddToDelta = true;
51                      break;
52                  }
53
54                  checkedBlock++;
55              }
56              checkedPotentialKSet++;
57          }
58      } while (thereMightBeMoreKSetsToAddToDelta);
59
60      concludeResults(checkedSUnionA, checkedPotentialKSet,
61              checkedBlock, addedNewKSetToDelta);
62      return false;
63  }
64
```

III. Results with purification

Table 5. Average execution time of the CQA algorithm (ms)

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12	q_13	q_14
10	0.04	0.16	0.44	0.04	0.04	0.08	0.08	0.04	0.08	0.04	0.04	1.28	2.76	2.6
100	0.1067	0.0267	3.8667	3.7067	0.0233	3.6	3.7267	0.0667	3.7167	3.6767	0.0167	329.78	911.8667	817.3833
1000	28.23	17.6233	5180.85	5255.5733	19.5933	4438.41	4721.51	16.94	4967.5933	5110.1833	16.1633	-	-	-
10000	13581.5133	12135.6433	-	-	14351.36	-	-	12010.7667	-	-	10401.37	-	-	-

Table 6. Average number of times the condition $S' \subseteq S \cup \{a\}$ was checked

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12	q_13	q_14
10	6.6	4.8	12.56	9.8	8.76	10.08	9.8	4.44	18.12	8.68	5.28	17.4	16.8	26.44
100	72.8433	39.49	181.3633	93.59	74.98	93.8467	94.29	39.53	196.51	94.2433	39.56	195.2	196.5	381.9067
1000	788.76	417.17	1859.9867	942.48	833.69	943.32	944.2767	416.8	1896.4467	942.5967	417.26	-	-	-
10000	8052.42	4185.81	-	-	8014.2	-	-	4187.27	-	-	4190.59	-	-	-

Table 7. Average number of times a candidate k-set was checked

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12	q_13	q_14
10	6.36	5.64	18.44	16.64	7.44	16.36	16	4.96	24.44	15.88	5.52	39.8	38.68	51.4
100	54.8767	43.11	203.67	153.7433	55.6367	148.8933	149.61	40.5667	216.54	154.9867	40.6433	415.5467	418.6933	572.5633
1000	583.8433	453.1233	2091.5567	1553.3067	598.1367	1497.5767	1498.94	426.9067	2118.7233	1552.59	427.2033	-	-	-
10000	5916.6333	4556.9233	-	-	5892.6067	-	-	4289.4	-	-	4291.5967	-	-	-

Table 8. Average number of times a block was checked

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12	q_13	q_14
10	5.72	5.44	29.92	27.88	6.68	27.36	26.6	4.44	38.44	26.76	5.28	101.24	98.08	124.28
100	45.6067	42.53	302.1733	265.0167	46.4687	254.79	255.9933	39.58	310.2167	267.5933	39.68	1059.4933	1067.9633	1283.13
1000	481.1267	447.7167	3066.8967	2679.9567	485.7967	2560.44	2563.0367	416.86	3082.7533	2678.2367	417.32	-	-	-
10000	4846.1933	4500.17	-	-	4842.4833	-	-	4187.51	-	-	4190.73	-	-	-

Table 9. Average number of times a k-set was added to Δ

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12	q_13	q_14
10	4.24	3.2	10.08	8.4	5.2	8.64	8.4	2.96	14.16	7.44	3.52	16.24	15.68	24.08
100	39.5933	26.26	124.8567	80.2	40.16	80.42	80.82	26.28	135.99	80.76	26.3	182.1867	183.4	100.1067
1000	423.59	278.0267	1293.68	807.84	437.2667	808.56	809.38	277.78	1317.3133	807.94	278.0933	-	-	-
10000	4306.51	2790.44	-	-	4283.9533	-	-	2791.4133	-	-	2793.6467	-	-	-

Table 10. Average Execution Time of SELECT Query (ms)

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12	q_13	q_14
10	0	0.04	0.12	0.04	0	0.04	0.04	0.04	0.04	0.04	0	0.56	0.88	0.52
100	0.0633	0.0433	5.7033	5.7033	0.0267	5.7	5.4633	0.01	5.47	5.5333	0.0067	684.1833	690.8267	672.82
1000	3.74	3.8667	5247.89	5023.6333	3.7867	5186.0667	4909.3067	3.92	5256.45	5207.6567	4.0967	-	-	-
10000	385.5567	483.91	-	-	483.7167	-	-	482.9	-	-	482.7367	-	-	-

IV. Results without purification

Table 11. Average execution time of the CQA algorithm (ms)

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12	q_13	q_14
10	0.96	1.6	1236.72	1052.2	0.28	1173.16	1100.72	0.92	872.96	1355.28	0.48	-	-	-
100	30267.01	44859.1467	-	-	31819.49	-	-	36834.52	-	-	37095.5367	-	-	-
1000	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10000	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 12. Average number of times the condition $S' \subseteq S \cup \{a\}$ was checked

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12	q_13	q_14
10	469.8	460.28	12513.44	11051.92	538.48	10292.64	11747.92	465.8	13070.4	11873.72	440.52	-	-	-
100	341540.7733	319692.9467	-	-	362352.46	-	-	306574.3767	-	-	328217.6533	-	-	-
1000	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10000	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 13. Average number of times a candidate k-set was checked

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_10	q_11	q_12	q_13	q_14
10	474	544.52	14862.16	15571	492.84	13462.36	13759.12	511.68	13927.4	18358.16	449.04	-	-	-
100	332141.21	366342.09	-	-	333154.03	-	-	322661.3967	-	-	321334	-	-	-
1000	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10000	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 14. Average number of times a block was checked

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}	q_{13}	q_{14}
10	1926.92	3796.76	205357.4	243733.8	1123.6	178026.96	195461.8	3686.04	142967.64	335045.8	1870.8	-	-	-
100	4132858.727	15864160.48	-	-	3858682.323	-	-	8056503.043	-	-	7614322.367	-	-	-
1000	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10000	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 15. Average number of times a k-set was added to Δ

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}	q_{13}	q_{14}
10	372.36	338	7295.76	6576.04	435.84	6899.16	6556.24	311.28	8672.72	5975.6	349	-	-	-
100	309774.48	280162.72	-	-	312385.7433	-	-	279083.6267	-	-	280279.35	-	-	-
1000	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10000	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 16. Average Execution Time of SELECT Query (ms)

rSize	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}	q_{13}	q_{14}
10	0	0.04	0.12	0.04	0	0.04	0.04	0.04	0.04	0.04	0	-	-	-
100	0.0633	0.0433	-	-	0.0267	-	-	0.01	-	-	0.0067	-	-	-
1000	-	-	-	-	-	-	-	-	-	-	-	-	-	-
10000	-	-	-	-	-	-	-	-	-	-	-	-	-	-

License

Non-Exclusive Licence to Reproduce the Thesis and Make the Thesis Public

I, **Mattias Kimst**,

1. grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis **Implementing and Testing a Simple Algorithm for Consistent Query Answering**, supervised by **Miika Juhani Hannula**;
2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;
3. am aware of the fact that the author retains the rights specified in points 1 and 2;
4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Mattias Kimst

12/05/2025