TUHH
Hamburg University of Technology

Master Thesis

Jan Holste

Ontology-Based Temporal Reasoning on Streams

October 13, 2014

supervised by:
Prof. Dr. Ralf Möller
Prof. Dr. Karl-Heinz Zimmermann

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

TUHH STS
Software Technology Systems

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den _____

Jan Holste

Contents

1 Introduction

Nowadays a lot of environments are monitored by sensors. With the measured sensor data, the current status of systems should be analysed and future failures should be predictable. The status of a system depends on the previous events in terms of sensor data and its current changes. The sensors are able to produce a stream of data, which represents their measurements. The stream that represents the status of the system is a sequence of intervals that contains the measurements of the sensors. Each interval represents a time slot that comprehends measurements of sensors.

To analyse the status of the system, one can create queries that describe specific behaviour of the system and checks whether the system behave in this described way. A query should be able to move over the sequence of intervals and should be queried against specific sensor data in the intervals. Queries in the context of Temporal Query Answering are queried against a temporal knowledge base that contains two parts. The first part is a sequence of A-Boxes. Each A-Box can represent an interval with sensor data. The sensor data is described by assertions. An A-Box is a set of assertions. The second part is the T-Box, also called ontology. It contains inclusions of assertions. An inclusion describes the definition of assertions based on other assertions.

The algorithm [1], [2] defines the Answering of Temporal Queries. It uses Conjunctive Queries (CQs) to query against single A-Boxes. To move over the sequence of A-Boxes it uses the operators of the Linear Temporal Logic (LTL). The combination of the operators and the Conjunctive Queries forms the Temporal Conjunctive Query (TCQ).

The operators can be divided into two types: The first type moves by one step over the sequence. The next or previous operator are of this type. They move only by one A-Box or interval. The second type moves over the sequence depending on the current sequence and position of the query. The operator always in the past uses all previous A-Boxes and calculates their intersection. Depending on the position of the query, the query evaluates the always in the past operator over all previous A-Boxes. If a given system runs a long time, the operator always in the past uses all sensor data to query the status of the system. Not always the status of the system is depending on the entire run time. Sometimes only a specific amount of time of the past is important. To describe the status of a system in a limited past, one can rewrite the always in the past to a construct, which uses the previous operators and conjunctions. The new query size depends on the size of the limited past. With a sliding window, the always in the past operator can be used and the handling of a limited past is done by the rewriting algorithm for the sliding window. This simplifies the creation of queries with limited history.

A sliding window is a window moving over the sequence of A-Boxes. The window has a fixed size. Furthermore it is moving over the sequence with a specific step size, also called sliding. The sliding window not only simplifies the creation of queries with a limited past, it is also needed to implement specific operations that need all individual data to calculate their result. The paper [6] classifies operations into classes that need different information at a time to evaluate a result. The classes holistic and context-

sensitive can only be calculated by knowing all individual assertions of the sequence of A-Boxes at once. The median and the histogram are examples for these classes. Due to this fact, a sliding window is needed to use this operations.

In this thesis, two different approaches to establish a sliding window are discussed. The first approach is the rewriting of the TCQ to meet the boundaries of the window and using the algorithm [1] to evaluate the query. The Chapter 2 describes the fundamentals of Temporal Query Answering in DL-Lite. I introduce the theory of the Sliding Window Algorithm in Chapter 3. It contains the rewriting of a TCQ to meet the boundaries of the sliding window and it represents an optimised version of the Sliding Window Algorithm for the case that the sliding is larger than the range.

The second approach is to consume the entire window at once. Therefore the input is a sequence of batches. Each batch is a sequence of A-Boxes and represents a window. The original algorithm can be queried over each batch to establish TCQ Answering with a sliding window. In Chapter 4, I introduce my approach of the Vector Algorithm. It calculates only intermediate results that are used to evaluate the query for a given batch. Furthermore I represent a rewriting algorithm to transform a TCQ into a Vector Algorithm Query. Chapter 5 contains my definition of the Optimised Rewriting Algorithm and the Optimised Vector Algorithm. It is an optimised version of the Vector Algorithm that re-uses already intermediate results, if the windows are overlapping.

I implemented all algorithms in Java. In Chapter 6, I represent the benchmark results of my implementations and discuss the optimal algorithm to be used. The Vector Algorithm should be used to establish a sliding window that supports holistic and context-sensitive operations. At the end, there is a conclusion of this thesis.

2 Temporal Query Answering in DL-Lite

The Temporal Query Answering Algorithm [1] answers queries against a stream of data. This data is represented by an Temporal Knowledge Base. The Temporal Knowledge Base consists of two parts. The first part is a sequence of A-Boxes. Each A-Box represents assertions for a specific point in time. Therefore the sequence of A-Boxes describes the change of the environment over time. The second part is the T-Box, also called ontology. It holds at all points in time and contains inclusions. An inclusion describes the entailment between two assertions. It represents that one assertion is a subset of a more general assertion.

Temporal Conjunctive Queries are queried over the sequence of A-Boxes. The algorithm [1] answers Temporal Conjunctive Queries by moving over the entire sequence. To achieve this, the algorithm defines four functions. The first function is the Initial Answer Formula, which calculates the first intermediate answer for the first A-Box of the sequence. The Next Answer Formula and the Update Formula are called for each next A-Box in the sequence, till the end. To get a result for a Temporal Conjunctive Query at the current position, the Evaluation Function is called. The project work [5] describes the implementation of the Temporal Query Answering Algorithm in detail.

2.1 Example

A Temporal Conjunctive Query (TCQ) uses Conjunctive Queries (CQs) to query against an A-Box. Every CQ is a TCQ. The combination of the temporal operator strong next ($\circ$) or strong previous ($\circ^-$) with an TCQ is also a TCQ. The result of the strong previous operator is the empty set, if there exists no previous A-Box. Otherwise the result is the result of the sub-query queried against the previous A-Box. The result of the strong next operator is the result of the sub-query queried against the next A-Box, if there exists such an A-Box. Otherwise it is the empty set. Let $\Psi = \circ^-(\circ(hasVal(S1, x)))$ be a TCQ. Ψ has following sub-formulas:

- $\psi_0 = hasVal(S1, x)$
- $\psi_1 = \circ(hasVal(S1, x))$
- $\psi_2 = \circ^-(\circ(hasVal(S1, x)))$

The sub-formulas are ordered. A sub-formula of another sub-formula has always a lower index. By calculating the sub-formulas in this order, every sub-formula has already calculated sub-formulas. Let the Temporal Knowledge Base be the following sequence of A-Boxes (Figure 2.1) and an empty ontology. The next Figure 2.2 shows all intermediate results as nodes. The call of the Initial Answer Formula Φ_0 calculates the results of all sub-formulas for the first ABox ABox_0.

$ABox_0$	$ABox_1$	$ABox_2$	$ABox_3$
`hasVal(S1,3.0)`	`hasVal(S1,2.0)`	`hasVal(S1,3.5)`	`hasVal(S1,4.0)`

Figure 2.1: Finite Sequence of A-Boxes

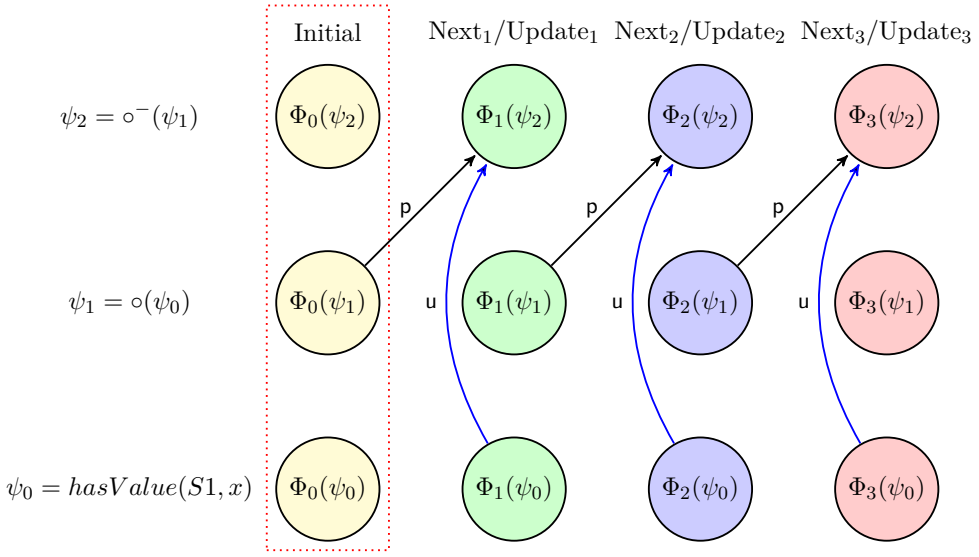


Figure 2.2: Intermediate Results

The Initial Answer Formula calculates the result for $\Phi_0(\psi_0)$ at first. The result is a mapping of free variables to fixed values. The sub-formula $\psi_0 = hasVal(S1, x)$ has the free variable x. The first ABox has the role assertion $hasVal(S1, 3.0)$. The result of this sub-formula queried against the $ABox_0$ is the mapping $\{\{x \leftarrow 3.0\}\}$.

The second sub-formula is $\psi_1 = \circ(\psi(0))$. It uses the strong next operator, which result will be the achieved by querying the sub-formula ψ_0 against the next ABox $ABox_1$. This will be done in the next ABox. Therefore a placeholder represents the result and will be replaced in the next step by the Update Function. The placeholder variable $x_0^{\circ\psi_0}$ denotes a placeholder variable introduced at point in time zero with the sub-formula ψ_0.

The last sub-formula is $\psi_2 = \circ^-(\psi_1)$. It uses the strong previous operator. The result for this operator is the empty set, because there is no previous result for the sub-query ψ_1. This is always the case for the Initial Answer Formula and operators using results of the past.

The entire result for the Initial Answer Formula are three answer formulas represented by yellow nodes in the Figure 2.2. The answer formulas are $\Phi_0(\psi_0) = \{\{x \leftarrow 3.0\}\}$, $\Phi_0(\psi_1) = x_0^{\psi_0}$ and $\Phi_0(\psi_2) = \emptyset$, where the empty set is equal to false.

To calculate the next set of answer formulas, represented by the green nodes in Figure 2.2, the Next Answer Formula is needed. It is like the Initial Answer Formula but uses also answer formulas of the previous ABox. This was not the case for the Initial Answer Formula, because there was no previous set of answer formulas. The first sub-

query $\psi_0 = hasVal(S1, x)$ queried against the ABox_1 has the result $\{\{x \leftarrow 2.0\}\}$. This mapping is equal to the first one of the Initial Answer Formula, but is the result of the sub-query queried against the second ABox ABox_1. The second sub-query $\psi_1(\circ(\psi_0))$ introduces a placeholder variable $x_1^{\circ\psi_0}$ as in the Initial Answer Formula, but the point in time is changed to 1. The last sub-query $\psi_2 = \circ^-(\psi_1)$ uses the strong previous operator. This time there is a set of previous answer formulas. Therefore the result is ψ_1 of the previous set of answer formulas, which is the placeholder variable $x_0^{\circ\psi_0}$. The arrow with p for previous denotes the use of previous sets of answer formulas in the Figure 2.2.

The Next Answer Formula has calculated three new answer formulas like the Initial Answer Formula. The only difference is the use of the previous calculated answer formulas. The previous answer formula in this example is a placeholder variable $x_0^{\circ\psi_0}$, which is introduced by the Initial Answer Formula. After the execution of the Next Answer Formula, the result of the sub-query ψ_2 is equal to this placeholder variable. This placeholder variable represents a set of answer formulas of the current point in time. Therefore it can be replaced by the actually set of answer formulas. In this example, the placeholder variable $x_0^{\circ\psi_0}$ is equal to $\Phi_1(\psi_0)$. The task of replacing placeholder variables of the previous step is done by the Update Formula. The Update Formula is always called after the Next Answer Formula. Therefore at every step of execution, there are only placeholder variables of the current and the previous point in time. All other placeholder variables are already replaced. Figure 2.2 represents the update function as an arrow with the label u. The execution of the Next Answer Formula and the Update Formula is repeated till the end of the finite sequence of A-Boxes at ABox_3.

The result of the algorithm is the set of answer formulas of the sub-query ψ_2. This set of answer formulas can contain placeholder variables, which are introduced by the Initial or Next Answer Formula. The placeholder variables are placeholder for sets of answer formulas in the next point in time. The current point in time is the last point, that means no set of answer formulas for the next point in time exists. The Evaluation Function replaces all placeholder variables with the specific values. In case of a strong next it would be the empty set. A weak next would result in a set of all possible solutions.

This example shows the main idea behind the algorithm [1]. In the next section, we introduce the syntax and semantics of a few more operators.

2.2 DL-Lite

A Temporal Conjunctive Query (TCQ) uses Conjunctive Queries (CQs) to query against a single ABox. A Conjunctive Query (CQ) has the form $\phi = \exists y_1, \ldots, y_m.\psi$, where ψ represents a conjunction of atoms. An atom is either a concept or a role. A concept has a predicate and subject. A role has an additional object. In the example above, $hasVal(S1, 3.0)$ is a role with the predicate $hasVal$ and the subject $S1$, which stands for sensor one. Furthermore there is an object 3.0, which is a double value of the temperature of the sensor. A CQ has also a scope for the variables. $y_1, \ldots, y_m$ are bounded variables, which are bounded to the CQ.

CQs are described by the description logic DL-Lite. This description logic can be built

on top of classical relational databases to retrieve the data as ABox assertions. The use of DL-Lite to query databases with ontology-based data access is described by [3].

2.2.1 Syntax

An ABox contains a finite set of assertions in terms of concepts and roles. This assertions can contain individual names. For instance $S1$ is an individual name for sensor one in the concept $isSensor(S1)$ or in the role $hasVal(S1, 3.90)$. DL-Lite has sets of symbols to represent the different names. The set N_C is the set of concept names, N_R is the set of role names and N_I is the set of individual names. These sets are not empty and pairwise disjoint.

DL-Lite has two types of concepts. The general concepts and the basic concepts. A basic concept has the form $A \in N_C$ or $\exists R \in N_R$. A general concept is a basic concept or the negation of it. A role expression is a role name $P_1 \in N_R$ or the inverse of a role $P_1^- = P_2$ with $P_2 \in N_R$. A concept inclusion is of the form $B \sqsubseteq C$, where B is a basic concept and C a general concept. A finite set of concept inclusions is a T-Box also called ontology.

2.2.2 Semantics

The non-empty set Δ^I is called domain. Together with the assignment function $\cdot^I$ it forms an interpretation $I = (\Delta^I, \cdot^I)$. The assignment function $\cdot^I$ assigns to every concept, role and individual name a subset of the domain Δ^I. To every concept $A \in N_C$ it assigns a subset $A^I \subseteq \Delta^I$. To every role P it assigns a binary relation $P^I \subseteq \Delta^I \times \Delta^I$ and every individual name $a \in N_I$ gets an element $a^I \in \Delta^I$. Furthermore an interpretation of an inverse role $(P^-)^I$ is the inverse of all pairs in P^I. This set of binary relations is defined as $(P^-)^I := \{(e, d) | (d, e) \in P^I\}$. A concept of the form $\exists R \in N_C$ is interpreted as all individual names that have an outgoing relation R. This set is defined as $(\exists R)^I := \{d|$ there is an e $\in \Delta^I$ such that $(d, e) \in R^I\}$. The interpretation of a negated concept is the interpretation of the entire domain without the interpretation of this specific concept. This is defined as $(\neg C)^I := \Delta^I \setminus C^I$.

The interpretation I is a model of an axiom α iff:

- $B^I \subseteq C^I$ if $\alpha = B \sqsubseteq C$
- $a^I \in A^I$ if $\alpha = A(a)$
- $(a^I, b^I) \in P^I$ if $\alpha = P(a, b)$

Iff the interpretation I is a model of an axiom α, then I models α ($I \models \alpha$) is true. Iff the interpretation I is a model for all assertions in the ABox A, then $I \models A$ is true. Iff I is a model for all concept inclusions in the T-Box T, then $I \models T$ is true. Assuming that the interpretations of ABox and T-Box have unique individual names and two individual names have two different interpretations, then the union of both interpretations is consistent.

2.3 Temporal Conjunctive Queries

In the previous section, the syntax and semantics of DL-Lite were introduced. With this knowledge, interpretations for a given ABox and T-Box can be created and checked against their correctness. An interpretation is correct, iff the interpretation models the ABox and T-Box. In Temporal Conjunctive Query Answering, the queries are temporal and queried against a Temporal Knowledge Base. A Temporal Knowledge Base is a sequence of A-Boxes and one T-Box. Let $K = \langle (A_i)_{0 \leq i \leq n}, T \rangle$ be a Temporal Knowledge Base, then T is a T-Box and $(A_i)_{0 \leq i \leq n}$ is a finite sequence of n A-Boxes, which describes the changes over time. This section extends the interpretation and correctness checks of a single ABox and T-Box to a Temporal Knowledge Base.

2.3.1 Syntax

A CQ is limited to query against a single ABox. With a Temporal Conjunctive Query (TCQ) a query over a sequence of A-Boxes is achieved. A TCQ uses CQs to query against one ABox. Every CQ is a TCQ. The set of all variables of a TCQ ϕ is denoted as $Var(\phi)$. The set $FVar(\phi)$ denotes the set of all free variables. A free variable is a placeholder for a substitution without a specific value. A variable is bounded, if a value is assigned to it. A TCQ is boolean, if the set of free variables is empty.

The temporal part of TCQs is achieved by the Linear Temporal Logic (LTL). The LTL operators specify the A-Boxes, a CQ is queried against. They can define ranges and specific points in time. The combination of a TCQ with an LTL operator is also a TCQ. Let ϕ_1 and ϕ_2 be TCQs, then the following expressions are also TCQs:

- $\phi_1 \wedge \phi_2$ (conjunction)
- $\phi_1 \vee \phi_2$ (disjunction)
- $\circ\phi_1$ (strong next), $\bullet\phi_1$ (weak next)
- $\circ^-\phi_1$ (strong previous), $\bullet^-\phi_1$ (weak previous)
- $\Box\phi_1$ (always), $\diamond\phi_1$ (eventually)
- $\Box^-\phi_1$ (always in the past), $\diamond^-\phi_1$ (history)
- $\phi_1 U \phi_2$ (since), $\phi_1 S \phi_2$ (until)

2.3.2 Semantics

To introduce the semantics, ψ defines a boolean CQ. A boolean CQ has a boolean result and the set of bounded variables through the exists operator is equal to the set of all free variables of the atoms. Let $Var(\psi)$ be the set of all variables of ψ, Δ the domain, N_I the set of all individual names and $I = (\Delta, \cdot^I)$ be an interpretation, then there is a homomorphism $\pi : Var(\psi) \cup N_I \to \Delta$ of ψ iff:

- $\pi(a) = a^I$ for all $a \in N_I$
- $\pi(z) \in A^I$ for all $A(z) \in \psi$
- $(\pi(z_1), \pi(z_2)) \in r^I$ for all $r(z_1, z_2) \in \psi$

If there is such a mapping π, then I is a model for the boolean CQ ψ and the mapping π is an answer. Let ϕ be a boolean TCQ and $J = (I_i)_{0 \leq i \leq n}$ a sequence of interpretations, then $J, i \models \phi$ is defined by induction as follows:

- $J, i \models \exists y_1 \dots y_m \cdot \psi$ iff $I_i \models \exists y_1 \dots y_m \cdot \psi$
- $J, i \models \phi_1 \wedge \phi_2$ iff $J, i \models \phi_1$ and $J, i \models \phi_2$
- $J, i \models \phi_1 \vee \phi_2$ iff $J, i \models \phi_1$ or $J, i \models \phi_2$
- $J, i \models \circ\phi_1$ iff $i < n$ and $J, i + 1 \models \phi_1$
- $J, i \models \bullet\phi_1$ iff $i < n$ implies $J, i + 1 \models \phi_1$
- $J, i \models \circ^-\phi_1$ iff $i > 0$ and $J, i - 1 \models \phi_1$
- $J, i \models \bullet^-\phi_1$ iff $i > 0$ implies $J, i - 1 \models \phi_1$
- $J, i \models \Box\phi_1$ iff for all k with $i \leq k \leq n$ i have $J, k \models \phi_1$
- $J, i \models \Diamond\phi_1$ iff there is a k such that $i \leq k \leq n$ and $J, k \models \phi_1$
- $J, i \models \Box^-\phi_1$ iff for all k, $0 \leq k \leq i$ i have $J, k \models \phi_1$
- $J, i \models \Diamond^-\phi_1$ iff there is some k, $0 \leq k \leq i$ such that $J, k \models \phi_1$
- $J, i \models \phi_1 U \phi_2$ iff $\exists k, i \leq k \leq n$ such that $J, k \models \phi_2$ and $J, j \models \phi_1$ for all $j, i \leq j < k$
- $J, i \models \phi_1 S \phi_2$ iff $\exists k, 0 \leq k \leq i$ such that $J, k \models \phi_2$ and $J, j \models \phi_1$ for all $j, k < j \leq i$

In TCQ Answering, the answer to a specific TCQ at the last point in time n is interesting, under the assumption that no point in time is before 0 and after n. A sequence of interpretations J is a model of the TCQ ϕ w.r.t. a Temporal Knowledge Base K, if $J \models K$ and $J, n \models \phi$. The TCQ ϕ is called satisfiable, if such a model exists.

The answering of a TCQ ϕ w.r.t a given Temporal Knowledge Base K is the answering of a compiled TCQ ϕ' against the sequence of interpretations of the database. These sequence is $\langle DB(A) \rangle_{0 \leq i \leq n}$. Each $DB(A_i)$ represents an interpretation of an ABox in the Temporal Knowledge Base. Every interpretation $DB(A) := (N_I, \cdot^{DB(A)})$ is defined as follows:

- $a^{DB(A)} := a$ for all $a \in N_I$
- $A^{DB(A)} := \{a | A(a) \in A\}$ for all $A \in N_C$
- $P^{DB(A)} := \{(a, b) | P(a, b) \in A\}$ for all $P \in N_R$

$ABox_0$
`hasVal(S1,3.0)`
`hasVal(S2,2.3)`
`isSensor(S3)`

TBox
`isSensor` $\sqsubseteq \exists$ `hasVal`

Figure 2.3: An ABox and a TBox

2.4 CQ Rewriting

In the last section, the TCQs are introduced. They use CQs to query against single A-Boxes, represented as interpretations of a database. This mappings only contain the assertions of A-Boxes, not the concept inclusions of the ontology. To include the ontology into the query, the query has to be rewritten. In this section, the rewriting algorithm for TCQs is presented.

2.4.1 Example

Let $\psi(x) \leftarrow hasVal(x, y)$ be a CQ with a bounded variable x and a free variable y. Then this query should return all possible mappings for x. In this case the result is the set of all sensor names. Given the following ABox and T-Box in Figure 2.3, then the ABox has three assertions. The result of the CQ $\psi(x) \leftarrow hasVal(x, y)$ is a mapping $\{\{x \leftarrow S1\}, \{x \leftarrow S2\}\}$, but it does not consider the TBox. This would be the correct result, if the TBox is empty. But the query should query all sensor names and $S3$ is a sensor name, because the TBox defines that every concept $isSensor$ is an inclusion of the role $hasVal$. In other words, every individual is a sensor with the concept $isSensor$, if there exists a measurement $hasVal$ of this sensor. Therefore the knowledge of the TBox has to be compiled into the CQ. Due to the fact that the CQ ϕ has the free variable y and this variable is not used in other parts of the CQ, the variable is replaced by a wildcard. The new CQ is $\phi(x) \leftarrow hasVal(x, \_)$. Using the concept inclusion of the TBox, the concept $isSensor(x)$ can be retrieved from the query. The new query $\phi(x) = \psi'(x) \leftarrow hasVal(x, \_) \cup \psi''(x) \leftarrow isSensor(x)$ is a union of conjunctive queries (UCQ). The first part of the UCQ has the same mapping as result as before. The second part has the mapping $\{\{x \leftarrow S3\}\}$. The union of both is $\{\{x \leftarrow S1\}, \{x \leftarrow S2\}, \{x \leftarrow S3\}\}$ and contains all available sensor names.

2.4.2 Perfect Rewriting Algorithm

The main idea of the algorithm Perfect Rewriting [4] is to generate new CQs that express the same query with different atoms. The result is an Union of Conjunctive Queries (UCQs) that is queried against the Temporal Knowledge Base. Every CQ in this UCQ represents the query with different knowledge of the TBox.

The Perfect Rewriting algorithm (Algorithm 1) takes a UCQ and return a rewritten UCQ. Every CQ is also a UCQ. The returned UCQ can be represented as TCQ, which represents a union of all CQs of the UCQ. To rewrite a CQ with a TBox, the TBox has

to be satisfiable. If the ontology is not satisfiable, the result of the CQ will be always true.

Algorithm 1 The algorithm PerfectRef of [4]

Input: UCQ q, DL-Lite$_A$ TBox T
Output: UCQ pr

```
pr := q;
repeat
  pr' := pr;
  for each CQ q' ∈ pr' do
   (a) for each atom g in q' do
         for each PI α in T do
           if α is applicable to g
           then pr := pr ∪ {q'[g/gr(g, α)]};
   (b) for each pair of atoms g1, g2 in q' do
         if g1 and g2 unify
         then pr := pr ∪ {anon(reduce(q', g1, g2))};
until pr' = pr;
return pr;
```

The algorithm has two main parts. The first part (a) is the generation of new CQs. It takes CQs and replace the atoms of it with new atoms that are derived from positive inclusions. The replaced version of the CQ is added to the UCQ. The UCQ contains the CQ with no modification and the modified CQ. An atom can only be replaced, if a concept inclusion is applicable to this atom. A concept inclusion α is applicable for an atom g, if there is a rule that matches the concept inclusion α and the atom g. In Table 2.1 are all rules for replacing listed.

Atom g	Positive inclusion α	$gr(g, \alpha)$
$A(x)$	$A_1 \sqsubseteq A$	$A_1(x)$
$A(x)$	$\exists P \sqsubseteq A$	$P(x, \_)$
$A(x)$	$\exists P^- \sqsubseteq A$	$P(\_, x)$
$P(x, \_)$	$A \sqsubseteq \exists P$	$A(x)$
$P(x, \_)$	$\exists P_1 \sqsubseteq \exists P$	$P_1(x, \_)$
$P(x, \_)$	$\exists P_1^- \sqsubseteq \exists P$	$P_1(\_, x)$
$P(\_, x)$	$A \sqsubseteq \exists P^-$	$A(x)$
$P(\_, x)$	$\exists P_1 \sqsubseteq \exists P^-$	$P_1(x, \_)$
$P(\_, x)$	$\exists P_1^- \sqsubseteq \exists P^-$	$P_1(\_, x)$
$P(x_1, x_2)$	$P_1 \sqsubseteq P$ or $P_1^- \sqsubseteq P^-$	$P_1(x_2, x_1)$
$P(x_1, x_2)$	$P_1 \sqsubseteq P^-$ or $P_1^- \sqsubseteq P$	$P_1(x_2, x_1)$

Table 2.1: Replacing rules of [4]

The second part (b) of the rewriting algorithm is reduction and simplification of CQs

and their atoms. For each pair of atoms in the CQ, the algorithm checks whether both atoms can be unified. In a CQ all atoms are connected by conjunction. This means that two atoms can be reduced to one atom, if both has the same predicate and the unification of the subject and object is possible. A unification of two subjects is possible, if both has the same value or one of them is a wildcard. The unification of objects is done in the same way. Given the two atoms $g_1 = hasVal(x, y)$ and $g_2 = hasVal(x, \_)$, then both atoms are nearly the same. Only g_2 has a wildcard instead of the variable y. The unification of both results in $g_3 = hasVal(x, y)$.

The anon function replaces variables with wildcards. A variable of an atom can be replaced, if the variable is not free and no other atom in this CQ uses the variable. If the atom $g_3 = hasVal(x, y)$ is used in a CQ and the variable y is not free, then the result would be $g_3' = hasVal(x, \_)$.

Both parts of the algorithm are executed for each CQ in the UCQ. This entire process is repeated until the UCQ is unchanged after the loop.

2.4.3 Execution of Perfect Rewriting

The CQ of the example above is $\psi(x) \leftarrow hasVal(x, y)$. In this subsection, the execution of the Perfect Rewriting Algorithm with this query is shown. The TBox has only one concept inclusion isSensor $\sqsubseteq \exists$ hasVal.

The TBox is satisfiable and therefore a rewriting possible. The input UCQ is $q = \psi(x) \leftarrow hasVal(x, y)$. The first step is the execution of the first part (a) of the algorithm, which checks each atom in the CQ for a replacement. No inclusion is applicable to the atom $hasVal(x, y)$, because it has two variables and the TBox has no inclusion of one role with two variables. The next step is the unification of pairs of atoms in the second part (b) of the algorithm. The atoms g_1 and g_2 can be the same. Therefore the reducing of $hasVal(x, y)$ with itself results in $hasVal(x, y)$. The anon function introduce the wildcard for y, because only x is free and y is not used elsewhere in the CQ. The UCQ is $pr = \psi_0(x) \leftarrow hasVal(x, y) \cup \psi_1(x) \leftarrow hasVal(x, \_)$ now. The UCQ is changed and triggers a second execution of the loop. This time, the atom $hasVal(x, \_)$ and the positive inclusion isSensor $\sqsubseteq \exists$ hasVal results into a application of the inclusion to the atom. The fourth rule of the Table 2.1 is used and results into $isSensor(x)$. Afterwards the UCQ is $pr = \psi_0(x) \leftarrow hasVal(x, y) \cup \psi_1(x) \leftarrow hasVal(x, \_) \cup \psi_2(x) \leftarrow isSensor(x)$. The reduction and anon function do not change this result of the UCQ, but the UCQ is changed during the execution of part (a) and (b). Therefore the loop is executed a third time. This next run results into the same UCQ and therefore the algorithm returns the UCQ.

2.5 Temporal Query Answering Algorithm

In the previous sections, the fundamentals of Temporal Query Answering were introduced. In this section, the functions of the Temporal Query Answering Algorithm are described. To answer a TCQ, each CQ in the TCQ is rewritten and afterwards integrated into the TCQ. Each UCQ can be written as a TCQ with unions of CQs.

The main idea is query answering over a stream of data. This stream of data is a sequence of A-Boxes and an ontology. Both together form a Temporal Knowledge Base. Due to the fact that the query rewriting compile the ontology into the query, only the sequence of A-Boxes is needed.

Temporal Query Answering over a sequence has three different tasks: The first task is the start of the answering at the begin of the sequence. This is done by the Initial Answer Formula. The second task is the consuming of the rest of the elements one by one. This is done by the Next Answer Formula and Update Formula. The last task is the evaluation of the result at the end of the sequence to answer the query. This task is done by the Evaluation Function. The Evaluation Function can also return the current intermediate result of a query. It can be called every time expect during the Next Answer Formula and the Update formula. Both functions have to be called at once in an atomic way.

2.5.1 Initial Answer Formula

The first main function of the algorithm is the Initial Answer Formula. This formula generates the first answer formulas for all sub-queries. The Initial Answer Formula is denoted by Φ_0, where 0 stands for the first element of the sequence of A-Boxes. An answer formula is a mapping of bounded variables to specific values. A sub-query has a set of answer formulas as solution. The Initial Answer Formula is recursively defined as:

- $\Phi_0(\psi_1) := Ans(\psi_1, J^{(0)})$ if ψ_1 is a CQ
- $\Phi_0(\psi_1 \wedge \psi_2) := \Phi_0(\psi_1) \cap \Phi_0(\psi_2)$
- $\Phi_0(\psi_1 \vee \psi_2) := \Phi_0(\psi_1) \cup \Phi_0(\psi_2)$
- $\Phi_0(\circ\psi_1) := x_0^{\circ\psi_1}$, $\Phi_0(\circ^-\psi_1) := \emptyset$
- $\Phi_0(\bullet\psi_1) := x_0^{\bullet\psi_1}$, $\Phi_0(\bullet^-\psi_1) := \Delta^{N_V}$
- $\Phi_0(\psi_1 U \psi_2) := \Phi_0(\psi_2) \cup (\Phi_0(\psi_1) \cap x_0^{\psi_1 U \psi_2})$
- $\Phi_0(\psi_1 S \psi_2) := \Phi_0(\psi_2)$

If the sub-query is a CQ, then the answer formulas for the CQ queried against the first ABox are returned. The conjunction of two TCQs is the intersection of both results of the two TCQs. Therefore it is important that sub-queries are calculated first. The disjunction is treated the same way as the conjunction. The future operators strong next ($\circ$), weak next ($\bullet$) and until (U) introducing placeholder variables of the form x_0^{op}, where 0 denotes the current point in time, which is the first ABox. The *op* denotes the operation to distinguish the different types of placeholder variables. The placeholder variables will be replaced by the the Update Formula or Evaluation Function later. The past operators strong previous ($\circ^-$), weak previous ($\bullet^-$) and since (S) using already calculated answer formulas of the previous ABox. This is the first ABox, therefore the since operator only uses the current ABox and strong previous and weak previous use

default answer formulas. The answer formula for the strong previous operator is an empty set of mappings, because no previous ABox exists. The set of answer formulas for the weak previous operator is the set of all mappings. The set of all mappings is represented as Δ^{N_V}.

2.5.2 Next Answer Formula

To calculate the answer formulas for the sub-queries of the next ABox, the Next Answer Formula is used. Φ_i^0 denotes the Next Answer Formula for a point in time i, which is in the sequence of A-Boxes, expect the first ABox. The first ABox is always only used by the Initial Answer Formula.

The Next Answer Formula is defined in nearly the same way as the Initial Answer Formula. The difference is the treatment of the previous operators. The Initial Answer Formula uses default values for strong and weak previous. The Next Answer Formula uses for both operators the already calculated answer formulas, because there is always a previous ABox. In the Initial Answer Formula, the since is calculated by the set of answer formulas of the second sub-query, because there is no previous point in time with a not empty set of answer formulas for the second sub-query. The Next Answer Formula uses the previous calculated set of answer formulas of the since operator and the answer formulas of both sub-queries to calculate the next answer formula.

The Next Answer Formula is defined as:

- $\Phi_i^0(\psi_1) := Ans(\psi_1, J^{(i)})$ if ψ_1 is a CQ
- $\Phi_i^0(\psi_1 \wedge \psi_2) := \Phi_i^0(\psi_1) \cap \Phi_i^0(\psi_2)$
- $\Phi_i^0(\psi_1 \vee \psi_2) := \Phi_i^0(\psi_1) \cup \Phi_i^0(\psi_2)$
- $\Phi_i^0(\circ\psi_1) := x_i^{\circ\psi_1}$, $\Phi_i^0(\circ^-\psi_1) := \Phi_{i-1}(\psi_1)$
- $\Phi_i^0(\bullet\psi_1) := x_i^{\bullet\psi_1}$, $\Phi_i^0(\bullet^-\psi_1) := \Phi_{i-1}(\psi_1)$
- $\Phi_i^0(\psi_1 U \psi_2) := \Phi_i^0(\psi_2) \cup (\Phi_i^0(\psi_1) \cap x_i^{\psi_1 U \psi_2})$
- $\Phi_i^0(\psi_1 S \psi_2) := \Phi_i^0(\psi_2) \cup (\Phi_i^0(\psi_1) \cap \Phi_{i-1}(\psi_1 S \psi_2))$

All operators expect the previous operators are defined the same way as in the Initial Answer Formula. In the Initial Answer Formula, the answer formulas can contain placeholder variables of the current point in time. The Next Answer Formulas uses previous answer formulas and introduces placeholder variables, too. Therefore an answer formula can contain placeholder variables of the current and the previous point in time. The placeholder variables of the previous point in time are placeholder for answer formulas of the current point in time. To replace the previous introduced placeholder variables, an Update Formula is needed.

2.5.3 Update Formula

The Update Formula replaces previous introduced placeholder variables with already calculated answer formulas. The Update Formula only replaces placeholder variables of the previous point in time. Placeholder variables of the last run of the Next Answer Formula are not replaced, because their answer formulas will be calculated in the next run of the Next Answer Formula.

The Update Formula replaces all placeholder variables with answer formulas right after their calculation. Therefore the algorithm only need to keep track of the answer formulas of the sub-queries of the current and previous point in time. All other answer formulas of the past will not be needed in the next steps to answer the query. All needed answer formulas are stored in the structure of the TCQ.

The Update Formula replaces all placeholder variables $x_{i-1}^{\psi^j}$, where $i-1$ is the previous point in time and ψ^j the sub-query. The Next Answer Formula is denoted by Φ_i^0, where i is the point in time and 0 is the set of answer formulas calculated by the Next Answer Formula. The Update Formula uses the answer formulas of the Next Answer Formula to update each placeholder variable of the previous point in time. The update of the placeholder variables uses the same order of sub-queries as the Initial or Next Answer Formula to replace all placeholder variables.

$$update(x_{i-1}^{\psi^j}) := \begin{cases} \Phi_i^{j-1}(\psi_1) & \text{if } \psi^j = \circ\psi_1 \text{ or } \psi^j = \bullet\psi_1 \\ \Phi_i^{j-1}(\psi^j) & \text{if } \psi^j = \psi_1 U \psi_2 \text{ or } \psi^j = \Box\psi_1 \text{ or } \psi^j = \Diamond\psi_1 \end{cases}$$

An answer formula for a query contains only placeholder variables of its sub-queries. The Update Formula starts with the set of answer formulas Φ_i^0. After the update of the set of answer formulas of the first sub-query, the set of all answer formulas is denoted by Φ_i^1. If there are k sub-queries, the update sequence would be $\Phi_i^0, \Phi_i^1, \ldots \Phi_i^k = \Phi_i$. The result of the Update Formula is the set of answer formulas Φ_i, which equals to the set of answer formulas of the last update.

The Update Formula replaces placeholder variables of strong next and weak next with the answer formulas for the sub-query of the query element. For the until, always and eventually operator, the placeholder variable is replaced by its set of answer formulas of the current point in time.

The algorithm queries over a Temporal Knowledge Base by executing the Initial Answer Formula followed by the loop of the Next Answer Formula and Update Formula till the end of the sequence of A-Boxes. The result of a query is Φ_n, if A_n is the last A-box in the sequence of the temporal knowledge base. The set of answer formulas Φ_n can contain placeholder variables introduced by the current point in time. To get a correct result for the query, the Evaluation Function replaces this placeholder variables.

2.5.4 Evaluation Function

The Evaluation Function replaces placeholder variables, which are present in the set of answer formulas representing the query result. In general, the result of the query at the end of the sequence of A-Boxes is interesting. To give a complete definition of the

Evaluation Function, also the evaluation of a query in the middle of a sequence is defined. The Evaluation Function is denoted by $eval^n$, where n is the length of the sequence of A-Boxes. The Evaluation Function is defined as:

- $eval^n(A) := A$ if $A \subseteq \Delta^{N_V}$
- $eval^n(x_j^\psi) := \begin{cases} Ans(\psi_1, J^{(n)}, j+1) & \text{if } j < n \text{ and } (\psi = \circ\psi_1 \text{ or } \psi = \bullet\psi_1) \\ Ans(\psi, J^{(n)}, j+1) & \text{if } j < n \text{ and } \psi = \psi_1 U \psi_2 \\ Ans(\psi, J^{(n)}, j+1) & \text{if } j < n \text{ and } (\psi = \Box\psi_1 \text{ or } \psi = \Diamond\psi_1) \\ \emptyset & \text{if } j = n \text{ and } (\psi = \circ\psi_1 \text{ or } \psi = \psi_1 U \psi_2) \\ \emptyset & \text{if } j = n \text{ and } \psi = \Diamond\psi_1 \\ \Delta^{N_V} & \text{if } j = n \text{ and } (\psi = \bullet\psi_1 \text{ or } \psi = \Box\psi_1) \end{cases}$
- $eval^n(\alpha_1 \cap \alpha_2) := eval^n(\alpha_1) \cap eval^n(\alpha_2)$
- $eval^n(\alpha_1 \cup \alpha_2) := eval^n(\alpha_1) \cup eval^n(\alpha_2)$

The evaluation of a subset A of all mappings Δ^{N_V} is the subset A, because the Evaluation Function has a set of mappings as result. Furthermore its task is to eliminate placeholder variables, intersections and unions. The evaluation of placeholder variables is treated in two different ways: The first one is the evaluation of a placeholder variable in the middle of the sequence. The answer formulas for the replacement are given in the next point in time. The Next Answer Formula at point in time $j+1$ for a sub-query ψ with the sequence of interpretations J and a sequence of A-Boxes with a length n is denoted by $Ans(\psi, J^{(n)}, j+1)$. The second one is the evaluation of a placeholder variable at the end of the sequence. The Evaluation Function uses default values for placeholder variables. The placeholder variables for the weak next and always operators are replaced by the set of all mappings Δ^{N_V}. The placeholder variables for strong next, until and eventually are replaced by the empty set of answer formulas.

The evaluation of an intersection of two sub-formulas is the intersection of the evaluations of both set of answer formulas. The union of two sub-formulas is treated the same way.

In the previous sections, the terms answer formula and mappings are used. To get a deep understanding, how answer formulas are represented and two answer formulas can be intersected or unified, the next section introduces the definition of answer formulas.

2.6 Answer Formulas

An answer formula is a set of mappings, which represents a solution for a query. A mapping is a set of bounded variables. A bounded variable is an assignment of a free variable to a specific value. Given the role assertion $hasVal(S1, 3.0)$ and the TCQ $Q(x) \leftarrow hasVal(x, 3.0)$, then a result for the query against this role assertion is a mapping of the free variables. In this example the TCQ has only the free variable x. The mapping is $\{x \leftarrow S1\}$, also called answer formula. If the TCQ is $Q(x, y) \leftarrow hasVal(x, y)$,

then the answer formula has two mappings $\{x \leftarrow S1, y \leftarrow 3.0\}$, because two variables are bounded.

Both answer formulas represent a solution for a given TCQ and the given role assertion $hasVal$. An ABox can contain more than one role assertion and there might be more than on possible mapping. Therefore every result of a query is a set of answer formulas. Given the ABox A with the role assertions $hasVal(S1, 2.0)$ and $hasVal(S2, 3.0)$, then the result for the query $Q(x, y) \leftarrow hasVal(x, y)$ is the set of answer formulas $\{\{\ x \leftarrow S1, y \leftarrow 2.0\}, \{\ x \leftarrow S2, y \leftarrow 3.0\}\}$.

A query is a boolean query, iff there are no free variables. The result of a boolean query is an empty set of answer formulas or the set of all possible answer formulas Δ^{N_V}. The answer formula $\top$ denotes an solution that contain all possible mappings.

A union of two sets of answer formulas is the union of the sets. Given the answer formulas $\{\{\ x \leftarrow S1, y \leftarrow 2.0\}, \{\ x \leftarrow S2, y \leftarrow 3.0\}\}$ and $\{\{\ x \leftarrow S3, y \leftarrow 4.0\}\}$, then the union of both sets is $\{\{\ x \leftarrow S1, y \leftarrow 2.0\}, \{\ x \leftarrow S2, y \leftarrow 3.0\}, \{\ x \leftarrow S3, y \leftarrow 4.0\}\}$. If one answer formula of both sets contains the answer formula $\top$, then the union of both is the set of all possible answer formulas $\{\top\}$, because $\top$ represents all possible answer formulas.

An intersection of two answer formulas is the set of all intersections of all combinations of answer formulas of both sets. An intersection of an answer formula with another one is empty, if one of the answer formulas is empty or both contain the same bounded variable with different values. Given the answer formulas $\{\{\ x \leftarrow S1, y \leftarrow 2.0\}, \{\ x \leftarrow S2, y \leftarrow 3.0\}\}$ and $\{\{\ x \leftarrow S3, y \leftarrow 4.0\}\}$, then the intersection of both sets is $\emptyset$, because the combination of each answer formula contains different values for x and y. Given the answer formulas $\{\{\ x \leftarrow S1\}, \{\ x \leftarrow S2\}\}$ and $\{\{\ x \leftarrow S1, y \leftarrow 2.0\}\}$, then the intersection of both is the set of answer formulas $\{\{\ x \leftarrow S1, y \leftarrow 2.0\}\}$, because there is an answer formula in both sets with the same values for the same bounded variables. In this case the bounded variable x has the same value in both answer formulas and the mapping of the bounded variable y is only present in one answer formula. If one of the sets contains an answer formula $\top$, then the result of the intersection is the other set. A special case is given, if both sets contain an answer formula $\top$, then the result of the intersection is $\{\top\}$.

2.7 Infinite Sequences

In Temporal Conjunctive Query Answering, the example of a Temporal Knowledge Base with a finite sequence of A-Boxes was shown. In this section, the query answering against an infinite sequence of A-Boxes will be explained.

To answer a query against an finite sequence of A-Boxes, the Initial Answer Formula for the first ABox and both next step functions, the Next Answer Formula and Update Formula as loop till the end of the sequence, are called. The result of the query is evaluated with the Evaluation Function at the end of the sequence. In the semantic context the result is the query queried against the last point in time. For infinite sequences of A-Boxes, there is no final end of the sequence. Therefore all consumed A-Boxes are treated

$ABox_0$	$ABox_1$	$ABox_2$	$ABox_3$	$ABox_4$	...
`hasVal(S1,0.0)`	`hasVal(S1,1.0)`	`hasVal(S1,2.5)`	`hasVal(S1,3.0)`	`hasVal(S1,4.5)`	

Figure 2.4: Infinite sequence of A-Boxes

as finite sequence. For every new ABox the Next Answer Formula and the Update Formula are called. If the Evaluation Formula is called, it takes the answer formulas of the entire query and replaces the placeholder variables. The original answer formulas stay the same to use them for future A-Boxes in terms of the Next Answer Formula. The result is the query over a finite sequence. At every new ABox, this sequence is extended. Due to the placeholder variables, the algorithm only need to touch the new A-Boxes. Every ABox is only needed once.

The semantics of an infinite sequence treated as a finite sequence is changing over time by extending the sequence. The semantic of the past operators stays the same, but the future operators change, because there is new information in terms of new A-Boxes. Given a sequence with the length n and a TCQ $Q(x) \leftarrow \circ hasVal(x, y)$, then the result is $\emptyset$. If the same TCQ is queried against the same position after the extension, the sequence would have the length $n + 1$ and may be there is an ABox with the role assertion $hasVal$ that results into a not empty set as result.

Algorithm 2 The algorithm of Temporal Query Answering

Input: A TCQ ϕ and an infinite sequence $J = (I_i)_{i \geq 0}$ of interpretations
Output: $Ans(\phi, J^{(i)})$ for $i \geq 0$

```
for i ← 0, 1 ... do
  if i=0 then
    compute Φ_0;                                  // Initial Answer Formula
  else
    compute Φ_i^0 from Φ_{i−1};                   // Next Answer Formula
    compute Φ_i^1, ..., Φ_i^k = Φ_i;              // Update Formula
  end if
  output eval^i(Φ_i(φ));                          // Evaluation Function
end for
```

Therefore, the query is always evaluated at the end of the sequence and the results are temporal, because future information can change the result, if they use future operators that operates over the end of the sequence.

The Algorithm 2 implements Temporal Query Answering over an infinite sequence and returns a stream of results. For each ABox in the sequence, it returns a set of answer formulas. Given the TCQ $Q(x) \leftarrow hasVal(S1, x)$ and a Temporal Knowledge Base with an empty ontology and the infinite sequence of A-Boxes in Figure 2.4, then the result of the TCQ Q is the infinite sequence of sets of answer formulas:

$$\{\{x \leftarrow 0.0\}\}, \{\{x \leftarrow 1.0\}\}, \{\{x \leftarrow 2.5\}\}, \{\{x \leftarrow 3.0\}\}, \{\{x \leftarrow 4.5\}\}, \ldots$$

3 Sliding Window

In Temporal Query Answering, the current status of the system is important. The queries for analysing the status of the system should only query with a limited history. Only the current status in terms of a limited history like a hour is interesting.

A limited history can be achieved by a window, representing a sub-sequence of A-Boxes with a fix size. At every extension of the sequence of A-Boxes, the window can slide one ABox by adding the new ABox and dropping the oldest one. This idea is called sliding window, where only the A-Boxes of the window are queried. A sliding window has two parameters. The first one is the range. The range defines the length or size of the window. The second one is the sliding. In the algorithm, there is only a sliding of one possible. To support sliding, the algorithm evaluates only at the points in time, which are a multiple of the sliding. Furthermore the first point in time of the evaluation is reached, if the length of a complete window is consumed.

3.1 Temporal Query Answering with Sliding

Algorithm 3 Sliding Window Algorithm

Input: A TCQ ϕ and an infinite sequence $J = (I_i)_{i \geq 0}$ of interpretations
Output: $Ans(\phi, J^{(i)})$ for $i \geq 0 \wedge i \mod s = 0$

```
for i ← 0, 1 ... do
  if i=0 then
    compute Φ_0;                                  // Initial Answer Formula
  else
    compute Φ_i^0 from Φ_{i-1};                   // Next Answer Formula
    compute Φ_i^1, ..., Φ_i^k = Φ_i;              // Update Formula
  end if
  if (i + 1 − r) mod s = 0 ∧ i + 1 ≥ r then
    output eval^i(Φ_i(φ));                        // Evaluation Function
  end if
end for
```

The first step is the algorithm with a sliding s and a range r. This sliding defines the frequency in which the algorithm returns results. The Algorithm 3 describes a Temporal Query Answering with a TCQ ϕ and infinite sequence of interpretations of the Temporal Knowledge Base. It returns only a result, if the current position is at the end of a window. The next evaluation is in s steps. The first evaluation of the first window with the range r is at position $r - 1$. Therefore to evaluate the window, the position i has the condition $i + 1 \geq r$. The next evaluation is at $i + 1 - r + s$. The equation $(i + 1 - r) \mod s = 0$ defines all evaluation points that are larger or equal than $i - r$. Given a Temporal Knowledge Base with an empty TBox and the infinite

$ABox_0$	$ABox_1$	$ABox_2$	$ABox_3$	$ABox_4$	...
`hasVal(S1,3.0)`	`hasVal(S1,2.0)`	`hasVal(S1,3.5)`	`hasVal(S1,4.0)`	`hasVal(S1,4.5)`	

Figure 3.1: Infinite sequence of A-Boxes

sequence of A-Boxes in Figure 3.1, then the TCQ $Q(x) \leftarrow hasVal(S1, x)$ is queried over this sequence with a range of one and a sliding of two. A range of one is a window that contains only the current ABox. The TCQ Q only queries against the current ABox, therefore the result with or without window sliding should be the same in terms of the range parameter. The sliding parameter is two. With a sliding parameter of one, the result equals the algorithm without sliding window. In this case the sliding parameter is two, which represents that only every second result should be returned. By querying the TCQ Q against the infinite sequence, the result is the infinite sequence:

$$\{\{x \leftarrow 3.0\}\}, \{\{x \leftarrow 3.5\}\}, \{\{x \leftarrow 4.5\}\}, \ldots$$

This result is correct and represents the idea of sliding. The sliding defines the frequency of calling the Evaluation Function, it does not change the implementation of Evaluation Formula, the Update Formula, the Initial and Next Answer Formula. The Algorithm 3 implements the frequency of calling the Evaluation Function, but not the handling of the range. A query is queried over the whole sequence, not only the sliding window. To achieve a binding of the query to the sliding window, the query has to be rewritten. The next section describes the rewriting of a TCQ to handle the range correctly.

3.2 Temporal Query Answering with Range

In the last section, the TCQ $Q(x) \leftarrow hasVal(S1, x)$ is queried against a infinite sequence of A-Boxes with a sliding of two and a range of one. The sliding defines the frequency of calling the Evaluation Function. It does not change the way, how the query is queried over the sequence of A-Boxes. Therefore the range of the window is not handled. To handle the range, the query has to be rewritten to contain the information of the range.

The sliding window always ends at the current position of evaluation. The introduced placeholder variables for the future operators are evaluated the same way with the range. The only difference is the handling of past operators. The past operators are since, strong previous, weak previous, always in the past and history. They have to stop at the left border of the sliding window and uses the default value for the ABox before. Therefore after rewriting of the query, the algorithm should threat the sliding window as finite sequence and should not touch the A-Boxes outside of the window.

Given the TCQ $Q(x) \leftarrow \Diamond^{-} hasVal(S1, x)$, which is queried against an infinite sequence with a sliding window of range 2 and slide 2. The infinite sequence is shown in Figure 3.2. The sliding always starts at the end of the first window of the infinite sequence. In this case this is the point in time $i = 1$. This point is a solution for the

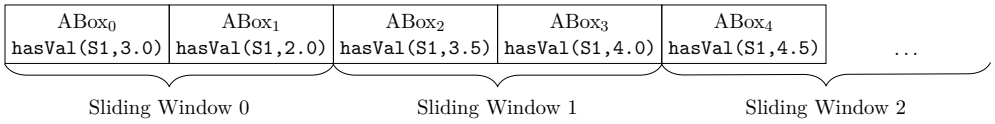


Figure 3.2: Sliding Window with range of 2 and sliding of 2

equations $i + 1 \geq r$ and $(i + 1 - r) \mod s$. The start point is always $r - 1$. Therefore the point in time of the evaluation depends on the range.

The Evaluation Function is always called for the next points, if $(i + 1 - r) \mod s$ holds. In this example the function is called at position $1, 3, 5, \ldots$.

The result of the query is a stream of sets of answer formulas. The first set of answer formulas is $\{\{x \leftarrow 3.0\}, \{x \leftarrow 2.0\}\}$, which represents the result for the sliding window 0. The result of the sliding window 1 is: $\{\{x \leftarrow 3.0\}, \{x \leftarrow 2.0\}, \{x \leftarrow 3.5\}, \{x \leftarrow 4.0\}\}$. This would be the correct result for a TCQ without sliding window, because this result contains also the first sliding window. The result contains the first sliding window, because the history operator $\Diamond^-$ is not bounded to the range. The range $r = 2$ defines the range of the sliding window. Each query should query only inside a sliding window. To achieve this, TCQs with history operators have to be rewritten. The history operator is defined as $\Diamond^-(\psi) = \psi \cup \circ^-(\Diamond^-\psi)$. To bound the history operator, the rewritten version of it uses the strong previous operator. Using the definition, the following equations can be retrieved:

$$\Diamond^-(\psi) = \psi \cup \circ^-(\psi \cup \circ^-(\psi \cup \circ^-(\psi \cup \circ^-(\ldots))))$$

The rewritten TCQ Q for a range $r = 2$ is $Q'(x) \leftarrow hasVal(S1, x) \cup \circ^-(hasVal(S1, x))$. With the new TCQ Q' the result of the second sliding window is $\{\{x \leftarrow 3.5\}, \{x \leftarrow 4.0\}\}$. For the rewriting, the function R_i is defined. i represents the position in the range. The rewriting function starts always at the position R_{r-1}, where r is the range. If the range is 2, the range is from 0 to 1. The Evaluation Function evaluates at range position 1. In this example, the equation $Q' = R_1(Q)$ holds.

The base case of the Rewriting Algorithm is the CQ. The rewriting of a CQ is also the same CQ. The definition of the rewriting function for CQs and the history operator is:

$R_i(\psi) = \psi$ if ψ is a CQ

$$R_i(\Diamond^-\psi) := \begin{cases} (\circ^- R_{i-1}(\Diamond^-\psi)) \cup R_i(\psi) & \text{if } i > 0 \\ R_0(\psi) & \text{if } i = 0 \end{cases}$$

With this two definitions, a rewriting of the TCQ Q to Q' is done. A TCQ can also contain other operators. The always in the past operator is analogously to the history operator:

$$R_i(\Box^-\psi) := \begin{cases} (\bullet^- R_{i-1}(\Box^-\psi)) \cap R_i(\psi) & \text{if } i > 0 \\ R_0(\psi) & \text{if } i = 0 \end{cases}$$

The weak previous and strong previous are past operators, that can also exit the range

and query against A-Boxes outside of the range. To limit the operations to the range, the previous operators weak and strong previous have default values for the first point in time of the range. For a strong previous R_0 the set of all possible answer formulas is Δ^{N_V} and the weak previous of R_0 is the empty set. Both definitions are as follows:

$$R_i(\circ^- \psi) := \begin{cases} \circ^- R_{i-1}(\psi) & \text{if } i > 0 \\ \Delta^{N_V} & \text{if } i = 0 \end{cases}$$

$$R_i(\bullet^- \psi) := \begin{cases} \bullet^- R_{i-1}(\psi) & \text{if } i > 0 \\ \emptyset & \text{if } i = 0 \end{cases}$$

The since operator is the last past operator. It uses also previous results and is not bounded to the range. The definition of the since operator is $\psi_1 \text{ S } \psi_2 = \psi_2 \cup (\psi_1 \cap (\psi_1 \text{ S } \psi_2))$. This operation can also represented by a sequence with the help of the strong previous operator:

$$\psi_1 \text{ S } \psi_2 = \psi_2 \cup (\psi_1 \cap \circ^-(\psi_2 \cup (\psi_1 \cap \circ^-(\psi_2 \cup (\psi_1 \cap \circ^-(\dots))))))$$

This rewriting of the since operator is recursively done by the Rewriting Algorithm as follows:

$$R_i(\psi_1 \text{ S } \psi_2) := \begin{cases} (\circ^-(R_{i-1}(\psi_1 \text{ S } \psi_2)) \cap R_i(\psi_1)) \cup R_i(\psi_2) & \text{if } i > 0 \\ R_i(\psi_2) & \text{if } i = 0 \end{cases}$$

With the Rewriting Algorithm an exact position of the query elements in terms of the range is introduced. The rewritten query defines exactly at which position of the range, the CQs are evaluated. With this knowledge the rewritten TCQ returns default values for CQs that are queried outside of the range. Every TCQ with past operators and CQs can be rewritten to such a TCQ.

Given the TCQ $Q(x) \leftarrow \circ^-(\Box(\Diamond^- hasVal(S1, x)))$ and the sequence defined above, then the strong previous and history operator can be rewritten by the rewriting function. If the always operator ($\Box$) stays the same, the postion of the history operator can be 0 or 1. To get an exact position, the always operator has to be transformed into a query that uses the weak next operator. With this step, the new expression for the always operator has for each position of the range an sub-query. This means that there are two sub-queries for the history operator. One starts at the position 0 and one at the position 1. It implies that future operators have to be rewritten. The definitions for the future operators are:

$$R_i(\circ \psi) := \begin{cases} \circ R_{i+1}(\psi) & \text{if } i < r \\ \Delta^{N_V} & \text{if } i = r \end{cases}$$

$$R_i(\bullet \psi) := \begin{cases} \bullet R_{i+1}(\psi) & \text{if } i < r \\ \emptyset & \text{if } i = r \end{cases}$$

$$R_i(\Box \psi) := \begin{cases} (\bullet R_{i+1}(\Box \psi)) \cap R_i(\psi) & \text{if } i < r \\ R_s(\psi) & \text{if } i = r \end{cases}$$

$$R_i(\Diamond\psi) := \begin{cases} (\circ R_{i+1}(\Diamond\psi)) \cup R_i(\psi) & \text{if } i < r \\ R_s(\psi) & \text{if } i = r \end{cases}$$

$$R_i(\psi_1 \text{ U } \psi_2) := \begin{cases} R_i(\psi_2) \cup (R_i(\psi_1) \cap \circ R_{i+1}(\psi_1 \text{ U } \psi_2)) & \text{if } i < r \\ R_i(\psi_2) & \text{if } i = r \end{cases}$$

The rewriting of the conjunction and disjunction operators are:

$$R_i(\psi_1 \wedge \psi_2) = R_i(\psi_1) \wedge R_i(\psi_2)$$
$$R_i(\psi_1 \vee \psi_2) = R_i(\psi_1) \vee R_i(\psi_2)$$

With all given rewriting definitions, the rewriting of the TCQ $Q(x) \leftarrow \circ^-(\Box(\Diamond^- hasVal(S1, x)))$ has the following steps:

$$R_1(Q)$$
$$= R_1(\circ^-(\Box(\Diamond^- hasVal(S1, x))))$$
$$= \circ^-(R_0(\Box(\Diamond^- hasVal(S1, x))))$$
$$= \circ^-(R_0(\Diamond^- hasVal(S1, x)) \cap (\bullet R_1(\Box(\Diamond^- hasVal(S1, x)))))$$
$$= \circ^-(R_0(hasVal(S1, x)) \cap (\bullet R_1(\Diamond^- hasVal(S1, x)))$$
$$= \circ^-(hasVal(S1, x) \cap (\bullet(\circ^- R_0(\Diamond^- hasVal(S1, x)) \cup R_1(hasVal(S1, x)))))$$
$$= \circ^-(hasVal(S1, x) \cap (\bullet(\circ^- R_0(hasVal(S1, x)) \cup R_1(hasVal(S1, x)))))$$
$$= \circ^-(hasVal(S1, x) \cap (\bullet(\circ^- hasVal(S1, x) \cup hasVal(S1, x))))$$

The rewritten TCQ is queried against an infinite sequence with the Algorithm 3. This algorithm with the rewritten TCQ represents Temporal Query Answering with a sliding window. The algorithm contains the sliding information and the rewritten TCQ contains the range information. The syntax and semantics are the same, if there is a finite sequence of A-Boxes representing the sliding window.

3.3 Temporal Query Answering with Sliding Window

To query a TCQ against an Temporal Knowledge Base with an infinite sequence, the first step is the rewriting of the TCQ to include the ontology into the TCQ. This is done by the Perfect Rewriting Algorithm (2.4.2). The next step is the second rewriting of the TCQ to bound the query to the range of the sliding window. This is done by the Rewriting Algorithm (3.2). The last step is to run the Algorithm (3.1), which calls the Evaluation Function in the given frequency.

An advantage of the rewriting algorithm to establish a sliding window is the unchanged definition of the original functions. The Initial Answer Formula, the Next Answer Formula, the Update Formula and the Evaluation Formula stayed untouched. Only the TCQ

and the frequency of calling the Evaluation Function is changed. As before, the algorithm consumes an ABox a time and only has to keep the sets of answer formulas of the current and the previous point in time. On the other hand, the structure of the TCQ stores the intermediate sets of answer formulas. By rewriting the query in terms of eliminating operators by replacing them with previous and next operations, the rewritten TCQs have more operations. Each temporal operator saves its own set of answer formulas. That means the performing of querying with sliding window needs more memory space. Given a sliding window with a range of 1000 and the TCQ $Q(x) \leftarrow \Diamond^{-} hasVal(S1, x)$, then the rewritten TCQ would contain 999 strong previous operators. Each of them has its own set of answer formulas. The original algorithm without window sliding has only one set of answer formulas for the history operator. Furthermore there are more sub-formulas, the Next Answer Formula and Update Formula have to treat every time.

3.4 Reducing the Overhead

The previous algorithm is well for calculating results for a sliding window with slow sliding value. Given a Temporal Knowledge Base with an infinite sequence, the TCQ $Q(x) \leftarrow hasVal(S1, x)$ and the sliding window has the range 1 and a sliding value of 1000. Then, the Evaluation Function evaluates the sets of answer formulas at the positions $0, 1000, 2000, \ldots$. The TCQ Q only uses the current point in time, therefore the query should only query against the points in time, the Evaluation Function evaluates. Given the example above, for the first three results, the query touches 2001 A-Boxes and queries the TCQ Q against each of them. Each query against an ABox results into a set of answer formulas. This operation produces a lot of memory input, which can be reduced by ignoring all A-Boxes that are outside the range. Therefore the overhead of calculating intermediate results that are not used in the evaluation process should be reduced.

The Algorithm 4 reduces the overhead by skipping needless A-Boxes. It has two main parts: The first part has no skipping, because the range of the sliding window is greater or equal than the sliding parameter. That means that every ABox of the infinite sequence is part of a sliding window. Furthermore there is overlapping if $r \neq s$. For this case the part one has the same implementation as the algorithm above.

If the sliding parameter is greater than the range, there are A-Boxes, which are no members of a sliding window. To calculate these A-Boxes is an overhead. To reduce the overhead, the A-Boxes are skipped by not calculating the Next Answer Formula and the Update Formula. At the beginning of a new sliding window, the Initial Answer Formulas has to be called, because the previous ABox was skipped and no previous set of answer formulas exists. The algorithm defines this skipping in the second part. The first ABox of a sliding window with the point in time i solves the equation $i \mod s = 0$. The last ABox of the window always solves the equation $i + 1 - r \mod s \wedge i + 1 \geq r$. The indexes of the other A-Boxes in the sliding window are a multiple of the indexes i in the range $s - r + 2 \leq i < s - 1$.

Both parts of the algorithm evaluate the result at the end of the sliding window. The

last ABox of the sliding windows always has an index of the multiple of s with an shift of the range. The second part of the algorithm is a special case of the first part. It uses the fact that A-Boxes can be skipped. The results of the second part are always the same to the calculated one of the first part. The first part uses the rewritten TCQ. The second part can use the original or the rewritten TCQ, because it always calls the Initial Answer Formula at the beginning of a window.

Algorithm 4 The algorithm with sliding window and ABox skipping

Input: A TCQ ϕ and an infinite sequence $J = (I_i)_{i \geq 0}$ of interpretations
Output: $Ans(\phi, J^{(i)})$ for $i \geq 0 \wedge i \mod s = 0$

```
if s ≤ r then
  for i ← 0, 1 ... do
    if i=0 then
      compute Φ_0;                                   // Initial Answer Formula
    else
      compute Φ_i^0 from Φ_{i−1};                    // Next Answer Formula
      compute Φ_i^1, ..., Φ_i^k = Φ_i;               // Update Formula
    end if
    if (i + 1 − r) mod s = 0 ∧ i + 1 ≥ r then
      output eval^i(Φ_i(φ));                         // Evaluation Function
    end if
  end for
else
  for i ← 0, 1 ... do
    if i mod s = 0) then
      compute Φ_i;                                   // Initial Answer Formula
    else
      if 1 ≤ i mod s < r then
        compute Φ_i^0 from Φ_{i−1};                  // Next Answer Formula
        compute Φ_i^1, ..., Φ_i^k = Φ_i;             // Update Formula
      end if
    end if
    if (i + 1 − r) mod s = 0 ∧ i + 1 ≥ r then
      output eval^i(Φ_i(φ));                         // Evaluation Function
    end if
  end for
end if
```

4 The Vector Algorithm

In the previous chapter, the algorithm with a sliding window by rewriting the TCQ and changing the frequency of calling the Evaluation Function was shown. As mentioned before, the rewriting of the TCQ always results into a TCQ that has a larger or equal number of operators. This has influence to the performance. In this chapter a new algorithm is presented that reduces the computation by optimisation of the operations and the calculation of intermediate results by introducing caching and elimination of needless intermediate results.

4.1 The needless Computations

The previous algorithm (Algorithm 4) skips all A-Boxes that are outside of all sliding windows and are not important to calculate the result. In this section, an example shows that there are a few more calculations that are needless.

Given a sliding window with the range 5, then the TCQ

$$Q(x, y, z) \leftarrow (\Diamond^{-} hasVal(S1, x)) \wedge hasVal(S2, y) \wedge hasVal(S3, z)$$

would produce intermediate results that are not used by the Evaluation Function at the last point in time of the sliding window. The sub-queries of Q are the following ones:

- $\psi_0 = hasVal(S1, x)$
- $\psi_1 = \Diamond^{-} \psi_0$
- $\psi_2 = hasVal(S2, y)$
- $\psi_3 = hasVal(S3, z)$
- $\psi_4 = \psi_2 \cap \psi_3$
- $\psi_5 = \psi_1 \cap \psi_4$

The Figure 4.1 shows that a lot of intermediate results are not used. There are 6 sub-queries and the range of the sliding window is 5. Therefore, the algorithm calculates 30 results but only 14 results are needed. The green nodes represent the needed intermediate results to answer the TCQ at the end of the sliding window. The red ones are needless sets of answer formulas, because they are not used by the last point in time. Each row represents a sub-query and each column represents the set of answer formulas for a given ABox.

In this chapter the main idea is that the range of sliding windows are small enough such that the entire finite sequence of a sliding window is handled at once. In other words, the algorithm starts to calculate the result for a sliding window, if the entire

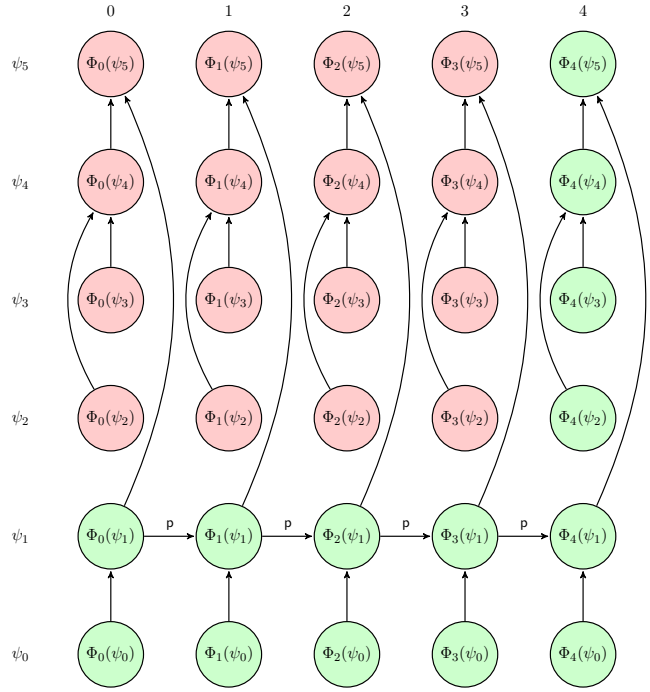


Figure 4.1: Intermediate results for the TCQ Q(x,y,z)

ABox_0	ABox_1	ABox_2	ABox_3	ABox_4
`hasVal(S1,3.0)`	`hasVal(S1,2.0)`	`hasVal(S1,3.5)`	`hasVal(S1,4.0)`	`hasVal(S1,4.5)`

Figure 4.2: Finite sequence of A-Boxes of a window with range 5

sliding window is present. With the knowledge of the range of a sliding window, a TCQ can be rewritten to calculate only the needed intermediate results. The next section introduces the Vector Algorithm that redefines a TCQ to handle a finite sequence at once.

4.2 The Vector Algorithm

The Vector Algorithm is based on the fact that the range and the sliding of the sliding window is known. It queries the TCQ as vector expression against the complete sliding window at once and calculates the results of it.

A first element of a sliding windows has the index p and the last element of the sliding window the index q. A sub-sequence of the sliding window is denoted by j and k, where j represents the start element of the sub-sequence and k the end.

Let v_1 be a CQ, then $V_{j,k}(v_1) = \{\Phi_i(v_1)\}_{j \leq i \leq k}$ denotes a sub-sequence of sets of answer formulas that are results of querying the CQ v_1 against the sub-sequence of the sliding window. Let be given the finite sequence of A-Boxes in Figure 4.2, then the result of the

vector query $V_{1,3}(hasVal(S1, x))$ is the following sequence of sets of answer formulas:

$$\{\{x \leftarrow 2.0\}\}, \{\{x \leftarrow 3.5\}\}, \{\{x \leftarrow 4.0\}\}$$

The conjunction of two vectors is defined as $V_{j,k}(A \wedge B) = \{A_i \cap B_i\}_{0 \leq i \leq k-j}$. A and B represent sequences of sets of answer formulas calculated by other vector operators. The variables j and k defines the sub-sequence of the sliding window, for which this sequence is a result. The disjunction is defined as $V_{j,k}(A \vee B) = \{A_i \cup B_i\}_{0 \leq i \leq k-j}$ analogously. By definition, both sets A and B always have the same length.

The history operator is defined as $V_{j,k}(\Diamond^- A) = \bigcup A_i$, where $j = k$. It gets a sequence of sets of answer formulas and build an union of all sets. With this 3 definitions, the previous example with the TCQ

$$Q(x, y, z) \leftarrow (\Diamond^- hasVal(S1, x)) \wedge hasVal(S2, y) \wedge hasVal(S3, z)$$

can be rewritten to the following vector query:

$$Q(x, y, z) \leftarrow V_{4,4}(\psi_0 \wedge \psi_1)$$

where

$$\psi_0 = V_{4,4}(\Diamond^- V_{0,4}(hasVal(S1, x)))$$

$$\psi_1 = V_{4,4}(V_{4,4}(hasVal(S2, y) \wedge V_{4,4}(hasVal(S3, z))$$

This new query calculates only needs results to answer the query against the sliding window. All green nodes of the Figure 4.1 are calculated and all red nodes are left out. The history operator can only be called at a subsequence with range of one. The next step is to define all operators of the vector algorithm and to provide a rewriting algorithm to transform a TCQ into a vector query.

The definition for the conjunction, disjunction and the querying of CQs stays the same. The history operator and all other operators are defined to deal with ranges larger or equal than one. The vector operators are defined as:

- $V_{j,k}(v_1) = \{\Phi_i(v_1)\}_{j \leq i \leq k}$ if v_1 is a CQ
- $V_{j,k}(A \wedge B) = \{A_i \cap B_i\}_{0 \leq i \leq k-j}$
- $V_{j,k}(A \vee B) = \{A_i \cup B_i\}_{0 \leq i \leq k-j}$
- $V_{j,k}(\circ_1 A) = A$
- $V_{j,k}(\circ_2 A) = A, \emptyset$
- $V_{j,k}(\bullet_1 A) = A$
- $V_{j,k}(\bullet_2 A) = A, \top$
- $V_{j,k}(\circ_1^- A) = A$

- $V_{j,k}(\circ_2^- A) = \emptyset, A$
- $V_{j,k}(\bullet_1^- A) = A$
- $V_{j,k}(\bullet_2^- A) = \top, A$
- $V_{j,k}(\Box A) = \{\bigcap_{i \geq h} A_i\}_{0 \leq h \leq k-j+1}$
- $V_{j,k}(\Box^- A) = \{\bigcap_{i \leq h} A_i\}_{0 \leq h \leq k-j+1}$
- $V_{j,k}(\Diamond A) = \{\bigcup_{i \geq h} A_i\}_{0 \leq h \leq k-j+1}$
- $V_{j,k}(\Diamond^- A) = \{\bigcup_{i \leq h} A_i\}_{0 \leq h \leq k-j+1}$
- $V_{j,k}(A \text{ S } B) = \{S_i\}_{j \leq i \leq k}$ with $S_0 = B_0, S_i = B_i \cup (A_i \cap S_{i-1})$
- $V_{j,k}(A \text{ U } B) = \{U_i\}_{0 \leq i \leq k-j+1}$ with $U_{q-j+1} = B_{q-j+1}, U_i = B_i \cup (A_i \cap U_{i+1})$

The next and previous operators have two different versions in each case. The first version represents the temporal move, where each element is in the sliding window. The second version is the case, that one element is outside the sliding window and therefore a default value is added to the sequence of sets of answer formulas. The strong next operator $V_{j,k}(\circ_1 A) = A$ with version one returns the sequence. It does not change the sequence A. Only the indexes j and k are different to the sub-query, because a strong next moves the sub range of j, k by on to the right. If k is equal to the last point in time q of the sliding window, there will be one index outside of the sliding window. Therefore the second version is used. It uses the empty set as a result for the none existing ABox. The weak next, strong previous and weak previous operators are treated analogously.

The always operator $V_{j,k}(\Box A)$ gets as input the sub-sequence of sets of answer formulas A. This sequence has the range (j, q), because the always operator is evaluated till the end of the sequence. The task for setting the sub-query to this range is done by the Vector Rewriting Algorithm, presented in the next section. To calculate the sub-sequence of sets of answer formulas representing the results for the always operator, the Algorithm 5 that only needs $q - j$ intersections is used.

Algorithm 5 The algorithm for the always operator

Input: A sequence of sets of answer formulas A, j, k, q
Output: A sequence of sets of answer formulas R

```
R = A_q
for i ← q − 1, . . . , k do
    R = A_i ∩ R
end for
R = {R}
for i ← k − 1, . . . , 0 do
    R = A_i ∩ R_0, R
end for
```

The result of the Algorithm 5 is a sequence of sets of answer formulas. This sequence has a length of $k - j + 1$. A result in terms of a set of answer formulas for an always operator at a specific point in time is the current set of answer formulas of the sub-query intersected with the set of answer formulas of itself at the next point in time. By calculating the next point in time at first and using this result to calculate the current result, only one intersection for every next point in time is used.

The algorithm for the operators history, eventually and always in the past are to be implemented analogously. All of them only need $n - 1$ intersections or unions, where n is the length of the sub-sequence. The since and until operations also need only $n - 1$ steps, but each step needs an intersection and one union.

The Vector Algorithm calculates over finite sequence of A-Boxes. The sequence is received at once and therefore no placeholder variables are used. Furthermore the algorithm knows which sub-sequence of the sliding window should be queried by a given TCQ. The operators of the vector algorithm have a range (j, k), but this is only important for the case that a CQ is queried against the sliding window. The other operators can check the length of the incoming sequences of sets of answer formulas and can perform their calculations without the range (j, k).

4.3 The Vector Rewriting Algorithm

The Vector Rewriting Algorithm (R^V) rewrites a TCQ into a new form that can be processed by the Vector Algorithm. The main idea is to specify the needed sub-sequences of the sliding window and only calculate the results of this sub-sequences. The Vector Rewriting Algorithm starts at the last point in time of the sliding window. Let Q be an TCQ, then the rewriting algorithm starts with $R^V_{j,k}(Q)$, where j and k are the range boundaries for the sub-sequence. The TCQ is always evaluated at the end of the sliding window, therefore $j = q$ and $k = q$. q is the last index of the sliding window and p is always the first A-Box. The first ABox always has index 0 in an finite sequence and therefore p has always the value 0.

The Vector Rewriting Algorithm is called recursively. Each operator can move or resize the sub-sequence j, k. The operator strong next moves the queried sub-sequence one to the right. This means it moves each point in time one forward to the next point in time. If the last point in time of the sub-sequence is already at the end of the sliding window, the sub-sequence is minimised by 1.

The always operator always queries till the end of the sliding window, therefore the sub-query always has an sub-sequence, which ends at the end of the sliding window.

The vector rewriting algorithm has the following definitions:

- $R^V_{j,k}(v_1) = V_{j,k}(v_1)$ if v_1 is a CQ
- $R^V_{j,k}(v_1 \wedge v_2) = V_{j,k}(R^V_{j,k}(v_1) \wedge R^V_{j,k}(v_2))$
- $R^V_{j,k}(v_1 \vee v_2) = V_{j,k}(R^V_{j,k}(v_1) \vee R^V_{j,k}(v_2))$

- $R^V_{j,k}(\circ v_1) = \begin{cases} V_{j,k}(\circ_1 R^V_{j+1,k+1}(v_1)) & \text{if } k < q \\ V_{j,k}(\circ_2 R^V_{j+1,k}(v_1)) & \text{if } j < k \wedge k = q \\ \{\emptyset\} & \text{otherwise} \end{cases}$
- $R^V_{j,k}(\bullet v_1) = \begin{cases} V_{j,k}(\bullet_1 R^V_{j+1,k+1}(v_1)) & \text{if } k < q \\ V_{j,k}(\bullet_2 R^V_{j+1,k}(v_1)) & \text{if } j < k \wedge k = q \\ \{\top\} & \text{otherwise} \end{cases}$
- $R^V_{j,k}(\circ^- v_1) = \begin{cases} V_{j,k}(\circ^-_1 R^V_{j-1,k-1}(v_1)) & \text{if } p < j \\ V_{j,k}(\circ^-_2 R^V_{j,k-1}(v_1)) & \text{if } j < k \wedge p = j \\ \{\emptyset\} & \text{otherwise} \end{cases}$
- $R^V_{j,k}(\bullet^- v_1) = \begin{cases} V_{j,k}(\bullet^-_1 R^V_{j-1,k-1}(v_1)) & \text{if } p < j \\ V_{j,k}(\bullet^-_2 R^V_{j,k-1}(v_1)) & \text{if } j < k \wedge p = j \\ \{\top\} & \text{otherwise} \end{cases}$
- $R^V_{j,k}(\Box v) = V_{j,k}(\Box R^V_{j,q}(v))$
- $R^V_{j,k}(\Diamond v) = V_{j,k}(\Diamond R^V_{j,q}(v))$
- $R^V_{j,k}(\Box^- v) = V_{j,k}(\Box^- R^V_{p,k}(v))$
- $R^V_{j,k}(\Diamond^- v) = V_{j,k}(\Diamond^- R^V_{p,k}(v))$
- $R^V_{j,k}(v_1 \text{ U } v_2) = V_{j,k}(R^V_{j,q}(v_1) \text{ U } R^V_{j,q}(v_2))$
- $R^V_{j,k}(v_1 \text{ S } v_2) = V_{j,k}(R^V_{p,k}(v_1) \text{ S } R^V_{p,k}(v_2))$

The next and previous operators have a third case in the Vector Rewriting Algorithm. They have a default value as result, if the entire sub-sequence is outside of the sliding window. For the other operators, the Vector Rewriting Algorithm only adjusts the ranges of the sub-queries. In the next chapter, this algorithm will be improved in terms of overlapping.

5 Optimising the Vector Algorithm

The previous presented Vector Algorithm gets the entire window as input. The input is a infinite sequence of batches. Each batch represents a window as sequence of A-Boxes. It calculates the result for a given query of the Vector Algorithm by calculating all sub-queries. For the next window, it starts from the beginning to calculate the result.

If the two windows are overlapping, the subsequence of A-Boxes of the overlapping is processed twice. At the first window, it is located at the end of the window and in the second window, the same sequence is located at the beginning of the window. By processing the sliding window, a reuse of already calculated answers for the overlapping part is useful. The intermediate results of the first window of the last part can be used in the first part of the second window.

To achieve a reusing of the overlapping part, the Vector Rewriting Algorithm has to calculate the number of reused results for each operator. Furthermore each operator has to store its results to reuse them as needed. The storage is a finite buffer, which handles new elements as first in first out principle.

The vector operator $V_{j,k}(v_1)$ has the following definition:

$$V_{j,k}(v_1) = \{\Phi_i(v_1)\}_{j \leq i \leq k} \text{ if } v_1 \text{ is a CQ}$$

It always returns the set of answer formulas for the entire subsequence with the range (j, k). Given a sliding window (Figure 5.1) with range $r = 2$ and a sliding of one over an infinite sequence of A-Boxes. For the first sliding window the vector operator $V_{0,1}(v_1)$ has to return the sequence of sets of answer formulas for ABox_0 and ABox_1. The idea is to return only new sets of answer formulas for the sliding window 1. In this case the vector operator returns the sequence of sets of answer formulas for ABox_2. The result for ABox_1 is already known for the next operator that receive the results.

To establish the behaviour of the vector operator for CQs, the new algorithm has two parts: The first function is the Initial Function. This function calculates the sequence of answer formulas for the entire range (j, k). The second part is the Next Function. This function calculates only a sub-sequence of the range (j, k) that contains only sets of answer formulas for the none overlapping part of the sliding window in terms of the previous window.

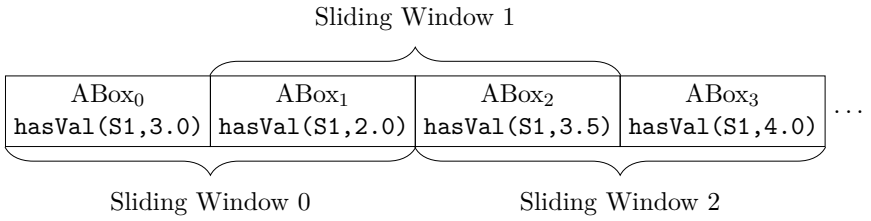


Figure 5.1: Sliding Window with a range of 2 and a sliding of 1

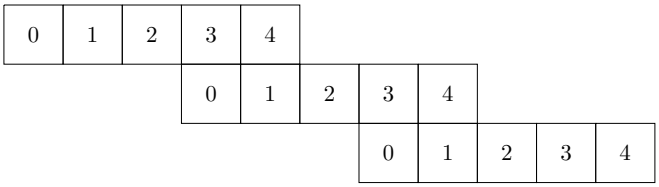


Figure 5.2: Sliding window with range $r = 5$ and a sliding of 3

The notion $V^I_{j,k,u}(v_1)$ defines a vector operator, where (j, k) is the range, I denotes the Initial Function and u is the overlapping parameter. The overlapping parameter defines the number of overlapping A-Boxes between two subsequences of the window. The optimised vector operator for CQs is defined as:

- $V^I_{j,k,u}(v_1) = \{\Phi_i(v_1)\}_{j \leq i \leq k}$

- $V^N_{j,k,u}(v_1) = \{\Phi_i(v_1)\}_{j+u \leq i \leq k}$

The Next Function $V^N_{j,k,u}$ only calculates results for the none overlapping part of the subsequence. Given a sequence and a sliding window with the range $r = 5$ and a sliding of $s = 3$, then the following Figure 5.2 is an example for the first three windows. Each window has the range $r = 5$. The sliding of $s = 3$ is less than the range, therefore overlapping exists. Given the vector operation $V_{j,k,u}$ of above and the query needs the result for the entire window, then the Initial Function returns the set of answer formulas for the complete first window. That is the range $(0, 4)$. The Next Function only returns the sets of answer formulas for the subsequence with the range $(2, 4)$, because the receiver already knows the sets of answer formulas for the points in time 0 and 1. They are equal to the previous window points 3 and 4. In this case the overlapping parameter is $u = 2$. The equation $u = \max\{0, \min\{(r - s) - (q - (k - j)), (r - s) - j\}\}$ with $r > s$ calculates the overlapping parameter for the vector operator of CQs. The overlapping parameter in this example is 2.

Given a TCQ $Q(x) \leftarrow \circ^- hasVal(S1, x)$ and the sliding window above, then the query only uses the ABox_3 of the windows. The vector operator for the $hasVal$ role has the form $V^I_{3,3,0}(hasVal(S1, x)$ with $j = 3, k = 3$ and $u = 0$, because $u = \max\{0, \min\{(5 - 3) - (4 - (3 - 3)), (5 - 3) - 3\}\} = 0$.

Given a query that uses the A-Boxes 0 till 3, then the overlapping parameter is $u = 1$, because the result of the ABox_3 can be reused in the next window as result for ABox_0. If the query touches the ABox 1 till 3, then there is no overlapping and $u = 0$ holds.

The main idea of the Optimised Vector Algorithm is to introduce intermediate results. In the same way as querying CQs against A-Boxes and reusing their sets of answer formulas to reduce the calculations for the next window, the operators can store the results for a sub-sequence and reuse them at the next window. The next section presents the complete Optimised Vector Algorithm that uses intermediate results to reduce the overhead in terms of overlapping.

5.1 The Optimised Vector Algorithm

The Optimised Vector Algorithm tries to reduce the calculations of sets of answer formulas in terms of union and intersection. It has two parts. The first part is the Initial Function. This function is always called for the first window. The second part is the Next Function. It evaluates all other windows and uses the previous calculated intermediate results to calculate its results.

5.1.1 Initial Function

The vector operator for CQs in the Initial Function is the same as in the previous vector algorithm. The definition is as follows:

- $V^I_{j,k,u}(v_1) = \{\Phi_i(v_1)\}_{j \leq i \leq k}$

The conjunction and disjunction use a storage S to store their result and pass it afterwards. The storage S is a sequence of sets of answer formulas. It has the finite length $k - j + 1$. The conjunction and disjunction operator for the Initial Function are defined as:

- $V^I_{j,k,u}(A \wedge B) = S$ where $S = \{S(A)_i \cap S(B)_i\}$ and $S(A) = A, S(B) = B$
- $V^I_{j,k,u}(A \vee B) = S$ where $S = \{S(A)_i \cup S(B_i)\}$ and $S(A) = A, S(B) = B$

To reduce the overhead and calculations the algorithm introduces different definitions for the same operators. Each definition is for a special case. Each previous and next operator has two special cases: The previous and next operators are moving the subsequence of the window by one. The first case handles that the new subsequence is also in the boundaries of the sliding window. The second case introduces a default set of answer formulas for the point in time that is not in the boundaries of the window anymore.

The first case forwards the sets of answer formulas. The second case introduce a storage to store the complete subsequence and returns the subsequence with the default set of answer formulas. It has to store the sequence, because at each step the result of the second case is a complete new sequence of answer formulas, due to the default set of answer formulas. A reusing is not possible by using the a buffer with first in first out principle. Therefore the overlapping parameter of this operators is always 0.

- $V^I_{j,k,u}(\circ_1 A) = A$
- $V^I_{j,k,0}(\circ_2 A) = S, \emptyset$ where $S = A$
- $V^I_{j,k,u}(\bullet_1 A) = A$
- $V^I_{j,k,0}(\bullet_2 A) = S, \top$ where $S = A$
- $V^I_{j,k,u}(\circ_1^- A) = A$
- $V^I_{j,k,0}(\circ_2^- A) = \emptyset, S$ where $S = A$

- $V^I_{j,k,u}(\bullet^-_1 A) = A$
- $V^I_{j,k,0}(\bullet^-_2 A) = \top, S$ where $S = A$

The always operator has three cases: The first case is the general case. It is used to calculate the results for always operators, which have a range larger than one or have a sub-query that has an overlapping parameter equals zero. The second case deals with always operators, which have a range of one and the range of the sub-query has a minimum size of twice this overlapping parameter. The third case is for always operators that have a range of one and the range of the sub-query is less than the twice of the overlapping parameter. The cases are defined as:

- $V^I_{j,k,0}(\Box_1 A) = \{\bigcap_{i \geq h} S_i\}_{j \leq h \leq k}$ where $S = A$
- $V^I_{j',j',0}(\Box_2 V_{j,k,u}(A)) = \bigcap S_i$ where $S = \bigcap A_{0 \leq i < u}, \bigcap A_{u \leq i \leq k-j+1-u}, \bigcap A_{k-j+1-u < i < k-j+1}$
- $V^I_{j',j',0}(\Box_3 V_{j,k,u}(A)) = \bigcap S_i$ where $S = \{\bigcap_{i*y \leq i < (i+1)*y} A_j\}_{0 \leq i < (k-j+1)/y}$, $x = ((k-j+1)\%u)$, $y = gcd(x,u)$

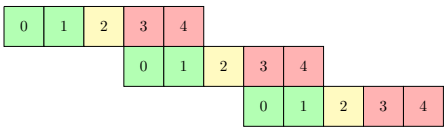


Figure 5.3: Always operator with case 2

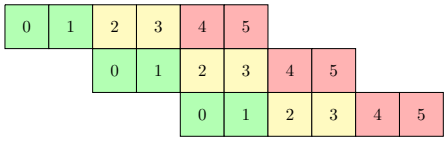


Figure 5.4: Always operator with case 3

In the first case the size of the storage S has the length $k-j+1$, because no intermediate results for a range larger than one can be created. A range larger than one has a sequence of sets of answer formulas as result. An intermediate result should combine results, but due to the larger range, each set needs to be touched individually.

For the second case, the storage S has the size of three. In the special case $k-j+1 = 2*u$, the size of the storage is two. The Figure 5.3 shows this case with sliding window of range $r = 5$ and an overlapping of $u = 2$. Each color represents one set of answer formulas in the storage S. The figure shows the results of the intersections of the first overlapping part in green, which overlaps with the previous window. The none overlapping part is marked as yellow. The second overlapping part that overlaps with the next window has the red color. For the next window, the set is moved from the end of the storage to the first position. Therefore the red sets of answer formulas are the same as the green ones in the next window.

The Figure 5.4 shows the third case. The range is smaller than twice the overlapping parameter. The storage has the size $(k-j+1)/y$, where y is the size of the input of sets of answer formulas for an intermediate result. In this example, the range is $r = 6$ and the sliding window has a sliding of 2, therefore $y = 2$ and the size of the storage

is 3. An intermediate result is used up to three times to calculate a final result. The red intermediate results are used in the next window marked as yellow. After the next window, they are used as the green results, before they are erased.

The always in the past operator has the same formula as the always operator for a given finite sequence of sets of answer formulas as input. Only the calculation of case one differs. Therefore, an always in the past operator with a range of one uses the cases 2 and 3 of the always operator. For a range larger than one, the always in the past operator is defined as:

- $V_{j,k,u}(\Box^{-}A) = \{\bigcap_{i\leq h} S_i\}_{j\leq h\leq k}$ where $S = A$

The eventually and the history operator are defined analogously as the always and always in the past operators:

- $V^{I}_{j,k,u}(\Diamond_1 A) = \{\bigcup_{i\geq h} A_i\}_{j\leq h\leq k}$
- $V^{I}_{j',j',0}(\Diamond_2 V_{j,k,u}(A)) = \bigcup S_i$ where $S = \bigcup A_{0\leq i<u}, \bigcup A_{u\leq j\leq k-j+1-u}, \bigcup A_{k-j+1-u<j<k-j+1}$
- $V^{I}_{j',j',0}(\Diamond_3 V_{j,k,u}(A)) = \bigcup S_i$ where $S = \{\bigcup_{i*y\leq i<(i+1)*y} A_j\}_{0\leq i<(k-j+1)/y}$, $x = ((k-j+1)\%u)$, $y = gcd(x,u)$
- $V_{j,k,u}(\Diamond^{-}A) = \{\bigcup_{i\leq h} S_i\}_{j\leq h\leq k}$ where $S = A$

The last operators are the since and the until operator. They are using a storage to store the incoming results in terms over reusing them at the next run. The operators do not reuse intermediate results of its own, because to compute intermediate results with their structure would be an overhead.

- $V^{I}_{j,k,0}(A \text{ U } B) = \{U_i\}_{0\leq i\leq k-j+1}$ with $U_{q-j+1} = S(B)_{q-j+1}, U_i = S(B)_i \cup (S(A)_i \cap U_{i+1})$ and $S(A) = A, S(B) = B$
- $V^{I}_{j,k,0}(A \text{ S } B) = \{S_i\}_{j\leq i\leq k}$ with $S_0 = S(B)_0, S_i = S(B)_i \cup (S(A)_i \cap S_{i-1})$ and $S(A) = A, S(B) = B$

5.1.2 The Next Function

The Initial Function initialises the storages of the operators with the results of the first sliding window. The Next Function updates this storages depending on the overlapping parameter. The expression $S << B$ means that the storage S drops as many elements of the first positions as the sequence B contains. Afterwards it adds all elements of the sequence B to the storage. In some cases, there is a storage with more than one sequence. For this reason, the storage can be accessed by $S(A)$, where A is a sequence of the storage S.

The vector operator for CQs in the Next Function is defined as:

- $V^N_{j,k,u}(v_1) = \{\Phi_i(v_1)\}_{j+u \leq i \leq k}$ where v_1 is a CQ

The conjunction and the disjunction operator of the Next Function are defined as:

- $V^N_{j,k,u}(V_{j,k,u_a}(A) \wedge V_{j,k,u_b}(B)) = S$ where $S << \{S(A)_i \cap S(B)_i\}_{i \geq u}$
  and $u = \max\{\min\{u_a, u_b\}, 0\}, S(A) << A, S(B) << B$

- $V^N_{j,k,u}(V_{j,k,u_a}(A) \vee V_{j,k,u_b}(B)) = S$ where $S << \{S(A)_i \cup S(B_i)\}_{i \geq u}$
  and $u = \max\{\min\{u_a, u_b\}, 0\}, S(A) << A, S(B) << B$

The next and previous operators have the same definition. Only the storage handling is changed by shifting the storage content with the new sets of answer formulas. The definition of the next and previous operators are defined as:

- $V^N_{j,k,u}(\circ_1 A) = A$

- $V^N_{j,k,0}(\circ_2 A) = S, \top$ where $S << A$

- $V^N_{j,k,u}(\bullet_1 A) = A$

- $V^N_{j,k,0}(\bullet_2 A) = S, \emptyset$ where $S << A$

- $V^N_{j,k,u}(\circ_1^- A) = A$

- $V^N_{j,k,0}(\circ_2^- A) = \top, S$ where $S << A$

- $V^N_{j,k,u}(\bullet_1^- A) = A$

- $V^N_{j,k,0}(\bullet_2^- A) = \top, S$ where $S << A$

The always, always in the past, eventually and history operator are changed also in the storage handling. They all shift the storage content by the new sets of answer formulas represented by A. The second case of the always and eventually operator only calculates the intermediate results for the not overlapping part and the overlapping part of the next window, because it uses the previous set of answer formulas of the overlapping part of the next window as the current part of the overlapping of the previous window. In Figure 5.3 the yellow and red part are calculated by the Next Function. The green part is equal to the red part of the previous window.

The third case of the always and eventually operator only calculates the intermediate results for the new overlapping part. This part is marked as red in Figure 5.4.

- $V^N_{j,k,0}(\Box_1 A) = \{\bigcap_{i \geq h} S_i\}_{j \leq h \leq k}$ where $S << A$

- $V^N_{j',j',0}(\Box_2 V_{j,k,u}(A)) = \bigcap S_i$ where $S << \bigcap A_{u \leq i \leq k-j+1-u}, \bigcap A_{k-j+1-u < i < k-j+1}$

- $V^N_{j',j',0}(\Box_3 V_{j,k,u}(A)) = \bigcap S_i$ where $\begin{array}{c} S << \{\bigcap_{i*y \leq j < (i+1)*y} A_j\}_{0 \leq i < len(A)/y} \\ x = ((k-j+1)\%u) \\ y = gcd(x,u) \end{array}$

- $V_{j,k,u}(\Box^- A) = \{\bigcap_{i \leq h} S_i\}_{j \leq h \leq k}$ where $S << A$

- $V^N_{j,k,u}(\Diamond_1 A) = \{\bigcup_{i \geq h} S_i\}_{j \leq h \leq k}$ where $S << A$

- $V^N_{j',j',0}(\Diamond_2 V_{j,k,u}(A)) = \bigcup S_i$ where $S << \bigcup A_{u \leq j \leq k-j+1-u}, \bigcup A_{k-j+1-u < j < k-j+1}$

- $V^N_{j',j',0}(\Diamond_3 V_{j,k,u}(A)) = \bigcup S_i$ where $\begin{array}{c} S << \{\bigcup_{i*y \leq j < (i+1)*y} A_j\}_{0 \leq i < len(A)/y} \\ x = ((k-j+1)\%u) \\ y = gcd(x,u) \end{array}$

- $V^N_{j,k,0}(\Diamond^- A) = \{\bigcup_{i \leq h} S_i\}_{j \leq h \leq k}$ where $S << A$

The since and until operator are the same as in the Initial Function. The handling of the storage is like the other operators before.

- $V^N_{j,k,0}(A \text{ U } B) = \{U_i\}_{0 \leq i \leq k-j+1}$ with $U_{q-j+1} = S(B)_{q-j+1}, U_i = S(B)_i \cup (S(A)_i \cap U_{i+1})$
  and $S(A) << A, S(B) << B$

- $V^N_{j,k,0}(A \text{ S } B) = \{S_i\}_{j \leq i \leq k}$ with $S_0 = S(B)_0, S_i = S(B)_i \cup (S(A)_i \cap S_{i-1})$
  and $S(A) << A, S(B) << B$

The next section describes the Optimised Vector Rewriting Algorithm for TCQs to the Optimised Rewriting Algorithm.

5.2 The Optimised Vector Rewriting Algorithm

The Vector Rewriting Algorithm of the Vector Algorithm is a recursive call over the entire TCQ. It defines the actually used ranges of the query to calculate the access of data. To calculate the overlapping of each operator, the Optimised Vector Rewriting Algorithm introduces an update function called U. The call of the update function is integrated into the Vector Reweiting Algorithm (Section 4.3) as follows:

- $R^O_{j,k}(v_1) = V_{j,k,u}(v_1)$ if v_1 is a CQ
  where $u = \max\{0, \min\{(r-s)-(q-(k-j)), (r-s)-j\}\}$

- $R^O_{j,k}(v_1 \wedge v_2) = U(V_{j,k}(R^O_{j,k}(v_1) \wedge R^O_{j,k}(v_2)))$

- $R^O_{j,k}(\circ v_1) = \begin{cases} U(V_{j,k}(\circ_1 R^O_{j+1,k+1}(v_1))) & \text{if } k < q \\ U(V_{j,k}(\circ_2 R^O_{j+1,k}(v_1))) & \text{if } j < k \wedge k = q \\ \{\emptyset\} & \text{otherwise} \end{cases}$

- $R^O_{j,k}(\Box v) = U(V_{j,k}(\Box R^O_{j,q}(v)))$

- ...

The Update Function takes a rewritten sub-query as input and return the new one with the overlapping parameter u. The overlapping parameter of the conjunction or disjunction operator is set to zero, if both sub-queries have different values for overlapping parameters. If the overlapping parameter has the same, then the overlapping of the conjunction or disjunction also has this value.

- $U(V_{j,k}(V_{j,k,u_1}(v_1) \wedge V_{j,k,u_2}(v_2))) = V_{j,k}(V_{j,k,u_1}(v_1) \wedge V_{j,k,u_2}(v_2))$
  where $u = \begin{cases} u = u_1 & \text{if } u_1 = u_2 \\ u = 0 & \text{otherwise} \end{cases}$
- $U(V_{j,k}(V_{j,k,u_1}(v_1) \vee V_{j,k,u_2}(v_2))) = V_{j,k,u}(V_{j,k,u_1}(v_1) \vee V_{j,k,u_2}(v_2))$
  where $u = \begin{cases} u = u_1 & \text{if } u_1 = u_2 \\ u = 0 & \text{otherwise} \end{cases}$

The first case of the previous and next operators passes through the sets of answer formulas. Therefore they can be removed. They are defined as:

- $U(V_{j,k}(\circ_1 v_1)) = v_1$
- $U(V_{j,k}(\bullet_1 v_1)) = v_1$
- $U(V_{j,k}(\circ_1^- v_1)) = v_1$
- $U(V_{j,k}(\bullet_1^- v_1)) = v_1$

The overlapping parameters of the second case of the previous and next operators are zero, because their sequence of sets of answer formulas is changing at every window.

- $U(V_{j,k}(\circ_2 v_1)) = V_{j,k,0}(\circ_2 v_1)$
- $U(V_{j,k}(\bullet_2 v_1)) = V_{j,k,0}(\bullet_2 v_1)$
- $U(V_{j,k}(\circ_2^- v_1)) = V_{j,k,0}(\circ_2^- v_1)$
- $U(V_{j,k}(\bullet_2^- v_1)) = V_{j,k,0}(\bullet_2^- v_1)$

The always, always in the past, eventually and history operators have an overlapping parameter of zero, because after each evaluation of an window, the results in terms of the sets of answer formulas are changing. The cases 2 and 3 of the operators using a storage to reuse intermediate results, if the sub-query has an overlapping larger than zero.

- $U(V_{j,k}(\Box V_{j',k',u}(v_1))) = \begin{cases} V_{j,k,0}(\Box_1 V_{j',k',u}(v_1))) & \text{if } j < k \\ V_{j,k,0}(\Box_2 V_{j',k',u}(v_2)) & \text{if } j = k \wedge k - j > 2u \\ V_{j,k,0}(\Box_3 V_{j',k',u}(v_3)) & \text{otherwise} \end{cases}$
- $U(V_{j,k}(\Box^- V_{j',k',u}(v_1))) = \begin{cases} V_{j,k,0}(\Box^- V_{j',k',u}(v_1))) & \text{if } j < k \\ V_{j,k,0}(\Box_2 V_{j',k',u}(v_2)) & \text{if } j = k \wedge k - j > 2u \\ V_{j,k,0}(\Box_3 V_{j',k',u}(v_3)) & \text{otherwise} \end{cases}$

- $U(V_{j,k}(\Diamond V_{j',k',u}(v_1))) = \begin{cases} V_{j,k,0}(\Diamond_1 V_{j',k',u}(v_1))) & \text{if } j < k \\ V_{j,k,0}(\Diamond_2 V_{j',k',u}(v_2)) & \text{if } j = k \wedge k - j > 2u \\ V_{j,k,0}(\Diamond_3 V_{j',k',u}(v_3)) & \text{otherwise} \end{cases}$

- $U(V_{j,k}(\Diamond^- V_{j',k',u}(v_1))) = \begin{cases} V_{j,k,0}(\Diamond^- V_{j',k',u}(v_1))) & \text{if } j < k \\ V_{j,k,0}(\Diamond_2 V_{j',k',u}(v_2)) & \text{if } j = k \wedge k - j > 2u \\ V_{j,k,0}(\Diamond_3 V_{j',k',u}(v_3)) & \text{otherwise} \end{cases}$

The since and until operators have different sets of answer formulas after each sliding of the window. Therefore their overlapping parameter is always equal to zero.

- $U(V_{j,k}(v_1 \text{ S } v_2)) = V_{j,k,0}(v_1 \text{ S } v_2)$

- $U(V_{j,k}(v_1 \text{ U } v_2)) = V_{j,k,0}(v_1 \text{ U } v_2)$

This is the complete update function for the rewriting of a TCQ into an Optimised Vector Query. The next chapter evaluates and compares the different approaches to establish a Temporal Query Answering with a sliding window.

6 Evaluation

All in all there are five algorithms to support a sliding window. The Chapter 3 introduces a rewriting for a TCQ to use the original algorithm [1] unchanged. The second algorithm (Algorithm 4) introduces the skipping of not used A-Boxes to reduce the overhead of useless intermediate results. Another approach is to consume the entire sliding window at once as batch. In Chapter 4 the Vector Algorithm represents this approach. It only calculates the needed intermediate results. The improvement of this algorithm to reuse the intermediate of the overlapping part is shown in Chapter 5. The last algorithm is the usage of the original algorithm over batches. With this approach, a comparison between the original algorithm and the other batch algorithms can be accomplished, because the input of the stream and batch algorithms is different.

In this chapter, the correctness of the implementations and their performance against each other is benchmarked. The section correctness describes the correctness testing of all five algorithms. It uses test sets to verify the correctness. The second section shows the benchmarks and discusses the benefits of each algorithm and their drown backs.

6.1 Correctness

In this section, the correctness tests for the five algorithms are described. There are two different types of implementations of Temporal Query Answering with sliding window. The first one is the stream implementation. It uses the original algorithm for Temporal Query Answering [1] and consumes an A-Box a time. The second type is the batch processing. It consumes the whole window at once. A batch is a sequence of A-Boxes that represents a window.

To check the correctness of the streaming implementations, each test represents a finite sequence of A-Boxes as input and defines the sliding and range of the sliding window. Furthermore each test has a sequence of answer formulas for the given sequence of A-Boxes that represents the right results for the given sliding window. This test results only contain the results for the sliding window in terms of range and sliding. This result should be equal to the result of the evaluation function at the end of each window. Therefore only the results at specific points in time are checked. The tests check the result against the sequence of answer formulas, if both checked algorithms have a result. To check the correctness of the results at all points in time, the test compares the results of both algorithms at each point. If they are not equal, the test fails. If both algorithm have no result for the current point, the test continuous with the next A-Box of the sequence.

The batch implementations consuming the entire sliding window at once. The entire window is represented as a sequence of A-Boxes, also called batch. The batch test uses the same test cases as the streaming test. To achieve that, the batch test has to build batches out of the sequence of A-Boxes with the specific range and slide of the sliding

window. A batch algorithm is correct, if it has the same result as the result of the sequence of answer formulas of the test case.

All in all both tests use the same test cases and every algorithm has the same result. A streaming algorithm should have the same result as the batch algorithm in terms of a sliding window. Each test uses two algorithm at the same time. This has two different reasons: The first reason is the check, whether the streaming algorithms return the results at the same positions. If this is not the case, one of the algorithm behave in a wrong way. The second reason is the expected result data. Not always a set of results for a specific sequence of A-Boxes is available. To check the correctness, a second verified algorithm can be used to produce the right result to compare them each other. This eliminates the manual work to calculate the right results of a given query with a Temporal Knowledge Base.

Each test only gets the test case with a sliding, a range, a Temporal Knowledge Base, the expected results and a TCQ as string. That means before the algorithm is tested, the parser has to parse the TCQ string and produce the query as object structure. The query has to be rewritten to contain the ontology of the test case. The next step is the rewriting or optimisation regarding the specific algorithm. The sliding algorithm, which uses the standard algorithm, has to rewrite the query to contain the boundaries of the window. The Vector Algorithm has to calculate the ranges of each sub-query. The Optimised Vector Algorithm calculates also the overlapping of each sub-query in terms of the sliding window. Therefore all test cases not only include the algorithms, but also the parser and the rewriting and optimisation algorithms that are used before the algorithms can be used.

To test the correctness of the algorithms, test cases are provided for each operator of TCQ. All test cases of an operator have different range and sliding values to test different combinations. The always operator has three different implementations in the Optimised Vector Algorithm. To check all of them, there are at least three test cases which cover all three implementations. All algorithms calculate the right results for all test cases. Therefore the algorithms behave all the same way. This means that the Vector Algorithm calculates the same result of answer formulas as the streaming algorithm with a rewritten TCQ in terms of Temporal Query Answering with sliding window.

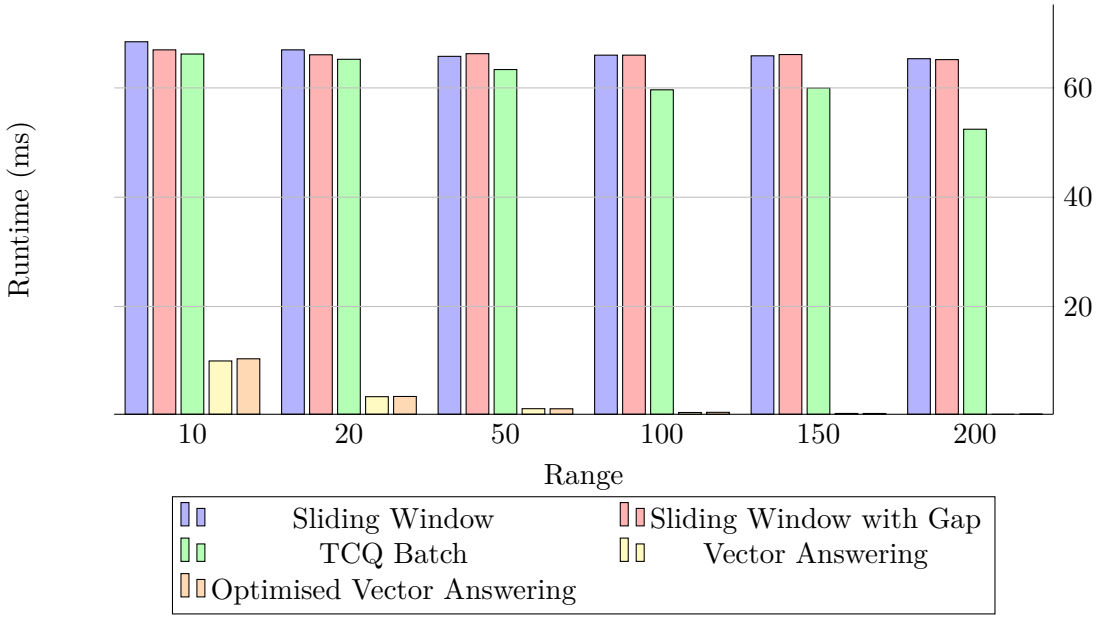
6.2 Benchmarks

In the last section, the correctness tests are described. They verify that all five algorithms have the same semantic to answer TCQs. In this section, the performance of them is compared to each other. The goal is to classify the different algorithms in regard to the performance. The classification should suggest which algorithm is able to perform best performance under a given configuration.

The performance of an algorithm is based on the amount of data, the complexity of the TCQ and the reuse of intermediate results in terms of window overlapping. The benchmarks compare the algorithms with different settings of those influences.

The benchmarks measure the execution time of the algorithms. To get a precise result,

Figure 6.1: Benchmark of TCQ $Q(x) = hasVal(x, y)$



the test data is loaded completely prior to execution of the algorithm and is measured by the CPU time in nano seconds. The test system has an i5-2520M as CPU and 8 GB RAM installed.

6.2.1 Range and Slide are equal

The first set of benchmarks deals with a sliding window that has the same value for the range and slide. This means that the windows are not overlapping. Furthermore there is no gap between two windows. Due to the specific slide and range values, the reuse of intermediate results in terms of overlapping is not possible.

In this subsection, all five algorithms are compared. All five algorithms can handle the setting and return the same result for a given Temporal Knowledge Base and TCQ. All benchmarks in this subsection have a Temporal Knowledge Base with an empty ontology. The sequence of A-Boxes contains 1000 A-Boxes. Each A-Box contains 50 sensor data as role assertions. Each role has as subject the name of the sensor and as object a value of type double. Each A-Box has the same assertions.

The first benchmark has the TCQ $Q(x) < -hasVal(x, y)$. It should query all sensor names with the role assertion $hasVal$. The query is queried over different window sizes. All different settings use the same sequence of A-Boxes. The Figure 6.1 represents the results. The first window size is ten. Therefore the input represents a sliding window with 100 different positions. The result of the TCQ is a sequence of 100 sets of answer formulas. The last window size is 200. Therefore the sequence of A-Boxes only represents 5 windows and has a sequence of 5 sets of answer formulas as result.

The first algorithm is the Sliding Window, which uses a rewritten TCQ to contain

the boundaries of the sliding window. The rewritten TCQ is the same as Q. Only the sequence of the Evaluation Function calls is changed. It is called every 10 A-Boxes. The second algorithm is the Sliding Window with Gap. It uses the original TCQ Q and calls the Initial Answer Formula at the begin of a window. These two algorithms use the sequence of A-Boxes as input. The other three ones have a sequence of batches as input. Each batch represents a window. The TCQ Batch algorithm is the original algorithm performed over batches. The Vector Algorithm and the Optimised Vector Algorithm are only calculating the needed intermediate results to answer the TCQ regarding the sliding window.

The Figure 6.1 shows that there are two different result areas. The first three algorithms have nearly the same results. They use the original algorithm. The execution of the first two algorithms is nearly the same. They are different in terms of the Initial Function calls of the second one. The third algorithm queries over the same A-Boxes, which are represented as batches. The second result area is represented by the Vector Algorithms. They need much less time to query the TCQ against the sequence of A-Boxes. The reason for this is the calculation of intermediate results. The first three algorithms calculate useless intermediate results and the Vector Algorithms only touch A-Boxes that are needed for the result of the sliding window. In the last run with a range of 200 A-Boxes, the first three algorithms touch 1000 A-Boxes, the Vector Algorithms only touch 5 A-Boxes.

The TCQ Q only queries the last point in time of the sliding window. The next algorithm queries over the complete window. Given the TCQ $Q_2(x) = \Box^-(hasVal(x, y))$, which queries over the complete window and the window has the same A-Boxes as before, then the query Q_2 has the same result as Q, because all A-Boxes are the same and the intersection of one A-Box with another one results into the same A-Box. The only difference to the benchmark above is the fact that every A-Box has to be touched. The runtime of the first algorithm Sliding Window for the TCQ Q_2 is shown in Figure 6.2. It needs much more time as the other algorithms to calculate the results for the TCQ Q_2. This is due to the rewriting of the query to contain the boundaries of the sliding window. The rewriting of the always in the past operator changes the complexity of Q_2 linear to the range of the window and computes every intermediate result. This is only the case in this given TCQ. If the TCQ contains more nested operations, the complexity would increase exponential. The runtime of the Sliding Window for the TCQ Q_2 is measured in seconds. The other algorithms are represented in Figure 6.3. Their runtime is represented in milliseconds. All four algorithms in Figure 6.3 are a lot faster than the Sliding Window Algorithm (Algorithm 3). The runtime of all four algorithms is nearly the same. The Sliding Window Algorithm with Gap uses the not rewritten version of the query. Therefore it is much faster than the rewritten version in Figure 6.2, but it is the slowest algorithm compared to the batch algorithms. One reason for this behavior could be the consumption of a sequence of A-Boxes instead batches. The algorithm has to control the handling of the A-Boxes to call the right function depending on the current A-Box of the sequence.

Figure 6.2: Benchmark of TCQ $Q_2(x) = \Box^-(hasVal(x, y))$ of Sliding Window

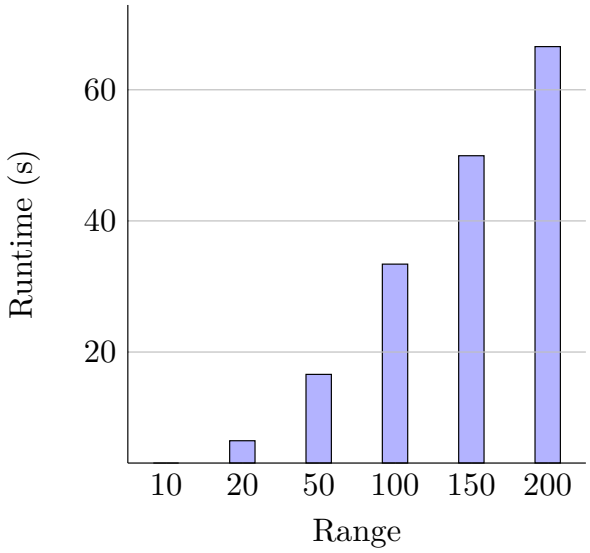
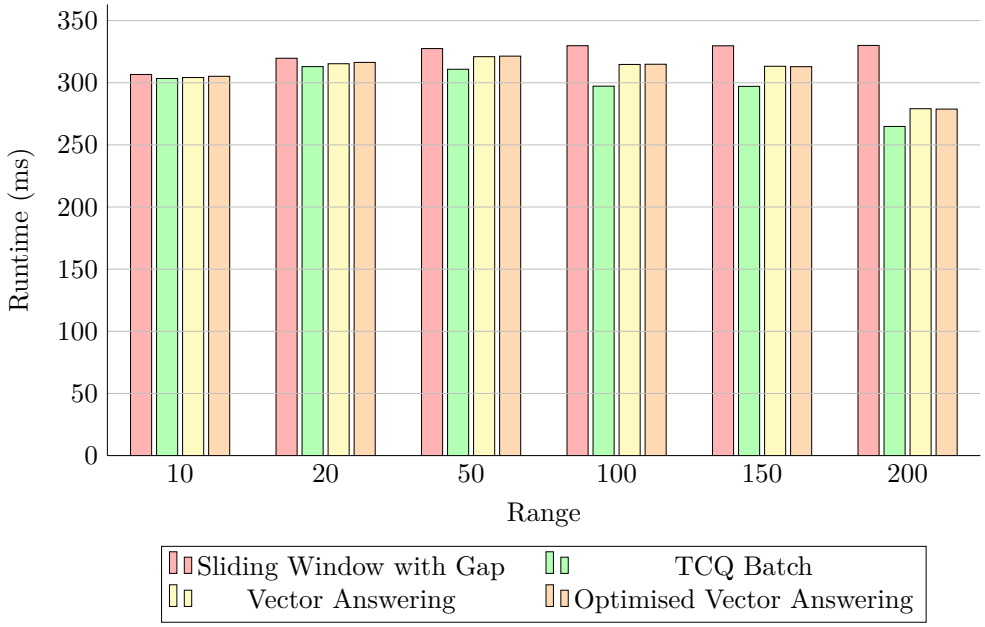


Figure 6.3: Benchmark of TCQ $Q_2(x) = \Box^-(hasVal(x, y))$



6.2.2 Range is larger than Slide

In this subsection, the previous benchmark is running over a sliding window with a much larger range than slide. In this case caching of intermediate results should be an improvement.

Figure 6.4 represents the runtime of the Sliding Window Algorithm depending on the window sizes 100 and 200 with different sliding parameters. The runtime is represented in seconds. The Sliding Window Algorithm uses the rewritten TCQ. It consumes the sequence of A-Boxes and evaluates the TCQ at the end points of the windows. The algorithm with the rewritten TCQ has a much longer runtime than the other algorithms. It touches each A-Box and calculates all intermediate results. As in Figure 6.4 shown, the runtime of the algorithm depends on the complexity of the rewritten TCQ and the range of the window. The runtime of the algorithm for the window size 100 is nearly equal. This holds also for the cases of the window size 200. The runtime of both different window sizes depends on the complexity of the TCQ. In this case, the original TCQ is the same, but the rewritten TCQ depends on the window size. The rewritten TCQ for the window size 100 is much simpler than the TCQ for the window size 200.

In Figure 6.5, the same TCQ is executed by the TCQ Batch, the Vector Answering and the Optimised Vector Answering Algorithm. The Sliding Window Algorithm that handles gaps can not be applied in this case, because the range of the sliding window is larger than its slide. That means an overlapping of the windows exists. The runtime in this figure is measured in seconds. It is nearly the same for all three algorithms. The first 4 measurements of the algorithm uses a range of 100. The runtime decreases by increasing the sliding. It is nearly linear. This is the case due to the structure of the benchmark. By changing the sliding parameter, the number of windows changes. In the previous Figure 6.4, the algorithm runtime depends on the rewritten TCQ in regard to the range, because they consume a sequence of A-Boxes that are equal at each run in this benchmark. In this figure the three algorithms are consuming batches. The number of batches changes depends on the sliding parameter. By increasing the sliding parameter the runtime and the number of windows decreases in a linear way. The runtime depends also on the window size.

All three algorithms nearly have the same runtime, but the handling of the TCQ is different. The Optimised Vector Algorithm uses previous calculated intermediate results of the overlapping part. The handling of the reusing of already calculated parts also takes time. Therefore the execution of the Vector Algorithm and the optimised version of it are nearly the same in this case.

6.2.3 Range is smaller than Slide

In this subsection, the TCQ $Q_2(x) = \Box^-(hasVal(x, y))$ is queried over the same sequence of A-Boxes with a larger sliding than window range. In Figure 6.6, the results for the Sliding Window Algorithm with the rewritten TCQ are shown. It uses the original algorithm and calculates all intermediate results for all sub-queries. Therefore the execution time depends on the window size. The sliding parameter only defines the

Figure 6.4: Benchmark of TCQ $Q_2(x) = \Box^-(hasVal(x, y))$ of Sliding Window

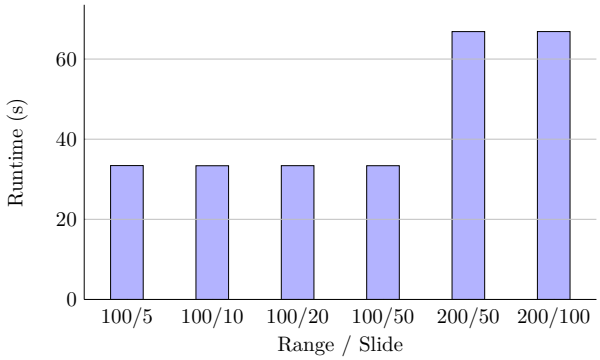


Figure 6.5: Benchmark of TCQ $Q_2(x) = \Box^-(hasVal(x, y))$

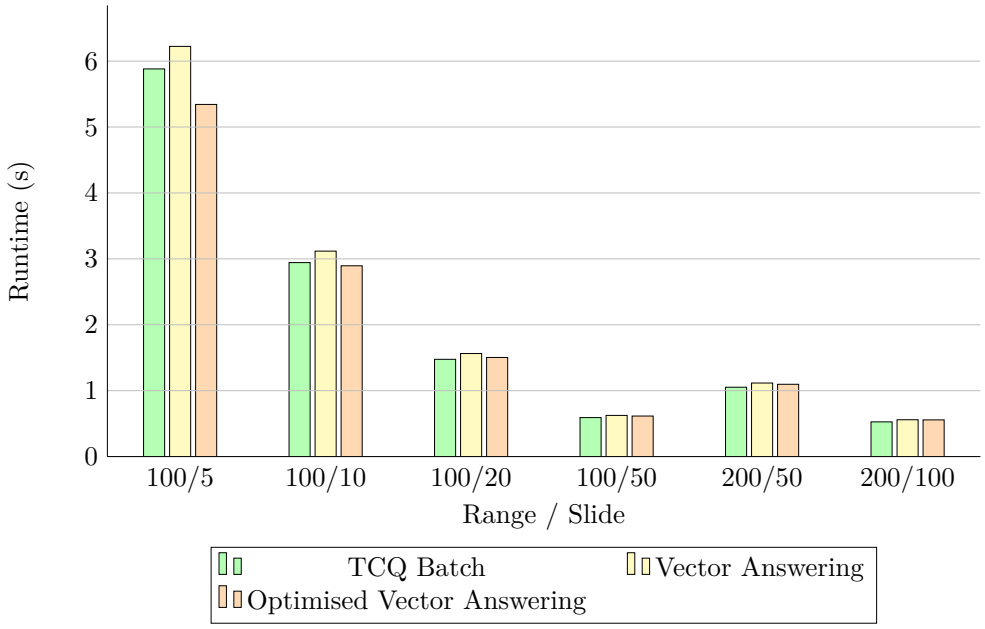
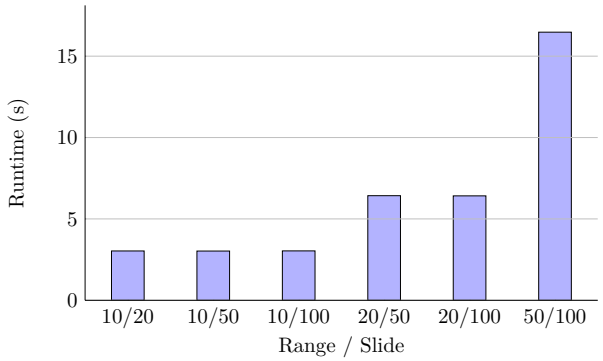


Figure 6.6: Benchmark of TCQ $Q_2(x) = \Box^-(hasVal(x, y))$ of Sliding Window

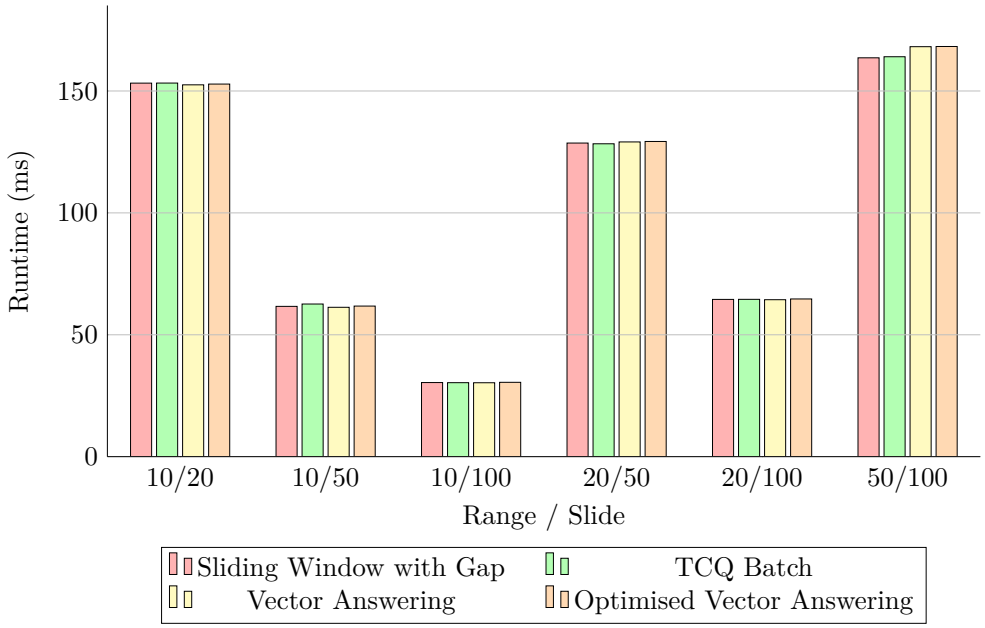


frequency of Evaluation Function calls. Therefore the runtime of the algorithm is equal for a given TCQ, if the window size is the same. Due to the rewriting of the TCQ, the rewritten TCQ is much larger than the original one. The runtime for the Sliding Window Algorithm is measured in seconds.

The other four algorithms in Figure 6.7 are much faster. Their runtimes are measured in milliseconds. All four algorithms are much faster, because they are skipping useless A-Boxes. All four algorithms nearly have the same execution time, but they are handling the gap in different ways. The Sliding Window Algorithm with Gap has a stream of A-Boxes as input. It handles the gap by itself and only calculates the necessary results. The other three algorithms have a sequence of batches as input. This sequence of batches already represents the set of windows with the specific sliding.

The TCQ $Q_2(x)$ is queried against each A-Box of the window the same way. The entire TCQ is an intersection of all A-Boxes. Therefore the execution time decreases linear by increasing the sliding parameter, because it can skip more A-Boxes. The Figure 6.7 represents the execution time for a sliding window with the range of 10, 20 and 50. The window range of 10 has three different sliding parameters. They have the value 20, 50 and 100. The figure shows that an increase of the sliding results in a decrease of the runtime. This is not only the case for a range of 10, but also the case for the range of 20. Furthermore the figure shows that the runtime depends on the range and the sliding value. Given the range of 10 and the sliding value 50, then the runtime time increases by doubling the range. This new runtime is represented by the case 20/50.

Figure 6.7: Benchmark of TCQ $Q_2(x) = \Box^-(hasVal(x, y))$



6.3 Result

The Sliding Window Algorithm uses a rewritten TCQ to query against a sequence of A-Boxes. The advantage of using the Sliding Window Algorithm is the consuming of a sequence of A-Boxes. No batches have to be created to use this algorithm. The disadvantage is the rewriting of the TCQ. The rewritten TCQ is exponential larger than the original TCQ. It introduces a lot of new sub-queries to create the same meaning without the unbounded operations like always or eventually. Due to this rewriting, the algorithm uses a lot of memory space to query the TCQ against the sequence. In the benchmarks above, the Sliding Window Algorithm is the slowest algorithm. It needs significant more runtime than the other algorithms to solve a TCQ. In practice, the other algorithms are preferred.

The Sliding Window Algorithm with Gap uses the original algorithm to query the TCQ against the sequence of A-Boxes. The advantage of this algorithm is the usage of the sequence of A-Boxes. A transformation into batches is not needed. The second advantage is the usage of the original TCQ. This keeps the complexity small and enables a better memory space usage against the algorithm before. The runtime of the algorithm is also faster in comparison to the Sliding Window Algorithm. The disadvantage is the leak of support for overlapping sequences. This algorithm is optimal to query none overlapping sliding windows over a sequence of A-Boxes without rewriting the TCQ.

The TCQ Batch Algorithm is the original algorithm of [1]. It has the same implementation to answer the TCQ against a sequence of A-Boxes as the previous two algorithms. The algorithm is executed over a sequence of batches. The advantage of

this implementation is the usage of the original algorihm and the low memory usage. The disadvantages of the algorithm are the consumption of batches, which have to be created and the inefficiency of calculating useless intermediate results. It queries against all A-Boxes, in depending of the TCQ. A TCQ without temporal operations would touch all A-Boxes instead only the last one. This algorithm is optimal for TCQ Answering with overlapping batches without rewriting the TCQ.

The Vector Algorithm is the second algorithm that consumes batches. It queries only against the needed A-Boxes to evaluate the result of a batch. Therefore the query is rewritten to work over ranges. The result of an operator is a vector of sets of answer formulas. The advantage of this algorithm is the reduction of accessing data of A-Boxes. If a database holds the A-Boxes, this algorithm reduces the access to the database to a minimum. The disadvantage of this algorithm is the leak of implementation to cache intermediate results for overlapping parts of the query.

The last algorithm is the Optimised Vector Algorithm. It is build on top of the Vector Algorithm to implement the leak of caching intermediate results. To implement a caching, the rewriting algorithm calculates the overlapping for each sub-query of the TCQ. Each sub-query gets its own overlapping parameter depending on their own sub-queries. The advantage of this approach is the reuse of intermediate results for large data requests against a database. In the benchmarks, the batches are already loaded in the memory. Therefore the disadvantage of this algorithm is the overhead to calculate the intermediate results to enable the reuse of them in the next point in time.

The implementation of a TCQ Answering with Sliding Window should be done with two different algorithms: The first algorithm should be the Vector Algorithm to answer a specific TCQ against a sequence of batches. It performs quite well regarding runtime. To implement operations that are defined as holistic or context-sensitive [6] the intermediate results can be used, because each sub-query has its intermediate results at once as vector. The optimised version of it has no benefit in the benchmarks above. Its overhead of handling the reusement is too high for our test cases. If the windows are not overlapping and their range is quite large, the Sliding Window Algorithm with Gap should be preferred, because it can answer TCQs over large window ranges that exceeds the memory space. The previous preferred algorithm is based on answering batches. If a batch is larger than the available memory, the memory limit is exceeded and the algorithm stops to work. Therefore the Sliding Window Algorithm with Gap should be used to consume a sequence of A-Boxes. Queries with memory exceeding batches that are overlapping can not be processed by both algorithms.

7 Future Work

This thesis represents approaches to deal with a sliding window for TCQ Answering. It introduces five different algorithms to handle a sliding window. The sliding window is introduced to support TCQ Answering with holistic and context-sensitive operations [6]. In the last chapter, the correctness of the implementation was checked and the runtime was measured. It shows that the Vector Algorithm is the best choice to deal with TCQs in order to contain holistic or context-sensitive operations. If the window range of a sliding window is quite larger than the entire batch exceeds the memory space, the Sliding Window Algorithm with Gap should be used, if the windows do not overlap.

The next step is the implementation of holistic and context-sensitive operations. For the Vector Algorithm, the operations can be implemented by using the already existing vector of intermediate results. In the context of the Sliding Window Algorithm with Gap, the original algorithm of [1] is used. Therefore a solution to collect all intermediate results and calculating the result at the end of the window has to be discovered. This can be achieved by constructing a data structure that consumes the results of the sub-query and saves it for the entire window by knowing the range and sliding of the sliding window.

The implementation is written in Java. It contains all five algorithms and a parser to parse a TCQ as string. The parser is based on the parser generator ANTLR [7]. For each algorithm there exists a factory class that the parser uses to build the query structure for the algorithm. Furthermore the implementation contains two Trident Assemblies. Trident is an abstraction framework of the Storm Cluster Framework. The first assembly is for processing an infinite sequence of A-Boxes with the algorithms Sliding Window and Sliding Window with Gap. Also the original algorithm can run with this assembly. The second assembly gets a stream of batches as input. It is build for the TCQ Batch, the Vector Algorithm and the Optimised Vector Algorithm.

The implementation only implements the algorithms with the Trident assemblies and the parser. An extension would be a user interface to build queries and show the results in real time to the end user. Another extension is the implementation of a cluster environment based on Storm that manages the committed algorithms and handles the streams of all assemblies.

8 Conclusion

This thesis introduced the sliding window for Temporal Conjunctive Query Answering. It presents five algorithms to deal with a sliding window. In Chapter 3 the Sliding Window Algorithm is introduced. It uses a rewritten TCQ to compile the boundaries of the window into the TCQ. In Section 3.4 the Sliding Window Algorithm is extended to the Sliding Window Algorithm with Gap. It reduces the overhead of touching useless A-Boxes. Both algorithms of Chapter 3 are consuming a sequence of A-Boxes.

The second approach of a sliding window for TCQ Answering is the batch processing. Each batch represents a window, therefore the original algorithm can be used. The algorithm TCQ Batch implements the answering of TCQs against batches with the original algorithm [1]. Another algorithm for TCQ Answering on batches is the Vector Algorithm. It is introduced in Chapter 4. This algorithm only calculates intermediate results that are used to answer the TCQ at the last point in time of the window. In Chapter 5 it is shown how to improve the Optimised Vector Algorithm in case of overlapping windows.

All five algorithms are implemented in Java. Their implementation was checked with correctness tests in Section 6.1. Afterwards their runtimes were benchmarked in Section 6.2 in terms of a sliding window. In Section 6.3, the results of the benchmarks were discussed.

All in all the sliding window for TCQ Answering to support holistic or context-sensitive operations is introduced and discussed. All five presented algorithms produce the same sets of answer formulas as result for a given sequence of A-Boxes. The Vector Algorithm is the best choice to answer TCQs with a sliding window. It is optimal to implement holistic and context-sensitive operations in the context of TCQ Answering, because all intermediate results of the sub-queries are represented as vector.

The benchmarks assume the present of the sequence at once. In reality, the A-Boxes of the sequence are not arriving at a time. Therefore the Sliding Window Algorithm with Gap and the TCQ Batch Algorithm can start the evaluation before the entire window is present, iff there is no holistic or context-sensitive operator present in the query. If a holistic or context-sensitive operator is present in the query, both algorithms can only start the evaluation, if the entire window is received. The Vector Algorithm has nearly the same runtime as both algorithms. Therefore the Vector Algortihm is the best choice to establish Temporal Query Answering regarding to holistic and context-sensitive operators.

Bibliography

[1] Stefan Borgwardt, Marcel Lippmann, and Veronika Thost. Temporal query answering in *DL-Lite*. In Thomas Eiter, Birte Glimm, Yevgeny Kazakov, and Markus Krötzsch, editors, *Description Logics*, volume 1014 of *CEUR Workshop Proceedings*, pages 80–92. CEUR-WS.org, 2013.

[2] Stefan Borgwardt, Marcel Lippmann, and Veronika Thost. Temporal query answering w.r.t. *DL-Lite*-ontologies. LTCS-Report 13-05, Chair of Automata Theory, TU Dresden, Dresden, Germany, 2013. Revised version. See http://lat.inf.tu-dresden.de/research/reports.html.

[3] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, and Riccardo Rosati. Ontologies and databases: *The DL-Lite Approach*. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutierrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web*, volume 5689 of *Lecture Notes in Computer Science*, pages 255–356. Springer, 2009.

[4] Diego Calvanese, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-muro, and Riccardo Rosati. Ontologies and databases: The dl-lite approach. In *In Reasoning Web, volume 5689 of LNCS*, pages 255–356, 2009.

[5] Jan Holste. *Implementation of an Algorithm for Temporal Query Answering on Ontologies*. Projektarbeit, TU Hamburg-Harburg, April 2014.

[6] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, December 2002.

[7] Terence Parr and Kathleen Fisher. Ll(*): *The Foundation of the ANTLR Parser Generator*. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.