INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelorarbeit

Propagation of ESCL Cardinality Constraints with Respect to CEP Queries

Thanh Son Dang

Aufgabensteller:

Prof. Dr. François Bry

Betreuerin:

Dipl.-Inform. Olga Poppe

December 6, 2011

Acknowledgements

I would like to thank Prof. Dr. François Bry, Simon Brodt, Steffen Hausmann, and especially my supervisor Olga Poppe for discussing and supporting my work. Olga Poppe contributed much to the elaboration of the concepts and writing of this report. I enjoyed getting to know the inner life of the research unit and being regarded as a valued part of it.

Abstract

Semantic optimization of database queries, i.e. the use of metadata (constraints) for query optimization, is well investigated and has led to significant performance gains. The algorithm semantically rewriting database queries presented in [2] is applicable to CEP queries with some adaptions. One of these adaptions concerns the treatment of views and the constraints for them.

To reduce the number of constraints which must be manually specified by the user, the algorithm semantically rewriting database queries presented in [2] leads views back to their respective base relations so that constraints have to be defined for base relations only. However, if a CEP engine involves automatic garbage collection [1] of events and states, the relations saving these events and states may become incomplete in the meantime such that CEP views cannot be led back to them without lost of tuples in some cases. Therefore, an approach opposite to the one in [2] is introduced in this work: ESCL [5] cardinality constraints are automatically propagated from base events (states) to the derived events (states) with respect to the queries deriving them, i.e. only the ESCL cardinality constraints for base events and states must be manually specified by the user. The propagation of the other kinds of ESCL constraints is to be investigated.

The approach presented in this report is independent from an event query language. To illustrate it, CEP queries are expressed in StreamLog [5], a first order logic language extended with temporal aspects as required for CEP.

Contents

Chapter 1

Introduction and Motivation

Event-based systems work with large amount of event data continuously arriving on potentially infinite event streams. Under these circumstances, garbage collection is often indispensable. Garbage collection means the possibility to delete events if they cannot contribute to new answers of a query (any more) [1]. Hence, some base relations possibly became incomplete in the meantime. As a consequence, CEP views cannot be led back to their base events (states) without lost of results in some cases. Therefore, in order to semantically optimize CEP queries which are based on CEP views, constraints have to be defined for these views too and not only for their base events and states. To reduce the number of constraints which must be defined by the user, only the constraints for base events and states will be manually specified and automatically propagated to CEP views with respect to the queries deriving the views.

Example 1.
If an event a is derived from two events b and c arriving during some time interval i and it is known that the events b and c are unique during i, then the derived event a is also unique during i.

Example 1 demonstrates an essential difference between the propagation of database constraints and the propagation of CEP constraints, namely database constraints are valid always in contrast to CEP constraints which are usually valid during particular time intervals (see [5] for more details). If the validity time of CEP constraints is not the same as the evaluation time of CEP queries (which is usually the case), the propagation of constraints is not that trivial. We are aware of existing approaches to this topic neither in database systems nor in CEP.

The idea of constraint propagation is dependent neither from an event query language nor from a constraint language. To illustrate the approach in this work, ESCL cardinality constraints are propagated with respect to StreamLog queries. The languages ESCL and StreamLog are formally defined in [5].

ESCL cardinality constraints specify the number of events or states during time intervals. These constraints make the following semantic query optimization techniques possible:

1. *Recognition of query unsatisfiability.* Assume a query derives an event a if there is no event b during a time interval i. If the cardinality of b during i is not zero then the query is unsatisfiable and will not be evaluated.

2. *Query suspension* as long as the queried data is incomplete. CEP queries are usually satisfiable during particular time intervals or states, i.e. the cardinality of queried events during these time intervals is either not null if the respective query is positive or null if the respective query is negative. If these time intervals or states have not begun or are already over the respective queries are not satisfiable and their evaluation can be suspended.

3. *Scan reduction*, i.e. the interruption of reading of buffer as soon as all relevant data has been found.

4. *Query termination* as soon as all relevant data has arrived. This is especially important for queries with grouping. If the number of events in a group is known then an aggregation function (e.g., *max* or *sum*) can be applied to the group as soon as the group is complete even if the time interval for the grouping is not over.

5. *Choice of join algorithms* depends on the number of joined events which is specified by cardinality constraints.

6. *Join introduction*, i.e. introduction of a join with a seldom event or state x into a query to suspend the query as long as x is not available and to reduce the amount of data which must be further processed.

These semantic optimization techniques (except for query suspension and query termination) are also applicable to database queries.

The main contribution of this work is the formal definition of the propagation of ESCL cardinality constraints with respect to StreamLog queries. To the best of our knowledge, this is the only existing approach going in this direction. Analogously to the propagation of cardinality constraints, the propagation of the other kinds of constraints can be defined both for database and CEP queries.

This work is organized as follows. Chapter 2 is devoted to the classification of ESCL cardinality constraints for the purpose of the step-wise inductive definition of their propagation. Chapter 3 introduces the cardinality propagation function. Chapter 4 presents the assumptions and describes the challenges of the approach. Chapter 5 and Chapter 6 are devoted to the formal specification of the cardinality propagation. In some cases the derived cardinality constraints can be simplified. Chapter 7 defines the rules for the simplification of cardinality constraints. Propagation of ESCL causal, temporal, and data constraints is a subject of future work. Examples are given in Chapter 8.

Chapter 2

Classification of ESCL Cardinality Constraints

An ESCL cardinality constraint specifies the number of events or states during a time interval. For the purpose of step-wise inductive specification of the propagation of these constraints, we differentiate between constraints with simple and complex cardinality specifications.

Definition 2.1 (**Simple cardinality specification**).
A simple cardinality specification consists of a comparison operator $=$, $\leq$ or $\geq$ and a natural number $n \in \mathbb{N}_0$, both written in the index of the existential quantifier ($\exists$).

Each simple cardinality specification describes an interval of natural numbers. Let $n \in \mathbb{N}_0$, $(= n)$ corresponds to the interval $[n, n]$, $(\leq n)$ describes the interval $[0, n]$ and $(\geq n)$ specifies the right-open interval $[n, \infty)$.

Simple cardinality specification involving comparison operators $<$ or $>$ can be simulated by $\leq$ or $\geq$, since $< n$ is equivalent to $\leq n - 1$ and $> n$ to $\geq n + 1$ where $n \in \mathbb{N}_0$.

Definition 2.2 (**Complex cardinality specification**).
A complex cardinality specification is a conjunction or a disjunction of an arbitrary number of simple or complex cardinality specifications in the index of the existentional quantifier.

A disjunction of cardinality specifications defines the union of the intervals specified by the disjuncts. For example, $(= 2 \vee \geq 8)$ is equivalent to $[2, 2] \cup [8, \infty)$. A conjunction of cardinalinality specifications defines the intersection of the intervals specified by the conjuncts. For example, $(\geq 4 \wedge \leq 8)$ is equivalent to $[4, 8]$, more precisely $[4, \infty) \cap [8, \infty) = [4, 8]$.

Note that a cardinality specification involving the comparison opeator $\neq$ is also complex, since, e.g., $\neq 3$ corresponds to $(\leq 2 \vee \geq 4)$.

Example 2.
$i : \exists_{(\geq 3 \wedge \leq 6) \vee (= 8 \vee \geq 12)}\, e \leftarrow$ is the constraint with the complex cardinality specification $(\geq 3 \wedge \leq 6) \vee (= 8 \vee \geq 12)$ of the events matching e during the time interval i. The constraint body is empty, since the constraint holds unconditionally. Consider the defintion of ESCL cardinality constraints in [5].

In order to exclude cardinality constraints which make no sense, the following definition formally specifies the validity of a cardinality constraint.

Definition 2.3 (**Valid cardinality specification**).
A simple cardinality specification is always valid. A disjunction of valid cardinality specifications is valid. A conjunction of valid cardinality specifications is valid if the intersection of the intervals specified by the conjuncts is not empty.

Example 3.
The cardinality specification $(\leq 2 \wedge \geq 4)$ is invalid, since $[0, 2] \cap [4, \infty) = \emptyset$. However, the cardinality specifiction $((\leq 2 \vee = 4) \wedge \geq 4)$ is valid, since $([0, 2] \cup [4, 4]) \cap [4, \infty) = [4, 4]$.

Chapter 3

Cardinality Propagation Function

In this section, we introduce the function that computes the cardinality specification of the propagated constraint.

Definition 3.1 (**Cardinality Propagation Function**).
Let *Queries* be a set of StreamLog queries. Let $q \in Queries$ be a query of the form $i : h \leftarrow b$ where i is the evaluation time interval, h is the head and b is the body of the query q. Note that b is an arbitrarily nested conjunction or disjunction of literals $l_1, \dots l_n$ and conditions on them. Moreover, let $i_1, \dots, i_n$ be the evaluation time intervals of $l_1, \dots, l_n$ respectively.

Let *Cardinalities* be the set of cardinality specifications (Definitions 2.1 and 2.2) of the cardinality constraints for the events or states matching atoms $m_1, \dots, m_k$ such that $\forall l_x \in \{l_1, \dots, l_n\}\ \exists m_y \in \{m_1, \dots, m_k\}\ \exists$ a substitution θ such that $m_y \supseteq_\theta l_x$ or $l_x \supseteq_\theta m_y$ with θ [5]. Let $j_1, \dots, j_k$ be the validity time intervals of these constraints respectively.

The **cardinality propagation function**:

$$\mathcal{C} : \ Cardinalities^k \ \rightarrow \ Cardinalities$$

takes the cardinality specifications of the cardinality constraints for the events or states matching $m_1, \dots, m_k$ as parameters and returns the cardinality specification of the constraint for h with the validity time i. This function will be formally defined in Chapter 5 and Chapter 6.

Note that the number of all events and states is defined in all time intervals since the default cardinality specification ≥ 0 always holds. Therefore, the propagation of cardinality constraints is possible even if the constraint set is incomplete.

Chapter 4

Assumptions and Challenges

Let q be a StreamLog [5] query with respect to which constraints are propagated. q is of the form $i : h \leftarrow b$ where i is the evaluation time interval, h is the head and b is the body of query q. The cardinality propagation function is defined under the following assumptions:

1. We assume all literals in b to be positive. The consideration of negative literals in b is a subject for future work.

2. As the informal introduction of the cardinality propagation function $\mathcal{C}$ in Chapter 3 shows, the result of cardinality propagation with respect to q can be based on the cardinality of either more specific or more general atoms as the atoms appearing in b. Consider Example 4.

   Example 4.
   Assume there are exactly two events matching the atom $f(a)$ within some time interval i (i.e. $i : \exists_{=2}\ f(a) \leftarrow$). Assume there is an atom $f(X)$ in b such that the cardinality of $f(X)$ is unknown. Instead of using the default cardinality constraint $i : \exists_{\geq 0}\ f(X) \leftarrow$, one could infer $i : \exists_{\geq 2}\ f(X) \leftarrow$.

   And vica versa: Assume there are exactly two events matching the atom $f(X)$ within some time interval i (i.e. $i : \exists_{=2}\ f(X) \leftarrow$). Assume there is an atom $f(a)$ in b such that the cardinality of $f(a)$ is unknown. Instead of using the default cardinality constraint $i : \exists_{\geq 0}\ f(a) \leftarrow$, one could infer $i : \exists_{\leq 2}\ f(a) \leftarrow$.

   For simplicity reasons in the following, we assume that the function $\mathcal{C}$ takes the cardinality constraints for atoms appearing in b. But keep in mind that these constraints can be inferred as shown in Example 4.

3. For simplicity reasons, we assume also that all cardinality constraints considered in the following are facts, i.e. their bodies are empty. But remember that cardinality

constraint propagation is also possible when the constraints are rules as the following example demonstrates.

Example 5.
Consider the cardinality constraint $i : \exists_{=2}\ l_1 \leftarrow l_2$. The constraint means that if there is an event matching l_2 during the time interval i than the number of events matching l_1 during i is exactly 2. Consider the rule $i : h \leftarrow l_1 \wedge l_2$. The above cardinality constraint can be used for the constraint propagation with respect to this rule since l_2 is implied by the rule. But the above cardinality constraint cannot be used for the constraint propagation with respect to the rule $i : h \leftarrow l_1 \vee l_2$ in general. Except for the case if the number of events matching l_2 in i cannot be zero, i.e. l_2 is implied by another constraint.

4. For the sake of brevity, cardinality constraints involving comparison operators $<, >$ and $\neq$ are not considered in the following. They can be simulated by the comparison operators $\leq, \geq$ and $=$ as described in Chapter 2.

5. And finally, we assume all considered cardinality constraints to be valid (Definition 2.3) and simplified, i.e. as short as possible (Chapter 7).

There are the following challenges of the formal definition of the cardinality propagation function:

1. The evaluation time of an atom in a query body does usually not coincide with the validity time interval of the cardinality constraint restricting the number of events (states) matching the atom. We start in Chapter 5 with the case in which both time intervals coincide and continue in Chapter 6 with the case in which they do not coincide.

2. The number of atoms in a conjunction or a disjunction within a query body can be arbitrary. Chapter 5 and Chapter 6 are divided into two sections. The first one of them defines the cardinality propagation function $\mathcal{C}$ with respect to a query the body of which is a conjunction or a disjunction of only two atoms. This is the base case of the function. The second section defines the inductive case of the function, i.e. the cardinality propagation with respect to a query the body of which is an arbitrarily nested conjunction or disjunction of atoms.

3. The cardinality specifications of a cardinality constraint can be arbitrarily nested. We introduce the propagation of cardinality constraints with simple cardinality specifications (Definition 2.1) first and then concentrate our attention on constraints with complex cardinality specifications (Definition 2.2).

Chapter 5

Constraint Validity Time Equals Query Evaluation Time

In this chapter, we consider with the case in which the query evaluation time concides with the validity time intervals of the constraints which are propagated with respect to the query. We start in Section 5.1 with the cardinality propagation with respect to a query with only two atoms in the body (this is the base case of the propagation) and continue in Section 5.2 with the cardinality propagation with respect to a query with an arbitrary number of atoms in the body (this is the inductive case of the propagation). In both cases, constraints with simple and complex cardinality specifications are considered.

5.1 Two Atoms in a Query Body

Let q be the StreamLog query of the form $i : h \leftarrow A \circ B$ where $\circ \in \{\wedge, \vee\}$, A, B and h are atoms and i is a time interval. Let c_A and c_B be cardinality specifications of the cardinality constraints for A and B respectively with the validity time i.

The application of cardinality propagation function $\mathcal{C}$ to c_A and c_B depends on the operation $\circ$. If $\circ$ equals $\wedge$, the body of q is a conjunction of two atoms and the function $\mathcal{C}_\wedge$ is used in place of $\mathcal{C}$, i.e. $\mathcal{C}(c_A, c_B) = \mathcal{C}_\wedge(c_A, c_B)$. Otherwise, the body of q is a disjunction of two atoms and the function $\mathcal{C}_\vee$ is used in place of $\mathcal{C}$, i.e. $\mathcal{C}(c_A, c_B) = \mathcal{C}_\vee(c_A, c_B)$. The functions $\mathcal{C}_\wedge$ and $\mathcal{C}_\vee$ are explicitly stated in the following Tables 5.1 - 5.4.

5.1.1 Simple Cardinality Specifications

In this section the case in which c_A and c_B are simple cardinality specifications is considered.

If the query q has the form $i : h \leftarrow A \wedge B$, the cardinality specification $\mathcal{C}_\wedge(c_A, c_B)$ of the cardinality constraint for h with the validity time i is stated in Table 5.1.

c_B \ c_A	$= 0$	$= c$	$\leq a$	$\geq a$
$= 0$	$= 0$	$= 0$	$= 0$	$= 0$
$= d$	$= 0$	$= cd$	$\leq ad$	$\geq ad$
$\leq b$	$= 0$	$\leq cb$	$\leq ab$	$= 0 \vee \geq a$
$\geq b$	$= 0$	$\geq cb$	$= 0 \vee \geq b$	$\geq ab$

Table 5.1: $\mathcal{C}_\wedge(c_A,\, c_B)$ defines the first base case of the cardinality propagation function $\mathcal{C}$: The query body is a conjunction of two atoms the cardinality specifications of which are simple. $a, b \in \mathbb{N}_0,\ c, d \in \mathbb{N}$.

Example 6.
Given $c_A = (= 4)$ and $c_B = (\leq 5)$. Since the body of q is a conjunction of two atoms, we obtain $\mathcal{C}_\wedge(c_A, c_B) = (\leq 4 \cdot 5) = (\leq 20)$ as the cardinality specification of the cardinality constraint for h.

If the query q has the form $i : h \leftarrow A \vee B$, the cardinality specification $\mathcal{C}_\vee(c_A, c_B)$ of the cardinality constraint for h with the validity time i is stated in Table 5.2.

c_B \ c_A	$= a$	$\leq a$	$\geq a$
$= b$	$= a + b$	$\geq b \wedge\ \leq a + b$	$\geq a + b$
$\leq b$	$\geq a \wedge\ \leq a + b$	$\leq a + b$	$\geq a$
$\geq b$	$\geq a + b$	$\geq b$	$\geq a + b$

Table 5.2: $\mathcal{C}_\vee(c_A,\, c_B)$ defines the second base case of the cardinality propagation function $\mathcal{C}$: The query body is a disjunction of two atoms the cardinality specifications of which are simple. $a, b \in \mathbb{N}_0$.

Example 7.
Given $c_A = (= 4)$ and $c_B = (\geq 3)$. Since the body of q is a disjunction of two atoms, we obtain $\mathcal{C}_\vee(c_A, c_B) = (\geq 4 + 3) = (\geq 7)$ as the cardinality specification of the cardinality constraint for h.

All the following cases of the cardinality constraint propagation are based upon these two base cases.

5.1.2 Complex Cardinality Specifications

In this section the case in which c_A and c_B are complex cardinality specifications is considered, i.e. c_A and c_B are a conjunction or a disjuncton of simple cardinality specifications $c_{A_1}, \ldots, c_{A_n}$ and $c_{B_1}, \ldots, c_{B_m}$ respectively.

If the StreamLog query q has the form $i : h \leftarrow A \wedge B$, the cardinality specification $\mathcal{C}_\wedge(c_A, c_B)$ of the cardinality constraint for h with the validity time i is stated in Table 5.3.

c_A c_B	$c_{A_1} \vee \cdots \vee c_{A_n}$	$c_{A_1} \wedge \cdots \wedge c_{A_n}$
$c_{B_1} \vee \cdots \vee c_{B_m}$	$\mathcal{C}_\wedge(c_{A_1}, c_{B_1}) \vee \cdots \vee \mathcal{C}_\wedge(c_{A_n}, c_{B_1}) \vee$ $\vdots \quad \vdots \quad \vdots$ $\mathcal{C}_\wedge(c_{A_1}, c_{B_m}) \vee \cdots \vee \mathcal{C}_\wedge(c_{A_n}, c_{B_m})$	$(\ \mathcal{C}_\wedge(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_1})\ ) \vee$ $\vdots \quad \vdots \quad \vdots \quad \vdots$ $(\ \mathcal{C}_\wedge(c_{A_1}, c_{B_m}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_m})\ )$
$c_{B_1} \wedge \cdots \wedge c_{B_m}$	$(\ \mathcal{C}_\wedge(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_1}, c_{B_m})\ ) \vee$ $\vdots \quad \vdots \quad \vdots \quad \vdots$ $(\ \mathcal{C}_\wedge(c_{A_n}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_m})\ )$	$\mathcal{C}_\wedge(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_1}) \wedge$ $\vdots \quad \vdots \quad \vdots$ $\mathcal{C}_\wedge(c_{A_1}, c_{B_m}) \wedge \cdots \wedge \mathcal{C}_\wedge(c_{A_n}, c_{B_m})$

Table 5.3: $\mathcal{C}_\wedge(c_A,\ c_B)$ defines the first inductive case of the cardinality propagation function $\mathcal{C}$: The query body is a conjunction of two atoms A, B the cardinality specifications c_A, c_B of which are complex, i.e. c_A and c_B are a conjunction or a disjunction of the simple cardinality specifications $c_{A_1}, \ldots, c_{A_n}$ and $c_{B_1}, \ldots, c_{B_m}$ respectively.

Example 8.
Given $c_A = (= 4 \vee \leq 2)$ and $c_B = (\geq 3 \wedge \leq 5)$. Since the body of q is a conjunction of two atoms, we obtain $\mathcal{C}(c_A, c_B) = \mathcal{C}_\wedge(c_A, c_B) = \mathcal{C}_\wedge((= 4 \vee \leq 2), (\geq 3 \wedge \leq 5)) = (\mathcal{C}_\wedge(= 4, \geq 3) \wedge \mathcal{C}_\wedge(= 4, \leq 5)) \vee (\mathcal{C}_\wedge(\leq 2, \geq 3) \wedge \mathcal{C}_\wedge(\leq 2, \leq 5)) = ((\geq 12 \wedge \leq 20) \vee ((= 0 \vee \geq 3) \wedge \leq 10))$ as the cardinality specification of the cardinality constraint for h.

If the StreamLog query q has the form $i : h \leftarrow A \vee B$, the cardinality specification $\mathcal{C}_\vee(c_A, c_B)$ of the cardinality constraint for h with the validity time i is stated in Table 5.4.

Example 9.
Given $c_A = (= 4 \vee \leq 2)$ and $c_B = (\geq 3 \wedge \leq 5)$. Since the body of q is a disjunction of two atoms, we obtain $\mathcal{C}(c_A, c_B) = \mathcal{C}_\vee(c_A, c_B) = \mathcal{C}_\vee((= 4 \vee \leq 2), (\geq 3 \wedge \leq 5)) = (\mathcal{C}_\vee(= 4, \geq 3) \wedge \mathcal{C}_\vee(= 4, \leq 5)) \vee (\mathcal{C}_\vee(\leq 2, \geq 3) \wedge \mathcal{C}_\vee(\leq 2, \leq 5)) = ((\geq 7 \wedge (\geq 4 \wedge \leq 9)) \vee (\geq 3 \wedge \leq 7))$ as the cardinality specification of the cardinality constraint for h.

As can be seen above, the first inductive case (Table 5.3) is led back to the first base case (Table 5.1) and the second inductive case (Table 5.4) is led back to the second base case (Table 5.4).

c_A \ c_B	$c_{A_1} \vee \cdots \vee c_{A_n}$	$c_{A_1} \wedge \cdots \wedge c_{A_n}$
$c_{B_1} \vee \cdots \vee c_{B_m}$	$\mathcal{C}_\vee(c_{A_1}, c_{B_1}) \vee \cdots \vee \mathcal{C}_\vee(c_{A_n}, c_{B_1}) \vee$ $\vdots \quad \vdots \quad \vdots$ $\mathcal{C}_\vee(c_{A_1}, c_{B_m}) \vee \cdots \vee \mathcal{C}_\vee(c_{A_n}, c_{B_m})$	$(\ \mathcal{C}_\vee(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_1})\ ) \vee$ $\vdots \quad \vdots \quad \vdots \quad \vdots$ $(\ \mathcal{C}_\vee(c_{A_1}, c_{B_m}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_m})\ )$
$c_{B_1} \wedge \cdots \wedge c_{B_m}$	$(\ \mathcal{C}_\vee(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_1}, c_{B_m})\ ) \vee$ $\vdots \quad \vdots \quad \vdots \quad \vdots$ $(\ \mathcal{C}_\vee(c_{A_n}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_m})\ )$	$\mathcal{C}_\vee(c_{A_1}, c_{B_1}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_1}) \wedge$ $\vdots \quad \vdots \quad \vdots$ $\mathcal{C}_\vee(c_{A_1}, c_{B_m}) \wedge \cdots \wedge \mathcal{C}_\vee(c_{A_n}, c_{B_m})$

Table 5.4: $\mathcal{C}_\vee(c_A,\, c_B)$ defines the second inductive case of the cardinality propagation function $\mathcal{C}$: The query body is a disjunction of two atoms A, B the cardinality specifications c_A, c_B of which are complex, i.e. c_A and c_B are a conjunction or a disjunction of the simple cardinality specifications $c_{A_1}, \ldots, c_{A_n}$ and $c_{B_1}, \ldots, c_{B_m}$ respectively.

The main idea of inductive cases is to apply the function $\mathcal{C}$ (in particular $\mathcal{C}_\wedge$ or $\mathcal{C}_\vee$) recursively to each (nested) complex cardinality specification occuring in c_A, c_B until simple cardinality specifications are reached such that the base cases can be applied. This idea is generalized and explained in the following algorithms. Tables 5.3 - 5.4 present special cases of these algorithms.

Algorithm for $\mathcal{C}_\wedge(c_A, c_B)$:

Step 1 (Case Distinction):

- ▶ If c_A is complex cardinality specification and c_B is simple cardinality specification, then, continue with Step 2.
- ▶ If c_A is simple cardinality specification and c_B is complex cardinality specification, then continue with Step 3.
- ▶ If c_A and c_B are both complex cardinality specifications, then continue with Step 4.
- ▶ Otherwise, c_A, and c_B are both simple cardinality specifications, hence continue with Step 5.

Step 2:

- ▶ Let c_{A_1}, c_{A_2} be the adjuncts (i.e. conjuncts or disjuncts) of c_A, i.e. $c_A = c_{A_1} \circ c_{A_2}$ where $\circ \in \{\wedge, \vee\}$. Apply the first inductive case (Table 5.3) to $\mathcal{C}_\wedge(c_A, c_B) = \mathcal{C}_\wedge(c_{A_1} \circ c_{A_2}, c_B)$ and obtain $(\mathcal{C}_\wedge(c_{A_1}, c_B) \circ \mathcal{C}_\wedge(c_{A_2}, c_B))$.
- ▶ Apply the algorithm to $\mathcal{C}_\wedge(c_{A_1}, c_B)$ and $\mathcal{C}_\wedge(c_{A_2}, c_B)$.

Step 3:

- ▶ Let c_{B_1}, c_{B_2} be the adjuncts of c_B, i.e. $c_B = c_{B_1} \circ c_{B_2}$ where $\circ \in \{\wedge, \vee\}$. Apply the first inductive case (Table 5.3) to $\mathcal{C}_\wedge(c_A, c_B) = \mathcal{C}_\wedge(c_A, c_{B_1} \circ c_{B_2})$ and obtain $(\mathcal{C}_\wedge(c_A, c_{B_1}) \circ \mathcal{C}_\wedge(c_A, c_{B_2}))$.

- ▶ Apply the algorithm to $\mathcal{C}_\wedge(c_A, c_{B_1})$ and $\mathcal{C}_\wedge(c_A, c_{B_2})$.

Step 4:

- ▶ Let c_{A_1}, c_{A_2} be the adjuncts of c_A, and c_{B_1}, c_{B_2} be the adjuncts of c_B (i.e. $c_A = c_{A_1} \circ c_{A_2}$ and $c_B = c_{B_1} \circ c_{B_2}$ where $\circ \in \{\wedge, \vee\}$). Apply the first inductive case (Table 5.3) to $\mathcal{C}_\wedge(c_A, c_B) = \mathcal{C}_\wedge(c_{A_1} \circ c_{A_2}, c_{B_1} \circ c_{B_2})$ and obtain $(\mathcal{C}_\wedge(c_{A_1}, c_{B_1}) \circ \mathcal{C}_\wedge(c_{A_1}, c_{B_2})) \circ (\mathcal{C}_\wedge(c_{A_2}, c_{B_1}) \circ \mathcal{C}_\wedge(c_{A_2}, c_{B_2}))$.

- ▶ Apply the algorithm to $\mathcal{C}_\wedge(c_{A_1}, c_{B_1})$, $\mathcal{C}_\wedge(c_{A_1}, c_{B_2})$, $\mathcal{C}_\wedge(c_{A_2}, c_{B_1})$ and $\mathcal{C}_\wedge(c_{A_2}, c_{B_2})$.

Step 5:

- ▶ Apply the first base case (Table 5.1).

Algorithm for $\mathcal{C}_\vee(c_A, c_B)$ is similar to the algorithm for $\mathcal{C}_\wedge(c_A, c_B)$. The only difference is that we use Tables 5.2 and 5.4 instead of Tables 5.1 and 5.3 respectively, and the function $\mathcal{C}_\vee$ instead of the function $\mathcal{C}_\wedge$.

The following example illustrates the mentioned algorithms.

Example 10.
Given the StreamLog query q of the form $i : h \leftarrow A \wedge B$ and the cardinality specifications $c_A = ((= 4 \vee \leq 2) \wedge \leq 1)$, $c_B = ((\geq 3 \wedge \leq 5) \vee (\geq 1 \wedge \leq 6))$ of the cardinality constraints for atoms A, B within the validity time i.

In order to derive the cardinality of h, the following computation steps are made:

$$
\begin{aligned}
\mathcal{C}(c_A, c_B) &\overset{(1)}{=} \mathcal{C}_\wedge(c_A, c_B) \\
&= \mathcal{C}_\wedge(((= 4 \vee \leq 2) \wedge \leq 1), ((\geq 3 \wedge \leq 5) \vee (\geq 1 \wedge \leq 6))) \\
&\overset{(2)}{=} (\mathcal{C}_\wedge((= 4 \vee \leq 2), (\geq 3 \wedge \leq 5)) \wedge \mathcal{C}_\wedge(\leq 1, (\geq 3 \wedge \leq 5))) \vee \\
&\quad\ (\mathcal{C}_\wedge((= 4 \vee \leq 2), (\geq 1 \wedge \leq 6)) \wedge \mathcal{C}_\wedge(\leq 1, (\geq 1 \wedge \leq 6))) \\
&\overset{(3)}{=} (((\mathcal{C}_\wedge(= 4, \geq 3) \wedge \mathcal{C}_\wedge(= 4, \leq 5)) \vee (\mathcal{C}_\wedge(\leq 2, \geq 3) \wedge \mathcal{C}_\wedge(\leq 2, \leq 5))) \wedge \\
&\quad\ (\mathcal{C}_\wedge(\leq 1, \geq 3) \wedge \mathcal{C}_\wedge(\leq 1, \leq 5))) \vee \\
&\quad\ (((\mathcal{C}_\wedge(= 4, \geq 1) \wedge \mathcal{C}_\wedge(= 4, \leq 6)) \vee (\mathcal{C}_\wedge(\leq 2, \geq 1) \wedge \mathcal{C}_\wedge(\leq 2, \leq 6))) \wedge \\
&\quad\ (\mathcal{C}_\wedge(\leq 1, \geq 1) \wedge \mathcal{C}_\wedge(\leq 1, \leq 6))) \\
&\overset{(4)}{=} ((((\geq 12 \wedge \leq 20) \vee ((= 0 \vee \geq 3) \wedge \leq 10)) \wedge ((= 0 \vee \geq 3) \wedge \leq 5)) \vee \\
&\quad\ (((\geq 4 \wedge \leq 24) \vee ((= 0 \vee \geq 1) \wedge \leq 12)) \wedge ((= 0 \vee \geq 1) \wedge \leq 6)))
\end{aligned}
$$

(1) $\mathcal{C}$ is converted to $\mathcal{C}_\wedge$, since the body of q is a conjunction of atoms A and B.

(2) c_A is a conjunction of cardinality specifications $(= 4 \vee \leq 2)$ and (≤ 1).

c_B is a disjunction of cardinality specifications $(\geq 3 \wedge \leq 5)$ and $(\geq 1 \wedge \leq 6)$.

Hence, *Step 4* in the algorithm for $\mathcal{C}_\wedge(c_A, c_B)$ is applied.

(3) The function $\mathcal{C}_\wedge$ and the inductive case (Table 5.3) are recursively applied to each (nested) complex cardinality specification until simple cardinality specifications are reached and remain as (first and second) parameters of each $\mathcal{C}_\wedge$.

(4) The first base case (Table 5.1) is applied according to *Step 5* in the algorithm for $\mathcal{C}_\wedge(c_A, c_B)$.

The correctness of the algorithms for $\mathcal{C}_\wedge$ and $\mathcal{C}_\vee$ can be proven by induction over the structure of c_A and c_B (similar to the proof of Theorem 6.1.4). The proof is skipped in this report for the sake of brevity.

In the following, we investigate the termination and complexity of the algorithms described above. To achieve this purpose, we introduce the definitions of the following three notions:

1) The set of all simple cardinality specifications of a cardinality specification (Definition 5.1).

2) The structure of a cardinality specification (Definition 5.2).

3) The number of recursive calls of a function (Definition 5.3).

Definition 5.1 (Set of simple cardinality specifications).
Let c be a cardinality specification. We define

i) $Simple(c)$ as the set of all simple cardinality specifications occuring in c.

ii) $|Simple(c)|$ as the cardinal number of $Simple(c)$ (i.e. $|Simple(c)|$ states the number of elements in $Simple(c)$).

Definition 5.2 (Structure of a cardinality specification).
The structure $str(c)$ of a cardinality specification c is presented by its binary parse tree, independently of the content of the nodes of the tree.

Example 11.
Let $c_1 := ((\geq 1 \wedge \leq 4) \vee (= 3 \vee \geq 8))$, $c_2 = (((= 2 \vee \leq 4) \wedge (\leq 3)) \vee (\geq 8))$ and $c_3 := ((\geq 3 \wedge \leq 6) \vee (= 1 \vee \geq 4))$ be complex cardinality specifications. Figures 5.1 - 5.3 below show their structures.

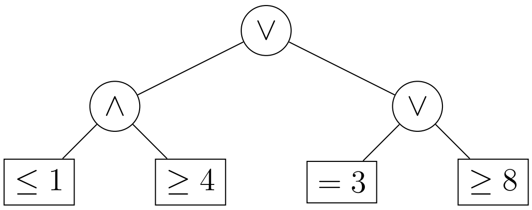


Figure 5.1: Binary parse tree of c_1.

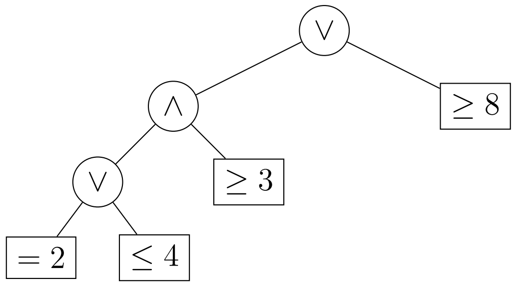


Figure 5.2: Binary parse tree of c_2.

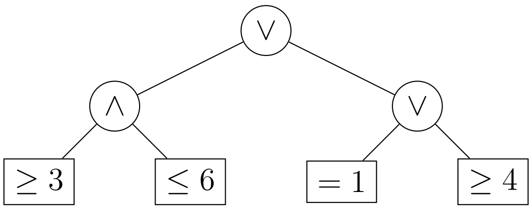


Figure 5.3: Binary parse tree of c_3.

That is clear to see that $str(c_1) = str(c_3)$ (we say c_1 and c_3 have the same structure), but $str(c_1) \neq str(c_2)$ and $str(c_3) \neq str(c_2)$.

Figure 5.4 shows the structure of c_1 and c_3.

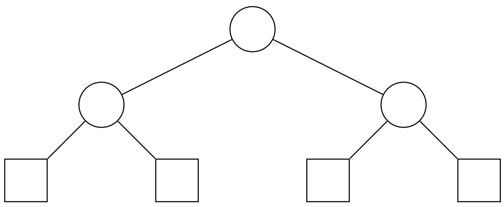


Figure 5.4: Structure of c_1 and c_3.

Definition 5.3.
Let *Functions* denote the set of (recursive) functions. Then

$$\begin{array}{rcccl} recursiveCalls & : & Functions & \rightarrow & \mathbb{N}_0 \\ & & f & \mapsto & recursiveCalls(f(x_1, x_2, \dots)) \end{array}$$

yields the number of all (recursive) function calls of a function f for the inputs $x_1, x_2, \dots$

Remark 5.1.1.
Consider the algorithms for $\mathcal{C}_\wedge(c_A, c_B)$ and $\mathcal{C}_\vee(c_A, c_B)$ where c_A and c_B are cardinality specifications, and Example 10 illustrating them. The functions $\mathcal{C}_\wedge$ and $\mathcal{C}_\vee$ are recursively

applied to each (nested) complex adjunct of c_A and c_B until simple adjuncts (i.e. simple cardinality specifications) are reached. In other words, the number of recursive calls of $\mathcal{C}_\wedge$ and $\mathcal{C}_\vee$ depends on the number of nested complex adjuncts within c_A and c_B. Furthermore, the number of nested complex adjuncts within a cardinality specification c is always $|Simple(c)| - 1$ (that also corresponds to the number of operators like $\wedge$ or $\vee$ in c). Hence, for cardinality specifications c_A, c'_A and c_B with $|Simple(c_A)| = |Simple(c'_A)|$ (and therefore with the same number of nested complex adjuncts), it holds that

$$recursiveCalls(c_A, c_B) = recursiveCalls(c'_A, c_B)$$

Theorem 5.1.1.
Let c_A be a simple cardinality specification and c_B be a (complex) cardinality specification. The equation $recursiveCalls(\mathcal{C}(c_A, c_B)) = 2 \cdot |Simple(c_B)| - 1$ holds.

Proof. (By induction over the number n of the simple cardinaliy specifications in c_B)

c_A is a simple cardinality specification, thus $|Simple(c_A)| = 1$.

- *Base Case*:
  - c_B is a simple cardinality specification (i.e. $|Simple(c_B)| = 1$) : $recursiveCalls(\mathcal{C}(c_A, c_B)) = 1$ is trivial, since $\mathcal{C}(c_A, c_B)$ is called only once.
  - $c_B = (c_{B_1} \circ c_{B_2})$ where c_{B_1}, c_{B_2} are simple cardinality specifications and $\circ \in \{\wedge, \vee\}$, $|Simple(c_B)| = 2$.

    First, $\mathcal{C}(c_A, c_B) = \mathcal{C}(c_A, c_{B_1} \circ c_{B_2})$ is called once. Afterwards, the inductive cases (Table 5.3 and Table 5.4) are applied and we obtain $\mathcal{C}(c_A, c_{B_1} \circ c_{B_2}) = (\mathcal{C}(c_A, c_{B_1}) \circ \mathcal{C}(c_A, c_{B_2}))$. Thereby the function $\mathcal{C}$ is still recursively called twice. This yields altogether 3 function calls and $recursiveCalls(\mathcal{C}(c_A, c_B)) = 3 = 2 \cdot 2 - 1 = 2 \cdot |Simple(c_B)| - 1$.

- *Induction Assumption*:

  It holds that $recursiveCalls(\mathcal{C}(c_A, c_B)) = 2 \cdot n - 1$, for every cardinality specification c_A, c_B with $|Simple(c_A)| = 1$ and $|Simple(c_B)| = n$ where $n \in \mathbb{N}$.

- *Inductive Step*: $n \mapsto n + 1$

  Because of Remark 5.1.1, c_B can have any structures such that $|Simple(c_B)| = n+1$. This has no effect to the rightness of the statement. Without restriction of generality, we assume that c_B has the structure $c_B = (c'_B \circ c_{B_{n+1}})$ where c'_B is a (complex) cardinality specification consisting of n simple cardinality specifications (i.e. $|Simple(c'_B)| = n$), and $c_{B_{n+1}}$ is a simple cardinality specification. First, the function $\mathcal{C}(c_A, c_B)$ is called once. The application of the inductive cases (Tables 5.3 and 5.4) to it yields $\mathcal{C}(c_A, c_B) = (\mathcal{C}(c_A, c'_B) \circ \mathcal{C}(c_A, c_{B_{n+1}}))$. According to the induction assumption, $2n-1$ recursive calls of $\mathcal{C}(c_A, c'_B)$ are required, i.e. $recursiveCalls(\mathcal{C}(c_A, c'_B)) = 2n-1$. And the function $\mathcal{C}(c_A, c_{B_{n+1}})$ is called only once, i.e. $recursiveCalls(\mathcal{C}(c_A, c_{Bn+1})) = 1$, because $c_{B_{n+1}}$ is simple.

There are altogether $1 + (2n-1) + 1 = 2n + 1 = 2(n+1) - 1 = 2\,|Simple(c_B)| - 1$ recursive functions calls for $\mathcal{C}(c_A, c_B)$.

□

Theorem 5.1.2.
Let c_A, c_B be (complex) cardinality specifications with $n := |Simple(c_A)| = |Simple(c_B)|$, for $n \geq 1$. Then $\quad recursiveCalls(\mathcal{C}(c_A, c_B)) = 2n^2 - 2n + 1 \quad$ holds.

Proof. (By induction over n)

- *Base Case*: $n = 1$

  Both c_A and c_B are simple cardinality specifications and $\mathcal{C}(c_A, c_B)$ is called only once, thus $recursiveCalls(\mathcal{C}(c_A, c_B)) = 1 = 2 \cdot 1^2 - 2 \cdot 1 + 1$

- *Induction Assumption*:

  $recursiveCalls(\mathcal{C}(c_A, c_B)) = 2n^2 - 2n + 1$ holds for all cardinality specifications c_A, c_B with $n := |Simple(c_A)| = |Simple(c_B)|$, for $n \geq 1$.

- *Inductive Step*: $n \mapsto n+1$

  According to Remark 5.1.1, c_A and c_B can have arbitrary structure such that $|Simple(c_A)| = |Simple(c_B)| = n+1$. Without restriction of generality, we assume that $c_A = (c'_A \circ c_{A_{n+1}})$ and $c_B = (c'_B \circ c_{B_{n+1}})$ where c'_A, c'_B consist of n simple cardinality specifications (i.e. $|Simple(c'_A)| = |Simple(c'_B)| = n$), and $c_{A_{n+1}}$, $c_{B_{n+1}}$ are simple cardinality specifications. First, the function $\mathcal{C}(c_A, c_B)$ is called once. The application of inductive cases (Tables 5.3 and 5.4) to it yields $\mathcal{C}(c_A, c_B) = (\mathcal{C}(c'_A, c'_B) \circ \mathcal{C}(c'_A, c_{B_{n+1}}) \circ \mathcal{C}(c_{A_{n+1}}, c_{B'}) \circ \mathcal{C}(c_{A_{n+1}}, c_{B_{n+1}}))$. According to the assumption, $recursiveCalls(\mathcal{C}(c'_A, c'_B)) = 2n^2 - 2n + 1$ holds. According to Theorem 5.1.1, $recursiveCalls(\mathcal{C}(c'_A, c_{B_{n+1}})) = 2n - 1$ holds, since $c_{B_{n+1}}$ is simple. Analogously, $recursiveCalls(\mathcal{C}(c_{A_{n+1}}, c_{B'})) = 2n - 1$ holds as well, according to Theorem 5.1.1 since $c_{A_{n+1}}$ is simple. Finally, the function $\mathcal{C}(c_{A_{n+1}}, c_{B_{n+1}})$ is called once, because $c_{A_{n+1}}$ and $c_{B_{n+1}}$ are both simple. Hence, there are altogether

  $$1 + (2n^2 - 2n + 1) + (2n-1) + (2n-1) + 1 = 2n^2 + 2n + 1 = 2(n+1)^2 - 2(n+1) + 1$$

recursive function calls for $\mathcal{C}(c_A, c_B)$.

□

Theorem 5.1.3 (Complexity of $\mathcal{C}_\wedge$ and $\mathcal{C}_\vee$).
Let c_A, c_B be cardinality specifications and $n := max\{|Simple(c_A)|, |Simple(c_B)|\}$. The complexity of the algorithms for $\mathcal{C}_\wedge(c_A, c_B)$ and $\mathcal{C}_\vee(c_A, c_B)$ is $\mathcal{O}(n^2)$.

Proof. The complexity of $\mathcal{C}_\wedge(c_A, c_B)$ and $\mathcal{C}_\vee(c_A, c_B)$ depends on the number of its recursive calls, which in tern depends on the number of simple cardinality specifications in c_A and c_B. Without restriction of generality, we assume that $|Simple(c_A)| \leq |Simple(c_B)| =: n$. Then it holds that

$$recursiveCalls(\mathcal{C}(c_A, c_B)) \leq recursiveCalls(\mathcal{C}(c_B, c_B)) \overset{(\text{Th.5.1.2})}{=} 2n^2 - 2n + 1 \in \mathcal{O}(n^2)$$

□

Theorem 5.1.4.
Let c_A, c_B be cardinality specifications. It holds that

$$|Simple(c_A)| \cdot |Simple(c_B)| \leq |Simple(\mathcal{C}(c_A, c_B))| \leq 2 \cdot |Simple(c_A)| \cdot |Simple(c_B)|$$

Proof. Remember that $\mathcal{C}$ (which is led back to $\mathcal{C}_\wedge$ or $\mathcal{C}_\vee$) is applied to each pair of nested (complex) cardinality specifications in c_A and c_B, then to the result of two such applications and so on until simple cardinality specifications are reached. Each simple cardinality specification c_{A_i} in $Simple(c_A)$ is associated with each simple cardinality specification c_{B_j} in $Simple(c_B)$ through $\mathcal{C}$ exactly once, i.e. each $\mathcal{C}(c_{A_i}, c_{B_j})$ occurs once in the equation, for $1 \leq i \leq |Simple(c_A)|$ and $1 \leq j \leq |Simple(c_B)|$. Thus, the number of occurrences of all $\mathcal{C}(c_{A_i}, c_{B_j})$ in the equation is $|Simple(c_A)| \cdot |Simple(c_B)|$. The number of simple cardinality specifications in the cardinality specification returned by $\mathcal{C}(c_A, c_B)$ is $|Simple(c_A)| \cdot |Simple(c_B)| \cdot \left|Simple(\mathcal{C}(c_{A_i}, c_{B_j}))\right|$ for all i and j. Furthermore, $1 \leq \left|Simple(\mathcal{C}(c_{A_i}, c_{B_j}))\right| \leq 2$, compare Table 5.1 and Table 5.2. Therefore, $|Simple(c_A)| \cdot |Simple(c_B)| \leq |Simple(\mathcal{C}(c_A, c_B))| \leq 2 \cdot |Simple(c_A)| \cdot |Simple(c_B)|$ holds. □

5.2 Arbitrary Number of Atoms in a Query Body

Let q be a StreamLog query of the form $i : h \leftarrow b$ where i is a time interval, h is an atom and the query head and b is the query body. Furthermore, b is an arbitrary nested conjunction or disjunction of atoms $A_1, .., A_n$, for $n \in \mathbb{N}$. Let $c_1, .., c_n$ be the cardinality specifications of the cardinality constraints for $A_1, .., A_n$ respectively with the validity time i.

In order to propagate the cardinaltiy constraints with respect to the query q, the function $\mathcal{C}$ as defined in Table 5.1 and Table 5.2 is applied to each pair of the cardinality specifications of the cardinality constraints for the atoms $A_1, \ldots, A_n$ in the query body b, then to the result of two such applications and so on until the cardinality specifications of the constraints for all atoms in b are considered.

The algorithm in Listing 5.1 implements this idea. It takes the query body b as input, accesses the cardinality specifications of the constraints for the atoms in b and derives the cardinality specification of the constraint for h. This algorithm is based upon Table 5.1 until Table 5.4.

```
propagatesCardSpec(query_body) {
   if query_body is a conjunction then
      subqueries ← getConjuncts(query_body);
      cardSpecLeft ← propagatesCardSpec(subqueries[0]);
      cardSpecRight ← propagatesCardSpec(subCardSpecs[1]);
      cardSpec ← C∧(cardSpecLeft,cardSpecRight);
   else if query_body is a disjunction then
      subqueries ← getDisjuncts(query_body);
      cardSpecLeft ← propagatesCardSpec(subqueries[0]);
      cardSpecRight ← propagatesCardSpec(subCardSpecs[1]);
      cardSpec ← C∨(cardSpecLeft,cardSpecRight);
   else /* query_body is an atom*/
      cardSpec ← getCardSpec(query_body);
   end end
   return cardSpec;
}
```

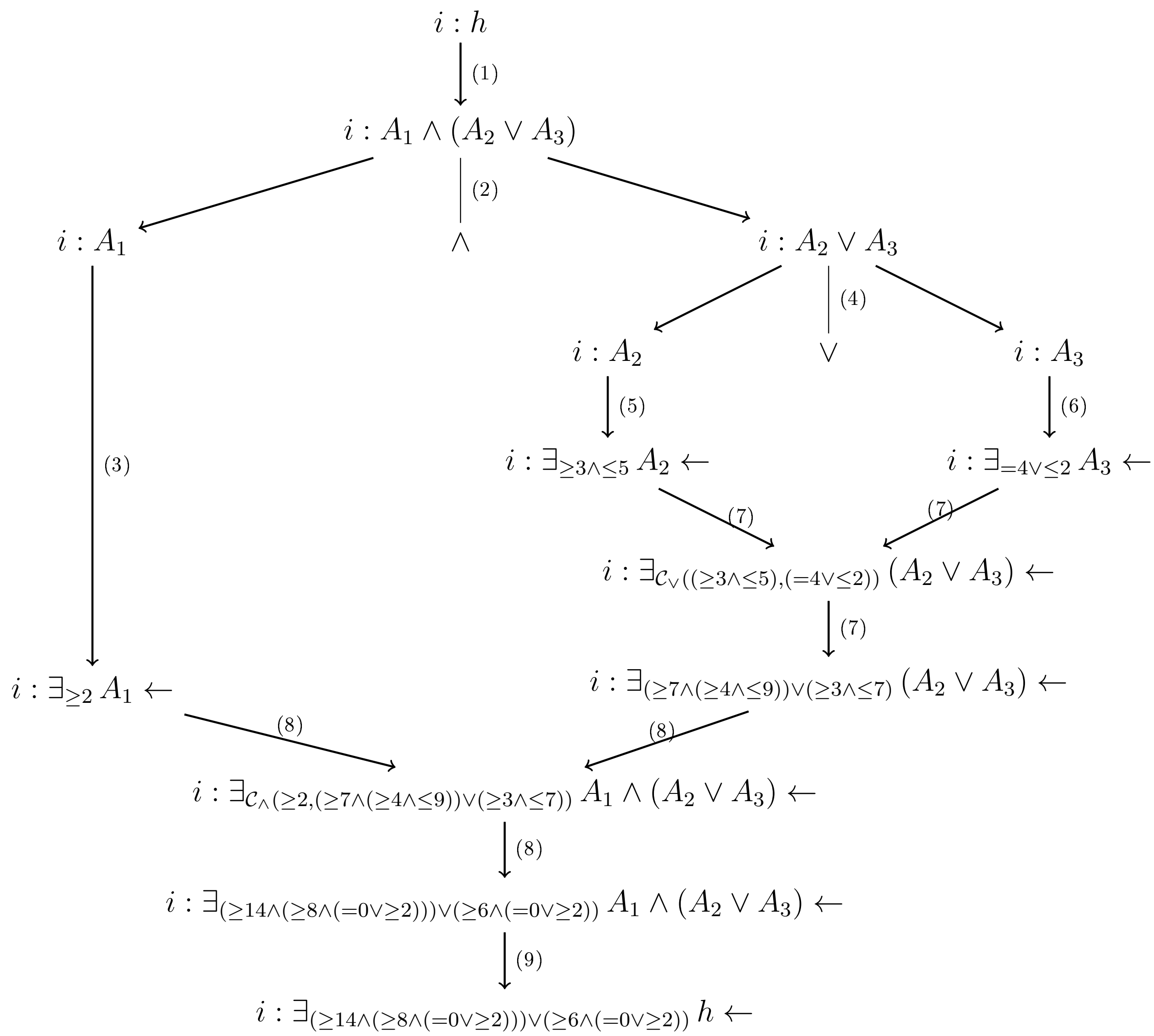
Listing 5.1: The function *propagateCardSpec* propagates the cardinality constraints with respect to the query the body of which is an arbitrarily nested conjunction or disjunction of atoms.

The function *getConjuncts* accepts only a conjunction of cardinality specifications as parameter and returns an array of size two the elements of which are the conjuncts. For example, given the cardinality specification $c := ((c_1 \land c_2) \land c_3)$, where c_1, c_2, c_3 are the simple cardinality specifications, so *getConjuncts*(c) yields the two-elements array $\{(c_1 \land c_2),\ c_3\}$. The function *getDisjuncts* processes analogously. For example, given the cardinality specification $d := ((d_1 \land d_2) \lor (d_3 \land d_4))$ where $d_1, \ldots, d_4$ are the simple cardinality specifications, then $getDisjuncts(d)$ yields the array $\{(d_1 \wedge d_2), (d_3 \wedge d_4)\}$.

The funciton $getCardSpec$ takes an atom as input and returns the cardinality specification of the constraint for this atom.

For better understanding how the algorithm works, we consider the following example.

Example 12.
Let q be the StreamLog query of the form $i : h \leftarrow (A_1 \wedge (A_2 \vee A_3))$ where i is a time interval and h, A_1, A_2, A_3 are atoms. Let $c_{A_1} := (\geq 2)$, $c_{A_2} := (\geq 3 \wedge \leq 5)$, $c_{A_3} := (= 4 \vee \leq 2)$ be the cardinality specifications of the constraints for the atoms A_1, A_2, A_3 respectively. The following flowchart shows the processing steps applied to q to receive the cardinality specification for h.



1. $i : h$ is led back to $i : A_1 \wedge (A_2 \vee A_3)$ because of q. (This is not part of the algorithm in Listing 5.1.)

2. $i : A_1 \wedge (A_2 \vee A_3)$ is split into two conjuncts $i : A_1$ and $i : (A_2 \vee A_3)$. This corresponds to Lines $2 - 5$ in Listing 5.1.

3. The algorithm is recursively applied to the first conjunct $i : A_1$. Since A_1 is an atom, its cardinality constraint $i : \exists_{\geq 2} A_1 \leftarrow$ is considered. This corresponds to Line $12 - 13$ in Listing 5.1.

4. The algorithm is recursively applied to the second conjunct $i : (A_2 \vee A_3)$. Since $(A_2 \vee A_3)$ is a disjunction of atoms, it is split into its disjuncts $i : A_2$ and $i : A_3$. This corresponds to Lines $7 - 10$ in Listing 5.1.

5. The algorithm is recursively applied to the disjunct $i : A_2$. Since A_2 is an atom, its cardinality constrraint $i : \exists_{\geq 3 \wedge \leq 5} A_2 \leftarrow$ is considered. This corresponds to Lines $12 - 13$ in Listing 5.1.

6. The algorithm is recursively applied to the disjunct $i : A_3$. Since A_3 is an atom, its cardinality constraint $i : \exists_{=4 \vee \leq 2} A_3 \leftarrow$ is considered. This corresponds to Lines $12 - 13$ in Listing 5.1.

7. Since A_2 and A_3 are disjuncts (compare Step 4), $\mathcal{C}_\vee$ is applied to the cardinality specifications of the constraints for A_2 and A_3 gained in Steps 5 and 6 respectively. The result is $i : \exists_{(\geq 7 \wedge (\geq 4 \wedge \leq 9)) \vee (\geq 3 \wedge \leq 7)} A_2 \vee A_3 \leftarrow$. This corresponds to Line 11 in Listing 5.1.

8. Since A_1 and $(A_1 \vee A_3)$ are conjuncts (compare Step 2), $\mathcal{C}_\wedge$ is applied to the cardinality specifications of the constraints for A_1 and $(A_2 \vee A_3)$ obtained in Steps 3 and 7 respectively. The result is $i : \exists_{(\geq 14 \wedge (\geq 8 \wedge (=0 \vee \geq 2))) \vee (\geq 6 \wedge (=0 \vee \geq 2))} A_1 \wedge (A_2 \vee A_3) \leftarrow$. This corresponds to Line 6 in Listing 5.1.

9. $i : \exists_{(\geq 14 \wedge (\geq 8 \wedge (=0 \vee \geq 2))) \vee (\geq 6 \wedge (=0 \vee \geq 2))} h \leftarrow$ is the derived cardinality constraint. Compare Line 15 of the algorithm. Moreover, it can be simplied to $i : \exists_{\geq 6} h \leftarrow$ according to the simplification rules in Chapter 7, but this is not part of the algorithm in Listing 5.1.

Chapter 6

Constraint Validity Time Differs from Query Evaluation Time

Let q be a StreamLog query of the form $i : h \leftarrow b$ where i is the evaluation time interval, h is the head and b is the body of the query. If b matches the stream during time interval i, then a new event (state) is derived as specified by h.

Let $j : \exists_c l \leftarrow$ be an ESCL cardinality constraint where j is the validity time interval, c is the cardinality specification and l is the atom of the constraint, i.e. there are c events (states) matching l during j.

In order to propagate the cardinality constraint for the above query q, we assume that l is one of the atoms in b (compare Chapter 4). Both i and j can be tumlbing or sliding windows or time intervals. These time intervals can be repeating or non-repeating, user-defined, now windows, simple or complex events, relative timer events, or application states (see [5] for definitions of these notions). The bounds of i and j have to be known, since otherwise the constraint propagation is not possible.

In the course of this section, the cardinality propagation function $\mathcal{C}$ is expanded by new cases in which the query evaluation time i and the constraint validity time j are not the same.

6.1 One Atom in a Query Body

This section is devoted to the base case in which the cardinality constraints are propagated with respect to a query the body of which is a single atom l.

In the following, we first consider the case in which the cardinality specification c of the contraint for l is simple. Afterwards, the more common case in which c is complex will be considered. In both cases, we will specify the rules and algorithms for the propagation of the cardinality constraints with respect to the query q.

In order to propagate cardinality constraints, it is necessary to compare the duration of the query evaluation time i to the constraint validity time j. The duration of a time interval is accurately expressed by the asolute-value bars. For example, $|i|$ denotes the duration of i, $|i| < |j|$ indicates that the duration of i is shorter than that of j. But for the sake of readability, we write i and j (without the absolute-value bars) when we mean the duration of the respective time intervals, i.e. $i < j$ instead of $|i| < |j|$.

6.1.1 Simple Cardinality Specifications

The cardinality specification c is simple. The following cases are to be considered:

Case 1. i and j are tumbling windows :

(a) $i < j$:

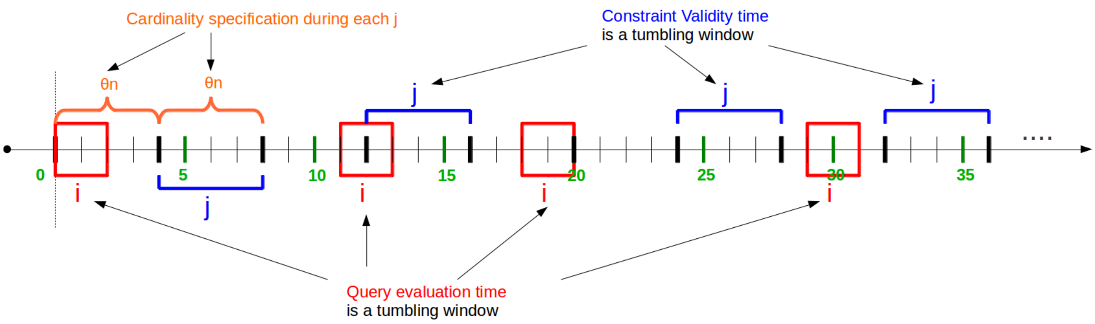


Figure 6.1: i and j are tumbling windows, $i < j$

i. $c = (= n)$:
The lower bound for the derived cardinality specification is 0, since, e.g., there may be no event matching l within the time interval $[0, 2)$ (compare Figure 6.1). The upper bound for the derived cardinality specification is $2n$, since, e.g., there may be n events matching l in each of the time intervals $[11, 12)$ and $[12, 13)$. Hence, the derived cardinality constraint for the atom derived by q during evaluation time i is $i : \exists_{\leq 2n}\, h \leftarrow$.

ii. $c = (\leq n)$:
By using the same argument as above, we obtain the cardinality constraint $i : \exists_{\leq 2n}\, h \leftarrow$.

iii. $c = (\geq n)$:

During the time interval $[0, 2)$ there may be no, less than, more than, or exactly n events matching l (compare Figure 6.1). Hence, 0 is the lower bound for the derived cardinality specification. Because of $c = (\geq n)$, it has no upper bound. Therefore, the derived cardinality constraint is $i : \exists_{\geq 0}\, h \leftarrow$.

(b) $i \geq j$:

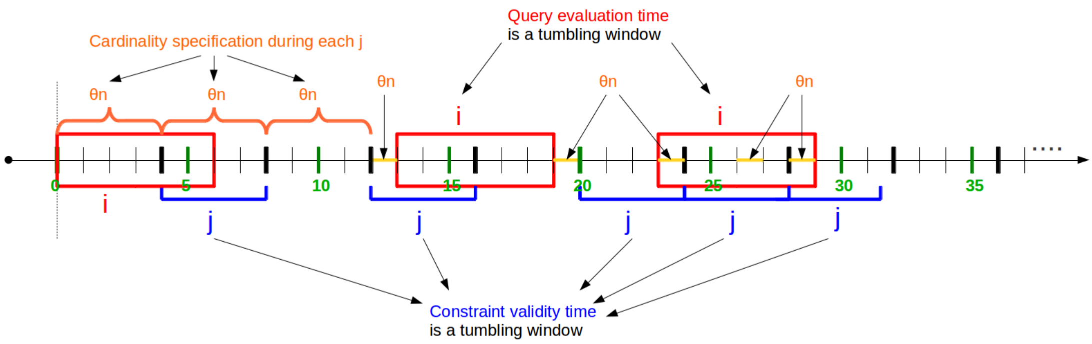


Figure 6.2: i and j are tumbling windows, $i \geq j$

i. $c = (= n)$:

The lower bound for the derived cardinality specification is $\left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$, since, for example, there can be exactly n events matching l in each of the time intervals $[12, 13)$ and $[19, 20)$ and no such event within the time interval $[13, 19)$ (consider Figure 6.2). So the cardinality specification $0 = \left(\lfloor \frac{6}{4} \rfloor - 1\right) \cdot n$ is yielded. Note that i is 6 and j is 4 units long in this example.

The upper bound for the derived cardinality specification is $\left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$, since, for example, there can be exactly n events matching l in each of the intervals $[23, 24)$, $[24, 28)$ and $[28, 29)$ (consider Figure 6.2). This yields $3n = \left(\lceil \frac{6}{4} \rceil + 1\right) \cdot n$.

Hence, the derived cardinality constraint is $i : \exists_w \leftarrow h$ where $w := \left(\geq \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n \wedge \leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n\right)$.

ii. $c = (\leq n)$:

Because of $c = (\leq n)$, there can be no event matching l during i. Thus, 0 is the lower bound of the derived cardinality specification. Its upper bound is $\left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$, since, e.g., there can be n events matching l in each of the time intervals $[23, 24)$, $[24, 28)$ and $[28, 29)$ as above. Altogether, the derived cardinality constraint is $i : \exists_w\, h \leftarrow$ where $w := \left(\leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n\right)$.

iii. $c = (\geq n)$:

Because of $c = (\geq n)$, the derived cardinality specification is not bounded above. Its lower bound is $\left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$, since, e.g., there may be no event matching l during the time interval $[13, 19)$ (consider Figure 6.2), i.e. $0 = \left(\lfloor\frac{6}{4}\rfloor - 1\right) \cdot n = \left(\lfloor\frac{i}{j}\rfloor - 1\right) \cdot n$. Hence, the derived cardinality constraint is $i : \exists_w h \leftarrow$ where $w := \left(\geq \left(\lfloor\frac{i}{j}\rfloor - 1\right) \cdot n\right)$.

Case 2. i is tumbling window and j is sliding window :

(a) $i < j$:

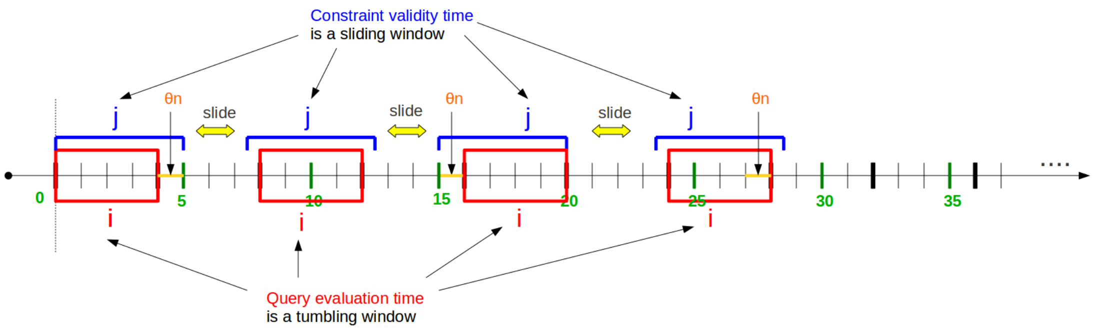


Figure 6.3: i is tumbling window and j is sliding windows, $i < j$. Each tumbling window i is completely within one sliding window j

i. $c = (= n)$:
Since $i < j$, the sliding window j can be shifted (moved) on the stream such that each tumbling window i is completely included within j. The lower bound for the derived cardinality constraint is 0, since, e.g., there may be n events matching l only during the time interval $[15, 16)$ and no event matching l during the interval $[16, 20)$ (consider Figure 6.3). Its upper bound is n, since, e.g., there may be exactly n events matching l within interval $[27, 28)$. Altogether, the derived cardinality constraint is $i : \exists_{\leq n} h \leftarrow$.

ii. $c = (\leq n)$:
This case is analogously argued as the previous one i. Hence, the derived cardinality constraint is $i : \exists_{\leq n} h \leftarrow$.
However, in the above cases i. and ii., if there is some i which is not completely included in one j, but overlaps two adjacent intervals j, then in both cases the cardinality constraint $i : \exists_{\leq 2n} h \leftarrow$ is derived, analogously to cases 1.(a)i. and 1.(a)ii.

iii. $c = (\geq n)$:
Let us take a look at Figure 6.3 again. The lower bound for the derived cardinality constraint 0, since, e.g., there may be no events matching l during the time interval $[0, 4)$. Furthermore, if there are (more than) n events matching l exactly from the point 27 of the time interval $[27, 28)$, then there can be $0..n-1$ such events during the time interval $[\frac{47}{2}, 27)$. And since $c = (\geq n)$,

the cardinality specification of the constraint is not bounded above. Hence, the derived cardinality constraint is $i : \exists_{\geq 0} h \leftarrow$.

(b) $i \geq j$:

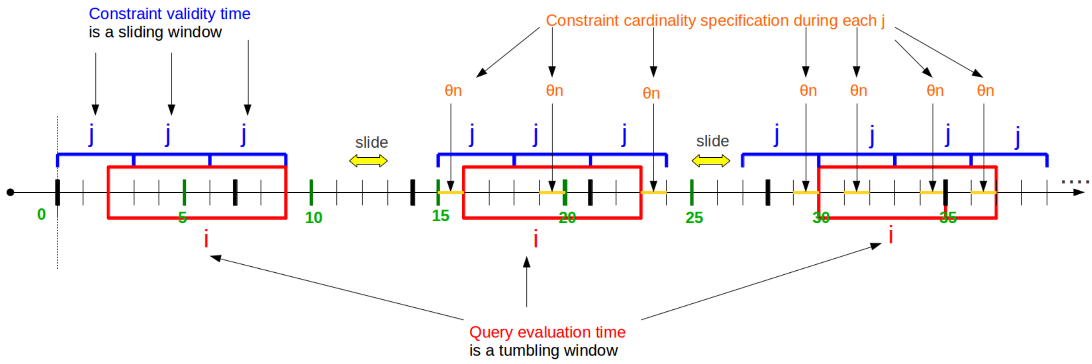


Figure 6.4: i is tumbling window and j is sliding windows, $i \geq j$

i. $c = (= n)$:
The lower bound for the derived cardinality specification is $\left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$, since, e.g., there may be n events matching l in each of the time intervals $[15, 16)$, $[19, 20)$ and $[23, 24)$, but no such events in the intervals $[16, 18)$ and $[21, 23)$ (consider Figure 6.4). Thus, there are (at least) $n = \left(\lfloor \frac{7}{3} \rfloor - 1\right) \cdot n = \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$ events matching l during $i = [16, 23)$. Note that i is 7 and j is 3 units long in this example.
If there are n events matching l in each of the intervals $[29, 30]$, $[31, 32)$, $[34, 35)$ and $[36, 37)$, then there exists $4n = \left(\lceil \frac{7}{3} \rceil + 1\right) \cdot n = \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$ such events during interval $[30, 37)$. Hence, the upper bound of the derived cardinality specification is $\left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$. Here, we take advantage of the property that every sliding window can be both-closed interval. Thus, if there exists n events matching l during $[29, 30]$, there may be no such events during $[29, 30)$, but n such events in $[30, 30]$ which belongs to the interval $[30, 37)$ as well.
Altogether, the derived cardinality constraint is $i : \exists_w h \leftarrow$ where $w := \left(\geq \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n, \leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n\right)$.

ii. $c = (\leq n)$:
Because of $c = (\leq n)$, there can be no event matching l during i. Thus, 0 is the lower bound of the derived cardinality specification. Its upper bound is $\left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$, since, e.g., there can be n events matching l in each of the time intervals $[29, 30]$ (in particular $[30, 30]$), $[31, 32)$, $[34, 35)$ and $[36, 37)$ as argued above. Altogether, the derived cardinality constraint is $i : \exists_w h \leftarrow$ where $w := \left(\leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n\right)$.

iii. $c = (\geq n)$:

Because of $c = (\geq n)$, the derived cardinality specification is not bounded above. Its lower bound is $\left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$, since, e.g., there may be (at least) n events matching l during $[16, 23)$ (compare Figure 6.4), i.e. $n = \left(\lfloor \frac{7}{3} \rfloor - 1\right) \cdot n = \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$. Hence, the derived cardinality constraint is $i : \exists_w h \leftarrow$ where $w := \left(\geq \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n\right)$.

Case 3. i is sliding window and j is tumbling window :

Since i is a sliding window, it can be moved (slid) with particular granularity such that it acts like a tumbling window. Therefore, the sliding window i can be mapped into correspondent tumbling window. Hence, in this case, the cardinality constraint is propagated analogously to Cases 1.(a)i. until 1.(b)iii. The results are summarized in Table 6.1 until Table 6.5.

Case 4. i and j are sliding windows :

Since i and j are sliding windows, they can be moved (slid) respectively with particular granularity (duration) such that they act like tumbling windows. In other words, for granularity conciding with the duration (window size), each of sliding windows i and j can be mapped into correspondent tumbling window. Therefore, this case can be led back to one of the previous Cases 1-3, for instance, i and j are tumbling windows, i is tumbling window and j is sliding window or vice versa. Thus, the cardinality constraint is derived analogously to those cases.

The next cases concern the propagation of cardinality constraints where the query evaluation time i or the constraint validity time j is a time interval. This time interval can be a now window, a user-defined window, an event, or an application state (consider [5] for the definitions of these notions). In addition, this time interval can be repeating or non-repeating. If it is repeating, it is neither tumbling nor sliding window, because these kinds of windows are explicitly treated. The bounds of this time interval must be known at compile time, since otherwise no constraint propagation is possible at complie time, instead of that the default cardinality specification (≥ 0) will be derived. Furthermore, the bounds of an event or an application state are often (but not always) known at compile time.

In next two cases 5 and 6, we assume that the constraint validity time j is either a tumbling window or a sliding window, while the query evaluation time i is only a time interval. Since j is a window and therefore a repeating time interval, there are always intervals j which overlap and even include i. So the cardinality constraint propagation in such cases can be led back to the previous cases in which i is considered as a tumbling window.

Case 5. i is a time interval and j is tumbling window :

According to the above arguments, this case can be led back to case 1 in which both i and j are tumbling windows. Hence, it has the same results as in Case 1.

Case 6. i is a time interval and j is sliding window :

Similar to Case 5, this case can be led back to Case 2 in which i is a tumbling window and j is sliding window. Hence, it has the same results as in Case 2. They are summarized in Tables 6.1-6.5.

If the constraint validity time j is a time interval and the query evaluation time i is a tumbling window, a sliding window, or a time interval, it may happen that there are some intervals i which overlap no interval j. As a consequence, no propagation of the constraints with the validity time j with respect to the queries with evaluation time i is possible. In this case, the default cardinality specification ≥ 0 is derived.

Seldom may the tumbling window, sliding window, or time interval i contain the time interval j or be included in j. In this case, we reduce the propagation of cardinality constraints with validity time j with respect to the queries with the evaluation time i to the cases in which i and j are tumbling or sliding windows.

The results of all cases described above are summarized in Tables 6.1-6.5.

Example 13.
Let $i : h \leftarrow l$ be a StreamLog query where $i := (\mathit{Range}\ 2h, \mathit{Slide}\ 15min)$ is the query evaluation time which is a sliding window, h is the query head, and l is the query body.

Let $j : \exists_{\leq 8}\, l \leftarrow$ be an ESCL cardinality constraint where $j := (\mathit{Range}\ 1h, \mathit{Slide}\ 5min)$ is the constraint validity time which is a sliding window, l is the atom of the constraint, and $c := (\leq 8)$ is the cardinality specification.

Since $i > j$ and i, j are sliding windows, the cardinality propagation function $\mathcal{C}$ with base case in Table 6.4 is applied to c. It yields $\mathcal{C}(c) = \mathcal{C}(\leq 8) = (\leq \left(\lceil \frac{2h}{1h} \rceil + 1\right) \cdot 8) = (\leq 24)$. Hence, the derived cardinality constraint is $(\mathit{Range}\ 2h, \mathit{Slide}\ 15min) : \exists_{\leq 24}\, h \leftarrow$.

6.1.2 Complex Cardinality Specifications

If the cardinality specification c is complex, then the transformation is recursively applied to each conjunct or disjunct of c as the algorithm in Listing 6.1 specifies.

j / i	tumbling window	sliding window	time interval
tumbling window, sliding window or time interval	$\leq 2n$	$\leq n$	≥ 0

Table 6.1: $i < j$ and $c = = n$ (or $c = \leq n$)

j / i	tumbling window, sliding window or time interval
tumbling window, sliding window or time interval	≥ 0

Table 6.2: $i < j$ and $c = \geq n$

This algorithm is based on Tables 6.1-6.5 the access to which is afforded by the function $lookTable(c, i, j)$. Which table is accessed depends on the time intervals i and j. c is a simple cardinality specification which is transformed from the time interval j into the time interval i.

```
transformCardSpec(c,i,j) {
  if c is a conjunction then
     subCardSpecs ← getConjuncts(c);
     conjunctLeft ← subCardSpecs[0];
     conjunctRight ← subCardSpecs[1];
     transformedConjunctLeft ← transformCardSpec(conjunctLeft,i,j);
     transformedConjunctRight ← transformCardSpec(conjunctRight,i,j);
     transformedCardSpec ← buildConjunction(transformedConjunctLeft,
                                            transformedConjunctRight);
  else if c is a disjunction then
     subCardSpecs ← getDisjuncts(c);
     disjunctLeft ← subCardSpecs[0];
     disjunctRight ← subCardSpecs[1];
     transformedDisjunctLeft ← transformCardSpec(disjunctLeft,i,j);
     transformedDisjunctRight ←transformCardSpec(disjunctRight,i,j);
     transformedCardSpec ← buildDisjunction(transformedDisjunctLeft,
                                            transformedDisjunctRight);
  else /* c is simple */
     transformedCardSpec ← lookTable(c,i,j);
  end end
  result ← simplify(transformedCardSpec);
  return result;
```

j i	tumbling window or sliding window	time interval
tumbling window, sliding window or time interval	$\geq \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n \wedge \leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$	≥ 0

Table 6.3: $i \geq j$ and $c == n$

j i	tumbling window or sliding window	time interval
tumbling window, sliding window or time interval	$\leq \left(\lceil \frac{i}{j} \rceil + 1\right) \cdot n$	≥ 0

Table 6.4: $i \geq j$ and $c = \leq n$

```
}
```

Listing 6.1: The function $transformCardSpec$ transforms the input cardinality specification c from the time interval i to the time interval j.

Remember that the function *getConjuncts* accepts a conjunction of cardinality specifications as parameter and returns an array of two elements which are the conjuncts. And the function *getDisjuncts* processes analogously for a disjunction of cardinality specifications.

The function *buildConjunction* or *buildDisjunction* take two cardinality specifications and connect them to one cardinality specification using the logical conjunction or disjunction respectively. For example, let α, β be two cardinality specifications, so $buildConjunction(\alpha, \beta)$ yields the conjunction $\alpha \wedge \beta$ and $buildDisjunction(\alpha, \beta)$ returns the disjunction $\alpha \vee \beta$.

The function $simplify$ takes a cardinality specification c as parameter and makes it shorter. It is described in detail in Chapter 7.

Theorem 6.1.1.
Let c be the cardinality specification of the constraint with the validity time j. It holds that the number of simple cardinality specifications in the result returned by $transformCardSpec(c, i, j)$ is bounded as follows

$$1 \leq |Simple(transformCardSpec(c, i, j))| \leq 2 \cdot |Simple(c)|$$

Proof. Let c be a cardinality specification with the validity time j. The function

i \ j	tumbling window or sliding window	time interval
tumbling window, sliding window or time interval	$\geq \left(\lfloor \frac{i}{j} \rfloor - 1\right) \cdot n$	≥ 0

Table 6.5: $i \geq j$ and $c = \geq n$

$transformCardSpec(c, i, j)$ applies the auxiliary function *lookTable* to each $c_k \in Simple(c)$ where $1 \leq k \leq n$ and $n := |Simple(c)|$. As a result, the function $lookTable(c_k, i, j)$ yields the transformed cardinality specification d_k with the validity time i and $1 \leq |Simple(d_k)| \leq 2$, for $1 \leq k \leq n$, according to Tables 6.1-6.5. Define $\mathcal{D} := \{d_1, \ldots, d_n\}$ as the set of transformed cardinality specifications d_k. Note that $|\mathcal{D}| = |Simple(c)|$ holds, since the function *lookTable* is performed for each $c_k \in Simple(c)$. In each recursion, the transformed cardinality specifications d_k are combined to conjunctions or disjunctions which are simplified then. In worst case, they cannot be simplified at all. So we obtain $|Simple(transformCardSpec(c, i, j))| \leq \sum_{k=1}^{n} |Simple(d_k)| = \sum_{k=1}^{n} 2 = 2n = 2 \cdot |Simple(c)|$. In the best case, all d_k can be simplified to a single simple cardinality specification. Altogether it yields

$$1 \leq |Simple(transformCardSpec(c, i, j))| \leq 2 \cdot |Simple(c)|$$

□

Theorem 6.1.2 (Termination of transformCardSpec).
The algorithm in Listing 6.1 terminates.

Proof. According to the above descriptions, we assume that the functions *getConjuncts*, *getDisjuncts*, *buildConjunction* and *buildDisjunction* trivially terminate. Furthermore, this can be proven by giving the correspondent algorithm. The function *lookTable* presents a table look-up with known indices, thus it terminates as well. The function *simplify* terminates after finite steps. The proof of its termination is given in Chapter 7.

The function *transformCardSpec* takes the cardinality specification c, the time intervals i and j as input values. The cardinality specification c is finite, i.e., consists of a finite number of simple cardinality specifications. Therefore, the number of conjuncts (disjuncts) in c is finite as well. If c is a simple cardinality specification, the funciton *transformCardSpec* will be finished in one step. Otherwise, it will be called recursively for each nested conjunct or disjunct in c. Since c is finite, there is a finite number of recursive calls of the function *transformCardSpec*. During each recursion, the number of simple cardinality specifications of the cardinality specification assigned to the variables

$conjunctLeft$, $conjunctRight$, $disjunctLeft$, $disjunctRight$ decreases steadily such that the break condition in Line 19 is reached after finite time. The result of each recursion is a simplified cardinality specification of the constraint with the validity time i. Afterwards, all these intermediate results are combined to the final result of the function $transformCardSpec(c, i, j)$. With the above assumption that other functions used in this algorithm terminate, we conclude that it terminates as well. $\square$

Theorem 6.1.3 (**Complexity of transformCardSpec**).
The complexity of the algorithm in Listing 6.1 is $O(n^2)$, where $n := |Simple(c)|$ for the cardinality specification c. In the best case, it reaches the complexity of $\Omega(n \cdot log_2(n))$.

Proof. First, we assume that the functions $getConjuncts$, $getDisjuncts$, $buildConjunction$ and $buildDisjunction$ have the complexity $\mathcal{O}(1)$. (This can be proven by giving the correspondent algorithms). The function $lookTable$ presents a table look-up with known indices, thus it has the complexity $\mathcal{O}(1)$ as well. The function $simplify$ has the complexity $\mathcal{O}(n)$. More details about it can be found in Chapter 7.

Let c be the a cardinality specification, and $n := |Simple(c)|$ be the number of all simple cardinality specifications in c. We define $\mathcal{T}(n)$ as the complexity of this algorithm depending on n. It holds $\mathcal{T}(c) \equiv \mathcal{T}(n)$ as well.

If c is complex, i.e., c is a nested conjunction or disjunction of cardinality specifications, then c is first divided in its two conjuncts or disjuncts. This corresponds to Lines $3 - 5$ and $11 - 13$ in Listing 6.1. Since this division phase is performed once in each function call, it requires constant time. Therefore it holds $\mathcal{D}(n) = \mathcal{O}(1)$, where $\mathcal{D}(n)$ denotes the complexity of the division phase depending on n.

As the result of the division phase, the initial problem is divided into two partial problems (divide and conquer principle) with the complexities $\mathcal{T}(n_1)$ and $\mathcal{T}(n_2)$ where $n_1 + n_2 = n$. Each of these partial problems is solved by the recursive calls of the algorithms (consider Lines $6 - 7$ and $14 - 15$ in Listing 6.1).

Afterwards, we combine the transformed conjuncts or disjuncts together to a cardinality specification - assigned to the variable $transformedCardSpec$. This combination phase corresponds to Lines 8 and 16 in the algorithm. We call $\mathcal{C}(n)$ its complexity. Since it is performed only once by the algorithm, it holds that $\mathcal{C}(n) = \mathcal{O}(1)$.

Finally, the transformed cardinality specification is simplified and returned as result. The complexity for the simplification is $\mathcal{S}(n) = \mathcal{O}(n)$ (see Theorem 7.0.8).

So we obtain the recursion equation

$$\begin{aligned}\mathcal{T}(n) &= \mathcal{D}(n) + \mathcal{T}(n_1) + \mathcal{T}(n_2) + \mathcal{C}(n) + \mathcal{S}(n)\\ &= \mathcal{O}(1) + \mathcal{T}(n_1) + \mathcal{T}(n_2) + \mathcal{O}(1) + \mathcal{O}(n)\\ &= \mathcal{O}(n) + \mathcal{T}(n_1) + \mathcal{T}(n_2)\end{aligned}$$

where $n_1 + n_2 = n$ for $n_1, n_2 \neq 0$ and c is a complex cardinality specification.

If c is simple, then $n = 1$. Hence $\mathcal{T}(c) \equiv \mathcal{T}(n) = \mathcal{T}(1) = \mathcal{O}(1)$, since only Line 19 is required. Furthermore, after c is transformed by calling *lookTable*, the length of the transformed cardinality specification is at most 2. So $\mathcal{S}(2) = \mathcal{O}(2) = \mathcal{O}(1)$ has no effect to the complexity of whole algorithm.

The complete recursion equation is

$$\mathcal{T}(n) = \begin{cases}\mathcal{O}(1) & \text{, if } c \text{ is simple}\\ \mathcal{O}(n) + \mathcal{T}(n_1) + \mathcal{T}(n_2) & \text{, otherwise, } n = n_1 + n_2\end{cases}$$

which can be formed to

$$\mathcal{T}(n) = \begin{cases}e & \text{, if } c \text{ is simple}\\ en + \mathcal{T}(n_1) + \mathcal{T}(n_2) & \text{, otherwise, } n = n_1 + n_2\end{cases}$$

where we can intuitively replace $\mathcal{O}(1)$ by a constant e and $\mathcal{O}(n)$ by en. (See [3]).

This recursion equation is illustrated by the following binary trees

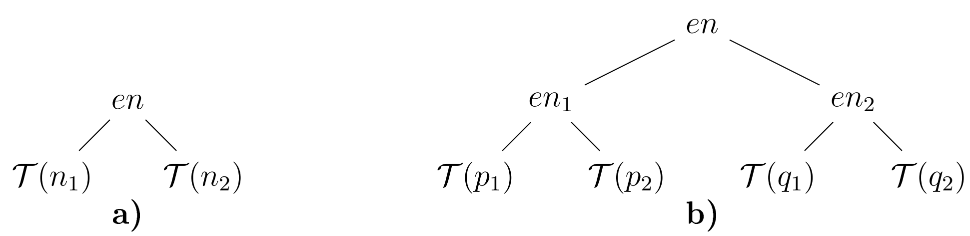


Figure 6.5: a) Binary tree for the recursion equation $\mathcal{T}(n)$, b) Expanded binary tree for $\mathcal{T}(n)$ with $p_1 + p_2 = n_1$ and $q_1 + q_2 = n_2$.

$\mathcal{T}(n)$ is determined by the number of nodes of the tree representing the structure of c. In the best case, the structure of c is the balanced binary tree all leaves of which are at the same level, consider Figure 6.6.

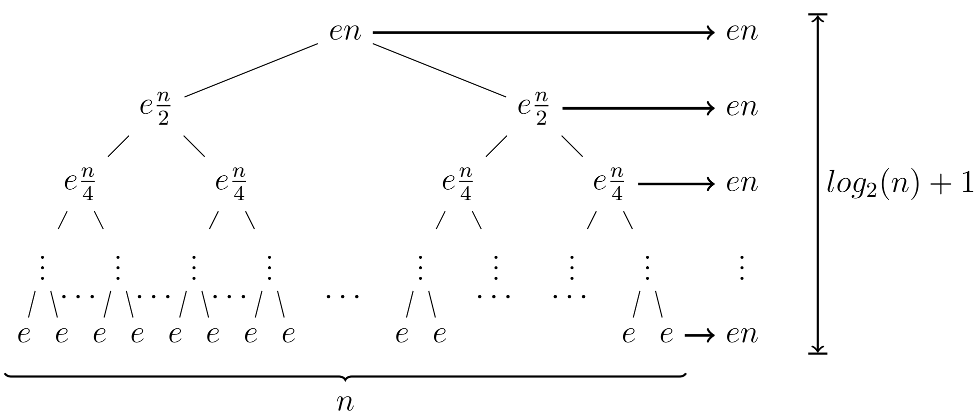


Figure 6.6: Recursion binary tree for the best case.

In this case, the recursion equation is as follows [6]:

$$\mathcal{T}(n) = \begin{cases} e & \text{, if } c \text{ is simple} \\ en + \mathcal{T}(\lfloor \frac{n}{2} \rfloor) + \mathcal{T}(\lceil \frac{n}{2} \rceil) & \text{, otherwise,} \end{cases}$$

Therefore, the complexity of the algorithm in the best case is

$$\mathcal{T}(n) = en + 2\mathcal{T}\left(\frac{n}{2}\right) = en \cdot (log_2(n) + 1) \in \Omega(n \cdot log_2(n))$$

since the height of the tree is $log_2(n)$ and there are at most n nodes at each level.

The worst case of the algorithm is reached if the structure of c is an unbalanced binary tree. An extreme case is shown in Figure 6.7.

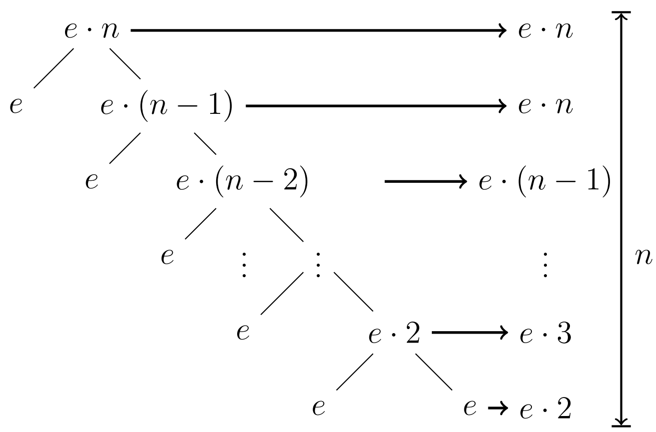


Figure 6.7: Binary recursion tree for the worst case.

The recursion equation for the worst case is

$$\mathcal{T}(n) = \begin{cases} e & \text{, if } c \text{ is simple} \\ en + \mathcal{T}(1) + \mathcal{T}(n-1) & \text{, otherwise} \end{cases} \tag{6.1}$$

Therefore, the complexity of the algorithm in the worst case is

$$\begin{aligned} \mathcal{T}(n) &= 2e + 3e + \cdots + (n-1)e + ne + ne = \left(\sum_{k=2}^{n} k\right) e + ne \\ &= \left(\left(\sum_{k=1}^{n} k\right) - 1\right) e + ne \\ &= \left(\tfrac{n(n+1)}{2} - 1 + n\right) e \\ &= (\tfrac{n^2+3n-2}{2})e \\ &\in \mathcal{O}(n^2) \text{ , because } e \text{ is a constant} \end{aligned}$$

since the height of the tree is at most n and there are at most n nodes at each level.

$\square$

Theorem 6.1.4 (Correctness of transformCardSpec).
Let $i : h \leftarrow l$ be the StreamLog query where i is the query evaluation time, h and l are atoms. Let $j : \exists_c l \leftarrow$ be the ESCL constraint where j is the constraint validity time, c is the cardinality specification, and l is an atom. For each interpretation $\mathcal{I}$ (as defined in [5]), it holds that

$$(\mathcal{I} \models (i : h \leftarrow l)) \wedge (\mathcal{I} \models (j : \exists_c l \leftarrow)) \wedge ((i : h \leftarrow l), (j : \exists_c l \leftarrow) \vdash (i : \exists_d l \leftarrow)) \Rightarrow (\mathcal{I} \models (i : \exists_d l \leftarrow))$$

where d is the result returned by the function $transformCardSpec(c, i, j)$, i.e. the transformed cardinality specification c from the time interval j to the time interval i.

Proof. (By induction over the number k of the simple cardinality speicfications in c)

- *Base Case*: $k = 1$

  i.e. c is a simple cardinality specification. Therefore, the algorithm in Listing 6.1 immediately calls the function *lookTable* in Line 19 to transform the cardinality specification c of the constraint with the validity time j into the cardinality specification d of the constraint with the validity time i, and the function $simplify$ in Line 21 to simplify d. The function $lookTable$ is correct, since it defines (the access to) the base cases (consider Tables 6.1-6.5). The function $simplify$ is correct according to Theorem 7.0.10. The query and the propagated constraint are also correct with respect to all interpretations $\mathcal{I}$ according to the precondition. Thus, the result of the algorithm is also correct, i.e. $I \models (i : \exists_d\, l \leftarrow)$.

- *Induction Assumption*: The theorem holds for all cardinality specifications c consisting of k simple cardinality specifications.

- *Induction Step*: $k \mapsto k+1$

  i.e. c consists of $k+1$ simple cardinality specifications. Without restriction of generality, we assume that $c = c_1 \circ c_2$ with $k+1 = k_1 + k_2$ where $k_1 := |Simple(c_1)|, k_2 := |Simple(c_2)|$ and $\circ \in \{\wedge, \vee\}$.

  Since $1 \leq k_1 \leq k$ and $1 \leq k_2 \leq k$, the theorem holds for both c_1 and c_2, according to the induction assumption. More exactly, c_1 is transformed into a correct cardinality specification d_1 and c_2 is transformed into a correct cardinality specification d_2. If $c = c_1 \wedge c_2$, the result of the algorithm is $d := d_1 \wedge d_2$. If $c = c_1 \vee c_2$, the result of the algorithm is $d := d_1 \vee d_2$. And d is correct, since its both adjuncts are correct. Furthermore, d is simplified by the function $simplify$ which is correct according to Theorem 7.0.10. Hence, it holds that $\mathcal{I} \models (i : \exists_d\, l \leftarrow)$.

$\square$

Theorem 6.1.5 (Completeness of transformCardSpec).
Let $i : h \leftarrow l$ be the StreamLog query and $j : \exists_c\, l \leftarrow$ and $i : \exists_d\, h \leftarrow$ be the ESCL cardinality constraints, where i and j are time intervals, c and d are cardinality specifications, and h and l are atoms. For each interpretation I (as defined in [5]), the following holds:

$$(I \models (i : h \leftarrow l)) \wedge (I \models (j : \exists_c\, l \leftarrow)) \wedge (I \models (i : \exists_d\, h \leftarrow)) \;\Rightarrow\; ((i : h \leftarrow l), (j : \exists_c\, l \leftarrow) \;\vdash\; (i : \exists_d\, h \leftarrow))$$

where d is the result of the transformation of the cardinality specification c from the time interval j to the time interval i returned by the algorithm $transformCardSpec(c, i, j)$ in Listing 6.1.

Proof. (By induction over the number k of the simple cardinality speicfications in c)

- *Base Case*: $k = 1$

  c is a simple cardinality specification. Therefore, the algorithm in Listing 6.1 calls the function $lookTable(c, i, j)$ in Line 19. The function defines (the access to) the base cases (consider Tables 6.1-6.5) of the transformation of the cardinality specification c from the time interval j to the time interval i. The result of this transformation is the cardinality specification d which is returned by the function, i.e. the function $lookTable$ is complete.

Afterwards, the function $simplify(d)$ is called in Line 21 to simplify d, i.e. to make it shorter. Therefore, the function $simplify$ is trivially complete.

Thus, the algorithm in Listing 6.1 is complete, i.e. $((i : h \leftarrow l), (j : \exists_c l \leftarrow) \vdash (i : \exists_d h \leftarrow))$.

- *Induction Assumption*: The theorem holds for all cardinality specifications c consisting of not more than k simple cardinality specifications.

- *Induction Step*: $k \mapsto k+1$

  Without loss of generality, we assume that $c = c_1 \circ c_2$, where $\circ \in \{\wedge, \vee\}$, c_1 consists of k_1 simple cardinality specifications, c_2 consists of k_2 simple cardinality specifications, such that $k_1 + k_2 = k + 1$.

  Remember that the function $transformCardSpec(c, i, j)$ is recursively applied to each (nested) adjunct of c (compare Lines $6-7$ and $14-15$ in Listing 6.1) and the result of each application is then appended by $\wedge$ or $\vee$ to the final result (compare Lines $8-9$ and $16-17$). According to the induction assumption, the theorem holds for both c_1 and c_2, since $k_1 \leq k$ and $k_2 \leq k$. The cardinality specifications d_1 and d_2 are returned by the function calls $transformedCardSpec(c_1, i, j)$ and $transformedCardSpec(c_2, i, j)$ respectively.

  Afterwards, d_1 and d_2 are combined to $d = d_1 \wedge d_2$ by the function $buildConjunction$ in Lines $8-9$ if $c = c_1 \wedge c_2$ (or to $d = d_1 \vee d_2$ by the function $buildDisjunction$ in Lines $16-17$ if $c = c_1 \vee c_2$). The functions $buildConjunction$ and $buildDisjunction$ are trivially complete.

  Then, d is simplified (made shorter) by the function $simplify(d)$ in Line 21. The function is trivially complete. The simplified d is returned by the algorithm in Listing 6.1. Therefore, the algorithm is complete, i.e. $((i : h \leftarrow l), (j : \exists_c l \leftarrow) \vdash (i : \exists_d h \leftarrow))$.

$\square$

6.2 Arbitrary Number of Atoms in a Query Body

This section is devoted to the case in which cardinality constraints are propagated with respect to a query the body of which is an arbitrary nested conjunction or disjunction of atoms. Consider the function *propagateCardSpec* in Listing 6.2 which is the extention of the algorithm in Listing 5.1 to deal with the case in which the query evaluation time does not coincide with the validity time intervals of the propagated constraints.

```
propagateCardSpecs(query_body) {
  if query_body is a conjunction then
     subqueries ← getConjuncts(query_body);
     cardSpecLeft ← propagateCardSpecs(subqueries[0]);
     cardSpecRight ← propagateCardSpecs(subCardSpecs[1]);
     cardSpec ← 𝒞∧(cardSpecLeft,cardSpecRight);
  else if query_body is a disjunction then
     subqueries ← getDisjuncts(query_body);
     cardSpecLeft ← propagateCardSpecs(subqueries[0]);
     cardSpecRight ← propagateCardSpecs(subCardSpecs[1]);
     cardSpec ← 𝒞∨(cardSpecLeft,cardSpecRight);
  else /* query_body is an atom*/
     i ← getQueryEvaluationTime(query_body);
     j ← getConstraintValidityTime(query_body);
     c ← getCardSpec(query_body);
     if i = j then
        cardSpec ← c;
     else
        cardSpec ← transformCardSpec(c,i,j);
     end end end
  return cardSpec;
}
```

Listing 6.2: The function *propagateCardSpecs* propagates cardinality constraints with respect to the query the body of which is an arbitrary nested conjunction or disjunction of atoms.

Remember that the function *getConjuncts* accepts a conjunction of cardinality specifications as input and yields an array of two elements which are the conjuncts. The function *getDisjunts* processes analogously for a disjunction of cardinality specifications.

The function *getQueryEvaluationTime* takes an atom as input and returns its evaluation time interval. The function *getConstraintValidityTime* also takes an atom as input and yields the validity time of the constraint for this atom. Similarly, the function *getCardSpec* takes an atom as input and returns the cardinality specification of the constraint for this atom. The function *transformCardSpec* is described in detail in Section 6.1.

To understand how the above algorithm works consider Example 14.

Example 14.
Let q of the form

$$Range\ 1h : h \leftarrow l_1 \vee (l_2 \wedge l_3)$$

be the StreamLog query. The evaluation time of q is the tumbling window i.

Let

$$\begin{array}{lll} Range\ 2h & : \exists_{(\geq 5 \wedge \leq 7) \vee (=2)} & l_1 \leftarrow \\ Range\ 3h,\ Slide\ 15min & : \exists_{\leq 2} & l_2 \leftarrow \\ Range\ 30min,\ Slide\ 10min & : \exists_{=4} & l_3 \leftarrow \end{array}$$

be the ESCL cardinality constraints. The validity time of the first constraint is the tumbling window. The validity time intervals of the second and the third constraints are sliding windows. The cardinality specification of the first constraint is complex. The cardinality specifications of the second and the third constraints are simple.

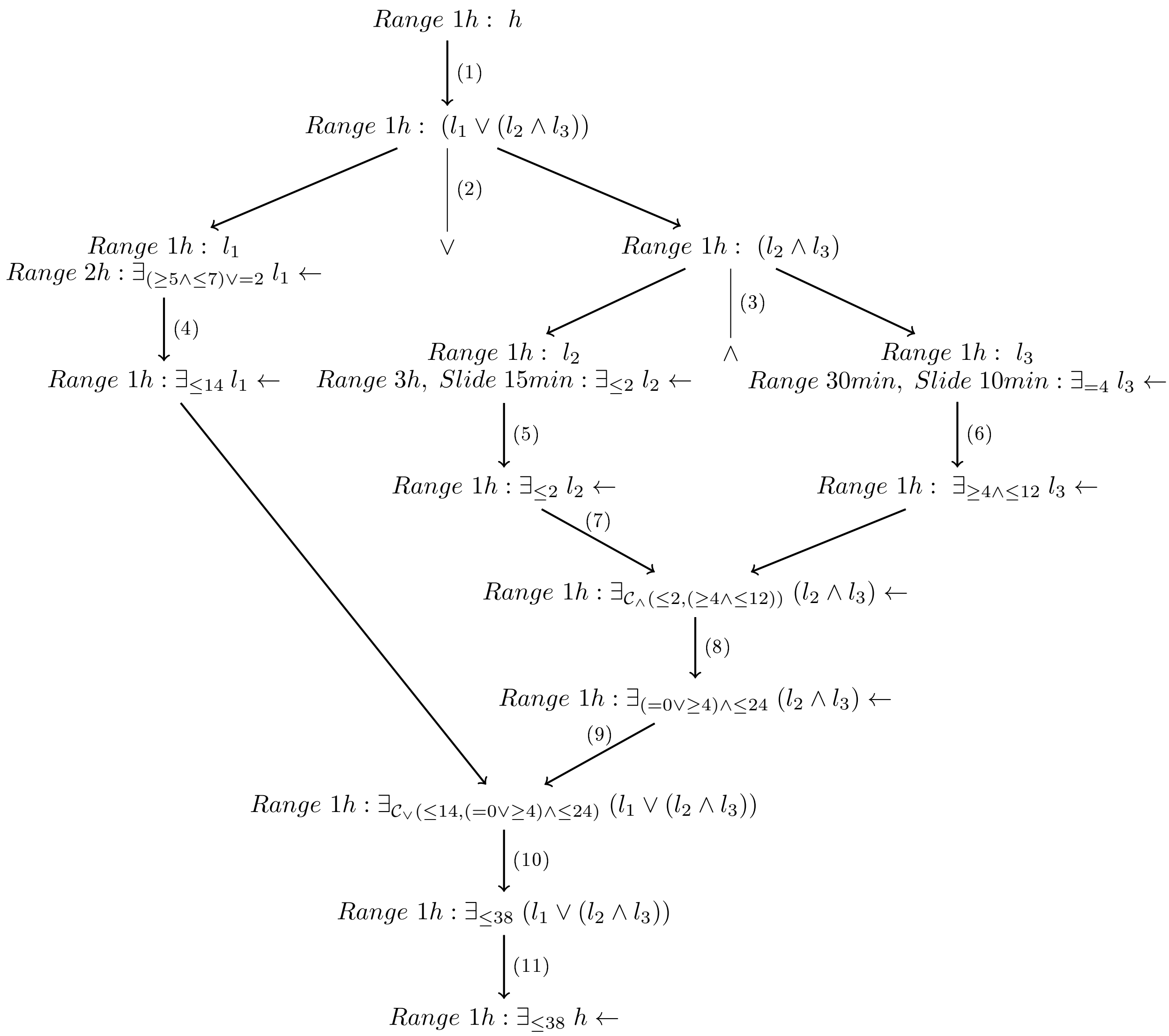


Figure 6.8: Illustration of Example 14

In order to propagate these constraints with respect to the query q, i.e. to derive the cardinality of h within the tumbling window i, the following steps are made (compare Figure 6.8):

1. $Range\ 1h:\ h$ is led back to $Range\ 1h:\ (l_1 \vee (l_2 \wedge l_3))$ because of the query q.

2. $Range\ 1h:\ (l_1 \vee (l_2 \wedge l_3))$ is split into disjuncts $Range\ 1h:\ l_1$ and $Range\ 1h:\ (l_2 \wedge l_3)$.

3. $Range\ 1h:\ (l_2 \wedge l_3)$ is split into conjuncts $Range\ 1h:\ l_2$ and $Range\ 1h:\ l_3$.

4. The cardinality constraint $Range\ 2h: \exists_{(\geq 5 \wedge \leq 7) \vee = 2}\ l_1 \leftarrow$ is transformed into the tumbling window i according to Tables 6.1 and 6.2. The result is $Range\ 1h: \exists_{\leq 14} l_1 \leftarrow$. More exactly (compare Listing 6.1): $transformCardSpec((\geq 5 \wedge \leq 7) \vee = 2,\ Range\ 1h,\ Range\ 2h) = (\geq 0 \wedge \leq 14) \vee \leq 14 = \leq 14$.

5. The cardinality constraint $Range\ 3h,\ Slide\ 15min : \exists_{\leq 2}\ l_2 \leftarrow$ is transformed into the tumbling window i according to Table 6.1. The result is $Range\ 1h : \exists_{\leq 2}\ l_2 \leftarrow$.

6. The cardinality constraint $Range\ 30min,\ Slide\ 10min : \exists_{=4}\ l_3 \leftarrow$ is transformed into the tumbling window i according to Table 6.3. The result is $Range\ 1h : \exists_{\geq 4 \wedge \leq 12}\ l_3 \leftarrow$.

7. Since l_2 and l_3 are conjuncts (compare Step 3), $\mathcal{C}_\wedge$ is applied to the transformed cardinality constraints for l_2 and l_3 obtained in Steps 5 and 6 respectively. The result is $Range\ 1h : \exists_{\mathcal{C}_\wedge(\leq 2,(\geq 4 \wedge \leq 12))}\ (l_2 \wedge l_3) \leftarrow$.

8. According to Table 5.3, the result of Step 7 is simplified to $Range\ 1h : \exists_{(=0 \vee \geq 4) \wedge \leq 24}\ (l_2 \wedge l_3) \leftarrow$.

   More exactly: $\mathcal{C}_\wedge(\leq 2, (\geq 4 \wedge \leq 12)) = \mathcal{C}_\wedge(\leq 2, \geq 4) \wedge \mathcal{C}_\wedge(\leq 2, \leq 12) = (= 0 \vee \geq 4) \wedge \leq 24$.

9. Since l_1 and $(l_2 \wedge l_3)$ are disjuncts (compare Step 2), $\mathcal{C}_\vee$ is applied to the transformed cardinality constraints for l_1 and $(l_2 \wedge l_3)$ obtained in Steps 4 and 8 respectively. The result is $Range\ 1h : \exists_{\mathcal{C}_\vee(\leq 14,(=0 \vee \geq 4) \wedge \leq 24)}\ (l_1 \vee (l_2 \wedge l_3)) \leftarrow$.

10. According to Table 5.4, the result of Step 9 is simplified to $Range\ 1h : \exists_{\leq 38}\ (l_1 \vee (l_2 \wedge l_3)) \leftarrow$.

    More exactly: $\mathcal{C}_\vee(\leq 14, (= 0 \vee \geq 4) \wedge \leq 24) = (\mathcal{C}_\vee(\leq 14, = 0) \vee \mathcal{C}_\vee(\leq 14, \geq 4)) \wedge \mathcal{C}_\vee(\leq 14, \leq 24) = (\leq 14 \vee \geq 4) \wedge \leq 38 = \leq 38$.

11. $Range\ 1h : \exists_{\leq 38}\ h \leftarrow$ is the derived cardinality constraint.

Theorem 6.2.1 (Termination of propagateCardSpecs).
The algorithm in Listing 6.2 terminates.

Proof. As mentioned above, the auxiliary functions $getConjuncts$ and $getDisjuncts$ terminate. Moreover, we assume that the functions $getQueryEvaluationTime$, $getConstraintValidityTime$ and $getCardSpec$ also terminate. The function $transformCardSpec$ terminates, according to Theorem 6.1.2.

The function *propagateCardSpecs* takes a query body as input. Since the number of atoms in it is finite, the number of propagated cardinality constraints is also finite. Thus the function $propagateCardSpec$ is recursively called finitely often. The cardinality specifications of the propagated constraints are finite, i.e. they consist of finitely many of simple cardinality specifications. Therefore, each recursive call of the function $propagateCardSpec$ terminates. Because of these reasons the algorithm in Listing 6.2 terminates. $\square$

Remark 6.2.1.
For the sake of convenience, in the following, we will write $|c|$ as the abbreviation form of $|Simple(c)|$, i.e. $|c| := |Simple(c)|$.

Let q be the StreamLog query of the form $i : h \leftarrow b$ where i is the query evaluation time, h is the query head, and b is the query body. In the course of this section, we call m the number of all atoms in b, $m \geq 1$. Let $A_1, .., A_m$ be the atoms in b. Let $j_{A_1}, .., j_{A_m}$ be the validity time intervals and $c_{A_1}, .., c_{A_m}$ be the cardinality specifications of the constraints for $A_1, .., A_m$ respectively. We define $n := \underset{1 \leq k \leq m}{max}\{|c_{A_k}|\}$, where $|c_{A_k}| = |Simple(c_{A_k})|$ (consider Remark 6.2.1). The following theorem will help to determine the complexity of the algorithm in Listing 6.2.

Theorem 6.2.2.
Let b be the query body as described above. Let c be the cardinality specification and final result returned by the function *propagateCardSpecs*(b) (i.e. $c :=$ *propagateCardSpecs*(b)). It holds that

$$|c| \leq 2^{2m-1} \cdot n^m$$

where m is the number of atoms in b and $n := \underset{1 \leq k \leq m}{max}\{|c_{A_k}|\}$ as stated above.

Proof. (By induction over the number m of atoms in b)

- *Base Case*: $m = 1$

  Since b consists of one atom A, Lines $13 - 15$ of the algorithm in Listing 6.2 are called. Let j_A be the validity time and c_A be the cardinality specification of the constraint for A. Furthermore, we define $n := |c_A|$. Let c be the final result of the function *propagateCardSpec*(b). If the query evaluation time i coincides with the constraint validity time j_A, Line 17 in Listing 6.2 is called and yields the cardinality specification $c = c_A$. Otherwise, Line 19 in Listing 6.2 is called and the function $transformCardSpec(c_A, i, j_A)$ yields the cardinality specification c where $|c| \leq 2n$, according to Theorem 6.1.1. Hence, it holds that $|c| \leq 2n = 2^{2m-1} \cdot n^m$.

- *Induction Assumption*: The theorem holds for all query bodies consisting of m atoms, $m \in \mathbb{N}$.

- *Induction Step*: $m \mapsto m + 1$

  b consists of $m + 1$ atoms. Without restriction of generality, we assume that $b = b_1 \circ b_2$ where $\circ \in \{\wedge, \vee\}$, b_1 consists of l atoms denoted by $A_1, .., A_l$ (for $l \geq 1$), b_2 consists of $(m + 1 - l)$ atoms denoted by $A_{l+1}, .., A_{m+1}$. Furthermore, we define $c_{A_1}, .., c_{A_l}, c_{A_{l+1}}, .., c_{A_{m+1}}$ as the cardinality specifications of the constraints for $A_1, .., A_l, A_{l+1}, .., A_{m+1}$ respectively, and $n := \max_{1 \leq k \leq m+1} \{|c_{A_k}|\}$.

Since b is not a single atom, Lines $2-6$ (or $7-10$) of the algorithm in Listing 6.2 are called. In Line 4 (or 9), the function *propagateCardSpecs* is applied to the (sub)query b_1. The result is the cardinality specification assigned to the variable *cardSpecLeft*. Since $1 \leq l \leq m$, the theorem holds for this (sub)query b_1. Hence, it holds that

$$|cardSpecLeft| \leq 2^{2l-1} \cdot n^l$$

In Line 5 (or 10), the function *propagateCardSpecs* is applied to the (sub)query b_2. The result is the cardinality specification assigned to the variable *cardSpecRight*. Since $1 \leq (m+1-l) \leq m$, the theorem holds for the (sub)query b_2 as well. Thus, it holds that

$$|cardSpecRight| \leq 2^{2\cdot(m-l)+1} \cdot n^{m+1-l}$$

In Line 6 (or 11), the function $\mathcal{C}_\wedge$ (or $\mathcal{C}_\vee$) is applied to *cardSpecLeft* and *cardSpecRight*. It yields the cardinality specification c. According to Theorem 5.1.4, it holds that

$$|c| \leq 2 \cdot \underbrace{|cardSpecLeft|}_{\leq 2^{2l-1} \cdot n^l} \cdot \underbrace{|cardSpecRight|}_{\leq 2^{2\cdot(m-l)+1} \cdot n^{m+1-l}} \leq 2^{2(m+1)-1} \cdot n^{m+1}$$

□

Theorem 6.2.3 (Complexity of propagateCardSpecs).
Let q be the StreamLog query as described above. The complexity for the algorithm in Listing 6.2 is

$$\begin{cases} \mathcal{O}(n^2) & \text{, if } m = 1 \\ \mathcal{O}(2^{4m-6} \cdot n^{2m-2}) & \text{, if } m \geq 2 \end{cases}$$

where m is the number of all atoms in the query body, and $n := \max_{1 \leq k \leq m} \{|c_{A_k}|\}$.

Proof. (By induction over the number m of atoms in the query body)

First, we assume that the functions *getConjuncts*, *getDisjuncts*, *getQueryEvaluationTime*, *getConstraintValidityTime* and *getCardSpec* have the complexity $\mathcal{O}(1)$. This can be proven by giving the correspondent algorithms for these functions.

Furthermore, we define $\mathcal{T}(m, n)$ as the complexity of the algorithm in Listing 6.2. If the body of q is an atom A (i.e. $m = 1$) with the cardinality specification c_A and the constraint validity time j_A, then the instructions in Lines $13 - 15$ are performed. The cost for them is clearly $\mathcal{O}(1)$. If $i = j_A$, the instruction in Line 17 is executed which has the cost of $\mathcal{O}(1)$ as well. Otherwise, the function $transformCardSpec$ is applied to c_A, i and j_A. According to Theorem 6.1.3, the cost for this function call is $\mathcal{O}(n^2)$ where $n := |c_A|$. Altogether, the complexity for all instructions in Lines $12 - 19$ is $\mathcal{O}(n^2)$, i.e. $\mathcal{T}(1, n) = \mathcal{O}(n^2)$.

If the query body is either a conjunction or a disjunction of numerous atoms (i.e. $m \geq 2$), it is divided in two conjuncts or two disjuncts. This correponds to Line 3 (or 8) in Listing 6.2. This division phase has the complexity of $\mathcal{D}(n, m) := \mathcal{O}(1)$.

In the next Lines $4 - 5$ (or $6 - 10$), two recursion calls of the function *propagateCardSpecs* are performed. Each of them has the complexity of $\mathcal{T}(m_1, n)$ and $\mathcal{T}(m_2, n)$ where $m = m_1 + m_2$, and $m_1, m_2 \in \mathbb{N}$.

Next, we define $c_l(m_1)$, $c_r(m_2)$ as the cardinality specifications stored in the variables $cardSpecLeft$, $cardSpecRight$ respectively. For the sake of simplicity, we will write c_l, c_r instead of $c_l(m_1)$, $c_r(m_2)$, and keep in memory that the number of simple cardinality specifications in c_l, c_r depends only on m_1, m_2 respectively.

Afterwards, the cardinality propagation function $\mathcal{C}_\wedge$ (or $\mathcal{C}_\vee$) is applied to c_l and c_r and returns a cardinality specification as result. This propagation phase corresponds to Line 6 (or 11) in Listing 6.2. According to Theorem 5.1.3, its complexity is $\mathcal{P}(|c_l|, |c_r|) := \mathcal{O}(max\{|c_l|, |c_r|\}^2)$, where $|c_l|$, $|c_r|$ denote the number of simple cardinality specifications in c_l, c_r (consider Remark 6.2.1). The recursion equation of $\mathcal{T}(m, n)$ is as follows

$$\mathcal{T}(m,n) = \begin{cases} \mathcal{O}(n^2) & \text{, if } m = 1 \\ \overbrace{\mathcal{D}(n,m)}^{=\mathcal{O}(1)} + \mathcal{T}(m_1,n) + \mathcal{T}(m_2,n) + \mathcal{P}(|c_l|, |c_r|) & \text{, otherwise \& } m_1 + m_2 = m \end{cases}$$

which can be simplified to

$$\mathcal{T}(m,n) = \begin{cases} \mathcal{O}(n^2) & \text{, if } m = 1 \\ \mathcal{T}(m_1,n) + \mathcal{T}(m_2,n) + \mathcal{P}(|c_l|, |c_r|) & \text{, otherwise \& } m_1 + m_2 = m \end{cases}$$

Now, we show $\mathcal{T}(m, n) = \mathcal{O}(2^{4m-6} \cdot n^{2m-2})$ by induction over $m \geq 2$.

- *Base Case*: $m = 2$

  Since the query body consists of two atoms which we denote by A_1 and A_2, the function *propagateCardSpecs* is recursively applied to each of them. This corresponds to Lines $4-6$ (or $9-10$) in Listing 6.2. The results are cardinality specifications stored in the variables c_l and c_r, i.e. $c_l := propagateCardSpecs(A_1)$ and $c_r := propagateCardSpecs(A_2)$. The complexity for each of the function calls *propagateCardSpecs* is $\mathcal{T}(1,n) = \mathcal{O}(n^2)$, according to Theorem 6.1.3, where $n := max\{|c_{A_1}|, |c_{A_2}|\}$ and c_{A_1}, c_{A_2} denote the cardinality specifications of the cardinality constraints for A_1, A_2.

According to Theorem 6.2.2, it holds that $|c_l| \leq 2n$ and $|c_r| \leq 2n$. Next, in Line 6 (or 11), the function $\mathcal{C}_\wedge$ (or $\mathcal{C}_\vee$) is applied to c_l and c_r. The complexity for this propation phase is also $\mathcal{P}(|c_l|, |c_r|) = \mathcal{O}((2n)^2)$, according to Theorem 5.1.3. We obtain the following recursion equation :

$$\begin{aligned}
\mathcal{T}(2,n) &= 2 \cdot \underbrace{\mathcal{T}(1,n)}_{=\mathcal{O}(n^2)} + \mathcal{P}(\underbrace{|c_l|}_{\leq 2n}, \underbrace{|c_r|}_{\leq 2n}) \\
&= \mathcal{O}(n^2) + \mathcal{O}((2n)^2) \\
&= \mathcal{O}(n^2) + \mathcal{O}(4n^2) \\
&= \mathcal{O}(2^2 \cdot n^2) \\
&= \mathcal{O}(2^{4\cdot 2-6} \cdot n^{2\cdot 2-2})
\end{aligned}$$

- *Induction Assumption*: It holds that $\mathcal{T}(m,n) = \mathcal{O}(2^{4m-6} \cdot n^{2m-2})$ for all $m \in \mathbb{N}$, $m \geq 2$.

- *Inductive Step:* $m \mapsto m+1$

  Let b denote the body of the query q. Similar to the proof of Theorem 6.1.3, the worst case of the complexity is reached if the structure of b is represented by an unbalanced binary tree depicted in Figure 6.7. Without restriction of generality, we assume that $b = b' \circ A_{m+1}$ where A_{m+1} is an atom, $\circ \in \{\wedge, \vee\}$, b' consists of m atoms $A_1, .., A_m$, and $b' = ((\ldots((A_1 \circ A_2) \circ A_3) \circ \ldots) \circ A_{m-1}) \circ A_m$. Let $c_{A_1}, .., c_{A_m}, c_{A_{m+1}}$ be the cardinality specifications of the constraints for $A_1, .., A_m, A_{m+1}$ respectively. Furthermore, we define $n := \max\limits_{1\leq k\leq m+1}\{|c_{A_k}|\}$.

  Since b is not a simple atom, Lines $2-6$ (or $7-10$) of the algorithm in Listing 6.2 are called. In Line 4 (or 9), the function *propagateCardSpecs* is applied to the (sub)query b'. The result is a cardinality specification stored in c_l. Note that $c_l = cardSpecLeft$. It holds that

  $$|c_l| \leq 2^{2m-1} \cdot n^m$$

  according to Theorem 6.2.2.

In Line 5 (or 10), the function *propagateCardSpecs* is applied to the (sub)query A_{m+1}. The result is a cardinality specification stored in c_r. Note that $c_r = cardSpecRight$. It holds that

$$|c_r| \leq 2n$$

according to Theorem 6.2.2.

The complexity for the propagation phase is

$$\mathcal{P}(|c_l|, |c_r|) = \mathcal{O}(max\{|c_l|, |c_r|\}^2) = \mathcal{O}((2^{2m-1} \cdot n^m)^2)$$

Therefore, we obtain the following recursion equation:

$$\begin{aligned}
\mathcal{T}(m+1, n) &= \mathcal{T}(m, n) + \underbrace{\mathcal{T}(1, n)}_{\mathcal{O}(n^2)} + \mathcal{P}(|c_l|, |c_r|) \\
&= \mathcal{T}(m, n) + \mathcal{O}(n^2) + \mathcal{O}((2^{2m-1} \cdot n^m)^2)
\end{aligned}$$

According to induction assumption, it holds that $\mathcal{T}(m, n) = \mathcal{O}(2^{4m-6} \cdot n^{2m-2})$. Thus

$$\begin{aligned}
\mathcal{T}(m+1, n) &= \mathcal{O}(2^{4m-6} \cdot n^{2m-2}) + \mathcal{O}(n^2) + \mathcal{O}((2^{2m-1} \cdot n^m)^2) \\
&= \mathcal{O}((2^{2m-1} \cdot n^m)^2) \\
&= \mathcal{O}(2^{4(m+1)-6} \cdot n^{2(m+1)-2})
\end{aligned}$$

□

Theorem 6.2.4 (Correctness of propagateCardSpecs).
Let q be a StreamLog query of the form $i : h \leftarrow b$ where i is the query evaluation time, h is the query head, and b is the query body. Let $l_1, .., l_m$ be the atoms in b such that $\mathcal{C} := \{(j_1 : \exists_{c_1} l_1 \leftarrow), .., (j_m : \exists_{c_m} l_m \leftarrow)\}$ is the set of ESCL constraints describing their cardinality, where $j_1, .., j_m$ are the constraint validity times and $c_1, .., c_m$ are the cardinality specifications.

The algorithm in Listing 6.2 is correct if the following holds

$$((\mathcal{I} \models q) \wedge (\mathcal{I} \models \mathcal{C}) \wedge (q, \mathcal{C} \vdash \mathcal{C}')) \Rightarrow (\mathcal{I} \models C') \tag{6.2}$$

for each interpreation $\mathcal{I}$ as definded in [5], where C' is the set of the cardinality constraints derived from C with respect to q.

Proof. (By induction over the number m of atoms in the body of q)

- *Base Case:* $m = 1$

Since the query body b is an atom which we denote by l, i.e. q has the the form $i : h \leftarrow l$, Lines $13 - 19$ in Listing 6.2 are performed. Let $j : \exists_c l \leftarrow$ be the ESCL cardinality constraint for l, where j is the constraint validity time and c is the cardinality specification. Moreover, we define $C := \{j : \exists_c l \leftarrow\}$.

If the query evaluation time coincides with the constraint validity time (i.e. $i = j$), the same cardinality specification c is returned. We obtain the derived cardinality constraint $i : \exists_c h \leftarrow$. This result is correct, since $i = j$, $\mathcal{I} \models (j : \exists_c l \leftarrow)$ according to the predition in (6.2) and l is the only atom in b. Therefore, $\mathcal{I} \models C'$ holds, where $C' := \{i : \exists_c l \leftarrow\}$.

Otherwise, the query evaluation time is different from the constraint validity time (i.e. $i \neq j$), the function $transformCardSpec$ is called and applied to c, i and j (consider Line 19 in Listing 6.2). It returns the cardinality specification d which is the result of the transformation of c from the time interval j to the time interval i. The constraint $i : \exists_d h \leftarrow$ is derived. It is correct, since the algorithm for $transformCardSpecs$ is correct according to Theorem 6.1.4. Thus, $\mathcal{I} \models C'$ holds, where $C' := \{i : \exists_d h \leftarrow\}$.

- *Induction Assumption:* This theorem holds for each query the body of which consists of m atoms, $m \in \mathbb{N}$,

- *Induction Step:* $m \mapsto m + 1$

  i.e. the query body b consists of $m + 1$ atoms. Without restriction of generality, we assume that $b = b_1 \circ b_2$ where $\circ \in \{\wedge, \vee\}$, b_1 consists of m_1 atoms, and b_2 consists of m_2 atoms such that $m + 1 = m_1 + m_2$.

  Since $1 \leq m_1 \leq m$ (and therefore $1 \leq m_2 \leq m$), the theorem holds for both (sub)queries b_1 and b_2. Hence, the cardinality specifications returned by the functions $propagateCardSpecs(b_1)$ and $propagateCardSpecs(b_2)$ are correct. We call them d_1 and d_2. Note that the query evaluation time of b_1, b_2 is also i. Define C_1 as the set of the ESCL cardinality constraints for all atoms in b_1, and C_2 as the set of the ESCL cardinality constraints for all atoms in b_2. Moreover, define C_1' as the set of cardinality constraints derived from C_1 with respect to b_1, and C_2' as the set of cardinality constraints derived from C_2 with respect to b_2.

  Afterwards, the function $\mathcal{C}_\wedge$ (or $\mathcal{C}_\vee$) is applied to d_1 and d_2, and returns a cardinality specification which we denote by d. This (final) result is correct, since the algorithm for $\mathcal{C}_\wedge$ ($\mathcal{C}_\vee$) is correct as well. It holds that $\mathcal{I} \models (i : \exists_d h \leftarrow)$. More exactly, it holds that

$$((\mathcal{I} \models q) \wedge (\mathcal{I} \models C) \wedge (q, C \vdash C')) \Rightarrow (I \models C')$$

where $C := C_1 \cup C_2$ and $C' := C_1' \cup C_2' \cup (i : \exists_d h \leftarrow)$.

□

Theorem 6.2.5 (Completeness of propagateCardSpec).
Let q be a StreamLog query of the form $i : h \leftarrow b$ where i is the query evaluation time, h is the query head, and b is the query body. Let $l_1, .., l_m$ be the atoms in b such that $C = \{(j_1 : \exists_{c_1} l_1 \leftarrow), .., (j_m : \exists_{c_m} l_m \leftarrow)\}$ is the set of the respective ESCL cardinality constraints where $j_1, .., j_m$ are the constraint validity time intervals and $c_1, .., c_m$ are the cardinality specifications.

For each interpretation I (as defined in [5]), the following holds:

$$((I \models q) \wedge (I \models C) \wedge (I \models (i : \exists_d h \leftarrow))) \Rightarrow (q, C \vdash (i : \exists_d h \leftarrow))$$

where $(i : \exists_d h \leftarrow)$ is the cardinality constraint derived from C with respect to q by the algorithm *propagateCardSpec*(b) in Listing 6.2.

Proof. (By induction over the number m of the atoms in b)

- *Base Case*: $m = 1$

  The query body is the atom l and the set C consists of only one cardinality constraint $(j : \exists_c l \leftarrow)$. Since b is an atom, Lines $13 - 23$ of the algorithm in Listing 6.2 are performed. If $i = j$, the constraint $(i : \exists_d h \leftarrow)$ with $d = c$ is derived. Otherwise $(i \neq j)$, the cardinality specification d is returned by the function call $transformCardSpec(c, i, j)$ according to Theorem 6.1.5. Therefore, the algorithm in Listing 6.2 is complete, i.e. $q, C \vdash (i : \exists_d h \leftarrow)$.

- *Induction Assumption*: The theorem holds for all queries q the bodies of which consist of not more than m atoms.

- *Induction Step*: $m \mapsto m + 1$

  Without loss of generality, we assume that $b = b_1 \circ b_2$, where $\circ \in \{\wedge, \vee\}$, b_1 consists of m_1 atoms, b_2 consists of m_2 atoms, such that $m_1 + m_2 = m + 1$.

  Remember that the function *propagateCardSpec*(b) is recursively applied to each (nested) adjunct of b (compare Lines $4 - 5$ and $9 - 10$ in Listing 6.2) and the result of each application is then appended by $\mathcal{C}_\wedge$ or $\mathcal{C}_\vee$ to the final result (compare Lines 6 and 11). According to the induction assumption, the theorem holds for both b_1 and b_2, since $m_1 \leq m$ and $m_2 \leq m$. The cardinality specifications d_1 and d_2 are returned by the function calls *propagateCardSpec*(b_1) and *propagateCardSpec*(b_2) respectively.

Afterwards, d_1 and d_2 are combined to $d = \mathcal{C}_\wedge(d_1, d_2)$ in Line 6 if $b = b_1 \wedge b_2$ (or to $d = \mathcal{C}_\vee(d_1, d_2)$ in Line 11 if $b = b_1 \vee b_2$). The functions $\mathcal{C}_\wedge$ and $\mathcal{C}_\vee$ are complete. They define the base cases of the cardinality propagation (Tables 5.3 and 5.4).

This cardinality specification d is returned by the algorithm in Listing 6.2. Therefore, the algorithm is complete, i.e. $q, C \vdash (i : \exists_d\, h \leftarrow)$.

$\square$

Chapter 7

Simplification of Complex Cardinality Specifications

Let c be a cardinality specification. The function $simplify$ (Listing 7.1) simplifies c by building the interval c represents (Listing 7.2) and by converting this interval into the respective cardinality specification which is the simplified equivalent of c. This function is implemented as follows

```
simplify(c) {
  interval ← buildInterval(c);
  simplifiedCardSpec ← convertToCardSpec(interval);
  return simplifiedCardSpec;
}
```

Listing 7.1: Function $simplify$

```
buildInterval(c) {
  if c is simple then
     interval ← generateInterval(c);
  else
     subCardSpecs ← getAdjuncts(c);
     adjunctLeft ← subCardSpecs[0];
     adjunctRight ← subCardSpecs[1];
     intervalLeft ← buildInterval(adjunctLeft);
     intervalRight ← buildInterval(adjunctRight);
     if c is conjunction then
        interval ← buildIntersection(intervalLeft,intervalRight);
     else /* c is disjunction */
        interval ← buildUnion(intervalLeft,intervalRight);
     end end
```

```
    return interval;
}
```

Listing 7.2: The function *buildInterval* takes a cardinality specification as input and returns its respective interval as an output.

The function *getAdjuncts* takes a complex cardinality specification c as input value and returns a two-elements array which contains either conjuncts or disjuncts of c. For example, let $c = (\geq 3 \wedge \leq 5) \vee (\geq 2)$ be the complex cardinality specification. It is a disjunction. So *getAdjuncts*(c) yields the array $\{(\geq 3 \wedge \leq 5), (\geq 2)\}$ consisting of two disjuncts of c. In other words, *getAdjuncts* is a generalization of the two functions *getConjuncts* and *getDisjuncts* used in Listing 6.1.

The function *generateInterval* takes a simple cardinality specification as argument and generates the correspondent interval specified by it, according to Definition 2.1. For example, given a simple cardinality specification $c = (\geq 2)$, the function *generateInterval* will associate c with the right-open interval $[2, \infty)$.

The function *buildIntersection* takes a pair of intervals as parameters and yields the intersection of them as result. For example, given two intervals $i_1 := [3, 5]$ and $i_2 := [2, \infty)$, then *buildIntersection*$(i_1, i_2) = [3, 5] \cap [2, \infty) = [3, 5]$.

The function *buildUnion* takes analogously a pair of intervals as input and computes the union of them as output. For example, let $i_1 := [3, 5]$ and $i_2 := [2, \infty)$ be two intervals, then *buildUnion*$(i_1, i_2) = [3, 5] \cup [2, \infty) = [2, \infty)$.

The function *convertToCardSpec*(i) maps one (or many) interval(s) i to its (their) respective cardinality specification. For example, let $i := [3, 5]$ be a (closed) interval, then *convertToCardSpec*(i) returns the cardinality specification $(\geq 3 \wedge \leq 5)$ as result. If the interval i is empty, *convertToCardSpec*(i) automatically yields the default cardinality specification (≥ 0) as result. Since there are complex cardinality specifications which cannot be simplified anymore, *convertToCardSpec* has the complexity $\mathcal{O}(r)$ where r denotes the number of all subintervals in i. Moreover, it is clear that r is at most $|c|$ where $|c|$ is the number of simple cardinality specifictions included in c. We retain this information in memory to use later to determine the complexity of *simplify*.

Example 15.
Given the cardinality specification $c = ((= 3 \vee = 5) \vee (= 7))$. c cannot be simplified. Its correspondent interval is $([3, 3] \cup [5, 5] \cup [7, 7])$. The function *convertToCardSpec* is applied to these three subintervals, it converts each of them into a cardinality specification and connects all these cardinality specifications to c again. There are 3 recursive function

calls of *convertToSpec*.

According to the above descriptions, the functions *buildIntersection*, *getAdjuncts*, *buildIntersection* and *buildUnion* trivially terminate and have the complexity $\mathcal{O}(1)$. This can be proven by giving the correspondent algorithms.

Theorem 7.0.6 (Termination of simplify).
The algorithm *simplify* in Listing 7.1 terminates.

Proof. First, we justify the termination of the auxiliary function *buildInterval* from which we then infer the termination of *simplify*. The input cardinality specification c is finite, i.e. has finite number of simple cardinality specifications. If c is simple, the function *generateInterval* is directly performed and returns the final result. Since *generateInterval* terminates, *buildInterval* terminates as well. Otherwise, if c is complex, it is divided in its two adjuncts which in turn are used as inputs for recursive calls of *buildInterval*. After each recursion, the number of simple cardinality specifications in the input values (assigned to the variables *adjunctLeft* and *adjunctRight*) gradually decreases until they become simple and the recursion is broken after finite time. The result of each recursion is an interval. Afterwards, a union or an intersection of these intervals is calculated. And that is the final result. The algorithm of *buildInterval* termiates. As mentioned above, *convertToCardSpec* finishes after finite time. Therefore, the algorithm of *simplify* also terminates. □

In order to prove the complexity of *simplify*, we show Theorem 7.0.7 first.

Theorem 7.0.7.
Let T be the full (proper) binary tree [4]. Let k denotes the number of all leaves in T. So the total number of nodes in T is $|T| := 2k - 1$. Remember that each leaf is a node, but not vice versa. And the root is a (inner) node as well. If $k = 1$, then the root is the only leaf.

Proof. (By induction over k)

Let T_k be a full binary tree with k leaves. We define $|T_k|$ as the total number of nodes in T_k.

- *Base Case:* $k = 1$: The root of T_k is its only leaf, thus $|T_k| = |T_1| = 1 = 2k - 1$.
- *Induction Hypothesis:* $|T_k| = 2k - 1$ for $k \in \mathbb{N}$
- *Induction Step:* $k \mapsto k + 1$

Consider a leaf l_o of T_k. In order to insert a new leaf l_n in T_k, an additional node n is necessary such that the properties of fully binary tree are not wounded (Figure 7.1).

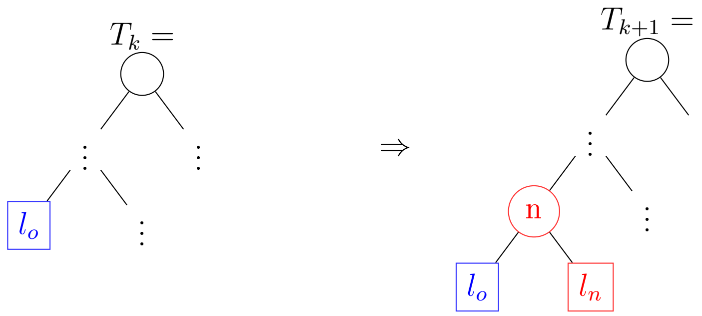


Figure 7.1: A new leaf l_n and an additional node n are inserted to T_k to obtain the bigger full binary tree T_{k+1} with $k+1$ leaves.

Since a leaf is considered as a node, two new nodes (i.e. l_n and n) are inserted in T_k to obtain the new bigger tree T_{k+1}. It is clear that T_{k+1} has $k+1$ leaves. Hence the total number of nodes of T_{k+1} is

$$|T_{k+1}| = 2 + |T_k| \stackrel{\text{I.H.}}{=} 2 + 2k - 1 = 2k + 1 = 2 \cdot (k+1) - 1$$

□

Theorem 7.0.8 (Complexity of simplify).
Let c be a cardinality specification. The algorithm *simplify* has the complexity of $O(n)$ where $n := |Simple(c)|$.

Proof. Define $\mathcal{T}(n)$ as the complexity of *simplify* where $n := |Simple(c)|$. The algorithm *simplify* is divided in two phases. First, the function *buildInterval* is called to build the respective interval of c. This phase corresponds to Line 2 in Listing 7.1. We define the complexity of *buildInterval* as $\mathcal{T}'(n)$ depending on n.

In the second phase, the function *convertToCardSpec* in Line 3 is used to transform an (or many) interval(s) into a cardinality specification. As mentioned above, the function *convertToCardSpec* has the complexity $\mathcal{O}(n)$. So the recursion equation of *simplify* is

$$\mathcal{T}(n) = \mathcal{T}'(n) + \mathcal{O}(n)$$

The algorithm of *buildInterval* itself is divided in three phases. If c is complex, then c is divided in its two conjuncts or disjuncts. This corresponds to Lines $5-7$ in Listing 7.2. Since this division phase is performed only once in each function call, it requires constant time. Therefore, it holds that $\mathcal{D}(n) = \mathcal{O}(1)$, where $\mathcal{D}(n)$ denotes the complexity of the division phase depending on n. Otherwise, if c is simple, it is directly transformed into the correspondent interval. This step requires constant time and has the cost of $\mathcal{O}(1)$, since the complexity of $generateInterval$ is $\mathcal{O}(1)$.

Next, we divide the primary problem (cost) of *buildInterval* in two partial problems, i.e. $\mathcal{T}'(n_1)$ and $\mathcal{T}'(n_2)$ where $n_1 + n_2 = n$, and solve them with two recursive calls of *buildInterval*. This phase is called trasnformation phase and corresponds to Lines $8 - 9$. The result of this phase is a pair of respective intervals.

Afterwards, a union or an intersection of these intervals is built. This phase corresponds to Line 10 (or 13) in the algorithm. We call $\mathcal{B}(n)$ its complexity. Since it is performed only once in the algorithm, it holds that $\mathcal{B}(n) = \mathcal{O}(1)$.

Line 18 is only an assigment of value. Therefore, its cost is $\mathcal{O}(1)$ and can be omitted.

So the recursion equation of *buildInterval* is

$$\mathcal{T}'(n) = \begin{cases} \mathcal{O}(1) & \text{, if } n = 1 \\ \overbrace{\mathcal{D}(n)}^{=\mathcal{O}(1)} + \mathcal{T}'(n_1) + \mathcal{T}'(n_2) + \overbrace{\mathcal{B}(n)}^{=\mathcal{O}(1)} & \text{, otherwise } n = n_1 + n_2 \end{cases}$$

or

$$\mathcal{T}'(n) = \begin{cases} \mathcal{O}(1) & \text{, if } n = 1 \\ \mathcal{O}(1) + \mathcal{T}'(n_1) + \mathcal{T}'(n_2) & \text{, otherwise, } n = n_1 + n_2 \end{cases}$$

which can be formed in

$$\mathcal{T}'(n) = \begin{cases} e & \text{, if } n = 1 \\ e + \mathcal{T}'(n_1) + \mathcal{T}'(n_2) & \text{, otherwise, } n = n_1 + n_2 \end{cases}$$

where $\mathcal{O}(1)$ is intuitively replaced by a constant e [3], and $n_1, n_2 \neq 0$.

Since the recursion equation $\mathcal{T}'(n)$ can also be illustrated by a balanced binary tree with n leaves,

$$\mathcal{T}'(n) = (2n - 1) \cdot e \in \mathcal{O}(n)$$

holds according to Theorem 7.0.7. This leads to

$$\mathcal{T}(n) = \mathcal{T}'(n) + \mathcal{O}(n) = \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$$

□

Theorem 7.0.9 (Correctness of buildInterval).
The algorithm in Listing 7.2 is correct for each cardinality specification.

Proof. Let c be a cardinality specification, $m := |Simple(c)|$. Without proof, we assume that the algorithms for *getAdjuncts*, *buildIntersection* and *buildUnion* are correct. If $m = 1$ (i.e. c is a simple cardinality specification), the result (interval) returned by the fuction $generateInterval(c)$ is trivially correct. Otherwise, the correctness can be proven by induction over m where the different structures of c are considered, simliar to the induction step in the proof of Theorem 6.1.4. □

Theorem 7.0.10 (Correctness of simplify).
The algorithm in Listing 7.1 is correct for each cardinality specification.

Proof. Let c be a cardinality specification. Without proof, we assume that the algorithm for *convertToCardSpec* is correct. Hence, Theorem 7.0.9 implies the correctness of the algorithm $simplify$. □

Chapter 8

Conclusions and Future Work

In this work, the propagation of ESCL cardinality constraints with respect to StreamLog queries is formally defined. The approach is novel and language independent in general. The propagation is defined in two steps:

1. The first step considers the case in which the validity time intervals of cardinality constraints coincide with the evaluation time interval of the query with respect to which the constraints are propagated. This algorithm is also applicable to database constraints and queries since in database systems there is usually no dependency from time, in contrast to CEP.

2. The second step treats the case in which the validity time intervals of cardinality constraints differ from the evaluation time interval of the query with respect to which the constraints are propagated. This algorithm is typical for CEP.

Query evaluation time and constraint validity time can be specified by a tumbling window, a sliding window or a time interval. The time interval can be repeating or non-repeating. It covers (relative timer) events and states, an unbounded window, a now window, and a time interval the bounds of which are specified by functions.

The cardinality specifications of propagated constraints may be complex, i.e. be an arbitrary nested conjunction or disjunction of simple cardinality specifications. The body of the query with respect to which the constraints are propagated may be an arbitrary nested conjunction or disjunction of atoms.

The base cases of cardinality constraint propagation are defined in the form of tables. The inductive cases are defined by algorithms. The algorithms are proven to terminate, to be correct and complete with respect the the declarative semantics of ESCL and StreamLog defined in [5]. The complexity of each algorithm is investigated. Both base and inductive cases are illustrated by examples.

There are the following future research diresctions which are based on this report:

1. Propagation of ESCL cardinality constraints independently from the assumptions described in Chapter 4, in particular, this propagation must work also with respect to StreamLog queries the bodies of which contain negative literals

2. Propagation of ESCL causal, temporal, and data constraints with respect to StreamLog queries

3. Derivation of ESCL constraints from StreamLog queries independently from other ESCL constraints

4. Implementation and experimental evaluation of the whole approach

The second and the third research directions are the most interesting and promising. They are described in more detail in the following.

Propagation of ESCL Causal, Temporal, and Data Constraints with Respect to StreamLog Queries

Analogously to the propagation of ESCL cardinality constraints, causal, temporal, and data constraints will be propagated with respect to StreamLog queries. The StreamLog rule in Listing 8.1 will illustrate the intuition behind the propagation of these constraints. The rule identifies the items profitably sold during an auction, i.e. their selling price is higher than the double of their start price. (Consider the detailed elaboration of the online auction use case in [5].)

```
auction(auctionID(A)):
  profitablySoldItem(auctionID(A), itemID(I)) ←
    itemDescription(auctionID(A), itemID(I), bidderID(B1), value(V1)) ∧
    sell(auctionID(A), itemID(I), bidderID(B2), value(V2)) ∧
    V2 > 2 · V1
```

Listing 8.1: StreamLog rule identifying profitably sold items during an auction

Causal Constraints

Whether there is at least one item description during an auction depends on whether at least two bidders are enrolled for the auction, i.e. *item description* events are caused by at least two *bidder enrollment* events. Whether an item is sold depends on whether there is at least one bid for it, i.e. *sell* events are caused by *bid* events. However, *profitablySoldItem* events are caused not only by *bidder enrollment* and *bid* events but also by the price difference of respective *item description* and *sell* events. Without the condition in Line 5 of Listing 8.1, the causal constraint could be derived for the *profitablySoldItem* events.

Temporal Constraints

Since the occurrence time of the derived event $profitablySoldItem$ comprises the occurrence time intervals of the events it was derived from (i.e., *itemDescription* and *sell*) and since description of an item always precedes its sell, all events and states happening before *itemDescription* events precede $profitablySoldItems$ events and all events and states following *sell* events happen after $profitablySoldItems$ events.

Data Constraints

Since there is a functional dependency of the auction identifier from the item identifier in all *itemDescription* and *sell* events and since both respective attributes are carried by the derived *profitablySoldItem* events, this functional dependency holds also for the *profitablySoldItem* events.

Derivation of ESCL Constraints from StreamLog Queries

Some constraints can be derived from CEP queries independently from constraints. Consider the following StreamLog rule computing the number of bids per item during an auction.

```
auction(auctionID(A)):
   bidNumber(auctionID(A), itemID(I), number(count(A,I))) ←
      bid(auctionID(A), itemID(I), bidderID(B), value(V))
```

Listing 8.2: StreamLog rule computing the number of bids per item during an auction

The value of the attribute *number* of the derived events depends on the number of bids per item in each auction. There is exactly one *bidNumber* event for each item with bids during an auction. Therefore the following ESCL constraint can be derived:

```
WHILE auction(auctionID(A))
LET
   IF ∃=k bid(auctionID(A), itemID(I), bidderID(B), value(V))
        group−by {A, I}
   THEN ∃=1 bidNumber(auctionID(A), itemID(I), number(N)) ∧ N = k
   END
END
```

Listing 8.3: ESCL constraint derived from the StreamLog rule in Listing 8.2.

Bibliography

[1] François Bry and Michael Eckert. On Static Determination of Temporal Relevance for Incremental Evaluation of Complex Event Queries. In *Proc. Int. Conf. on Distributed Event-Based Systems*, volume 332, pages 289–300. ACM, 2008.

[2] Upen S. Chakravarthy, John Grant, and Jack Minker. Logic-based Approach to Semantic Query Optimization. volume 15, pages 162–207. ACM, 1990.

[3] Th. H. Cormen, Ch. E. Leiserson, R. Rivest, and C. Stein. *Algorithmen - Eine Einführung*. Oldenbourg, 2nd edition, 2001.

[4] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java*. John Wiley & Sons, Inc., 5th edition.

[5] Olga Poppe. A Method for Semantic Optimization of Complex Event Processing. Research report, Institute for Informatics, University of Munich, 2011.

[6] Aaron M. Tenenbaum, Yedidyah Langsam, and Moshe J. Augenstein. *Data structures using C*. Prentice Hall, 1990.