I4-D15b

Scalable, Space-optimal Implementation of Web Queries Part 2: CIQCAG Algebra and Evaluation

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R/P (report and prototype)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D15b/D/PU/a1
Responsible editors:	Tim Furche
Reviewers:	Benedikt Linse and Georg Gottlob
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	31 August 2007
Actual submission date:	15 January 2008

Abstract

Tree queries and tree data have proved to be interesting limitations on the quest to scalable query evaluation. In this deliverable, we explore two directions on how to extent the results from tree query languages such as XPath beyond tree data: (1) Can we find an interesting and practically relevant class of (data) graphs that is a proper superset of trees, yet to which algorithms such as twig joins, so far limited to trees or DAGs, can be extended without affecting (time and space) complexity? (2) Second, we observe that data and, particularly, queries are often mostly trees with only limited non-tree parts. However, any non-tree part makes most of the techniques discussed above for tree queries inapplicable. Can we integrate the above technologies in the processing of general graph queries in such a way that (often significant) hierarchical components can be evaluated using polynomial algorithms, limiting the degradation of query complexity to non-tree parts of the query?

We answer both questions essentially positively by introducing a novel algebra, called CIQCAG, the **c**ompositional, **i**nterval-based **q**uery and **c**omposition **a**lgebra for **g**raphs. CIQCAG is a fully algebraic approach to querying Web data (be it in XML, RDF, or other semi-structured shape) that is build around two central contributions: (1) A novel *characterization of (data) graphs* admissible to interval-based compaction. For this new class of data, called **continuous-image graphs** (or CIGs for short), we can provide linear-space and almost linear-time algorithms for evaluating tree queries rivaling the best known algorithms for tree data. (2) An algebra for a two-phase evaluation of queries separating a tree core of the query from the remaining non-tree constraints. The operations of the algebra closely mirror relational algebra (and can, in fact, be implemented in standard SQL), but a novel *data structure* influenced by Xcerpt's memoization matrix and the complete answer aggregates approach allows for exponentially more succinct storage of intermediate results for the tree core of a query. Together with a set of operators on this data structure, called **sequence map**, this enables CIQCAG to evaluate almost-tree queries with nearly polynomial time limiting the degradation in performance to non-tree parts of the query.

Keyword List
algebra,complexity,Xcerpt,XPath,XQuery,SPARQL,queries,data model,semantics

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Scalable, Space-optimal Implementation of Web Queries Part 2: CIQCAG Algebra and Evaluation

Tim Furche[1], **François Bry**[1]

[1] Institute for Informatics, University of Munich, Germany
`http://pms.ifi.lmu.de`

15 January 2008

Abstract
Tree queries and tree data have proved to be interesting limitations on the quest to scalable query evaluation. In this deliverable, we explore two directions on how to extent the results from tree query languages such as XPath beyond tree data: (1) Can we find an interesting and practically relevant class of (data) graphs that is a proper superset of trees, yet to which algorithms such as twig joins, so far limited to trees or DAGs, can be extended without affecting (time and space) complexity? (2) Second, we observe that data and, particularly, queries are often mostly trees with only limited non-tree parts. However, any non-tree part makes most of the techniques discussed above for tree queries inapplicable. Can we integrate the above technologies in the processing of general graph queries in such a way that (often significant) hierarchical components can be evaluated using polynomial algorithms, limiting the degradation of query complexity to non-tree parts of the query?

We answer both questions essentially positively by introducing a novel algebra, called CIQCAG, the **c**ompositional, **i**nterval-based **q**uery and **c**omposition **a**lgebra for **g**raphs. CIQCAG is a fully algebraic approach to querying Web data (be it in XML, RDF, or other semi-structured shape) that is build around two central contributions: (1) A novel *characterization of (data) graphs* admissible to interval-based compaction. For this new class of data, called **continuous-image graphs** (or CIGs for short), we can provide linear-space and almost linear-time algorithms for evaluating tree queries rivaling the best known algorithms for tree data. (2) An algebra for a two-phase evaluation of queries separating a tree core of the query from the remaining non-tree constraints. The operations of the algebra closely mirror relational algebra (and can, in fact, be implemented in standard SQL), but a novel *data structure* influenced by Xcerpt's memoization matrix and the complete answer aggregates approach allows for exponentially more succinct storage of intermediate results for the tree core of a query. Together with a set of operators on this data structure, called **sequence map**, this enables CIQCAG to evaluate almost-tree queries with nearly polynomial time limiting the degradation in performance to non-tree parts of the query.

Keyword List
algebra,complexity,Xcerpt,XPath,XQuery,SPARQL,queries,data model,semantics

Contents

Chapter 1

Principles and Motivation

1.1 Introduction

We assign meaning to things by enumerating their features (or properties or attributes) and placing them in relation with other things. This enables us to distinguish, classify, and, eventually, act upon such things. The same applies to digital data items: to find, analyse, classify, and, eventually, use as basis for actions we need to place data items in relation to other data items: A book to its author, a bank transaction to the bank, the source and the target of the transaction, a patient to its treatment history, its doctor, etc.

How we describe these relations between data items (as well as their features) is the purview of data models. Recently the relational data model, tailored to relations of arbitrary shape, has been complemented by semi-structured data models tailored to Web data. What sets these data models apart from relational data is an even greater focus on relations or links while delegating features or attributes to second-class citizens or dropping them entirely, as in RDF. At the same time most of these data models share a strong hierarchical bias: XML is most often considered tree data.[1] RDF and other ontology languages allow arbitrary graphs but ontologies often have dominantly hierarchical "backbones", formed, e.g., by subclass or part-of relations. To summarize, *structure* is a central property of data and data models determine what shape those structures may take.

When we want to actually *do* something with the data represented in any of these data models, we use queries. Again, exploiting the relations among sought-for data items is essential: to find all authors of books on a given topic, to find the bank with the highest transaction count, to identify an illness by analysing patterns in a patient's health records. Thus queries mirror the structure of the sought-for data, though often with a richer vocabulary, allowing, e.g., for recursive relation traversal or don't-care parts: a patient is chronically ill if there is some illness (we don't care which illness) that recurs regularly in that patient's health records. Queries may contain additional derived (i.e.., not explicitly represented or "extensional") relations—such as equalities or order between the value of data items. The shape of a query is, thus, not limited to the shape of the data but may contain additional relations. Only if we limit a query to extensional relations must the shape of the query mirror (a substructure) of the shape of the data. For instance, when querying tree data queries may take the only the shape of trees, if we allow only

[1]Though ID-links justify a more graph-like view of XML.

extensional relations[2], but may have arbitrary shape if we allow derived relations. To summarize, as for data, structure is a central feature of queries and mirrors the structure of the sought-for data. However, the structure of a query is linked to the structure of the query only if we consider exclusively extensional relations in the query.

The reason we should care about the shape of the data or queries we are working with is a growing canon of approaches that obtain better complexity and performance for query evaluation if certain limits are imposed on the shape of data, queries, or both.

If we consider arbitrary data, we have little reliable means for compacting relation information. On tree data, in contrast, we can use any number of encodings, e.g., interval encodings [16, 15], hierarchical or path-based labeling [47], or schemes based on structural summaries [54]. In essence, these encodings exploit the observation that structural relations in trees follow certain rules, e.g., each node has a unique parent, the descendants of each node are contained in the descendants of all its ancestors, each node has a unique following and preceeding sibling, etc. Interval encodings on trees, e.g., allow us to compact closure relations quadratic in the tree size into a linear size interval encoding.

For queries, we can make a similar observation: if we allow arbitrary "links" in a query, we need to manage relations between bindings for all nodes in the query at once. However, the relations between the nodes may be limited, e.g., if the query is tree shaped, bindings for each node are directly related only to bindings of its "parent". In fact, if we consider the answers to a query as a relation with the nodes as columns, answers of a tree-shaped query always exhibit multivalued dependencies [18]: In fact, we can normalize or decompose such a relation for a query with n nodes into $n-1$ separate relations that together faithfully represent the original relation (*lossless-join* decomposition to binary relations over each pair of adjacent variables in the query). This allows us to compact an otherwise potentially exponential answer (in the data size) into a polynomial representation.

Neither observation is particularly new: acyclic or tree queries on relational data as interesting polynomial-time subclass have been studied, e.g., in [55] and [26]. More recently, the increasing popularity of Web data such as XML triggered renewed interest and reinvestigation of tree data and tree queries as interesting restrictions of general relational structures and queries. Several novel techniques tailored to XML data and XPath or similar tree queries have shown the benefit of exploiting the hierarchical nature of the data for efficient query evaluation: polynomial twig joins [8]; XPath evaluation [24]; polynomial evaluation of tree queries against XML streams [4, 45, 44], linear tree labeling schemes [29, 54] allowing constant time access to structural closure relations such as descendant or following; and path indices [12] enabling constant time evaluation of path queries.

As stated, these techniques have received considerable attention when data and queries are tree shaped. However, data often contains some non-tree aspects, even if the tree aspects are dominant, e.g., ID-links in XML or many ontologies (such as the GeneOntology [17]). Practical queries (such as XQuery or Xcerpt) often go beyond tree queries, e.g., to express value or identity joins, even if they contain a majority of structural conditions. Driven by such considerations, interest in adapting above techniques beyond trees is growing (e.g., labeling and reachability for graph data [53, 52] or hypertree decomposition for polynomial queries beyond trees [25]).

Therefore, we explore in this work means of building on the above mentioned techniques but pushing

[2] Even allowing transitive closure on extensional relations allows already for queries where there are, e.g., two paths between two query nodes. However, as shown in [46] for XPath and in [43] for the general case of tree queries, such graph shaped queries can always be reduced to sets of tree-shaped queries—though potentially for the cost of an exponential increase in query size.

them beyond trees. We orient this exploration along the following two questions:

(1) Can we find an interesting and practically relevant class of (data) graphs that is a proper superset of trees, yet to which algorithms such as twig joins [8], so far limited to trees or DAGs [11], can be extended without affecting (time and space) complexity?

(2) Second, we observe that data and, particularly, queries are often mostly trees with only limited non-tree parts. However, any non-tree part makes most of the techniques discussed above for tree queries inapplicable. Can we integrate the above technologies in the processing of general graph queries in such a way that (often significant) hierarchical components can be evaluated using polynomial algorithms, limiting the degradation of query complexity to non-tree parts of the query?

In the following, we answer both questions essentially positive by introducing a novel algebra, called CIQCAG, the **c**ompositional, **i**nterval-based **q**uery and **c**omposition **a**lgebra for **g**raphs. CIQCAG is a fully algebraic approach to querying Web data (be it in XML, RDF, or other semi-structured shape) that is build around two central contributions:

(1) a novel *characterization of (data) graphs* admissible to interval-based compaction. For this new class of data, called **continuous-image graphs** (or CIGs for short), we can provide linear-space and almost linear-time algorithms for evaluating tree queries rivaling the best known algorithms for tree data.

(2) An algebra for a two-phase evaluation of queries separating a tree core of the query from the remaining non-tree constraints. The operations of the algebra closely mirror relational algebra (and can, in fact, be implemented in standard SQL), but **(a)** a novel *data structure* influenced by Xcerpt's memoization matrix [49, 10] and the complete answer aggregates approach [41] allows for exponentially more succinct storage of intermediate results for the tree core of a query. Together with a set of operators on this data structure, called **sequence map**, this enables CIQCAG to **(b)** evaluate almost-tree queries with nearly polynomial time limiting the degradation in performance to non-tree parts of the query. **(c)** Finally, the algebra is tailored to be agnostic of the actual realization of the used relations. This makes it particularly easy to integrate approaches for *arbitrary derived relations* and *indices* in addition to extensional relations, reachability indices such as interval labeling [28] for tree data or [52] for graph data and path indices such as DataGuides [22] or [12] for tree data.

In the following chapters, we focus on the basic query and construction algebra which covers non-recursive, single-rule Xcerpt as well as non-compositional XQuery (as defined in [39]), see . However, in Section 3.9 we briefly discuss an extension of this algebra with an iteration operator. This extension allows the evaluation of full Xcerpt and a considerably larger fragment of XQuery than currently covered.

Before we begin the formal introduction of algebra in Chapters 2 and 3, the remainder of this chapter serves to give a first intuition of the CIQCAG algebra. First, we briefly (Section 1.2) introduce continuous-image graphs as a class of graph data where the proposed algebra performs as well as the best approaches for tree data. This is achieved using the sequence map data structure, a novel representation of intermediary results of tree queries (or tree cores of graph queries), introduced in Section 1.3. This data structure is embedded into the CIQCAG algebra in Section 1.4 where we give a first glance at the structure of the algebra and its evaluation phases. We wrap up this introduction with a brief overview of complexity results for the discussed algebra, as well as a first comparison with existing approaches in Section 1.5.

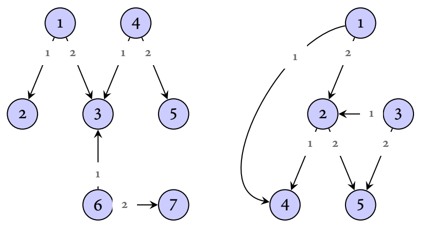
Figure 1. Sharing: On the Limits of Continuous-image Graphs

1.2 Data Beyond Trees: Continuous-Image Graphs

Tree data, as argued above, allows us to represent relations on that data more compact, e.g., using various interval-based labeling schemes. Here, we introduce a new class of graphs, called *continuous-image graphs* (or CIGs for short), that generalize features of tree data in such a way that we can evaluate (tree) queries on CIGs with the same time and space complexity as techniques such as twig joins [8] which are limited to tree data only.

Continuous-image graphs are a proper superset of (ordered) trees where we require not that each node has at most one parent, as we do for trees, but that we can find a single order on all nodes of the graph such that the children of each parent form a continuous interval in that order. A formal definition is given in Chapter 2. This definition allows graphs where some or all children of two parents are "shared" but limits the degree of sharing: Figure 1 shows two minimal graphs that are *not* CIGs. Incidentally, both graphs are acyclic and, if we take away any one edge in either graph, the resulting graph becomes a CIG. The second graph is actually the smallest (w.r.t. number of nodes and edges) graph that is not a CIG. The first is only edge minimal but illustrates an easy to grasp sufficient but not necessary condition for violating the interval property: if a node has at least three parents and each of the parents has at least one (other) child not shared by the others then the graph can not be a CIG.

On continuous-image graphs we can exploit similar techniques for compacting structural relations as on trees, most notably representing the nodes related to a given node as a single, continuous interval and thus with constant space. This applies also for derived relations such as closure (XPath's descendant) or order relations (XPath's following-sibling) on CIGs.

Moreover, whether a given graph is a CIG (and in what order its node must be sorted to arrive at continuous intervals for each parent's children) is just another way of saying that its adjacency matrix carries the *consecutive ones* property [19]. For the consecutive-ones problem [7] gives the first linear time (in the size of the matrix) algorithm based on so called PQ-trees, a compact representation for permutations of rows in a matrix. More recent refinements in [31] and [35] show that simpler algorithms, e.g., based on the PC-tree [34], can be achieved. We *adapt* these algorithms to obtain a linear time (in the size of the adjacency matrix) algorithm for deciding whether a graph is a CIG and computing a CIG-order.

From a practical perspective, CIGs are actually quite common, in particular, where time-related or hierarchical data is involved: If relations, e.g., between Germany and kings, are time-related, it is quite likely that there will be some overlapping, e.g., for periods where two persons were king of Germany at the same time. Similarly, hierarchical data often has some limited anomalies that make a modelling

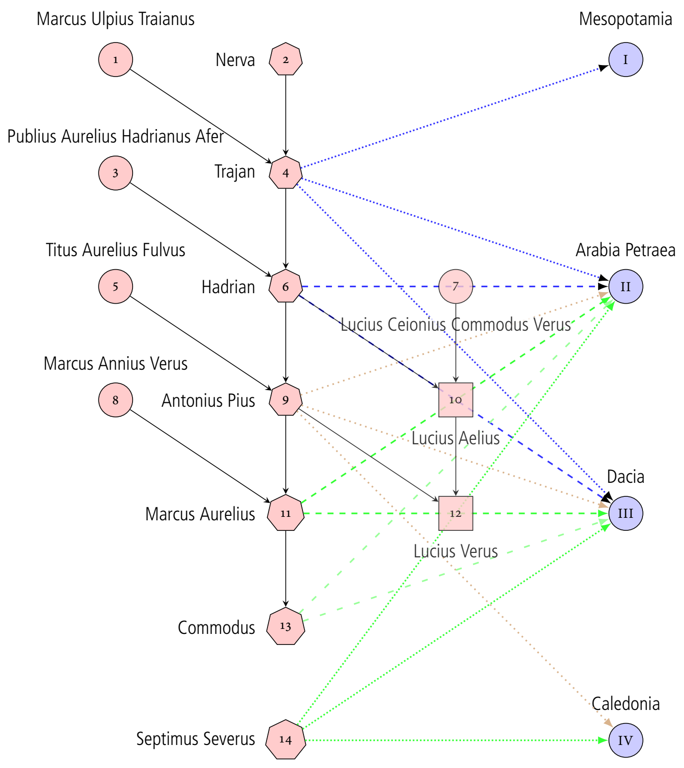

Figure 2. “The Five Good Emperors” (after Edward Gibbon), their relations, and provinces.

as strict tree data impossible. Figure 2 shows actual data[3] on relations between the family (red nodes, non-ruling member ①, co-emperor or heir designate $\boxed{10}$, emperors 2) of the Roman emperors in the time of the "Five Good Emperors" (Edward Gibbon) in the 2nd century. It also shows, for actual emperors, which of the four new provinces (Ⓘ) added to the roman empire in this period each emperor ruled (the other provinces remained mostly unchanged and are therefore omitted). Arrows between family members indicate, natural or adoptive, fathership[4]. Arrows between emperors and provinces show rulership, different colors are used to distinguish different emperors. Despite the rather complicated shape of the relations (they are obviously not tree-shaped and there is considerable overlapping, in particular w.r.t. province rulership).

1.3 Sequence Map: Structure-aware Storage of Tree Core Results

When we evaluate tree queries, we can observe that for determining matches for a given query node only the match for its parent and child in the query tree are relevant. Intuitively, this "locality" property holds as in a tree there is at most one path between two nodes. To illustrate, consider, e.g., the XPath query //a//b//c selecting c descendants of b descendants of a's. Say there are n a's in the data nested into each other with m b's nested inside the a's and finally inside the b's (again nested in each other) l c's. Then a naive evaluation of the above query considers all triples (a, b, c) in the data, i.e., $n \times m \times l$ triples. However, whether a c is a descendant of a b is independent of whether a b is a descendant of an a. If a b is a descendant of several a's makes no difference for determining its c descendants. It suffices to determine in at most $n \times m$ time and space all b's that are descendants of a, followed by a separate determination of all c's that are descendants of such b's in at most $m \times l$ time and space.

Indeed, if we consider the answer relation for a tree query, i.e., the relation with the complete bindings as rows and the query's nodes as columns, this relation always exhibits multivalue dependencies [18]: We can normalize or decompose such a relation for a query with n nodes into $n - 1$ separate relations that together faithfully represent the original relation (and from which the original relation can be reconstructed using $n - 1$ joins). This allows us to compact an otherwise potentially exponential answer (in the data size) into a polynomial representation.

This is the first principle of the **sequence map**: decompose the query into separate binding sequences for each query node with "links" or pointers relating bindings of different nodes. We thus obtain an exponentially more succinct data structure for (intermediary) answers of tree queries than if using standard (flat) relational algebra. In this sense, a sequence map can be considered a fully decomposed *column store* for the answer relation.

[3]The name and status of the province between the wall of Hadrian and the wall of Antonius Pius in northern Britain is controversial. For simplicity, we refer to it as "Caledonia", though that actually denotes all land north of Hadrian's wall.

[4]Note that all emperors of the Nervan-Antonian dynasty except Nerva and Commodus were adopted by their predecessor and are therefore often referred to as "Adoptive Emperors".

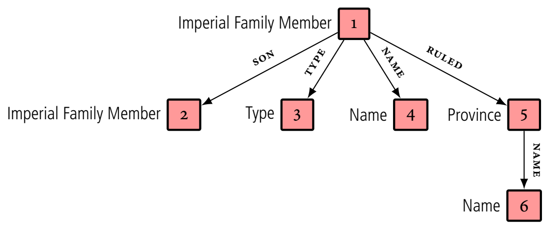

Figure 3. Selecting sons, type, name, and ruled provinces for all members of the imperial family in the data of Figure 2.

Imp-ID	Type	Name	Son-ID	Ruled-ID	Ruled-Name
1	non-ruling	Marcus Ulpius Traianus	4	–	–
2	augustus	Nerva	4	–	–
3	non-ruling	P. Aurelius Hadrianus Afer	6	–	–
4	augustus	Trajan	6	I	Mesopotamia
4	augustus	Trajan	6	II	Arabia Petraea
4	augustus	Trajan	6	III	Dacia
5	non-ruling	Titus Aurelius Fulvus	9	–	–
6	augustus	Hadrian	9	II	Arabia Petraea
6	augustus	Hadrian	10	II	Arabia Petraea
6	augustus	Hadrian	9	III	Dacia
6	augustus	Hadrian	10	III	Dacia
7	non-ruling	L. Ceionius Commodus Verus	10	–	–
8	non-ruling	M. Annius Verus	11	–	–
9	augustus	Antonius Pius	11	II	Arabia Petraea
9	augustus	Antonius Pius	12	II	Arabia Petraea
9	augustus	Antonius Pius	11	IIi	Dacia
9	augustus	Antonius Pius	12	III	Dacia
9	augustus	Antonius Pius	11	IV	Caledonia
9	augustus	Antonius Pius	12	IV	Caledonia
10	caesar	Lucius Aelius	12	–	–
11	augustus	Marcus Aurelius	13	II	Arabia Petraea
11	augustus	Marcus Aurelius	13	III	Dacia
12	caesar	Lucius Verus	–	–	–
13	augustus	Commodus	–	II	Arabia Petraea
13	augustus	Commodus	–	III	Dacia
14	augustus	Septimus Severus	–	II	Arabia
14	augustus	Septimus Severus	–	III	Arabia
14	augustus	Septimus Severus	–	IV	Caledonia

Figure 4. Answers for query from Figure 3, single, flat relation.

Imp-ID	Type	Name
1	non-ruling	Marcus Ulpius Traianus
2	augustus	Nerva
3	non-ruling	P. Aurelius Hadrianus Afer
4	augustus	Trajan
5	non-ruling	Titus Aurelius Fulvus
6	augustus	Hadrian
7	non-ruling	L. Ceionius Commodus Verus
8	non-ruling	M. Annius Verus
9	augustus	Antonius Pius
10	caesar	Lucius Aelius
11	augustus	Marcus Aurelius
12	caesar	Lucius Verus
13	augustus	Commodus
14	augustus	Septimus Severus

Imp-ID	Son-ID	Imp-ID	Prov-ID
1	4	4	I
2	4	4	II
3	6	4	III
4	6	6	II
5	9	6	III
6	9	9	II
6	10	9	III
7	10	9	IV
8	11	13	II
9	11	13	III
9	12	14	II
10	12	14	III
11	13	14	IV

Prov-ID	Name
I	Mesopotamia
II	Arabia Petraea
III	Dacia
IV	Caledonia

Figure 5. Answers for query from Figure 3, multiple relations, normalized, no multivalue dependencies.

To illustrate this, consider the query in Figure 3 on the data of Figure 2. The query selects sons and ruled provinces of members of the imperial family. We also record type and name of the family member and name of the province to easier talk about the retrieved data. The answers for such a query, if expressed, e.g., in relational algebra or any language using standard, flat relations to represent n-ary answers, against the data from Figure 2 yields the flat relation represented in Figure 4. As argued above, we can detect multivalue dependencies and thus redundancies, e.g., from emperor to province, from province to province name, from emperor (Imp-ID) to type and name.

To avoid these redundancies, we first decompose or normalize this relation along the multivalue dependencies as in Figure 5. For the sequence map, we use always a full decomposition, i.e., we would also partition type and name into separate tables as in a column store.

1.3.1 Sequence Map for Trees and Continuous-Image Graphs

Once we have partitioned the answer relation into what subsumes to only link tables as in column stores, we can observe even more regularities (and thus possibilities for compaction) if the underlying data is a tree or continuous-image graph. Look again at the data in Figure 2 and the resulting answer representation in Figure 5: Most emperors have not only ruled one of the new provinces Mesopotamia, Arabia Petraea, Dacia, and Caledonia but several. However, since the data is a continuous-image graph there is an order (indeed, the order of the province IDs if interpreted as roman numerals) on the provinces such that the provinces ruled by each emperor form a continuous interval w.r.t. that order. Thus we can actually represent the same information much more compact using interval pointers or links as in Figure 6 where we do the same also for the father-son relation (although there is far less gain since most emperors already have only a single son).

Instead of a single relation spanning 28 rows and 6 columns (168 cells), we have thus reduced the information to $5 \cdot 2 + 11 \cdot 2 + 14 \cdot 3 = 74$ cells. This compaction increases exponentially if there are longer paths in a tree query (e.g., if the provinces would be connected to further information not related to the emperors). It increases quadratically if with the increasing size of the tables, e.g., if we added the remaining n provinces of the Roman empire ruled by all emperors in our data we would end up with $7 \cdot n$ additional rows of 6 columns in the first case (each of the 7 emperors in our data ruled all these provinces), but only $n \cdot 2$ additional cells when using multiple relations and interval pointers. A detailed study of the space complexity of sequence maps is given in Section 2.4 and summarized below in Section 1.5.

Formally, we introduce the sequence map $\mathsf{SM}^{D\to Q}$ on a relational structure D and a query Q in Chapter 2 as a mapping from a set of query nodes V to sequences of matches for that query node. A *match* for query

Imp-ID	Son Range
1	4
2	4
3	6
4	6
5	9
6	9–10
7	10
8	11
9	11–12
10	12
11	13

Imp-ID	Prov Range
4	I–III
6	II–III
9	II–IV
13	II–III
14	II–IV

Figure 6. Answers for query from Figure 3, multiple relations, interval pointers. The first table from Figure 5 remains unchanged.

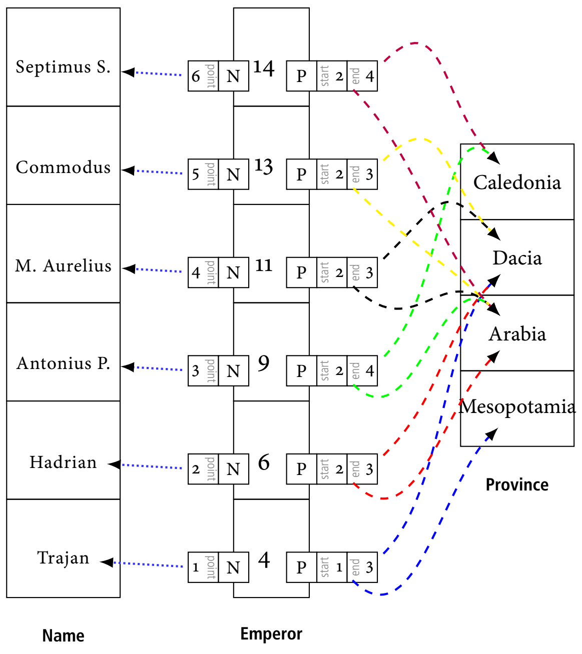

Figure 7. Sequence Map: Example. For a query selecting roman emperors together with their name and ruled provinces on the data of Figure 2.

1.3.2 Sequence Maps for Diamond-Free DAG Queries

Beyond tree queries, the sequence map data structure can also be used, with a slight modification, for *diamond-free DAG queries*. Diamond-free DAG queries are queries in the shape of a DAG where there are no two distinct paths between two nodes (and thus no diamond-shaped sub-graph). Diamond-free DAG queries are also used in [45, 42] (there named single-join DAG queries).

If we consider diamond-free DAG queries, there may be nodes in the query that have multiple parents. However, as for tree queries only the parent and child nodes are relevant to decide whether a data item is a match for a query node. The multiple parents, however, may be connected using different relations, e.g., XPath's child, descendant, and following relations. Above, we only demand that for each relation the continuous-image property of the data graph holds. If we have DAG queries this might lead to different, possibly incompatible orders for the image relations (e.g., child needs a breadth-first order to obtain continuous-images vs. depth-first order for descendant). However, we do not need to "strengthen" the CIG property for diamond-free DAG queries. Rather, a node with multiple parents carries a separate "order" number for each different incoming relation with incompatible orders and interval pointers are resolved as range queries over the appropriate order number. Though this increases the space need this is justified by the accompanying increase in query size. For details, see Section 1.3.2.

1.3.3 Representing intermediary results: A Comparison

As stated above, the sequence map is heavily influenced by previous data structures for representing intermediary answers of tree queries. Figure 8 shows the most relevant influences. Complexity and supported data shapes are compared in Section 1.5 below after discussing the actual evaluation of tree queries using the sequence map data structure. Here, we illustrate that the above discussed choices when designing a data structure for intermediary answers of tree queries are actually present in many related systems: We can find systems such as Xcerpt 1.0 [49], many early XPath processors (according to [23]), and tree algebras such as TAX [37] that use exponential size for storing all combinations of matches for each query node explicitly. [23] shows that XPath queries can in fact be evaluated in polynomial time and space, which is independently verified in SPEX [44], the first streaming processor for navigational XPath with all structural axes. Like SPEX and CIQCAG, complete answer aggregates [40] use interval compaction for relating matches between different nodes in a tree query. CAAs are also most closely related to CIQCAG w.r.t. the decomposition of the answer relation: fully decomposed without multivalue dependencies. In contrast, Pathfinder [6] uses standard relational algebra but for the evaluation of structural joins a novel staircase join [30] is employed that exploits the same interval principles used in CAAs and CIQCAG.

Streaming or cursor-based approaches such as twig join approaches [8, 11] consider the data in a certain order rather than all at once. In such a model, it is possible and desirable to skip irrelevant portions of the input stream (or relations) and to prune partial answers as soon as it is clear that we can not complete such answers. Recent versions of SPEX [9, 44] contain as most twig join approaches, mechanisms to skip over parts of the stream (at least for some query nodes) if there can not be a match (e.g., because there is no match for a parent node and we know that matches for parent nodes must come before matches for child nodes). Both twig joins and SPEX also prune results as soon as possible. However, twig joins are limited to vertical relations (child and descendant) whereas SPEX and CIQCAG can evaluate all XPath axes, though only on tree data. In Section 3.9 we discuss briefly a cursor interface for the CIQCAG algebra that

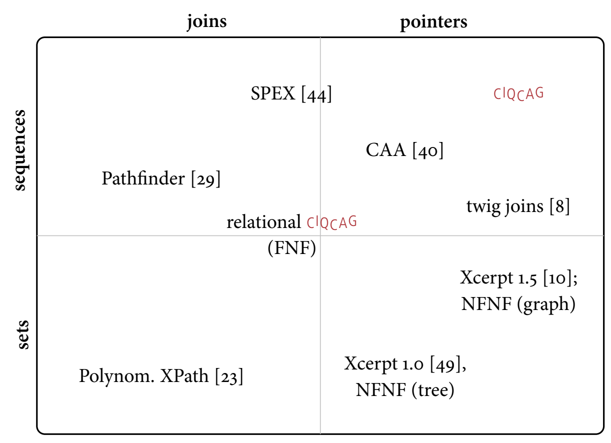

Figure 8. Data structures for intermediary results (of a tree query)

iterates over the basic relations in order (of storage or CIG property). In general graphs, however, skipping and pruning is impossible, since we can never be sure that there are no more related matches. The CIG property is also too weak to ensure that we can skip and prune as match as on tree data. However, in Section 3.9.1 we give a condition on the CIG-order of parent and child nodes in a tree query that, if it holds, ensures that these orders are "compatible" enough to allow as much skipping and pruning as in twig join approaches. This condition is more general than basic tree data, but more restrictive than the CIG property discussed above.

To summarize, though CIQCAG's sequence map is similar in its principles to several of the related approaches in Figure 8, it combines *efficient intermediary answer storage* as in CAAs with *fully algebraic* processing as in Pathfinder and *efficient skipping and pruning* as in twig joins.

Furthermore, where most of the related approaches are limited to tree data (with the notable exception of Xcerpt), CIQCAG allows processing of *many graphs*, viz. CIGs, as efficient as previous approaches allow for trees.

The sequence map data structure is exploited in the CIQCAG algebra for processing both tree and arbitrary graph queries. CIQCAG takes advantage of sequence maps to store intermediary results for the entire or for the tree core of a query as described in the following section.

1.4 Queries Beyond Trees: Graphs with Tree Core

The sequence map enables us to store (intermediary) answers to tree queries efficiently, in particular if the data is CIG-shaped. However, what if the query is not of tree shape?

Our answer to that question is the CIQCAG algebra. Instead of treating only tree queries (like the above mentioned approaches for XML) or entirely dropping the advantages tree queries offers (as standard relational algebra does), CIQCAG separates query processing in two phases: in the first phase we exploit *efficient algorithms for tree queries* evaluating a *tree core* of the original query. Any remaining parts of the query, if any, are evaluated on top of the resulting sequence map from the tree core evaluation.

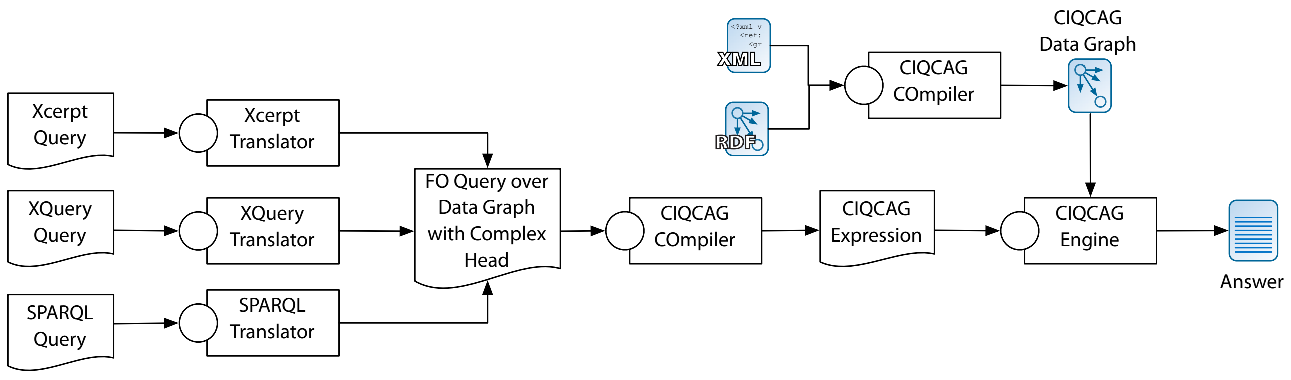

Figure 9. CIQCAG Architecture

CIQCAG is **(1)** *designed around the sequence map* data structure and, if used in conjunction with a sequence map, achieves the same or better time and space complexity for CIG data graphs as the best known approaches for tree data. **(2)** *immediately familiar* to anyone with knowledge on the standard relational algebra. In fact, the semantics of all CIQCAG operators is purely relational (can be expressed in terms of standard relational algebra[5]). Using the sequence map and its associated algorithms provides an *equivalent yet more efficient* implementation of the CIQCAG operators. **(3)** *able to process tree and graph queries* efficiently by means of a three-stage evaluation: in the *map construction* stage a tree core of the query is computed and specific operators allow the efficient evaluation of that tree core. In the *map expansion* stage the remaining non-tree relations are considered. Finally, there is a set of *result construction* operators that allow the construction of new data graphs (where the previous two stages operate exclusively on the input data). **(4)** for most of the presentation, limited to composition-free queries (in the sense of [39]), but in Section 3.9 we discuss the addition of iteration and recursion on top of the three stage processing described here. **(5)** able to be combined with *most approaches for indexing Web* data such as reachability indices [28, 52] or path indices [22, 12] as well as other "computed" or *derived relations* such as computed closure relations, complex binary path expressions [5], etc. This allows a great flexibility in choosing how a relation is actually realized: extensional, using some index or labeling scheme, computed ad-hoc, etc. **(6)** *accompanied by a rich set of algebraic laws*, many familiar from relational algebra, but with some additions to handle new operators and the specificity of the multi-stage evaluation.

Based on these principles, however, there are still a number of trade-offs when realizing the CIQCAG algebra, cf. Section 3.1 for details. Most are related to how the "links" between the decomposed relations in a sequence map are realized. In the purely relational variant we use table with order numbers and range queries. Though theoretically only slightly more costly than the other variants ($\mathcal{O}(\log n)$ instead of $\mathcal{O}(1)$ for deciding whether two matches for parent-child query nodes are related), practical performance of range queries is often disappointing in current relational databases (cf. [29]). In [10], we proposed a representation of the links by nesting (and sharing) of matches for child nodes within parent nodes

[5]With value invention, grouping, and ordering, if construction of new graphs is also considered.

(similar to nested relational data but with sharing to avoid duplication). Though this yields better lookup time, we pay with quadratic space in worst-case even for tree and CIG data. The third variant uses ordered relations (like SQL tables) and random access to rows in these ordered relations. This allows for constant lookup time and (in case of tree or CIG data) linear space, but comes at the cost of slightly more involved reorganization, cf. Section 3.

The above principles also make CIQCAG eminently suitable for implementing practical Web query languages. Indeed, in Part [20], we show how to translate (large subsets of) XQuery, Xcerpt, and SPARQL into CIQLog, a high-level calculus for CIQCAG, which is then translated into CIQCAG expressions as illustrated in Figure 9. CIQCAG is flexible enough to evaluate programs in any of these languages and to actually mix them within the same query. It is worth noting, that both the translation from (composition-free) XQuery to CIQLog and the equivalence of calculus and algebra are accompanied by formal proofs. CIQCAG is, thus, one of the few *formally* correct algebras for (composition-free) XQuery.

1.4.1 Operator Overview

The main operators of the CIQCAG algebra are immediately familiar from relational algebra. However, CIQCAG operator's differ in a few noteworthy points (the full details as well as the operators' implementation on top of the sequence map are explained in Chapter 3): **(1)** There are *three sets of operators*, as shown in Table 9, one for each stage of the query evaluation. Mostly identical to their relational counterpart are the operators for *map expansion* (renaming ρ, join $\bowtie$, union $\cup$, difference $-$, projection π, and duplicate elimination δ). However, we add an operator for accessing specific columns from a sequence map computed in the previous stage as flat relation ($\digamma$). **(2)** For *map construction*, we also introduce a new access operator μ that creates a sequence map from a given relation R. The other operators closely mirror relational operators and thus the operators of the map expansion phase, but there is one more addition: the propagation operator. **(3)** When we restrict the values for a given column in a classical, flat relation (e.g., by selection or join), we can omit tuples with values violating the selection or join criterion in a single pass over the table. However, the sequence maps store the (so far) computed answers like a column store. Though this allows more efficient storage in face of multivalue dependencies as in tree queries, eliminating values in a single column no longer immediately affects other columns. Rather when a value is eliminated in one column we have to *propagate* this fact to all related columns (and from there on recursively). CIQCAG provides for this propagation an explicit operator. This allows us to perform several value restrictions on the same column before propagation all eliminated values at once, rather than propagation after each restriction. The price is that a sequence map may, after a restriction and before the corresponding propagation, be in an inconsistent state, where values in one column link to no longer valid values of another one. However, this price is clearly offset by the gain in complexity and performance: Implicit propagation at each restriction raises the worst-case complexity of query evaluation by a *multiplicative* factor q (query size). Using explicit separate propagation, it suffices to propagate each column's values to its parent columns once at the end of the map construction leading to an *additive* factor of $q \times n$. For more details on explicit propagation see Section 3. **(4)** Finally, result construction is similar to other complex value algebras (cf. [29, 39] with operators for value invention ν, ordering (ω), conditional construction, and graph construction.

To illustrate, how these operators play together to implement a typical Web query consider the query in Figure 10 ($\boxed{4}$ indicates a answer node, all other nodes are existentially quantified; $\neq_{id}$ indicates a anti-

access	join (conjunction)	union	difference
$\dddot{\mu}_{v,v'}(D,Q)$, $\dddot{\mu}_{v}(D,Q,R)$	$\dddot{\bowtie}_{\cap}^{(\not\sharp)}, \dddot{\bowtie}^{(\not\sharp)}, \dddot{\ltimes}^{(\not\sharp)}(S,S')$	$\dddot{\cup}(S,S')$	$\dddot{\setminus}(S,S')$
projection, rename	**selection**	**propagation**	**expansion**
$\dddot{\pi}_{V}(S), \dddot{\rho}_{v_1\to v_2}(S)$	$\dddot{\sigma}_{c}^{(\not\sharp)}(S)$	$\dddot{\omega}_{v}^{\uparrow}(S)$, $\dddot{\omega}^{v}_{\downarrow}(S)$	$\digamma_{V}(S)$

Table 9. Overview of sequence map operators in CIQCAG (all operators return a single sequence map S except $\digamma$ which returns a (standard) relation)

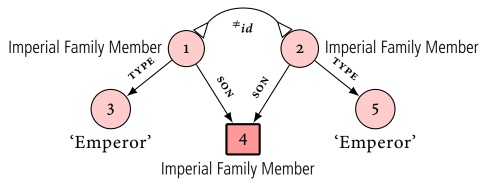

Figure 10. Selecting all sons of Roman emperors that have a double claim to the throne (two (distinct) fathers that where both emperors).

join based on node identity). This is a graph query, but a spanning tree covers almost the entire query. The query is realized in CIQCAG by the following expression (we decide to treat the SON edge from 2 to 4 as only non-tree edge[6]):

$$
\begin{aligned}
\digamma_{v_2,v_4}\Big(\; &(\dddot{\mu}_{v_1,v_4}(D,Q)\; \dddot{\bowtie}_{\cap}\; \dddot{\pi}_{v_1}(\dddot{\mu}_{v_1,v_3}(D,Q)\; \dddot{\bowtie}_{\cap}\; \dddot{\mu}_{v_3}(D,Q,\text{'Emperor'})))\\
&\dddot{\bowtie}_{\cap}\; \dddot{\mu}_{v_1,v_2}(D,Q)\; \dddot{\bowtie}_{\cap}\; \dddot{\pi}_{v_2}(\dddot{\mu}_{v_2,v_5}(D,Q)\; \dddot{\bowtie}_{\cap}\; \dddot{\mu}_{v_5}(D,Q,\text{'Emperor'}))\;\Big)
\end{aligned}
$$

Within the flatten operator, we find the expression responsible for evaluating the tree part of the query: we join sequence maps for type and son edges as well as label restrictions for nodes 3 and 5. Since we are not interested in the actual matches for 3 and 5 we use semi-joins instead of full joins. In the non-tree part we grab v_2 and v_4 from the sequence map (v_1 is not needed any more) and join them with the SON relation between v_2 and v_4. The result is projected to v_4 as this is the only answer node.

1.4.2 Tree Cores and Hypertrees

As seen in the previous example, we have considerable leeway how to choose the tree core of a graph query. In addition to usual considerations about selectivity and size of the involved relations, we have in our case an additional guide: if at all possible the chosen relations should carry the CIG property, i.e., exhibit an order on the data nodes such that the image of a node forms a continuous interval.

[6]The tree core of a query is not unique at all: We could also have chosen to consider also the $\neq_{id}$ as non-tree or to duplicate node 4 and introduce a identity join between the original and the duplicated node in the map expansion phase.

For relational queries, it has been shown that acyclic or tree queries are not the largest class of polynomial time queries. Rather queries with bounded tree- or hypertree-width [25] can still be evaluated in polynomial time. Nevertheless, we choose a tree decomposition. Hypertree decompositions yield groups of query nodes (variables) that are strongly connected in a query, but loosely connected with the rest of the query. If there is a bound on the size of these groups, we can evaluate each group separately and only expose the few connection points to the rest of the query. In contrast, for tree queries the groups are always single query nodes. For our purpose, however, hypertree decompositions seem less attractive than for standard relational queries for two reasons: **(1)** to treat multiple incoming edges (even if there number is bound by some constant) the sequence map data structure needs to be extended considerably and **(2)** the CIG property also needs to be adapted to guarantee *one* order among all relations involved in a hypertree node. In addition, the integration of techniques such as path indices and twig joins that have been developed for tree queries with CIQCAG is almost free if we consider tree cores, but is less than obvious if we consider hypertree decompositions.

However, for future work a closer inspection of hypertree decompositions (and, in particular, their effect on the CIG property) would be strongly desirable.

1.5 Complexity and Contributions

With a first intuition on the data structures and operations that form the CIQCAG algebra, we can turn to investigate some of its properties. For a detailed analysis of the space complexity of the sequence map see Section 2.4, the complexity of query evaluation with CIQCAG is analysed in Chapter 3.

Table 11 summarizes the complexity results for query evaluation with the CIQCAG algebra (using the sequence map as data structure). It is worth highlighting data and query space complexity are both linear for tree and CIG data and tree queries. Also note, that there is no penalty at all to going from tree data to CIG data, but arbitrary graph data does incur a linear penalty (both in space and time). The logarithmic factor in the time complexity can be discarded, if we assume that all relations can be accessed in CIG order. This is possible, since the CIG property is a static property of the data and does not depend on the query. Otherwise, we need to sort relations to add them to the sequence map. For Table 11, we assume the latter.

One factor in the time complexity of CIQCAG is the cost of the membership test in a relation. For extensional relations this is constant. However, as argued above, CIQCAG is also well suited to handle derived relations such as descendant with out without index. In this cases m may in fact have significant influence on the query evaluation. For the evaluation of the Web query languages discussed in [20] (Xcerpt, XQuery, SPARQL), extensional relations and structural closure relations (XPath's descendant, following, etc.) suffice. As discussed in [20], for tree data, membership in closure relations can be tested in constant or almost constant time (e.g., using interval encodings [16] or other labeling schemes such as [54]). However, for graph data this is not so obvious. Fortunately, there has been considerable research on reachability or closure relations and their indexing in arbitrary graph data in recent years. Table 13 summarizes the most relevant approaches for our work. Theoretically, we can obtain constant time for the membership test if we store the full transitive closure matrix. However, for large graphs this is clearly infeasible. Therefore, two classes of approaches have been developed that allow with significantly lower space to obtain sub-linear time for membership test.

The first class are based on the idea of a 2-hop cover [13]: Instead of storing a full transitive closure, we

data \ queries	path queries	tree queries	graph queries
tree	**T** $= \mathcal{O}(q \cdot n)$	**T** $= \mathcal{O}(q \cdot n)$	$\mathcal{O}(n^{q_g} + T_{\text{tree}})$
	S $= \mathcal{O}(q \cdot n)$	**S** $= \mathcal{O}(q \cdot n)$	$\mathcal{O}(n^{q_g} + S_{\text{tree}})$
CIG	**T** $= \mathcal{O}(q \cdot n)$	**T** $= \mathcal{O}(q \cdot n)$	$\mathcal{O}(n^{q_g} + T_{\text{tree}})$
	S $= \mathcal{O}(q \cdot n)$	**S** $= \mathcal{O}(q \cdot n)$	$\mathcal{O}(n^{q_g} + S_{\text{tree}})$
graph	**T** $= \mathcal{O}(q \cdot n^2 \cdot m)$	**T** $= \mathcal{O}(q \cdot n^2 \cdot m)$	$\mathcal{O}(n^{q_g} + T_{\text{tree}})$
	S $= \mathcal{O}(q \cdot n^2)$	**S** $= \mathcal{O}(q \cdot n^2)$	$\mathcal{O}(n^{q_g} + S_{\text{tree}})$

Table 11. Complexity of query evaluation with CIQCAG algebra (q query size, n data size, m complexity of membership test—assumed constant for all tree, forest, or CIG shaped relations, q_g: number of "graph" variables, i.e., variables with multiple incoming query edges)

allow that reachable nodes are reached via at most one other node (i.e., in two "hops"). More precisely, each node n is labeled with two connection sets, $in(n)$ and $out(n)$. $in(n)$ contains a set of nodes that can reach n, $out(n)$ a set of nodes that are reachable from n. Both sets are assigned in such a way, that a node m is reachable from n iff $out(n) \cup in(m) \neq \emptyset$. Unfortunately, computing the optimal 2-hop cover is NP-hard and even improved approximation algorithms [50] have still rather high complexity.

A different approach [1, 11, 53, 52] is to use interval encoding for labeling a tree core and treating the remaining non-tree edges separately. This allows for sublinear or even constant membership test, though constant membership test incurs lower but still considerable indexing cost, e.g., in Dual Labeling [53] where a full transitive closure over the non-tree edges is build. GRIPP [52] and SSPI [11] use a different trade-off by attaching additional interval labels to non-tree edges. This leads to linear index size and time at the cost of increased query time.

For CIQCAG we can choose any of the approaches. For the following, we assume constant time membership, since that is easily achieved on trees and feasible with approaches such as Dual Labeling even for graphs.

With this understanding about the complexity of query evaluation with CIQCAG, we can move to compare CIQCAG with representative approaches for Web querying developed in recent years (mostly for XML data). Table 15 summarizes this comparison: It shows that CIQCAG rivals the best known approaches for tree data (twig joins and SPEX), but in contrast to those approaches extends the same complexity to CIG data and can also, albeit at a cost, handle graph queries (like structural joins used in most other XML or RDF algebras). Note, that in all cases we consider pointers of constant size (as done in [8], [44], and [40]). In fact, all approaches need an additional $\log n$ multiplicative factor if pointer size is taken into consideration.

To summarize, CIQCAG is a novel algebra for Web queries that

– is based on a novel data structure for *efficient storage of intermediary results* of tree queries that is exponentially more succinct than purely relational approaches.

– extends previous approaches for querying tree data to a larger class of data graphs, called *continous-image graphs*. The CIG property can be tested in polynomial time and is independent of the query.

– *rivals the best known approaches* for tree query evaluation on tree data yet extends their properties (complexity and skipping, cf. Section 3.9.1) to CIG data.

– *gracefully degrades* with the increasing "graphness" of data and queries .

approach	characteristics	query time	index time	index size
Shortest path [48]	no index	$\mathcal{O}(n+e)$	–	–
Transitive closure	full reachability matrix	$\mathcal{O}(1)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$
2-Hop [13]	2-hop cover[a]	$\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n \cdot \sqrt{e})$
HOPI [50]	2-hop cover, improved approximation algorithm	$\mathcal{O}(\sqrt{e}) \leq \mathcal{O}(n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n \cdot \sqrt{e})$
Graph labeling [1]	interval-based tree labeling and propagation of intervals of non-tree descendants.	$\mathcal{O}(n)$[b]	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$[c]
SSPI [11]	interval-based tree labeling and recursive traversal of non-tree edges	$\mathcal{O}(e-n)$	$\mathcal{O}(n+e)$	$\mathcal{O}(n+e)$
Dual labeling [53]	interval-based tree labeling and transitive closure over non-tree edges	$\mathcal{O}(1)$[d]	$\mathcal{O}(n+e+e_g^3)$	$\mathcal{O}(n+e_g^2)$
GRIPP [52]	interval-based tree labeling plus additional interval labels for edges with incoming non-tree edges	$\mathcal{O}(e-n)$	$\mathcal{O}(n+e)$	$\mathcal{O}(n+e)$

[a] Index time for approximation algorithm in [13].

[b] More precisely, the number of intervals per node. E.g., in a bipartite graph this can be up to n, but in most (sparse) graphs this is likely considerably lower than n.

[c] More precisely, the total number of interval labels.

[d] [53] introduces also a variant of dual labeling with $\mathcal{O}(\log e_g)$ query time using a, in practical cases, considerably smaller index. However, worst case index size remains unchanged.

Table 13. Cost of Membership Test for Closure Relations. n, e: number of nodes, edges in the data, e_g: number of non-tree edges, i.e., if $T(D)$ is a spanning tree for D with edges $E_{T(D)}$, then $e_g = |E_D \setminus E_{T(D)}|$.

	query type	**time**	**space**
Structural Joins decompose query; test each structural constrain individually; join the results;	path queries	*path index:* (DataGuide [22], IndexFabric [14], [12])[a]	
		$\mathcal{O}(m_{path}) \sim \mathcal{O}(d)$	$\mathcal{O}(n)$; $\mathcal{O}(d \cdot n)$ index
		no path index, *standard join* [2, 28, 23])	
		$\mathcal{O}(n^{q_a} + q \cdot n \cdot \log n \cdot m)$[b]	
			$\mathcal{O}(n^{q_a} + q \cdot n^2)$
		no path index, *structure-aware join*, ([6]; tree data only)	
		$\mathcal{O}(n^{q_a} + q \cdot n)$	$\mathcal{O}(n^{q_a} + q \cdot n^2)$
	tree queries	with path index (tree/DAG data only)	
		$\mathcal{O}(n^{q_a} + b \cdot n \cdot \log n \cdot m_{path}))$	$\mathcal{O}(n^{q_a} + b \cdot n^2)$
		no path index	
		$\mathcal{O}(n^{q_a} + q \cdot n \cdot \log n \cdot m)$	$\mathcal{O}(n^{q_a} + q \cdot n^2)$
	graph queries	$\mathcal{O}(n^{q})$	$\mathcal{O}(n^{q})$
Twig or Stack Joins holistic (single operator) [8]; partial answers in q stacks; parent pointers connect stack entries;	tree data	$\mathcal{O}(q \cdot n)$[c]	$\mathcal{O}(q \cdot n + n \cdot d)$[d]
limited to only child/descendant relations in trees and DAGs [11]; "longer" skip distance when using indices [Jiang];	graph data	$\mathcal{O}(q \cdot n^2 + e)$[e]	$\mathcal{O}(q \cdot n + e)$[f]
SPEX (and similar streaming engines) [45, 44]; similar to twig join but with a single input stream and pointers realized as conditions; additionally supports horizontal axes; skipping added in [9];		$\mathcal{O}(q \cdot n \cdot d)$[g]	$\mathcal{O}(q \cdot n)$[h]
Complete Answer Aggregates [40]; manages *all* answers as CIQCAG; limited to tree data and structural relations (e.g., no value joins); similar to twig join without stack management		$\mathcal{O}(q \cdot n \cdot \log n \cdot d)$	$\mathcal{O}(q \cdot n \cdot d)$
	w/o closure axes	$\mathcal{O}(q \cdot n \cdot \log n)$	$\mathcal{O}(q \cdot n)$
CIQCAG sequence map variant; skipping may reduce average case	tree queries	$\mathcal{O}(q \cdot n)$[i]	$\mathcal{O}(q \cdot n)$[i]
		$\mathcal{O}(q \cdot n^2 \cdot m)$	$\mathcal{O}(q \cdot n^2)$
	graph queries	$\mathcal{O}(n^{q_g} + q \cdot n)$[i]	$\mathcal{O}(n^{q_g} + q \cdot n)$[i]
		$\mathcal{O}(n^{q_g} + q \cdot n^2 \cdot m)$	$\mathcal{O}(n^{q_g} + q \cdot n^2)$

[a] limited to unary path queries with only child/descendant relations against trees

[b] skipping reduces average case by using indices such as [28], XR-Tree [38], BIRD [54]

[c] More precisely, $\mathcal{O}(q \cdot \max(b_i, d))$ where b_i is the average of bindings per query variable. Both b_i and d are, in worst case, n. $b_i \ll n$ only if the selectivity of the node tests in the query is high.

[d] Answers are progressively generated.

[e] More precisely, $\mathcal{O}(q \cdot \max(b_i, d)^2)$ where b_i is the average of bindings per query variable. Both b_i and d are, in worst case, n.

[f] Answers are progressively generated.

[g] Lower complexity for limited fragments, e.g., $\mathcal{O}(q \cdot n \cdot d)$ if only horizontal axes are present.

[h] In some fringe cases, complexity degenerates to $\mathcal{O}(q \cdot n^2)$, for details see [44]. Answers are progressively generated.

[i] For tree and continuous-image graphs.

Table 15. Comparison of Related Approaches. n: number of nodes in the data, d: depth, resp. diameter of data; e: number of edges; q: size of query, q_a: number of result or answer variables; q_g: number of "graph" variables, i.e., variables with multiple incoming query edges; m maximum time complexity for relation membership test; m_{path} time complexity for path index access.

Chapter 2

Sequence Map

2.1 Introduction

As discussed in the previous section, the sequence map data structure stands at the core of the CIQCAG algebra: It allows us the polynomial, in many cases even linear, storage of (intermediary) answers to a tree core of a query. For tree queries all evaluation is done directly on the sequence map, for graph queries we must evaluate the remaining non-tree relations separately, but still profit in most cases greatly by reducing the size of the non-tree answers.

Let us briefly recall, from the previous section, the main motivations for designing a new data structure:

(1) When we evaluate tree queries, we can observe that for determining matches for a given query node only the match for its parent and child in the query tree are relevant.

(2) Indeed, if we consider the answer relation for a tree query, i.e., the relation with the complete bindings as rows and the query's nodes as columns, this relation always exhibits multivalued dependencies [18].

(3) To avoid these dependencies, we fully decompose the answer relation as in a column store: binding sequences for each query node with "links" or pointers relating bindings of different nodes. This gives us an exponentially more succinct storage than a flat relation.

(4) Once we have partitioned the answer relation into what subsumes to only link tables as in column stores, we can observe even more regularities (and thus possibilities for compaction) if the underlying data is a tree or continuous-image graph. These regularities allow us to represent the image of a node under as a single, continuous interval and thus yield an even smaller representation of the intermediary answers.

In the remainder of this section, we illustrate how these principles and observations are exploited in the definition of the sequence map data structure to obtain a space optimal data structure for tree queries on tree, forest, and CIG data (linear space data complexity). The same data structure can also be used on arbitrary graph data where it is more compact, for most cases, than a decomposed relation without interval pointers, though its worst-case space complexity is the same. We start this illustration with the formal definition of the sequence map in Section 2.2, continue it by taking a closer look at the effect of data shape on the interval representation of related answers in sequence maps (cf. Section 2.3, and conclude with a study of the space complexity of the sequence map in Section 2.4. We also briefly glance at some variations of the basic sequence map that allow us to cover a slightly larger fragment of queries albeit at a slight increase in the time complexity of most operations on the sequence map (which are discussed in the next chapter).

2.2 Sequence Map: A Data-Structure for the Decomposed Representation of Intermediary Answers to Tree Queries

To hold intermediary results of tree cores of queries, we define a compact data structure, called SEQUENCE MAP. As sketched above, this data structure holds (intermediary) results of n-ary tree queries while avoiding the multivalued dependencies that occur if flat relations are employed for this purpose. By avoiding these dependencies and storing the results fully decomposed, we obtain an exponentially more succinct data structure than a flat relation.

Additionally, we exploit the properties of the queried data: Where the data permits (as in the case of trees and CIGs), we compact the pointers (sometimes also called "links", references, or foreign keys) between the decomposed representations for bindings of different variables into intervals. For this, we store bindings of each variable as a single sequence with interval points to related child variables associated to each binding.

Formally, we define a sequence map over the queried data and the (tree) query to be evaluated. As data, we consider a (slightly extended)[1] *relational structure* D. D is defined over a relational schema $\Sigma = (R_1[U_1], \dots R_k[U_k])$ and a *finite* domain N of nodes (or objects or elements or records) in the data. Each $R_i[U_i]$ is a relation schema consisting in a relation name and a nonempty set of attribute names, We assume an equality relation $=$ on the nodes that relates each node to itself only (*identity*). D is a tuple $(R_1^D, \dots, R_k^D, O)$. Each R_i^D is a finite, unary or binary relation over N with name R_i. For a relation R, $\mathsf{ar}(R)$ denotes its arity. We extend D with an order mapping O that associates with each (binary) R_i a total order on N such that all $n \in \operatorname{rng} R_i$ are before all $n' \in N \setminus \operatorname{rng} R_i$. We denote with $O(D) = \{o : \exists R_i \in D : O(R_i) = o\}$ the set of orders to which the relations in D are mapped. These orders serve to represent the image of each node in a relation as one or more continuous intervals over the order associated with that relation. Choosing an appropriate order for a relation is discussed in Sections 2.3.1 (for tree data) and 2.3.2 (for CIG data).

Note, that we do not assume that all relations in D are extensional. Rather some might be derived (e.g., as closure) from other relations. See [20] for a discussion on relations to represent XML and RDF documents and how to use these relations to translate XQuery and Xcerpt queries into CIQCAG expressions.

For the example data in Figure 2, appropriate relations are the son-relation between members of the imperial family, the ruled-relation between emperors and (newly constituted) provinces. For both relation the node IDs in Figure 2 give a suitable associated order that allows the representation of the relation with a single interval pointer per parent node as described in Section 2.3.1. We use unary relations to classify

[1]The deviation lies in the addition of order for each relation. Furthermore, we restrict ourselves to binary relations.

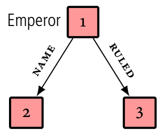

Figure 11. Selecting all Roman emperors together with their name and ruled provinces.

nodes by type. In this case, we use type relations for imperial family members, emperors, co-emperors, and provinces each. Furthermore, we explicitly represent names of provinces and family members attached by a name relation.

A sequence map is used to store intermediate results for one specific tree query. A tree query, in our context, is any tree over a set of variables (or query nodes or attributes of the result relation). Each node is a query variable, each edge is labeled with a relation that must hold between bindings for the query variables connected by the edge. Formally, a query Q, over a relational schema Σ using the relation names $\mathcal{R}_1$ for unary relations and the relation names $\mathcal{R}_2$ for binary relations, is a tree[2] $(V, \mathcal{L}_V, E = V \times V, r, \mathcal{L}_E)$. The set of variables V serve as nodes of the tree with the root r and are labeled with (sets of) relation names by $\mathcal{L}_V : V \to 2^{\mathcal{R}_1}$. Edges in E connect pairs of variables and are labeled with the labeling function $\mathcal{L} : E \to \mathcal{R}_2$ connects each edge in Q with a relation name. For each variable (or node) v in Q, we denote **(1)** with $\mathsf{children}(v) = \{v' \in V : (v, v') \in E\}$ the child variables of v; **(2)** if n is not the root, with $\mathsf{parent}(v) = v'$, if $(v', v) \in E$, the parent variable of v **(3)** if n is not the root, with $\mathsf{rel}(v) = \mathcal{L}((\mathsf{parent}(n), n))$ the name of the relation between the parent variable and v.

In the following, we assume that all relations used in a query Q are also defined in the relational structure D the query is evaluated against.

Figure 11 illustrates an example query tree with the nodes $\{1, 2, 3\}$, the edges $\{(1, 2), (1, 3)\}$, the root 1, the edge labeling $\{(1, 2) \to \text{RULED}, (1, 3) \to \text{NAME}\}$, and the node labeling $\{1 \to \mathsf{Emperor}\}$.

Bindings for variables are stored in the sequence map fully decomposed, one sequence per variable.

Definition 2.1 (Sequence). A SEQUENCE S from some (finite) set N is a finite sequence on N, or more formally a bijective function from $\{1, \ldots, |N|\}$ to N. For a sequence S, the i^{th} element of the sequence is $S(i)$ and is sometimes denoted as $S[i]$ to emphasize that S is a sequence. i is referred to as *index* of $S[i]$ in S.

Note, that we allow only *duplicate free* sequences in this definition, i.e., a $n \in N$ occurs at most once in a sequence S. Therefore, we can denote with $S^{-1}(n)$ the index of n in S.

For a finite set N, we define the *set of all subsequences*, denoted $\mathit{SubSeq}(N)$, as the set $\{S : \{1, \ldots, k\} \to N' : N' \subset N \wedge k = |N'| \wedge S \text{ bijective}\}$.

We call a sequence $S \in \mathit{SubSeq}(N)$ *consistent* with an order $<_N$ over N if for all $n_1, n_2 \in N$ it holds that, if there are i, j with $S[i] = n_1$ and $S[j] = n_2$, then $n_1 <_N n_2$ implies $i < j$. If $<_N$ is total, $n_1 <_N n_2$ iff $i < j$.

Finally, any total order on N is naturally represented by a sequence over N and vice versa:

[2] In fact, sequence maps can be used for forest an diamond-free DAG queries, see Section 2.5.2. For clarity of presentation, we limit ourselves here to tree queries.

Definition 2.2 (Induced sequence)**.** Let $<$ be a *total* order over some domain N. Then $<$ induces a sequence $S_<$ over N with $S_<[i] = n \in N$ such that $|\{n' \in N : n' < n\}| = i$. $S_<$ is, by definition, *consistent* with $<$.

Obviously, computing the induced sequence is the same as sorting N w.r.t. $<$ and thus has $\Omega(n \log n)$ time complexity.

Definition 2.3 (Sequence Map)**.** A SEQUENCE MAP $\overset{D\to Q}{\mathsf{SM}}$ on an (extended) relational structure D and a (tree) query Q over D is a mapping from the set of variables $\mathsf{Vars}(Q)$ to sequences of bindings for these variables to nodes in D. For each variable binding, we also record a set of intervals of related bindings for each child variable. This way, a binding b for a variable v (at index i) is associated with a set of triples (v', s, e) that indicates that all bindings b' with index $s \leq j \leq e$ for the child variable $v' \in \mathsf{children}(v)$ are related to b.

Let *Intervals* $= \{(i, j) \in \mathbb{N}^2 : i \leq j\}$ be the set of all intervals of integers. Then, we obtain the following signature for a sequence map:

$$\overset{D\to Q}{\mathsf{SM}} : \mathsf{Vars}(Q) \to \mathit{SubSeq}(\mathsf{Nodes}(D) \to 2^{\mathsf{Vars}(Q) \times \mathit{Intervals}})$$

Note, that in each sub-sequence each $n \in \mathsf{Nodes}(D)$ occurs at most once and is associated to a single set of pairs of variables and intervals.

For any $v \in V$, we write $\overset{D\to Q}{\mathsf{SM}}(v)$ to indicate the sequence of bindings for v. With $\mathsf{binding}(\overset{D\to Q}{\mathsf{SM}}(v)[i])$ we denote the actual binding node in the i-th entry for v (or some distinct value $\perp$ with $\perp > n$ for all $n \in \mathsf{Nodes}(D)$ if $i > |\overset{D\to Q}{\mathsf{SM}}(v)|$), with $\mathsf{intervals}_{v'}(\overset{D\to Q}{\mathsf{SM}}(v)[i])$ the set of intervals associated with v' in the i-the entry for v in the sequence map (or $\varnothing$ if $i > |\overset{D\to Q}{\mathsf{SM}}(v)|$). Finally, we write $\mathsf{Nodes}_{v'}(\overset{D\to Q}{\mathsf{SM}}(v)[i])$ to indicate the set of bindings for v' covered by an interval in $\mathsf{intervals}_{v'}(\overset{D\to Q}{\mathsf{SM}}(v)[i])$.

In addition to the above signature, we place three further restrictions on any sequence map $\overset{D\to Q}{\mathsf{SM}}$:

(1) For each variable v and index i, the set of intervals $\mathsf{intervals}_{v'}(\overset{D\to Q}{\mathsf{SM}}(v)[i])$ is *non-overlapping*. A set M of intervals is non-overlapping if, for each pair of intervals $(s_1, e_1), (s_2, e_2) \in M$, it holds that $s_1 \leq s_2$ iff $e_1 \leq s_1$.

(2) For each variable v, child variable v' of v, and index i, all intervals in $\mathsf{intervals}_{v'}(\overset{D\to Q}{\mathsf{SM}}(v)[i])$ are *grounded* in $\overset{D\to Q}{\mathsf{SM}}$. An interval $(s, e) \in \mathsf{intervals}_{v'}(\overset{D\to Q}{\mathsf{SM}}(v)[i])$ is grounded in $\overset{D\to Q}{\mathsf{SM}}$ if $s \leq e \leq |\overset{D\to Q}{\mathsf{SM}}(v')|$.

(3) For each $v \neq \mathsf{root}(Q)$, the sequence $\overset{D\to Q}{\mathsf{SM}}(v)$ is consistent with the order associated with $\mathsf{rel}(v)^D$ in D.

(4) Finally, we record for each sequence map, the set of edges in Q covered by that sequence map. This set of edges is denoted by $\mathsf{edgeCover}(\overset{D\to Q}{\mathsf{SM}}) \subset (\operatorname{dom} S)^2 \cap \mathsf{Edges}(Q) \subset \mathsf{Vars}(Q)^2$. A sequence map may only contain references between bindings for variable pairs contained in its edge cover: For each pair of variables $v, v' \in \mathsf{Vars}(Q)$ with binding indices $i \leq |\overset{D\to Q}{\mathsf{SM}}(v)|, i' \leq |\overset{D\to Q}{\mathsf{SM}}(v')|$ such that $\overset{D\to Q}{\mathsf{SM}}(v')[i'] \in \mathsf{Nodes}_{v'}(\overset{D\to Q}{\mathsf{SM}}(v)[i])$, it must hold that $(v, v') \in \mathsf{edgeCover}(\overset{D\to Q}{\mathsf{SM}})$.

Together the two restrictions guarantee that, with each binding of a variable v, there are at most associated $|N|$ intervals over the bindings for $v' \in \mathsf{children}(v)$ in a sequence map: The *set* of associated intervals does not contain duplicate intervals (since it is a set), each interval covers at least one index, no

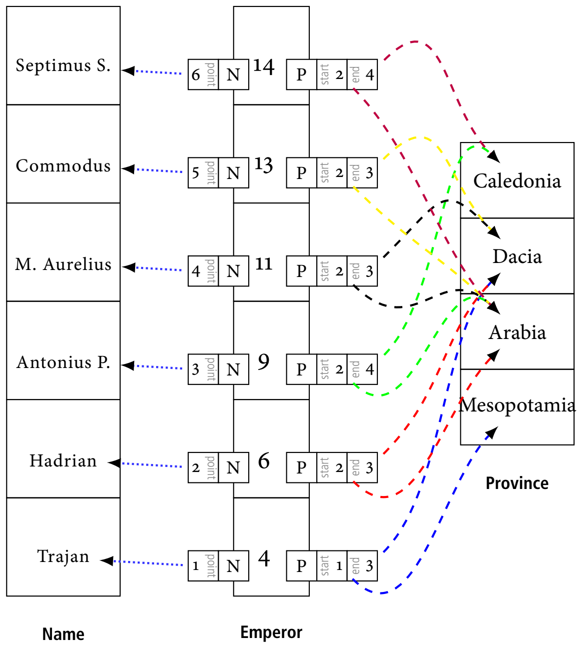
Septimus S.
Commodus
M. Aurelius
Antonius P.
Hadrian
Trajan
Name
6 point N 14 P start 2 end 4
5 point N 13 P start 2 end 3
4 point N 11 P start 2 end 3
3 point N 9 P start 2 end 4
2 point N 6 P start 2 end 3
1 point N 4 P start 1 end 3
Emperor
Caledonia
Dacia
Arabia
Mesopotamia
Province

Figure 12. Sequence Map: Example. For the query from Figure 11 on the data of Figure 2.

two intervals overlap, and all intervals are grounded. Thus we can, at most, have $|\overset{D\to Q}{\mathsf{SM}}(v')| \leq |N|$ intervals each covering one of the bindings of v' in the sequence map.

To illustrate the notion of sequence map consider again the query in Figure 11. The sequence map representing all answers to that query against the data in Figure 2 is illustrated in Figure 12: From bindings for "Emperors" (i.e., members of the unary type relation Emperor) we reference related bindings for name and province (where we use the name of the province instead of the ID for ease of presentation). The relations are expressed as intervals associated with the abbreviated query variable (N for name, P for province). Observe, that since the data is CIG shaped and the bindings are ordered accordingly we need always only single intervals. We abbreviate single element intervals as standard pointers.

It is worth emphasizing that we allow *multiple* intervals to represent the related bindings for a child variable. This is necessary to represent answers to queries on arbitrary graphs. As discussed in Sections 2.3.1 and 2.3.2, we can guarantee a single continuous interval for more restrictive shapes of data, viz. trees, forest, and CIGs. Even for arbitrary graphs the use of interval pointers is beneficial in many cases, cf. Section 2.3.3.2.

A sequence map serves as a compact representation of an answer relation. This relation can be retrieved from a sequence map as follows:

Definition 2.4 (Induced relation). Let $\overset{D\to Q}{\mathsf{SM}}$ be a sequence map with associated edge cover EC. Let $U = \text{dom}\,\overset{D\to Q}{\mathsf{SM}} = \{v_1, \ldots, v_k\} = \{v \in \mathsf{Vars}(Q) : \exists n \in \mathsf{Nodes}(D) : n \in \overset{D\to Q}{\mathsf{SM}}(v)\}$. Then $\overset{D\to Q}{\mathsf{SM}}$ induces a relation $\overset{D\to Q}{R_{\mathsf{SM}}}$

such that

$$\overset{D\to Q}{R}_{\mathsf{SM}} = \big\{(b_1,\dots,b_k) \in \mathsf{Nodes}(D)^k : \exists k \in \mathbb{N} : \overset{D\to Q}{\mathsf{SM}}(v_i)[k] = b_i \wedge \\ ((v_i, v_j) \in \mathrm{EC} \implies b_j \in \mathsf{Nodes}_{v_j}(\overset{D\to Q}{\mathsf{SM}}(v_i)[k]))\big\}$$

Note, that this implies that for pairs of variables that are not in the edge cover of the sequence map *all* combinations of bindings are included in the induced relation.

In Section 3.7, we introduce the sequence map extraction that allows to extract bindings of one or more variables from a sequence map in form of a relation. If bindings for all variables in S are extracted, this operation yields exactly the induced relation.

The empty relation is induced by the empty sequence map, denoted by $\overset{D\to Q}{\mathsf{SM}}_{\varnothing}$. It is also induced by a sequence map where some of the "links" imposed by the query are missing, e.g., if for some child variable all bindings of its parent variable include no interval pointers to bindings of the child variable. In this sense, such a sequence map and the empty sequence map can be considered equivalent:

Definition 2.5 (Equivalent sequence maps)**.** Let $\overset{D\to Q}{\mathsf{SM}}_1$ and $\overset{D\to Q}{\mathsf{SM}}_2$ be two sequence maps. We call $\overset{D\to Q}{\mathsf{SM}}_1$ and $\overset{D\to Q}{\mathsf{SM}}_2$ equivalent, if they induce the same relation. However, they may differ, e.g., in the chosen order of elements, the intervals used, etc.

Note, that a sequence map does not need to map all variables in $\mathsf{Vars}(Q)$. If some variables are not mapped, the resulting answer is incomplete w.r.t. any constraints in Q involving the missing variables. In other words, absent variables are ignored when considering the induced relation of a sequence map. In the following section, we define a class of sequence maps, called complete sequence maps, that represents an answer relation to a full query rather than to only a part of a query.

2.2.1 Consistent and Inconsistent Sequence Maps

Sequence maps store (intermediary) answers to tree queries decomposed. They are an exponentially more succinct store than flat relations. On flat relations, if we restrict bindings of one variable we implicitly affect bindings for all variables since each tuple in the answer relation represents one particular assignment of bindings to query variables. Assume, e.g., we drop all tuples where the binding of v has a value ≤ 10. A binding b' for some other variable v' may, however, only occur together with bindings for v that have value ≤ 10. Thus dropping the above tuples also drops all occurrences of b'.

On sequence maps, however, bindings are stored per variable (or column of the flat relation). Thus, when we modify bindings for one variable, bindings for other variables are not implicitly affected, rather these changes must be *explicitly propagated*. Details of this propagation are discussed later in Section 3. Intuitively, in the above case b' remains among the bindings of v' still pointing (assuming, for simplicity, that v' is the parent variable of v) to the now dropped bindings for v. However, b' could be dropped without loosing any proper answer (there is no way to extend b' bindings for v' to full answers). In a sense, such a sequence map is *inconsistent* as there are bindings for v' that refer to invalidated (or "bombed") bindings for v. We can address this in two ways: **(1)** *Immediate propagation:* All operations on a sequence map that restrict variable bindings for one variable (or column) ensure before the conclusion of the operation that those restrictions are propagated to all possibly affected variables. **(2)** *Separate propagation from (local) restriction:* Operations on a sequence map may restrict bindings of one or more variables without immediately propagating the effect to other connected variables. The advantage is that we can perform a series of restrictions on one or even a subset of the query variables and only at the end of that series propagate all changes at once. The disadvantage is that we have to mark temporary inconsistencies and must ensure that they are propagated at some point. However, we can simulate the first case by following each operation on the sequence map immediately by a propagation to all related variables.

In the following, we adopt the second approach since it is more flexible and requires, for many queries, significantly less propagation operation. To support this approach we introduce the concept of inconsistent sequence map, i.e., sequence maps where some restrictions on variable bindings have not yet been fully propagated.

To distinguish invalidated variable bindings, we mark each invalidated variable binding with a failure marker $\lightning$ from a (finite) set of failure markers $\mathcal{B}$[3] with $|\mathcal{B}| \leq |\mathsf{Nodes}(D)|$. Furthermore, we extend the signature of a sequence map to include failure markers:

$$\overset{D\to Q}{\mathsf{SM}}: \mathsf{Vars}(Q) \to \mathit{SubSeq}((\mathsf{Nodes}(D) \to 2^{\mathsf{Vars}(Q)\times \mathit{Intervals}}) \cup \mathcal{B})$$

We limit the number of failure markers by the number of nodes in D, since failed bindings only result from invalidating existing bindings and a sequence map can, for a single variable, contain at most $|\mathsf{Nodes}(D)|$ bindings. Note also, that we do not need to record related bindings (for child variables) if a binding is "bombed". Also, by definition of the induced relation of a sequence map, failure markers do not affect the induced relation as all bindings in a induced relation must be nodes from D.

Failure markers address the invalidation of entire bindings for a variable (e.g., due to additional unary constraints). Another form of local restriction, however, is the removal of references to bindings of a child variable in the bindings of a parent variable. E.g., the province of Mesopotamia in the sequence map of Figure 12 is related only to emperor Trajan. If we remove the reference from Trajan to Mesopotamia, i.e., change the P interval of the emperor with ID 4 to 2 – 3 instead of 1 – 3. (or "bomb" Trajan), Mesopotamia is not any more part of any full answer to the query and could thus be dropped. So, again, the sequence map retains information that, if we propagate immediately, could be dropped. We call bindings such as Mesopotamia *dangling bindings*. A dangling binding n is *direct*, if there is no binding for the immediate parent variable with an interval pointer covering n. Otherwise it is *indirect*.

Definition 2.6 (Consistent and inconsistent sequence maps). A sequence map $\overset{D\to Q}{\mathsf{SM}}$ is called *inconsistent*, if **(1)** it contains any "failed" bindings $\lightning \in \mathcal{B}$, or **(2)** it contains any "dangling" binding, i.e., a binding $b \in \overset{D\to Q}{\mathsf{SM}}(v)$ for some variable v such that $v' = \mathsf{parent}(v)$, $v' \in \mathrm{dom}\,\overset{D\to Q}{\mathsf{SM}}$, and there is no $i \in \mathbb{N}$ such that $b \in \mathsf{Nodes}_v(\overset{D\to Q}{\mathsf{SM}}(v')[i])$. Otherwise it is called *consistent*.

The notion of equivalent sequence maps immediately extends to inconsistent sequence maps (note that the bindings in the induced relation are from $\mathsf{Nodes}(Q)$ excluding $\lightning$). In Section 3.5.3 we introduce an algorithm for efficiently propagating changes in a sequence map. Using this algorithm, we obtain the following result:

Theorem 2.1. *Let $\overset{D\to Q}{\mathsf{SM}}$ be an inconsistent sequence map. Then there is a consistent sequence map $\overset{D\to Q}{\mathsf{SM}}'$ equivalent to $\overset{D\to Q}{\mathsf{SM}}$. This sequence map can be computed in $\mathcal{O}(\tilde{q}\cdot n \cdot i)$ where $\tilde{q} = |\mathrm{dom}\,\overset{D\to Q}{\mathsf{SM}}|$, $n = |\mathit{Nodes}(D)|$, and i the maximum number of intervals per binding in S. For tree, forest, and CIG data $i = 1$.*

[3]For consistency, we use a set of failure markers to be able to continue to consider a sequence as duplicate free.

Proof. See Section 3.5.3. □

Figure 13 illustrates a more complex sequence map than the one from Figure 12. It is inconsistent since there are some failure markers and binding d_{12} for variable v_3 is dangling. Furthermore, if the failure markers are propagated d_3 for v_1 is also dropped (there are no proper related bindings for v_2 which in consequence makes also d_{11} in v_3 dangling.

Even if we consider only consistent sequence maps, there are multiple sequence maps with the same induced relation: They contain the same bindings for each variable, but the interval pointers between bindings may vary as long as they cover the same set of bindings. E.g., in one sequence map a binding may contain $\{(v,1,3)\}$ to point to bindings of v, in another $\{(v,1,1),(v,2,2),(v,3,3)\}$, and in yet another $\{(v,1,2),(v,3,3)\}$. However, there is a unique *minimal* interval representation for the interval pointers of each binding (here the one of the first sequence map):

Definition 2.7 (Interval-minimal sequence map)**.** A sequence map $\overset{D\to Q}{\mathsf{SM}}$ is called *interval-minimal*, if the set of interval pointers for any binding in $\overset{D\to Q}{\mathsf{SM}}$ is minimal, i.e., there is no smaller set of interval pointers that covers the same bindings for each child variable.

Theorem 2.2. *A* interval-minimal, consistent *sequence map is uniquely identified by its induced relation, i.e., there is no other* interval-minimal, consistent *sequence map with the same induced relation.*

Proof. Let R be the induced relation of a given sequence map S. Then, **(1)** we can not add or remove a binding for any variable $v \in \operatorname{dom} S$: If we add a binding for a single variable, the binding is dangling and the resulting sequence map is not consistent. If we add a binding and reference if from some interval pointer of some binding for the parent variable, the induced relation is no longer the same but contains additional tuples. **(2)** we can not extend or shrink, collapse or divide an interval pointer for a binding n of $v \in \operatorname{dom} S$. If we extend an interval pointer i and the added indices (which are adjacent to i) are covered in the old interval set, then the original set of intervals is not minimal (as we could collapse i with the interval pointer covering the adjacent indices). If the added indices are not covered in the old interval set, we introduce new tuples into the induced relation. Analog for shrinking, we remove tuples in the induced relation. For collapsing and dividing, either the original interval set is not minimal (if we can collapse) or the resulting interval set is not minimal (if we divide). □

2.2.2 Answers: Consistent and Complete Sequence Maps

Sequence maps may be only a partial mapping of query variables to bindings and thus contain only partial or intermediary answers to a query, many of which may actually not contribute to any complete answer of the query.

Definition 2.8 (Complete sequence map)**.** For a tree query Q a sequence map $\overset{D\to Q}{\mathsf{SM}}$ is called *complete* if **(1)** all variables of Q are covered by the sequence map: $\operatorname{dom}\overset{D\to Q}{\mathsf{SM}} = \mathsf{Vars}(Q)$; **(2)** all relations of Q are covered in all bindings for the involved variables: for all $v, v' \in \mathsf{Vars}(Q)$ it holds that $v = \mathsf{parent}(v')$ if and only if, for all $i \le |\overset{D\to Q}{\mathsf{SM}}(v)|$, $\mathsf{intervals}_{v'}(\overset{D\to Q}{\mathsf{SM}}(v)[i]) \neq \varnothing$; **(3)** all relations in the sequence map are covered by the query: for all $v, v' \in \mathsf{Vars}(Q)$ it holds that if there is a $i \in \mathbb{N}$ such that $\mathsf{intervals}_{v'}(\overset{D\to Q}{\mathsf{SM}}(v)[i]) \neq \varnothing$, then $v = \mathsf{parent}(v')$.

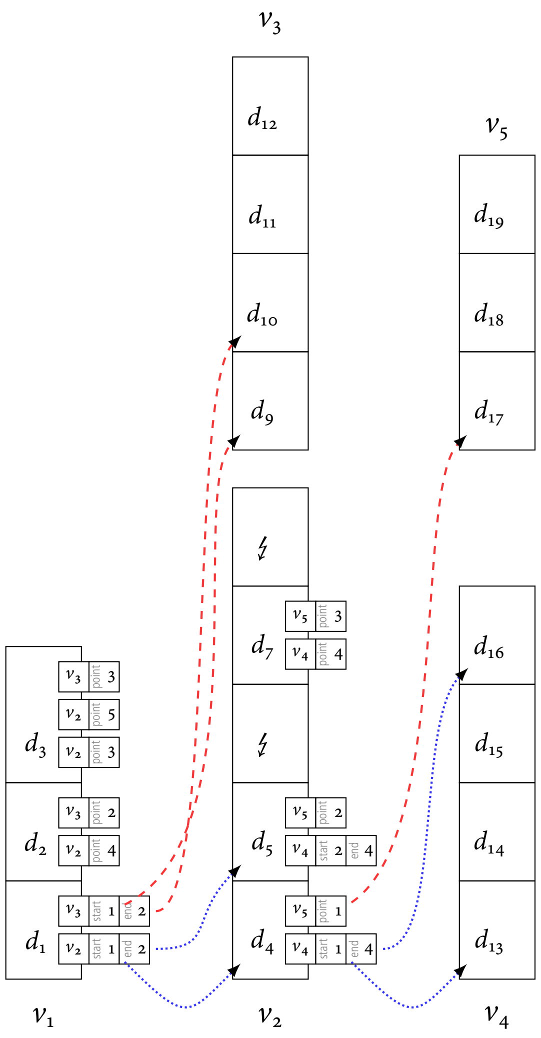

Figure 13. Inconsistent Sequence Map.

Complete sequence maps can still be inconsistent, though failure markers can only occur for leaf variables (for inner variables, they violate condition (2) of the definition as there are no related bindings to a failure marker by definition). "Dangling" bindings may occur for any variable.

Before we can compile queries into sequence map operations, we first need to define those operations in Chapter 3. Before we can define the operations, we establish a number of properties for sequence maps on different shapes of data in Section 2.3. With these properties, we can establish, in Section 2.4 bounds for the space needed to represent (intermediary) results of a tree query in a sequence map for the different kinds of data. These results are used, finally, to define operations on sequence maps such as creation of sequence maps, projection, join, union, subtraction, and propagation to remove inconsistencies, cf. Section 3. We conclude with a number of variants of the sequence map, in particular, a purely relational sequence map.

2.3 On The Influence of Data Shape

Sequence maps are capable of representing (intermediary) answers to tree queries on any relational structure as defined above. However, when we pose certain restrictions on the shape of the relations involved, we can place bounds on the number of intervals required to represent relations between bindings of adjacent query nodes and thus on the size of the sequence map.

2.3.1 Exploiting Tree-Shape of Data: Single Interval Pointers

Tree data exhibits a number of regularities that have been exploited by most previous approaches to querying structural relations on tree data: labeling schemes such as pre-/post-encoding [16, 28]allow constant time, constant space membership test of tree-shaped relations as well as their closure (descendant or ancestor in XPath); similarly, twig join approaches [8] exploit the fact that in tree shaped data, their is at most one parent for each node in the data and at most d related nodes, with d depth of the tree, if the closure relation is considered, i.e., at most d ancestor nodes; like twig joins, we exploit limits on the number of parents and ancestors of a node in SPEX. Additionally, we observe that, for horizontal closure relations, the bounds are less favorable, e.g., breadth of tree for following-sibling and size of tree for following. But also for horizontal relations we observe that the images for nodes follow certain patterns (e.g., the followings of a node are a subset of the followings of all its pre-order predecessors).

In the following, we give alternative characterisations of tree-shaped relations and their transitive closures that allow us to identify in what way we can order the domain to allow a linear representation of the related nodes in the given relation. Furthermore, we use these characterisations to show in Section 2.3.2 how to go beyond tree data and still maintain the linear representation.

To start with, let us consider relations that directly form trees, i.e., direct structural relations such as child, next-sibling, and next (in the sense of [23]). If we look at how the images of nodes are shaped under these relations, we can note that the images of two different nodes never overlap. We call this property *image disjointness*:

Definition 2.9 (Image Disjointness Property). Let R be a binary relation over some domain N. Then R is said to carry the IMAGE DISJOINTNESS property, if and only if, for any two nodes $n_1 \neq n_2 \in N$, it holds that $R(n_1) \cap R(n_2) = \varnothing$.[4]

We choose this formulation of image disjointness as it immediately induces a set of orders on N that guarantee that the image of a node $n \in N$ can be represented as a single interval: keep each $R(n)$ together (with arbitrary "internal" order) but choose an arbitrary order among the $R(n)$. More formally, Let $<_{arbitrary}$ be some arbitrary total order on N, $<_{\text{dom}}$ some total order on dom R, and $<_n$ a total order on each $R(n)$ for $n \in \text{dom}\, R$. Then $<$ is a total order on N with

$$<= \left\{(n,n') \in (\text{rng}\, R)^2 : n \in R(m) \wedge n' \in R(m') \wedge \begin{cases} n <_m n' & \text{if } m = m' \\ m <_{\text{dom}} m' & \text{if } m \neq m' \end{cases} \right\}$$

All $n \notin \text{rng}\, R$ either follow or precede the $n \in \text{rng}\, R$.

It is easy to see that a sequence over N consistent with any such order allows the representation of the images of any node n under R as a single interval.

In terms of orders on the tree, any breadth-first traversal induces such an order. Whether the traversal is top-down or bottom-up, left-to-right, or right-to-left is immaterial. In fact, the order in which the nodes of the tree are visited is entirely arbitrary as long as the children of each node are visited together.

The image disjointness property captures exactly all tree- and forest-shaped relations:

Theorem 2.3. *Let R be a binary relation over some domain N. Then (N,R) is a forest iff R carries the image disjointness property.*

Proof. Recall, that a (directed rooted) tree is a rooted connected simple graph with a unique simple path between the root and any other node. A forest is a disjoint union of trees, i.e., we all multiple roots but each node is part of exactly one tree. In other words, there is only one root per node from which that node is reachable and, as in a tree, there is a unique simple path between that root and the node.

If (N,R) is a forest and $n_1 \neq n_2 \in N$, then $R(n_1) \cap R(n_2) = \varnothing$: any node $n' \in R(n_1) \cap R(n_2)$ violates the forest property as it is either, if n_1's root is different from n_2, reachable from multiple roots (i.e., (N,R) is no disjoint union of trees) or, if n_1 and n_2 have the same root, there are two unique simple paths between the root and n' by appending n' to the path from the root to n_1 and to n_2.

If R has the image disjointness property, then let $roots(R) = \{n \in N : \nexists\, n' \in N : (n',n) \in R\}$. **(1)** Each node $n \in N$ is reachable from at most one root $r \in roots(R)$: If n is directly reachable (a child) from r then it can not be directly reachable from any other $n' \in N$ including any $r \in roots(R)$ due to the image interval property and as r is not reachable from any node in n by definition of $roots(R)$. If n is indirectly reachable from r the same argument can be made recursively for the nodes in the path from r to n. **(2)** For each node $n \in N$, there is a unique path from its root r to n. If there are two distinct paths from r to n then either n or some node on the path from r to n lies in the image of two distinct nodes in violation of the image interval property. □

However, on tree data also closure relations can be represented and tested in linear time and space, e.g., the descendants of a node. In fact, for any of the structural *closure* relations such as descendant, ancestor, following, following-sibling, etc. we can observe the same. However, the images of two distinct nodes are clearly not distinct in these cases. E.g., the descendants of a child are always a subset of the descendants

[4] Recall, the we denote with $R(n) = \{n' \in N : (n,n') \in R'\}$.

of its parent. For all these relations, the images of two different nodes can overlap, but on tree data only in a "disciplined" manner: either they do not overlap at all or one is entirely contained in the other. We formalize this property as follows:

Definition 2.10 (Image Containment Property). Let R be a binary relation over some domain N. Then R is said to carry the IMAGE CONTAINMENT property, if and only if, for any two nodes $n_1 \neq n_2 \in N$, it holds that $R(n_1) \cap R(n_2) \neq \varnothing$ implies that $R(n_1) \subset R(n_2) \vee R(n_2) \subset R(n_1)$.

Again, given this property we can readily define an order over N such that the images of each node can be represented as a single, continuous interval: the containment of images constitute a hierarchy on the domain of R. As long as that hierarchy is respected by the order, the images of a node can be represented as a single, continuous interval on a sequence over N consistent with that order.

More formally, let $<_n$ be some order on $R(n)$ for each $n \in \operatorname{dom} R$. Let $direct(n)$ for some $n \in \operatorname{rng} R$ be the $n' \in \operatorname{dom} R$ such that $n \in R(n')$ and there is no $n'' \in \operatorname{dom} R$ such that $n \in R(n'')$ and $R(n'') \subset R(n')$. Finally, let $<_{\text{dom}}$ be an arbitrary order on $\operatorname{dom} R$ and $<_{incl} = \{(n, n') \in (\operatorname{dom} R)^2 : R(n) \subset R(n') \vee (R(n) \cap R(n') = \varnothing \wedge n <_{\text{dom}} n')\}$. Then $\prec$ is a total order on N with

$$\prec \; = \{(n, n') \in (\operatorname{rng} R)^2 : m = direct(n) \wedge m' = direct(n') \wedge \begin{cases} n <_m n' & \text{if } m = m' \\ m <_{incl} m' & \text{if } m \neq m' \end{cases}$$

All $n \notin \operatorname{rng} R$ either follow or precede the $n \in \operatorname{rng} R$.

The image containment property captures exactly closure relations over forest-shaped base relations:

Theorem 2.4. *Let R be a binary relation over N. Then R carries the image containment property if and only if there is a forest-shaped base relation R' such that R is the transitive closure of R'.*

Proof. If R is the transitive closure of a forest-shaped relation R' and $m \in R(n), m \in R(n')$. Then either n is an ancestor of n' or the other way round as R' is a forest, i.e., $R(n) \subset R(n')$ or vice versa.

If R carries the image containment property, let $R' = \{(n, n') \in N^2 : R(n') \neq \varnothing \wedge R(n') \subset R(n) \wedge \nexists\, m \in N : R(n') \subset R(m) \wedge R(m) \subset R(n)\} \cup \{(n, n') \in N^2 : n' \in R(n) \wedge \nexists\, m \in N : n' \in R(m) \wedge R(m) \subset R(n)\}$. The first set represents the hierarchy of the inner nodes, the second set the leaf nodes. Then (N, R') is a forest, since **(1)** leaf nodes l have by definition a single parent: if there are two $n \neq n' \in N$ with $l \in R(n)$ and $l \in R(n')$ but neither $R(n) \subset R(n')$ nor $R(n') \subset R(n)$ (otherwise n or n' do not fulfill the condition for parents of lead nodes), then $R(n) \cap R(n') \neq \varnothing$ but neither is subset of the other in violation of the image containment property of R. **(2)** inner nodes i have also a single parent: if there are two nodes n, n' with $i \subset R(n)$ and $i \subset R(n')$ but no $m, m' \in N$ with $i \subset R(m) \subset R(n)$ and $i \subset R(m') \subset R(n')$, then $R(n) \cap R(n') \neq \varnothing$, yet neither is a subset of the other. Thus, the image containment property is violated. □

2.3.2 Beyond Trees: Consecutive Ones Property

Tree data, as argued above, allows us to represent relations on that data more compact, e.g., using various interval-based labeling schemes. Here, we introduce a new class of graphs, called *continuous-image graphs* (or CIGs for short), that generalize features of tree data in such a way that we can evaluate (tree) queries on CIGs with the same time and space complexity as techniques such as twig joins [8] which are limited

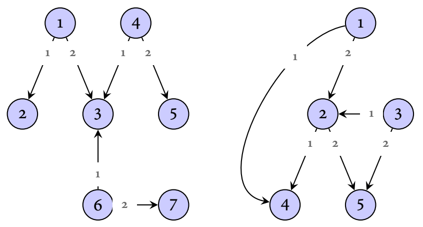

Figure 14. Sharing: On the Limits of Continuous-image Graphs

to tree data only. Moreover, we show that even skipping and pruning techniques used for tree data carry over to continuous-image graphs (cf. Section 3.9.1).

Continuous-image graphs are a proper superset of (ordered) trees where we require not that each node has at most one parent, as we do for trees, but that we can find a single order on all nodes of the graph such that the children of each parent form a continuous interval in that order. This definition allows graphs where some or all children of two parents are "shared" but limits the degree of sharing: Figure 14 shows two minimal graphs that are *not* CIGs. Incidentally, both graphs are acyclic and, if we take away any one edge in either graph, the resulting graph becomes a CIG. The second graph is actually the smallest (w.r.t. number of nodes and edges) graph that is not a CIG. The first is only edge minimal but illustrates an easy to grasp sufficient but not necessary condition for violating the interval property: if a node has at least three parents and each of the parents has at least one (other) child not shared by the others then the graph can not be a CIG.

On continuous-image graphs we can exploit similar techniques for compacting structural relations as on trees, most notably representing the nodes related to a given node as a single, continuous interval and thus with constant space. This applies also for derived relations such as closure (XPath's descendant) or order relations (XPath's following-sibling) on CIGs.

From a practical perspective, CIGs are actually quite common, in particular, where time-related or hierarchical data is involved: If relations, e.g., between Germany and kings, are time-related, it is quite likely that there will be some overlapping, e.g., for periods where two persons were king of Germany at the same time. Similarly, hierarchical data often has some limited anomalies that make a modelling as strict tree data impossible. Figure 2 shows actual data[5] on relations between the family (red nodes, non-ruling member 1, co-emperor or heir designate 10, emperors 2) of the Roman emperors in the time of the "Five Good Emperors" (Edward Gibbon) in the 2nd century. It also shows, for actual emperors, which of the four new provinces (I) added to the roman empire in this period each emperor ruled (the other provinces remained mostly unchanged and are therefore omitted). Arrows between family members indicate, natural or adoptive, fathership[6]. Arrows between emperors and provinces show rulership, different colors are used to distinguish different emperors. Despite the rather complicated shape of the relations (they are obviously not tree-shaped and there is considerable overlapping, in particular w.r.t. province rulership).

[5] The name and status of the province between the wall of Hadrian and the wall of Antonius Pius in northern Britain is controversial. For simplicity, we refer to it as "Caledonia", though that actually denotes all land north of Hadrian's wall.

[6] Note that all emperors of the Nervan-Antonian dynasty except Nerva and Commodus were adopted by their predecessor and are therefore often referred to as "Adoptive Emperors".

Formally, we define a CIG as an arbitrary relation (or graph) that carries the following property:

Definition 2.11 (Image interval property). Let R be a binary relation over some domain N. Then R is said to carry the IMAGE INTERVAL property, if and only if there exists a total order $<_i$ on N with its induced sequence S over N such that for all nodes $n \in N$, $R(n) = \varnothing$ or $R(n) = \{S[s], \ldots, S[e] : s \leq e \in \mathbb{N}\}$.

This property merely formalizes the original observation that one means of compacting a binary relation on N (for linear space storage with linear time membership test) is through the use of intervals. Here, we demand that there is an order and thus a sequence over N that allows us such an interval representation.

The image interval property is a generalization of both image disjointness and image containment but characterizes a strictly larger class of relations.

Theorem 2.5. *Let R be a binary relation over some domain N. If R carries the image disjointness (image containment) property, then R also carries the image interval property.*

Proof. For relation with image disjointness or image containment property, we choose an order over N as described when defining the two properties. Together with any such order, R fulfils the definition of image interval property. □

Theorem 2.6. *The image interval property covers a strictly larger class of relations than either the image disjointness or the image containment property.*

Proof. Figure 2 as well as Figure 15 show relations that carry the image interval property but do not carry either image disjointness or image containment. In general, CIGs allow more freedom in overlapping than relations with image containment property: two nodes may share some but not all nodes in their image. However, as discussed above, cf. Figure 14, they still pose a limit on the sharing. □

It is easy to see, that the weakest of the three properties, image disjointness, can be tested in quadratic time over the size of the domain. Surprisingly, the same holds for the image interval property: For that we observe that it is merely another formulation of to the consecutive-ones property introduced for $\{0, 1\}$ matrices in [19]. A $\{0, 1\}$ matrix is said to exhibit the *consecutive-ones* property if its rows may be permuted in such a way as to make the ones in each column consecutive.

Theorem 2.7. *Let R be a binary relation over some domain N. Then R carries the image interval property iff its adjacency matrix carries the* consecutive-ones *property.*

Proof. Recall, that the adjacency matrix of a binary relation $R \subset N^2$ is a quadratic $\{0, 1\}$ matrix M where rows and columns correspond to the nodes in N and $M(i, j) = 1$ iff the nodes corresponding to the ith row and jth column stand in relation R. Let M_R be a corresponding matrix for a relation R. M_R exhibits the consecutive-ones property if and only if the image interval property holds for R: Each column represents the images of a node. A permutation of the rows is merely a specific order of the nodes in N. Thus, if a permutation of the rows exists such that the ones in each column are consecutive, then an order on the nodes in N exists such that the images of each node form a single continuous interval on the sequence of nodes represented by the row permutation. □

For the consecutive-ones problem [7] gives the first linear time (in the size of the matrix) algorithm based on so called PQ-trees, a compact representation for permutations of rows (or columns) in a matrix. Consider the relation represented in Figure 15 as an adjacency matrix. It is neither forest-shaped (e.g., 1 has many parents: $(A, 1)$, $(A, 2)$, $\dots$) nor does it carry the image containment property (2 is in image of both A and B but neither image is a subset of the other).

However, we can compute a PQ-tree for this relation that represents all permutations of column orders such that the 1s are consecutive in each row. This PQ-tree is shown in Figure 16. A PQ-tree contains, as the name suggests, two kinds of inner nodes: Q nodes and P nodes. P nodes indicate that any permutation of its children guarantees consecutive 1s in each row. Q nodes indicate that its children must be traversed in order or in the inverse order. For Figure 16 this means, that the PQ-tree represents the permutations 4271356, 4271536, 4217356, 4217536, 6357124, 6351724, 6537124, 6531724.

Figure (b) shows one such permutation where, indeed, the 1s are consecutive in all rows. It is also easy to see that we can flip 7 and 1, as well as 3 and 5 arbitrarily (they are each identical). And, of course, we can invert the order of the columns without violating the consecutive ones property.

The PQ-tree algorithm gives us a quadratic decision algorithm whether any given relation carries the image interval property (details cf. [7]).

Theorem 2.8. *Let R be a binary relation over some domain N. Then deciding whether R carries the image interval property and computing a corresponding order $<_i$ has space and time complexity $\mathcal{O}(|N|^2)$*

More recent refinements of the PQ-tree, viz. the PC-tree described in [34, 36], give a slightly simpler test for the consecutive-ones property (albeit with the same complexity). Figure 17 shows the PC-tree for the relation in Figure 15. In a PC-tree, we have again two kinds of inner nodes: P nodes where we can freely permute the children and C (or cyclic) nodes where we can only traverse the children either in clockwise or in counter-clockwise order.

2.3.3 Open Questions: Beyond Single Intervals

With the image interval property, we now have a characterisation of a large class of graphs that contains all forest-shaped relations and their closures, yet covers a substantially larger class of graphs. This characterisation allows us, as for forest-shaped relations, to use a single interval to represent the related nodes of any node in a relation carrying the property. Moreover, deciding whether a relation carries that property is decidable in quadratic time and gives, as a by-product, an appropriate order. Note, that the decision is entirely independent of the query and can thus be computed when storing the relation rather than an query evaluation time.

2.3.3.1 Bounded Intervals: k-Interval Property

The image interval property ensures that we need at most one interval to represent the image of a node under a relation. Linear time representation and linear time membership test, however, can also be achieved if we relax that a bit more: If there is a (preferably small) $k \in N$ independent of $|N|$ that represents an upper limit to the number of intervals needed to represent the image of a node, that still gives us a linear space representation and linear time membership test.

Thus, even for a relation R that carries the k-interval property for a small $k \in N$ we can still profit from the sequence map representation over a plain relational representation:

	1	2	3	4	5	6	7
A	1	1	0	1	0	0	1
B	1	1	1	0	1	0	1
C	1	0	1	0	1	1	1

(a) Adjacency matrix representation of a sample relation

	4	2	7	1	3	5	6
A	1	1	1	1	0	0	0
B	0	1	1	1	1	1	0
C	0	0	1	1	1	1	1

(b) Consecutive-ones permutation of relation from (a)

Figure 15. Relation with (1-) interval property

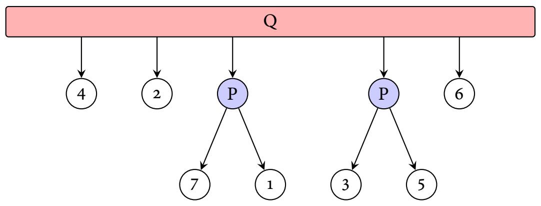

Figure 16. PQ-Tree for Relation in Figure 15

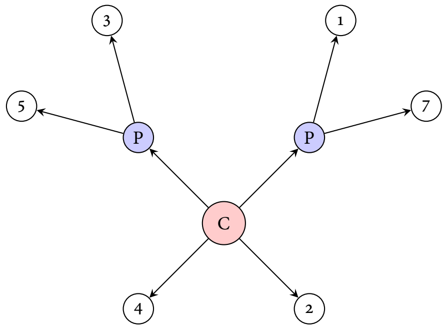

Figure 17. PC-Tree for Relation in Figure 15

Definition 2.12 (Image k-interval property)**.** Let R be a binary relation over some domain N and $k \in \mathbb{N}$. Then R is said to carry the k-IMAGE INTERVAL property, if and only if there exists a total order $<_i$ on N with its induced sequence S over N such that for all nodes $n \in N$, $R(n) = \emptyset$ or

$$R(n) = \{S[s_1], \ldots, S[e_1], S[s_2], \ldots, S[e_2], \ldots, S[s_l], \ldots S[e_l] : s_i \leq e_i \wedge l \leq k\}.$$

If we modify the relation as in Figure 18 by adding 4 to the image of A and 6 to the image of C, the resulting relation does no longer carry the consecutive-ones property (and thus has no associated PQ-tree). It carries, however, the 2-interval property as illustrated by the permutation in Figure 18. Furthermore, it also has a corresponding PC-tree (in fact, the same as the original relation). This is due to the fact that PC-trees actually test for the *circular-ones* property *circular-ones* property, as defined in [36]. The circular ones property of a $\{0,1\}$-matrix indicates that there is a permutation of the rows such that either the 0s or the 1s are consecutive in each column. It is called "circular ones" as, the 1s are still consecutive in all columns if we consider the matrix circular, i.e., after the last row we continue with the first row.

Whether there is a polynomial decision algorithm for the general k-interval property, remains an open question though the test for "circular ones" points towards a decision algorithm for $k = 2$. In the following, we use only the 1-interval property characterising CIG data.

2.3.3.2 Arbitrary, but Optimal Intervals for Arbitrary Graphs

Even on arbitrary graphs, where we can no longer guarantee liner space representation and linear time membership test, we nevertheless can often profit from selecting a suitable order on the nodes of N: for most graph the number of intervals needed to represent the images of a node is often much lower than the worst case of $\mathcal{O}(|N|)$.

To characterise such an order, we first define the (minimal) interval representation for a relation R:

Definition 2.13 (Minimal interval representation of a relation)**.** Let $R \subset N^2$ be a (finite) binary relation and $<$ a total order over N. Let $S_<$ be the induced sequence over N for $<$. Then $I : \operatorname{dom} R \to 2^{Intervals}\}$ is called the *minimal interval representation* of R over $<$ if $(s,e) \in I(n)$ for any $n \in domR$ iff for all $s < i < e$: $(n, S[i]) \in R$ (all elements in the interval are in $R(n)$), $s = 1$ or $(n, S[s-1]) \notin R$ and $e = |S_<|$ or $(n, S[e+1]) \notin R$ (there is no larger interval over $S_<$ that also fulfils the first condition and includes (s,e)).

Note, that the intervals in $I(n)$ are non-overlapping and cover $R(n)$ for each $n \in \operatorname{dom} R$. They are non-overlapping, as $(s_1,e_1),(s_2,e_2) \in I(n)$ with $s_1 < s_2 < e_1$ either $e_2 \leq e_1$ which means (s_2,e_2) violates the second condition of the definition or $e_2 > e_1$ which means both violate the second condition and (s_1,e_2) must be included in $I(n)$ by definition.

Using the notion of interval representation, we can now define the minimal interval representation and thus the cost of representing a relation by intervals over some given order $<$.

Definition 2.14 (Interval cost of a relation)**.** Let $I \in \mathcal{I}_<(R)$ be the minimal interval representation of a relation R under $<$. Then we call $\mathsf{IN}_<(R) = \sum_{n \in N} |I(n)|$ the size of I and the *interval cost* for R under $<$.

This gives us the cost of representing a relation R under an order on N. What we would like to find, is an order on N with the lowest cost for the representation:

Definition 2.15 (Interval-optimal order). Let $R \subset N^2$ be a (finite) binary relation and $\mathcal{O}$ the set of all total orders over N. Then an order $< \in \mathcal{O}$ is called *interval optimal* if its associated interval cost $\mathsf{IN}_<(R)$ is minimal among the interval costs of all orders over R.

Obviously, we can find the interval-optimal order by trying each permutation of the nodes in N. As for the k-interval property, it is an open question whether there is a *polynomial* decision algorithm for the optimal interval order.

For a given order <, however, we can compute a minimal interval representation in polynomial time by Algorithm 1.

Algorithm 1: Compute interval representation from relation

```
input  : Relation $R$ total order $<$ over rng $R$.
output: Interval representation $I$ of $R$.
1  $S \leftarrow$ induced sequence for $<$ over rng $R$;
2  $I \leftarrow \emptyset$;
3  foreach $n \in$ dom $R$ do
4      Intervals $\leftarrow \emptyset$;
5      start $\leftarrow \bot$;
6      for $i \leftarrow 1$ to $|S|$ do
7          if $(n, S[i]) \in R$ and start $= \bot$ then
8              start $\leftarrow i$ ;
9          if $(n, S[i]) \notin R$ and start $\neq \bot$ then
10             Intervals $\leftarrow$ Intervals $\cup \{(\text{start}, i-1)\}$ ;
11             start $\leftarrow \bot$ ;
12     if start $\neq \bot$ then
13         Intervals $\leftarrow$ Intervals $\cup \{(\text{start}, |S|)\}$ ;
14     $I \leftarrow I \cup \{(n, \text{Intervals})\}$;
15 return $I$
```

Theorem 2.9. *Algorithm 1 computes the minimal interval representation of the input relation R under the order < in time $\mathcal{O}(n^2)$ where $n = |Nodes(R)|$.*

Proof. Let I be the result of Algorithm 1 for a given R under the order $<$ and $S_<$ the induced sequence over rng R for $<$. Then I is the minimal interval representation of R under $<$ as (1) there is an interval $(s, e) \in I(n)$ for each $n' \in R(n)$ such that $s \leq S^{-1}(n') \leq e$ (as at each $n' \in R(n)$ either a new interval is started, l. 7–8, or an open interval is continued); (2) for all intervals $(s, e) \in I(n)$ and $s < i < e$, $S_<[i] \in R(n)$ (as if $S_<[i] \notin R(n)$ the previous interval is closed at $i - 1$, l. 9–11, or start $= \bot$); (3) for all intervals $(s, e) \in I(n)$ either $s = 1$ or $S_<[s-1] \notin R(n)$ (if $S_<[s-1] \in R(n)$ then s can not be the start of an interval as the interval either starts at $s - 1$, l. 7–9, or before); (4) for all intervals $(s, e) \in I(n)$ either $e = |S_<|$ or $S_<[e+1] \notin R(n)$ (only if $S_<[e+1] \notin R(n)$ or we are at the end of the sequence is an interval added with the current interval position as end index).

Algorithm 1 runs in $|\mathrm{dom}\, R| \cdot |\mathrm{rng}\, R| = \mathcal{O}(|\mathsf{Nodes}(R)|)$: the induced sequence can be computed in $\mathcal{O}(n \log n)$. The main loop (l. 3–14) iterates over all elements in dom R, the inner loop over the elements of S which are all elements in rng R. □

2.4 Space Bounds for Sequence Maps

With the above properties of data represented in sequence maps, we can now give precise characterisations of the space used by a sequence map $\overset{D \to Q}{\mathsf{SM}}$ over a relational structure D and a query Q.

Theorem 2.10. *Let $\overset{D \to Q}{\mathsf{SM}}$ be a sequence map over a relational structure D and a query Q. Then the size of $\overset{D \to Q}{\mathsf{SM}}$ is bounded by $\mathcal{O}(q \cdot n \cdot i) \leq \mathcal{O}(q \cdot n^2)$ where $n = |\mathsf{Nodes}(D)|$, $q = |\mathsf{Vars}(Q)|$, and i is the maximum number of intervals needed to represent the image of a node $n \in N$ under any query relation R and the order $O(R)$ associated with R in D.*

Proof. Recall, the signature of a sequence map from Section 2.2:

$$\overset{D \to Q}{\mathsf{SM}} : \mathsf{Vars}(Q) \to \mathit{SubSeq}(\mathsf{Nodes}(D) \to 2^{\mathsf{Vars}(Q) \times \mathit{Intervals}})$$

Each of the q variables in $\mathsf{Vars}(Q)$ is mapped to a sub-sequence over $\mathsf{Nodes}(D)$ each of which is associated with a subset of $\mathsf{Vars}(Q) \times \mathit{Intervals}$. Since sequences are by definition (see Section 2.2) duplicate-free, the size of each sub-sequence over $\mathsf{Nodes}(D)$ is bounded by $n = |\mathsf{Nodes}(D)|$. Thus, we have at most $q \cdot n$ bindings represented in a sequence map. However, each binding is associated with a sub-set over $\mathsf{Vars}(Q) \times \mathit{Intervals}$ the size of which is bounded by $q \cdot i$ where i is the maximum number of intervals needed to represent the image of a node $n \in N$ under any query relation R and its associated order $O(R)$.

This indicates a bound of $q \cdot n \cdot q \cdot i$. However, for each variable v, its bindings are referenced only from bindings of its parent variable v', but not from the bindings of any other variable. Thus, for each variable, we have at most $n \cdot i$ intervals referencing bindings of that variable and the total size of the associated interval sets is limited by $q \cdot n \cdot i$.

In total, we arrive at a bound of $q \cdot n$ for the bindings and a bound of $q \cdot n \cdot i$ for the interval sets and thus at an overall bound of $q \cdot n + q \cdot n \cdot i = \mathcal{O}(q \cdot n \cdot i)$.

Note, that $i \leq n$, as all intervals are, by definition of a sequence map, non-overlapping and grounded: Since they are grounded, the largest index covered by any interval is bounded by n (the maximum length of the sequence of child variable bindings). Since they are non-overlapping, there are at most n intervals to cover a sequence from 1 to n, viz. n intervals of size 1.

The edge cover associated with a sequence map does not affect the space complexity: since Q is a tree query, there are at most $q - 1$ edges in Q and thus in the edge cover of a sequence map for Q. □

Note, that the above result holds for arbitrary relations and tree queries. It shows that the sequence map provides a polynomial storage for (intermediary) answers of tree queries on arbitrary relations, i.e., an exponentially more succinct storage than flat relations.

Together with the results from the previous sections we can immediately infer a number of conditions when the sequence map provides linear data complexity for answer storage:

Corollary 2.1. *Let $\overset{D \to Q}{\mathsf{SM}}$ be a sequence map over a relational structure D and a query Q. Let all query relations carry the image disjointness, image containment, or image interval property together with their associated order $O(R)$ in D. Then the size of $\overset{D \to Q}{\mathsf{SM}}$ is bounded by $\mathcal{O}(q \cdot n)$ where $n = |\mathsf{Nodes}(D)|$ and $q = |\mathsf{Vars}(Q)|$.*

Proof. For relations that fulfill one of these properties, $i = 1$ since the $R(n)$ can be represented as a single interval for each $n \in \mathsf{Nodes}(D)$ over the induced sequence for $O(R)$. □

Since image disjointness, containment and interval are precise characterisations of tree/forest-shaped relations, closure relations over tree/forest-shaped relations, or CIG-shaped relations resp., we can also state this result as follows:

Corollary 2.2. *Let $\overset{D\rightarrow Q}{\mathbf{SM}}$ be a sequence map over a relational structure D and a query Q. Let all query relations be tree- or forest-shaped, closure relations over tree- or forest-shaped relations, or* CIG*-shaped. Then the size of $\overset{D\rightarrow Q}{\mathbf{SM}}$ is bounded by $\mathcal{O}(q \cdot n)$ where $n = |\mathsf{Nodes}(D)|$ and $q = |\mathsf{Vars}(Q)|$.*

Note, if all variables in Q are answer variables (in other words, if we evaluate Q by pattern matching in the classification of [51]), this bound becomes tight if we ensure that only nodes from N are retained in the sequence map that are actual matches for the full query, cf. Section 3.9 for details and conditions on the data needed to ensure that property.

Analogously, for relations with k-interval property we have at most k intervals needed to represent the image of each node an thus $\mathcal{O}(q \cdot n \cdot k)$ space bound.

Corollary 2.3. *Let $\overset{D\rightarrow Q}{\mathbf{SM}}$ be a sequence map over a relational structure D and a query Q. Let all query relations carry the k-image interval property together with their associated order $O(R)$ in D. Then the size of $\overset{D\rightarrow Q}{\mathbf{SM}}$ is bounded by $\mathcal{O}(q \cdot n \cdot k)$ where $n = |\mathsf{Nodes}(D)|$ and $q = |\mathsf{Vars}(Q)|$.*

Finally, note that these results hold also for inconsistent sequence maps as the number of failure markers is bounded by $|N|$ as well as the number of "dangling" bindings (since the latter are nodes from N).

The space bounds for sequence maps established in this section form the foundation for the time complexity of the sequence map operations discussed in Section 3. Before, we turn to the sequence map operations and thus the CIQCAG algebra proper, we finish our discussion of the sequence map data structure by a brief outlook on variants of the sequence map: a purely relational variant of the sequence map and a variant of the sequence map supporting not only tree queries but also some graph queries, viz. diamond-free DAG queries.

2.5 Sequence Map Variations

2.5.1 Purely Relational Sequence Map

As defined above, the sequence map uses sequences to represent bindings and associates with each binding a set of variable-interval pairs. Both are features that are not provided by pure relational databases (though sequences and order, e.g., are provided in SQL and most practical DBS).

Definition 2.16 (Purely relational sequence map)**.** A purely relational sequence map, denoted $\overset{*\rightarrow D}{\mathbf{SM}}Q$, over a relational structure D and a query Q is a set of relations: For each variable v in Q, there is a relation $R_v \in \overset{*\rightarrow D}{\mathbf{SM}}Q$ with schema $\langle$*index* $\in \mathbb{N}$, *binding* $\in \mathsf{Nodes}(D)$, *child variable* $\in \mathsf{children}(v)$, *start index* $\in \mathbb{N}$, *end index* $\in \mathbb{N}\rangle$. Each tuple in R_v stores a node binding together with its index and one interval of related nodes for one child variable.

In contrast to the definition of a sequence map in Section 2.2, we duplicate the index and binding node information for each related child variable interval. To obtain the related bindings of some child variable v' for a given binding of the parent variable v, we evaluate a *range query* on $R_{V'}$ for all tuples with *index* between *start index* and *end index*.

Though general range queries require $\mathcal{O}(n \log n)$ time to iterate all nodes in a given interval, we can exploit in this case the fact that the *index* column can be stored as a sequence or ordered relation allowing indexed access and $\mathcal{O}(n)$ time iteration. In this sense, the sequence map as defined in Section 1.3 can be seen as a specific realisation of the purely relational sequence map with linear time iteration for related bindings of a given node.

2.5.2 Multi-Order Sequence Map for Diamond-Free DAG Queries

The above definitions of a sequence map allow only tree queries (or tree cores of arbitrary queries). However, we can in fact extend the sequence map approach also to forest queries and even certain classes of DAG queries. In fact, the above definitions are also amenable to forest queries. Beyond forest queries, we can still use the sequence map for *diamond-free DAG queries*, albeit with a slight modification. Diamond-free DAG queries are queries in the shape of a DAG where there are no two distinct paths between two nodes (and thus no diamond-shaped sub-graph). Diamond-free DAG queries are also used, e.g., in [45, 42] (there named single-join DAG queries).

If we consider diamond-free DAG queries, there may be nodes in the query that have multiple parents. However, as for tree queries only the parent and child nodes are relevant to decide whether a data item is a match for a query node. The multiple parents, however, may be connected using different relations, e.g., XPath's child, descendant, and following relations. In the above definitions, we only demand that for each relation the continuous-image property of the data holds. If we have DAG queries this might lead to different, possibly incompatible orders for the image relations (e.g., child needs a breadth-first order to obtain continuous-images vs. depth-first order for descendant). However, we do not need to "strengthen" the CIG property for diamond-free DAG queries. Rather, we use the purely relational variant of the sequence map but add in R_v one separate index column for each different incoming relation of v with incompatible orders and interval pointers are resolved as range queries over the appropriate order number.

The downside of this adaptation is that we can now no longer use the index for ordered storage of the relation as there are several index columns. Therefore, we have to fall back to general, $\mathcal{O}(n \log n)$ time range queries rather than indexed access. For most of the operations discussed in Section 3, this increases the time complexity with a logarithmic factor. Moreover, range queries are in most practical SQL database systems not very efficient, cf. [29], if compared with indexed access.

	1	2	3	4	5	6	7
A	1	1	0	1	0	1	1
B	1	1	1	0	1	0	1
C	1	0	1	1	1	1	1

(a) Adjacency matrix representation of a sample relation with 2-interval property

	4	2	7	1	3	5	6
A	1	1	1	1	0	0	1
B	0	1	1	1	1	1	0
C	1	0	1	1	1	1	1

(b) Permutation of relation from (a) illustrating the circular-ones property

Figure 18. Relation with 2-interval property

Chapter 3

Sequence Map Operators

3.1 Introduction and Overview

In this chapter, we concentrate on the first set of operators in the CIQCAG algebra, the operators for constructing and manipulating a sequence map. The remaining operators are as in the relational algebra. The sequence map operators, summarized in Table 17, roughly fall into three groups: initialization or access operators (Section 3.3) create a sequence map from a given relation, combination operators (Section 3.4) such as join, union, and difference combine two sequence maps, and reduction operators (Section 3.5) such as projection, selection, or propagation drop some of the bindings contained in a sequence map. The role of the "odd men out" is played by the extraction operator (Section 3.7) that returns parts of the induced relation of its input sequence map and plays the bridge between sequence maps, used in the evaluation of the tree core of a query, and relations, used in the evaluation of the remaining query, if there is any.

The sequence map operators closely mirror their relational counterparts, where such exists. In general, they are defined by reduction to the relational counterpart on the induced relation(s) of the input sequence map(s). In contrast to relations, however, sequence maps are, in general, not closed under union, difference, or even projection, see Section 3.4. This is addressed by placing certain restrictions on

access	join (conjunction)	union	difference
$\dddot{\mu}_{v,v'}(D,Q)$, $\dddot{\mu}_{v}(D,Q,R)$	$\dddot{\bowtie}_{\cap}^{(\lightning)}$, $\dddot{\bowtie}^{(\lightning)}$, $\dddot{\ltimes}^{(\lightning)}(S,S')$	$\dddot{\cup}(S,S')$	$\dddot{\setminus}(S,S')$
projection, rename	**selection**	**propagation**	**expansion**
$\dddot{\pi}_V(S)$, $\dddot{\rho}_{v_1 \to v_2}(S)$	$\dddot{\sigma}_c^{(\lightning)}(S)$	$\dddot{\omega}_v^{\uparrow}(S)$, $\dddot{\omega}^{v}_{\downarrow}(S)$	$\digamma_V(S)$

Table 17. Overview of sequence map operators in CIQCAG (all operators return a single sequence map S except $\digamma$ which returns a (standard) relation)

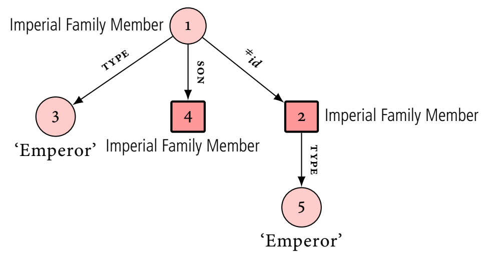

Figure 19. Spanning tree for the query from Figure 10

the input sequence maps for each of these operators.

The three most unusual operators are the sequence map join, propagation, and extraction. For the two latter operators, the reason is mostly that there are no relational counterparts. For the sequence map join the main reason is that we present a number of variants for the sequence map join with different characteristics. Most notably, we introduce (as we also do for selection and initialization) both consistent and inconsistent variants (marked with a $\lightning$ as superscript) of the join: The former ensures that a resulting sequence map is consistent if the input sequence maps are, the latter allows and in many cases introduces inconsistencies. However, as discussed in Section 3.4.1, the use of the inconsistent variant actually yields, in general, an evaluation plan with lower complexity as we propagate inconsistencies once per query variable instead of at each join.

To illustrate, how these operators play together to implement a typical Web query consider again the query in Figure 10 and the spanning tree for that query shown in Figure 19. Note, that we mark also v_2 as answer node (i.e., with a red rectangle 🟥). This is necessary as both v_2 and v_4 are used in the non-tree part of the query. We can evaluate this (spanning) tree query using sequence maps with many different sequence map expressions (see Section 3.8 for equivalences between CIQCAG expressions formed from sequence map operators).

One approach is to use the, in most cases more efficient, inconsistent variants of the operators where possible, at the price of additional propagation operations at the end of the query:

$$\begin{aligned}\mathcal{F}_{v_2,v_4}\Big(&\dddot{\varpi}^{v_4,v_2\downarrow}(\dddot{\varpi}_{v_4,v_2}^{\uparrow}(\\&(\dddot{\mu}^{\lightning}_{v_1,v_4}(D,Q) \dddot{\ltimes}^{\lightning}\ \dddot{\pi}_{v_1}(\dddot{\mu}^{\lightning}_{v_1,v_3}(D,Q) \dddot{\ltimes}^{\lightning}\ \dddot{\mu}_{v_3}(D,Q,\text{'Emperor'})))\\&\dddot{\bowtie}^{\lightning}\ \dddot{\mu}^{\lightning}_{v_1,v_2}(D,Q) \dddot{\ltimes}^{\lightning}\ \dddot{\pi}_{v_2}(\dddot{\mu}^{\lightning}_{v_2,v_5}(D,Q) \dddot{\ltimes}^{\lightning}\ \dddot{\mu}_{v_5}(D,Q,\text{'Emperor'}))\ \Big)\end{aligned}$$

Instead of the semi-join operators we could also choose (inconsistent) join operators. However, we could not replace the inconsistent joins with consistent ones without adapting also the contained expressions as consistent join operators require that the input sequence maps are consistent.

A variant using only consistent operators is, nevertheless, also possible and, as expected, even more compact:

$$\begin{aligned}\mathcal{F}_{v_2,v_4}\Big(\ &(\dddot{\mu}_{v_1,v_4}(D,Q)\ \dddot{\bowtie}_{\cap}\ \dddot{\pi}_{v_1}(\dddot{\mu}_{v_1,v_3}(D,Q)\ \dddot{\bowtie}_{\cap}\ \dddot{\mu}_{v_3}(D,Q,\text{'Emperor'})))\\&\dddot{\bowtie}_{\cap}\ \dddot{\mu}_{v_1,v_2}(D,Q)\ \dddot{\bowtie}_{\cap}\ \dddot{\pi}_{v_2}(\dddot{\mu}_{v_2,v_5}(D,Q)\ \dddot{\bowtie}_{\cap}\ \dddot{\mu}_{v_5}(D,Q,\text{'Emperor'}))\ \Big)\end{aligned}$$

Here, we use a join variant (denoted by a $\cap$ subscript) that allows only input sequence maps with disjoint edge covers and is slightly more efficient than the general join, $\ddot{\bowtie}$, which could be employed here as well.

For more examples, see Section 3.8 and 3.9.

The sequence map operators are introduced as sequence-at-a-time operators, i.e., given one or more input sequence maps they compute all their results at once. In Section 3.9, we show how to obtain a iterator-based evaluation where results are generated tuple-at-a-time. The disadvantage of that scheme is that we are no longer free to rearrange the expression of the CIQCAG expression, but rather cluster all expressions modifying values for a certain variable. Also, for general graph or even CIG queries the benefit of a tuple-at-a-time evaluation is fairly low as there are no bounds on the number of nodes related to a single node under a given relation and thus all previously computed answers may still be related to nodes (matching with their parent variable) that come "later" in the processing.

On tree or forest data, however, there are such bounds: Namely, the number of parents of a node is limited by 1 for direct tree relations, i.e., relations with image disjointness property, and by d for closure relations, i.e., relations with image containment property, where d is the depth of the base relation. However, the presence of closure relations still prevents effective pruning of already considered bindings in a sequence map unless we assume that the orders associated with the incoming relations of all pairs of parent-child variables are consistent, i.e., if a node n_1 is before another node n_2 in the parent order, all the nodes in n_1's image are before or the same as the nodes in n_2's image. An example of such relations are child and descendant from XPath and, actually, all pairs of base relation and corresponding closure relation. For queries containing only such order-compatible relations, we can prune already considered bindings earlier than in the general case and arrive at a tighter bound for the space used by their evaluation, viz. $\mathcal{O}(q \cdot d)$ which coincides with the lower bound for such queries shown in [51].

3.2 Interval Access to a Relational Structure

Before we turn to the actual operators, we briefly outline the physical storage for a relational structure:

In Section 2.2, we define a relational structure D as a tuple $(R_1^D, \dots, R_k^D, O^D)$ over some relational schema $(R_1[U_1], \dots, R_k[U_k])$. O^D associates with each binary relation R_i^D a total order on the (finite) domain.

For the description of the CIQCAG operators, we assume a *specific way of accessing* a given relational structure (which provides the logical description of the queried data):

(1) For each order $<_N \in \operatorname{rng} O^D$, we access the nodes of D in the induced sequence of $<_N$ over $\mathsf{Nodes}(D)$. Recall, that $<_N$ is a total order and thus has a unique induced sequence that faithfully represents that order. Intuitively, instead of querying pairs of nodes (n_1, n_2) with $n_1 <_N n_2$ we iterate over all nodes of D such that a node n is accessed at index i_n where $n_1 <_N n_2$ iff $i_{n_1} < i_{n_2}$.
For each induced sequence of an order $<_R = O^D(R)$ we also store the "flip" index $\mathsf{flip}_{<_R} \in \mathbb{N}$ such that, for all $i \le \mathsf{flip}_{<_R}$, $S_{<_R}[i] \in \operatorname{rng} R$ and, for all $j > \mathsf{flip}_{<_R}$, $S_{<_R}[j] \notin \operatorname{rng} R$. Recall, from the definition of a relational structure in Section 2.2, that in the associated order $<_R$ of each relation R all nodes in $\operatorname{rng} R$ precede all nodes not in $\operatorname{rng} R$. Thus, each induced sequence has a (unique) flip index.

(2) Each binary relation R is accessed through its minimal interval representation $I(R)$ over its associated order $O^D(R)$. Recall, that a minimal interval representation maps each node in $n \in \operatorname{dom} R$ to the minimal set of intervals needed to represent $R(n)$ over the induced sequence of $O^D(R)$.

From Theorem 2.9 and the fact that the induced sequence of a total order can be computed in $\mathcal{O}(n \log n)$, we conclude that, even if we store the relations and orders in D as sets of pairs (or tables), we can provide the above access in polynomial time:

Corollary 3.1. *Let $D = (R_1^D, \dots, R_k^D, O)$ be a relational structure and m be the maximum cost of membership test for $R_1^D, \dots, R_k^D$. Then its physical storage can be computed in $\mathcal{O}(k \cdot |\mathsf{Nodes}(D)|^2 \cdot m)$.*

If all relations in D are extensional, we assume m to be constant and thus $\mathcal{O}(k \cdot |\mathsf{Nodes}(D)|^2)$ as bound for the computation of the physical storage.

In the following, we assume that the iteration over the induced sequences is in $\mathcal{O}(n)$, membership test is constant, and the iteration over the interval representations $I(n)$ of a single node is in $\mathcal{O}(|I(n)|)$.

These time bounds can be achieved for arbitrary relations with associated order $<$ by storing the induced sequence for each order (at the same cost as storing the orders itself) and storing for each binary relation its minimal interval representation. The minimal interval representation $I_<$ can be stored in $\mathcal{O}(|I_<|) \leq \mathcal{O}(n \cdot \max\{|I(n)| : n \in \operatorname{dom} R\} \cdot \log n)$ space (if we assume constant pointer size, we can drop the logarithmic factor).

For tree, forest, and CIG data, this yields linear space in the number of nodes in D, for arbitrary relations the same or less space than storing the relation as a table of pairs.

It should also be noted, that the above time and space bounds can be achieved for structural relations over tree or forest data using an interval labeling of the elements in the tree, e.g., pre/post-encoding [16, 28] or BIRD [54]: Given two nodes, we can determine in constant time whether they are child, parent, descendant, ancestor, following, preceding, following-sibling, preceding-sibling, etc. just from looking at the node labels. We can also iterate over all, e.g., descendants of a node by a single range query on the pre- and post-values. The advantage of such labeling schemes is that they encode an entire set of structural relations using only $n \cdot l$ where l is the size of a label space. For most of these encodings, l is in the same order as the size of a pointer to a sequence over n, i.e., $l \leq \log n$.

In the following, we record bindings for all query variables in the order associated with the relation the *incoming* edge of the variable is labeled with in the query. For root variables we use some fixed, but arbitrary order $\prec_{std}$.[1] Thus, in a query Q against a relational structure D, we can associate with each query variable $v \in \mathsf{Vars}(Q)$ an order $\prec_v$ such that

$$\prec_v = \begin{cases} \prec_{std} & \text{if } v \text{ root variable} \\ O^D(\mathsf{rel}(v_1)^D) & \text{otherwise} \end{cases}$$

3.2.1 Storing and Managing Interval Sets

Bindings in a sequence map may be associated with a set of intervals and their associated child variables, i.e., a relation over $\mathsf{Vars}(Q) \times \mathbb{N} \times \mathbb{N}$. For each binding and child variable, the intervals are non-overlapping. We assume that this set is stored partitioned by child variable and, for each such variable, the non-overlapping intervals are in order of their *start index*.

[1]The fixed order ensures that over several constraints for the same root variable bindings are recorded in the same order. This is exploited in the (merge) join algorithm discussed in Section 3.4.1.

For the sequence map operations, we introduce four algorithms on such interval sets (or, where convenient, on interval sets for a single child variable): Adapt, Algorithm 8, slides and/or shrinks a set of intervals with the help of a change set, that maps indices of the sequence those intervals reference to a indices of a subsequence of that sentence such that the same binding is at the position of the index in the original sequence as at the position of its image in the subsequence. For details, see Section 3.4.1. JoinInts, Algorithm 9, computes the minimal interval sets representing the join (or intersection) of the two interval sets. DifferenceInts, Algorithm 14, computes the minimal interval set representing the difference of two interval sets. UnionInts, Algorithm 25, computes the minimal interval set representing the union of two interval sets. All three algorithms construct the new interval set ordered by increasing start index.

Note, that all these algorithms are linear in the size of the input sets as they can exploit **(1)** that the interval sets are stored in order of their *start index* and **(2)** that the intervals in each set are non-overlapping. The ordered storage allows us to avoid sorting the intervals. Thus, the complexity for these algorithm does not contradict results on interval merge or union, e.g., in [33].

3.3 Initialize (from Relation)

The basic operation when constructing a sequence map representing the answers of a tree query is the *initialization* of a sequence map from a single given relation such that the resulting sequence map represents the given relation. There are two variants of the initialization operation, one for unary relations and one for binary relations. These are separate as for binary relations we need to store not only bindings for the two query variables involved but also interval pointers between those relations.

Initializing a sequence map with a unary relation is essentially the same as turning a unary relation into a sequence over the order associated with its query variable. The result is a *consistent* sequence map such that its induced relation represents the given unary relation:

Definition 3.1 (Initialization of unary relations)**.** Let D be a relational structure, Q a tree query, $v \in \mathsf{Vars}(Q)$, and R a relation with relation name in $\mathcal{L}_V^D(v)$. Then, $\dddot{\mu}_v(D, Q, R)$ returns a *consistent* sequence map $\overset{Q\to D}{\mathsf{SM}}$ such that

(1) R is the induced relation of $\overset{Q\to D}{\mathsf{SM}}$ (represents the unary relation on v) and

(2) $\overset{Q\to D}{\mathsf{SM}}|_v = \overset{Q\to D}{\mathsf{SM}}$ (contains bindings only for v).

The edge cover associated with $\dddot{\mu}_v(D, Q, R)$ is empty.

Algorithm 2 computes such a sequence map for a given input relation. The resulting sequence map is consistent as there are no failure markers (the algorithm does not introduce any and the nodes in S are distinct from any failure marker) and no dangling bindings (as there are bindings in the sequence map only for v and, thus, the parent of v, if there is any, is not in the sequence map's domain).

At first glance, asking for a consistent sequence map is the obvious choice for this operator. However, e.g., when the selectivity for a unary relation is very low, we can actually profit from an inconsistent sequence map containing an entry for all nodes in D but failure markers where those entries are not in the given unary relation as this slightly simplifies, e.g., the join algorithm (cf. Section 3.4.1).

We denote with $\dddot{\mu}_v^{\lightning}(D, Q, R)$ the sequence map initialization where we allow also inconsistent sequence maps and realize this operation by replacing lines 1–7 as follows:

```
Algorithm 2: $\ddot{\mu}_v(D, Q, R)$
    input : Relational structure $D$, tree query $Q$, variable $v \in \mathsf{Vars}(Q)$, unary relation $R$ with relation
            name in $\mathcal{L}^D_V(v)$.
    output: Consistent sequence map representation of $R$ and associated edge cover.
1 $S \leftarrow$ induced sequence for $<_v$ over $\mathsf{Nodes}(D)$ ;
2 $S' \leftarrow \varnothing$ ;
3 $j \leftarrow 1$ ;
4 for $i \leftarrow 1$ to $|S|$ do
5     if $S[i] \in R$ then
6         $S'[j] \leftarrow S[i]$;
7         $j \leftarrow j + 1$ ;
8 return $\{(v, S')\}, \varnothing$
```

```
$S \leftarrow$ induced sequence for $<_v$ over $\mathsf{Nodes}(D)$ ;
foreach $i \leftarrow 1$ to $|S|$ do
        if $S[i] \notin R$ then
            $S[i] \leftarrow \lightning$
return $\{(v, S)\}$
```

Here, we employ *in-place* editing of the sequence S and merely "bomb" entries not in the given relation with a failure marker.

Theorem 3.1. *Both, $\ddot{\mu}_v(D, Q, R)$ and $\dddot{\mu}^{\lightning}_v(D, Q, R)$, can be computed in $\mathcal{O}(n \cdot m)$ where $n = |\mathsf{Nodes}(D)|$ and m is the cost for the membership test in R.*

Proof. Algorithm 2 computes $\ddot{\mu}_v(D, Q, R)$. It loops over $|S| = |\mathsf{Nodes}(D)|$ elements of S. For each element, the membership in R is tested. The same reasoning applies to the modified version of Algorithm 2 for computing $\dddot{\mu}^{\lightning}_v(D, Q, R)$. □

For a binary relation, we not only need to record bindings for two variables, the relations parent and child variable, but also the interval pointers referencing related bindings from bindings of the parent variable to bindings of the child variable. Again, we require that the resulting sequence map is consistent.

Definition 3.2 (Initialization of binary relations)**.** Let D be a relational structure, Q a tree query, and $v_1, v_2 \in \mathsf{Vars}(Q)$. Then $\ddot{\mu}_{v_1,v_2}(D, Q)$ returns a *consistent* sequence map $\overset{Q \to D}{\mathsf{SM}}$ such that

(1) $\mathsf{rel}(v_2)^D$ is the induced relation of $\overset{Q \to D}{\mathsf{SM}}$ (represents the relation between v_1 and v_2) and
(2) $\overset{Q \to D}{\mathsf{SM}}|_{v_1,v_2} = \overset{Q \to D}{\mathsf{SM}}$ (contains bindings only for v_1 and v_2).

The associated edge cover for $\ddot{\mu}_{v_1,v_2}(D, Q)$ is $\{(v_1, v_2)\}$.

Recall, that by the definition of a sequence map (part (3)), the bindings of v_1 in $\ddot{\mu}_{v_1,v_2}(D, Q)$ are ordered by $O(\mathsf{rel}(v_1)^D)$ and those of v_2 by $O(\mathsf{rel}(v_2)^D)$.

Algorithm 3 computes such a sequence map in linear time:

Algorithm 3: $\dddot{\mu}_{v_1,v_2}(D,Q)$

```
   input  : Relational structure $D$, tree query $Q$, variables $v_1, v_2 \in \mathsf{Vars}(Q)$.
   output: Consistent sequence map representation of $\mathsf{rel}(v_2)^D$.
1  $I$ ← minimal interval representation of $\mathsf{rel}(v_2)^D$ ;
2  $S_1$ ← induced sequence for $<_{v_1}$ over $\mathsf{Nodes}(D)$ ;
3  $S_1' \leftarrow \varnothing$ ;
4  $j \leftarrow 1$ ;
5  for $i \leftarrow 1$ to $|S_1|$ do
6      if $I(S_1[i]) \neq \varnothing$ then
7          $S_1'[j] \leftarrow (S_1[i], I(S_1[i]))$ ;
8  $S_2$ ← induced sequence for $<_{v_2}$ over $\mathsf{Nodes}(D)$ ;
9  return $\{(v_1, S_1'), (v_2, S_2|_{\{1,\ldots,\mathsf{flip}_{<_2}\}})\}, \{(v_1, v_2)\}$
```

Theorem 3.2. *$\dddot{\mu}_{v_1,v_2}(D,Q)$ can be computed in $\mathcal{O}(n \cdot m_I)$ where $n = |\mathsf{Nodes}(D)|$ and m_I is the cost for accessing $I(n)$ for any node $n \in \mathsf{Nodes}(D)$.*

Proof. Algorithm 3 computes a consistent sequence map with the induced relation $\mathsf{rel}(v_2)^D$: The sequence map is consistent as there are no failure markers (the algorithm does not introduce failure markers and the entries in S_1 and S_2 are nodes from D, not failure markers) and no dangling bindings (the parent variable of v_1, if there is any, is not mapped; for v_2, each binding has a corresponding binding of v_1 by definition of $\mathsf{flip}_{<_2}$). The induced relation of the sequence map is $\mathsf{rel}(v_2)^D$: For each pair $(n_1, n_2) \in \mathsf{rel}(v_2)^D$, n_1 is in $\overset{D\to Q}{\mathsf{SM}}(v_1)$ by the construction of S_1' in line 3–7; n_2 is in $\overset{D\to Q}{\mathsf{SM}}(v_2)$ as all $n_2 \in \operatorname{rng} \mathsf{rel}(v_2)^D$ are associated with an index $\leq \mathsf{flip}_{<_2}$ by definition. Finally, since I is an interval representation of $\mathsf{rel}(v_2)^D$ there is an interval in $I(n_1)$ and thus in $\overset{D\to Q}{\mathsf{SM}}(v_1)$ such that the index associated with n_2 in $\overset{D\to Q}{\mathsf{SM}}(v_2)$ is in that interval.

Given an interval representation of $\mathsf{rel}(v_2)^D$ and m_I time access to $I(n)$ for each $n \in \mathsf{Nodes}(D)$, Algorithm 3 runs in time $\mathcal{O}(n \cdot m_I)$: it loops over at most n elements of S_1, for each obtaining $I(n)$. The restriction of S_2 to elements before $\mathsf{flip}_{<_2}$ can be done in $\mathcal{O}(|S_2|) \leq \mathcal{O}(n)$ time. □

As for the unary initialization, we can define a variant of the binary initialization that drops the consistency requirement from the above definition. We denoted this variant as $\dddot{\mu}^{\lightning}_{v_1,v_2}(D,Q)$. Algorithm 4 marks bindings for v_1 with no related bindings for v_2 with a failure marker (instead of dropping them) and does not limit S_2 to only those elements in $\operatorname{rng} \mathsf{rel}(v_2)^D$. It has obviously the same time complexity as the one computing a consistent sequence map, but operates *in-place*, i.e., without building a second sequence for bindings of v_1.

3.4 Combine

Once a sequence map is initialized by a single unary or binary relation, we need to be able to combine multiple sequence maps to evaluate a tree query containing more than one relation.

The essential operation for combining sequence maps is the *join* in analogy to a natural join on relations: take two sequence maps and, for all shared variables, retain bindings contained in both sequence

Algorithm 4: $\ddot{\mu}^{\lightning}_{v_1,v_2}(D, Q)$

```
input  : Relational structure $D$, tree query $Q$, variables $v_1, v_2 \in \mathsf{Vars}(Q)$.
output: Sequence map representation of $\mathsf{rel}(v_2)^D$.
1 $I \leftarrow$ minimal interval representation of $\mathsf{rel}(v_2)^D$ ;
2 $S_1 \leftarrow$ induced sequence for $\prec_{v_1}$ over $\mathsf{Nodes}(D)$ ;
3 for $i \leftarrow 1$ to $|S_1|$ do
4     if $I(S_1[i]) \neq \emptyset$ then
5         $S_1[i] \leftarrow (S_1[i], I(S_1[i]))$ ;
6     else
7         $S_1[i] \leftarrow \lightning$ ;
8 $S_2 \leftarrow$ induced sequence for $\prec_{v_2}$ over $\mathsf{Nodes}(D)$ ;
9 return $\{(v_1, S_1'), (v_2, S_2)\}, \{(v_1, v_2)\}$
```

maps, for variables occurring only in one sequence map, retain the bindings in those sequence map. In fact, this corresponds to a join of the induced relations of the two sequence maps and yields an important property of sequence maps: They are *closed under join*, i.e., the join of the induced relations of two sequence maps can be represented as a sequence map, cf. Theorem 3.9. There is no direct analog to the relational cross product, since a sequence map, in contrast to a relation, may not contain more than one sequence of bindings for a variable/attribute. If we join two sequence maps with disjoint domains, the result is, as in the relational case, the cross product (of the induced relations).

For the evaluation of proper tree queries, join is the only combine operation needed. However, we also investigate two more combine operations, *union* and *difference* with similar semantics to those for relations. These allow also queries where some parts are negated or alternatives to other parts to be evaluated in a sequence map (rather than on the level of flat relations). Intersection is omitted as it can be expressed through union and difference and, moreover, as the natural join yields intersection if the domain and edge covers of the input sequence maps are the same.

As for the corresponding operations on relations, we place certain restrictions on the shape of the sequence maps allowed in these operations: At least the involved sequence maps must map the same variables (i.e., the same nodes of the underlying query are covered) and edge cover (i.e., the same edges of the underlying query are covered). However, it turns out that this restriction does not suffice for combining sequence maps with union or difference:

Consider the sequence maps S_1 and S_2 in Figure 20 ($S_1 = \big\{v_1 \mapsto \{(1, d_1 \mapsto \{(v_2, 1, 1)\})\}, v_2 \mapsto \{(1, d_3 \mapsto \{(v_3, 1, 1)\})\}, v_3 \mapsto \{(1, d_4 \mapsto \emptyset)\}\big\}$, $S_2 = \big\{v_1 \mapsto \{(1, d_2 \mapsto \{(v_2, 1, 1)\})\}, v_2 \mapsto \{(1, d_3 \mapsto \{(v_3, 1, 1)\})\}, v_3 \mapsto \{(1, d_5 \mapsto \emptyset)\}\big\}$). The induced relations are $R_1 = \{(d_1, d_3, d_4)\}$ and $R_2 = \{(d_2, d_3, d_5)\}$ and $R_1 \cup R_2 = \{(d_1, d_3, d_4), (d_2, d_3, d_5)\}$. This is **(1)** different from the induced relation $\{(d_1, d_3, d_4), (d_1, d_3, d_5), (d_2, d_3, d_4), (d_2, d_3, d_5)\}$ of sequence map S_3 which is the result if we "union" the bindings for adjacent pairs of variables independently (as we can do for the join). Nevertheless, S_3 is the smallest sequence map whose induced relation contains $R_1 \cup R_2$. **(2)** a relation that *can not be represented by any sequence map* as it does not exhibit a lossless-join decomposition into binary relations over each pair of adjacent variables. In fact, there are *no* multivalue dependencies in $R_1 \cup R_2$. Similar

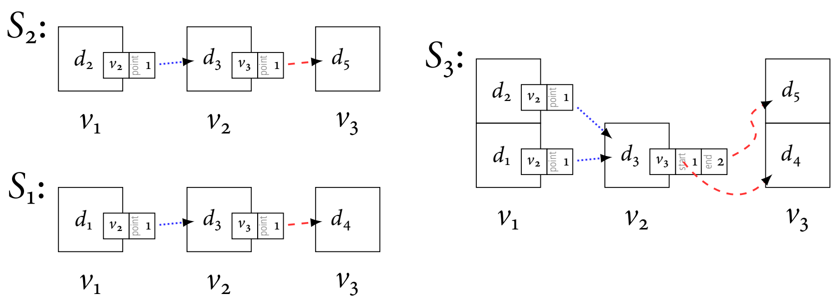

Figure 20. Sequence maps for illustrating "separate union" problem

observations apply for difference, thus yielding the following result:

Theorem 3.3. *The union (difference) of the induced relations of two sequence maps is, in general, not an induced relation of any sequence map.*

To obtain a union and difference operation on sequence maps that is well-defined and intuitive w.r.t. the induced relations of the involved sequence maps, we restrict union and difference to single-variable or single-edge sequence maps: A single-variable sequence map contains bindings for a single query variable only (and, thus, has an empty edge cover). A single-edge sequence map contains bindings for two adjacent query variables v and v' and has an associated edge cover $\{(v, v')\}$.

On single-variable (single-edge) sequence maps, we can define union and difference operations such that the result is a single-variable (single-edge) sequence map and such that the union or difference of the induced relations of two single-variable (single-edge) sequence maps is the induced relation of the resulting sequence map.

3.4.1 Join

The first combination operator for sequence maps, $\overset{\dots\lightning}{\bowtie}_{\cap}(\overset{D\to Q}{\mathsf{SM}}_1, \overset{D\to Q}{\mathsf{SM}}_2)$, *joins* two sequence maps into one, such that the resulting sequence map represents the natural join of the two induced relations of the two input sequence maps. We first introduce a more restrictive variant of the general join that limits the overlapping between the edges covered by two sequence map. This allows us to carry over the interval pointers from the input sequence maps unchanged or only slightly adapted.

Definition 3.3 (Sequence map join (disjoint edge covers)). Let D be a relational structure, Q a tree query, and S_1, SM_2 two sequence maps for D over Q such that *their associated edge covers are disjoint*. Then $\overset{\dots\lightning}{\bowtie}_{\cap}(S_1, S_2)$ returns a sequence map $\overset{D\to Q}{\mathsf{SM}}_3$ such that

(1) the induced relation of $\overset{D\to Q}{\mathsf{SM}}_3$ is the natural join of the induced relations of S_1 and S_2, i.e., $R_{\overset{D\to Q}{\mathsf{SM}}_3} = R_{S_1} \bowtie R_{S_2}$.

(2) $\overset{D\to Q}{\mathsf{SM}}_3|_{\mathrm{dom}\, S_1 \cup \mathrm{dom}\, S_2} = \overset{D\to Q}{\mathsf{SM}}_3$ (contains bindings only for variables mapped either in S_1 or in S_2).

The associated edge cover for $\overset{\dots\lightning}{\bowtie}_{\cap}(S_1, S_2)$ is the *union* of the edge covers associated with S_1 and S_2.

Note that this definition yields a sequence map that leaves bindings for *non-shared* variables unchanged from either sequence map (correspond to attributes of their induced relations that occur only in one of the two relations, for these attributes the relational join subsumes to a cross product and thus retains any combination of the bindings). For *shared* variables, only those bindings are retained that occur in both sequence maps. This also applies to the (interval pointer) references from bindings of a parent variable v to a child variable v' of v: They are contained only in one of the sequence maps (due to the edge cover restriction), for the other sequence map the induced relation records *any* combination of bindings by definition (cf. Section 2.2).

The restriction on the edge covers on S_1 and S_2 is imposed to ensure that for any pair of variables v, v' only one of the sequence maps may contain interval pointers from v to v', though sequence maps may contain *bindings* for v and v'. In other words, each *edge* of the query is enforced by at most one of the two sequence maps.

For a given CIQCAG expression, the edge cover of each sequence map (created as result of any subexpression) can be determined statically, without knowledge about the data the expression is to be evaluated against: For each CIQCAG operation, we define here either how the edge cover is computed from its input (for combine and reduce operations) or the edge cover is independent from the input data (initialization). Thus, we can also statically determine whether a CIQCAG join expression is valid or violates the edge cover restriction defined above.

The above definition does not demand that the resulting sequence map is consistent. Therefore, Algorithm 5 computes a sequence map that represents the join of the induced relations as demanded in the definition of $\dddot{\bowtie}^{\lightning}_{\cap}()$, but may be inconsistent: It "bombs" bindings not contained in both sequence maps rather than dropping them entirely. This has the effect that interval points can remain unchanged (but no reference an interval containing possibly bombed entries). Note, that interval pointers to bindings of a variable occur only in one of the two sequence maps as the incoming edge of each variable is unique (since the query is tree-shaped) and the edge covers are disjoint (and thus the unique incoming edge is only covered by one of the two sequence maps). This allows line 16 where we simply throw together intervals from both sequence maps. Finally, observe that by the definition of the initialization of a sequence map, bindings for the same query variable occur in the *same* order in all sequence maps for that query. Thus the bindings of a variable shared between the two sequence maps to be joined are ordered the same.

These observations are exploited in Algorithm 5 to give a merge-join [21] style algorithm for the join of two sequence maps with disjoint edge cover that has linear time complexity in the (combined) size of the input sequence maps. Since the bindings are already in the same order, we can omit the sort phase of the merge join and immediately merge the two binding sequences. However, we need to ensure that not only the order but also the number of bindings (and the position of eventual failure markers, cf. lines 18–20) reflects that for the same variable v in the sequence map where v's incoming edge is in the edge cover (lines 9–11).

Theorem 3.4. *Algorithm 5 computes $\dddot{\bowtie}^{\lightning}_{\cap}(S_1, S_2)$ for sequence maps with disjoint edge cover and set of shared variables Shared in $\mathcal{O}(b^{total}_{Shared} \cdot i) \leq \mathcal{O}(|Shared| \cdot n \cdot i)$ time where b^{total}_{Shared} is the total number of bindings associated in either sequence map with a variable in Shared and i is the maximum number of intervals associated with any such binding. For tree, forest, and CIG data $i = 1$, for arbitrary graph data $i \leq n$.*

Proof. Algorithm 5 computes $S = \dddot{\bowtie}^{\lightning}_{\cap}(S_1, S_2)$: For any variable v, if a binding for v occurs in the induced relation of both sequence maps, it occurs also in S due to lines 15–17. If v's incoming edge is in the edge

Algorithm 5: $\ddot{\bowtie}^{\lightning}_{\cap}(S_1, S_2)$

```
input  : Sequence maps $S_1$ and $S_2$ with *disjoint* edge covers
output: Sequence map res representing the join of the induced relations of the input maps
$\text{EC}_1 \leftarrow \mathsf{edgeCover}(S_1)$; $\text{EC}_2 \leftarrow \mathsf{edgeCover}(S_2)$;
$\mathsf{AllVars} \leftarrow \mathrm{dom}\, S_1 \cup \mathrm{dom}\, S_2$;
$\mathsf{SharedVars} \leftarrow \mathrm{dom}\, S_1 \cap \mathrm{dom}\, S_2$;
$\mathsf{res} \leftarrow \varnothing$ ;
foreach $v \in \mathsf{AllVars}$ do
    if $v \notin \mathrm{dom}\, S_2$ then  $\mathsf{res} \leftarrow \mathsf{res} \cup \{(v, S_1(v))\}$ ;
    else if $v \notin \mathrm{dom}\, S_1$ then  $\mathsf{res} \leftarrow \mathsf{res} \cup \{(v, S_2(v))\}$;
    else                                                            // v is in both
        // 1 is the primary (fallback if v is in neither edge cover)
        $\mathsf{iter} \leftarrow S_1(v)$; $\mathsf{alt} \leftarrow S_2(v)$ ;
        if $(v', v) \in \text{EC}_2$ for some $v'$ then
            // v is sink in EC$_2$, thus the order and number of entries must be as in 2 (it can not be sink
              in EC$_1$ as Q tree query and edge covers disjoint)
            $\mathsf{iter} \leftarrow S_2(v)$; $\mathsf{alt} \leftarrow S_1(v)$;
        $S \leftarrow \varnothing$; $i, j, k \leftarrow 1$ ;
        while $i \leq |\mathsf{iter}|$ do
            $(n_1, i) \leftarrow \mathsf{nextBinding}(S_1(v), i)$; $(n_2, j) \leftarrow \mathsf{nextBinding}(S_2(v), j)$; if $n_1 = n_2$ then
            // Retain binding if same
                $S[k] = (n_1, \mathsf{intervals}(\mathsf{iter}[i]) \cup \mathsf{intervals}(\mathsf{alt}[j]))$ ;
                $i{+}{+}$; $j{+}{+}$; $k{+}{+}$;
            else if $n_1 < n_2$ then                        // "bomb" if in iter but not in alt
                $S[k] = \lightning$;
                $i{+}{+}$; $k{+}{+}$;
            else                                        // skip binding if in alt but not in iter
                $j{+}{+}$;

        $\mathsf{res} \leftarrow \mathsf{res} \cup \{(v, S)\}$ ;

return res
```

Algorithm 6: NextBinding(S, i)

```
input  : Sequence $S$ containing, possibly, failure markers and start index $i$
output: The next element in $S$ at or after $i$ that is not a failure marker and its index or $(\infty,\infty)$ if no
        such binding exists
1 for $j \leftarrow i$ to $|S|$ do
2     if $S[j]$ is not a failure marker then break ;
3 if $j = |S|$ and $S[j]$ failure marker then return $(\infty,\infty)$ ;
4 return $(S[j], j)$
```

cover of one of the sequence maps S', lines 9–11 ensure that the sequence of bindings for v is the same (except that some bindings are "bombed") in S as in S'. For the parent v' of v, if a binding is retained the set of intervals from both sequence maps are copied en block. There are only intervals in S' (as (v', v) is not in the edge cover of the other sequence map) and thus only those relations between bindings of v and v' as in the induced relation of S' are retained. This is proper as in the induced relation of the other sequence map *all* bindings of v are related to all bindings of v' by definition of the induced relation. Both input sequences may be inconsistent: The presence of failure markers in either sequence does not affect the correctness of the algorithm: failure markers in alt are skipped, failure markers in iter are retained (lines 15–17) as intended. Dangling bindings do not affect the algorithm.

Algorithm 5 loops over all shared variables of S_1 and S_2 and for each such variable it iterates over all bindings in the primary sequence map iter and corresponding bindings in alt, skipping, if necessary, bindings in alt not in iter. In the loop lines 13–22 i or j is incremented (possibly multiple times, if failure markers are skipped in NextBinding) until either $i > |\mathsf{iter}|$. If j ever becomes $> |\mathsf{alt}|$ subsequent calls of $\mathsf{binding}(()\mathsf{alt}[j])$ return, by definition, a value larger than all $n \in \mathsf{Nodes}(D)$.

Thus the algorithm touches, for each shared variable, each entry in either sequence map at most once (and touches one proper (not a failure mark) entry in each step of the loop 13–22). Thus it runs in $\mathcal{O}(b^{total}_{\mathsf{Shared}} \cdot i)$ where $b^{total}_{\mathsf{Shared}}$ is the total number of bindings in both sequence maps for a shared variable and i is the maximum number of intervals per binding. This is bound by $\mathcal{O}(|\mathsf{Shared}| \cdot n \cdot i)$ for any sequence map (including sequence maps for arbitrary graphs) as shown in Section 2.4. □

It is worth pointing out, that in a CIQCAG expression for a tree query any variable is shared at most once for each in- or outgoing edge and for each unary relation associated with the variable. Thus, even if there are $\mathcal{O}(q)$ joins in the expression, the accumulated number of shared variables among all those joins is also only $\mathcal{O}(q)$ an thus the complexity for only those joins is bounded by $\mathcal{O}(q \cdot n \cdot i)$.

3.4.1.1 Consistent Join

In contrast to the map initialization, joining two sequence maps becomes considerably more complicated if we modify the definition to require that the resulting sequence map is *consistent*.

Definition 3.4 (Consistent sequence map join (disjoint edge covers)). Let D be a relational structure, Q a tree query, and S_1, S_2 two *consistent* sequence maps for D over Q such that *their associated edge covers* *are disjoint*. Then $\dddot{\bowtie}_{\cap}(S_1, S_2)$ returns a *consistent* sequence map $\overset{D\to Q}{\mathbf{SM}}_3$ that also fulfills all the conditions for $\dddot{\bowtie}^{\lightning}_{\cap}(S_1, S_2)$.

When computing the consistent join sequence map, we have to adapt the interval referencing from one binding to a sequence of bindings of one of its child variables. This is necessary even if both input sequence maps are consistent, as some bindings may occur in one but not in the other sequence map and thus intervals shrink or even collapse.

We can use the previous algorithm for the possibly inconsistent variant as a starting point. However, we process now all variables that are either shared or from which a shared variable is reachable by edges covered by the union of the edge covers of the input sequence map. This is necessary, as changes to one variable v have to be propagated to its parent v', as some of the parents bindings may reference only bindings for v that are dropped. Such bindings for v' must be dropped and, accordingly, may affect bindings for the parent of v' and so on. These variables are processed in inverse topological order w.r.t. the edges in the edge covers of the two sequence maps, i.e., child variables before parent variables. For each variable, we record how the indices of bindings have changed: Let i be an index for a binding n of variable v in one of the sequence map. Then, **(1)** if n also occurs in the other sequence map, we retain n and log $(v, \downarrow, i, k)$ where k is the new index for n and $\downarrow$ is the type of the log entry, here a mapping for a retained binding. **(2)** if n does not occur in the other sequence map, we drop it and log both the index (in the original sequence of bindings for v) of the next retained entry (with type $\hookrightarrow$) and of the last preceding retained entry (with type $\hookleftarrow$).

With these change logs, we can then adapt the intervals referring to bindings of v when we process the parent variable for v later (it comes after v in inverse topological order).

Algorithm 7 gives the computation of the consistent join for sequence maps with disjoint edge covers and reflects these modifications from Algorithm 5. Note, that for each variable and index there is *either* a $\downarrow$ entry in the change log or both a $\hookleftarrow$ and a $\hookrightarrow$.

We use an additional helper function Adapt that is detailed in Algorithm 8 and actually applies a "change log" to a set of intervals: For each interval, we look at the start index. If it has a $\hookrightarrow$ entry (and thus the referenced binding is not retained) we set s to the index in that entry, i.e., the index (in the original sequence of bindings) of the next retained binding. The same we do for the end index, but with its $\hookleftarrow$ entry, if there is any. After these adaptations, the start and end index might actually overlap if all bindings in the original interval have been dropped. Note, that the border cases are covered as we use ∞ to indicate that the start index is no "outside" the sequence and 0 to indicate that the end index is "outside", cf. line 21 and 24 in Algorithm 7 if the first element is bombed its $\hookleftarrow$ becomes 0 and line 26 if the last (or last few) element is bombed.

If that is not the case, the new start and end indices are retained and we can retrieve their $\downarrow$ entries with the indices in the new sequence. Finally, we ensure that intervals separated in the original sequence by entries dropped in the new sequence collapse (lines 9–12). This guarantees that the new set of intervals is actually minimal.

It is worth pointing out, that Adapt requires that the intervals are non-overlapping (as is guaranteed if they come from a sequence map, by definition of a sequence map). The generated intervals are again overlapping for log entries resulting from Algorithm 7.

For example for the interval set $\{(v, 3, 5), (v, 7, 7), (v, 10, 12)\}$ over a sequence from 1 to 12 and the log entries $\{(v, \hookrightarrow, 3, 4), (v, \hookleftarrow, 3, 0), (v, \downarrow, 4, 2), (c, \hookrightarrow, 5, 7), (v, \hookleftarrow, 5, 4), (v, \downarrow, 7, 3), (v, \hookrightarrow, 8, 9), (v, \hookleftarrow, 8, 7), (v, \downarrow$

Algorithm 7: $\ddot{\bowtie}_{\cap}(S_1, S_2)$

```
input  : Consistent $S_1$ and $S_2$ with disjoint edge covers
output: Consistent sequence map representing the join of the input maps

1  $\mathrm{EC}_1 \leftarrow \mathsf{edgeCover}(S_1)$; $\mathrm{EC}_2 \leftarrow \mathsf{edgeCover}(S_2)$; $\mathsf{Shared} \leftarrow \mathrm{dom}\, S_1 \cap \mathrm{dom}\, S_2$;
2  $\mathsf{PendingVars} \leftarrow \{v \in \mathsf{Vars}(Q) : v \in$
   $\mathsf{Shared} \vee$            $\exists$ path from $v$ to $v'$ in $\mathrm{EC}_1 \cup \mathrm{EC}_2$ with $v' \in \mathsf{Shared}\}$;
3  $\mathsf{res} \leftarrow \varnothing$; $\mathsf{AdaptedVars} \leftarrow \varnothing$; $\mathsf{Log} \leftarrow \{(S_1, \varnothing), (S_2, \varnothing)\}$;
4  while $\mathsf{PendingVars} \neq \varnothing$ do
       // Select some variable v without children
5      $v \in \{v' \in \mathsf{PendingVars} : \nexists\, v'' \in \mathsf{PendingVars} : \mathsf{parent}(v'') = v'\}$ ;
       // last records the start of the current interval of dropped bindings, $\infty$ indicates that there is no
         such interval
6      $\mathsf{last}_I \leftarrow \mathsf{last}_A \leftarrow \infty$; $\mathsf{iter} \leftarrow S_1$; $\mathsf{alt} \leftarrow S_2$; $S \leftarrow \varnothing$; $i, j, k \leftarrow 1$ ;
7      if $v \notin \mathrm{dom}\, S_1$ then $\mathsf{iter} \leftarrow S_2$; $\mathsf{alt} \leftarrow \bot$;
8      if $v \notin \mathrm{dom}\, S_2$ then $\mathsf{alt} \leftarrow \bot$;
9      while $i \leq |\mathsf{iter}(v)|$ do
10         $I \leftarrow \varnothing$; $n_1 \leftarrow \mathsf{binding}(\mathsf{iter}(v)[i])$;
11         $n_2 \leftarrow \mathsf{binding}(\mathsf{alt}(v)[j])$ or $n_1$ if $\mathsf{alt} = \bot$;
           // Compute the adapted intervals (if necessary) and bomb entry if all intervals dropped
12         if $(v, v') \in \mathrm{EC}_1 \cup \mathrm{EC}_2$ for some variable $v'$ then
13             $I_1 \leftarrow \mathsf{intervals}(\mathsf{iter}(v)[i])$; $I_2 \leftarrow \mathsf{intervals}(\mathsf{alt}(v)[j])$ or $\varnothing$ if $\mathsf{alt} = \bot$;
14             $I \leftarrow \mathtt{Adapt}(I_1, \mathsf{Log}(\mathsf{iter})) \cup \mathtt{Adapt}(I_2, \mathsf{Log}(\mathsf{alt}))$;
15             if $I = \varnothing$ then $n_1 \leftarrow 0$
16         if $n_1 = n_2$ then                                         // retain binding if same
17             $S[k] \leftarrow (n_1, I)$;
               // Log: i changes to k; i next good binding for all “bombed” bindings directly before i (if
                  there are none $\mathsf{last}_i = \infty$)
18             $\mathsf{Log}(\mathsf{iter}) \leftarrow \mathsf{Log}(\mathsf{iter}) \cup \{(v, \downarrow, i, k)\} \cup \{(v, \hookrightarrow, l, i) : \mathsf{last}_I \leq l < i\}$
               $\mathsf{Log}(\mathsf{alt}) \leftarrow \mathsf{Log}(\mathsf{alt}) \cup \{(v, \downarrow, j, k)\} \cup \{(v, \hookrightarrow, l, j) : \mathsf{last}_A \leq l < j\}$ $i$++; $j$++; $k$++;
               $\mathsf{last}_I \leftarrow \mathsf{last}_A \leftarrow \infty$;                        // incr. counters; reset last
19         else if $n_1 < n_2$ then                          // skip binding if in iter but not in alt
20             $\mathsf{last}_I \leftarrow \min(\mathsf{last}_I, i)$;                // start of current interval of only “bombs”
               // Record the last “good” binding before i
21             $\mathsf{Log}(\mathsf{iter}) \leftarrow \mathsf{Log}(\mathsf{iter}) \cup \{(v, \hookleftarrow, i, \mathsf{last}_I - 1)\}$; $i$++;
22         else                                              // skip binding if in alt but not in iter
23             $\mathsf{last}_A \leftarrow \min(\mathsf{last}_A, j)$ ;
24             $\mathsf{Log}(\mathsf{alt}) \leftarrow \mathsf{Log}(\mathsf{alt}) \cup \{(v, \hookleftarrow, j, \mathsf{last}_A - 1)\}$; $j$++;
25
26     set next “good” binding for open “bombed” intervals to $\infty$ in $\mathsf{Log}$;
27     $\mathsf{PendingVars} \leftarrow \mathsf{PendingVars} \setminus \{v\}$; $\mathsf{AdaptedVars} \leftarrow \mathsf{AdaptedVars} \cup \{v\}$;
28     $\mathsf{res} \leftarrow \mathsf{res} \cup \{(v, S)\}$;
29 copy bindings for variables not in $\mathsf{AdaptedVars}$ to $\mathsf{res}$
30 return $\mathsf{res}$
```

$, 9, 4), (v, \hookrightarrow, \infty), (v, \hookleftarrow, 10, 9), (v, \hookrightarrow, 12, \infty), (v, \hookleftarrow, 12, 9), \ldots\}$ (the log is larger but the remaining entries are not of interest). From the log, we can conclude that only the bindings for $4, 7, 9$ are retained. The interval $[3, 5]$ collapses to the interval $[1, 1]$ (the old 4 is now the 1st binding) and then collapses with $[7, 7]$ that is now $[2, 2]$. The interval $[10, 12]$ is dropped entirely. Thus the adapted set of intervals is $\{[1, 2]\}$ (note that 9, now 3, is not covered by the intervals in this set).

Algorithm 8: Adapt(Ints, Log)

```
input : Non-overlapping set of intervals Ints and set of “log” entries Log as generated from
        Algorithm 7
output: Set of intervals modified according to Log

 1 NewInts ← $\emptyset$ ;
   // For variables with entries in the “Log”: adapt
 2 foreach $v \in \pi_1$(Log) do
 3     lastStart ← $\infty$; lastEnd ← $\infty$;
 4     foreach $(v, s, \in$ Ints in order of start index do
 5         if $(v, \hookrightarrow, s,$ next$) \in$ Log then $s \leftarrow$ next;
 6         if $(v, \hookleftarrow, e,$ prev$) \in$ Log then $e \leftarrow$ prev;
 7         if $s > e$ then continue ;
           // Now there must be $\downarrow$ entries for s and e, as $\hookrightarrow$ and $\hookleftarrow$ entries always reference “good”
              entries or $\infty$ or 0 which are excluded above
 8         $(v, \downarrow, s, s_{\text{new}}), (v, \downarrow, e, e_{\text{new}}) \in$ Log ;
           // If we cannot extend the last interval, add it ...
 9         if lastEnd $\neq s_{\text{new}} + 1$ then
10             NewInts ← NewInts $\cup \{(v,$ lastStart, lastEnd$)\}$;
11             lastStart = $s_{\text{new}}$ ;
           // ... otherwise lastStart remains unchanged
12         lastEnd ← $e_{\text{new}}$ ;
       // Collect remaining interval
13     if lastStart $\neq \infty$ then NewInts ← NewInts $\cup \{(v,$ lastStart, lastEnd$)\}$;
   // For variables without entries in the “Log”: copy
14 foreach $(v, s, e) \in$ Ints with $v \notin \pi_1$(Log) do
15     NewInts ← $\{(v, s, e)\}$;
16 return NewInts
```

Theorem 3.5. *Algorithm 8 computes the adapted set of intervals for a given set of intervals I and a change log L in $\mathcal{O}(|I|)$ time assuming constant membership test in L.*

Proof. Note, that L contains for each pair (v, i) either one $\downarrow$ or one $\hookleftarrow$ and one $\hookrightarrow$ entry. Thus L's size is bounded by $\mathcal{O}(q \cdot n)$ where for a given sequence map $\overset{D \to Q}{\text{SM}}$ and constant membership test can be realized by an array over variables and indices, each cell containing either the $\downarrow$ or the $\hookleftarrow$ and $\hookrightarrow$ entry, with the same space bound.

The iteration over the intervals in order of start indices (line 4) is linear, since the intervals are stored ordered, cf. Section 3.2.1.

The algorithm iterates over all intervals in I and for each interval performs a series of membership tests in L or simply copies the interval if there are no change entries for the involved variable. □

The advantage of $\dddot{\bowtie}_{\cap}()$ vs. $\dddot{\bowtie}_{\cap}^{\lightning}()$ is quite obvious: we get a consistent and thus in most cases smaller resulting sequence map. The sequence map computed by $\dddot{\bowtie}_{\cap}^{\lightning}()$, on the other hand, most likely contains some redundancies. This effect is all the more pronounced the more selective the involved sequence maps are.

However, the computation of $\dddot{\bowtie}_{\cap}()$ is also significantly more involved than that of $\dddot{\bowtie}_{\cap}^{\lightning}()$ and requires $\mathcal{O}(|\mathcal{V}| \cdot n)$ additional space for the change log where $\mathcal{V}$ is the set of variables that are either shared or from which a shared variable is reachable by edges in the union of the edge covers of the input sequence maps (PendingVars in line 3). Where $\dddot{\bowtie}_{\cap}^{\lightning}()$ ignores non-shared variables (and can copy all intervals en-block), $\dddot{\bowtie}_{\cap}()$ must touch all shared variables and their ancestors and for each adapt all its intervals at least once to adapt start and end index. Thus, the total number of intervals associated to any binding for a variable in $\mathcal{V}$ forms a lower bound for $\dddot{\bowtie}_{\cap}()$. Algorithm 7 runs in $\mathcal{O}(b_{\mathcal{V}}^{total} \cdot i)$ where $b_{\mathcal{V}}^{total}$ is the total number of bindings for all variables in $\mathcal{V}$ in both sequence maps and i the maximal number of intervals per binding. This is bounded by $\mathcal{O}(q \cdot n \cdot i)$ where i is the maximal number of intervals per binding. For tree, forest, and CIG queries $i = 1$, for arbitrary queries $i \leq n$, cf. Section 2.4.

Theorem 3.6. *Algorithm 7 computes $\dddot{\bowtie}_{\cap}(S_1, S_2)$ for sequence maps with disjoint edge cover in $\mathcal{O}(b_{\mathcal{V}}^{total} \cdot i) \leq \mathcal{O}(q \cdot n \cdot i)$ time where i is the maximal number of intervals per binding and $b_{\mathcal{V}}^{total}$ is the total number of bindings for shared variables or variables from which shared variables are reachable by edges in the union of the edge covers of the two sequence maps, $n = |\mathsf{Nodes}(D)|$ and $q = |\mathsf{Vars}(Q)|$. It requires $\mathcal{O}(|V| \cdot n)$ additional space for the change log.*

Proof. Algorithm 7 computes $S = \dddot{\bowtie}_{\cap}(S_1, S_2)$: A tuple $t \in R_{S_1} \bowtie R_{S_2}$, iff for all variables v with $t[v] = n$ (1) n occurs among the bindings of v in both sequence maps and thus among the bindings of v in the resulting sequence map (lines 16–18) and, (2) for all variables v' with (v', v) in the edge cover of either sequence map, the interval pointer in all bindings for v' are adapted using the change log (note that only one of sequence maps contains bindings for v' with interval pointers to v since the edge covers are disjoint. The change log contains a mapping from each index i of S_1 to an index k in the result sequence map, if the binding with index i is retained. Otherwise, it references back and forward to the last previous and next following index of a binding that is not dropped. Adapt adapts the old intervals to the new indices. If all bindings in an interval are dropped, the interval is removed. If all bindings for v in all intervals for a binding n of v' are dropped (and thus, one of the sequence maps contains no tuples with bindings for v in any of the intervals for n and thus no tuple containing n), n is eliminated (cf. line 15 and 19–21).

The result of Algorithm 7 is consistent: There are no failure markers as the algorithm does not introduce any (and the input sequence maps are consistent). If there are no dangling bindings in the original sequence maps, then there are no dangling bindings in the resulting sequence map, as any binding that is retained and is covered in the original sequence map is also covered in the new sequence map by the construction of the adapted intervals (lines 17–18 and lines 21 and 24).

The algorithm loops over the variables in AllVars, each iteration removing one variable until AllVars is empty. For the inner loop reasoning analog to Algorithm 5 applies except if the binding is retained: Then we also call Adapt which requires time linear in the size of the intervals.

As Algorithm 5, the algorithm touches each entry in either sequence map at most once (and touches one such entry in each step of the loop 13–25), but for each entry it can not copy all intervals en block, but needs to adapt each interval. Thus it runs in $\mathcal{O}((b_{S_1} + b_{S_2}) \cdot i)$ where $b_{\mathsf{SM}}^{D \to Q}$ is the total number of bindings in a sequence map and i the maximal number of intervals per binding. This is bound by $\mathcal{O}(q \cdot n \cdot i)$ for any sequence map (including sequence maps for arbitrary graphs) as shown in Section 2.4.

Regarding the space bound, see AdaptIntervals and the proof of Theorem 3.5. □

Note that the algorithm actually handles and even corrects failure markers in the input sequence maps, but does not allow dangling bindings if the result is expected to be consistent. In other words, we could weaken the requirement that both input sequence maps are consistent to only that they do not contain dangling bindings.

Though, at the first glance, the time bounds for the two algorithms are similar, the above observation that variables in CIQCAG expressions for tree queries are shared at most once per edge and at most once per associated unary relation, does not hold. Rather, variables with many descendants in the query are likely to be considered in many joins, in particular if the constraints of a query a enforced in bottom-up fashion. In general, this means that for a CIQCAG expression with q joins evaluating a tree query we incur a total cost of $\mathcal{O}(q \cdot q \cdot n \cdot i)$ for the processing of all joins. This is an increase by a multiplicative factor q compared to the use of $\dddot{\bowtie}^{\lightning}_{\cap}()$.

If we compare this result with the combination of the possibly inconsistent $\dddot{\bowtie}^{\lightning}_{\cap}()$ with the propagate operator (cf. Section 3.5.3) we can observe the essential advantage of allowing (temporarily) inconsistent sequence maps: If we use explicit propagate we can touch and adapt the intervals of each variable once (after all restrictions for that variable and its descendant variables have been evaluated). With implicit propagate, we have to potentially "touch" them in each join and thus introduce an additional multiplicative factor in the order of the size of the query. For details, see Section 3.5.3.

For the join, as for many of the operators defined in the following, there is a variant for consistent and a variant for inconsistent variants, the prior requiring consistent sequence maps and returning consistent sequence maps, the latter allowing any sequence map and returning sequence maps with possible inconsistencies. All operators for consistent sequence *preserve interval-minimality*, i.e., if given interval-minimal, consistent sequence maps as input, they produce interval minimal consistent sequence maps. The propagation operator actually ensure that the resulting sequence map is *interval-minimal*.

3.4.1.2 General Join

The second variation of the sequence map join revolves around the restriction to sequence maps with disjoint edge covers. What is the effect if we allow the edge covers to overlap?

If we relax the edge cover restriction, we allow that both sequence maps represent some subset of the possible combinations of bindings for, e.g., the three variables v_1, v_2, v_3. At first glance, this seems to make it impossible to join such sequence maps and represent the result as a sequence map (rather than a relation where we track combinations of all variables rather than only references from v_1 to v_2 and, separately, from v_2 to v_3). Figure 21 shows two such relations where we can not join separately: If we consider only references from v_1 to v_2, we retain the pair (d_1, d_3). If we consider only references from v_2 to v_3, we retain the pair (d_3, d_4) since both pairs occur in each relation. Thus the resulting sequence map represents the relation $\{(d_1, d_3, d_4)\}$ which is different from $R_1 \bowtie R_2$.

v_1	v_2	v_3
d_1	d_3	d_4

v_1	v_2	v_3
d_1	d_3	d_5
d_2	d_3	d_4

Figure 21. Relations R_1 and R_2 where join on decomposed relations seems insufficient

However, the reason for this behavior is that R_2 from Figure 21 can actually not be represented as a sequence map (it does not exhibit a lossless-join decomposition into binary relations over the pairs of adjacent attributes): Should we reference d_4, d_5, or both from d_3's binding for v_2? In contrast to R_2, all relations representable as a sequence map actually ensure that if there is any tuple with (n, n') as bindings for v_2, v_3 then, for any combination of values for the remaining variables (here v_1), there must be a tuple with (n, n') as bindings for v_2, v_3, cf. Section 2.2. In our example R_2 must be extended with the tuples (d_1, d_3, d_4) and (d_2, d_3, d_5) to be amenable for representation as sequence map.

This property is what makes it possible to define a variant of the sequence map join that allows overlapping edge cover yet still computes the join for each adjacent pair of variables separately. Thus we can define the general sequence map join as follows:

Definition 3.5 (General sequence map join)**.** Let D be a relational structure, Q a tree query, and S_1, S_2 two *arbitrary* (consistent) sequence maps for D over Q. Then $\dddot{\bowtie}^{\lightning}(S_1, S_2)$ ($\dddot{\bowtie}(S_1, S_2)$returns a (consistent) sequence map $\overset{D\to Q}{\mathbf{SM}}_3$ that also fulfills all the conditions for $\dddot{\bowtie}^{\lightning}_{\cap}(S_1, S_2)$.

Algorithms for this join variant can be fairly easily derived from the Algorithms for $\dddot{\bowtie}^{\lightning}_{\cap}$ and $\dddot{\bowtie}_{\cap}$: The previous algorithms for a sequence map join use a union (line 16 in Algorithm 5, line 17 in Algorithm 7) to combine intervals from both sequence maps, since we know that one of the two sets of intervals is always empty (otherwise the edge covers overlap) and thus no further "joining" is necessary. For the general sequence map join, *both* sequence maps may contain interval pointers for the same edge and we have to ensure that only those references are retained for that an interval pointer exists in both input sequence maps.

To compute $\dddot{\bowtie}(S_1, S_2)$, we modify line 14 in Algorithm 7 to use a new function $\mathsf{JoinInts}$, instead of $\cup$ to combine the two adapted sets of intervals. This function is defined in Algorithm 9. First, for variables covered in both interval sets, we iterate (lines 3–16) in parallel over the two sets of intervals (that are non-overlapping and thus can be ordered, e.g., by start index). For each interval, we look obtain the next intervals of the other set as long as there is an overlapping or the interval is entirely to the right. In the latter case, we take another interval from the first set etc. For variables covered only in one interval set, we simply retain the existing intervals. With this adaptation, we obtain an algorithm for $\dddot{\bowtie}()$.

Note, that $\mathsf{JoinInts}$ is surprisingly simple as both sets contain non-overlapping intervals. Thus a simple traversal in the order of the start (or, equivalently, end) indices is possible. An alternative is the use of an interval tree or, since our intervals are non-overlapping, even of B-trees indexing start and end intervals. Though interval trees are, in general, very efficient (logarithmic) at answering point or stab queries (where we give an index and retrieve all intervals containing that index), we can exploit here that we are interested in the full interval cover instead of a single point query. Realizing $\mathsf{JoinInts}$ as a sequence

Algorithm 9: JoinInts(Intervals$_1$, Intervals$_2$)

```
input : Two sets Intervals$_1$, Intervals$_2$ of non-overlapping intervals with associated variables
output: Minimal, non-overlapping set of intervals covering all indices contained in intervals of
        both sets

NewIntervals ← $\varnothing$ ;
start ← $\bot$ ;
foreach $v \in \pi_1$(Intervals$_1$) $\cap$ $\pi_1$(Intervals$_2$) do
    $(v, s_1, e_1)$ ← interval in Intervals$_1$ with minimal start index $s_1$;
    $(v, s_2, e_2)$ ← interval in Intervals$_2$ with minimal start index $s_2$;
    while true do
        while $e_2 < s_1$ do
            // Current $(s_2, e_2)$ do not overlap, get the next
            Intervals$_2$ ← Intervals$_2$ $\setminus \{(v, s_2, e_2)\}$;
            if Intervals$_2$ = $\varnothing$ then break 2 (line 6);
            $(v, s_2, e_2)$ ← interval in Intervals$_2$ with minimal start index $s_2$;
        // There is some overlapping
        if $s_2 \leq e_1$ then
            NewIntervals ← NewIntervals $\cup \{(v, \max(s_1, s_2), \min(e_1, e_2)\}$;
        if $e_1 \leq e_2$ or $s_2 > e_1$ then
            // Current $(s_1, e_1)$ do not overlap (any further), get the next
            Intervals$_1$ ← Intervals$_1$ $\setminus \{(v, s_1, e_1)\}$;
            if Intervals$_1$ = $\varnothing$ then break (line 6);
            $(v, s_1, e_1)$ ← interval in Intervals$_1$ with minimal start index $s_1$;
foreach $v \in \pi_1$(Intervals$_1$) $\setminus$ $\pi_1$(Intervals$_2$) do
    NewIntervals ← NewIntervals $\cup \{(v, s, e) \in$ Intervals$_1\}$;
foreach $v \in \pi_1$(Intervals$_2$) $\setminus$ $\pi_1$(Intervals$_1$) do
    NewIntervals ← NewIntervals $\cup \{(v, s, e) \in$ Intervals$_2\}$;
return NewIntervals ;
```

of point queries (for all covered indices of a child relation) is possible, but comes at a higher complexity (by the logarithmic look-up time factor) than the above algorithm.

The algorithm touches each interval in both sets A and B exactly once. Each interval in A may be compared to several intervals of B, though all but the last of these are not compared to any other from interval from A. In total, this gives a bound of $\mathcal{O}(|A| + |B|)$. We assume that the access to the intervals in order of the start indices (lines 4–5) is constant for each interval and linear for the entire set. In other words, we assume that the intervals are stored ordered by the start index (cf. Section 3.2.1).

Theorem 3.7. *Algorithm 9 computes the join over the set of non-overlapping intervals I_1 and I_2 in $\mathcal{O}(|I_1| + |I_2|)$ time.*

Thus, the use of join interval does not significantly affect the complexity of Algorithm 7:

Theorem 3.8. *Algorithm 7 with JoinInts instead of $\cup$ in line 14, computes $\ddot{\bowtie}_{\cap}(S_1, S_2)$ for sequence maps with disjoint edge cover in $\mathcal{O}(b_{\mathcal{V}}^{total} \cdot i) \leq \mathcal{O}(q \cdot n \cdot i)$ time where i is the maximal number of intervals per binding and $b_{\mathcal{V}}^{total}$ is the total number of bindings for shared variables or variables from which shared variables are reachable by edges in the union of the edge covers of the two sequence maps, $n = |\mathsf{Nodes}(D)|$ and $q = |\mathsf{Vars}(Q)|$. It requires $\mathcal{O}(|V| \cdot n)$ additional space for the change log.*

Proof. The modified algorithm ensures that the resulting sequence map contains a reference from a binding n for variable v to a binding n' for a variable v' if and only if both input sequence maps contain such a reference.

In addition to the argument $\ddot{\bowtie}_{\cap}$, we can observe one more case: a pair of variables v, v' is in the edge cover of *both* sequence maps. In this case, any tuple t in the induced relation of either sequence map with $t[v] = n$ and $t[v'] = n'$ implies that there is an interval pointer in the set associated with n among the bindings for v that covers the index of n' in the bindings of v'. If t is in both sequence maps, such pointers exists in both of them and t is also in the join of the two induced relations. Then JoinInts ensures that there is an interval pointers in the new sequence map from n to an interval containing the index of n' among the bindings for v'. With the observation that the same holds for every pair of variables, we obtain that t is in the induced relation of the resulting sequence map.

The complexity follows from Theorems 3.6 and 3.7. □

For the variant that accepts possibly inconsistent sequence maps as result, denoted with $\ddot{\bowtie}^{\not\downarrow}(S_1, S_2)$, the modifications are a bit more extensive and move the inconsistent variant closer to the consistent one: we need to introduce change tracking for the secondary sequence (alt in Algorithm 5) and use those changes to adapt the intervals in alt so that we can join them with the intervals from iter (that, as before, do not need to be adapted). However, as for Algorithm 5 and in contrast to the consistent case, only intervals of shared variables and their direct parents, but not of other ancestor variables, must be adopted. The the move from union to JoinInts does not affect the cost for a single variable, however, we also need to cover direct parent variables. Since each shared variable has a unique parent (if any), this, nevertheless, increases the required time only a constant multiplicative factor.

Since we are now able to compute the join of two arbitrary sequence maps such that the induced relation of the resulting sequence map is the natural join of the induced relation of the input sequence maps, we can conclude the following theorem:

Theorem 3.9. *For any two sequence maps S_1, S_2, the join of their induced relations can be represented as a sequence map.*

As discussed above, the reason this statement holds is that the induced relations of the input sequence maps themselves can be represented in a sequence map and thus have lossless-join decompositions to binary relations over each pair of adjacent variables. Therefore, either a particular binding pair for two adjacent variables is eliminated from *all* binding tuples (if it is only in one of the sequence maps) or from *none*. In both cases, the resulting relation still is fully decomposable and thus can be represented by a sequence map.

3.4.1.3 Semi-Join

As a convenience, we also introduce a specific (left) semi-join operator $\ddot{\ltimes}^{\lightning}(S_1, S_2)$. The definitions and algorithms can be easily derived from the join operator and are only sketched here. First we adopt the (possibly inconsistent) definition of $\ddot{\ltimes}_{\cap}()$:

Definition 3.6 (Sequence map semi-join (disjoint edge covers)). Let D be a relational structure, Q a tree query, and S_1, S_2 two sequence maps for D over Q such that *their associated edge covers are disjoint* and S_2 is a single-variable sequence map. Then $\ddot{\ltimes}^{\lightning}(S_1, S_2)$ returns a sequence map $\overset{D\rightarrow Q}{\mathsf{SM}}_3$ such that

(1) the induced relation of $\overset{D\rightarrow Q}{\mathsf{SM}}_3$ is the left semi-join of the induced relations of S_1 and S_2, i.e., $R_{\overset{D\rightarrow Q}{\mathsf{SM}}_3} = R_{S_1} \ltimes R_{S_2}$.

(2) $\overset{D\rightarrow Q}{\mathsf{SM}}_3|_{\operatorname{dom} S_1 \cup \operatorname{dom} S_2} = \overset{D\rightarrow Q}{\mathsf{SM}}_3$ (contains bindings only for variables mapped either in S_1 or in S_2).

The associated edge cover for $\ddot{\ltimes}^{\lightning}(S_1, S_2)$ is the edge covers associated with S_1.

We limit the second sequence map to a single-variable sequence map to avoid cases, where the resulting relation no longer exhibits a full join decomposition and can thus not be represented in a sequence map. Thus, the semi-join operators is often combined with a projection (see Section 3.5.1) to a single variable.

We can compute $\ddot{\ltimes}^{\lightning}$ by a slightly modified version of Algorithm 5: We drop line 7 and thus do not retain bindings for variables only in the second sequence map. Furthermore, in line 15 we only retain intervals from iter not from alt (i.e., we drop the second operand from the union). The resulting algorithm computes $\ddot{\ltimes}^{\lightning}$ with the same complexity as Algorithm 5.

An analog modification for Algorithm 7 yields a consistent sequence map (we copy only bindings from the first sequence map in line 26 and drop I_2 and all its references in lines 12–15. Again, the change does not affect significantly the complexity of the algorithm.

3.4.2 Union

Where the sequence map join requires that only bindings of shared variables that are contained in both input sequence maps are retained, the sequence map union, denoted by $\ddot{\cup}()$, accepts bindings contained in either sequence map.

The union (and difference) of sequence maps is defined in the following only for single-variable or single-edge sequence maps. A sequence map $\overset{D\rightarrow Q}{\mathsf{SM}}$ over a relational structure D and a tree query Q is called *single-variable*, iff $|\operatorname{dom} SM| = 1$. It is called *single-edge*, iff $\operatorname{dom} \overset{D\rightarrow Q}{\mathsf{SM}} = \{v, v'\}$ and $\mathsf{edgeCover}(\overset{D\rightarrow Q}{\mathsf{SM}}) =$

$\{(v, v')\}$, i.e., if there is a single edge in the edge cover of $\overset{D \to Q}{\mathsf{SM}}$ and the only variables mapped by $\overset{D \to Q}{\mathsf{SM}}$ are that edge's source and sink. This restriction allows that union is well-defined and closed over such sequence maps and in accordance to the union over the induced relations of their input sequence maps.

Definition 3.7 (Sequence map union)**.** Let D be a relational structure, Q a tree query, and S_1, S_2 two single-variable sequence maps or two single-edge sequence maps for D over Q with $\operatorname{dom} S_1 = \operatorname{dom} S_2$ and the same associated edge cover (empty if single-variable, the same single edge if single-edge). Then $\ddot{\cup}(S_1, S_2)$ returns a (single-variable or single-edge) sequence map $\overset{D \to Q}{\mathsf{SM}}_3$ such that

(1) the induced relation $R_{\overset{D \to Q}{\mathsf{SM}}_3}$ of $\overset{D \to Q}{\mathsf{SM}}_3$ is $R_{\overset{D \to Q}{\mathsf{SM}}_3} = R_{S_1} \cup R_{S_2}$, i.e., the union of the induced relations of S_1 and S_2.

(2) $\overset{D \to Q}{\mathsf{SM}}_3|_{\operatorname{dom} S_1 \cup \operatorname{dom} S_2} = \overset{D \to Q}{\mathsf{SM}}_3$ (contains bindings only for variables mapped either in S_1 or in S_2).

The associated edge cover for $\ddot{\cup}(S_1, S_2)$ is the edge cover associated with S_1 and S_2, i.e., either empty if both input sequence maps are single-variable or the single edge in their edge cover.

The limitation to single-variable and single-edge sequence maps mirrors the restrictions for relational union (same schema for both relations), but is necessarily stronger, as discussed above.

Algorithm 10 gives an implementation of this operator. Notice, the similarity and differences to Algorithm 7: Though the main skeleton is similar, we actually retain bindings in each of the cases in lines 11–22, only failure markers are skipped (line 10). We also only record $\downarrow$ mappings from old indices to new ones, but no $\hookleftarrow$ and $\hookrightarrow$ mappings.

If both input maps are single-edge, the child variable is processed first (by means of line 5) and, for each index of any binding (except for failure markers) for both sequence maps a mapping to the new index is established. Note, that all interval sets are empty and thus line 9 has no effect. Second, the parent variable is processed using the change log for the child variable. Now, there are interval sets pointing to bindings of the child variable and these intervals are adapted to the new indices in line 9 by means of $\mathsf{RecreateInts}$. Algorithm 11 shows how $\mathsf{RecreateInts}$ is realized: For each variable covered by the passed interval sets (if called from Algorithm 10, there is always only a single such variable), we recreate the intervals. We can not simply adapt the intervals as in Adapt, Algorithm 8 (which is limited by the number of intervals in the input set), since within the range of an interval additional bindings (from the "opposite" sequence map) are introduce that are not actually mapped to that parent binding. This is impossible in the case of join and difference, where we only *restrict* but do not *extent* the bindings contained in each of the input sequence maps.

Depending on whether the input sequence maps are consistent (contain dangling bindings), we obtain the following result:

Theorem 3.10. *Algorithm 11 recreates a minimal set of non-overlapping interval pointers to bindings in a given sequence map S such that the index of each binding covered by an interval in that set is mapped by a given change log Log to an index covered in the set of intervals I_1 or the set of intervals I_2. It computes this set in $\mathcal{O}(b_V^{total} + |I_1| + |I_2|)$ where b_V^{total} is the total number of bindings for variables occurring in I_1 or I_2, if the input sequence maps contain no dangling bindings, $\mathcal{O}(b_V^{total} \cdot (|I_1| + |I_2|))$ otherwise.*

Proof. The set of generated intervals is minimal and non-overlapping as the next index after any end index is either out-of-bound (line 15) or not covered by any interval (line 12–14) and the previous index before any start index is either out-of-bound or not covered by any interval (an interval starts only in line 11 and only if start is ∞ which is only the case at the beginning of the loop, line 3, or if the preceding index

Algorithm 10: $\ddot{\cup}(S_1, S_2)$

input : Single-variable or single-edge Sequence maps S_1 and S_2 as in Definition 3.7

output: Sequence map res representing the union of the induced relations of the input maps

```
SharedVars $\leftarrow$ dom $S_1$(= dom $S_1 \cap$ dom $S_2$) ;
EC $\leftarrow$ edgeCover($S_1$)(= edgeCover($S_2$));
Log $\leftarrow \{(S_1 \mapsto \varnothing), (S_2 \mapsto \varnothing)\}$; res $\leftarrow \varnothing$;
while SharedVars $\neq \varnothing$ do
    $v \in \{v' \in$ SharedVars $: \nexists\, v'' \in$ SharedVars $: (v'', v') \in$ EC$\}$ ;
    $S \leftarrow \varnothing$; $i, j, k \leftarrow 1$;
    while $i \leq |S_1(v)|$ or $j \leq |S_2(v)|$ do
        // bindings returns $\infty$ for index out of bound
        $n_1 \leftarrow$ binding($S_1(v)[i]$); $n_2 \leftarrow$ binding($S_2(v)[j]$);
        // intervals returns $\varnothing$ for index out of bound
        $I \leftarrow$ RecreateInts($S$, intervals($S_1(v)[j]$), intervals($S_2(v)[j]$), Log);
        if $n_1 = \lightning$ then $i$++ else if $n_2 = \lightning$ then $j$++
        else if $n_1 = n_2$ then
            Log($S_1$) $\leftarrow$ Log($S_1$) $\cup \{(v, \downarrow, i, k)\}$;
            Log($S_2$) $\leftarrow$ Log($S_2$) $\cup \{(v, \downarrow, j, k)\}$;
            $i$++; $j$++; $k$++;
        else if $n_1 < n_2$ then
            $S[k] \leftarrow (n_1, I)$;
            Log($S_1$) $\leftarrow$ Log($S_1$) $\cup \{(v, \downarrow, i, k)\}$;
            $i$++; $k$++;
        else if then
            $S[k] \leftarrow (n_2, I)$;
            Log($S_2$) $\leftarrow$ Log($S_2$) $\cup \{(v, \downarrow, j, k)\}$;
            $j$++; $k$++;
    res $\leftarrow$ res $\cup \{(v, S)\}$;
    SharedVars $\leftarrow$ SharedVars $\setminus \{v\}$;
return res
```

Algorithm 11: RecreateInts(S, I_1, I_2, Log)

```
input  : Sequence map $S$, non-overlapping interval sets $I_1, I_2$, and change log Log
output: Set of intervals modified according to Log

NewInts ← ∅ ;
// For variables with entries in either interval set ...
foreach $v \in \pi_1(I_1) \cup \pi_1(I_2)$ do
    $i \leftarrow j \leftarrow k \leftarrow 1$; start ← ∞;
    for $k \leftarrow 1$ to $|S(v)|$ do
        $n \leftarrow S(v)[k]$;
        $(v, \downarrow, \text{idx}_1, k) \in$ Log or $\text{idx}_1 \leftarrow 0$;
        $(v, \downarrow, \text{idx}_2, k) \in$ Log or $\text{idx}_2 \leftarrow 0$;
        covered ← false;
        if FallsIn($\text{idx}_1, \pi_{2,3}(I_1)$) then
            $I_1 \leftarrow$ FallsIn($\text{idx}_1, \pi_{2,3}(I_1)$); covered ← true;
        if FallsIn($\text{idx}_2, \pi_{2,3}(I_2)$) then
            $I_2 \leftarrow$ FallsIn($\text{idx}_2, \pi_{2,3}(I_2)$); covered ← true;
        if covered = true and start = ∞ then start ← $k$;
        else if covered = false and start ≠ ∞ then
            NewInts ← NewInts ∪ $\{(v, \text{start}, k - 1)\}$;
            start ← ∞;
    // Collect remaining interval
    if start ≠ ∞ then NewInts ← NewInts ∪ $\{(v, \text{start}, |S(v)|)\}$;
return NewInts
```

Algorithm 12: FallsIn(index, I)

```
input  : Index idx and non-overlapping interval set $I$
output: The interval set $I'$ containing only those intervals from $I$ that cover or follow idx, or false if
        idx not covered by $I$

foreach $(s, e) \in I$ in increasing order on start index $s$ do
    if idx < $s$ then break ;
    if idx ≤ $e$ then return $I$ ;
    $I \leftarrow I \cup \{(s, e)\}$ ;
// idx is not covered by any interval in I
return false ;
```

is not covered, line 12–14). Thus, we can expand neither interval. Also all intervals are by construction non-overlapping.

For each index k that lies in an interval, there is an old index idx_1 (or idx_2) such that the old index lies in one of the intervals of I_1 (or I_2), line 9–11. Otherwise, it is not covered by an interval, line 12–14.

For the complexity, consider the loop 4–15 over the bindings for any variable in $\pi_1(I_1) \cup \pi_2(I_2)$. For each iteration, we call $\mathsf{FallsIn}$ which returns either the interval covering the index (and removes all previous intervals) or **false**.

If the underlying sequence map contains no dangling bindings each binding is covered by some index and the total run-time is bound by $\mathcal{O}(b_V^{total} + |I_1| + |I_2|)$ as each call to $\mathsf{FallsIn}$ runs in amortised constant time. If there are dangling bindings, however, for those bindings there is an associated old index, but this old index is not covered by any interval. Thus, in worst case, $\mathsf{FallsIn}$ runs in $|I|$ for each call. □

If both input maps are single-variable, we can actually eliminate the change log and all references to it. We can also eliminate line 9 entirely, though $\mathsf{RecreateInts}$ anyway returns immediately in this case as the interval sets are always empty.

With this observation the following corollary follows directly from Theorem 3.4 and Theorem 3.10:

Corollary 3.2. *Algorithm 10 computes $\ddot{\cup}(S_1, S_2)$ for consistent single-variable or single-edge sequence maps with shared domain Shared and edge cover in $\mathcal{O}(b_{\mathsf{Shared}} \cdot b_{\mathsf{Shared}}) \leq \mathcal{O}(|\mathsf{Shared}| \cdot n^2)$ time where b_{Shared} is the maximum number of bindings associated in either sequence map with a variable in Shared. If either input map is inconsistent $\mathcal{O}(b_{\mathsf{Shared}} \cdot b_{\mathsf{Shared}} + b_{\mathsf{Shared}}^{dangle} \cdot b_{\mathsf{Shared}}^{dangle} \cdot i) \leq \mathcal{O}(|\mathsf{Shared}| \cdot n^2 \cdot i)$ where $b_{\mathsf{Shared}}^{dangle}$ is the maximum number of dangling bindings associated in either sequence map with a variable in Shared, i is the maximum number of intervals associated with any such binding. For tree, forest, and CIG data $i = 1$, for arbitrary graph data $i \leq n$.*

There is no specific version that generates consistent sequence maps, as the above algorithm ensure that the resulting sequence map contains no failure markers if neither input sequence maps contained any (for shared variables, failure markers are actually removed due to $\mathsf{NextBinding}$, for others the existing bindings are copied in line 3–4). Furthermore, if neither input sequence maps contains dangling bindings, no such bindings are introduced (all interval pointers remain untouched as do the binding sequences they reference due to the second condition imposed on the input sequence maps in Definition 3.7.

3.4.3 Difference

The final combination operation for the sequence map removes bindings from a given sequence map if they are also contained in another sequence map. We pose much the same restrictions as on the sequence maps serving as input for the union operator.

Definition 3.8 (Sequence map difference). Let D be a relational structure, Q a tree query, and S_1, S_2 two single-variable sequence maps or two single-edge sequence maps for D over Q with $\operatorname{dom} S_1 = \operatorname{dom} S_2$ and the same associated edge cover (empty if single-variable, the same single edge if single-edge). Then $\dddot{\setminus}(S_1, S_2)$ returns a (single-variable or single-edge) sequence map $\overset{D\to Q}{\mathsf{SM}}_3$ such that

(1) the induced relation $R_{\overset{D\to Q}{\mathsf{SM}}_3}$ of $\overset{D\to Q}{\mathsf{SM}}_3$ is $R_{\overset{D\to Q}{\mathsf{SM}}_3} = R_{S_1} \setminus R_{S_2}$, i.e., the difference of the induced relations of S_1 and S_2.

(2) $\overset{D\to Q}{\mathsf{SM}}_3|_{\operatorname{dom} S_1 \cup \operatorname{dom} S_2} = \overset{D\to Q}{\mathsf{SM}}_3$ (contains bindings only for variables mapped either in S_1 or in S_2).

The associated edge cover for $\dddot{\setminus}(S_1, S_2)$ is the edge cover associated with S_1 and S_2, i.e., either empty if both input sequence maps are single-variable or the single edge in their edge cover.

$\dddot{\setminus}$ can be computed by an algorithm similar to the one for inconsistent sequence map join with the same time and space properties, cf. Algorithm 13.

It uses a helper function DifferenceInts that computes the difference between two given sets of intervals. Algorithm 13 computes a possibly inconsistent sequence map. Since the input is either single-variable or single-edge we refrain from defining a consistent version as it can be achieved with the same complexity by a $\dddot{\omega}_v^{\uparrow}$ followed by $\dddot{\omega}_{\downarrow}^{v}$ where v is either the single variable covered by the two sequence maps or the child variable in the edge cover.

Since we only consider single-edge sequence maps, the disadvantages of the join with overlapping edge sets (in contrast to the one with disjoint edge sets) are not exhibited by the sequence map difference.

Theorem 3.11. *Algorithm 14 computes a minimal, non-overlapping sequence of intervals representing the union of two sequences of intervals I_1 and I_2 in $\mathcal{O}(|I_1| + |I_2|)$ time and constant additional space.*

Proof. Algorithm 14 computes minimal, non-overlapping intervals by the same observation as for Algorithm 8: an interval is added only, if it is clear that the next binding after its end index is covered also in the second sequence map and thus should not be included in the result (line 7 and 11).

The result of Algorithm 14 is the difference of the intervals in the input sequences: if a binding is covered by an interval in I_1 but not in I_2, then it is either entirely before any interval in I_2 (lines 6–8) or it is partially overlapping (lines 10–13) in which case we split, if some prefix of it is not covered in I_2 (line 11) and continue with any suffix (if there is any).

The algorithm relies on the fact that the interval sets are ordered by increasing start index and non-overlapping. This allows us to infer that, if an interval from I_1 lies before the first (remaining) interval from I_2, then it lies before *all* intervals of I_2 and vice versa. Thus deciding overlapping becomes amortised constant rather than linear in the size of the intervals.

Algorithm 14 runs in $\mathcal{O}(|I_1| + |I_2|)$ as in each iteration of the loop 3–14 either i or j or both are incremented. Getting the next interval from I_1 and I_2 (line 4–5) is by assumption constant, as is adding an interval at the end of NexIntervals (lines 7, 11). □

3.5 Reduce

Given a single input map, we can perform several operations on that sequence map in isolation that drop certain variables (projection), certain bindings (selection), or propagate changes from bindings of one variable to those for another (propagate) removing any inconsistencies w.r.t. the propagated pair.

Sequence maps are closed under projection and propagation, but under selection only if we allow only conditions that refer only to one variable or to two variables adjacent in the edge cover of the sequence map.

3.5.1 Project

Projection is defined analogous to relational projection: We retain only bindings for variables specified in the projection, bindings for other variables are dropped (including interval pointers to those bindings). Formally, we define the sequence map projection as follows:

Algorithm 13: $\ddot{\setminus}(S_1, S_2)$

```
input  : Single-variable or single-edge Sequence maps $S_1$ and $S_2$ as in Definition 3.8
output: Sequence map res representing the difference of the induced relations of the input maps
SharedVars ← dom $S_1$ (= dom $S_1 \cap$ dom $S_2$); res ← $S_1$;
if |SharedVars| = 1 then
    // Single-variable sequence maps
    let $v$ be the single variable in SharedVars;
    res($v$) ← ∅;
    while $i \leq |S_1(v)|$ do
        // bindings returns ∞ for index out of bound
        $n_1$ ← binding($S_1(v)[i]$); $n_2$ ← binding($S_2(v)[j]$);
        if $n_1 = n_2$ then $i$++; $j$++;
        else if $n_1 < n_2$ then
            res($v$)[$k$] ← ($n_1, I$);
            $i$++; $k$++;
        else if $n_1 = \lightning$ then $i$++;
        else $j$++;
else
    // Single-edge sequence maps
    let $c$ be the child, $p$ the parent variable in SharedVars;
    $i, j, k$ ← 1;
    while $i \leq |S_1(p)|$ do
        // bindings returns ∞ for index out of bound
        $n_1$ ← binding($S_1(v)[i]$); $n_2$ ← binding($S_2(v)[j]$);
        // intervals returns ∅ for index out of bound
        if $n_1 = n_2$ then
            $I$ ← DifferenceInts($\pi_{2,3}$(intervals($S_1(v)[i]$)),
                                  $\pi_{2,3}$(intervals($S_2(v)[j]$)));
            if $I = \emptyset$ then $n_1 \leftarrow \lightning$ else res($v$)[$k$] ← ($n_1, I$); $k$++;
            $i$++; $j$++;
        else if $n_1 < n_2$ or $n_1 = \lightning$ then
            res($v$)[$k$] ← ($n_1, I$);
            $i$++; $k$++;
        else $j$++;
return res
```

Algorithm 14: DifferenceInts(Intervals$_1$, Intervals$_2$)

input : Two sequences Intervals$_1$, Intervals$_2$ of non-overlapping intervals (*without* associated variables) in order of start index

output: Minimal, non-overlapping sequence of intervals covering all indices contained in intervals of the first input sequence, but not in the second

```
NewIntervals ← ∅ ;
$i, j, k \leftarrow 1$;
while $i \le |\text{Intervals}_1|$ or $j \le |\text{Intervals}_2|$ do
    $(v, s_1, e_1) \leftarrow \text{Intervals}_1[i]$ or $\infty$ if $i > |\text{Intervals}_1|$;
    $(v, s_2, e_2) \leftarrow \text{Intervals}_2[i]$ or $\infty$ if $j > |\text{Intervals}_2|$;
    if $e_1 < s_2$ then                                   // $(s_1, e_1)$ before $(s_2, e_2)$
        $\text{NewIntervals}[k] \leftarrow (s_1, e_1)$; $k{+}{+}$;
        $i{+}{+}$;
    else if $e_2 < s_1$ then $j{+}{+}$ ;                  // $(s_2, e_2)$ before $(s_1, e_1)$
    else                                                  // $(s_1, e_1)$ overlaps $(s_2, e_2)$
        if $s_1 < s_2$ then $\text{NewIntervals}[k] \leftarrow (s_1, s_2 - 1)$; $k{+}{+}$;
        if $e_2 < e_1$ then $\text{Intervals}_1[i] \leftarrow (e_2 + 1, e_1)$;
        $j{+}{+}$;
        else $i{+}{+}$;

return NewIntervals ;
```

Definition 3.9 (Sequence map projection)**.** Let D be a relational structure, Q a tree query, S a sequence maps for D over Q, and $V \subset \operatorname{dom} S$ a sub-set of variables in Q such that for each pair $(v, v') \in V^2$ either v, v' have no least common ancestor, $\operatorname{lca}(v, v')$, in the edge cover of S or all variables on the path from $\operatorname{lca}(v, v')$ to v and v', respectively, are also in V.

Then, $\dddot{\pi}_V(S)$ returns a sequence map $\overset{D\to Q}{\mathsf{SM}}_3$ such that

(1) the induced relation $R_{\overset{D\to Q}{\mathsf{SM}}_3}$ of $\overset{D\to Q}{\mathsf{SM}}_3$ is $R_{\overset{D\to Q}{\mathsf{SM}}_3} = \pi_V(R_S)$, i.e., the projection to V of the induced relations of S.

(2) $\overset{D\to Q}{\mathsf{SM}}_3|_{\operatorname{dom} S} = \overset{D\to Q}{\mathsf{SM}}_3 (= \overset{D\to Q}{\mathsf{SM}}_3|_V)$ (contains bindings only for variables mapped in S).

The associated edge cover for $\dddot{\pi}_V(S)$ is $\mathsf{edgeCover}(S) \cap V^2$.

The restriction on the shape of V ensures, that there is no variable with parent in $\operatorname{dom} S \setminus V$ but ancestor in V w.r.t. the edge cover of S. In that case, the parent must be removed, however, it is needed to represent which bindings for the ancestor relate to which bindings of the descendant. Simply dropping the parent's bindings yields a cross product between the bindings of ancestor and descendant and thus more tuples than allows by the definition. The alternative to the restriction is that the resulting sequence map is no longer over D and Q but over a query Q' and relational structure D' such that **(1)** the variables of Q' are as in Q and the edges are $\mathsf{Edges}(Q) \setminus (V \times \operatorname{dom} S \cup \operatorname{dom} S \times V) \cup \mathsf{Edges}(Q) \cap V^2 \cup \mathcal{E}$ with $\mathcal{E}\{(v_1, v_k) \in V : \exists v_2, \dots, v_{k-1} \in \operatorname{dom} S \setminus V : (v_1, v_2), \dots, (v_{k-1}, v_k) \in \mathsf{Edges} Q\}$. **(2)** For each such edge $(v_1, v_k) \in \mathcal{E}$, there is a new relation name R in the relational schema for D' that is the label of that edge in Q' and relation instance $R^{D'}$ in D' such that $R^{D'} = \pi_{1,k}(R_1 \bowtie_{2=1} R_2 \bowtie_{2=1} \dots \bowtie_{2=1} R_{k-1})$ if R_i is the relation name of the edge between v_i and v_{i+1}. In the following, we only present the first approach.

The above definition allows both consistent and inconsistent sequence maps as result of $\dddot{\pi}$. However, Algorithm 15 ensures that, if the input sequence map is consistent, the result is consistent. If the input sequence map is inconsistent, the result is only consistent, if the inconsistencies are limited to bindings of variables not contained in the projection set.

Algorithm 15 gives a straightforward implementation for the sequence map projection: For all variables in the projection set $V \subset \operatorname{dom} S$ we either copy the bindings of the variable unchanged, if none of its children (in the edge cover of the input sequence map) are dropped, or copy them retaining only interval pointers to variables in V (line 7 $\sigma_{1\in V}$ selects those tuples from $\mathsf{intervals}(S(v)[i])$ that have a variable from V as first component).

Theorem 3.12. *Algorithm 15 computes $\dddot{\pi}_V(S)$ for a sequence map S and a projection set $V \subset \operatorname{dom} S$ in $\mathcal{O}(b^{total}_{\mathsf{DropParent}} \cdot i + |V|) \leq \mathcal{O}(q \cdot n \cdot i)$ time and constant additional space where i is the maximal number of intervals per binding and $b^{total}_{\mathsf{DropParent}}$ is the total number of bindings for variables with children in* $\operatorname{dom} S \setminus V$.

Proof. The resulting sequence map for Algorithm 15 retains bindings (line 3) and interval pointers (line 7) only for variables from V. For variables from V neither the bindings nor the interval pointers are touched. Together with the edge cover $\{(v, v') \in \mathsf{edgeCover}(S) : v, v' \in V\}$ as in Definition 3.9, this yields an induced relation that is the projection to V of the induced relation of the input sequence map.

The Algorithm loops over all variables in V and either copies the bindings unchanged (line 8) or touches each binding and modifies it by dropping the non-V interval pointers (line 4–7), thus touching each (of maximum i) intervals associated with that binding.

If the data is tree, forest, or CIG shaped, $i = 1$. □

Algorithm 15: $\ddot{\pi}_V(S)$

```
input  : Sequence map $S$ over a query $Q$ and set of variables $V \subset \text{dom}\, S$
output: Sequence map res representing the projection of $S$ to the variables in $V$
1 res ← ∅; // All variables with a covered edge to a dropped child
2 DropParent ← $\{v \in V : \exists v' \in \text{dom}\, S \setminus V : (v, v') \in \text{edgeCover}(S)\}$;
3 foreach $v \in V$ do
4     if $v \in$ DropParent then
5         res$(v) \leftarrow \emptyset$;
6         foreach $i \leftarrow 1$ to $|S(v)|$ do
7             res$(v)[i] \leftarrow (\text{binding}(S(v)[i]), \sigma_{1 \in V}(\text{intervals}(S(v)[i])))$;
8     else res$(v) \leftarrow S(v)$;
9 return res
```

Note, that each variable has a unique parent, if any, in a tree query. Thus, even if a CIQCAG expression for evaluating a given tree query drops all but one variable, whether in a single project operation, or in a sequence of $q - 1$ projections, at most $q - 1$ variables become "drop parents", i.e., variables with dropped child variable, over the whole expression and, thus, the overall complexity for the projections is also bounded by $\mathcal{O}(q \cdot n \cdot i)$.

Obviously, the result of a projection is a sequence map and $\ddot{\pi}_V$ can be applied to any sequence map S with $V \subset \text{dom}\, S$.

3.5.2 Select

Sequence map selection we also based on relational selection: We retain only bindings that fulfill a given selection condition. However, in contrast to relational selection, conditions may not relate multiple variables. Rather each condition may only reference a single variable from the query. This restriction is necessary to ensure the tree query property, where only relations between variables adjacent in the tree query are allowed. Consider, e.g., the sequence map S shown in Figure 22: A selection on the induced relation of S with a condition $v_1 = v_2$ (reading = as node identity) yields the relation $\{(d_1, d_3, d_1), (d_2, d_3, d_2)\}$. This relation can not be represented as a sequence map, since it can no longer be decomposed into binary relations over the pairs of adjacent variables without loss: only the d_1 (not the d_2) binding for v_1 remains related to the d_1 binding for v_2. However, both remain related to d_3 (as (d_2, d_3, d_2) remains in the relation). Thus, either we allow also (d_2, d_3, d_1) (and (d_1, d_3, d_2) by retaining an interval pointer 1–2 in d_3 or we drop all references from d_3 to bindings for v_3, thus yielding a sequence map representing the empty relation.

Thus we define the sequence map selection as follows:

Definition 3.10 (Sequence map selection). Let D be a relational structure, Q a tree query, S a sequence maps for D over Q, and c a single atom containing (possibly multiple) references to a *single variable*. Then $\ddot{\sigma}^{\lightning}_c(S)$ returns a sequence map $\overset{D \to Q}{\mathbf{SM}}_3$ such that

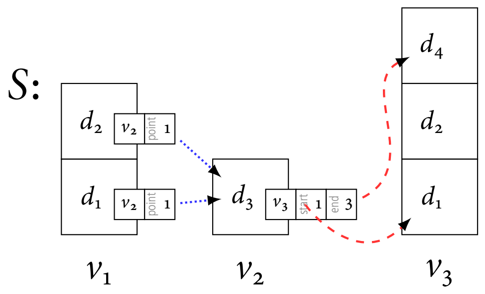

Figure 22. Sequence maps for illustrating the single-variable condition restriction for $\ddot{\sigma}$

(1) the induced relation $R_{\overset{D\to Q}{\mathsf{SM}}_3}$ of $\overset{D\to Q}{\mathsf{SM}}_3$ is $R_{\overset{D\to Q}{\mathsf{SM}}_3} = \sigma_c(R_S)$, i.e., the selection on c over the induced relations of S.

(2) $\overset{D\to Q}{\mathsf{SM}}_3|_{\operatorname{dom} S} = \overset{D\to Q}{\mathsf{SM}}_3$ (contains bindings only for variables mapped in S).

The associated edge cover for $\ddot{\sigma}^{\lightning}_V(S)$ is the same as for S.

We allow in the definition only a single atom, since conjunctions, disjunctions, and quantification can be achieved by combining selection with union, join, or difference. As a convenience, we can easily extend c to conjunctions and/or disjunctions of atoms, all over the same single variable, without changes to the algorithms and results.

Algorithm 16 gives a straightforward implementation for the inconsistent case: Here, we merely "bomb" all bindings where c does not hold if we replace v with its binding in c (line 3). Since the number and order of bindings is untouched, the interval pointers can remain unchanged (but now possibly point to some or all failure markers).

Algorithm 16: $\ddot{\sigma}^{\lightning}_c(S)$

```
input  : Sequence map $S$ over a query $Q$ and condition $c$ on variable $v \in \operatorname{dom} S$
output: Possibly inconsistent sequence map res representing the selection of only bindings for $v$
        from $S$ matching $c$
1 res ← $S$;
2 foreach $i \leftarrow 1$ to $|\mathsf{res}(v)|$ do
3     if $\neg c\{v/\mathsf{res}(v)[i]\}$ then res$(v)[i] \leftarrow \lightning$;
4 return res
```

Theorem 3.13. *Algorithm 16 computes $\ddot{\sigma}^{\lightning}_V(S)$ for a sequence map S and a condition c over a single variable v in $\mathcal{O}(b_v \cdot m_c) \leq \mathcal{O}(n \cdot m_c)$ time and constant additional space where m_c is the maximum time for evaluating c if all references to v are replaced by a binding for v in S and b_v the number of bindings for v in S. For most conditions, m_c is constant.*

Proof. Algorithm 16 computes $\ddot{\sigma}^{\lightning}_V(S)$: No v binding for that c does not hold is retained (line 2–3). All bindings for other variables are untouched (line 1).

It is obviously bounded by the number of bindings for v in S (line 2), for each of which c is evaluated in line 3. $\square$

Definition 3.10 allows the input to be inconsistent and a inconsistent sequence maps as result of $\dddot{\sigma}^{\lightning}$. As for most other operators, we can drop this restrictions and require consistent sequence maps as input and result. We denote the resulting operation with $\dddot{\sigma}$. The required changes are similar to the case of the sequence map join (cf. Section 3.4.1): We introduce a change log for the changes applied to the variable v involved in c and use this change log to adapt the intervals of the parent variable of v. Since we may also drop bindings for the parent variable (where all related bindings for v are dropped), we need to propagate such changes upwards in the query to all ancestors to v. The corresponding algorithm is given in Algorithm 17 and shows roughly the same relation to the above Algorithm for $\dddot{\sigma}^{\lightning}$ as Algorithm 7 to Algorithm 5.

Theorem 3.14. *Algorithm 17 computes $\dddot{\sigma}_V(S)$ for a consistent sequence map S and a condition c over a single variable v in $\mathcal{O}(b_{\mathcal{V}}^{total} \cdot (\max(i, m_c)) \leq \mathcal{O}(q \cdot n \cdot \max(i, m_c))$ time and $\mathcal{O}(n)$ additional space where m_c is the maximum time for evaluating c if all references to v are replaced by a binding for v in S, i the maximum number of associated intervals for a binding in S, and $b_{\mathcal{V}}^{total}$ the total number of bindings for all variables in $\mathcal{V} = \{v' \in \operatorname{dom} S : \exists$ path from v' to v in $\mathsf{edgeCover}(S)\}$.*

3.5.3 Propagate

As discussed above, in particular for the sequence map join in Section 3.4.1, many of the previous operators are easier to implement with better complexity if we allow the results to be inconsistent. Though this advantage may, to some extent, be offset by the increase in result size, if the involved operations are selective, we can still profit in many cases from inconsistent sequence maps: Instead of propagating changes from a often small subset of variables involved in an operation to all variables after each operation, we propagate these changes once at the end of an entire sequence of operations. See Section **??** for a more detailed discussion of when to employ inconsistent sequence maps.

For some operations, however, a *consistent* sequence map is either required or beneficial. This is particularly true for $\digamma$, which extracts bindings for some variables from a sequence map. Consider, e.g., that we are interested only in the bindings of a single variable v. If the given sequence S map is consistent, it suffices to return the bindings in $S(v)$, without even considering the other variables or the relation of them to v. This is possible, since every binding $n \in S(v)$ occurs in at least one tuple of the induced relation of S (i.e., can be extended to a full answer if S is complete). If, however, S may be inconsistent, a binding $n \in S(v)$ (or, for that matter, its parents or their parents …) may be dangling in which case there is no binding for v's parent and n does *not* contribute to a tuple in the induced relation. If n is a failure marker, it does not contribute to the induced relation by definition and, worse, bindings for v's parent may only be related to failure markers among the binding for v and thus, in fact, also not contribute to the induced relation.

To enable this mode of processing, we require explicit propagation operators, that, progressively, ensure that a resulting sequence map is consistent. For the two types of inconsistency, two different operators are provided:

(1) $\dddot{\omega}_v^{\uparrow}(S)$ removes all failure markers from the bindings of v and, if there is a parent v' of v covered by S, adopts the interval pointers of v accordingly. This may lead to some bindings of v' without related

Algorithm 17: $\ddot{\sigma}_c(S)$

```
input  : Consistent sequence map $S$ over a query $Q$ and condition $c$ on variable $v \in \mathrm{dom}\, S$
output: Consistent sequence map res representing the selection of only bindings for $v$ from $S$ for
        which $c$ holds
1  EC ← edgeCover($S$);
2  res ← $S$;
   // Process the selection variable
3  res($v$) ← ∅; $k$ ← 1; Log ← ∅; last ← ∞;
4  for $i$ ← 1 to $|S(v)|$ do
5      if $c\{v/S(v)[i]\}$ then
6          res($v$)[$k$] ← $S(v)[i]$;
7          Log ← Log ∪ $\{(v, \downarrow, i, k)\}$ ∪ $\{(v, \hookrightarrow, l, i)$ : last ≤ $l < i\}$;
8          $k$++; last ← ∞;
9      else
10         last ← min(last, $i$); Log ← Log ∪ $\{(v, \hookleftarrow, i, \text{last} - 1)\}$;
11 if last ≠ ∞ then
12     Log ← Log ∪ $\{(v, \hookrightarrow, l, \infty)$ : last ≤ $l \leq |S(v)|\}$
   // Process the ancestors of the selection variable
13 Ancestors ← $\{v' \in \mathrm{dom}\, S : \exists$ path from $v'$ to $v$ in EC$\}$
14 foreach $v'$ ∈ Ancestors in topological order w.r.t. EC do
15     NoDrop ← true;
16     res($v'$) ← ∅; $k$ ← 1; last ← ∞;
17     for $i$ ← 1 to $|S(v')|$ do
18         $I$ ← Adapt(intervals($S(v')[i]$), Log);
19         if $I = \emptyset$ then
20             last ← min(last, $i$);
21             Log ← Log ∪ $\{(v', \hookleftarrow, i, \text{last} - 1)\}$;
22             NoDrop ← false;
23         else
24             res($v'$)[$i$] ← (binding($S(v')[i]$), $I$);
25             Log ← Log ∪ $\{(v', \downarrow, i, k)\}$ ∪ $\{(v, \hookrightarrow, l, i)$ : last ≤ $l < i\}$;
26             $k$++; last ← ∞;
27     if NoDrop then break ;
28     if last ≠ ∞ then
29         Log ← Log ∪ $\{(v', \hookrightarrow, l, \infty)$ : last ≤ $l \leq |S(v)|\}$
30 return res
```

bindings for v. These bindings can no longer contribute to a tuple in the induced relation of S and are thus "bombed". This way we move the consistency up one level in the (tree) query, from v to its unique parent v'. Since v' is the only variable whose bindings have interval pointers to bindings of v, no other variable is directly affected. However, to remove all failure markers from S, we need to perform $\dddot{\omega}^{\uparrow}_{v}(S)$ for all variables covered by S in inverse topological order (i.e., child before parent).

(2) $\dddot{\omega}^{v}_{\downarrow}(S)$ discovers and removes all "dangling" bindings of v. Note, that this may affect the interval pointers from $v' = \mathsf{parent}(v)$ to v: They may "slide" to the left, as "dangling" bindings for v are removed from the beginning of the binding sequence $S(v)$, and merge with the previous interval(s), if all the intermediary bindings are dangling. The number of related bindings of v, however, remains unchanged for each binding of v'. Therefore, the sequence of bindings for v' remains the same, only the interval pointers change. To remove all "dangling" bindings from S, we need to perform $\dddot{\omega}^{v}_{\downarrow}$ for all non-root variables covered by S in topological order (i.e., parent before child).

If we perform first $\dddot{\omega}^{\uparrow}_{v}(S)$ for all variables by S in inverse topological order, then $\dddot{\omega}^{v}_{\downarrow}(S)$ in the inverse order, the resulting sequence map is consistent, see Theorem 3.17.

Formally, we define the up-propagation operator as follows:

Definition 3.11 (Sequence map propagation)**.** Let D be a relational structure, Q a tree query, S a sequence maps for D over Q, and $v \in \operatorname{dom} S$ a variable covered by S. Then $\dddot{\omega}^{\uparrow}_{v}(S)$ ($\dddot{\omega}^{v}_{\downarrow}(S)$) returns a sequence map $\overset{D\to Q}{\mathsf{SM}}_3$ such that:

(1) the induced relation $R_{\overset{D\to Q}{\mathsf{SM}}_3}$ of $\overset{D\to Q}{\mathsf{SM}}_3$ is unchanged, i.e., $R_{\overset{D\to Q}{\mathsf{SM}}_3} = R_S$.

(2) $\overset{D\to Q}{\mathsf{SM}}_3|_{\operatorname{dom} S} = \overset{D\to Q}{\mathsf{SM}}_3$ (contains bindings only for variables mapped in S).

For $\dddot{\omega}^{v}_{\downarrow}(S)$ in addition to 1 and 2 it also holds that

(3a) there are no failure markers in $\overset{D\to Q}{\mathsf{SM}}_3(v)$ (among the bindings for v).

For $\dddot{\omega}^{v}_{\downarrow}(S)$ in addition to 1 and 2 it also holds that

(3b) there are no direct dangling bindings in $\overset{D\to Q}{\mathsf{SM}}_3(v)$, i.e., no bindings of v that are not covered by some interval pointer of a binding for v's parent in the edge cover of S, if there is such a parent.

The associated edge cover for $\dddot{\omega}^{\uparrow}_{v}(S)$ ($\dddot{\omega}^{v}_{\downarrow}(S)$) is the edge cover of the input sequence map.

Note that, for of $\dddot{\omega}^{v}_{\downarrow}(S)$, if v is a root node in Q or $v' = \mathsf{parent}(v)$ in Q but $(v', v) \notin \mathsf{edgeCover}(S)$, the above condition (3b) implies that no binding of v is dangling.

We require that w.r.t. v (and only v) the result of $\dddot{\omega}^{\uparrow}_{v}(S)$ ($\dddot{\omega}^{v}_{\downarrow}(S)$) is consistent wrt. failure markers (direct dangling bindings) even if S contains failure markers (direct dangling bindings) for v. However, for all other variables there may still be (direct) dangling bindings or failure markers. In fact, to remove failure markers from the bindings of v we need, in general, to adapt the bindings of any variable with an edge to v in the edge cover. Since Q is a tree query, there is a single such variable, v'. When we adapt the interval pointers referring from bindings of v' to bindings of v, we may end up with bindings for v' with no corresponding binding for v. Those bindings are then "bombed". Thus, we may actually *introduce* failure markers for v'. Those failure markers may, e.g., be propagated and removed by a later $\dddot{\omega}^{\uparrow}_{v'}(S')$ where S' is the result of $\dddot{\omega}^{\uparrow}_{v}(S)$.

Algorithm 18 shows in detail, how to compute $\dddot{\omega}^{\uparrow}$. It is, basically, the "consistency" component of Algorithm 7 for a single variable. It is divided in two phases, one for v and one for its parent, if there is any: **(1)** In lines 2–9, we remove all failure markers from $S(v)$ and store the result in S'. This yields a change log mapping indices for bindings in $S(v)$ to indices in S'. The change log is shaped exactly the same as the change log for Algorithm 7: For each retained binding a ↓ entry associating the old index to the index in S'. For each dropped binding (i.e., failure marker), a $\hookleftarrow$ and a $\hookrightarrow$ entry referencing the index of the last preceding and next following retained binding. **(2)** This change log is used in the second phase, lines 10–16, to adapt each of the interval pointers from bindings for v' to bindings for v. At the end of phase 2, all intervals reference now indices in S' rather than in $S(v)$. Note, that, if all interval pointers of a binding are dropped (because they pointed only to failure markers), that binding is itself "bomb"ed (line 13).

Algorithm 18: $\ddot{\omega}^{\updownarrow}_{v}(S)$

```
input : Sequence map $S$ over a query $Q$ and variable $v \in \mathrm{dom}\, S$
output: Sequence map res representing the induced relation for $S$ with no failure markers among
        the bindings of $v$

1  $\textsc{ec} \leftarrow \mathsf{edgeCover}(S)$; $\mathsf{res} \leftarrow S$;
   // 1—Child Phase: For v, remove failure markers
2  $S' \leftarrow \varnothing$; $k \leftarrow 1$; $\mathsf{Log} \leftarrow \varnothing$; $\mathsf{last} \leftarrow \infty$;
3  for $i \leftarrow 1$ to $|S(v)|$ do
4      if $S(v) = \lightning$ then $\mathsf{last} \leftarrow \min(\mathsf{last}, i)$; $\mathsf{Log} \leftarrow \mathsf{Log} \cup \{(v, \hookleftarrow, i, \mathsf{last} - 1)\}$;
5      else
6          $S'[k] \leftarrow S(v)[i]$;
7          $\mathsf{Log} \leftarrow \mathsf{Log} \cup \{(v, \downarrow, i, k)\} \cup \{(v, \hookrightarrow, l, i) : \mathsf{last} \le l < i\}$;
8          $k{+}{+}$; $\mathsf{last} \leftarrow \infty$;
9  if $\mathsf{last} \neq \infty$ then $\mathsf{Log} \leftarrow \mathsf{Log} \cup \{(v, \hookrightarrow, l, \infty) : \mathsf{last} \le l \le |S(v)|\}$
   // 2—Parent Phase: For parent v', shrink intervals
10 if $(v', v) \in \textsc{ec}$ for a $v' \in \mathrm{dom}\, S$ then
11     take that $v'$ (there is at most one);
12     $\mathsf{res}(v') \leftarrow \varnothing$;
13     for $i \leftarrow 1$ to $|S(v')|$ do
14         $I \leftarrow \mathtt{Adapt}(\mathsf{intervals}(S(v')[i]), \mathsf{Log})$;
15         if $I = \varnothing$ then $\mathsf{res}(v')[i] \leftarrow \lightning$;        // We introduce a failure marker!
16         else $\mathsf{res}(v')[i] \leftarrow (\mathsf{binding}(S(v')[i]), I)$;
17 return res
```

Theorem 3.15. *Algorithm 18 computes $\ddot{\omega}^{\updownarrow}_{v}(S)$ for a sequence map S and variable $v \in \mathrm{dom}\, S$ $\mathcal{O}(b_v + b_{v'} \cdot i) \le \mathcal{O}(n \cdot i)$ time and $\mathcal{O}(b_v) \le \mathcal{O}(n)$ additional space where i is the maximum number of associated intervals pointing to v for any binding in S, and b_v ($b_{v'}$) the number of bindings for v (v' if $v' \in \mathrm{dom}\, S$ with $(v', v) \in \mathsf{edgeCover}(S)$ or 0 otherwise) in S.*

Proof. Algorithm 18 computes a sequence map S' such that $S'(v)$ contains no failure markers. It contains no failure markers due to line 4.

S' has the same induced relation as the input sequence map, as only failure markers are dropped which do not contribute to a tuple contained in the induced relation by Definition 2.4.

Phase 1 of the algorithm runs in $\mathcal{O}(b_v)$ time as it iterates over all bindings for v in S. For each such binding, the entry is copied to the new sequence map and the change log is updated (in constant time).

Phase 2 runs in constant time if there is no parent in EC, otherwise in $\mathcal{O}(b_{v'} \cdot i)$ time, as it iterate over the bindings for v' in S. For each such binding, they call Adapt which runes, by Theorem 3.5 in time i where i is the maximum number of intervals pointing to v associated with a binding of v' (which are the only bindings pointing to v). $\square$

For $\dddot{\omega}_{\downarrow}$, we obtain a similar algorithm 19 with three phases: In the first phase, the *skip index* for the parent v' of v (if there is any) is created: It contains for each index i of a binding of v the maximum end index of any interval pointer starting at i and associated with a binding of v'. The second phase uses the skip index find all dangling bindings in a single pass over the bindings of v. In a third phase, the interval pointers from v' to v are slided (and, possibly, merged) to adapt to the new sequence of bindings for v. In detail: **(1)** In the first parent phase, we build a skip index Skip that initially points to 0 for all indices in S'. Whenever we find an (adapted, i.e., already pointing to S') interval that starts at an index s, we set the skip index for s to the maximum of its current value and the end index e of the interval.

At the end of phase 2, the skip index records, for each index i, the maximum end index of any interval that starts at i (or 0 if there is none). **(2)** The skip index is used in the second phase, lines 17–25, for a second pass over the bindings of the child variable. Here, we consider for each binding, if it lies within a interval of one of the bindings for the parent variable. This is recognized using the skip index: maxEnd always points to the highest end point of any interval whose start index we have already passed. If maxEnd is smaller than the current index and the current index is not a start index ($\mathsf{Skip}(i) = 0$), we have found a dangling binding that must be eliminated. Otherwise, we copy the binding and update maxEnd (with the maximum end index of an interval that starts at the current index, if it is larger than maxEnd). **(3)** Finally, in a third phase we slide the intervals of all bindings for the parent variable to adapt to the changes to the sequence of bindings for the child variable in the previous step. Note, that this step does not introduce any new dangling bindings, as intervals do not shrink in this phase, but only slide since the bindings associated with start and end indices can not have been dropped in the previous phase (they are, after all, covered by at least the current interval). Thus, there are $\downarrow$ entries in the change log for them (line 24) and the interval is simply slided according to those entries and possibly collapsed with a previous interval (if the only bindings between the two intervals have been dangling bindings and thus dropped in the previous phase.

Theorem 3.16. *Algorithm 19 computes $\dddot{\omega}^{v}_{\downarrow}(S)$ for a sequence map S and variable $v \in \operatorname{dom} S$ $\mathcal{O}(b_v + b_{v'} \cdot i) \leq \mathcal{O}(n \cdot i)$ time and $\mathcal{O}(b_v) \leq \mathcal{O}(n)$ additional space where i is the maximum number of associated intervals pointing to v for any binding in S, and b_v ($b_{v'}$) the number of bindings for v (v' if $v' \in \operatorname{dom} S$ with $(v', v) \in \mathit{edgeCover}(S)$ or 0 otherwise) in S.*

Proof. Algorithm 19 computes a sequence map S' such that $S'(v)$ contains no direct dangling bindings. Due to lines 17–25, it contains no direct dangling binding, since, for each direct dangling binding n with index i, $\mathsf{Skip}(i) = 0$ (there can be no interval with i as start index otherwise n is not dangling) and there is no previous index $j \leq i$ that is the start index of an interval that cover i (i.e., with end index $\geq i$). Thus maxEnd $< i$, all conditions of line 19 are fulfilled, and the binding is dropped.

S' has the same induced relation as the input sequence map, as only dangling bindings are dropped which do not contribute to a tuple contained in the induced relation by Definition 2.4.

Algorithm 19: $\dddot{\omega}^{v}_{\downarrow}(S)$

```
input  : Sequence map $S$ over a query $Q$ and variable $v \in \mathrm{dom}\, S$
output: Sequence map res representing the induced relation for $S$ with no direct dangling bindings
        among the bindings of $v$

EC ← edgeCover($S$); res ← $S$;
if $(v', v) \in$ EC then
    take that unique $v'$;
    // 1—Parent Phase: Compute skip index
    Skip ← $\{(i, 0) : 1 \le i \le |S'|\}$; res($v'$) ← $\varnothing$;
    foreach $(n, I) \in S(v')$ do
        foreach $(s, e) \in I$ do Skip($s$) ← max(Skip($s$), $e$);

    // 2—Child Phase: For v, drop dangling bindings
    // maxEnd maximum end point of any “open” interval
    maxEnd ← 0; res($v$) ← $\varnothing$; $k$ ← 1; Log ← $\varnothing$; last ← $\infty$; $i$ ← 1;
    for $i$ ← 1 to $|S'|$ do
        if maxEnd < $i$ and Skip($i$) = 0 then
            last ← min(last, $i$); maxEnd ← 0;
        else
            res($v$)[$k$] ← $S(v)[i]$;
            Log ← Log ∪ $\{(v, \downarrow, i, k)\}$;
            $k$++; last ← $\infty$; maxEnd ← max(maxEnd, Skip($i$));

    // 3—Parent Phase: For v′, slide intervals
    for $i$ ← 1 to |res($v'$)| do
        if res($v'$) = ↯ then continue ;
        $I$ ← Adapt(intervals(res($v'$)[$i$]), Log);
        res($v'$)[$i$] ← (binding(res($v'$)[$i$]), $I$);
return res
```

Phase 1 and 3 of the algorithm run in $\mathcal{O}(b_{v'} \cdot i)$ time, as they both iterate over the bindings for v' in S. For each such binding, phase 1 iterates over all its i intervals and updates the skip index (single comparison and assignment). Phase 3 calls Adapt which runs, by Theorem 3.5, in time i where i is the maximum number of intervals pointing to v associated with a binding of v'.

Phase 2 of the algorithm runs in $\mathcal{O}(b_v)$ time as they both iterate at most over all bindings for v in S (phase 3 already skips failure markers). For each such binding, the entry is copied to the new sequence map and the change log is updated (in constant time). □

From these results, we can immediately conclude the following result:

Theorem 3.17. *Let $\overset{D\to Q}{\mathsf{SM}}$ be an inconsistent sequence map. Then there is a consistent sequence map $\overset{D\to Q}{\mathsf{SM}'}$ equivalent to $\overset{D\to Q}{\mathsf{SM}}$. This sequence map can be computed in $\mathcal{O}(\tilde{q} \cdot n \cdot i)$ where $\tilde{q} = |\mathrm{dom}\,\overset{D\to Q}{\mathsf{SM}}|$, $n = |\mathsf{Nodes}(D)|$, and i the maximum number of intervals per binding in S. For tree, forest, and* CIG *data $i = 1$.*

Proof. To compute the consistent sequence map S' from the inconsistent input sequence map S, we use a CIQCAG-expression using $\ddot{\omega}^{\uparrow}$ and $\ddot{\omega}_{\downarrow}$. Let $\mathrm{dom}\, S = \{v_1, \dots, v_k\}$ such that, for any v_i, v_j with $v_i < v_j$ wrt. the topological order on $\mathrm{dom}\, S$ induced by $\mathsf{edgeCover}(S)$ (i.e., $v < v'$ if $\exists$ path from v' to v in $\mathsf{edgeCover}(S)$), it holds that $i < j$. Then the following CIQCAG-expression computes a consistent sequence map that is equivalent to S:

$$\dddot{\omega}^{v_1}_{\downarrow}(\dddot{\omega}^{v_2}_{\downarrow}(\dots \dddot{\omega}^{v_k}_{\downarrow}(\ddot{\omega}^{\uparrow}_{v_k}(\ddot{\omega}^{\uparrow}_{v_{k-1}}(\dots(\ddot{\omega}^{\uparrow}_{v_1}(S)))))))$$

The resulting sequence map is consistent, as it contains: (1) no failure markers for any v_i, as there is a $\ddot{\omega}^{\uparrow}_{v_i}(\mathcal{E}_i)$ for each v_i and only $\ddot{\omega}^{\uparrow}_{v_j}(\mathcal{E}_j)$ with $j < i$ may create failure markers for v_i (viz. $\ddot{\omega}^{\uparrow}_{v}(\mathcal{E}_v)$ for all children v of v_i), all of which are contained in $\mathcal{E}_i$. (2) no direct dangling bindings for any v_i, as there is a $\dddot{\omega}^{v_i}_{\downarrow}(\mathcal{E}_i)$ only $\ddot{\omega}^{\uparrow}$'s and $\dddot{\omega}^{v_j}_{\downarrow}(\mathcal{E}_j)$ with $j > i$ influence the bindings of the parent of v_i and all these are contained in $\mathcal{E}$. (3) no indirect dangling bindings as there are no direct dangling bindings.

The CIQCAG-expression has a size of $2 \cdot |\mathrm{dom}\, S|$ and consists only of $\ddot{\omega}_{\downarrow}$ and $\ddot{\omega}^{\uparrow}$ expressions, all of which operate on S or a sequence map smaller than S. Thus, all of them are bound by $\mathcal{O}(n \cdot i)$ where i is the maximum number of intervals per binding in S.

Overall, the computation is thus bound by $\mathcal{O}(|\mathrm{dom}\, S| \cdot n \cdot i)$. □

In the rest of this work, we use often sequences $v_1, \dots, v_n$ of variables instead of a single variable as index for $\ddot{\omega}^{\uparrow}$ ($\ddot{\omega}_{\downarrow}$). This notation is a shorthand for a sequence of n nested $\ddot{\omega}^{\uparrow}$ ($\ddot{\omega}_{\downarrow}$) expressions each for one variable in the order of the variables in the index sequence.

3.6 Rename

For convenience, we briefly discuss an analog to the rename operator on (named) relational algebra.

Definition 3.12 (Sequence map renaming). Let D be a relational structure, Q a tree query, S a sequence maps for D over Q, and $v_1 \in \mathrm{dom}\, S, v_2 \in \mathsf{Vars}(Q) \setminus \mathrm{dom}\, S$. Then $\ddot{\rho}_{v_1 \to v_2}(S)$ returns a sequence map $\overset{D\to Q}{\mathsf{SM}'}$ such that

(1) the induced relation $R_{\overset{D\to Q}{\mathsf{SM}'}}$ of $\overset{D\to Q}{\mathsf{SM}'}$ is $R_{\overset{D\to Q}{\mathsf{SM}'}} = \rho_{v_1 \to v_2}(R_S)$, i.e., the induced relation of the input sequence map with attribute v_1 renamed to v_2.

(2) $\overset{D\to Q}{\mathsf{SM}'}|_{\mathrm{dom}\,S} = \overset{D\to Q}{\mathsf{SM}'}$ (contains bindings only for variables mapped in S).
The associated edge cover for $\dddot{\rho}_{v_1 \to v_2}(S)$ is the edge cover of the input sequence map with all occurrences for v_1 replaced by v_2.

Algorithm 20: $\dddot{\rho}_{v_1 \to v_2}(S)$

```
input  : Sequence map $S$ over a query $Q$ two variables $v_1 \in \mathrm{dom}\, S$, $v_2 \in \mathsf{Vars}(Q) \setminus \mathrm{dom}\, S$
output: Sequence map res representing the induced relation of $S$ with $v_1$ renamed to $v_2$

res ← $S$;
// All variables with a covered edge to $v_1$
ParentVars ← $\{v \in \mathrm{dom}\, S : (v, v_1) \in \mathsf{edgeCover}(S)\}$;
foreach $v \in$ ParentVars do
    res($v$) ← $\varnothing$;
    foreach $i \leftarrow 1$ to $|S(v)|$ do
        $I \leftarrow \sigma_{1 \neq v_1}(\mathsf{intervals}(S(v)[i])) \cup \{v_2\} \times \pi_{2,3}(\sigma_{1=v_1}(\mathsf{intervals}(S(v)[i])))$;
        res($v$)[$i$] ← ($\mathsf{binding}(S(v)[i]), I$);
res($v_2$) ← res($v_1$);
res ← res $\setminus \{(v_1, \mathsf{res}(v_1))\}$;
return res
```

Note, that $\dddot{\rho}_{v_1 \to v_2}$ can be applied only, if v_2 is not covered by S and if v_2 has at least the same in- and outgoing edges in Q as v_1 has in the edge cover of S. This limits the applicability compared to the relational case.

Algorithm 20 shows how to compute the sequence map rename operation. Note, that only bindings for the parent of v_1, if there is any in the edge cover of S, are processed.

Theorem 3.18. *Algorithm 20 computes $\dddot{\rho}_{v_1 \to v_2}(S)$ for a sequence map S and variables and $v_1 \in \mathrm{dom}\, S, v_2 \in \mathsf{Vars}(Q) \setminus \mathrm{dom}\, S$ in $\mathcal{O}(b_{\mathsf{parent}(v_1)} \cdot i) \leq \mathcal{O}(n \cdot i)$ time and constant additional space where $b_{\mathsf{parent}(v_1)}$ is the number of bindings for the parent of v_1 in S and i the maximum number of associated intervals associated with such a binding.*

3.7 Back to Relations: Extract

Fittingly, we conclude the introduction and definition of the sequence map operators in CIQCAG with the sequence map extraction, where we obtain the variable bindings for a subset V of a sequence maps variables as a relation. A tuple $t = \langle v_1 : n_1, \ldots, v_k : n_k \rangle$ is contained in that relation, if $V = \{v_1, \ldots, v_k\}$ and t is in the projection to V of the induced relation of the sequence map S. Thus, for each $v_i : n_i$ there is an index l such that $\mathsf{binding}(S(v_i)[l]) = n_i$ and, if $(v, v') \in \mathsf{edgeCover}(I)$, then $v : n \in t$ and $v' : n' \in t$ if there are indices l, l' such that $S(v)[l] = (n, I)$, $\mathsf{binding}(S(v')[l']) = n'$ and l' is covered by some interval in I. If the data is tree, forest, or CIG shaped, $|I| \leq 1$ and l' must lie within the boundaries of the single interval in I.

Formally, we define the sequence map extract operator as follows:

Definition 3.13 (Sequence map extraction). Let D be a relational structure, Q a tree query, S a sequence maps for D over Q with the induced relation R_S, and $V = \{v_1, \dots, v_k\} \subset \operatorname{dom} S$. Then $\digamma_V(S)$ returns a relation $R = \pi_V(R_S)$.

By definition, $\digamma_{\operatorname{dom} S} S$ returns exactly the induced relation of S, yielding an algorithm for computing the induced relation.

Algorithm 21: $\digamma_V(S)$

```
input  : Sequence map $S$ and variables $V = \{v_1, \dots, v_k\} \subset \operatorname{dom} S$
output: $\pi_{v_1,\dots,v_k}(R_S)$
1 res ← {};
2 RootVars ← $\{v \in \operatorname{dom} S : \nexists v' : (v, v') \in \text{edgeCover}(S)\}$;
3 foreach $v \in$ RootVars do
4     res ← res × Relation($S, v, 1, |S(v)|$);
5 return $\pi_{v_1,\dots,v_k}(\text{res})$ or $\emptyset$ if res = $\{\langle\rangle\}$
```

First, we present a naive version of $\digamma$ that *always* computes the induced relation and simply applies a relational projection to the computed relation. It is presented in Algorithm 21 and uses Algorithm 22 to compute the induced relation for each connected component.

Theorem 3.19. *Algorithm 21 computes $\digamma_V(S)$ for a sequence map S and a set of variables $V \subset \operatorname{dom} S$ in $\mathcal{O}(b_+^{|\operatorname{dom} S|}) \leq \mathcal{O}(n^q)$ where b_+ is the maximum number of "good" bindings (neither failure markers nor dangling) for a variable in S.*

Proof. The result of Algorithm 21 is the projection to V of the induced relation of S: Relation computes the induced relation for each connected component of the edge cover EC of S rooted at one of the root vars. This is extended to the full induced relation in line 4 (note that there are no covered edges between variables from different connected components and thus a mere cross product suffices.

Algorithm 22 computes the induced relation for each connected component of the edge cover: it combines each binding for the root variable (line 5) with all bindings generated for each of its children (w.r.t. the edge cover of S), line 6–9. If all bindings for v are either failure markers or dangling, $\emptyset$ is returned (and, if this is the only all to Relation for v, the entire induced relation becomes $\emptyset$.

Algorithm 22 performs, for each root variable (lines 3–4 in Algorithm 21), the expansion of the induced relation over all descendant variables. The outer loop (lines 2–6) iterates over all elements of the given interval. There are at most $b^{int} \leq b \leq n$ such elements. For each such element, it iterates (lines 4–5) over all its associated intervals of which there are at most $i \leq n$ per child variable of which there are at most b_{EC} where b_{EC} is the maximum degree of a variable in the edge cover associated with S. For each such interval, a recursive call to Relation with an interval of maximum size $b^{int} \leq b \leq n$ (line 5). The recursive calls return, if a leaf variable in the edge cover of S is found, at a recursion depth of at most d_{EC} where d_{EC} is the maximum depth of the edge cover associated with S. Overall, Algorithm 22 runs in $\mathcal{O}((b_{\text{EC}} \cdot b^{int} \cdot i)^{d_{\text{EC}}} + o)$ where o is the size of the result relation.

The result relation is bound by $b_+(\tilde{q})^{|\tilde{q}|}$ where $\tilde{q}$ is the number of descendant variables of a node v in EC and $b_+(\tilde{q})$ is the maximum number of "good" bindings for any variable in $\tilde{q}$.

In Algorithm 21, we always compute the full induced relation (lines 3–4) only to drop bindings for variables not in V in line 5. The complexity of Algorithm 21 is dominated by the time and space for construction and storing the induced relation and thus by $\mathcal{O}(b_+^{|\text{dom } S|})$. □

Algorithm 22: Relation(S, v, start, end)

```
input  : Sequence map $S$, variable $v \in \text{dom } S$, start and end index
output: Relation containing one tuple for each combination of bindings for $v$ and each of its
        children represented in $S$
ChildVars ← $\{v' \in \text{dom } S : (v, v') \in \text{edgeCover}(S)\}$; res ← $\{\langle\rangle\}$;
for $i$ ← start to min(end, $|S(v)|$) do
    $(n, I) \leftarrow S(v)[i]$;
    if $n = \lightning$ then continue ;
    $R \leftarrow \{\langle v : n\rangle\}$;
    foreach $v' \in$ ChildVars $\cap\ \pi_1(I)$ do
        $R' \leftarrow R$; $R \leftarrow \emptyset$;
        foreach $(v', \text{start}', \text{end}') \in I$ do
            $R \leftarrow R \cup (R' \times \texttt{Relation}(S, v', \text{start}', \text{end}'))$;
    res ← res $\cup\ R$;
return res
```

In most cases, we are not interested to compute the full induced relation, but only in a few of the variables covered by a sequence map. It is also not sufficient to use $\ddot{\pi}$ to remove all the variables we are not interested in as $\ddot{\pi}$ is limited w.r.t. the shape of the variable sets allowed (no variable with parent outside the projection set but ancestor within is allowed). The above algorithm, however, always computes the entire induced relation even for parts of the query not relevant for the variables we are actually interested in and that are specified in V.

Therefore, we present a second algorithm Algorithm 23 that tries to minimize the amount of unnecessary expansion of the sequence map under the assumption that the input sequence map is *consistent*. Recall that using the two propagation operators we can obtain a consistent sequence map in $\mathcal{O}(b \cdot |\text{dom } S| \cdot i)$ time.

Intuitively, by assuming a consistent sequence map, we can avoid even looking at bindings for variables that do not lead to (i.e., are ancestors of) variables in V. Furthermore, even, for variables that lead to variables in V, we do not really care about actual bindings, but only that it is related to bindings of a child variable that leads to a variable in V. If a variable has more than one child variable that is an ancestor of a variable in V (we call such a variable a *branch variable* in Algorithm 23), only then, it is necessary to ensure that only combinations of bindings contributed by each of the child variables are accepted that have a common binding for the parent.

This intuition is realized in Algorithm 23 by first marking (by inclusion in AncestorVars) in lines 2–8 all ancestors of a variable in V.

This set of ancestors is used in lines 9–11 to compute the projected relation of any connected component rooted (w.r.t. the edge cover) at a variable that is in V or ancestor of a variable in V.

```
Algorithm 23: $\mathcal{F}_V(S)$
    input  : Sequence map $S$ and variables $V = \{v_1, \ldots, v_k\} \subset \text{dom}\, S$
    output: $\pi_{v_1,\ldots,v_k}(R_S)$
 1  res ← $\{\langle\rangle\}$;
 2  AncestorVars ← CurrentVars ← $V$;
    // Compute all ancestors
 3  while CurrentVars $\neq \emptyset$ do
 4      $v \in$ CurrentVars;
 5      if $\exists : v' \in \text{dom}\, S \setminus$ AncestorVars $: (v', v) \in EC$ then
 6          AncestorVars ← AncestorVars $\cup \{v'\}$;
 7          CurrentVars ← CurrentVars $\cup \{v'\}$;
 8      CurrentVars ← CurrentVars $\cap \{v\}$;
    // Take all root variables that are ancestors
 9  RootVars ← $\{v \in \text{dom}\, S : \nexists\, v' \in \text{dom}\, S : (v, v') \in$ edgeCover$(S)\}$;
10  foreach $v \in$ RootVars do
11      res ← res × ProjectedRelation$_V^{\text{AncestorVars}}(S, v, \{1 \to (1, |S(v)|)\})$;
12  return res
```

For each such variable, we call Algorithm 24 with a sequence of intervals containing one single interval over all bindings for that variable in S.

Algorithm 24 computes the projection to V of the induced relation of the component rooted at v. However, it avoids, where possible, to compute the induced relation for the variables not in V. For this, the central observation is that, if a variable is neither part of V nor the least common ancestor of two variables in V, it has only a single child that is ancestor of a variable in V (otherwise it would be least common ancestor of all pairs of variables in V where one is contained in the sub-trees rooted at one of the children and the other in the sub-tree of the other child). For computing the projection to V of the induced relation, it matters in this case not which binding of the parent is connected to which binding of the child. It only matters that a bindings of the child variable is related to *any* binding of the parent (and even that only matters if we are in a sub-tree rooted at the least-common ancestor of at least one pair of variables in V, otherwise we just go to the next variable, line 24). This is exploited in line 23 by computing the union over the intervals of all parent bindings. Observe, that the union of non-overlapping intervals, as computed by Algorithm 25, is still a (minimal) sequence of non-overlapping intervals and can be computed efficiently (in linear time over the two sequences of intervals). We can then continue with that single sequence of intervals. In comparison, at a branch variable (with more than one child in the ancestors A of a variable in V) we call ProjectedRelation *for each binding*. In the worst case, the intervals associated with each binding overlap entirely and ProjectedRelation is called for each binding with an interval set that covers as many bindings of the child as the single interval set computed for a skip variable (lines 18–25).

To compute that single interval we use UnionInts, the last of the operations on sets of intervals, as shown in Algorithm 25.

Algorithm 24: ProjectedRelation$^{A}_{V}(S, v, \mathcal{I}, \mathsf{inLCA})$

```
input : Sequence map $S$, set of variables $V \subset \operatorname{dom} S$, variable $v \in \operatorname{dom} S$, sequence $\mathcal{I}$ of
        non-overlapping sets of intervals in order of start index
output: Projection of the induced relation of $S$ to $V$
ChildVars ← $\{v' : (v, v') \in \mathsf{edgeCover}(S)\}$;
if $v \in V$ then
    // Result variable: we need to return bindings for the variable in the result.
    res ← $\varnothing$;
    foreach $(s, e) \in$ do
        for $i \leftarrow s$ to $e$ do
            $(n, I) \leftarrow S(v)[i]$; $R \leftarrow \{\langle v : n \rangle\}$;
            foreach $v' \in \pi_1(I) \cap$ ChildVars do
                $R \leftarrow R \times \mathtt{ProjectedRelation}_V(S, v', \pi_2(\sigma_{1=v}(I)), \mathbf{true})$;
            res ← res $\cup R$;
else if |ChildVars| > 1 then
    // Branch variable: we need to ensure that bindings from two branches are connected to the same
       binding of the branch variable before combining
    res ← $\varnothing$;
    foreach $(s, e) \in$ do
        for $i \leftarrow s$ to $e$ do
            $(n, I) \leftarrow S(v)[i]$; $R \leftarrow \{\langle\rangle\}$;
            foreach $v' \in \pi_1(I) \cap$ ChildVars do
                $R \leftarrow R \times \mathtt{ProjectedRelation}_V(S, v', \pi_2(\sigma_{1=v}(I)), \mathbf{true})$;
            res ← res $\cup R$;
else if |ChildVars| = 1 then
    // Skip variable: we can “skip” to the next variable.
    $v' \in$ ChildVars;
    if inLCA = true then
        // We care that only bindings covered by any of the intervals in $\mathcal{I}$ are considered
        CurrentInts ← $\varnothing$;
        foreach $(s, e) \in$ do
            for $i \leftarrow s$ to $e$ do
                CurrentInts ← $\mathtt{UnionInts}(\mathsf{CurrentInts}, \mathsf{intervals}_{v'}(S(v)[i]))$;
        res ← $\mathtt{ProjectedRelation}_V(S, v', \mathsf{CurrentInts}, \mathsf{inLCA})$;
    else res ← $\mathtt{ProjectedRelation}_V(S, v', \{(1, |S(v')|)\}, \mathsf{inLCA})$;
return res
```

Algorithm 25: UnionInts($\mathsf{Intervals}_1$, $\mathsf{Intervals}_2$)

input : Two sequences $\mathsf{Intervals}_1, \mathsf{Intervals}_2$ of non-overlapping intervals (*without* associated variables) in order of start index

output: Minimal, non-overlapping sequence of intervals covering all indices contained in intervals of either input sequence, in order of start index

```
NewIntervals ← $\varnothing$ ;
start ← $\infty$; end ← 0;
$i, j, k \leftarrow 1$;
while $i \leq |\mathsf{Intervals}_1|$ or $j \leq |\mathsf{Intervals}_2|$ do
    $(v, s_1, e_1) \leftarrow \mathsf{Intervals}_1[i]$ or $\infty$ if $i > |\mathsf{Intervals}_1|$;
    $(v, s_2, e_2) \leftarrow \mathsf{Intervals}_2[i]$ or $\infty$ if $j > |\mathsf{Intervals}_2|$;
    if $e_1 < s_2$ then                                    // $(s_1, e_1)$ before $(s_2, e_2)$
        $s \leftarrow s_1$; $e \leftarrow e_1$;
        $i$++;
    else if $e_2 < s_1$ then                               // $(s_2, e_2)$ before $(s_1, e_1)$
        $s \leftarrow s_2$; $e \leftarrow e_2$;
        $j$++;
    else                                                   // $(s_1, e_1)$ overlaps $(s_2, e_2)$
        $s \leftarrow \min(s_1, s_2)$;
        $e \leftarrow \max(e_1, e_2)$;
        $i$++; $j$++;
    if end $< s$ then
        if start $\neq \infty$ then NewIntervals$[k]$ ← (start, end); $k$++;
        start ← $s$; end ← $e$;
    else
        start ← min(start, $s$);                           // only for illustration, always start $\leq s$
        end ← max(end, $e$);
if start $\neq \infty$ then NewIntervals$[k]$ ← (start, end); $k$++;
return NewIntervals ;
```

Theorem 3.20. *Algorithm 25 computes a minimal, non-overlapping sequence of intervals representing the union of two sequences of intervals I_1 and I_2 in $\mathcal{O}(|I_1| + |I_2|)$ time and constant additional space.*

Proof. Algorithm 25 computes minimal, non-overlapping intervals by the same observation as for Algorithm 8: an interval is added only, if it is clear that the next binding after its end index is not covered (line 18) or if the end of the sequence is reached (line 23).

The result of Algorithm 25 are the union of the intervals in the input sequences: if a binding is covered by either interval in I_1 or I_2, then either that interval overlaps with some interval in the other sequence (line 13–16) and we extend the currently active interval by all bindings covered in these overlapping intervals (lines 20–22) or start a new interval covering these bindings (line 17–19). If it does not overlap (lines 7–12), we also either extend or create a new interval, but only for the bindings covered by the interval itself.

The algorithm relies on the fact that the interval sets are ordered by increasing start index and non-overlapping. This allows us to infer that, if an interval from I_1 lies before the first (remaining) interval from I_2, then it lies before *all* intervals of I_2 and vice versa. Thus deciding overlapping becomes amortised constant rather than linear in the size of the intervals.

Algorithm 25 runs in $\mathcal{O}(|I_1| + |I_2|)$ as in each iteration of the loop 4–22 either i or j or both are incremented. Getting the next interval from I_1 and I_2 (line 5–6) is by assumption constant, as is adding an interval at the end of NexIntervals (line 18). □

From this result, we can derive the following properties of $\mathcal{F}$ depending on the number of variables in V, the number of least common ancestors of (pairs of) variables in V, and the size of the sub-tree rooted at such least common ancestors.

Theorem 3.21. *Algorithm 21 computes $\mathcal{F}_V(S)$ for a sequence map S and a set of variables $V \subset \operatorname{dom} S$ in $\mathcal{O}(b_{\mathcal{V}}^{|\mathcal{V}|} \cdot b_{\mathcal{W}}^2 + b_{\mathcal{W}}^2 + |\operatorname{dom} S|) \leq \mathcal{O}(n^2 + n^{|\mathcal{V}|} + |\operatorname{dom} S|)$ where $\mathcal{V} = V \cup \{v \in \operatorname{dom} S : \exists v' \neq v'' \in V : \operatorname{lca}_{EC}(v', v'') = v\}$ is the set of all variables in V and their least common ancestors in $EC = \text{edgeCover}(S)$, $\mathcal{W} = \{v \in \operatorname{dom} S : \exists$ path from $v' \in \mathcal{V}$ to $v\} \setminus \mathcal{V}$ is the set of all variables in a sub-tree rooted at a variable in $\mathcal{V}$ except $\mathcal{V}$, and b_N is the maximum number of bindings for a variable in a set of variables N or 1 if $N = \emptyset$.*

Proof. In Algorithm 21, computing the ancestors of any variable in V is bound by $\mathcal{O}(|\operatorname{dom} S|)$ time by stopping the mark process whenever a previously marked ancestor is found (line 5). This means, at worst each variable in $\operatorname{dom} S$ is marked once by lines 5–7.

Algorithm 24 is called for each root of a connected component containing with variables in V (determined by looking at AncestorVars).

If Algorithm 24 is called with a skip node and is in inLCA mode, it first computes a single sequence of intervals from the sequences for each bindings. This is done in $\mathcal{O}(b \cdot (i + b)) = \mathcal{O}(b^2)$ as the size of a non-overlapping sequence of intervals and thus the size of CurrentInts is bound by b and we compute the union of the intervals of each of the b bindings with CurrentInts. Then it calls itself once with a single sequence of intervals for the child variable of v. If it is not in inLCA mode, we just skip to the next variable (constant time).

On a tree, forest, or CIG data, the second interval sequence of UnionInts is always a single interval. In this case, storing the current intervals in an interval tree and querying and modifying that interval tree with each of the single intervals associated with a binding can be done in $\mathcal{O}(b \cdot \log(b))$ time rather than $\mathcal{O}(b^2)$ as the unmodified Algorithm 25. However, in the general case, an interval tree based algorithm performs worse, as the interval sequence associated with each binding is already of size $i \leq b$ and thus the overall time is $\mathcal{O}(b^2 \cdot \log(b))$.

If Algorithm 24 is called with a branch or result variable, it calls itself once for each child variable and interval set associated with a binding of v. Each such interval set is bound by b and the number of calls are bound by $b \times b_{\text{EC}}$ where b_{EC} is the maximum degree of the edge cover of S.

Overall, there are $|\mathcal{V}|$ branch and result nodes and thus the overall run time is bound by $\mathcal{O}(b_{\mathcal{V}}^{|\mathcal{V}|} \cdot b_{\mathcal{W}}^2 + b_{\mathcal{W}}^2)$. □

If $V = \{v\}$ for some variable v, then we can omit the computation of the ancestors and the call to project relation and directly return the set of all $\mathsf{binding}(S(v)[i])$ for $i \leq |S(v)|$. This yields an algorithm linear in the number of bindings for v.

For $|V| > 1$, however, the algorithm needs to know the lca's of each pair of variables in V and the path from these lca's to a variable in V. There is a wealth of well-established approaches for finding the least common ancestor [32, 3] that, at a linear pre-processing cost for the tree (here the query), allow constant look-up of the least common ancestor. However, since we also need the *path* between $\text{lca}(v, v')$ and v, v', the gains by adopting such an approach are limited.

With $\digamma$ the set of operators on sequence maps is complete. The following sections conclude the discussion of the operators by highlighting some of their properties.

3.8 Algebraic Equivalences

The sequence map operations defined above as part of the CIQCAG algebra mirror, where possible, closely relational algebra expressions. This is reflected not only in the definitions throughout the previous sections which reduce most of the operations to operations on the induced relations of the involved sequence maps, but also in the algebraic properties that govern these operations. In the following, we briefly summarize and compare the most important algebraic laws that govern the the sequence map operations. Particularly, we consider the effect of consistent and inconsistent operator variants and the propagation operators.

In the following, we denote with $\text{EC}(S)$ the edge cover of a sequence map S, **SM**(R) a sequence map with the induced relation R, and $S_\varnothing = \textbf{SM}(\varnothing)$ a sequence map with induced relation $R_{S_\varnothing} = \varnothing$. For brevity, when there are both consistent and inconsistent variants of an operator that exhibit the same laws, we denote that with $^{(\lightning)}$. Note, however, that either all, or none of the operators in each formula are of the consistent variant.

Neutral and absorbing element laws. For $\ddot{\cup}$ and $\ddot{\setminus}$, any single-variable or single-edge sequence map with induced relation $\varnothing$ is a the *neutral element*. Note, that there are, as for most other relations, many sequence maps that represent $\varnothing$ as long as we allow inconsistent (where failure markers alone give $\mathsf{Nodes}(D)^{\text{dom}\, S}$ variations of a consistent sequence map that represent the same induced sequence) or not interval-minimal (where non minimal interval sets give up to $2^{\mathsf{Nodes}(D)}$ variants) sequence maps. For $\dddot{\bowtie}_{\cap}^{\lightning}$, $S_\varnothing$ forms is an absorbing element. Table 20 summarizes the laws for neutral and absorbing elements. We can also observe, that $\dddot{\bowtie}^{(\lightning)}$ and $\ddot{\cup}$ return the unmodified input sequence map if it is combined with itself.

(N1)	$S \dddot{\cup} S_\varnothing \leftrightarrow S^*$	(N2)	$S \dddot{\cup} S \leftrightarrow S$
(N3)	$S \dddot{\setminus} S_\varnothing \leftrightarrow S^*$		
(N4)	$S\ \theta\ S_\varnothing \leftrightarrow {S_\varnothing}^*$	(N5)	$S\ \theta\ S \leftrightarrow S$

Precondition: $\textsc{ec}(S_\varnothing) = \textsc{ec}(S)$, both single-variable or both single-edge
Precondition: $\textsc{ec}(S_\varnothing) \cap \textsc{ec}(S) = \varnothing$

Table 20. Neutral and absorbing elements for combination operators ($\theta \in \{\dddot{\bowtie}_\cap^{(\lightning)}, \dddot{\bowtie}^{(\lightning)}, \dddot{\ltimes}^{(\lightning)}\}$)

(C1)	$S_1 \dddot{\cup} S_2 \leftrightarrow S_2 \dddot{\cup} S_1{}^*$
(C2)	$S_1 \dddot{\bowtie}_\cap^{(\lightning)} S_2 \leftrightarrow S_2 \dddot{\bowtie}_\cap^{(\lightning)} S_1{}^*$
(C3)	$S_1 \dddot{\bowtie}^{(\lightning)} S_2 \leftrightarrow S_2 \dddot{\bowtie}^{(\lightning)} S_1$
(A1)	$(S_1 \dddot{\cup} S_2) \dddot{\cup} S_3 \leftrightarrow S_1 \dddot{\cup} (S_2 \dddot{\cup} S_3)^*$
(A2)	$(S_1 \dddot{\bowtie}_\cap^{(\lightning)} S_2) \dddot{\bowtie}_\cap^{(\lightning)} S_3 \leftrightarrow S_1 \dddot{\bowtie}_\cap^{(\lightning)} (S_2 \dddot{\bowtie}_\cap^{(\lightning)} S_3)^*$
(A3)	$(S_1 \dddot{\bowtie}^{(\lightning)} S_2) \dddot{\bowtie}^{(\lightning)} S_3 \leftrightarrow S_1 \dddot{\bowtie}^{(\lightning)} (S_2 \dddot{\bowtie}^{(\lightning)} S_3)$
(DM1)	$S_1 \dddot{\setminus} (S_2 \dddot{\cup} S_3) \leftrightarrow (S_1 \dddot{\setminus} S_2) \dddot{\bowtie}^{(\lightning)} (S_1 \dddot{\setminus} S_3)^*$
(DM2)	$S_1 \dddot{\setminus} (S_2 \dddot{\bowtie}^{(\lightning)} S_3) \leftrightarrow (S_1 \dddot{\setminus} S_2) \dddot{\cup} (S_1 \dddot{\setminus} S_3)^*$
(D1)	$S_1 \dddot{\cup} (S_2 \dddot{\bowtie}^{(\lightning)} S_3) \leftrightarrow (S_1 \dddot{\cup} S_2) \dddot{\bowtie}^{(\lightning)} (S_1 \dddot{\cup} S_3)^*$
(D2)	$S_1 \dddot{\bowtie}^{(\lightning)} (S_2 \dddot{\cup} S_3) \leftrightarrow (S_1 \dddot{\bowtie}^{(\lightning)} S_2) \dddot{\cup} (S_1 \dddot{\bowtie}^{(\lightning)} S_3)^*$
(D3)	$S_1 \dddot{\bowtie}_\cap^{(\lightning)} (S_2 \dddot{\bowtie}^{(\lightning)} S_3) \leftrightarrow (S_1 \dddot{\bowtie}_\cap^{(\lightning)} S_2) \dddot{\bowtie}^{(\lightning)} (S_1 \dddot{\bowtie}_\cap^{(\lightning)} S_3)^*$
(D4)	$S_1 \dddot{\ltimes}^{(\lightning)} (S_2\ \theta\ S_3) \leftrightarrow (S_1 \dddot{\ltimes}^{(\lightning)} S_2)\ \theta\ (S_1 \dddot{\ltimes}^{(\lightning)} S_3)^*$

Precondition: $\textsc{ec}(S_1) = \textsc{ec}(S_2) = \textsc{ec}(S_3)$, S_1, S_2, S_3 all single-variable or all single-edge
Precondition: $\textsc{ec}(S_1) \cap \textsc{ec}(S_2) = \varnothing$, $\textsc{ec}(S_1) \cap \textsc{ec}(S_3) = \varnothing$, $\textsc{ec}(S_2) \cap \textsc{ec}(S_3) = \varnothing$
Precondition: $\textsc{ec}(S_1) \cap \textsc{ec}(S_2) = \varnothing$, $\textsc{ec}(S_1) \cap \textsc{ec}(S_3) = \varnothing$

Table 21. Commutative, associative, distributive, de Morgan laws ($\theta \in \{\dddot{\bowtie}_\cap^{(\lightning)}, \dddot{\bowtie}^{(\lightning)}\}$)

$$\text{(S1)}\quad \dddot{\sigma}_c^{(\lightning)}(\dddot{\sigma}_c^{(\lightning)}(S)) \leftrightarrow \dddot{\sigma}_c^{(\lightning)}(S)$$

$$\text{(S2)}\quad \dddot{\sigma}_c^{(\lightning)}(S_1\ \theta\ S_2) \leftrightarrow \dddot{\sigma}_c^{(\lightning)}(S_1)\ \theta\ \dddot{\sigma}_c^{(\lightning)}(S_2)^{*} \qquad \text{(S2a)}\quad \dddot{\sigma}_c^{(\lightning)}(S_1\ \phi\ S_2) \leftrightarrow \ddot{\sigma}_c(S_1)\ \phi\ \ddot{\sigma}_c(S_2)^{*}$$

$$\text{(S3)}\quad \dddot{\sigma}_c^{(\lightning)}(S_1\ \theta\ S_2) \leftrightarrow \dddot{\sigma}_c^{(\lightning)}(S_1)\ \theta\ S_2{}^{*} \qquad \text{(S3a)}\quad \dddot{\sigma}_c^{(\lightning)}(S_1\ \phi\ S_2) \leftrightarrow \ddot{\sigma}_c(S_1)\ \phi\ S_2{}^{*}$$

$$\text{(S4)}\quad \dddot{\sigma}_c^{(\lightning)}(S_1\ \theta\ S_2) \leftrightarrow S_1\ \theta\ \dddot{\sigma}_c^{(\lightning)}(S_2)^{*} \qquad \text{(S4a)}\quad \dddot{\sigma}_c^{(\lightning)}(S_1\ \phi\ S_2) \leftrightarrow S_1\ \phi\ \ddot{\sigma}_c(S_2)^{*}$$

Precondition: $\mathsf{Vars}(c) \in \operatorname{dom} S_1 \cap \operatorname{dom} S_2$
Precondition: $\mathsf{Vars}(c) \in \operatorname{dom} S_1 \setminus \operatorname{dom} S_2$
Precondition: $\mathsf{Vars}(c) \in \operatorname{dom} S_2 \setminus \operatorname{dom} S_1$

Table 22. Selection laws ($\theta \in \{\dddot{\cup}, \dddot{\bowtie}_{\cap}^{\lightning}, \dddot{\bowtie}^{\lightning}, \dddot{\ltimes}^{(\lightning)}, \dddot{\setminus}\}$, $\phi \in \{\dddot{\bowtie}_{\cap}, \dddot{\bowtie}\}$)

Commutative, associative, and distributive laws. Figure 21 summarizes commutative, associative, distributive, and de-Morgan laws for sequence map combination operators: $\ddot{\cup}$ and all join variants are commutative and associative as is easy to see from their definition (C1–C3, A1–A3). Moreover, a variation (DM1–DM2) of de-Morgan laws hold for $\ddot{\setminus}$, $\ddot{\cup}$, and $\dddot{\bowtie}^{(\lightning)}$ (which takes the place of $\cap$ in usual set-theoretic formulations of de-Morgan laws), but only if all involved sequence maps are either all single-variable or all single-edge and have the same edge cover. This is, in fact, the precondition for all laws involving $\ddot{\setminus}$ or $\ddot{\cup}$. Recall, that in this case $\dddot{\bowtie}^{(\lightning)}$ is equivalent to intersection as all variables and edges are shared. $\dddot{\bowtie}_{\cap}^{(\lightning)}$ can not be used, here, as it prohibits overlapping edge covers (which are required by $\ddot{\cup}$ and $\ddot{\setminus}$). For the same reason, only $\dddot{\bowtie}^{(\lightning)}$ and $\ddot{\cup}$ distribute over each other (D1–D2) on all single-edge or all single-variable sequence maps, but not $\dddot{\bowtie}_{\cap}^{(\lightning)}$. Finally, distribution between the two join types $\dddot{\bowtie}_{\cap}^{(\lightning)}$ and $\dddot{\bowtie}^{(\lightning)}$ is limited, $\dddot{\bowtie}_{\cap}^{(\lightning)}$ distributes over $\dddot{\bowtie}^{(\lightning)}$ (D3), but, in general, not the other way around.

Table 21. The commutative and associative laws are easy to verify in the respective definition. For the de-Morgan laws, consider that a tuple is in the induced relation of $S_1 \ddot{\setminus} (S_2 \ddot{\cup} S_3)$ if it is in R_{S_1} but neither in R_{S_2} nor in R_{S_3}. It is in $(S_1 \ddot{\setminus} S_2)\ \dddot{\bowtie}^{(\lightning)}\ (S_1 \ddot{\setminus} S_3)$ if it is both in R_{S_1} but not in R_{S_2} and in R_{S_1} but not in R_{S_3}. This is the case due to the precondition that the edge covers of all three sequence maps are the same. Thus, $R_{S_1} \bowtie R_{S_2} = R_{S_1} \cap R_{S_2}$. Analog for DM2.

For the distributive laws D1 and D2, the core observation is again that all involved sequence maps have the same edge cover. Thus the induced relation of $\dddot{\bowtie}^{(\lightning)}$ is the intersection of the induced relations of its input sequence maps.

For D3, consider that $S_1\ \dddot{\bowtie}_{\cap}^{(\lightning)}\ (S_2\ \dddot{\bowtie}^{(\lightning)}\ S_3)$ is a valid expression only if $\textsc{ec}(S_2\ \dddot{\bowtie}^{(\lightning)}\ S_3) \cap \textsc{ec}(S_1) = \textsc{ec}(S_2) \cup \textsc{ec}(S_3) \cap \textsc{ec}(S_1) = \varnothing$. Thus it is valid if $\textsc{ec}(S_1) \cap \textsc{ec}(S_3) = \varnothing$ and $\textsc{ec}(S_2) \cap \textsc{ec}(S_3) = \varnothing$, in which case also $(S_1\ \dddot{\bowtie}_{\cap}^{(\lightning)}\ S_2)\ \dddot{\bowtie}^{(\lightning)}\ (S_1\ \dddot{\bowtie}_{\cap}^{(\lightning)}\ S_3)$ is valid. □

Selection. Selection is generally a good candidate for optimization, pushing selections (a fast but possibly fairly selective operation) inside of an expression thus limiting the size of intermediary results. Also selection can generally be propagated "down" into an expression: Selection distributes over $\ddot{\cup}$, $\dddot{\bowtie}_{\cap}^{\lightning}$, $\dddot{\bowtie}^{\lightning}$ and $\ddot{\setminus}$, see Table 22. For $\ddot{\bowtie}_{\cap}$ and $\ddot{\bowtie}$, we need to ensure that the consistency of the input sequence maps is retained and thus can only push $\ddot{\sigma}$ inside (regardless of the outer selection variant), cf. (S2a, S3a, S4a).

$$\text{(P1)} \quad \dddot{\pi}_V(\dddot{\pi}_{V'}(S)) \leftrightarrow \dddot{\pi}_{V \cup V'}(S)^{*} \qquad \text{(P2)} \quad \dddot{\pi}_V(\dddot{\sigma}_c^{(\not{\iota})}(S)) \leftrightarrow \dddot{\sigma}_c^{(\not{\iota})}(\dddot{\pi}_V(S))^{*}$$

$$\text{(P3)} \quad \dddot{\pi}_V(S_1 \; \theta \; S_2) \leftrightarrow \dddot{\pi}_{V_1}(S_1) \; \theta \; \dddot{\pi}_{V_2}(S_2)^{*} \qquad \text{(P4)} \quad \dddot{\pi}_V(S_1 \; \dddot{\ltimes}^{(\not{\iota})} \; S_2) \leftrightarrow \dddot{\pi}_{V_1}(S_1) \; \dddot{\ltimes}^{(\not{\iota})} \; S_2{}^{*}$$

Precondition: the projection condition holds for V, V' and no pair of variables from $v \in V$ and $v' \in V'$ has a common ancestor in the edge cover of S

Precondition: $\mathsf{Vars}(c) \notin V$

Precondition: $V_1 \cup V_2 = V$, V_1 (V_2) fulfills the projection condition for S_1 (S_2) with $\operatorname{dom} S_1 \cap V \setminus V_1 = \emptyset$ ($\operatorname{dom} S_2 \cap V \setminus V_2 = \emptyset$)

Table 23. Projection laws ($\theta \in \{\dddot{\bowtie}_{\cap}^{(\not{\iota})}, \dddot{\bowtie}^{(\not{\iota})}\}$)

$$(\varpi 1) \quad \dddot{\varpi}_{v}^{\uparrow}(S) \leftrightarrow S^{*} \qquad (\varpi 2) \quad \dddot{\varpi}^{v}_{\downarrow}(S) \leftrightarrow S^{*}$$

$$(\varpi 3) \quad \dddot{\varpi}(S_1 \; \theta \; S_2) \leftrightarrow S_1 \; \theta \; S_2{}^{*} \qquad (\varpi 4) \quad \dddot{\varpi}(\dddot{\sigma}_c(S)) \leftrightarrow \dddot{\sigma}_c(S)^{*}$$

$$(\varpi 5) \quad \dddot{\varpi}(\dddot{\mu}_v(D, Q, R)) \leftrightarrow \dddot{\mu}_R(D, Q)v \qquad (\varpi 6) \quad \dddot{\varpi}(\dddot{\mu}_{v_1, v_2}(D, Q)) \leftrightarrow \dddot{\mu}_{v_1, v_2}(D, Q)$$

Precondition: $v \in \operatorname{dom} S$ equivalence does not preserve consistency

Precondition: $v \in \operatorname{dom} S$ equivalence does not preserve consistency

Precondition: S_1, S_2 consistent

Precondition: S consistent

Table 24. Propagation laws ($\dddot{\varpi} \in \{\dddot{\varpi}_{v}^{\uparrow}, \dddot{\varpi}^{v}_{\downarrow}\}$, $\theta \in \{\dddot{\setminus}, \dddot{\bowtie}_{\cap}, \dddot{\bowtie}, \dddot{\ltimes}\}$)

For $\dddot{\setminus}$ and $\dddot{\ltimes}^{(\not{\iota})}$, we can actually drop the selection around S_2 in (S2), since we only retain tuples from the first input sequence map.

Projection. In contrast to selection, we can only propagate selection "down" in an expression to the point where the projection condition that $V \subset \operatorname{dom} S$ and, for all pairs $v, v' \in V$ all variables on the path from $\operatorname{lca}(v, v')$ to each variable are in V still hold in the sub-expressions. Conversely, an expression might benefit from introducing additional projections to get rid of attributes not used in the remainder of an expression as early as possible, viz. immediately after the innermost expression referencing them.

Given two sets of variables V and V' such that the projection condition on S holds for both sets and no pair $v \in V, v' \in V'$ has a common ancestor in the edge cover of S, we can combine a sequence of projections for these two sets into a single projection for $V \cup V'$ (P1). Projection and selection can be arbitrarily ordered as long as $\mathsf{Vars}(c) \notin V$ (P2). Finally, $\dddot{\pi}$ distributes over $\dddot{\bowtie}_{\cap}^{(\not{\iota})}$, $\dddot{\bowtie}^{(\not{\iota})}$, $\dddot{\ltimes}^{(\not{\iota})}$ if there are subsets V_1, V_2 of V such that each subset affects one of the input sequence maps but not the other. For $\dddot{\ltimes}^{(\not{\iota})}$, we can omit the projection on S_2 since only bindings from S_1 are retained.

Propagation. Both propagation operators do not affect the induced sequence only the consistency state. Thus, as long as we are only interested in the induced sequence, we can add or remove propagation operators arbitrarily (ϖ1–ϖ2). Finally, for each of the consistent sequence map operators (such as $\dddot{\bowtie}_{\cap}$ or $\dddot{\setminus}$) we can drop any surrounding propagation operators if the input sequence maps are consistent itself (ϖ3–ϖ6).

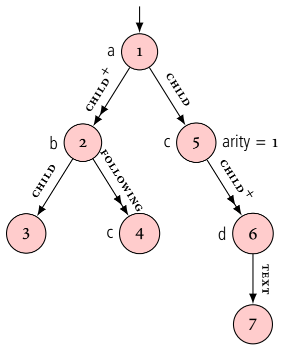

Figure 23. Example query for iterator approach

Extract. Extract returns a relation rather than a sequence map. Thus it can not be distributed over any of the sequence map operations. However, it may interact with relational expressions as discussed in Section **??**.

This concludes the brief overview of the most important algebraic laws for sequence map operators in CIQCAG. Before we extend the sequence map operators that are limited to the evaluation of tree queries, to the full CIQCAG algebra with (standard relational) operators for the evaluation of non-tree parts of an arbitrary query and operators for construction and iteration, we briefly outline in the following section an iterator model for the physical realisation of the sequence map operators.

3.9 Iterator Implementation

Iterator or stream model has proved essential for the scalable evaluation of relational queries, see, e.g., [27]. Here, we briefly outline how CIQCAG's sequence map operators can be implemented in an iterator model that reduces the space complexity of the evaluation compared to the sequence-at-a-time model discussed above. This holds in particular on tree or forest data and if the number of variables that are either part of the answer or used in the non-tree part of the query (i.e., the number of variables extracted by $\digamma$) is small compared to the full number of variables in the query.

Let $\digamma_V(\mathfrak{E})$ be a CIQCAG expression such that $\mathfrak{E}$ contains only join, semi-join, selection and initialization operators. We limit the operators to keep the discussion brief, but we can extend the approach to allow also projection, union, and difference. Note, that we can order $\mathfrak{E}$ in such a way that all unary conditions (i.e., the unary initialization operators and joins to connect them) for each query variable v are clustered and the result of all these unary conditions is connected by a single join to the expressions representing the conditions on each child variable of v. We denote with $\mathcal{L}(v)$ the unary conditions relating to v and with $\mathcal{E}(v)$ the full expression for v including the expressions for its children.

Consider the query in Figure 23. It can be realized by the CIQCAG expression $\mathcal{F}_{v_7}(\mathfrak{E})$ where

$$
\left.\begin{aligned}
\mathfrak{E} = {} & \overbrace{\dddot{\mu}_{v_1}(D, Q, \mathsf{Label}_{‘a’})}^{\mathcal{L}(v_1)} \,\dddot{\bowtie}\, \dddot{\mu}_{v_1,v_2}(D, Q) \,\dddot{\bowtie}\, \dddot{\mu}_{v_1,v_5}(D, Q) \\
& \left.\begin{aligned}
& \dddot{\bowtie}\, \dddot{\mu}_{v_2}(D, Q, \mathsf{Label}_{‘a’}) \\
& \quad \dddot{\bowtie}\, \dddot{\mu}_{v_2,v_3}(D, Q) \\
& \quad \dddot{\bowtie}\, \dddot{\mu}_{v_2,v_4}(D, Q)\dddot{\mu}_{v_4}(D, Q, \mathsf{Label}_{‘c’})
\end{aligned}\right\} \mathcal{E}(v_2) \\
& \left.\begin{aligned}
& \dddot{\bowtie}\, \dddot{\mu}_{v_5}(D, Q, \mathsf{Label}_{‘c’}) \,\dddot{\ltimes}\, \dddot{\mu}_{v_5}(D, Q, \mathsf{Arity}_{=1}) \\
& \quad \dddot{\bowtie}\, \dddot{\mu}_{v_5,v_6}(D, Q) \,\dddot{\bowtie}\, \dddot{\mu}_{v_6}(D, Q, \mathsf{Label}_{‘d’}) \\
& \qquad \dddot{\bowtie}\, \dddot{\mu}_{v_6,v_7}(D, Q)
\end{aligned}\right\} \mathcal{E}(v_5)
\end{aligned}\right\} \mathcal{E}(v_1)
$$

This is an example of the general shape of an expression such that it is amenable to the iterator approach discussed in this section. Figure 24 illustrates this shape in detail: blue colored parts of the query together with their root represent the part of the query rooted at the corresponding variable v, i.e., $\mathcal{E}(v)$. The yellow colored parts represent the unary conditions for each variable, i.e., $\mathcal{L}(v)$. For each query variable v, there is a *representing node* (indicated by ● and labeled with the variable in Figure 23). Under this node we find, in general, two children: one for $\mathcal{L}(v)$ and one join node grouping all of v's children (we use here a multi-way join instead of a sequence of binary joins to simplify the presentation). For each child variable v' of v, this node has a *connection node* $\dddot{\mu}_{v,v'}(D, Q)$ and the representative join node for v'. Some variables have no unary restrictions, some no children in which case the respective parts are omitted (see v_4). If a variable has neither, we use the connection node as representative node for that variable (see v_3, v_7).

Based on this ordering of the CIQCAG expression, we can now define a processing scheme that computes the resulting sequence map for the entire expression incrementally, outputs potential answers as soon as there is an extension to a full answer, and discards potential answers as soon as there can be no more extension to a full answer.

The fundamental observation for this scheme is that all unary operators are already implemented in an iterator fashion, see Section 3, i.e., as a single pass over the input sequence using, for each binding in the sequence, only conditions on that binding itself (and not on any other binding) to determine its inclusion or exclusion from the result.

For $\dddot{\bowtie}$ and $\dddot{\mu}_{v,v'}(D, Q)$, this is not so easy to see from the Algorithms in Section 3. In the following, we focus on these two operators to detail the incremental processing scheme. First, we annotate each join with the intersection of the variables of the sequence maps computed by its children. For all variable nodes in Figure 24, this yields a single variable, viz. the one shown as label in Figure 24. For the semi-joins (which only connect unary operators), this yields also a single variable (viz. the variable their parent is annotated with). For child joins, i.e., joins with one $\dddot{\mu}_{v,v'}(D, Q)$ and, possibly, one variable node, this yields the two variables v, v' (note that the multi-way joins in Figure 24 are realized as a sequence of binary joins) such that $v = \mathsf{parent}(v')$.

The first change is that an operator no longer computes the entire result sequence map at once, but that each operator provides an interface with three functions next, out, and close. This extends the iterator interface used, e.g., in [27] for relational operators with out and modifies the semantics for next: We can call next on each operator with or without a pair (variable, binding).

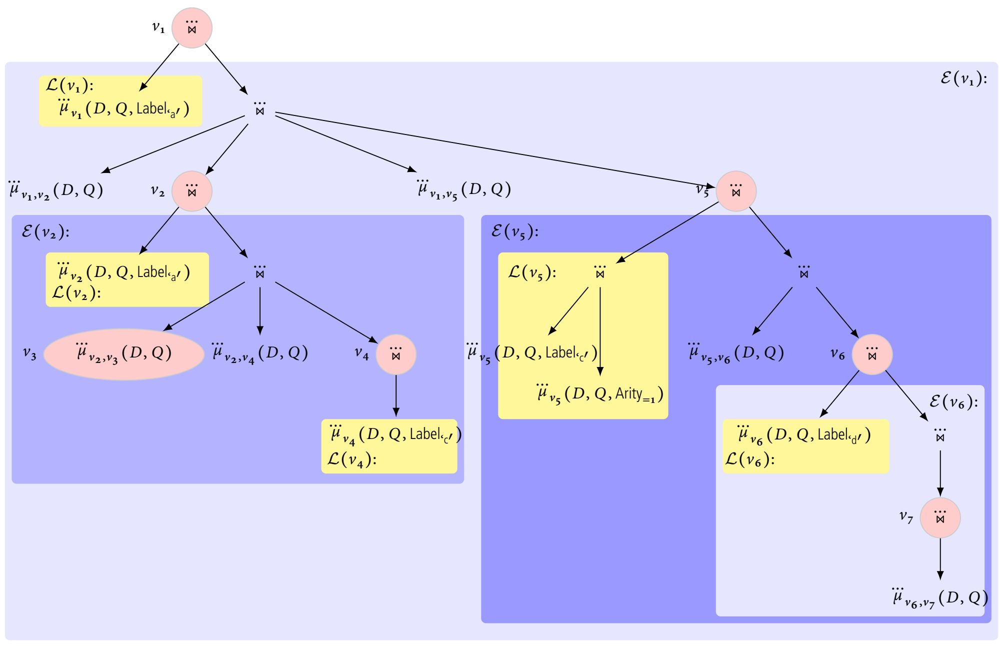
v_1
$\mathcal{E}(v_1)$:
$\mathcal{L}(v_1)$:
$\ddot{\mu}_{v_1}(D, Q, \text{Label}_{a'})$
$\ddot{\mu}_{v_1,v_2}(D, Q)$
v_2
$\ddot{\mu}_{v_1,v_5}(D, Q)$
v_5
$\mathcal{E}(v_2)$:
$\ddot{\mu}_{v_2}(D, Q, \text{Label}_{a'})$
$\mathcal{L}(v_2)$:
v_3
$\ddot{\mu}_{v_2,v_3}(D, Q)$
$\ddot{\mu}_{v_2,v_4}(D, Q)$
v_4
$\ddot{\mu}_{v_4}(D, Q, \text{Label}_{c'})$
$\mathcal{L}(v_4)$:
$\mathcal{E}(v_5)$:
$\mathcal{L}(v_5)$:
$\ddot{\mu}_{v_5}(D, Q, \text{Label}_{c'})$
$\ddot{\mu}_{v_5}(D, Q, \text{Arity}_{=1})$
$\ddot{\mu}_{v_5,v_6}(D, Q)$
v_6
$\mathcal{E}(v_6)$:
$\ddot{\mu}_{v_6}(D, Q, \text{Label}_{d'})$
$\mathcal{L}(v_6)$:
v_7
$\ddot{\mu}_{v_6,v_7}(D, Q)$

Figure 24. Operator tree for query from Figure 23

Unary initialization, if called without such a pair, compute the next binding in their result sequence and return that binding (or **false** if there are no more bindings), remembering the point in the result sequence reached by the computation as well as the computed partial sequence of bindings; if called with such a binding pair they return the binding, if it is included in the result sequence and the variable is consistent. They also advance the computation of that result sequence up to the point where the binding is computed, if necessary (i.e., if the binding has not already been computed). Otherwise they return **false**.

A join remembers the *all* bindings returned by its first operand as well as the last pair next was called with. If next is called on a *join* and the passed pair is the same as in the last call, it calls the right operand's next with the last remembered pair returned by the first operand. If that call returns **false** or the passed pair is different from the last call, it calls the left operand's next with the passed pair, if there is any, and returns **false** if that call returns **false**. Otherwise, it remembers the new pair returned and calls next on the right operand with that pair. If that returns **false**, we loop and call next on the left operand again etc. As soon as the right operand's next returns a pair, that pair is returned.

For *binary initialization* operators for v, v', we return for a call with next (1) if a pair (v'', n) is passed with $v'' = v$, and n is in the partial sequence of bindings for v already computed, we return the next binding for v' related to n (i.e., covered by an interval pointer associated with n) together with v', if there is none return **false**. Otherwise, we expand the partial sequence of bindings for v until we find n or the first binding n' with $n > n'$ wrt. the order associated to the relation between v and v'. If we find n, return the first binding for v' related to n, if there is any, together with v'. Otherwise return **false**. (2) if no pair is passed, we take the first binding n for v where not all related bindings for v' have been generated (and return **false** if there is no such binding). For that binding, we compute the next binding for v' and return that binding. Note, that the binary initialization stores the partial binding sequences for v and v' as well as all interval pointers from bindings in the partial binding sequence for v to bindings of v' for which not all covered bindings have been returned by next. For each such binding n in the partial binding sequence of v, we also store a single pointer into the sequence of bindings for v' indicating the last returned binding for v' that is related to n.

The second function close removes from all operators the current state, i.e., resets the iterations (as for the iterator approach on relational operators discussed in [27]).

The final function of the iterator interface, out, each operand calls out on its child operators. In addition, if an operator is a variable node, it outputs the current binding for the variable it is labeled with together with any intervals associated with it.

The query is embedded in an outer loop, that calls next on the first operand of the query (for queries with size > 1 always a join). If the call returns **true**, it calls **out** on that join and continues the loop. Otherwise it terminates.

Consider again the operator tree from Figure 24: First next is called on the top-level join. The join calls its first operand ($\ddot{\mu}_{v_1}(D, Q, \text{Label}_{`a'})$) without parameter. $\ddot{\mu}_{v_1}(D, Q, \text{Label}_{`a'})$ returns the first node in the document with label 'a'. With that binding for v_1 the top-level join calls the "child join". That join calls first $\ddot{\mu}_{v_1,v_2}(D, Q)$ with the binding pair and then with the result the variable join node for v_2, … Two observations are crucial here: Except for the leftmost, all initialization operators are always called with a binding pair. Partial sequence maps are stored in initialization operators (joins only retain the last successful binding from the second operand). E.g., $\ddot{\mu}_{v_2,v_3}(D, Q)$ builds a sequence map consisting of bindings for v_2 in the associated order of the relation between v_1 and v_2, and bindings for v_3 in the associated order of the relation between v_2 and v_3. Only bindings for v_2 are "restricted" by binding pairs passed in next, not bindings for v_3. This avoids dropping intermittent bindings for v_3 and thus allows to keep the interval pointers as given by the interval representation discussed in Section 3.2. Each of these sequence maps is bound by $\mathcal{O}(n \times i)$ where n is the number of nodes in D and i the maximum size of an interval per binding. For join operators that are also variable nodes, we also store a sequence map over the bindings of the variable the node is labeled with. These sequence map are bound by $\mathcal{O}(n)$ since we do not retain intervals and can, at no additional cost, be ordered in the associated order of the incoming edge of the variable the join is labeled with.

3.9.1 Optimal Space Bounds for Tree Data

In contrast to the operators discussed in Section 3, here we compute sequence maps incrementally. For tree data, we can profit from another regularity to further decrease the space complexity: Recall, that tree data corresponds to relations with image disjointness property and, if we also allow closure axis, with image containment property. For a relation with image disjointness property, the number of parents of a node is bound by 1. In other words, as soon as we have found a single parent for a node, there can be no further nodes related to it among the bindings for the parent variable. For a relation R with image containment, the number of parents is limited by the depth of the forest represented by the relation whose closure is R (by Theorem 2.4 there is such a forest-shaped base relation for each R). Consider, e.g., XPath closure relations descendant, following, or following-sibling. For descendant the base relation is child and as such the number of parents under descendant is limited by the depth of the queried XML tree. For following, the base relation is the relation associating with each element the next element in document order. For that relation, the number of ancestors is limited only by the size of the XML tree. For following-sibling it is the relation that associates with each node the next element in document order that has the same parent. Thus, the number of ancestors is limited by the degree of the XML tree.

How can we exploit this observation in the iterator algorithm sketched above? The aim is to reduce the size of the partial sequence maps and the stores in join operators by deleting nodes as soon as they can no longer contribute to any further match under the assumption that all relations carry image containment or image disjointness property. We know, that as soon as a binding is related to 1, resp. d (depth of base relation), different bindings of its parent, it can not contribute to any further and, thus no longer needs to be stored. However, for image containment the related bindings may scattered over the entire binding sequence for the parent variable and thus a binding added at the beginning of the processing (i.e., related to the first binding of the parent variable) may be amenable to be removed only at the very end (if the last binding of the parent variable is also related to it).

Thus, we impose a further property to hold for the relations used in a query, that guarantees us that all parent bindings related to a given child binding are "clustered" together. It also ensures, that there are no parent bindings n that are related to child bindings prior to child bindings related to parent bindings prior to n:

Definition 3.14 (Order-compatible query). Let D be a relational structure and Q a query on D. Then Q is called *order-compatible* if, for each pair (v, v') of parent-child variables in Q with $\mathsf{rel}(v) = R$ and $\mathsf{rel}(v') = R'$, it holds that $n <_R n'$ implies that, for all nodes $c \in R'(n), c' \in R'(n')$, $c, c' \in R'(n) \cap R'(n')$ or $c \leq_{R'} c'$.

Intuitively, the two orders associated with the parent and the child variable are compatible in the sense that if a given child binding is either related to a binding of the parent or neither it or any child bindings after the given one are related to a binding of the parent.

The class of order-compatible queries is interesting because of the following result:

Theorem 3.22. *For a tree query Q containing only a single, forest-shaped base relation R and its closure relation C there are orders for R and C such that Q is* order-compatible *and the images of all nodes under R and C form a single continuous interval under the respective orders.*

Proof. In the following, we consider only tree based relations for simplicity. However, for forest we simply add some fix order on the connected components of the forest to the definitions.

Order the children of each node in R in some order and call the ordered tree induced by R and this order T. Choose as order for R the breadth-first left-to-right preorder traversal $<_b$ of T. Choose as order for C the depth-first left-to-right preorder traversal $<_d$ of T.

Then, if $n <_b n'$, either n ancestor of n' ($<_b$ is *preorder*) and $R'(n') \subset R'(n)$ or there is an ancestor a of n and an ancestor a' of n' such that a is a preceding sibling of a' (and both are children of $\mathrm{lca}(a, a')$). Then, also $a <_c a'$ (since the depth-first traversal is *left-to-right*) and all descendants of a come before all descendants of a' in $<_c$. In particular any $c \in C(n)$ is a descendant of a, any $c' \in C(n')$ a descendant of a' and thus $c <_c c'$. □

This yields, e.g., that tree queries containing only child and descendant.

Corollary 3.3. *Tree queries containing only* child *and* descendant *are* order-compatible *(for some orders on* child *and* descendant*, resp.).*

Under these assumptions (tree or forest data, order-compatible query wrt. the associated orders of the involved relations), we can now adapt the algorithm: **(1)** As soon as we find no more child bindings for a parent, that parent is dropped from the sequence map of a binary initialization operator, since the data is tree shaped and thus each parent has a single interval pointer. **(2)** Each child binding is dropped as soon as the first parent binding is encountered, that does not relate to it, since no further parent binding can related to that child binding. **(3)** When next is called for a binary initialization operator, we alternate between the two binding sequences: We find the first binding for the parent together with its first related binding for the child. The next call to next returns not the next binding for the child variable related to current parent binding (as in the original algorithm), but the next parent binding related to the current child binding. If there is no such binding left, we delete the child binding and continue with the first binding of the parent variable that has yet more related bindings for the child variable (and delete all parents before that parent binding). We let the binary initialization operator report the deleted bindings (as pairs of variable and the largest deleted bindings) as a further result of next to its parent (join) operator. **(4)** For unary initialization operators, we only store the last binding. **(5)** In a join operator, we delete all binding up to and including the delete bindings returned by a next and propagate them upwards, unless they are for the variable the join is labeled with. If the left operand of the join operator is no binary initialization operator, that join operator only stores the last binding.

Theorem 3.23. *For order-compatible queries on tree or forest data, the above algorithm runs in $\mathcal{O}(q \cdot n + o)$ time and $\mathcal{O}(q \cdot d + o)$ space where o is the size of the output, q is the size of the query, n the size of the data, and d the depth of the tree or forest.*

Proof. The modifications of the algorithm do not affect correctness. For the binary initialization operators this follows from the forest shape of the data and the order-compatibility of the query. For the join operators without left binary initialization operator, there is either always an ancestor join operator that memoizes the bindings (e.g., for the "child join" operators in Figure 24) or it is the top-level join. For unary initialization operators, we can observe that same fact. The reason this holds is that the query is connected and tree-shaped. Thus for each variable except for the root variable there is a binary initialization operator (and a corresponding join) that "generates" bindings for the variable from bindings of the parent variable.

The time complexity is the same as for the sequence-at-a-time operators from Section 3. Note that we operate on tree or forest data and thus the number of intervals per binding is at most 1.

It retains the complexity of the set-at-a-time algorithm by memoizing already computed bindings in variable join nodes and initialization operators, as described above. The deletion of bindings does not affect the complexity (it adds an additive factor of n to the complexity).

However, by alternating between parent and child nodes we can ensure that all the partial binding sequences in the sequence maps of the binary initialization operators contain at most d bindings for the parent variable and at most 1 binding for the child variable.

The memoization structures in the join operators remove bindings at the same time as their subordinate binary initialization operators. Thus, they are also bound by d. □

Corollary 3.4. *Queries containing only XPath's* child *and* descendant *relation are evaluated by the above* *algorithm in $\mathcal{O}(q \cdot n + o)$ time and $\mathcal{O}(q \cdot d + o)$ space where o is the size of the output, q is the size of the query, n the size of the data, and d the depth of the tree or forest.*

The last results are optimal wrt. data complexity as they coincide with the $\Omega(d + o)$ lower bound for the data complexity of such queries shown in [51].

However, it is an open question, whether order-compatible queries are the largest class of queries that can be evaluated with this complexity.

Acknowledgements.

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

Bibliography

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 253–262, New York, NY, USA, 1989. ACM.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. Int. Conf. on Data Engineering*, page 141, Washington, DC, USA, 2002. IEEE Computer Society.

[3] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *Proc. ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 258–264, New York, NY, USA, 2002. ACM.

[4] M. Altinel and M. J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[5] A. Berlea and H. Seidl. Binary Queries for Document Trees. *Nordic Journal of Computing*, 11(1):41–71, 2004.

[6] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery Processor powered by a Relational Engine. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 479–490, New York, NY, USA, 2006. ACM Press.

[7] K. S. Booth and G. S. Lueker. Linear Algorithms to Recognize Interval Graphs and Test for the Consecutive Ones Property. In *Proc. of ACM Symposium on Theory of Computing*, pages 255–265, New York, NY, USA, 1975. ACM Press.

[8] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 310–321, New York, NY, USA, 2002. ACM Press.

[9] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The XML Stream Query Processor SPEX. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, pages 1120–1121, 2005.

[10] F. Bry, T. Furche, B. Linse, and A. Schroeder. Efficient Evaluation of n-ary Conjunctive Queries over Trees and Graphs. In *Proc. ACM Int'l. Workshop on Web Information and Data Management (WIDM)*. ACM Press, 2006.

[11] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 493–504. VLDB Endowment, 2005.

[12] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, and D. Srivastava. Index structures for matching xml twigs using relational query processors. *Data & Knowledge Engineering (DKE)*, 60(2):283–302, 2007.

[13] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and Distance Queries via 2-hop Labels. In *Proc. ACM Symposium on Discrete Algorithms*, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[14] B. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A Fast Index for Semistructured Data. In *Proc. Int. Conf. on Very Large Databases*, pages 341–350, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[15] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 365–372, New York, NY, USA, 1987. ACM.

[16] P. F. Dietz. Maintaining order in a linked list. In *Proc. ACM Symp. on Theory of Computing (STOC)*, pages 122–127, New York, NY, USA, 1982. ACM.

[17] A. Doms, T. Furche, A. Burger, and M. Schroeder. How to query the geneontology. In *Symposium on Knowledge Representation in Bioinformatics (KRBIO)*, 2005.

[18] R. Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems*, 2(3):262–278, 1977.

[19] D. R. Fulkerson and O. A. Gross. Incidence Matrices and Interval Graphs. *Pacific Journal of Mathematics*, 15(3):835–855, 1965.

[20] T. Furche, A. Weinzierl, and F. Bry. Scalable, space-optimal implementation of xcerpt single rule programs—part 1: Data model, queries, and translation. Deliverable I4-D15a, REWERSE, 2007.

[21] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[22] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[23] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. *ACM Transactions on Database Systems*, 2005.

[24] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The Complexity of XPath Query Evaluation and XML Typing. *Journal of the ACM*, 2005.

[25] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 21–32, New York, NY, USA, 1999. ACM Press.

[26] G. Gottlob, N. Leone, and F. Scarcello. The Complexity of Acyclic Conjunctive Queries. *Journal of the ACM*, 48(3):431–498, 2001.

[27] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993.

[28] T. Grust. Accelerating XPath Location Steps. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, 2002.

[29] T. Grust, M. V. Keulen, and J. Teubner. Accelerating XPath Evaluation in any RDBMS. *ACM Transactions on Database Systems*, 29(1):91–131, 2004.

[30] T. Grust, M. van Keulen, and J. Teubner. Staircase Join: Teach A Relational DBMS to Watch its (Axis) Steps. In *Proc. Int. Conf. on Very Large Databases*, 2003.

[31] M. Habib, R. McConnell, C. Paul, and L. Viennot. Lex-BFS and Partition Refinement, with Applications to Transitive Orientation, Interval Graph Recognition and Consecutive Ones Testing. *Theoretical Computer Science*, 234(1-2):59–84, 2000.

[32] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[33] S.-Y. Hsieh. The interval-merging problem. *Information Systems*, 177(2):519–524, 2007.

[34] W.-L. Hsu. PC-Trees vs. PQ-Trees. In *Proc. Int'l. Conf. on Computing and Combinatorics*, volume 2108 of *LNCS*, 2001.

[35] W.-L. Hsu. A Simple Test for the Consecutive Ones Property. *Journal of Algorithms*, 43(1):1–16, 2002.

[36] W.-L. Hsu and R. M. McConnell. PC Trees and Circular-ones Arrangements. *Theoretical Computer Science*, 296(1):99–116, 2003.

[37] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A Tree Algebra for XML. In *Proc. Int. Workshop on Database Programming Languages*, 2001.

[38] H. Jiang, H. Lu, W. Wang, and B. Ooi. Xr-tree: Indexing xml data for efficient structural join. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, pages 253–264, 2003.

[39] C. Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. *tods*, 31(4), 2006.

[40] H. Meuss and K. U. Schulz. Complete Answer Aggregates for Treelike Databases: A Novel Approach to Combine Querying and Navigation. *ACM Transactions on Information Systems*, 19(2):161–215, 2001.

[41] H. Meuss, K. U. Schulz, and F. Bry. Towards Aggregated Answers for Semistructured Data. In *Proc. Intl. Conf. on Database Theory*, pages 346–360. Springer-Verlag, 2001.

[42] D. Olteanu. *Evaluation of XPath Queries against XML Streams*. Dissertation/Ph.D. thesis, University of Munich, 2005.

[43] D. Olteanu. Forward node-selecting queries over trees. *ACM Transactions on Database Systems*, 32(1):3, 2007.

[44] D. Olteanu. SPEX: Streamed and Progressive Evaluation of XPath. *IEEE Transactions on Knowledge and Data Engineering*, 2007.

[45] D. Olteanu, T. Furche, and F. Bry. Evaluating Complex Queries against XML streams with Polynomial Combined Complexity. In *Proc. British National Conf. on Databases (BNCOD)*, pages 31–44, 2003.

[46] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Proc. EDBT Workshop on XML-Based Data Management*, volume 2490 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[47] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML Node Labels. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 903–908. ACM Press, 2004.

[48] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[49] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. Dissertation/Ph.D. thesis, University of Munich, 2004.

[50] R. Schenkel, A. Theobald, and G. Weikum. HOPI: An Efficient Connection Index for Complex XML Document Collections. In *Proc. Extending Database Technology*, 2004.

[51] M. Shalem and Z. Bar-Yossef. The space complexity of processing xml twig queries over indexed documents. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, 2008.

[52] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. ACM Symp. on Management of Data (SIGMOD)*, pages 845–856, New York, NY, USA, 2007. ACM.

[53] H. Wang, H. He2, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proc. Int'l. Conf. on Data Engineering (ICDE)*, page 75, Washington, DC, USA, 2006. IEEE Computer Society.

[54] F. Weigel, K. U. Schulz, and H. Meuss. The bird numbering scheme for xml and tree databases – deciding and reconstructing tree relations using efficient arithmetic operations. In *Proc. Int'l. XML Database Symposium (XSym)*, volume 3671 of *LNCS*, pages 49–67. Springer-Verlag, 2005.

[55] M. Yannakakis. Algorithms for Acyclic Database Schemes. In *Proc. Int. Conf. on Very Large Data Bases*, pages 82–94, 1981.