Sequenced Semantics for Temporal Multiset Relations (Extended Version)

Anton Dignös[1], Boris Glavic[2], Xing Niu[2], Michael Böhlen[3], Johann Gamper[1]

Free University of Bozen-Bolzano[1] Illinois Institute of Technology[2] University of Zurich[3]

{dignoes,gamper}@inf.unibz.it {bglavic@, xniu7@hawk.}iit.edu boehlen@ifi.uzh.ch

ABSTRACT

Sequenced semantics is widely used for evaluating queries over temporal data: temporal relations are seen as sequences of snapshot relations, and queries are evaluated at each snapshot. In this work, we demonstrate that current approaches for sequenced semantics over interval-timestamped multiset relations are subject to two bugs regarding sequenced aggregation and sequenced bag difference. We introduce a novel temporal data model based on K-relations that overcomes these bugs and prove it to correctly encode sequenced semantics. Furthermore, we present an efficient implementation of our model as a database middleware and demonstrate experimentally that our approach is competitive with native implementations and significantly outperforms such implementations on queries that involve aggregation.

PVLDB Reference Format:
. . *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/TBD

1. INTRODUCTION

Recently, there is renewed interest in temporal databases fueled by the fact that abundant storage has made long term archival of historical data feasible. This has led to the incorporation of temporal features into the SQL:2011 standard [27] which defines an encoding of temporal data associating each tuple with a validity period. We refer to such relations as *SQL period relations*. Note that SQL period relations use multiset semantics. Period relations are supported by many DBMSs, e.g., PostgreSQL [34], Teradata [44], Oracle [30], IBM DB2 [35], and MS SQLServer [29]. However, none of these systems, with the partial exception of Teradata, supports *sequenced semantics* [7], an important class of temporal queries. Given a temporal database, a non-temporal query Q interpreted under sequenced semantics returns a temporal relation that assigns to each point in time the result of evaluating Q over the snapshot of the database at this point in time. This fundamental property of sequenced semantics is known as *snapshot-reducibility* [28, 42].

Example 1.1 (Sequenced Aggregation)**.** *Consider the SQL period relation* `works` *in Figure 1a that records factory workers, their skills, and when they are on duty. The validity period of each tuple is stored in the temporal attribute* `period`*. To simplify examples, we restrict the time domain to the hours of 2018-01-01 represented as integers* 00 *to* 23*. The company requires that at least one SP worker is in the factory at any given time. This can be checked by evaluating the following query under sequenced semantics.*

```
Q_onduty :  SELECT count(*) AS cnt FROM works
            WHERE skill = 'SP'
```

Evaluated under sequenced semantics, a query returns a sequenced (time-varying) result that records when the result is valid, i.e., Q_{onduty} *returns the number of SP workers that are on duty at any given point of time. The result is shown in Figure 1b. For instance, at 08:00am two SP workers (Ann and Joe) are on duty. The query exposes several safety violations, e.g., no SP worker is on duty between 00 and 03.*

works

name	skill	period
Ann	SP	$[03, 10)$
Joe	NS	$[08, 16)$
Sam	SP	$[08, 16)$
Ann	SP	$[18, 20)$

assign

mach	skill	period
M1	SP	$[03, 12)$
M2	SP	$[06, 14)$
M3	NS	$[03, 16)$

(a) Input period relations

Q_{onduty}

cnt	period
0	$[00, 03)$
1	$[03, 08)$
2	$[08, 10)$
1	$[10, 16)$
0	$[16, 18)$
1	$[18, 20)$
0	$[20, 24)$

(b) Sequenced aggregation

$Q_{skillreq}$

skill	period
SP	$[06, 08)$
SP	$[10, 12)$
NS	$[03, 08)$

(c) Sequenced difference

Figure 1: Sequenced semantics query evaluation – highlighted tuples are erroneously omitted by approaches that exhibit the aggregation gap (AG) and bag difference (BD) bugs.

In the example above, safety violations correspond to gaps, i.e., periods of time where the aggregation's input is empty. As we will demonstrate, all approaches for sequenced semantics that we are aware of do not return results for gaps (tuples marked in red) and, therefore, violate snapshot-reducibility. Teradata [44, p.149] for instance, realized the importance of reporting results for gaps, but in contrast to snapshot-reducibility provides gaps in the presence of grouping, while omitting them otherwise. As a consequence, in our example these approaches fail to identify safety violations. We refer to this type of error as the *aggregation gap bug* (*AG bug*).

Similar to the case of aggregation, we also identify a common error related to sequenced bag difference (**EXCEPT ALL**).

Example 1.2 (Sequenced Bag Difference)**.** *Consider again Figure 1. Relation* `assign` *records machines (mach) that need to be*

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
Copyright 2018 VLDB Endowment 2150-8097/18/10... $ 10.00.
DOI: https://doi.org/TBD

assigned to workers with a specific skill over a specific period of time. For instance, the third tuple records that machine M3 requires a non-specialized (NS) worker for the time period $[03, 16)$. *To determine which skill sets are missing during which time period, we evaluate the following query under sequenced semantics:*

```
Q_skillreq :  SELECT skill FROM assign
              EXCEPT ALL
              SELECT skill FROM works
```

The result in Figure 1c indicates that one more SP worker is required during the periods $[06, 08)$ *and* $[10, 12)$.

Many approaches treat bag difference as a **NOT EXISTS** subquery, and therefore do not return a tuple t from the left input if this tuple exists in the right input (independent of their multiplicity). For instance, the two tuples for the SP workers (highlighted in red) are not returned, since there exists an SP worker at each snapshot in the `works` relation. This violates snapshot-reducibility. We refer to this type of error as the *bag difference bug* (*BD bug*).

The interval-based representation of temporal relations creates an additional problem: the encoding of a temporal query result is typically not unique. For instance, tuple $(Ann, SP, [03, 10))$ from the `works` relation in Figure 1 can equivalently be represented as two tuples $(Ann, SP, [03, 08))$ and $(Ann, SP, [08, 10))$. We refer to a method that determines how temporal data and sequenced query results are grouped into intervals as an *interval-based representation system*. A unique and predictable representation of temporal data is a desirable property, because equivalent relational algebra expressions should not lead to syntactically different result relations. This problem can be addressed by using a representation system that associates a unique encoding with each temporal database. Furthermore, overlap between multiple periods associated with a tuple and unnecessary splits of periods complicate the interpretation of data and, thus, should be avoided if possible. Given these limitations and the lack of implementations for sequenced semantics queries over bag relations, users currently resort to manually implementing such queries in SQL which is time-consuming and error-prone [39]. We address the above limitations of previous approaches for sequenced semantics and develop a framework based on the following desiderata: (i) support for set and multiset relations, (ii) snapshot-reducibility for all operations, and (iii) a unique interval-based encoding of temporal relations. We address these desiderata using a three-level approach. Note that we focus on data with a single time dimension, but are oblivious to whether this is transaction time or valid time. First, we introduce an *abstract model* that supports both sets and multisets, and by definition is snapshot-reducible. This model, however, uses a verbose encoding of temporal data and, thus, is not practical. Afterwards, we develop a more compact *logical model* as a representation system, where the complete temporal history of all equivalent tuples from the abstract model is stored in an annotation attached to one tuple. The abstract and the logical models leverage the theory of K-relations, which are a general class of annotated relations that cover both set and multiset relations. For our *implementation*, we use SQL over period relations to ensure compatibility with SQL:2011 and existing DBMSs. We prove the equivalence between the three layers (i.e., the abstract model, the logical model and the implementation) and show that the logical model determines a unique interval-encoding for the implementation and a correct rewriting scheme for queries over this encoding.

Our main technical contributions are:

- *Abstract model*: We introduce *sequenced K-relations* as a generalization of sequenced set and multiset relations. These relations are by definition snapshot-reducible.
- *Logical model*: We define an interval-based representation, termed *period K-relations*, and prove that these relations are a compact and unique representation system for sequenced semantics over sequenced K-relations. We show this for the full relational algebra plus aggregation ($\mathcal{RA}^{agg}$).
- We achieve a unique encoding of temporal data as period K-relations by generalizing set-based coalescing [10].
- We demonstrate that the multiset version of period K-relations can be encoded as *SQL period relations*, a common interval-based model in DBMSs, and how to translate queries with sequenced semantics over period K-relations into SQL.
- We implement our approach as a database middleware and present optimizations that eliminate redundant coalescing steps. Our experimental shows that we do not need to sacrifice performance to achieve correctness.

2. RELATED WORK

Temporal Query Languages. There is a long history of research on temporal query languages [6, 22]. Many temporal query languages including TSQL2 [38, 40], ATSQL2 (Applied TSQL2) [8], IXSQL [28], ATSQL [9], and SQL/TP [46] support sequenced semantics. In this paper, we provide a general framework that can be used to correctly implement sequenced semantics over period set and multiset relations for any language.

Interval-based Approaches for Sequenced Semantics. In the following, we discuss interval-based approaches for sequenced semantics. Table 1 shows for each approach whether it supports multisets, whether it is free of the aggregation gap and bag difference bugs, and whether its interval-based encoding of a sequenced query result is unique. An N/A indicates that the approach does not support the operation for which this type of bug can occur or the semantics of this operation is not defined precisely enough to judge its correctness. Note that while temporal query languages may be defined to apply sequenced semantics and, thus, by definition are snapshot-reducible, (the specification of) their implementation might fail to be snapshot-reducible. In the following discussion of the temporal query languages in Table 1, we refer to their semantics as provided in the referenced publication(s).

Interval preservation (ATSQL) [9, Def. 2.10] is a representation system for SQL period relations (multisets) that tries to preserve the intervals associated with input tuples, i.e., fragments of all intervals (including duplicates) associated with the input tuples "survive" in the output. Interval preservation is snapshot-reducible for multiset semantics for positive relational algebra [36] (selection, projection, join, and union), but exhibits the aggregation gap and bag difference bug. Moreover, the period encoding of a query result is not unique as it depends both on the query and the input representation. *Teradata* [44] is a commercial DBMS that supports sequenced operators using ATSQL's statement modifiers. The implementation is based on query rewriting [2] and does not support difference. Teradata's implementation exhibits the aggregation gap bug. Since the application of coalescing is optional, the encoding of snapshot relations as period relations is not unique. *Change preservation* [18, Def. 3.4] determines the interval boundaries of a query result tuple t based on the maximal interval for which there is no change in the input. To track changes, it employs the lineage provenance model in [16] and the PI-CS model in [18]. The approach uses timestamp adjustment in combination with traditional database operators, but does not provide a unique encoding, exhibits the AG bug, and only supports set semantics. Our work addresses these issues and significantly generalizes this approach, in

Table 1: Interval-based approaches for sequenced semantics.

Approach	Multisets	AG bug free	BD bug free	Unique encoding
Interval preservation [9] (ATSQL)	✓	×	×	×
Teradata [44]	✓	×	N/A	×[1]
Change preservation [16, 18]	×	×	N/A	×
TSQL2 [38, 40, 42]	×	N/A	N/A	✓
ATSQL2 [8]	✓	N/A	×	×
TimeDB [43] (ATSQL2)	✓	N/A	×	×
SQL/Temporal [41]	✓	×	×	×
SQL/TP [46][2]	✓	✓	✓	×
Our approach	✓	✓	✓	✓

particular by supporting bag semantics. *TSQL2* [38, 40, 42] implicitly applies coalescing [10] to produce a unique representation. Thus, it only supports set semantics, and it does not support aggregation. Snodgrass et al. [41] present a validtime extension of *SQL/Temporal* and an algebra with sequenced semantics. The algebra supports multisets, but exhibits both the aggregation gap and bag difference bug. Since intervals from the input are preserved where possible, the interval representation of a snapshot relation is not unique. *TimeDB* [43] is an implementation of ATSQL2 [8]. It uses a semantics for bag difference and intersection that is not snapshot-reducible (see [43, pp. 63]). Our approach is the first that supports set and multiset relations, is resilient against the two bugs, and specifies a unique interval-encoding.

Non-sequenced Temporal Queries. Non-sequenced temporal query languages, such as IXSQL [28] and SQL/TP [46], do not explicitly support sequenced semantics. Nevertheless, we review these languages here since they allow to express queries with sequenced semantics. SQL/TP [46] introduces a point-wise semantics for temporal queries [12, 45], where time is handled as a regular attribute. Intervals are used as an efficient encoding of time points, and a normalization operation is used to split intervals. The language supports multisets and a mechanism to manually produce sequenced semantics. However, sequenced semantics queries are specified as the union of non-temporal queries over snapshots. Even if such subqueries are grouped together for adjacent time points where the non-temporal query's result is constant this still results in a large number of subqueries to be executed. Even worse, the number of subqueries that is required is data dependent. Also, the interval-based encoding is not unique, since time points are grouped into intervals depending on query syntax and encoding of the input. While this has no effect on the semantics since SQL/TP queries cannot distinguish between different interval-based encodings of a temporal database, it might be confusing to users that observe different query results for equivalent queries/inputs.

Implementations of Temporal Operators. A large body of work has focused on the implementation of individual temporal algebra operators such as joins [11,17,32] and aggregation [5,31,33]. Some exceptions supporting multiple operators are [13, 18, 25]. These approaches introduce efficient evaluation algorithms for a particular semantics of a temporal algebra operator. Our approach can utilize efficient operator implementations as long as (i) their semantics is compatible with our interval-based encoding of sequenced query results and (ii) they are snapshot-reducible.

Coalescing. Coalescing produces a unique representation of a *set* semantics temporal database. Böhlen et al. [10] study optimizations for coalescing that eliminate unnecessary coalescing operations. Zhou et al. [47] and [1] use analytical functions to efficiently implement coalescing in SQL. We generalize coalescing to K-relations to define a unique encoding of interval-based temporal relations, including *multiset* relations. Similar to [10], we remove unnecessary K-coalescing steps and, similar to [47], we use OLAP functions for efficient implementation.

Temporality in Annotated Databases. Kostiley et al. [26] is to the best of our knowledge the only previous approach that uses semiring annotations to express temporality. The authors define a semiring whose elements are sets of time points. This approach is limited to set semantics, and no interval-based encoding was presented. The LIVE system [15] combines provenance and uncertainty annotations with versioning. The system uses interval timestamps, and query semantics is based on snapshot-reducibility [15, Def. 2]. However, computing the intervals associated with a query result requires provenance to be maintained for every query result.

3. SOLUTION OVERVIEW

In this section, we give an overview of our three-level framework, which is illustrated in Figure 2.

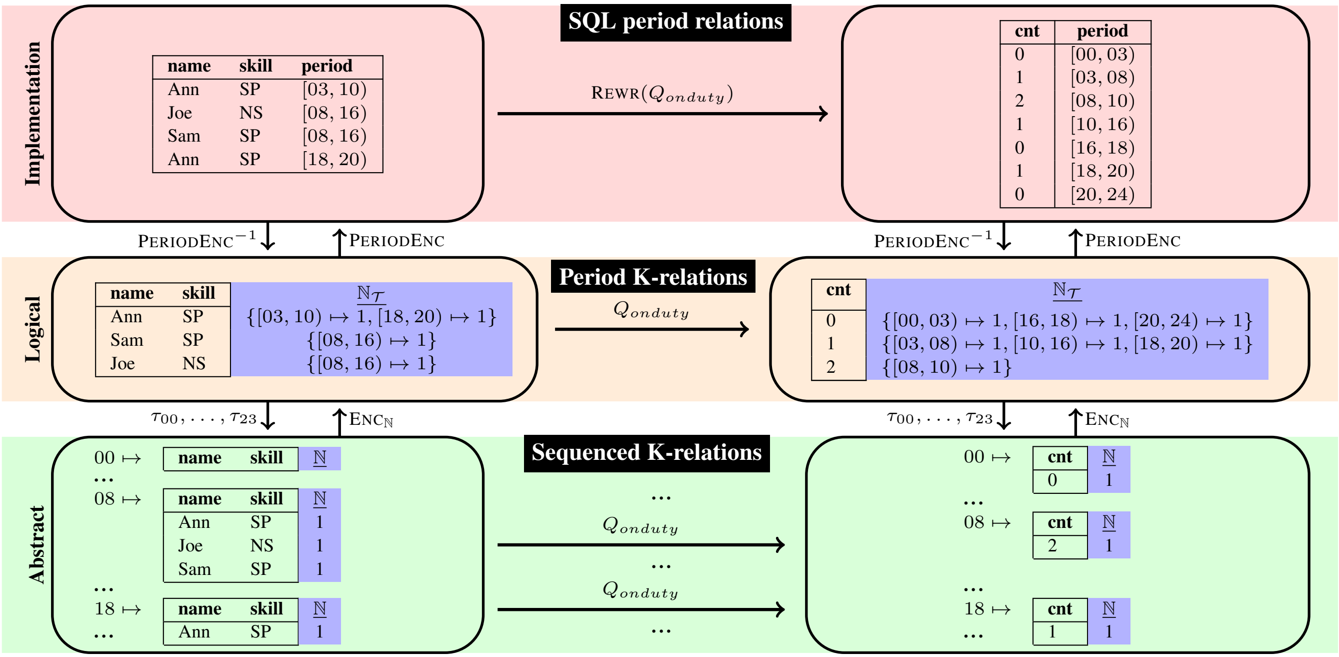
Abstract model – Sequenced K-relations. As an *abstract model* we use sequenced relations which map time points to snapshots. Queries over such relations are evaluated over each snapshot, which trivially satisfies snapshot-reducibility. To support both sets and multisets, we introduce *sequenced K-relations* [20], which are sequenced relations where each snapshot is a K-relation. In a K-relation, each tuple is annotated with an element from a domain K. For example, relations annotated with elements from the semiring $\mathbb{N}$ (natural numbers) correspond to multiset semantics. The result of a sequenced query Q over a sequenced K-relation is the result of evaluating Q over the K-relation at each time point.

Example 3.1 (Abstract Model)**.** *Figure 2 (bottom) shows the snapshots at times `00`, `08`, and `18` of an encoding of the running example as sequenced $\mathbb{N}$-relations. Each snapshot is an $\mathbb{N}$-relation where tuples are annotated with their multiplicity (shown with shaded background). For instance, the snapshot at time `08` has three tuples, each with multiplicity 1. The result of query Q_{onduty} is shown on the bottom right. Every snapshot in the result is computed by running Q_{onduty} over the corresponding snapshot in the input. For instance, at time `08` there are two SP workers, i.e., $cnt = 2$.*

Logical Model – Period K-relations. We introduce period K-relations as a *logical model*, which merges equivalent tuples over all snapshots from the abstract model into one tuple. In a *period K-relation*, every tuple is annotated with a *temporal K-element* that is a unique interval-based representation for all time points of the merged tuples from the abstract model. We define a class of semirings called *period semirings* whose elements are temporal K-elements. Specifically, for any semiring K we can construct a period semiring $K_{\mathcal{T}}$ whose annotations are temporal K-elements. For instance, $\mathbb{N}_{\mathcal{T}}$ is the period semiring corresponding to semiring $\mathbb{N}$ (multisets). We define necessary conditions for an interval-based model to correctly encode sequenced K-relations and prove that period K-relations fullfil these conditions. Specifically, we call an interval-based model a *representation system* iff the encoding of every sequenced K-relation R is (i) unique and (ii) snapshot-equivalent to R. Furthermore, (iii) queries over encodings are snapshot-reducible.

[1] Optionally, coalescing (`NORMALIZE ON` in Teradata) can be applied to get a unique encoding at the cost of loosing multiplicities.

[2] Sequenced semantics can be expressed, but this is inefficient



name	skill	period
Ann	SP	$[03, 10)$
Joe	NS	$[08, 16)$
Sam	SP	$[08, 16)$
Ann	SP	$[18, 20)$

cnt	period
0	$[00, 03)$
1	$[03, 08)$
2	$[08, 10)$
1	$[10, 16)$
0	$[16, 18)$
1	$[18, 20)$
0	$[20, 24)$

name	skill	$\mathbb{N}_{\mathcal{T}}$
Ann	SP	$\{[03, 10) \mapsto 1, [18, 20) \mapsto 1\}$
Sam	SP	$\{[08, 16) \mapsto 1\}$
Joe	NS	$\{[08, 16) \mapsto 1\}$

cnt	$\mathbb{N}_{\mathcal{T}}$
0	$\{[00, 03) \mapsto 1, [16, 18) \mapsto 1, [20, 24) \mapsto 1\}$
1	$\{[03, 08) \mapsto 1, [10, 16) \mapsto 1, [18, 20) \mapsto 1\}$
2	$\{[08, 10) \mapsto 1\}$

$00 \mapsto$

name	skill	$\mathbb{N}$

$08 \mapsto$

name	skill	$\mathbb{N}$
Ann	SP	1
Joe	NS	1
Sam	SP	1

$18 \mapsto$

name	skill	$\mathbb{N}$
Ann	SP	1

$00 \mapsto$

cnt	$\mathbb{N}$
0	1

$08 \mapsto$

cnt	$\mathbb{N}$
2	1

$18 \mapsto$

cnt	$\mathbb{N}$
1	1

Figure 2: Overview of our approach. Our *abstract model* is *sequenced K-relations* and nontemporal queries over snapshots (sequenced semantics). Our *logical* model is *period K-relations* and queries corresponding to the abstract model's sequenced queries. Our *implementation* uses *SQL period relations* and rewritten non-temporal queries implementing the other model's sequenced queries. Each model is associated with transformations to the other models which commute with queries (modulo the rewriting $\textsc{Rewr}$ when mapping to the implementation).

Example 3.2 (Logical Model)**.** *Figure 2 (middle) shows an encoding of the running example as period K-relations. For instance, all tuples (Ann, SP) from the abstract model are merged into one tuple in the logical model with annotation $\{[03, 10) \mapsto 1, [18, 20) \mapsto 1\}$, because at each time point during $[03, 10)$ and $[18, 20)$ a tuple (Ann, SP) with multiplicity 1 exists. In Section 4.2, we will introduce a mapping $\textsc{Enc}_{\mathbb{N}}$ from sequenced $\mathbb{N}$ to $\mathbb{N}_{\mathcal{T}}$-relations and the time slice operator τ_T which restores an the snapshot at time T.*

Implementation – SQL Period Relations. To ensure compatibility with the SQL standard, we use *SQL period relations* in our *implementation* and translate sequenced semantics queries into SQL queries over these period relations. For this we define an encoding of $\mathbb{N}_{\mathcal{T}}$-relations as SQL period relations ($\textsc{PeriodEnc}$) together with a rewriting scheme for queries ($\textsc{Rewr}$).

Example 3.3 (Implementation)**.** *Consider the SQL period relations shown on the top of Figure 2. Each interval-annotation pair of a temporal $\mathbb{N}$-element in the logical model is encoded as a separate tuple in the implementation. For instance, the annotation of tuple (Ann, SP) from the logical model is encoded as two tuples, each of which records one of the two intervals from this annotation*

We present an implementation of our framework as a database middleware that exposes sequenced semantics as a new language feature in SQL and rewrites sequenced queries into SQL queries over SQL period relations. That is, we directly evaluate sequenced queries over data stored natively as period relations.

4. SEQUENCED K-RELATIONS

We first review background on the semiring annotation framework (K-relations). Afterwards, we define *sequenced K-relations* as our abstract model and sequenced semantics for this model. Importantly, queries over sequenced K-relations are snapshot-reducible by construction. Finally, we state requirements for a logical model to be a representation system for this abstract model.

4.1 K-relations

In a K-relation [20], every tuple is annotated with an element from a domain K of a commutative semiring K. A structure $(K, +_K, \cdot_K, 0_K, 1_K)$ over a set K with binary operations $+_K$ and $\cdot_K$ is a commutative semiring iff (i) addition and multiplication are commutative, associative, and have a neutral element (0_K and 1_K, respectively); (ii) multiplication distributes over addition; and (iii) multiplication with zero returns zero. Abusing notation, we will use K to denote both a semiring structure as well as its domain.

Consider a universal countable domain $\mathcal{U}$ of values. An n-ary K-relation R over $\mathcal{U}$ is a (total) function that maps tuples (elements from $\mathcal{U}^n$) to elements from K with the convention that tuples mapped to 0_K are not in the relation. Furthermore, we require that $R(t) \neq 0_K$ only holds for finitely many t. Two semirings are of particular interest to us: The semiring $(\mathbb{B}, \vee, \wedge, false, true)$ with elements *true* and *false* using $\vee$ as addition and $\wedge$ as multiplication corresponds to set semantics. The semiring $(\mathbb{N}, +, \cdot, 0, 1)$ of natural numbers with standard arithmetics corresponds to multisets.

The operators of the positive relational algebra [36] ($\mathcal{RA}^+$) over K-relations are defined by applying the $+_K$ and $\cdot_K$ operations of the semiring K to input annotations. Intuitively, the $+_K$ and $\cdot_K$ operations of the semiring correspond to the alternative and conjunctive use of tuples, respectively. For instance, if an output tuple t is produced by joining two input tuples annotated with k and k', then the tuple t is annotated with $k \cdot_K k'$. Below we provide the standard definition of $\mathcal{RA}^+$ over K-relations [20]. For a tuple t, we use $t.A$ to denote the projection of t on a list of projection expressions A and $t[R]$ to denote the projection of t on the attributes of relation R. For a condition θ and tuple t, $\theta(t)$ denotes a function that returns 1_K if $t \models \theta$ and 0_K otherwise.

Definition 4.1 ($\mathcal{RA}^+$ over K-relations)**.** *Let K be a semiring, R, S denote K-relations, t, u denote tuples of appropriate arity, and*

$k \in K$. $\mathcal{RA}^+$ *on* K*-relations is defined as:*

$$\sigma_\theta(R)(t) = R(t) \cdot \theta(t) \qquad \textit{(selection)}$$

$$\Pi_A(R)(t) = \sum\nolimits_{u:u.A=t} R(u) \qquad \textit{(projection)}$$

$$(R \bowtie S)(t) = R(t[R]) \cdot S(t[S]) \qquad \textit{(join)}$$

$$(R \cup S)(t) = R(t) + S(t) \qquad \textit{(union)}$$

We will make use of homomorphisms, functions from the domain of a semiring K_1 to the domain of a semiring K_2 that commute with the semiring operations. Since $\mathcal{RA}^+$ over K-relations is defined in terms of these operations, it follows that semiring homomorphisms commute with queries, as was proven in [20].

Definition 4.2 (Homomorphism)**.** *A mapping* $h : K_1 \rightarrow K_2$ *is called a homomorphism iff for all* $k, k' \in K_1$*:*

$$h(0_{K_1}) = 0_{K_2} \qquad h(1_{K_1}) = 1_{K_2}$$

$$h(k +_{K_1} k') = h(k) +_{K_2} h(k') \qquad h(k \cdot_{K_1} k') = h(k) \cdot_{K_2} h(k')$$

Example 4.1. *Consider the* $\mathbb{N}$*-relations shown below which are non-temporal versions of our running example. Query* $Q = \Pi_{mach}(\ works \bowtie assign)$ *returns machines for which there are workers with the right skill to operate the machine. Under multiset semantics we expect M1 to occur in the result of Q with multiplicity* 8 *since* $(M1, SP)$ *joins with* $(Pete, SP)$ *and with* (Bob, SP)*. Evaluating the query in* $\mathbb{N}$ *yields the expected result by multiplying the annotations of these join partners. Given the* $\mathbb{N}$ *result of the query, we can compute the result of the query under set semantics by applying a homomorphism* h *which maps all non-zero annotations to true and* 0 *to false. For example, for result* $(M1)$ *we get* $h(8) = true$*, i.e., this tuple is in the result under set semantics.*

works

name	**skill**	$\mathbb{N}$
Pete	SP	1
Bob	SP	1
Alice	NS	1

assign

mach	**skill**	$\mathbb{N}$
M1	SP	4
M2	NS	5

Result

A	$\mathbb{N}$
M1	$1 \cdot 4 + 1 \cdot 4 = 8$
M2	$5 \cdot 1 = 5$

4.2 Sequenced K-relations

We now formally define sequenced K-relations, sequenced semantics over such relations, and then define representation systems. We assume a totally ordered and finite domain $\mathbb{T}$ of time points and use $\leq_{\mathbb{T}}$ to denote its order. T_{min} and T_{max} denote the minimal and maximal (exclusive) time point in $\mathbb{T}$ according to $\leq_{\mathbb{T}}$, respectively. We use $T + 1$ to denote the successor of $T \in \mathbb{T}$ according to $\leq_{\mathbb{T}}$.

A sequenced K-relation over a relation schema $\mathbf{R}$ is a function $\mathbb{T} \rightarrow \mathcal{R}_{K,\mathbf{R}}$, where $\mathcal{R}_{K,\mathbf{R}}$ is the set of all K-relations with schema $\mathbf{R}$. Sequenced K-databases are defined analog. We use $\mathcal{DB}_{\mathbb{T},K}$ to denote the set of all sequenced K-databases for time domain $\mathbb{T}$.

Definition 4.3 (Sequenced K-relation)**.** *Let* K *be a commutative semiring and* $\mathbf{R}$ *a relation schema. A sequenced* K*-relation* R *is a function* $R : \mathbb{T} \rightarrow \mathcal{R}_{K,\mathbf{R}}$*.*

For instance, a sequenced $\mathbb{N}$-relation is shown in Figure 2 (bottom). Given a sequenced K-relation, we use the *timeslice operator* [23] to access its state (snapshot) at a time point T:

$$\tau_T(R) = R(T)$$

The evaluation of a query Q over a sequenced database (set of sequenced relations) D under *sequenced semantics* returns a sequenced relation $Q(D)$ that is constructed as follows: for each time point $T \in \mathbb{T}$ we have $Q(D)(T) = Q(D(T))$. Thus, sequenced temporal queries over sequenced K-relations behave like queries over K-relations for each snapshot, i.e., their semantics is uniquely determined by the semantics of queries over K-relations.

Definition 4.4 (Sequenced Semantics)**.** *Let* D *be a sequenced* K*-database and* Q *be a query. The result* $Q(D)$ *of* Q *over* D *is a sequenced* K*-relation that is defined point-wise as follows:*

$$\forall T \in \mathbb{T} : Q(D)(T) = Q(\tau_T(D))$$

For example, consider the sequenced $\mathbb{N}$-relation shown at the bottom of Figure 2 and the evaluation of Q_{onduty} under sequenced semantics as also shown in this figure. Observe how the query result is computed by evaluating Q_{onduty} over each snapshot individually using multiset ($\mathbb{N}$) query semantics. Furthermore, since $\tau_T(Q(R)) = Q(R)(T)$, per the above definition, the timeslice operator commutes with queries: $\tau_T(Q(R)) = Q(\tau_T(R))$. This property is *snapshot-reducibility*.

4.3 Representation Systems

To compactly encode sequenced K-relations, we study representation systems that consist of a set of representations $\mathcal{E}$, a function $\textsc{Enc} : \mathcal{E} \rightarrow \mathcal{DB}_{\mathbb{T},K}$ which associates an encoding in $\mathcal{E}$ with the sequenced K-database it represents, and a timeslice operator τ_T which extracts the snapshot at time T from an encoding. If $\textsc{Enc}$ is injective, then we use $\textsc{Enc}^{-1}(D)$ to denote the unique encoding associated with D. We use τ to denote the timeslice over both sequenced databases and representations. It will be clear from the input which operator τ refers to. For such a representation system, we consider two encodings D_1 and D_2 from $\mathcal{E}$ to be *snapshot-equivalent* [21] (written as $D_1 \sim D_2$) if they encode the same sequenced K-database. Note that this is the case if they encode the same snapshots, i.e., iff for all $T \in \mathbb{T}$ we have $\tau_T(D_1) = \tau_T(D_2)$. For a representation system to behave correctly, the following conditions have to be met: 1) **uniqueness**: for each snapshot K-database D there exists a unique element from $\mathcal{E}$ representing D; 2) **snapshot-reducibility**: the timeslice operator commutes with queries; and 3) **snapshot-preservation**: the encoding function $\textsc{Enc}$ preserves the snapshots of the input.

Definition 4.5 (Representation System)**.** *We call a triple* $(\mathcal{E}, \textsc{Enc}, \tau)$ *a* representation system *for sequenced* K*-databases with regard to a class of queries* $\mathcal{C}$ *iff for every sequenced database* D*, encodings* E*,* $E' \in \mathcal{E}$*, time point* T*, and query* $Q \in \mathcal{C}$ *we have*

1. $\textsc{Enc}(E) = \textsc{Enc}(E') \Rightarrow E = E'$ *(uniqness)*
2. $\tau_T(Q(E)) = Q(\tau_T(E))$ *(snapshot-reducibility)*
3. $\textsc{Enc}(E) = D \Rightarrow \tau_T(E) = \tau_T(D)$ *(snapshot-preservation)*

5. TEMPORAL K-ELEMENTS

We now introduce temporal K-elements that are the annotations we use to define our logical model (representation system). Temporal K-elements record, using an interval-based encoding, how the K-annotation of a tuple in a sequenced K-relation changes over time. We introduce a unique normal form for temporal K-elements based on a generalization of coalescing [10].

5.1 Defining Temporal K-elements

To define temporal K-elements, we need to introduce some background on intervals. Given the time domain $\mathbb{T}$ and its associated total order $\leq_{\mathbb{T}}$, an interval $I = [T_b, T_e)$ is a pair of time points from $\mathbb{T}$, where $T_b <_{\mathbb{T}} T_e$. Interval I represents the set of contiguous time points $\{T \mid T \in \mathbb{T} \wedge T_b \leq_{\mathbb{T}} T <_{\mathbb{T}} T_e\}$. For an interval $I = [T_b, T_e)$ we use I^+ to denote T_b and I^- to denote T_e. We use $I, I', I_1, \ldots$ to represent intervals. We define a relation $adj(I_1, I_2)$ that contains all interval pairs that are adjacent: $adj(I_1, I_2) \Leftrightarrow ({I_1}^- = {I_2}^+) \vee ({I_2}^- = {I_1}^+)$. We will implicitly understand set operations, such as $t \in I$ or $I_1 \subseteq I_2$, to be interpreted over the set of points represented by an interval. Furthermore, $I \cap I'$ denotes the interval that covers precisely the intersection of the sets of time points defined by I and I' and $I \cup I'$ denotes their union (only well-defined if $I \cap I' \neq \emptyset$ or $adj(I, I')$). For convenience, we define $I \cup I' = \emptyset$ iff $I \cap I' = \emptyset \land \neg adj(I, I')$. We use $\mathbb{I}$ to denote the set of all intervals over $\mathbb{T}$.

Definition 5.1 (Temporal K-elements)**.** *Given a semiring K, a* temporal K-element *$\mathcal{T}$ is a function $\mathbb{I} \to K$. We use $\mathbb{TE}_K$ to denote the set of all such temporal elements for K.*

We represent temporal K-elements as sets of input-output pairs. Intervals that are not explicitly mentioned are mapped to 0_K.

Example 5.1. *Reconsider our running example with $\mathbb{T} = \{00, \dots, 23\}$. The history of the annotation of tuple $t =$ (Ann, SP) from the* works *relation is as shown in Figure 2 (middle). For sake of the example, we change the multiplicity of this tuple to* 3 *during* $[03, 09)$ *and* 2 *during* $[18, 20)$. *This information is encoded as the temporal $\mathbb{N}$-element $\mathcal{T}_1 = \{[03, 09) \mapsto 3, [18, 20) \mapsto 2\}$.*

Note that a temporal K-element $\mathcal{T}$ may map overlapping intervals to non-zero elements of K. We assign the following semantics to overlap: the annotation at a time point T recorded by $\mathcal{T}$ is the sum of the annotations assigned to intervals containing T. For instance, the annotation at time 04 for the $\mathbb{N}$-element $\mathcal{T} = \{[00, 05) \mapsto 2, [04, 05) \mapsto 1\}$ would be $2 + 1 = 3$. To extract the annotation valid at time T from a temporal K-element $\mathcal{T}$, we define a timeslice operator for temporal K-elements as follows:

$$\tau_T(\mathcal{T}) = \sum_{T \in I} \mathcal{T}(I) \qquad \text{(timeslice operator)}$$

Given two temporal K-elements $\mathcal{T}_1$ and $\mathcal{T}_2$, we would like to know if they represent the same history of annotations. For that, we define *snapshot-equivalence* ($\sim$) for temporal K-elements:

$$\mathcal{T}_1 \sim \mathcal{T}_2 \Leftrightarrow \forall T \in \mathbb{T} : \tau_T(\mathcal{T}_1) = \tau_T(\mathcal{T}_2) \quad \text{(snapshot-equivalence)}$$

5.2 A Normal Form Based on K-Coalescing

The encoding of the annotation history of a tuple as a temporal K-element is typically not unique.

Example 5.2. *Reconsider the temporal $\mathbb{N}$-element $\mathcal{T}_1$ from Example 5.1. Recall that intervals not shown are mapped to* 0. *The $\mathbb{N}$-elements shown below are snapshot-equivalent to $\mathcal{T}_1$.*

$$\mathcal{T}_2 = \{[03, 09) \mapsto 1, [03, 06) \mapsto 2, [06, 09) \mapsto 2, [18, 19) \mapsto 2\}$$

$$\mathcal{T}_3 = \{[03, 05) \mapsto 3, [05, 09) \mapsto 3, [18, 19) \mapsto 2\}$$

To be able to build a representation system based on temporal K-elements we need a unique way to encode the annotation history of a tuple as a temporal K-element (condition 1 of Definition 4.5). That is, we need to define a normal form that is unique for snapshot-equivalent temporal K-elements. To this end, we generalize *coalescing*, which was defined for temporal databases with set semantics in [10, 37]. The generalized form, which we call K-coalescing, coincides with standard coalescing for semiring $\mathbb{B}$ (set semantics) and, for any semiring K, yields a unique encoding.

K-coalescing creates maximal intervals of contiguous time points with the same annotation. The output is a temporal K-element such that (a) no two intervals mapped to a non-zero element overlap and (b) adjacent intervals assigned to non-zero elements are guaranteed to be mapped to different annotations. To determine such intervals, we define annotation changepoints, time points T where the annotation of a temporal K-element differs from the annotation at $T - 1$, i.e., $\tau_T(\mathcal{T}) \neq \tau_{T-1}(\mathcal{T})$). It will be convenient to also consider T_{min} as an annotation changepoint.

sal	period
50k	$[1, 13)$
30k	$[3, 13)$
30k	$[3, 10)$
40k	$[11, 13)$

$$\mathcal{T}_{50k} = \{[1, 13) \mapsto 1\}$$

$$\mathcal{T}_{30k} = \{[3, 10) \mapsto 1, [3, 13) \mapsto 1\}$$

$$\mathcal{T}_{40k} = \{[11, 13) \mapsto 1\}$$

Figure 3: Example period multiset relation S and temporal $\mathbb{N}$-elements encoding the history of tuples.

Definition 5.2 (Annotation Changepoint)**.** *Given a temporal K-element $\mathcal{T}$, a time point T is called a* changepoint *in $\mathcal{T}$ if one of the following conditions holds:*

- $T = T_{min}$ *(smallest time point)*
- $\tau_{T-1}(\mathcal{T}) \neq \tau_T(\mathcal{T})$ *(change of annotation)*

We use $CP(\mathcal{T})$ to denote the set of all annotation changepoints for $\mathcal{T}$. Furthermore, we define $CPI(\mathcal{T})$ to be the set of all intervals that consist of consecutive change points:

$$\begin{aligned} CPI(\mathcal{T}) = \{[T_b, T_e) \mid T_b <_{\mathbb{T}} T_e \land T_b \in CP(\mathcal{T}) \land \\ (T_e \in CP(\mathcal{T}) \lor T_e = T_{max}) \land \\ \nexists T' \in CP(\mathcal{T}) : T_b <_{\mathbb{T}} T' <_{\mathbb{T}} T_e\} \end{aligned}$$

In Definition 5.2, $CPI(\mathcal{T})$ computes maximal intervals such that the annotation assigned by $\mathcal{T}$ to each point in such an interval is constant. In the coalesced representation of $\mathcal{T}$ only such intervals are mapped to non-zero annotations.

Definition 5.3 (K-Coalesce)**.** *Let $\mathcal{T}$ be a temporal K-element. We define K-coalescing $\mathcal{C}_K$ as a function $\mathbb{TE}_K \to \mathbb{TE}_K$:*

$$\mathcal{C}_K(\mathcal{T})(I) = \begin{cases} \tau_{I^+}(\mathcal{T}) & \text{if } I \in CPI(\mathcal{T}) \\ 0_K & \text{otherwise} \end{cases}$$

We use $\mathbb{TEC}_K$ to denote all normalized temporal K-elements, i.e., elements $\mathcal{T}$ for which $\mathcal{C}_K(\mathcal{T}) = \mathcal{T}'$ for some $\mathcal{T}'$.

Example 5.3. *Consider the SQL period relation shown in Figure 3. The temporal $\mathbb{N}$-elements encode the history of tuples* $(30k)$, $(40k)$ *and* $(50k)$. *Note that $\mathcal{T}_{30k}$ is not coalesced since the two non-zero intervals of this $\mathbb{N}$-element overlap. Applying $\mathbb{N}$-coalesce we get:*

$$\mathcal{C}_{\mathbb{N}}(\mathcal{T}_{30k}) = \{[3, 10) \mapsto 2, [10, 13) \mapsto 1\}$$

That is, this tuple occurs twice within the time interval $[3, 10)$ *and once in* $[10, 13)$, *i.e., it has annotation changepoints* 3, 10, *and* 14. *Interpreting the same relation under set semantics (semiring $\mathbb{B}$), the history of* $(30k)$ *can be encoded as a temporal $\mathbb{B}$-element $\mathcal{T}_{30k}{}' = \{[3, 10) \mapsto true, [3, 13) \mapsto true\}$. Applying $\mathbb{B}$-coalesce:*

$$\mathcal{C}_{\mathbb{B}}(\mathcal{T}_{30k}{}') = \{[3, 13) \mapsto true\}$$

That is, this tuple occurs (is annotated with true*) within the time interval* $[3, 13)$ *and its annotation changepoints are* 3 *and* 14.

We now prove several important properties of the K-coalesce operator establishing that $\mathbb{TEC}_K$ (coalesced temporal K-elements) is a good choice for a normal form of temporal K-elements.

Lemma 5.1. *Let K be a semiring and $\mathcal{T}$, $\mathcal{T}_1$ and $\mathcal{T}_2$ temporal K-elements. We have:*

$$\mathcal{C}_K(\mathcal{C}_K(\mathcal{T})) = \mathcal{C}_K(\mathcal{T}) \qquad \text{(idempotence)}$$

$$\mathcal{T}_1 \sim \mathcal{T}_2 \Leftrightarrow \mathcal{C}_K(\mathcal{T}_1) = \mathcal{C}_K(\mathcal{T}_2) \qquad \text{(uniqueness)}$$

$$\mathcal{T} \sim \mathcal{C}_K(\mathcal{T}) \qquad \text{(equivalence preservation)}$$

Proof. All proofs are shown in Appendix A. □

6. PERIOD SEMIRINGS

Having established a unique normal form of temporal K-elements, we now proceed to define period semirings as our logical model. The elements of a period semiring are temporal K-elements in normal form. We prove that these structures are semirings and ultimately that relations annotated with period semirings form a representation system for sequenced K-relations for $\mathcal{RA}^+$. In Section 7, we then prove them to also be a representation system for $\mathcal{RA}^{agg}$, i.e., queries involving difference and aggregation.

When defining the addition and multiplication operations and their neutral elements in the semiring structure of temporal K-elements, we have to ensure that these definitions are compatible with semiring K on snapshots. Furthermore, we need to ensure that the output of these operations is guaranteed to be K-coalesced. The latter can be ensured by applying K-coalesce to the output of the operation. For addition, snapshot reducibility is achieved by pointwise addition (denoted as $+_{K_\mathcal{P}}$) of the two functions that constitute the two input temporal K-elements. That is, for each interval I, the function that is the result of the addition of temporal K-elements $\mathcal{T}_1$ and $\mathcal{T}_2$ assigns to I the value $\mathcal{T}_1(I) +_K \mathcal{T}_2(I)$. For multiplication, the multiplication of two K-elements assigned to an overlapping pair of intervals I_1 and I_2 is valid during the intersection of I_1 and I_2. Since both input temporal K-elements may assign non-zero values to multiple intervals that have the same overlap, the resulting K-value at a point T would be the sum over all pairs of overlapping intervals. We denote this operation as $\cdot_{K_\mathcal{P}}$. Since $+_{K_\mathcal{P}}$ and $\cdot_{K_\mathcal{P}}$ may return a temporal K-element that is not coalesced, we define the operations of our structures to apply $\mathcal{C}_K$ to the result of $+_{K_\mathcal{P}}$ and $\cdot_{K_\mathcal{P}}$. The zero element of the temporal extension of K is the temporal K-element that maps all intervals to 0 and the 1 element is the temporal element that maps every interval to 0_K except for $[T_{min}, T_{max})$ which is mapped to 1_K.

Definition 6.1 (Period Semiring)**.** *For a time domain $\mathbb{T}$ with minimum T_{min} and maximum T_{max} and a semiring K, the period semiring $K_\mathcal{T}$ is defined as:*

$$K_\mathcal{T} = (\mathbb{TEC}_K, +_{K_\mathcal{T}}, \cdot_{K_\mathcal{T}}, 0_{K_\mathcal{T}}, 1_{K_\mathcal{T}})$$

where for $k, k' \in \mathbb{TEC}_K$ and :

$$\forall I \in \mathbb{I} : 0_{K_\mathcal{T}}(I) = 0_K \quad 1_{K_\mathcal{T}}(I) = \begin{cases} 1_K & \text{if } I = [T_{min}, T_{max}) \\ 0_K & \text{otherwise} \end{cases}$$

$$k +_{K_\mathcal{T}} k' = \mathcal{C}_K(k +_{K_\mathcal{P}} k')$$

$$\forall I \in \mathbb{I} : (k +_{K_\mathcal{P}} k')(I) = k(I) +_K k'(I)$$

$$k \cdot_{K_\mathcal{T}} k' = \mathcal{C}_K(k \cdot_{K_\mathcal{P}} k')$$

$$\forall I \in \mathbb{I} : (k \cdot_{K_\mathcal{P}} k')(I) = \sum_{\forall I', I'' : I = I' \cap I''} k(I') \cdot_K k'(I'')$$

Example 6.1. *Consider the $\mathbb{N}_\mathcal{T}$-relation* `works` *shown in Figure 2 (middle) and query $\Pi_{skill}(works)$. Recall that the annotation of a tuple t in the result of a projection over a K-relation is the sum of all input tuples which are projected onto t. For result tuple* `(SP)` *we have input tuples* `(Ann, SP)` *and* `(Sam, SP)` *with $\mathcal{T}_1 = \{[03, 10) \mapsto 1, [18, 20) \mapsto 1\}$ and $\mathcal{T}_2 = \{[08, 16) \mapsto 1\}$, respectively. The tuple* `(SP)` *is annotated with the sum of these annotations, i.e., $\mathcal{T}_1 +_{\mathbb{N}_\mathcal{T}} \mathcal{T}_2$. Substituting definitions we get:*

$$\begin{aligned} &\mathcal{T}_1 +_{\mathbb{N}_\mathcal{T}} \mathcal{T}_2 = \mathcal{C}_\mathbb{N}(\mathcal{T}_1 +_{\mathbb{N}_\mathcal{P}} \mathcal{T}_2) \\ =&\mathcal{C}_\mathbb{N}(\{[03, 10) \mapsto 1, [18, 20) \mapsto 1, [08, 16) \mapsto 1\}) \\ =&\{[03, 08) \mapsto 1, [08, 10) \mapsto 2, [10, 16) \mapsto 1, [18, 20) \mapsto 1\} \end{aligned}$$

*Thus, as expected, the result records that, e.g., there are two skilled workers (*SP*) on duty during time interval* $[08, 10)$.

Having defined the family of period semirings, it remains to be shown that $K_\mathcal{T}$ with standard K-relational query semantics is a representation system for sequenced K-relations.

6.1 $K_\mathcal{T}$ is a Semiring

As a first step, we prove that for any semiring K, the structure $K_\mathcal{T}$ is also a semiring. The following lemma shows that K-coalesce can be redundantly pushed into $+_{K_\mathcal{P}}$ and $\cdot_{K_\mathcal{P}}$ operations.

Lemma 6.1. *Let K be a semiring and $k, k' \in \mathbb{TEC}_K$. Then,*

$$\mathcal{C}_K(k +_{K_\mathcal{P}} k') = \mathcal{C}_K(\mathcal{C}_K(k) +_{K_\mathcal{P}} k')$$

$$\mathcal{C}_K(k \cdot_{K_\mathcal{P}} k') = \mathcal{C}_K(\mathcal{C}_K(k) \cdot_{K_\mathcal{P}} k')$$

Using this lemma, we now prove that for any semiring K, the structure $K_\mathcal{T}$ is also a semiring.

Theorem 6.2. *For any semiring K, structure $K_\mathcal{T}$ is a semiring.*

6.2 Timeslice Operator

We define a timeslice operator for $K_\mathcal{T}$-relations based on the timeslice operator for temporal K-elements. We annotate each tuple in the output of this operator with the result of τ_T applied to the temporal K-element the tuple is annotated with.

Definition 6.2 (Timeslice for $K_\mathcal{T}$-relations)**.** *Let R be a $K_\mathcal{T}$-relation and $T \in \mathbb{T}$. The timeslice operator $\tau_T(R)$ is defined as:*

$$\tau_T(R)(t) = \tau_T(R(t))$$

We now prove that the τ_T is a homomorphism $K_\mathcal{T} \rightarrow K$. Since semiring homomorphisms commute with queries [20], $K_\mathcal{T}$ equipped with this timeslice operator does fulfill the snapshot-reducibility condition of representation systems (Definition 4.5).

Theorem 6.3. *For any $T \in \mathbb{T}$, the timeslice operator τ_T is a semiring homomorphism from $K_\mathcal{T}$ to K.*

As an example of the application of this homomorphism, consider the period $\mathbb{N}$-relation `works` from our running example as shown on the left of Figure 2. Applying τ_{08} to this relation yields the snapshot shown on the bottom of this figure (three employees work between 8am and 9am out of whom two are specialized). If we evaluate query Q_{onduty} over this snapshot we get the snapshot shown on the right of this figure (the count is 2). By Theorem 6.3 we get the same result if we evaluate Q_{onduty} over the input period $\mathbb{N}$-relation and then apply τ_{08} to the result.

6.3 Encoding of Sequenced K-relations

We now define a bijective mapping $\textsc{Enc}_K$ from sequenced K-relations to $K_\mathcal{T}$-relations. We then prove that the set of $K_\mathcal{T}$-relations together with the timeslice operator for such relations and the mapping ${\textsc{Enc}_K}^{-1}$ (the inverse of $\textsc{Enc}_K$) form a representation system for sequenced K-relations. Intuitively, $\textsc{Enc}_K(R)$ is constructed by assigning each tuple t a temporal K-element where the annotation of the tuple at time T (i.e., $R(T)(t)$) is assigned to a singleton interval $[T, T+1)$. This temporal K-element $\mathcal{T}_{R,t}$ is then coalesced to create a $\mathbb{TEC}_K$ element.

Definition 6.3. *Let K be a semiring and R a sequenced K-relation, $\textsc{Enc}_K$ is a mapping from sequenced K-relations to $K_\mathcal{T}$-relations defined as follows.*

$$\forall t : \textsc{Enc}_K(R)(t) = \mathcal{C}_K(\mathcal{T}_{R,t})$$

$$\forall t, I : \mathcal{T}_{R,t}(I) = \begin{cases} R(T)(t) & \text{if } I = [T, T+1) \\ 0_K & \text{otherwise} \end{cases}$$

We first prove that this mapping is bijective, i.e., it is invertible, which guarantees that $\textsc{Enc}_K{}^{-1}$ is well-defined and also implies uniqueness (condition 1 of Definition 4.5).

Lemma 6.4. *For any semiring K, $\textsc{Enc}_K$ is bijective.*

Next, we have to show that $\textsc{Enc}_K$ preserves snapshots, i.e., the instance at a time point T represented by R can be extracted from $\textsc{Enc}_K(R)$ using the timeslice operator.

Lemma 6.5. *For any semiring K, sequenced K-relation R, and time point $T \in \mathbb{T}$, we have $\tau_T(\textsc{Enc}_K(R)) = \tau_T(R)$.*

Based on these properties of $\textsc{Enc}_K$ and the fact that the timeslice operator over $K_\mathcal{T}$-relations is a homomorphism $K_\mathcal{T} \rightarrow K$, our main technical result follows immediately. That is, the set of $K_\mathcal{T}$-relations equipped with the timeslice operator and $\textsc{Enc}_K{}^{-1}$ is a representation system for positive relational algebra queries ($\mathcal{RA}^+$) over sequenced K-relations.

Theorem 6.6 (Representation System)**.** *Given a semiring K, let $\mathcal{DB}_{K_\mathcal{T}}$ be the set of all $K_\mathcal{T}$-relations. The triple $(\mathcal{DB}_{K_\mathcal{T}}, \textsc{Enc}_K{}^{-1}, \tau)$ is a representation system for $\mathcal{RA}^+$ queries over sequenced K-relations.*

7. COMPLEX QUERIES

Having proven that $K_\mathcal{T}$-relations form a representation system for $\mathcal{RA}^+$, we now study extensions for difference and aggregation.

7.1 Difference

Extensions of K-relations for difference have been studied in [3, 19]. For instance, the difference operator on $\mathbb{N}$ relations corresponds to bag difference (SQL's **EXCEPT ALL**). Geerts et al. [19] apply an extension of semirings with a monus operation that is defined based on the *natural order* of a semiring and demonstrated how to define a difference operation for K-relations based on the monus operation for semirings where this operations is well-defined. Following the terminology introduced in this work, we refer to semirings with a monus operation as m-semirings. We now prove that if a semiring K has a well-defined monus, then so does $K_\mathcal{T}$. From this follows, that for any such K, the difference operation is well-defined for $K_\mathcal{T}$. We proceed to show that the timeslice operator is an m-semiring homomorphism, which implies that $K_\mathcal{T}$-relations for any m-semiring K form a representation system for $\mathcal{RA}$ (full relational algebra). The definition of a monus operator is based on the so-called natural order $\preceq_K$. For two elements k and k' of a semiring K, $k \preceq_K k' \Leftrightarrow \exists k'' : k +_K k'' = k'$. If $\preceq_K$ is a partial order then K is called *naturally ordered*. For instance, $\mathbb{N}$ is naturally ordered ($\preceq_\mathbb{N}$ corresponds to the order of natural numbers) while $\mathbb{Z}$ is not (for any $k, k' \in \mathbb{Z}$ we have $k \preceq_\mathbb{Z} k'$). For the monus to be well-defined on K, K has to be naturally ordered and for any $k, k' \in K$, the set $\{k'' \mid k \preceq_K k' +_K k''\}$ has to have a smallest member. For any semiring fulfilling these two conditions, the monus operation $-_K$ is defined as $k -_K k' = k''$ where k'' is the smallest element such that $k \preceq_K k' + k''$. For instance, the monus for $\mathbb{N}$ is the truncating minus: $k -_\mathbb{N} k' = max(0, k - k')$.

Theorem 7.1. *For any m-semiring K, semiring $K_\mathcal{T}$ has a well-defined monus, i.e., is an m-semiring.*

Let $k -_{K_\mathcal{P}} k'$ denote an operation that returns a temporal K-element which assigns to each singleton interval $[T, T+1)$ the result of the monus for K: $\tau_T(k) -_K \tau_T(k')$ (this is k_{pmin} as defined in the proof of Theorem 7.1, see Appendix A). In the proof of Theorem 7.1, we demonstrate that $k -_{K_\mathcal{T}} k' = \mathcal{C}_K(k -_{K_\mathcal{P}} k')$. Obviously, computing $k -_{K_\mathcal{P}} k'$ using singleton intervals is not effective. In our implementation, we use a more efficient way to compute the monus for $K_\mathcal{T}$ that is based on normalizing the input temporal K-elements k and k' such that annotations are attached to larger time intervals where $k -_{K_\mathcal{P}} k'$ is guaranteed to be constant. Importantly, τ_T is a homomorphism for monus-semiring $K_\mathcal{T}$.

Theorem 7.2. *Mapping τ_T is an m-semiring homomorphism.*

For example, consider $Q_{skillreq}$ from Example 1.2 which can be expressed in relational algebra as $\Pi_{skill}(assign) - \Pi_{skill}(worker)$. The $\mathbb{N}_\mathcal{T}$-relation corresponding to the period relation `assign` shown in this example annotates each tuple with a singleton temporal $\mathbb{N}$-element mapping the period of this tuple to 1, e.g., `(M1, SP)` is annotated with $\{[03, 12) \mapsto 1\}$. The annotation of result tuple `(SP)` is computed as

$$
\begin{aligned}
&(\{[03,12) \mapsto 1\} +_{\mathbb{N}_\mathcal{T}} \{[06,14) \mapsto 1\}) \\
&\quad -_{\mathbb{N}_\mathcal{T}} (\{[03,10) \mapsto 1\} +_{\mathbb{N}_\mathcal{T}} \{[08,16) \mapsto 1\} +_{\mathbb{N}_\mathcal{T}} \{[18,20) \mapsto 1\}) \\
=&\{[03,06) \mapsto 1, [06,12) \mapsto 2, [12,14) \mapsto 1\} \\
&\quad -_{\mathbb{N}_\mathcal{T}} \{[03,08) \mapsto 1, [08,10) \mapsto 2, [10,16) \mapsto 1, [18,20) \mapsto 1\} \\
=&\{[06,08) \mapsto 1, [10,12) \mapsto 1\}
\end{aligned}
$$

As expected, the result is the same as the one from Example 1.2.

7.2 Aggregation

The K-relational framework has previously been extended to support aggregation [4]. This required the introduction of attribute domains which are symbolic expressions that pair values with semiring elements to represent aggregated values. Since the construction used in this work to derive the mathematical structures representing these symbolic expressions is applicable to all semirings, it is also applicable to our period semirings. It was shown that semiring homomorphisms can be lifted to these more complex annotation structures and attribute domains. Thus, the timeslice operator, being a semiring homomorphism, commutes with queries including aggregation, and it follows that using the approach from [4], we can define a representation system for sequenced K-relations under $\mathcal{RA}$ with aggregation, i.e., $\mathcal{RA}^{agg}$.

One drawback of this definition of aggregation over K-relations with respect to our use case is that there are multiple ways of encoding the same sequenced K-relation in this model. That is, we would loose uniqueness of our representation system. Recall that one of our major goals is to implement sequenced query semantics on-top of DBMS using a period multiset encoding of $\mathbb{N}_\mathcal{T}$-relations. The symbolic expressions representing aggregation function results are a compact representation which, in case of our interval-temporal semirings, encode how the aggregation function results change over time. However, it is not clear how to effectively encode the symbolic attribute values and comparisons of symbolic expression as multiset semantics relations, and how to efficiently implement our sequenced semantics over this encoding. Nonetheless, for $\mathbb{N}$, we can apply a simpler definition of aggregation that returns a $K_\mathcal{T}$ relation and is also a representation system. For simplicity, we define aggregation ${}_G\gamma_{f(A)}(R)$ grouping on G to compute a single aggregation function f over the values of an attribute A. For convenience, aggregation without group-by, i.e., $\gamma_{f(A)}(R)$ is expressed using an empty group-by list.

Definition 7.1 (Aggregation)**.** *Let R be a $\mathbb{N}_\mathcal{T}$ relation. Operator ${}_G\gamma_{f(A)}(R)$ groups the input on a (possibly empty) list of attributes*

$G = (g_1, \dots, g_n)$ and computes aggregation function f over the values of attribute A. This operator is defined as follows:

$$
{}_G\gamma_{f(A)}(R)(t) = \mathcal{C}_{\mathbb{N}}(k_{R,t})
$$

$$
k_{R,t}(I) = \begin{cases} 1 & \textit{if } \exists T : I = [T, T{+}1) \wedge t \in {}_G\gamma_{f(A)}(\tau_T(R)) \\ 0 & \textit{otherwise} \end{cases}
$$

In the output of the aggregation operator, each tuple t is annotated with a $\mathbb{N}$-coalesced temporal $\mathbb{N}$-element which is constructed from singleton intervals. A singleton interval $I = [T, T + 1)$ is mapped to 1 if evaluating the aggregation over the multiset relation corresponding to the snapshot at T returns tuple t. We now demonstrate that $\mathbb{N}_{\mathcal{T}}$ using this definition of aggregation is a representation system for snapshot $\mathbb{N}$-relations.

Theorem 7.3. *$\mathbb{N}_{\mathcal{T}}$-relations form a representation system for sequenced $\mathbb{N}$-relations and $\mathcal{RA}^{agg}$ queries using aggregation according to Definition 7.1.*

8. SQL PERIOD RELATION ENCODING

While provably correct, the annotation structure that we have defined is quite complex in nature raising concerns on how to efficiently implement it. We now demonstrate that $\mathbb{N}_{\mathcal{T}}$-relations (multisets) can be encoded as SQL period relations (as shown on the top of Figure 2). Recall that SQL period relations are multiset relations where the validity time interval (period) of a tuple is stored in an interval-valued attribute (or as two attributes storing interval end points). Queries over $\mathbb{N}_{\mathcal{T}}$ are then translated into non-temporal multiset queries over this encoding. In addition to employing a proven and simple representation of time this enables our approach to run sequenced queries over such relations without requiring any preprocessing and to implement our ideas on top of a classical DBMS. For convenience we represent SQL period relations using non-temporal $\mathbb{N}$-relations in the definitions. SQL period relations can be obtained based on the well-known correspondence between multiset relations and $\mathbb{N}$-relations: we duplicate each tuple based on the multiplicity recorded in its annotation. To encode $\mathbb{N}_{\mathcal{T}}$-relations as $\mathbb{N}$-relations we introduce an invertible mapping PeriodEnc. We rewrite queries with $\mathbb{N}_{\mathcal{T}}$-semantics into non-temporal queries with $\mathbb{N}$-semantics over this encoding using a rewriting function Rewr. This is illustrated in the commutative diagram below.

$$
\begin{tikzcd}
\boxed{R} \arrow[r, "\textsc{PeriodEnc}"] \arrow[d, "Q"'] & \boxed{R'} \arrow[d, "Q' = \textsc{Rewr}(Q)"] \\
\boxed{Q(R)} & \boxed{Q'(R')} \arrow[l, "\textsc{PeriodEnc}^{-1}"]
\end{tikzcd}
\tag{1}
$$

Our encoding represents a tuple t annotated with a temporal element $\mathcal{T}$ as a set of tuples, one for each interval I which is assigned a non-zero value by $\mathcal{T}$. For each such interval, the interval's end points are stored in two attributes A_{begin} and A_{end}, which are appended to the schema of t. Again, we use $t \mapsto k$ to denote that tuple t is annotated with k and $\mathcal{U}$ to denote a universal domain of values. We use $SCH(R)$ to denote the schema of relation R and $arity(R)$ to denote its arity (the number of attributes in the schema).

Definition 8.1 (Encoding as SQL Period Relations)**.** *PeriodEnc is a function from $\mathbb{N}_{\mathcal{T}}$-relations to $\mathbb{N}$-relations. Let R be a $\mathbb{N}_{\mathcal{T}}$ relation with schema $SCH(R) = \{A_1, \dots, A_n\}$. The schema of PeriodEnc(R) is $\{A_1, \dots, A_n, A_{begin}, A_{end}\}$. Let R' be PeriodEnc(R) for some $\mathbb{N}_{\mathcal{T}}$-relation. PeriodEnc and its inverse are defined as follows:*

$$
\textsc{PeriodEnc}(R) = \bigcup_{t \in \mathcal{U}^{arity(R)}} \bigcup_{I \in \mathbb{I}} \{(t, I^+, I^-) \mapsto R(t)(I)\}
$$

$$
\textsc{PeriodEnc}^{-1}(R') = \bigcup_{t \in \mathcal{U}^{arity(R)}} \{t \mapsto \mathcal{T}_{R',t}\}
$$

$$
\forall I \in \mathbb{I} : \mathcal{T}_{R',t}(I) = R'(t_I) \textit{ for } t_I = (t, I^+, I^-)
$$

Before we define the rewriting Rewr that reduces a query Q with $\mathbb{N}_{\mathcal{T}}$ semantics to a query with $\mathbb{N}$ semantics, we introduce two operators that we will make use of in the reduction. The $\mathbb{N}$-coalesce operator applies $\mathcal{C}_{\mathbb{N}}$ to the annotation of each tuple in its input.

Definition 8.2 (Coalesce Operator)**.** *Let R be PeriodEnc(R') for some $\mathbb{N}_{\mathcal{T}}$-relation R'. The coalesce operator $\mathcal{C}(R)$ is defined as:*

$$
\mathcal{C}(R) = \textsc{PeriodEnc}(R')
$$

$$
\forall t : R'(t) = \mathcal{C}_{\mathbb{N}}(\textsc{PeriodEnc}^{-1}(R)(t))
$$

The split operator $\mathcal{N}_G(R, S)$ splits the intervals in the temporal elements annotating a tuple t based on the union of all interval end points from annotations of tuples t' which agree with t on attributes G. Inputs R and S have to be union compatible. The effect of this operator is that all pairs of intervals mapped to non-zero elements are either the same or are disjoint. This operator has been applied in [16,18] and in [12,46]. We use it to implement snapshot-reducible aggregation and difference over intervals instead of single snapshots as in Section 7. Recall that in Section 7, the monus (difference) and aggregation were defined in a point-wise manner. The split operator allows us to evaluate these operations over intervals directly by generating tuples with intervals for which the result of these operations is guaranteed to be constant.

Definition 8.3 (Split Operator)**.** *The split operator $\mathcal{N}_G(R_1, R_2)$ takes as input two $\mathbb{N}$-relations R_1 and R_2 that are encodings of $\mathbb{N}_{\mathcal{T}}$-relations. For a tuple t in such an encoding let $I(t) = [t.A_{begin}, t.A_{end})$. The split operator is defined as:*

$$
\begin{aligned}
\mathcal{N}_G(R_1, R_2)(t) &= \textit{split}(t, R_1, EP_G(R_1 \cup R_2, t)) \\
EP_G(R, t) &= \bigcup_{t' \in R : t'.G = t.G \wedge R(t') > 0} \{t'.A_{begin}\} \cup \{t'.A_{end}\} \\
\textit{split}(t, R, EP) &= \sum_{t' : I(t) \subseteq I(t') \wedge I(t) \in EPI(t, EP)} R(t') \\
EPI(t, EP) &= \{[T_b, T_e) \mid T_b <_{\mathbb{T}} T_e \wedge T_b \in EP \wedge \\
&\qquad (T_e \in EP \vee T_e = T_{max}) \wedge \\
&\qquad \nexists T' \in EP : T_b <_{\mathbb{T}} T' <_{\mathbb{T}} T_e\}
\end{aligned}
$$

Note that the PeriodEnc and PeriodEnc^{-1} mappings are only used in the definitions of the coalesce and split algebra operators for ease of presentation. These operators can be implemented as SQL queries executed over an PeriodEnc-encoded relation.

Definition 8.4 (Query Rewriting)**.** *We use $overlaps(Q_1, Q_2)$ as a shortcut for $Q_1.A_{begin} < Q_2.A_{end} \wedge Q_2.A_{begin} < Q_1.A_{end}$. The definition of rewriting Rewr is shown in Figure 4. Here $\{t\}$ denotes a constant relation with a single tuple t annotated with 1.*

Example 8.1. *Reconsider query Q_{onduty} from Example 1.1 and its results for the logical model and period relations (Figure 2). In relational algebra, the input query is written as $Q_{onduty} =$*

$$\underline{\textsc{Rewr}(R)} = R \qquad \underline{\textsc{Rewr}(\sigma_\theta(Q))} = \mathcal{C}(\sigma_\theta(\textsc{Rewr}(Q))) \qquad \underline{\textsc{Rewr}(\Pi_A(Q))} = \mathcal{C}(\Pi_{A,A_{begin},A_{end}}(\textsc{Rewr}(Q)))$$

$$\underline{\textsc{Rewr}(Q_1 \bowtie_\theta Q_2)} = \mathcal{C}(\Pi_{SCH(Q_1 \bowtie_\theta Q_2),max(Q_1.A_{begin},Q_2.A_{begin}),min(Q_1.A_{end},Q_2.A_{end})}(\textsc{Rewr}(Q_1) \bowtie_{\theta \wedge overlaps(Q_1,Q_2)} \textsc{Rewr}(Q_2)))$$

$$\underline{\textsc{Rewr}(Q_1 - Q_2)} = \mathcal{C}(\mathcal{N}_{SCH(Q_1)}(\textsc{Rewr}(Q_1), \textsc{Rewr}(Q_2)) - \mathcal{N}_{SCH(Q_2)}(\textsc{Rewr}(Q_2), \textsc{Rewr}(Q_1)))$$

$$\underline{\textsc{Rewr}(\gamma_{f(A)}(Q))} = \mathcal{C}({}_{A_{begin},A_{end}}\gamma_{f(A)}(\mathcal{N}_\emptyset(\textsc{Rewr}(Q) \cup \{(null, T_{min}, T_{max})\}, \textsc{Rewr}(Q))))$$

$$\underline{\textsc{Rewr}(\gamma_{count(*)}(Q))} = \textsc{Rewr}(\gamma_{count(A)}(\Pi_{1 \to A}(Q)))$$

$$\underline{\textsc{Rewr}({}_G\gamma_{f(A)}(Q))} = \mathcal{C}({}_{G,A_{begin},A_{end}}\gamma_{f(A)}(\mathcal{N}_G(\textsc{Rewr}(Q), \textsc{Rewr}(Q)))) \qquad \underline{\textsc{Rewr}(Q_1 \cup Q_2)} = \mathcal{C}(\textsc{Rewr}(Q_1) \cup \textsc{Rewr}(Q_2))$$

Figure 4: Rewriting REWR that reduces queries over $\mathbb{N}_\mathcal{T}$ to queries over a multiset encoding produced by PERIODENC.

$\gamma_{count(*)}(\underbrace{\sigma_{skill=SP}(works)}_{Q_1})$. *Applying* REWR *we get:*

$$\begin{aligned}\textsc{Rewr}(Q_{onduty}) = \mathcal{C}({}_{A_{begin},A_{end}}\gamma_{count(A)}(\mathcal{N}_\emptyset(\\ \Pi_{1 \to A, A_{begin}, A_{end}}(\textsc{Rewr}(Q_1)) \cup \{(null, 0, 24)\},\\ \textsc{Rewr}(Q_1))))\\ \textsc{Rewr}(Q_1) = \mathcal{C}(\sigma_{skill=SP}(works))\end{aligned}$$

Subquery REWR(Q_1) *filters out the second tuple from the input (see Figure 2). The split operator is then applied to the union of the result of* REWR(Q_1) *and a tuple with the neutral element* $null$ *for the aggregation function and period* $[T_{min}, T_{max})$*, where* $T_{min} = 0$ *and* $T_{max} = 24$ *for this example. After the split* $\mathcal{N}$*, the aggregation is evaluated grouping the input on* A_{begin}, A_{end}*. The* `count` *aggregation function then either counts a sequence of* 1*s and a single* $null$ *value producing the number of facts that overlap over the corresponding period* $[A_{begin}, A_{end})$*, or counts a single* $null$ *value over a "gap" producing* 0*. For instance, for* $[08, 10)$ *there are two facts whose intervals cover this period (Ann and Sam) and, thus,* $(2, [08, 10))$ *is returned by* REWR(Q_{onduty})*. While for for* $[20, 24)$ *there are no facts and thus we get* $(0, [20, 24))$*.*

Theorem 8.1. *The commutative diagram in Equation* (1) *holds.*

9. IMPLEMENTATION

We have implemented the encoding and rewriting introduced in the previous section in a middleware which supports sequenced multiset semantics through an extension of SQL. To instruct the system to interpret a subquery using sequenced semantics, the user encloses the subquery in a `SEQ VT (...)` block. We assume that the inputs to a sequenced query are encoded as period multiset relations, i.e., each relation has two temporal attributes that store the begin and end timestamp of their validity interval. For each relation access within a `SEQ VT` block, the user has to specify which attributes store the period of a tuple.

Our coalescing and split operators can be expressed in SQL. Thus, a straightforward way of incorporating these operators into the compilation process is to devise additional rewrites that produce the relational algebra code for these operators where necessary. However, preliminary experiments demonstrated that a naive implementation of these operators is prohibitively expensive.

We address this problem in two ways. First, we observe that it is sufficient to apply coalesce as a last step in a query instead of applying it as part of every operator rewrite. Applying this optimization, the rewritten version of a query will only contain one coalesce operator. Recall from Lemma 6.1 that coalescing can be redundantly pushed into the addition and multiplication operations of period semirings, e.g., $\mathcal{C}_K(k +_{K_\mathcal{P}} k') = \mathcal{C}_K(\mathcal{C}_K(k) +_{K_\mathcal{P}} k')$. We prove that this Lemma also holds for monus in Appendix E. Interpreting this equivalence from right to left and applying it repeatedly to a semiring expression e, e can be rewritten into an equivalent expression of the form $\mathcal{C}_K(e')$, where e' is an expression that only uses operations $+_{K_\mathcal{P}}$, $\cdot_{K_\mathcal{P}}$, $-_{K_\mathcal{P}}$. Since relational algebra over K-relations is defined by applying multiplication, addition, and monus to input annotations, this implies that it is sufficient to apply coalescing only as a final operation in a query. For an example and additional discussion see Appendix E

We developed an optimized implementation of multiset coalescing using SQL analytical window functions, similar to set-based coalescing in [47], that counts for value-equivalent attributes the number of open intervals per time point, determines change points based on differences between these counts, and then only output maximal intervals using a filter step. This implementation uses sorting in its window declarations and has time complexity $\mathcal{O}(n \log n)$ for n tuples. A native implementation would require only one sorting step. The number of sorting steps required by our SQL implementation depends on whether the DBMS is capable of sharing window declaration (we observe 2 and 7 sorting steps for the systems used in our experimental evaluation).

For aggregation we integrate the split operator into the aggregation. It turned out to be most effective to pre-aggregate the input before splitting and then compute the final aggregation results during the split step by further aggregating the results of the pre-aggregation step. We apply a similar optimization for difference.

10. EXPERIMENTS

In our experimental evaluation we focus on two aspects. First, we evaluate the cost of our SQL implementation of $\mathbb{N}$-coalescing (multiset coalescing). Then, we evaluate the performance of sequenced queries with our approach over three DBMSs and compare it against native implementations of sequenced semantics that are available in two of these systems (using our implementation of coalescing to produce a coalesced result).

10.1 Workloads and Experimental Setup

Datasets. We use three datasets in our experiments. The *MySQL Employees dataset* (`https://github.com/datacharmer/test_db`) which contains ≈4 million records and consists of the following six period tables: table `employee` stores basic information about employees; table `departments` stores department information; table `titles` stores the job titles for employees; table `salaries` stores employee salaries; table `dept_manager` stores which employee manages which department; and table `dept_emp` stores which employee is working for which department. *TPC-BiH* is the bi-temporal version of the TPC-H benchmark dataset as described in [24]. Since our approach supports only one time dimension we only generated the valid time dimension for this dataset. In this configuration a scale factor 1 (SF1) database corresponds to roughly 1GB of data. The *Tourism* dataset (835k records) consists of a single table storing hotel reservations in South Tyrol. Each record corresponds to one reservation. The validity end points of the time period associated with a record is the arrival and departure time.

Workloads. We have created a workload consisting of 10 queries to evaluate the efficiency of sequenced queries. Queries `join-1` to `join-4` are join queries, `agg-1` to `agg-3` are aggregation-heavy queries, `agg-join` is a join with an aggregation value, and `diff-1` and `diff-2` use difference. Furthermore, we use one query template varying the selectivity to evaluate the performance of coalescing. `C-Sn` denotes the variant of this query that returns approximately nK rows, e.g., `C-S1` returns 1,000 rows. For the Tourism dataset we use the following queries. `join`: tourist from same country to same destination using a self join of tourismdata table. `agg-0`: number of tourists per destination together with the average number of tourists for all other destinations. This query first computes the number of tourists per destination and do a self unequal join on it. `agg-1`: number of enquiries and the number of tourists per destination with more than 1000 enquiries using two aggregations on tourismdata table. `agg-2`: maximum number of tourists per destination using an aggregation on tourismdata table. `tou-agg-x`: the destination with the most number of tourists. This query has no join but two aggregations, one to compute the number of tourists per destination and a second one to compute the maximum one. More detailed descriptions of these queries are provided in Appendix B. For the TPC-BiH dataset we took 9 of the 22 standard queries [14] from this benchmark that do not contain nested subqueries or **LIMIT** (which are not supported by our or any other approach for sequenced queries we are aware of) and evaluated these queries under sequenced semantics. Note that some of these queries use the **ORDER BY** clause that we do not support for sequenced queries. However, we can evaluate such a query without **ORDER BY** under sequenced semantics and then sort the result without affecting what rows are returned. The number of rows returned by these queries over the dataset are shown in Table 2.

join-1	join-2	join-3	join-4	agg-1	agg-2	agg-3	agg-join	diff-1	diff-2
2.8M	28.3M	10	177	57.4k	177	210	260	300k	2.8M

TPC-H	Q1	Q3	Q5	Q6	Q7	Q8	Q9	Q10	Q12	Q14	Q19
1GB	4.3k	10	386	529	1.6k	742	69.7k	20	785	479	220
10GB	4.3k	10	579	532	1.7k	867	74.8k	20	786	487	1.3k

tou-join-agg	tou-agg-1	tou-agg-2	tou-agg-3	tou-agg-join
64.3k	954	14.5k	3.2k	822

Table 2: Number of query result rows

Systems. We ran experiments on three different database management systems: a version of Postgres (*PG*) with native support for temporal operators as described in [16, 18]; a commercial DBMS, *DBX*, with native support for sequenced semantics (only available as a virtual machine); and a commercial DBMS, *DBY*, without native support for sequenced semantics. We used our approach to translate sequenced queries into standard SQL queries and ran the translated queries on all three systems (denoted as *PG-Seq*, *DBX-Seq*, and *DBY-Seq*). For PG and DBX, we ran the queries also with the native solution for sequenced semantics paired with our implementation of coalescing to produce a coalesced result (referred to as *PG-Nat* and *DBX-Nat*). As explained in Section 2, no system correctly implements sequenced multiset semantics for difference and aggregation, and many systems do not support sequenced semantics for these operators at all. *DBX-Nat* and *PG-Nat* both support sequenced aggregation, however, their implementations are not snapshot-reducible. *DBX-Nat* does not support sequenced difference, whereas *PG-Nat* implements temporal difference with set semantics. Despite such differences, the experimental comparison allows us to understand the performance impact of our provably correct approach.

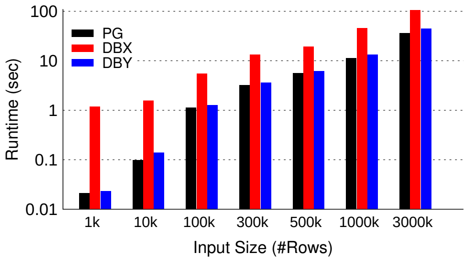


Figure 5: Multiset coalescing for varying input size.

All experiments were executed on a machine with 2 AMD Opteron 4238 CPUs, 128GB RAM, and a hardware RAID with 4 × 1TB 72.K HDs in RAID 5. For Postgres we set the buffer pool size to 8GB. For the other systems we use values recommended by the automated configuration tools of these systems. We execute queries with warm cache. For short-running queries we show the median runtime across 100 consecutive runs. For long running queries we computed the median over 10 runs. In general we observed low variation in runtimes (a few percent).

10.2 Multiset Coalescing

To evaluate the performance of coalescing, we use a selection query that returns employees that earn more than a specific salary and materialize the result as a table. The selectivity varies from 1K to 3M rows. We then evaluate the query **SELECT** * **FROM** ... over the materialized tables under sequenced semantics in order to measure the cost of coalescing in isolation. Figure 5 shows the results of this experiment. The runtime of coalescing is linear in the input size for all three systems. Even though the theoretical worst-case complexity of the sorting step, which is applied by all systems to evaluate the analytics functions that we exploit in our SQL-based implementation of multiset coalescing, is $\mathcal{O}(n \cdot log(n))$, an inspection of the execution plans revealed that the sorting step only amounts to 5%-10% of the execution time (for all selectivities) and, hence, is not a dominating factor.

10.3 Sequenced Semantics - Employee

Table 3 provides an overview of the performance results for our sequenced query workloads. For every query we indicate in the rightmost column whether native approaches are subject to the aggregation gap (AG) or bag difference (BD) bugs.

Join Queries. The performance of our approach for join queries is comparable with the native implementation in *PG-Nat*. For join queries with larger intermediate results (`join-2`), the native implementation outperforms our approach by ≈73%. Running the queries produced by our approach in *DBY* is slightly faster than both. *DBX-Nat* uses merge joins for temporal joins, while both *PG* and *DBY* use a hash-join on the non-temporal part of the join condition. The result is that *DBX-Nat* significantly outperforms the other methods for temporal join operations. However, the larger cost for the SQL-based coalescing implementation in this system often outweighs this effect. This demonstrates the potential for improving our approach by making use of native implementations of temporal operators in our rewrites for operators that are compatible with our semantics (note that joins are compatible).

Aggregation Queries. Our approach outperforms the native implementations of sequenced semantics on all systems by several

Employee dataset						
Query	**PG-Seq**	**PG-Nat**	**DBX-Seq**	**DBX-Nat**	**DBY-Seq**	**Bug**
`join-1`	91.97	118.01	118.95	116.03	64.00	
`join-2`	1543.81	888.13	1569.45	1200.36	763.70	
`join-3`	0.01	4.91	0.55	0.43	0.01	
`join-4`	0.52	12.85	0.83	0.60	0.22	
`agg-1`	7.02	5980.85	56.47	**OOTS**	5.24	
`agg-2`	0.06	10.31	0.82	0.82	0.01	AG
`agg-3`	1.42	0.02	0.78	0.55	0.01	AG
`agg-join`	6643.61	19195.03	**OOTS**	**OOTS**	7555.97	
`diff-1`	14.18	6.88	30.15	**N/A**	10.29	BD
`diff-2`	63.58	79.63	129.87	**N/A**	61.90	BD

Tourism				
Query	**PG-Seq**	**PG-Nat**	**DBY-Seq**	**Bug**
`tou-join-agg`	300.28	694.88	171.09	
`tou-agg-1`	2.41	94.58	1.61	
`tou-agg-2`	123.79	92.32	87.31	
`tou-agg-3`	6.68	98.07	7.66	AG
`tou-agg-join`	1.06	263.61	0.94	

TPC-BiH							
	SF1 (∼1 GB)			**SF10 (∼10 GB)**			
Query	**PG-Seq**	**PG-Nat**	**DBY-Seq**	**PG-Seq**	**PG-Nat**	**DBY-Seq**	**Bug**
`Q1`	12.02	3686.47	11.80	63.85	**TO (2h)**	82.61	
`Q5`	0.58	142.91	1.14	5.85	1794.10	14.89	
`Q6`	0.79	12.65	1.14	7.70	126.91	7.28	AG
`Q7`	1.14	285.91	5.33	28.70	1642.20	21.75	
`Q8`	1.77	108.63	2.20	21.78	1484.61	17.33	
`Q9`	10.12	**TO (2h)**	8.09	129.01	**TO (2h)**	71.37	
`Q12`	1.10	23.85	1.81	10.49	264.57	13.30	
`Q14`	1.72	403.92	2.75	26.55	3436.30	23.79	AG
`Q19`	0.92	203.83	2.55	9.60	2873.13	22.35	AG

Table 3: Runtimes (sec) of sequenced queries: **N/A** = not supported, **OOTS** = system ran out of temporary space (2GB), **TO (2h)**= timed out (2 hours).

orders of magnitude for aggregation queries as long as the aggregation input exceeds a certain size (`agg-1` and `agg-2`). Our approach as well as the native approaches split the aggregation input which requires sorting and then apply a standard aggregation operator to compute the temporal aggregation result. The main reason for the large performance difference is that the SQL code we generate for a sequenced aggregation includes several levels of pre-aggregation that are intertwined with the split operator. Thus, for our approach the sorting step for split is applied to a typically much smaller pre-aggregated dataset. This turned out to be quite effective. The only exception is if the aggregation input is very small (`agg-3`) in which case an efficient implementation of split (as in *PG-Nat*) outweighs the benefits of pre-aggregation. Query `agg-1` did not finish on *DBX-Nat* as it exceeded the 2GB temporary space restriction (memory allocated for intermediate results) of the freely available version of this DBMS.

Mixed Aggregation and Join. Query `agg-join` applies an aggregation over the result of several joins. Our approach is more effective, in particular for the aggregation part of this query, compared to *PG-Nat*. This query did not finish on *DBX* due to the 2GB temporary space restriction per query imposed by the DBMS.

Difference Queries. For difference queries we could only compare our approach against *PG-Nat*, since *DBX-Nat* does not support difference in sequenced queries. Note that, *PG-Nat* applies set difference while our approach supports multiset difference. While our approach is less effective for `diff-1` which contains a single difference operator, we outperform *PG-Nat* on `diff-2`.

10.4 Sequenced Semantics - TPC-BiH

The runtimes for TPC-H queries interpreted under sequenced semantics (9 queries are currently supported by the approaches) over the 1GB and 10GB valid time versions of TPC-BiH is also shown in Table 3. For this experiment we skip *DBX* since the limitation to 2GB of temporary space of the free version we were using made it impossible to run most of these queries. Overall we observe that our approach scales roughly linearly from 1GB to 10GB for these queries. We significantly outperform *PG-Nat* because all of these queries use aggregation. Additionally, some of these queries use up to 7 joins. For these queries the fact that *PG-Nat* aligns both inputs with respect to each other [16] introduces unnecessary overhead and limits join reordering. The combined effect of these two drawbacks is quite severe. Our approach is 1 to 3 orders of magnitude faster than *PG-Nat*. For some queries this is a lower bound on the overhead of *PG-Nat* since the system timed out for these queries (we stopped queries that did not finish within 2 hours).

10.5 Sequenced Semantics - Tourism

The results for the queries over the Tourism database are shown in the middle of Table 3. We only report our approach for Postgres and DBY, and the native implementation in Postgres. With the exception of query *tou-agg-2* our approach outperforms *PG-Nat* quite significantly since all these queries contain aggregation. Since query *tou-agg-2* does use `max` we do not apply our sweeping technique (see Appendix E.3). *PG-Nat*'s native implementation of the split operator results in 30% better performance for this query. Query *tou-join-agg* applies an inequality self-join over an aggregation result ($\approx$ 100k rows under sequenced semantics) and then applies a final aggregation to the join. The large size of this join result is the main reason

10.6 Summary

Our experiments demonstrate that an SQL-based implementation of multiset coalescing is feasible – exhibiting runtimes linear in the size of the input, albeit with a relatively large constant factor. We expect that it would be possible to significantly reduce this factor by introducing a native implementation of this operator. Using pre-aggregation during splitting, our approach significantly outperforms native implementations for aggregation queries. DBX uses merge joins for temporal joins (interval overlap joins) which is significantly more efficient than hash joins which are employed by Postgres and DBY. This shows the potential of integrating such specialized operators with our approach in the future. For example, we could compile sequenced queries into SQL queries that selectively employ the temporal extensions of a system like DBX.

11. CONCLUSIONS AND FUTURE WORK

We present the first provably correct interval-based representation system for sequenced semantics over multiset relations and its implementation in a database middleware. We achieve this goal by addressing a more general problem: snapshot-reducibility for temporal K-relations. Our solution is a uniform framework for evaluation of queries under sequenced semantics over an interval-based encoding of temporal K-relations for any semiring K. That is, in addition to sets and multisets, the framework supports sequenced temporal extensions of probabilistic databases, databases annotated with provenance, and many more. In future work, we will study how to extend our approach for updates over annotated relations, will study its applicability for combining probabilistic and temporal query processing, investigate implementations of split and K-coalescing inside a database kernel, and study extensions for bitemporal data.

12. REFERENCES

[1] M. Al-Kateb, A. Ghazal, and A. Crolotte. An efficient SQL rewrite approach for temporal coalescing in the teradata RDBMS. In *DEXA*, pages 375–383, 2012.

[2] M. Al-Kateb, A. Ghazal, A. Crolotte, R. Bhashyam, J. Chimanchode, and S. P. Pakala. Temporal query processing in Teradata. In *EDBT*, pages 573–578, 2013.

[3] Y. Amsterdamer, D. Deutch, and V. Tannen. On the limitations of provenance for queries with difference. In *TaPP*, 2011.

[4] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011.

[5] M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, pages 257–275, 2006.

[6] M. H. Böhlen, J. Gamper, C. S. Jensen, and R. T. Snodgrass. Sql-based temporal query languages. In *Encyclopedia of Database Systems*, pages 2762–2768. 2009.

[7] M. H. Böhlen and C. S. Jensen. Sequenced semantics. In *Encyclopedia of Database Systems*, pages 2619–2621. 2009.

[8] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Evaluating and enhancing the completeness of tsql2. Technical Report TR 95-5, Computer Science Department, University of Arizona, 1995.

[9] M. H. Böhlen, C. S. Jensen, and R. T. Snodgrass. Temporal statement modifiers. *ACM Trans. Database Syst.*, 25(4):407–456, 2000.

[10] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in temporal databases. In *VLDB*, pages 180–191, 1996.

[11] P. Bouros and N. Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *PVLDB*, 10(11):1346–1357, 2017.

[12] I. T. Bowman and D. Toman. Optimizing temporal queries: efficient handling of duplicates. *Data Knowl. Eng.*, 44(2):143–164, 2003.

[13] F. Cafagna and M. H. Böhlen. Disjoint interval partitioning. *VLDB J.*, 26(3):447–466, 2017.

[14] T. P. P. Council. TPC Benchmark$^{\text{TM}}$H (Decision Support) Standard Specification Revision 1.17.3, 2017.

[15] A. Das Sarma, M. Theobald, and J. Widom. Live: A lineage-supported versioned dbms. In *SSDBM*, pages 416–433, 2010.

[16] A. Dignös, M. H. Böhlen, and J. Gamper. Temporal alignment. In *SIGMOD*, pages 433–444, 2012.

[17] A. Dignös, M. H. Böhlen, and J. Gamper. Overlap interval partition join. In *SIGMOD*, pages 1459–1470, 2014.

[18] A. Dignös, M. H. Böhlen, J. Gamper, and C. S. Jensen. Extending the kernel of a relational DBMS with comprehensive support for sequenced temporal queries. *ACM Trans. Database Syst.*, 41(4):26:1–26:46, 2016.

[19] F. Geerts and A. Poggi. On database query languages for K-relations. *Journal of Applied Logic*, 8(2):173–185, 2010.

[20] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[21] C. S. Jensen and R. T. Snodgrass. Snapshot equivalence. In *Encyclopedia of Database Systems*, page 2659. 2009.

[22] C. S. Jensen and R. T. Snodgrass. Temporal query languages. In *Encyclopedia of Database Systems*, pages 3009–3012. 2009.

[23] C. S. Jensen and R. T. Snodgrass. Timeslice operator. In *Encyclopedia of Database Systems*, pages 3120–3121. 2009.

[24] M. Kaufmann, P. M. Fischer, N. May, A. Tonder, and D. Kossmann. Tpc-bih: A benchmark for bitemporal databases. In *TPCTC*, pages 16–31, 2013.

[25] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD*, pages 1173–1184, 2013.

[26] E. V. Kostylev and P. Buneman. Combining dependent annotations for relational algebra. In *ICDT*, pages 196–207, 2012.

[27] K. G. Kulkarni and J. Michels. Temporal features in SQL: 2011. *SIGMOD Record*, 41(3):34–43, 2012.

[28] N. A. Lorentzos and Y. G. Mitsopoulos. SQL extension for interval data. *IEEE Trans. Knowl. Data Eng.*, 9(3):480–499, 1997.

[29] Microsoft. Sql server 2016 - temporal tables. `https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables`, 2016.

[30] Oracle. Database development guide - temporal validity support. `https://docs.oracle.com/database/121/ADFNS/adfns_design.htm#ADFNS967`, 2016.

[31] D. Piatov and S. Helmer. Sweeping-based temporal aggregation. In *SSTD*, pages 125–144, 2017.

[32] D. Piatov, S. Helmer, and A. Dignös. An interval join optimized for modern hardware. In *ICDE*, pages 1098–1109, 2016.

[33] M. Pilman, M. Kaufmann, F. Köhl, D. Kossmann, and D. Profeta. Partime: Parallel temporal aggregation. In *SIGMOD*, pages 999–1010, 2016.

[34] PostgreSQL. Documentation manual postgresql - range types. `https://www.postgresql.org/docs/current/static/rangetypes.html`, 2012.

[35] C. Saracco, M. Nicola, and L. Gandhi. A matter of time: Temporal data management in db2 10. `http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/dm-1204db2temporaldata-pdf.pdf`, 2012.

[36] C. Sirangelo. Positive relational algebra. In *Encyclopedia of Database Systems*, pages 2124–2125. 2009.

[37] R. T. Snodgrass. The temporal query language tquel. *ACM Trans. Database Syst.*, 12(2):247–298, 1987.

[38] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. 1995.

[39] R. T. Snodgrass. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann, 1999.

[40] R. T. Snodgrass, I. Ahn, G. Ariav, D. S. Batory, J. Clifford, C. E. Dyreson, R. Elmasri, F. Grandi, C. S. Jensen, W. Käfer, N. Kline, K. G. Kulkarni, T. Y. C. Leung, N. A. Lorentzos, J. F. Roddick, A. Segev, M. D. Soo, and S. M. Sripada. TSQL2 language specification. *SIGMOD Record*, 23(1):65–86, 1994.

[41] R. T. Snodgrass, M. H. Böhlen, C. S. Jensen, and A. Steiner. Adding valid time to sql/temporal. *ANSI X3H2-96-501r2, ISO/IEC JTC*, 1, 1996.

[42] M. D. Soo, C. S. Jensen, and R. T. Snodgrass. An algebra for tsql2. In *The TSQL2 temporal query language*, pages 505–546. 1995.

[43] A. Steiner. *A generalisation approach to temporal data*

models and their implementations. PhD thesis, ETH Zurich, 1998.

[44] Teradata. Teradata database - temporal table support. `http://www.info.teradata.com/download.cfm?ItemID=1006923`, Jun 2015.

[45] D. Toman. Point vs. interval-based query languages for temporal databases. In *PODS*, pages 58–67, 1996.

[46] D. Toman. Point-based temporal extensions of SQL and their efficient implementation. In *Temporal databases: research and practice*, pages 211–237. 1998.

[47] X. Zhou, F. Wang, and C. Zaniolo. Efficient temporal coalescing query support in relational database systems. In *DEXA*, pages 676–686, 2006.

APPENDIX

A. PROOFS

Proof of Lemma 5.1. Equivalence preservation: Proven by contradiction. Assume that $\exists T : \tau_T(\mathcal{T}) \neq \tau_T(\mathcal{C}_K(\mathcal{T}))$. We have to distinguish two cases. If $T \in CP(\mathcal{T})$, then by definition of K-coalesce we have the contradiction: $\tau_T(\mathcal{C}_K(\mathcal{T})) = \tau_T(\mathcal{T})$. If $T \notin CP(\mathcal{T})$, then let T' be the largest change point that is smaller than T (this point has to exist). From the definition of change points follows that $\tau_T = \tau_{T'}$. By construction there has to exists exactly one interval overlapping T that is assigned a non-zero value in $\mathcal{C}_K(\mathcal{T})$ and this interval starts in T'. Hence, we have the contradiction.

Uniqueness: Note that change points are defined using τ_T only and $(\forall T \in \mathbb{T} : \tau_T(\mathcal{T}_1) = \tau_T(\mathcal{T}_2)) \Leftrightarrow \mathcal{T}_1 \sim \mathcal{T}_2$. Since the result of coalescing is uniquely determined by the change points of a temporal element it follows that $\mathcal{T}_1 \sim \mathcal{T}_2 \Leftrightarrow \mathcal{C}_K(\mathcal{T}_1) = \mathcal{C}_K(\mathcal{T}_2)$

Idempotence: Idempotence follows from the other two properties. If we substitute $\mathcal{T}$ and $\mathcal{C}_K(\mathcal{T})$ for $\mathcal{T}_1$ and $\mathcal{T}_2$ in the uniqueness condition, we get idempotence: $\mathcal{C}_K(\mathcal{C}_K(\mathcal{T})) = \mathcal{C}_K(\mathcal{T})$. □

Proof of Lemma 6.1. Push Through Addition: We prove this part by proving that for any k'' if $k \sim k'$ then $(k +_{K_\mathcal{P}} k'') \sim (k' +_{K_\mathcal{P}} k'')$. We have to show that for all $T \in \mathbb{T}$ we have $\tau_T(k + k'') = \tau_T(k' + k'')$. Substituting definitions we get:

$$\sum_{T \in I}(k(I) +_K k''(I)) = (\sum_{T \in I} k(I)) +_K (\sum_{T \in I} k''(I))$$

Using $k \sim k'$ and substituting the definition of $\sim$, i.e., $\sum_{T \in I} k(I) = \sum_{T \in I} k'(I)$, we get:

$$= (\sum_{T \in I} k'(I)) +_K (\sum_{T \in I} k''(I)) = \sum_{T \in I}(k'(I) +_K k''(I))$$

Push Through Multiplication: Analog to the proof for addition, we prove this part by showing that snapshot equivalence of inputs implies snapshot equivalence of outputs for multiplication.

$$\tau_T(k \cdot_{K_\mathcal{P}} k'') = \sum_{\forall I', I'' : I = I' \cap I'' \wedge T \in I} k(I') \cdot_K k''(I'')$$

Based on the fact that timeslice is a homomorphism $K_\mathcal{T} \to K$ which we will prove in Theorem 6.3, time slice commutes with multiplication and addition:

$$\begin{aligned}
&=(\sum_{\forall I \wedge T \in I} k(I)) \cdot_K (\sum_{\forall I \wedge T \in I} k''(I)) \\
&=(\sum_{\forall I \wedge T \in I} k'(I)) \cdot_K (\sum_{\forall I \wedge T \in I} k''(I)) \\
&=\tau_T(k' \cdot_{K_\mathcal{P}} k'')
\end{aligned}$$

□

Proof of Theorem 6.2. We have to show that the structure we have defined obeys the laws of commutative semirings. Since the elements of $K_\mathcal{T}$ are functions, it suffices to show $k(I) = k'(I)$ for every $I \in \mathbb{I}$ to prove that $k = k'$. For all $k, k' \in K_\mathcal{T}$ and $I \in \mathbb{I}$:

Addition is commutative:

$$(k +_{K_\mathcal{P}} k')(I) = k(I) +_K k'(I) = k'(I) +_K k(I) = (k +_{K_\mathcal{P}} k')(I)$$
$$k +_{K_\mathcal{T}} k' = \mathcal{C}_K(k +_{K_\mathcal{P}} k') = \mathcal{C}_K(k' +_{K_\mathcal{P}} k) = k' +_{K_\mathcal{T}} k$$

Addition is associative:

$$\begin{aligned}
&((k +_{K_\mathcal{P}} k') +_{K_\mathcal{P}} k'')(I) = (k(I) +_K k'(I)) +_K k''(I) \\
&= k(I) +_K (k'(I) +_K k''(I)) = (k +_{K_\mathcal{P}} (k' +_{K_\mathcal{P}} k''))(I)
\end{aligned}$$

$$(k +_{K_\mathcal{T}} k') = \mathcal{C}_K(k +_{K_\mathcal{P}} k') = \mathcal{C}_K(k' +_{K_\mathcal{P}} k) = (k' +_{K_\mathcal{T}} k)$$

Zero is neutral element of addition:

$$(k +_{K_\mathcal{P}} 0_{K_\mathcal{T}})(I) = k(I) +_K 0_{K_\mathcal{T}}(I) = k(I) +_K 0_K = k(I)$$
$$k +_{K_\mathcal{T}} 0_{K_\mathcal{T}} = \mathcal{C}_K(k +_{K_\mathcal{P}} 0_{K_\mathcal{T}}) = \mathcal{C}_K(k) = k$$

Multiplication is commutative:

$$\begin{aligned}
(k \cdot_{K_\mathcal{P}} k')(I) &= \sum_{\forall I', I'' : I = I' \cap I''} k(I') \cdot_K k'(I'') \\
&= \sum_{\forall I'', I' : I = I'' \cap I'} k(I'') \cdot_K k'(I') \\
&= \sum_{\forall I', I'' : I = I' \cap I''} k'(I') \cdot_K k(I'') = (k' \cdot_{K_\mathcal{P}} k)(I)
\end{aligned}$$

$$k \cdot_{K_\mathcal{T}} k' = \mathcal{C}_K(k \cdot_{K_\mathcal{P}} k') = \mathcal{C}_K(k' \cdot_{K_\mathcal{P}} k) = k' \cdot_{K_\mathcal{T}} k$$

Multiplication is associative:

$$\begin{aligned}
&((k \cdot_{K_\mathcal{P}} k') \cdot_{K_\mathcal{P}} k'')(I) \\
&= \sum_{\forall I_1, I_2 : I = I_1 \cap I_2} (\sum_{\forall I_3, I_4 : I_1 = I_3 \cap I_4} k(I_3) \cdot_K k'(I_4)) \cdot_K k''(I_2) \\
&= \sum_{\forall I_1, I_2 : I = I_1 \cap I_2} \sum_{\forall I_3, I_4 : I_1 = I_3 \cap I_4} (k(I_3) \cdot_K k'(I_4) \cdot_K k''(I_2)) \\
&= \sum_{\forall I_1, I_2, I_3 : I = I_1 \cap I_2 \cap I_3} k(I_1) \cdot_K k'(I_2) \cdot_K k''(I_3) \\
&= \sum_{\forall I_1, I_2 : I = I_1 \cap I_2} k(I_1) \cdot (\sum_{\forall I_3, I_4 : I_2 = I_3 \cap I_4} k'(I_3) \cdot_K k''(I_4)) \\
&= (k \cdot_{K_\mathcal{P}} (k' \cdot_{K_\mathcal{P}} k''))(I)
\end{aligned}$$

$$\begin{aligned}
&(k \cdot_{K_\mathcal{T}} k') \cdot_{K_\mathcal{T}} k' \\
&= \mathcal{C}_K(\mathcal{C}_K(k \cdot_{K_\mathcal{P}} k') \cdot_{K_\mathcal{P}} k'') \\
&= \mathcal{C}_K(k \cdot_{K_\mathcal{P}} k' \cdot_{K_\mathcal{P}} k'') \\
&= \mathcal{C}_K(k \cdot_{K_\mathcal{P}} \mathcal{C}_K(k' \cdot_{K_\mathcal{P}} k'')) \\
&= k \cdot_{K_\mathcal{T}} (k' \cdot_{K_\mathcal{T}} k'')
\end{aligned}$$

One is neutral element of multiplication:

$$(k \cdot_{K_\mathcal{P}} 1_{K_\mathcal{T}})(I) = \sum_{\forall I', I'' : I = I' \cap I''} k(I') \cdot_K 1_{K_\mathcal{T}}(I'')$$
$$= k(I) \cdot_{K_\mathcal{P}} 1_{K_\mathcal{T}}([t_{min}, t_{max})) = k(I) \cdot_K 1_K$$
$$= k(I)$$
$$k \cdot_{K_\mathcal{T}} 1_{K_\mathcal{T}}$$
$$= \mathcal{C}_K(k \cdot_{K_\mathcal{P}} 1_{K_\mathcal{T}}) = \mathcal{C}_K(k) = k$$

Distributivity:

$$(k \cdot_{K_\mathcal{P}} (k' +_{K_\mathcal{P}} k''))(I)$$
$$= \sum_{\forall I', I'' : I = I' \cap I''} k(I') \cdot_K (k'(I'') +_K k''(I''))$$
$$= \sum_{\forall I', I'' : I = I' \cap I''} (k(I') \cdot_K k'(I'')) +_K (k(I') \cdot_K k''(I''))$$
$$= \sum_{\forall I', I'' : I = I' \cap I''} (k(I') \cdot_K k'(I''))$$
$$+ \sum_{\forall I', I'' : I = I' \cap I''} (k(I') \cdot_K k''(I''))$$
$$= ((k \cdot_{K_\mathcal{P}} k') +_{K_\mathcal{P}} (k \cdot_{K_\mathcal{P}} k''))(I)$$

$$k \cdot_{K_\mathcal{T}} (k' +_{K_\mathcal{T}} k'')$$
$$= \mathcal{C}_K(k \cdot_{K_\mathcal{P}} \mathcal{C}_K(k' +_{K_\mathcal{P}} k''))$$
$$= \mathcal{C}_K(k \cdot_{K_\mathcal{P}} (k' +_{K_\mathcal{P}} k''))$$
$$= \mathcal{C}_K((k \cdot_{K_\mathcal{P}} k') +_{K_\mathcal{P}} (k \cdot_{K_\mathcal{P}} k'')))$$
$$= \mathcal{C}_K(\mathcal{C}_K(k \cdot_{K_\mathcal{P}} k') +_{K_\mathcal{P}} \mathcal{C}_K(k \cdot_{K_\mathcal{P}} k'')))$$
$$= (k \cdot_{K_\mathcal{T}} k') +_{K_\mathcal{T}} (k \cdot_{K_\mathcal{T}} k'') \qquad \square$$

Proof of Theorem 6.3. Proven by substitution of definitions:
Preserves neutral elements:

$$\mathcal{C}_K(\tau_T(0_{K_\mathcal{T}})) = \sum_{I \in \mathbb{I} : T \in I} 0_{K_\mathcal{T}}(I) = \sum_{I \in \mathbb{I} : T \in I} 0_K = 0_K$$

$$\tau_T(1_{K_\mathcal{T}}) = \sum_{I \in \mathbb{I} : T \in I} 1_{K_\mathcal{T}}(T)$$

Since $T \in [T_{min}, T_{max})$ for any $T \in \mathbb{T}$ and $1_{K_\mathcal{T}}(I) = 0_K$ for any interval I except for $[T_{min}, T_{max})$ where $1_{K_\mathcal{T}}([T_{min}, T_{max})) = 1_K$ we get $\sum_{I \in \mathbb{I} : T \in I} 1_{K_\mathcal{T}}(T) = 1_K$
Commutes with addition:

$$\tau_T(k +_\mathcal{T} k') = \sum_{I \in \mathbb{I} : T \in I} (k +_\mathcal{T} k')(I) = \sum_{I \in \mathbb{I} : T \in I} k(I) +_K k'(I)$$

$$= \sum_{I \in \mathbb{I} : T \in I} k(I) +_K \sum_{I \in \mathbb{I} : T \in I} k'(I) = \tau_T(k) + \tau_T(k')$$

Commutes with multiplication:

$$\tau_T(k \cdot_\mathcal{T} k') = \sum_{I \in \mathbb{I} : T \in I} (k \cdot_\mathcal{T} k')(I)$$
$$= \sum_{I \in \mathbb{I} : T \in I} \sum_{\forall I', I'' : I = I' \cap I''} k(I') \cdot_K k'(I'')$$
$$= \sum_{\forall I', I'' : T \in I' \wedge T \in I''} k(I') \cdot_K k'(I'')$$

Let $n_1, \ldots, n_l$ denote the elements $k(I)$ for all intervals from the set of intervals with $T \in I$ and $k(I) \neq 0$. Analog, let $m_1, \ldots, m_o$ bet the set of elements with the same property for k'. Then the sum can be rewritten as:

$$= \sum_{i=1}^{l} \sum_{j=1}^{o} n_i \cdot_K m_j = \sum_{i=1}^{l} n_i \cdot_K (\sum_{j=1}^{o} m_j) = (\sum_{i=1}^{l} n_i) \cdot_K (\sum_{j=1}^{o} m_j)$$

replacing this again with the interval notation we get:

$$= (\sum_{\forall I : T \in I} k(I)) \cdot_K (\sum_{\forall I : T \in I} k'(I)) = \tau_T(k) \cdot_K \tau_T(k') \qquad \square$$

Proof of Lemma 6.4. injective: We have to show that for any two snapshot K-relations R and R', $\textsc{Enc}_K(R) = \textsc{Enc}_K(R') \Rightarrow R = R'$. Since, $\mathcal{C}_K$ preserves snapshot equivalence and is a unique representation of any temporal K-element $\mathcal{T}$, it is sufficient to show that for all t, we have $\mathcal{T}_{R,t} = \mathcal{T}_{R',t}$ instead. For sake of contradiction, assume that there exists a tuple t such that $\mathcal{T}_{R,t} \neq \mathcal{T}_{R',t}$. Then there has to exist $T \in \mathbb{T}$ such that $\mathcal{T}_{R,t}([T, T+1)) \neq \mathcal{T}_{R',t}([T, T+1))$. However, based on the definition of $\mathcal{T}_{R,t}$ this implies that $R(T)(t) \neq R'(T)(t)$ which contradicts the assumption.
surjective: Given a $K_\mathcal{T}$-relation R, we construct a snapshot K-relation R' such that $\textsc{Enc}_K(R') = R$: $R'(T)(t) = \sum_{T \in I} R(t)(I)$. □

Proof of Lemma 6.5. By virtue of snapshot equivalence between $\mathcal{C}_K(\mathcal{T})$ and $\mathcal{T}$ and based on the singleton interval definition of $\mathcal{T}_{R,t}$ in $\textsc{Enc}_K$, we have for any tuple t:

$$\tau_T(\textsc{Enc}_K(R))(t) = \tau_T(\mathcal{T}_{R,t}) = \mathcal{T}_{R,t}([T, T+1)) = R(T)(t)$$
$$= \tau_T(R)(t) \qquad \square$$

Proof of Theorem 6.6. We have to show that $(\mathcal{DB}_{K_\mathcal{T}}, {\textsc{Enc}_K}^{-1}, \tau)$ fulfills conditions (1), (2), and (3) of Definition 4.5 to prove that this triple is a representation system for K-relations. Conditions (1) and (2) have been proven in Lemmas 6.4 and 6.5, respectively. Condition (3) follows from the fact that τ_T is a homomorphism (Theorem 6.3) and that semiring homomorphisms commute with $\mathcal{RA}^+$-queries ([20], Proposition 3.5). □

Proof of Theorem 7.1. To prove that $K_\mathcal{T}$ has a well-defined monus, we have to show $K_\mathcal{T}$ is naturally ordered and that for any k and k', the set $\{k'' \mid k \preceq_{K_\mathcal{T}} k' + k''\}$ has a unique smallest element according to $\preceq_{K_\mathcal{T}}$. A semiring is a naturally ordered if $\preceq_{K_\mathcal{T}}$ is a partial order (reflexive, antisymmetric, and transitive). $k \preceq_{K_\mathcal{T}} k' \Leftrightarrow \exists k'' : k +_{K_\mathcal{T}} k'' = k'$. Substituting the definition of addition, we get $\exists k'' : \mathcal{C}_K(k +_{K_\mathcal{P}} k'') = k'$. Since $\mathcal{C}_K(k') = k'$ and coalesce preserves snapshot equivalence, we have $\mathcal{C}_K(k +_{K_\mathcal{P}} k'') = k' \Leftrightarrow \forall T \in \mathbb{T} : \tau_T(k) +_K \tau_T(k'') = \tau_T(k')$. From $\mathcal{C}_K(k +_{K_\mathcal{P}} k'') = k' \Leftrightarrow \forall T \in \mathbb{T} : \tau_T(k) +_K \tau_T(k'') = \tau_T(k')$ follows that $k \preceq_{K_\mathcal{T}} k' \Leftrightarrow \forall T \in \mathbb{T} : \tau_T(k) \preceq_K \tau_T(k')$.

Note that we only have to prove that $\preceq_{K_\mathcal{T}}$ is antisymmetric, since reflexivity and transitivity of the natural order follows from the semiring axioms and, thus, holds for all semirings.
Antisymmetric: We have to show that $\forall k, k' \in K_\mathcal{T} : k \preceq_{K_\mathcal{T}} k' \wedge k' \preceq_{K_\mathcal{T}} k \rightarrow k = k'$. This holds because, $k \preceq_{K_\mathcal{T}} k'$ and $k' \preceq_{K_\mathcal{T}} k$ iff for all $T \in \mathbb{T}$ we have $\tau_T(k) \preceq_K \tau_T(k')$ and $\tau_T(k') \preceq_K \tau_T(k)$ which implies $\tau_T(k) = \tau_T(k')$ for all $T \in \mathbb{T}$ which can only be the case if $k \sim k'$. Since k and k' are coalesced it follows that $k = k'$.
Unique Smallest Element Exists: It remains to be shown that $\{k'' \mid k \preceq_{K_\mathcal{T}} k' + k''\}$ has a smallest member for all $k, k' \in K_\mathcal{T}$. We

give a constructive proof by constructing the smallest such element k_{min}. k_{min} is defined by coalescing an element k_{pmin} that consists of singleton intervals ($[T, T+1)$) as follows:

$$k_{min} = \mathcal{C}_K(k_{pmin})$$

$$\forall I \in \mathbb{I} : k_{pmin}(I) = \begin{cases} \tau_T(k) -_K \tau_T(k') & \text{if } I = [T, T+1) \\ 0_K & \text{else} \end{cases}$$

First we have to demonstrate that indeed $k \preceq_{K_\mathcal{T}} k' +_K k_{min}$. Recall that $\mathcal{C}_K(k' +_{K_\mathcal{P}} k_{min}) = k \Leftrightarrow \forall T \in \mathbb{T} : \tau_T(k') + \tau_T(k_{min})) = \tau_T(k')$. Substituting the definition of k_{min} and using the fact that τ_T commutes with addition, for every time point T we distinguish two cases. Either $\tau_T(k') \succeq_K \tau_T(k)$ in which case $\tau_T(k) -_K \tau_T(k') = 0_K$ and we have: $\tau_T(k') +_K (\tau_T(k) -_K \tau_T(k')) = \tau_T(k') + 0_K = \tau_T(k')$. Thus, $\tau_T(k') +_K (\tau_T(k) -_K \tau_T(k')) \succeq_K k\tau_T(k')$ reduces to $\tau_T(k') \succeq_K \tau_T(k)$ which was assumed to hold.

Otherwise for $\tau_T(k') \preceq_K \tau_T(k)$ we have: $\tau_T(k') +_K (\tau_T(k) -_K \tau_T(k'))$. Let $k'' = (\tau_T(k) -_K \tau_T(k'))$. Substituting the definition of $-_K$, we get $k'' = min_{k'''}\tau_T(k') + k''' \succeq_K \tau_T(k)$. Thus,

$$\tau_T(k') +_K k'' \succeq_K k.$$

It remains to be shown that k_{min} is minimal. For contradiction assume that there exists a smaller such member k_{alt}. Then there has to exist at least one time point T such that $\tau_T(k_{alt}) \prec_K \tau_T(k_{min})$. We have to distinguish two cases. If $\tau_T(k) \preceq_K \tau_T(k')$, then $\tau_T(k_{min}) = 0_K$. However, since $0_K \leq k$ for any $k \in K$ this leads to a contradiction. Otherwise $\tau_T(k') + \tau_T(k_{alt}) \prec_K \tau_T(k') +_K \tau_T(k_{min}) = \tau_T(k)$ contradicting the assumption that $k \preceq_K k' +_K k_{alt}$. □

Proof of Theorem 7.2. We have to prove that $\tau_T(k -_{K_\mathcal{T}} k') = \tau_T(k) -_K \tau_T(k')$. We start with $\tau_T(k -_{K_\mathcal{T}} k') = \tau_T(\mathcal{C}_K(k -_{K_\mathcal{P}} k'))$. Since $\mathcal{C}_K$ preserves $\sim$ and $\tau_T(k) = \tau_T(k')$ if $k \sim k'$, we get:

$$= \sum_{T \in I} (k -_{K_\mathcal{P}} k')(I) = \qquad \tau_T(k) -_K \tau_T(k') \qquad \square$$

Proof of Theorem 7.3. By construction, the result of aggregation is a $\mathbb{N}_\mathcal{T}$ relation (it is coalesced). Also by construction, we have $\tau_T({}_G\gamma_{f(A)}(R)) = {}_G\gamma_{f(A)}(\tau_T(R))$. □

Proof of Theorem 8.1. To prove the relationships in the commutative diagram of Equation (1), we have to prove that $\textsc{PeriodEnc}^{-1}(\textsc{PeriodEnc}(R)) = R$ and that queries commute with $\textsc{PeriodEnc}$ if rewritten using $\textsc{Rewr}$, i.e., $\textsc{PeriodEnc}(Q(R)) = \textsc{Rewr}(Q)(\textsc{PeriodEnc}(R))$.

$\textsc{PeriodEnc}^{-1}(\textsc{PeriodEnc}(R)) = R$: Let R be a $\mathbb{N}_\mathcal{T}$-relation and R' denote $\textsc{PeriodEnc}(R)$. Consider an arbitrary tuple t and let $\mathcal{T}$ denote the temporal element associated with t, i.e., $R(t) = \mathcal{T}$. Consider any interval $I \in \mathbb{I}$ and let $n_I = \mathcal{T}(I)$ (the multiplicity assigned by $\mathcal{T}$ to I). According to Definition 8.1, this implies that tuple $t_I = (t, I^+, I^-)$ is annotated with n_I. Let $\mathcal{T}_t$ denote the temporal element assigned by $\textsc{PeriodEnc}^{-1}$ to t. By construction $\mathcal{T}_t(I) = n_I = \mathcal{T}(I)$.

$\textsc{PeriodEnc}(Q(R)) = \textsc{Rewr}(Q)(\textsc{PeriodEnc}(R))$: We prove this part by induction over the structure of a query. Let $R' = \textsc{PeriodEnc}(R)$.

Base case: Assume that $Q = R$ for some relation R. The claim follows immediately from $\textsc{Rewr}(R) = R$.

Induction Step: Assume the claim holds for queries with up to n operators. We have to prove the claim for any query Q with $n+1$ operators. For unary operators, WLOG let $Q = op(Q_n)$ for an operator op and query Q_n with n operators and let $Q' = \textsc{Rewr}(Q)$.

Selection: $op = \sigma_\theta$: A selection is rewritten as $Q' = \mathcal{C}(\sigma_\theta(\textsc{Rewr}(Q)))$. Consider an input tuple t from R. The temporal K-element $\mathcal{T}$ annotating tuple t is represented as a set of tuples of the form (t, I^+, I^-) for some interval I. If t fulfills the selection, then t is annotated with $\mathcal{T}$ in the result. In R', all of these tuples are in the result of Q' if $t \models \theta$ and applying $\textsc{PeriodEnc}^{-1}$ we get $\mathcal{T}$ as the annotation of t. If t does not fulfill the condition then t is annotated with 0 in both encodings.

Projection: $op = \Pi_A$: A projection is rewritten by adding the attributes encoding the interval associated to a tuple to the projection expressions. There will be one tuple (t, I^+, I^-) in the result for each interval I assigned a non-zero annotation in $R(u)$ for any tuple u projected on tuple t. Function $\textsc{PeriodEnc}^{-1}$ creates the annotation of an output as a temporal element that maps each interval mapping to a non-zero annotation in $\textsc{PeriodEnc}(R)$ to that annotation. This corresponds to addition of singleton temporal elements and based on the fact that addition is associative this implies that the annotation of t in the output will be the sum of temporal elements $R(u)$ for each u projected onto t. Thus, the claim holds.

Aggregation: $op = \gamma_{f(A)}$: The rewrite for aggregation without group-by utilizes the split operation $\mathcal{N}$ we have defined. Note that $\mathcal{N}_\emptyset$ returns a $\mathbb{N}_\mathcal{T}$-relation S where for any pair of tuples t and t' and any pair of intervals I_1 and I_2 we have $I_1 \neq I_2 \wedge I_1 \cap I_2 \neq \emptyset \Rightarrow S(t')(I_1) = 0 \vee S(t')(I_2) = 0$. That is, all intervals with non-zero annotations from any pair of temporal elements do not overlap or are the same. From that follows that for any two time points $T_1, T_2 \in I$ for an interval I that is mapped to $n \neq 0$ in the annotation of at least one tuple S, the value of the result of aggregation is the same for the snapshots at T_1 and T_2. Thus, grouping by the interval boundaries yields the expected result with the exception of an empty snapshot. However, since a tuple $(0_f, T_{min}, T_{max})$ is added to the input, the aggregation will produce 0 (count) or **NULL** (other aggregation functions) for intervals containing only empty snapshots. This does not effect the result of the aggregation for non-empty snapshots, because 0_f is the neutral element of the aggregation function f.

Aggregation: $op = {}_G\gamma_{f(A)}$: For aggregation with group-by, split is applied grouping on G and no additional tuple $(0_f, T_{min}, T_{max})$ is added to the input. Since the tuples within one group are split, the argument we have used above for aggregation without group-by applies also to aggregation with group-by.

For binary operators WLOG let $Q = op(Q_l, Q_r)$ where the total number of operators in Q_l and Q_r is n.

Join: $op = Q_l \bowtie_\theta Q_r$: Consider a tuple t that is the result of joining tuples u and v. Let $\mathcal{T}_u$ and $\mathcal{T}_v$ be the temporal elements annotating u and v in the input, respectively. Based on the definition of the rewriting, in the result of the rewritten join there will be a tuple t, I^+, I^- annotated with $\sum_{I_u, I_v} Q_l(u) \cdot Q_r(v)$ for all intervals I_u and I_v such that $I = I_u \cap I_c$. This corresponds to the definition of multiplication (join) in $\mathbb{N}_\mathcal{T}$.

Union: $op = Q_l \cup Q_r$: Union is rewritten as a union of the rewritten inputs. For any tuple t, let $\mathcal{T}_l = Q_l(t)$ and $\mathcal{T}_r = Q_r(t)$. In the result of the union applied by Q' a tuple (t, I^+, I^-) for each interval I will be annotated with $\mathcal{T}_l(t) + \mathcal{T}_r(t)$. The result of the union is then coalesced. Applying $\textsc{PeriodEnc}^{-1}$ the annotation computed for t is equivalent to $\mathcal{C}_\mathbb{N}(\mathcal{T}_l +_{K_\mathcal{P}} \mathcal{T}_r)$.

Difference: $op = Q_l - Q_r$: A difference is rewritten by applying difference to the pairwise normalized inputs. Recall that the monus operator of $\mathbb{N}_\mathcal{T}$ associates the result of the monus for $\mathbb{N}$ to each snapshot of a temporal $\mathbb{N}$-element. Since the split operator adjusts intervals such that there is no overlap, the claim holds. □

B. QUERY DESCRIPTIONS

B.1 MySQL Employee Dataset

join-1. Return the salary and department for every employee.

```
SELECT a.emp_no, dept_no, salary
FROM dept_emp a JOIN salaries b ON (a.emp_no = b.emp_no)
```

join-2. Return the department, salary, and title for every employee.

```
SELECT title, salary, dept_no
FROM dept_emp a JOIN salaries b ON (a.emp_no = b.emp_no)
                JOIN titles c ON (a.emp_no = c.emp_no)
```

join-3. Return employees that manage a particular department and earn more then $70,000.

```
SELECT a.emp_no, dept_no
FROM dept_manager a
     JOIN salaries b ON (a.emp_no = b.emp_no)
WHERE salary > 70000
```

join-4. Returns information about the manager of each department.

```
SELECT a.emp_no, a.dept_no, b.salary, first_name,
       last_name
FROM dept_manager a, salaries b, employees e
WHERE a.emp_no = b.emp_no and a.emp_no = e.emp_no
```

agg-1. Returns the average salary of employees per department.

```
SELECT dept_no, avg(salary) as avg_salary
FROM dept_emp a
     JOIN salaries b ON (a.emp_no = b.emp_no)
GROUP BY dept_no
```

agg-2. Returns the average salary of managers.

```
SELECT avg(salary) as avg_salary
FROM dept_manager a
     JOIN salaries b ON (a.emp_no = b.emp_no)
```

agg-3. Returns the number of departments with more than 21 employees.

```
SELECT count(1)
FROM (SELECT count(*) AS c, dept_no
    FROM dept_emp WHERE emp_no < 10282
    GROUP BY dept_no HAVING count(*) > 21) s
```

agg-join. Returns the names of employees with the highest salary in their department. It contains a 4-way join where one of the join inputs is the result of a subquery with aggregation.

```
SELECT d.emp_no, e.first_name, e.last_name,
       maxS.max_salary, d.dept_no
FROM (SELECT max(salary) as max_salary,dept_no
      FROM dept_emp a
           JOIN salaries b ON (a.emp_no = b.emp_no)
      GROUP BY dept_no) maxS,
      salaries s, dept_emp d, employees e
WHERE e.emp_no = s.emp_no
      AND s.salary = maxS.max_salary
    AND d.dept_no = maxS.dept_no
      AND d.emp_no = e.emp_no
```

diff-1. Returns employees that are not managers of any department.

```
SELECT emp_no FROM dept_emp
EXCEPT ALL
SELECT emp_no FROM dept_manager
```

diff-2. Returns salaries of employees that are not managers.

```
SELECT a.emp_no, salary
FROM (SELECT emp_no FROM dept_emp
      EXCEPT ALL
      SELECT emp_no FROM dept_manager) a
      JOIN salaries b ON (a.emp_no = b.emp_no)
```

C-Sn. To evaluate the performance of coalescing we use the following query template varying the selection condition on salary to control the size of the output. The query returns employee salaries. We materialize the result of this query for each selectivity and use this as the input to coalescing.

```
SELECT a.EMP_NO, salary
FROM employees a
     JOIN salaries b ON (a.emp_no = b.emp_no)
WHERE salary > ?
```

B.2 Tourism Dataset

Recall that this dataset stores travel booking inquiries in the South Tyrol area in Italy.

tou-join-agg. This query returns for each destination the sum of the number of persons of all bookings for this destination paired with the average number of this sum for all other destinations.

```
WITH numPerDest AS (
    SELECT sum(adults + children) AS numT,
           destination AS dest
    FROM tourismdata
    GROUP BY destination
)
SELECT n.numT, n.dest, avg(o.numT) AS otherAvg
FROM numPerDest n, numPerDest o
WHERE n.dest <> o.dest
GROUP BY n.numT, n.dest
```

tou-agg-1. For destinations with more than 1000 inquiries return the number of inquiries for this destination and the total number of persons for which inquiries were made.

```
SELECT destination,
       count(*) AS numEnquiry,
       sum(adults + children) AS numTourists
FROM tourismdata
GROUP BY destination
HAVING count(*) > 1000;
```

agg-2. For each destination return the maximum number of persons per inquiry.

```
SELECT destination,
       max(adults + children) AS maxTourists
FROM tourismdata
GROUP BY destination;
```

tou-agg-3. Find the maximum number of inquiries from the total number of inquiries per destination.

```
SELECT max(cnt) AS maxInq
FROM (SELECT count(*) AS cnt
      FROM tourismdata
      GROUP BY destination)
```

agg-join. This query returns the total number of inquiries per continent.

```
SELECT continent, count(*) AS numEnquiries
FROM tourismdata t, country c
WHERE t.countrycode = c.countrycode
GROUP BY continent
```

C. PULLING-UP COALESCING

The main overhead of our approach for sequenced temporal queries compared to non-temporal query processing is the extensive use of coalescing, which can be expensive if naively implemented in SQL. Furthermore, the application of coalescing after each operation may prevent the database optimizer from applying standard optimizations such as join reordering. To address this issue, we now investigate how to reduce the number of coalescing steps. In fact, we demonstrate that it is sufficient to apply coalescing as a last step in query processing instead of applying it to intermediate results. Similar optimizations have been proposed by Bowman et al. [12] for their multiset temporal normalization operator and by Böhlen et al. [10] for set-coalescing.

Consider how a $\mathcal{RA}^+$ query Q is evaluated over an $K_\mathcal{T}$-database. $\mathcal{RA}^+$ over K-relations computes the annotation of a tuple in the result of a query using the addition and multiplication operations of the semiring. That is, the annotation of any result tuple is computed using an arithmetic expression over the annotations of tuples from the input of the query. In the case of a semiring $K_\mathcal{T}$, addition and multiplication are defined as coalescing a temporal element that is computed based on point-wise application of the addition (multiplication) operations of semiring K (denoted as $+_{K_\mathcal{P}}$ and $\cdot_{K_\mathcal{P}}$). Recall from Lemma 6.1 that coalescing can be redundantly pushed into the addition and multiplication operations of interval-temporal semirings, e.g., $\mathcal{C}_K(k +_{K_\mathcal{P}} k') = \mathcal{C}_K(\mathcal{C}_K(k) +_{K_\mathcal{P}} k')$. Interpreting this equivalence from right to left and applying it repeatedly to an arithmetic expression e using $+_{K_\mathcal{T}}$ and $\cdot_{K_\mathcal{T}}$, the expression can be rewritten into an equivalent expression of the form $\mathcal{C}_K(e')$, where e' is an expression that only uses operations $+_{K_\mathcal{P}}$ and $\cdot_{K_\mathcal{P}}$. Now consider expressions that also include applications of the monus operator $-_{K_\mathcal{T}}$. This operator is defined as $\mathcal{C}_K(k -_{K_\mathcal{P}} k')$. The $-_{K_\mathcal{P}}$ operator computes the timeslice of the inputs at every point in time and then applies $-_K$ to each timeslice. According to Lemma 5.1, $\tau_T(k) \sim \tau_T(\mathcal{C}_K(k'))$. Thus, the result of $-_{K_\mathcal{P}}$ is independent of whether the input is coalesced or not.

Lemma C.1. *Any arithmetic expression* e *using operations and elements from an period m-semiring $K_\mathcal{T}$ is equivalent to an expression of the form $\mathcal{C}_K(e')$, where e' only contains operations $+_{K_\mathcal{P}}$, $\cdot_{K_\mathcal{P}}$, and $-_{K_\mathcal{P}}$.*

Lemma C.1 implies that it is sufficient to apply coalescing as a last step in a rewritten query $\text{REWR}(Q)$ instead of after each operator.

Corollary C.2 (Coalesce Pullup)**.** *For any $\mathcal{RA}$ query Q, $\text{REWR}(Q)$ is equivalent to a query Q' which is derived from $\text{REWR}(Q)$ by removing all but the outermost coalescing operator.*

Proof. Operations $+_{K_\mathcal{T}}$, $\cdot_{K_\mathcal{T}}$, and $-_{K_\mathcal{T}}$ are defined as applying $\mathcal{C}_K$ to the result of operations $+_{K_\mathcal{P}}$, $\cdot_{K_\mathcal{P}}$, and $-_{K_\mathcal{P}}$, respectively. Thus, expression e is equivalent to an expression that interleaves the $\mathcal{C}_K$ as well as $+_{K_\mathcal{P}}$, $-_{K_\mathcal{P}}$, and $\cdot_{K_\mathcal{P}}$ operations. To prove this, we first prove that the following equivalence holds: $\mathcal{C}_K(k -_{K_\mathcal{P}} k') \Leftrightarrow \mathcal{C}_K(\mathcal{C}_K(k) -_{K_\mathcal{P}} k') \Leftrightarrow \mathcal{C}_K(k -_{K_\mathcal{P}} \mathcal{C}_K(k'))$. Consider the definition of $-_{K_\mathcal{P}}$. Every interval $I = [T, T+1)$ is assigned the annotation $\tau_T(k) -_K \tau_T(k')$. Applying Lemma 5.1 we get $\tau_T(k) = \tau_T(\mathcal{C}_K(k))$ and $\tau_T(k') = \tau_T(\mathcal{C}_K(k'))$. Thus, the equivalence holds. By repeatedly applying this equivalence and the equivalences proven in Lemma 6.1, all except the outermost K-coalesce operations can be removed resulting in an expression of the form $\mathcal{C}_K(e')$ where e' does not contain any coalesce operations. □

Example C.1. *Consider the following query $Q = S - \Pi_{sal}(\sigma_{sal<sal'}(S \times \rho_{sal' \leftarrow sal}(S))$ that returns the largest salary from relation S as shown in Figure 3 (consider the corresponding $\mathbb{N}_\mathcal{T}$-relation using the annotation shown on the right in this figure coalesced as shown in Example 5.3). Consider how the annotation of tuple $r = (50k)$ in the result of Q is computed. Applying the definitions of difference, projection, and join over K-relations and denoting the database instance of S as D, we obtain:*

$$Q(D)(t) = \mathcal{C}_{\mathbb{N}}(S(t) -_{K_\mathcal{P}} \mathcal{C}_{\mathbb{N}}(\sum_{u=(v,w):u.sal=t} \mathcal{C}_{\mathbb{N}}(\mathcal{C}_{\mathbb{N}}((S(v) \cdot_{K_\mathcal{P}} S(w))) \cdot_{K_\mathcal{P}} (sal < sal')(u))))$$

Pulling up coalesce we get:

$$Q(D)(t) = \mathcal{C}_{\mathbb{N}}(S(t) -_{K_\mathcal{P}} \sum_{u=(v,w):u.sal=t} (S(v) \cdot_{K_\mathcal{P}} S(w)) \cdot_{K_\mathcal{P}} (sal < sal')(u))$$

D. INTERACTION OF OUR APPROACH WITH QUERY OPTIMIZATION

In this section we briefly discuss the impact of our rewrite-based approach for implementing sequenced semantics on query optimization. Importantly, the combination of uniqueness and snapshot reducibility guarantees that queries are equivalent wrt. our logical model precisely when they are equivalent under regular K-relational semantics. As a special case of this result, queries over $\mathbb{N}_\mathcal{T}$ relations are equivalent iff they are equivalent under bag semantics ($\mathbb{N}$-relations). That is, any query equivalence that is applied by classical database optimizers, e.g., join reordering, can be applied to optimize sequenced queries.

That being said, we pass a rewritten query to the DBMS optimizer which is not aware of the fact that this query implements sequenced semantics. The preservation of bag semantics query equivalences does not necessary imply that these rewritten queries can be successfully optimized by a general purpose query optimizer. However, as we will explain in the following, our approach is designed to aid the database optimizer in finding a successful plan. First off, note that our rewrites essentially keep the structure of the input query intact with the exception of the introduction of split before aggregation and difference, and coalescing which is applied as a final step for every sequenced query. Every other operator is preserved in the rewritten query, e.g., joins, are rewritten into joins.

Example D.1. *Consider the following query $Q = \Pi_{name,city}(person \bowtie_{name=pName} livesAt \bowtie_{address=aId} address)$ over relations*

person(name, age, A_{Begin}, A_{End})
livesAt(pName, address, A_{Begin}, A_{End})
address(aId, city, zip, street, A_{Begin}, A_{End})

This query returns for each person the city(ies) they live in. Applying REWR *we get the query shown in Figure 6. Note how the* *structure of the input query was preserved. The exception are the coalescing operator at the end and the introduction of new projections. However, typically database optimizers will at least consider a transformation called subquery pull-up (called view merging in Oracle) which would pull-up and merge these projections. Thus, these projections do not hinder join reordering.*

E. SQL IMPLEMENTATIONS OF BAG COALESCING AND SPLIT

In the following, we explain our implementation of bag coalescing in SQL using a step by step example. Afterwards, we present the implementation of the split operator integrated with aggregation and (bag) difference.

E.1 Bag Coalesce

Figure 8 shows the SQL code for computing bag coalescing for a table recording the activity of production machines. Figure 9 shows an example instance of this table and the intermediate and final results produced by the query for this instance. Here we assume that periods are stored as two timestamp attributes $\boldsymbol{t_{start}}$ and $\boldsymbol{t_{end}}$ recording the start and the end of the period. The input table **active** with the schema (**mach**, $\boldsymbol{t_{start}}$, $\boldsymbol{t_{end}}$) is shown on the top-left of Figure 9. Each row in the table records a time interval (from $\boldsymbol{t_{start}}$ to $\boldsymbol{t_{end}}$) during which a machine (**mach**) is running. For convenience we show a timeline with the intervals encoded by this table.

Before explaining the steps of the SQL implementation, we review bag coalescing. To coalesce an input we have to determine for each tuple t its annotation change points, i.e., the end points of maximum intervals during which the multiplicity of the tuple does not change. Then for each adjacent pair of change points we output a number of duplicates of tuple t that is equal to the number of duplicates of tuple t in the input whose intervals cover the two change points. This could be implemented as a native operator which splits each tuples associated with a period into two tuples with the intervals end points where each generated tuple is marked to indicate whether it represents an interval start or end point. Then any aggregation algorithm can be applied to calculate the number of intervals associated with a tuple that open and close at a particular time point. The output of this step is then sorted on the non-temporal attributes and secondary on the timestamp attribute. The final result is produced by scanning through the sorted output once outputting for each tuple and adjacent pair of change points a number of duplicates determined based on the number of intervals covering these change points which is determined based on the counts of opening and closing intervals. We leave a native implementation and further optimizations (e.g., we could partition the input on the non-temporal attributes and then process multiple such partitions in parallel) to future work and now explain how our SQL implementation realizes the computational steps outlined above.

Determine the Number of Opening and Closing Intervals Per Change Point. In lines 3 - 24 of Figure 9 we compute the annotation change points for each tuple and the number of intervals that are opening and closing for each such change point. This is done by counting for each tuple and one of its change points the number of opening and closing intervals separately and then for each such pair merge the number of opening and closing counts into a single output tuple. Note that strictly speaking not all of the time points returned by this query are guaranteed to be annotation change points. The actual change points are computed in one of the following steps as explained below. For the example instance there is only one pair $(M1, 5)$ where time point T (attribute `t`) is both the start and end point of an interval associated with tuple $(M1)$. As another example consider time point 6 which is the end point of two intervals associated with tuple $(M2)$ corresponding to the last tuple in the result of subquery `change_points`. The pre-aggregation before the union is merely a performance tweak. It turns a single aggregation over $2 \cdot |active|$ tuples into two aggregations over $|active|$ tuples.

Counting Open Intervals. Line 26 - 35 of the query create the common table expression `num_intervals` which returns the number of open intervals for a tuple per time point T. This is achieved by subtracting the number of intervals for this tuple with an end point that is less than or equal to T (attribute `t`) from the number of intervals with a start point that is less than or equal to T. Intuitively, the number of open intervals for a tuple is the number of duplicates of the tuple that exist in the time interval between T and the adjacent following change point. We compute these running sums using SQL's window functions partitioning the input on the non-temporal attributes (`mach` in this example) and within each partition order the tuple based on the timestamp `t` computing the aggregate over a window including all tuples with a timestamp less than or equal to `t`. For example, consider the first tuples in the instance of `num_intervals` as shown in Figure 9. This tuple records that there two duplicates of tuple $(M1)$ exist at time point 1.

Removing Spurious Change Points. Recall that bag coalescing determines maximal intervals during which the annotation (multiplicity in the case of bag semantics) of a tuple is constant. As shown in the example, `num_intervals` may contain adjacent time points with the same number of open intervals which, according to the definition of K-coalescing, are not annotation change points. Subquery `diff_previous` (Lines 38-47) computes the difference between the number of open intervals at a time point and the previous time point. Subquery `changed_intervals` (Lines 49-53) removes tuples where this difference is zero (the number of duplicates has not changed).

Reconstructing Intervals. At this point in the computation we have calculated the set of annotation changepoints for each tuple and the number of duplicates of the tuple that exist during the time interval between each two adjacent annotation change points. Subquery `pair_points` (Lines 55-65) computes pairs of adjacent annotation change points. For instance, the first tuple in the result for the example records that there exists two duplicates of tuple `(M1)` during time interval $[1, 7]$. The subquery of `pair_points` also returns tuples with t_{start} equal to the last change point of teach tuple. These tuples are filtered in the **`WHERE`** clause of the outer query. For example, the tuples marked in red in the result of the subquery as shown in Figure 9 are such tuples.

Generating Duplicates. In the last step, we generate duplicates of tuples based on the counts stored in attribute $\#_{open}$. One way to realize this would be to use a set-returning function that takes as input a tuple t and a count c, and returns c duplicates of t. While perfectly viable, to avoid the overhead of calling a user-defined function for every distinct output tuple, we use subquery `max_seq` (Lines 67-72) to generate a table storing a sequence of numbers $\{1, \ldots, m\}$ where m is the maximum number of duplicates of any tuple in the query result. We then join this table with `pair_points` (lines 74-76). For the example database the maximum number of duplicates for any tuple and time point is 2. Hence, subquery `max_seq` returns $\{(1), (2)\}$. The final join with `pair_points` then returns the appropriate number of duplicates for each tuple using the counts stored in $\#_{open}$, e.g., there are 2 duplicates of tuple $(M2)$ during time interval $[3, 6]$.

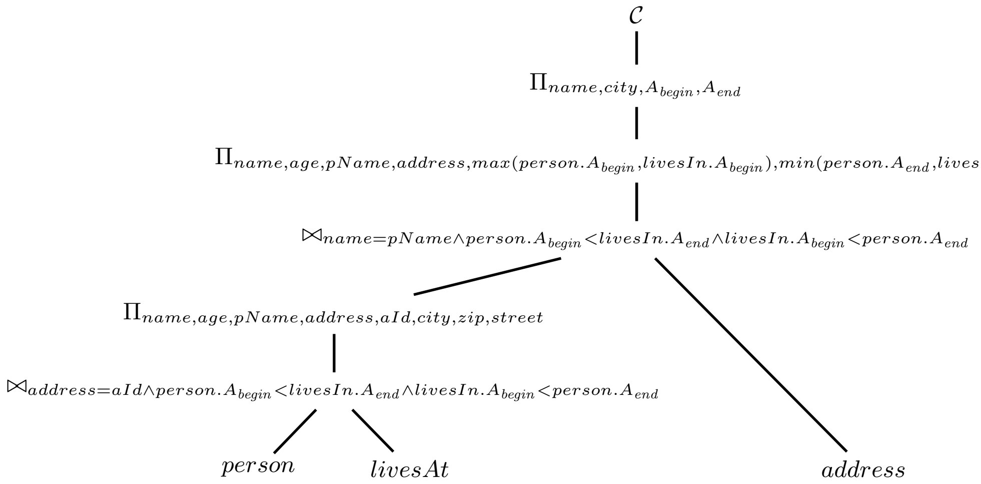

$\mathcal{C}$
$\Pi_{name,city,A_{begin},A_{end}}$
$\Pi_{name,age,pName,address,max(person.A_{begin},livesIn.A_{begin}),min(person.A_{end},livesIn.A_{end})}$
$\bowtie_{name=pName \wedge person.A_{begin}<livesIn.A_{end} \wedge livesIn.A_{begin}<person.A_{end}}$
$\Pi_{name,age,pName,address,aId,city,zip,street}$
$\bowtie_{address=aId \wedge person.A_{begin}<livesIn.A_{end} \wedge livesIn.A_{begin}<person.A_{end}}$
$person$
$livesAt$
$address$

Figure 6: Rewriting $\text{REWR}(Q)$ for SPJ query Q from Example D.1

E.2 Split Operator Implementation

We first introduce our implementation of the split operator and then afterwards discuss the optimized versions of the aggregation and difference which incorporate split. We show the SQL code for $\mathcal{N}_{mach}(active, active)$, i.e., splitting the intervals of relation `active` based on its own interval boundaries for attribute `mach` (the only attribute of this relation). Figure 10 shows the SQL code generated by our system and Figure 11 shows an example database the intermediate results of produced by the SQL implementation for this example. In lines 2-9 we assign aliases to the left and right input. For this particular example, both inputs are table `active`.

Computing Interval End Points. The first of the computation (lines 10-23) generates the set of all interval end points for both inputs. Note that for this example where a relation is split wrt. itself the four-way union is not necessary and can be replaced with a two-way union. In our implementation we apply this optimization, but for sake for the example we show the four-way union to illustrate how the approach would work when a relation is split wrt. to another relation.

Creating Unique Identifiers for Intervals. Next we assign a unique identifier to each tuple from the left input (lines 25-32). For instance, there are three such tuples in the example shown in Figure 11.

Pair Intervals with End Points. We now join the left input with all endpoints we have computed beforehand (lines 34-46) such that each interval from the left input is paired with all end points it contains with the exception of the maximum point in the interval. Intuitively the purpose of this step is to creating sufficiently many duplicates of each input tuples to be able to generate the split versions of the interval for this tuple. Furthermore, the end points we have paired with an interval will be the starting points of the split intervals. In Figure 11 we highlight tuples with colors to indicate which tuples correspond to the same input interval. For example, the first two tuples in the result of `split_points` correspond to the tuple with id 1 and the starting points of the two intervals this interval will be split into (end point 4 is contained in the interval $[1, 7[$).

Generating Split Intervals. Finally, we adjust the start (t_{start}) and end points (t_{end}) of each interval produced in the previous step (lines 48-52). The start point is set to the time point t (the time point from the set of interval end points we have paired with the interval) and the end point is the next larger time point associated with the same interval identifier (or the end point of the interval is no such time point exists). For example, for the first tuple from the result of subquery `split_points` we output tuple `(M1,1,4)`. As can be seen in the timeline representation of the result shown on the bottom right of Figure 11 in the result of split any two intervals associated with the same values of the non-temporal attributes are either equal or disjoint.

E.3 Combining Split with Temporal Aggregation

There is synergy in combining the split operator with temporal aggregation. The resulting implementation is similar to temporal aggregation algorithms which utilize end point indexes (e.g., aggregation over a timeline index [25]). These approaches calculate the result of an aggregation function over time using "sweeping" by sorting the endpoints of intervals on time and then scan over the data in sort order adding the values of tuples whose intervals start at the current point in time to the current aggregation result and subtract the values of tuples whose intervals end at this point in time. Note that this only can be applied to aggregation functions like sum and count where it is possible to retract a value (the underlying function, e.g., addition in the case of sum, has an inverse). For aggregation functions min and max it is necessary to maintain a list of previously seen values (although it is not necessary to keep all previous values [31]). We do not use the sweeping technique for min and max, but still apply the pre-aggregation optimization described below. We explain how to combine split with aggregation using the example query shown in Figure 13 which computes the average consumption (`consum`) of machines.

Pre-aggregation. For aggregation functions like sum and count that are commutative, associative, and where the underlying operation has an inverse, we can compute pre-aggregate the input data before computing split points. For that we group on the input query's group-by attribute plus the attributes t_{start} and t_{end} which store the end points of a tuple's period. The pre-aggregation step return partial aggregation results for each list of group-by attribute values and period that occurs with this group. During split these periods may be further subdivided and the final aggregation results will be computed by accumulating results for these subdivisions. For aggregation functions like average that do not fulfill the conditions required for pre-aggregation, but which can be computed by evaluating an arithmetic expression over the result of other aggregation functions that do, we can still apply this trick to calculate the other aggregation functions and delay the computation of the aggregation we are actually interested in until the end. For example, the query shown in Figure 13 computes an average that can be computed as $sum/count$. Thus, as shown in lines 2-12 of Figure 12 we compute two aggregation functions grouping on mach, t_{start}, and t_{end}. The example instance of table active contains two tuples belonging to the same group which also have the same period: (M1,10,1,5) and (M1,20,1,5). Based on these two tuples we compute the pre-aggregated result (M1,30,1,5). Note that no matter what aggregation function we are computing, we always will also compute count since it is needed later in the implementation to determine intervals without results for aggregation with group-by.

Calculate Increase and Decrease of Aggregation Values. Our approach for computing aggregation functions sum and count uses a sweeping technique which scans over the set of all interval end points paired with in time order. We keep a partial aggregation result and for each time point adds the values of the aggregation input attribute for tuples with intervals that open at this time point and "retracts" the values of aggregation input attributes for tuples with intervals that close at this time point. For this purpose, we aggregate to total increase (opening intervals) and retraction (closing intervals) for each time point and group. Consider lines 14-38 in Figure 12. Since we are computing aggregation functions sum and count, we store for each time point the increase/decrease for both functions. For that, we use attributes add_c and dec_c (count) and add_s and dec_s (sum). For interval start points we set attributes recording decrease to 0 while for end points we points we set the add_* attributes to 0. Afterwards, we compute the total increase and decrease per time point using aggregation. For instance, consider time point 5 in the example shown in Figure 14. Two intervals with a total consumption of 30 close at this time point and one new interval opens with a consumption of 40. This is encoded in the third tuple (M1,1,40,2,30,5) in the result of subquery increase_decrease.

Compute Accumulative Totals. We then calculate the aggregation function result for each group and each point in time where at least one interval for this group starts or ends as the sum of the increases up to and including this point in time and subtract from that the sum of decreases. For example, the third tuple in the result of subquery accumulation shows that at time 5 there are 2 open intervals with their consum values summing up to 80.

Generate Output Intervals. Finally, we pair each split point and its count and sum with the following split point to produce output intervals and compute the average as the sum divided by the count. This is realized by the inner query of the subquery shown in lines 56-69 in Figure 12. Note that it may be the case that no periods start at a given split point. In this case the count would be 0 (no intervals open during between this time point and the next split point). This is dealt with by the **WHERE** clause of the outer query which filters out tuples where the count is 0.

Aggregation Without Group-by. Recall that for aggregation without group-by we have to return results for time periods where the relation is empty. This is easily achieved in our implementation by adding a dummy interval $[T_{min}, T_{max}]$ associated with the neutral value of the aggregation function to the result of subquery pre_agg (0 for count and *null* otherwise). For time periods where the input relation is empty the split operator creates an interval covering the "gap" and will return the value we did associate with the dummy interval which is chosen to correspond to the result of an aggregation over an input relation as defined in the SQL standard. For periods where the input is non-empty the result is not affected since the dummy interval is associated with the neutral value of an aggregation function. An additional change that is required is that the final **WHERE** clause (Figure 12, line 68) has to be changed to t_{end} **IS NOT NULL** to (i) return results for gaps (where the count is 0) and not return a tuple where t_{start} is the last split point (equal to T_{max} for the case of aggregation without group-by).

E.4 Combining Split with Difference

To explain the combined implementation of split with bag difference we evaluate the example query shown in Figure 16 under sequenced semantics. The query returns all machines and their consumption removing consumptions of machines which have been incorrectly recorded (table faulty). The SQL implementation for the sequenced version of this query which uses combined split and difference is shown in Figure 15. We show an example instance and intermediate results for the query in Figure 17. We combine the split operator with bag difference by reducing bag difference to the problem of count aggregation. Consider a snapshot at time T and tuple t and assume that t appears in the left input with multiplicity n and in the right input with multiplicity m at T. Then we have to return $max(0, n - m)$ duplicates of tuple t for this snapshot. This can be achieved by computing counts for each interval end point in the left and in the right input and then subtracting the counts of the right hand side from the counts of the left hand side. The combination of counting and split essentially uses the approach described in Appendix E.3.

Computing Changes in Multiplicities. Subquery end_point_counts (Figure 15, lines 15-36) computes the number of opening and closing intervals for both inputs. We count the end points from the right input negatively. For instance, in the example the second tuple in the result of this subquery records that there are two opening intervals for tuple (M1, 40) at time 1.

Aggregate Multiplicities. Next, we use subquery acc_counts aggregate the multiplicities to get a single count of opening and closing intervals per time point (lines 49-63). Note that in the result of this subquery both $\#_{open}$ and $\#_{close}$ may be negative. This has to be interpreted as that there is a larger number of opening/closing intervals from the right input than the left input.

Generating Intervals. We now pair adjacent time points (lines 49-63) to create intervals and compute the final multiplicity for each tuple.

Final Result. To compute the final result of the difference operator we have to create the right amount of duplicates for each tuple. The method we apply here is exactly the same as the one applied for aggregation: we join the result of subquery intervals with a table contain numbers 1 to n where n is the maximum multiplicity across all tuples and time points.

E.5 Coalesce after Split

We can also apply coalesce (introduced in Section E.1) after split (introduced in Section E.2), for example, we apply split the table **active** with the schema (**mach**, $\boldsymbol{t_{start}}$, $\boldsymbol{t_{end}}$) and apply coalesce afterwards, Figure 7 shows this workflow.

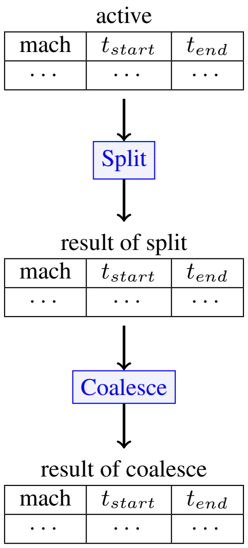


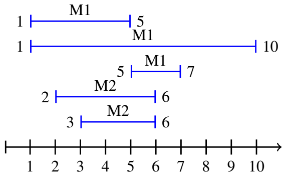
Figure 7: Coalesce after split

```
-- Count opening/closing intervals per change point
WITH
change_points (mach, #start, #end, t) AS
(
    SELECT mach,
           sum(#start) AS #start,
           sum(#end) AS #end,
           t
    FROM (
          SELECT mach,
                 count(*) AS #start,
                 0 AS #end,
                 t_start AS t
          FROM active
          GROUP BY t_start, mach
          UNION ALL
          SELECT mach,
                 0 AS #start,
                 count(*) AS #end,
                 t_end AS t
          FROM active
          GROUP BY t_end, mach)
    GROUP BY t, mach
),
-- Count the open intervals per tuple and time point
num_intervals (mach, #open, t) AS
(
    SELECT DISTINCT mach,
           sum(#start) OVER w
           - sum(#end) OVER w AS #open,
           t
    FROM change_points
    WINDOW w AS (PARTITION BY mach ORDER BY t
                 RANGE UNBOUNDED PRECEDING)
),
-- Compute the difference between the number of open
-- intervals at t and at the previous change point
diff_previous (mach, #open, diffPrevious, t) AS
(
    SELECT mach,
           #open,
           COALESCE(#open - (lag(#open,1) OVER w,
                    -1) AS diffPrevious,
           t
    FROM num_intervals
    WINDOW w AS (PARTITION BY mach ORDER BY t)
),
-- Remove unchanged intervals
changed_intervals (mach, t, #open, diffPrevious) AS
(
    SELECT * FROM diff_previous
    WHERE diffPrevious != 0
),
-- Pair each change point with the following change point
pair_points (mach, #open, t_start, t_end) AS
(
    SELECT  mach, #open, t_start, t_end
    FROM (SELECT mach, #open, t AS t_start,
                 last_value(t) OVER w AS t_end
          FROM changed_intervals)
    WHERE t_end IS NOT NULL
    WINDOW w AS (PARTITION BY mach
                 ORDER BY t
                 ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
),
-- Create a sequence (1, ..., max(#open))
max_seq (n) AS
(
    SELECT n
    FROM (SELECT max(#open) AS mopen FROM pair_points) x,
         generate_sequence(1,mopen) AS y(n)
),
-- Create the right number of duplicates for each tuple
SELECT mach, t_start, t_end
FROM pair_points p, max_seq s
WHERE p.#open >= s.n
```

Figure 8: Applying the SQL implementation of bag coalescing to the example table active.

active

mach	t_{start}	t_{end}
M1	1	5
M1	1	10
M1	5	7
M2	2	6
M2	3	6



change_points (inputs of the union subquery)

mach	$\#_{start}$	$\#_{end}$	t
M1	2	0	1
M1	1	0	5
M2	1	0	2
M2	1	0	3

mach	$\#_{start}$	$\#_{end}$	t
M1	0	1	5
M1	0	1	7
M1	0	1	10
M2	0	2	6

change_points

mach	$\#_{start}$	$\#_{end}$	t
M1	2	0	1
M1	1	1	5
M1	0	1	7
M1	0	1	10
M2	1	0	2
M2	1	0	3
M2	0	2	6

num_intervals

mach	t	$\#_{open}$
M1	1	2
M1	5	2
M1	7	1
M1	10	0
M2	2	1
M2	3	2
M2	6	0

diff_previous

mach	t	$\#_{open}$	diffPrevious
M1	1	2	2
M1	5	2	0
M1	7	1	-1
M1	10	0	-1
M2	2	1	1
M2	3	2	1
M2	6	0	-2

changed_intervals

mach	t	$\#_{open}$	diffPrevious
M1	1	2	2
M1	7	1	-1
M1	10	0	-1
M2	2	1	1
M2	3	2	1
M2	6	0	-2

pair_points (before WHERE)

mach	$\#_{open}$	t_{start}	t_{end}
M1	2	1	7
M1	1	7	10
M1	0	10	NULL
M2	1	2	3
M2	2	3	6
M2	0	6	NULL

pair_points

mach	$\#_{open}$	t_{start}	t_{end}
M1	2	1	7
M1	1	7	10
M2	1	2	3
M2	2	3	6

max_seq

n
1
2

result

mach	t_{start}	t_{end}
M1	1	7
M1	1	7
M1	7	10
M2	2	3
M2	3	6
M2	3	6

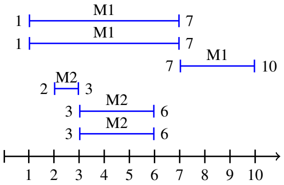


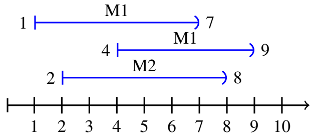
Figure 9: Example database and intermediate results of the query implementing bag coalescing for table active.

```sql
-- name left and right inputs
WITH
left AS (
    SELECT * FROM active
),
right AS (
    SELECT * FROM active
),
-- Gather change points
end_points AS
(
    SELECT mach, t_start AS t
    FROM left
    UNION
    SELECT mach, t_end AS t
    FROM left
    UNION
    SELECT mach, t_start AS t
    FROM right
    UNION
    SELECT mach, t_end AS t
    FROM right
),
-- Gather intervals of LEFTY with a unique ID
interval_id AS
(
    SELECT row_number() OVER (ORDER BY 1) AS id,
           mach,
           t_start,
           t_end
    FROM left
),
-- Join intervals with change points
split_points AS
(
    SELECT l.id,
           l.mach,
           l.t_start,
           l.t_end,
           c.t
    FROM interval_id l,
         end_points c
    WHERE c.mach = l.mach
          AND c.T >= l.t_start
          AND c.T < l.t_end
)
-- Produce output by input on change points
SELECT mach,
       t AS t_start,
       COALESCE(lead(t) OVER w, t_end) AS t_end
FROM split_points
WINDOW w AS (PARTITION BY id ORDER BY t) ;
```

Figure 10: SQL implementation of the split operator applied to example table `active`.

active

mach	t_{start}	t_{end}
M1	1	7
M1	4	9
M2	2	8



change_points

mach	t
M1	1
M1	4
M1	7
M1	9
M2	2
M2	8

interval_id

id	mach	t_{start}	t_{end}
1	M1	1	7
2	M1	4	9
3	M2	2	8

time_points

id	mach	t_{start}	t_{end}	t
1	M1	1	7	1
1	M1	1	7	4
2	M1	4	9	4
2	M1	4	9	7
3	M2	2	8	2

result

mach	t_{start}	t_{end}
M1	1	4
M1	4	7
M1	4	7
M1	7	9
M2	2	8

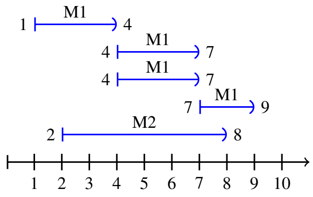


Figure 11: Example table `active` and the (intermediate) results of the SQL query implementing the split operator.

```
-- Pre-aggregate before splitting
WITH
pre_agg (mach, c, s, t_start, t_end) AS
(
  SELECT mach,
         count(*) AS c,
         sum(consum) AS s,
         t_start,
         t_end
  FROM active
  GROUP BY mach, t_start, t_end
),
-- Compute amount of increase/decrease at each time point
increase_decrease (mach, add_c, add_s, dec_c, dec_s, t) AS
(
  SELECT mach,
         sum(add_c) AS add_c,
         sum(add_s) AS add_s,
         sum(dec_c) AS dec_c,
         sum(dec_s) AS dec_s,
         t
  FROM (SELECT mach,
               c AS add_c,
               s AS add_s,
               0 AS dec_c,
               0 AS dec_s,
               t_start AS t
        FROM pre_agg
        UNION ALL
        SELECT mach,
               0 AS add_c,
               0 AS add_s,
               c AS dec_c,
               s AS dec_s,
               t_end AS t
        FROM pre_agg)
  GROUP BY mach, t
),
-- Calculate accumulative total for interval start
-- points up to and including time point t and
-- subtract the total for "closing" intervals
accumulation (mach, c, s, t) AS
(
    SELECT mach
           sum(add_c) OVER w
           - sum(dec_c) OVER w AS c,
           sum(add_s) OVER w
           - sum(dec_s) OVER w AS s,
           t
    FROM increase_decrease
    WINDOW w AS (PARTITION BY mach
                 ORDER BY t
                 RANGE UNBOUNDED PRECEDING)
),
-- output results for adjacent "split" points
SELECT mach, avg_con, t_start, t_end
FROM (SELECT mach,
             c
             CASE WHEN (c = 0) THEN NULL
                  ELSE s / c END AS avg_con,
             t AS t_start,
             last_value(t) OVER w AS t_end
      FROM accumulation
      WINDOW w AS (PARTITION BY mach
                   ORDER BY t
                   ROWS BETWEEN 1 FOLLOWING AND 1 FOLLOWING)
)
WHERE c > 0
```

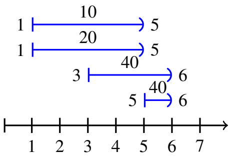
Figure 12: Example SQL implementation of split + aggregation (average) applied to example table active

```
SELECT mach, avg(consum) as avg_con
FROM active
GROUP BY mach;
```

Figure 13: Example aggregation query

active

mach	consum	t_{start}	t_{end}
M1	10	1	5
M1	20	1	5
M1	40	3	6
M1	40	5	6



pre_agg

mach	c	s	t_{start}	t_{end}
M1	2	30	1	5
M1	1	40	3	6
M1	1	40	1	6

increase_decrease (inputs of the union subquery)

mach	add_c	add_s	dec_c	dec_s	t
M1	2	30	0	0	1
M1	1	40	0	0	3
M1	1	40	0	0	5

mach	add_c	add_s	dec_c	dec_s	t
M1	0	0	2	30	5
M1	0	0	1	40	6
M1	0	0	1	40	6

increase_decrease

mach	add_c	add_s	dec_c	dec_s	t
M1	2	30	0	0	1
M1	1	40	0	0	3
M1	1	40	2	30	5
M1	0	0	2	80	6

accumulation

mach	c	s	t
M1	2	30	1
M1	3	70	3
M1	2	80	5
M1	0	0	6

result

mach	avg_con	t_{start}	t_{end}
M1	15	1	3
M1	23.3	3	5
M1	40	5	6

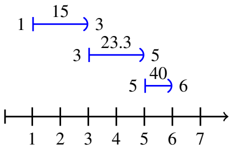


Figure 14: Intermediate results of the query implementing split + aggregation for the query from Figure 13

```sql
WITH
left (mach, consum, t_start, t_end) AS
(
  SELECT mach, consum, t_start, t_end
  FROM active
),
right (mach, consum, t_start, t_end) AS
(
  SELECT mach, consum, t_start, t_end
  FROM faulty
),
-- Count opening and closing intervals for each interval
-- end point.  Intervals from the right input are
-- counted negatively.
end_point_counts (mach, consume, #open, #close, t) AS
(
  SELECT mach, consum, t_start AS t,
         count(*) AS #open, 0 AS #close
  FROM left
  GROUP BY t_start, mach, consum
  UNION ALL
  SELECT mach, consum, t_end AS t,
         0 AS #open, count(*) AS #close
  FROM left
  GROUP BY t_end, mach, consum
  UNION ALL
  SELECT mach, consum, t_start AS t,
         - count(*) AS #open, 0 AS #close
  FROM right
  GROUP BY t_start, mach, consum
  UNION ALL
  SELECT mach, consum, t_end AS t,
         0 AS #open, - count(*) AS #close
  FROM right
  GROUP BY t_end, mach, consum
),
-- Accumulate counts to get multiplicities
acc_counts (mach, consum, t, #open, #close) AS
(
  SELECT mach,
         consum,
         sum(#open) AS #open,
         sum(#close) AS #close,
         t
  FROM end_point_counts
  GROUP BY t, mach, consum
),
-- Produce intervals with the corresponding multiplicities
intervals (mach, consume, t_start, t_end, multiplicity) AS
(
    SELECT mach,
           consum,
           t AS t_start,
           lead(t) OVER w1 AS t_end,
           sum(#open) OVER w2
           - sum(#close) OVER w2 AS multiplicity
    FROM acc_counts
    WINDOW w1 AS (PARTITION BY mach, consum
                  ORDER BY t),
           w2 AS (PARTITION BY mach, consum
                  ORDER BY t
                  RANGE UNBOUNDED PRECEDING)
),
-- Compute max multiplicity
max_seq (n) AS
(
    SELECT n
    FROM (SELECT max(numOpen) AS max_open FROM intervals) x,
         generate_sequence(1,max_open) AS y(n)
)
-- Produce duplicates based on multiplicities
SELECT mach, consum, t_start, t_end
FROM intervals i, max_seq m
WHERE multiplicity > 0 AND i.multiplicity >= m.n;
```

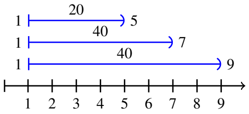
Figure 15: SQL implementation of split + bag difference applied to example table `active`.

```sql
SELECT mach, consum, t_start, t_end
FROM active
EXCEPT ALL
SELECT mach, consum, t_start, t_end
FROM faulty;
```

Figure 16: Example query using bag difference.

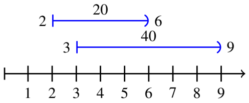
active (left input)

mach	consum	t_{start}	t_{end}
M1	20	1	5
M1	40	1	7
M1	40	1	9



faulty (right)

mach	consum	t_{start}	t_{end}
M1	20	2	6
M1	40	3	5



end_point_counts

mach	consum	t	$\#_{open}$	$\#_{close}$
M1	20	1	1	0
M1	40	1	2	0
M1	20	5	0	1
M1	40	7	0	1
M1	40	9	0	1
M1	20	2	-1	0
M1	40	3	-1	0
M1	20	6	0	-1
M1	40	5	0	-1

acc_counts

mach	consum	t	$\#_{open}$	$\#_{close}$
M1	20	1	1	0
M1	20	2	-1	0
M1	20	5	0	1
M1	20	6	0	-1
M1	40	1	2	0
M1	40	3	-1	0
M1	40	7	0	1
M1	40	9	0	0

intervals

mach	consum	t_{start}	t_{end}	multiplicity
M1	20	1	2	1
M1	20	2	5	0
M1	20	5	6	-1
M1	20	6	NULL	0
M1	40	1	3	2
M1	40	3	7	1
M1	40	7	9	0
M1	40	9	NULL	0

max_seq

n
1
2

result

mach	consum	t_{start}	t_{end}
M1	20	1	2
M1	40	1	3
M1	40	1	3
M1	40	3	7

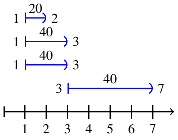


Figure 17: Example instance of table `active` and intermediate results of the query implementing split + bag difference for the query from Figure 16