Snapshot Semantics for Temporal Multiset Relations

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

Snapshot 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 snapshot semantics over interval-timestamped multiset relations are subject to two bugs regarding snapshot aggregation and bag difference. We introduce a novel temporal data model based on K-relations that overcomes these bugs and prove it to correctly encode snapshot 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.

PVLDB Reference Format:
Anton Dignös, Boris Glavic, Xing Niu, Michael Böhlen, and Johann Gamper. Snapshot Semantics for Temporal Multiset Relations. *PVLDB*, 12(6): 639-652, 2019.
DOI: https://doi.org/10.14778/3311880.3311882

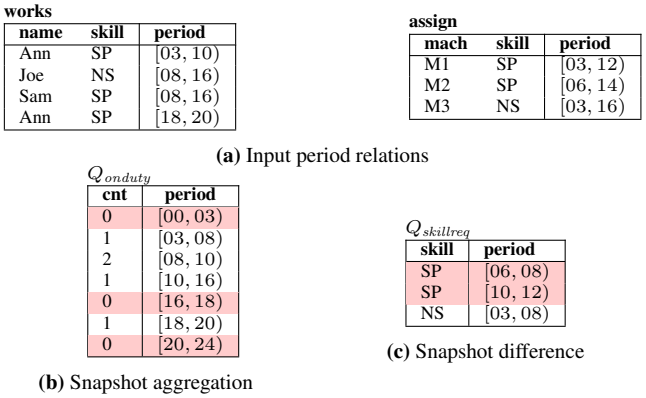
1. INTRODUCTION

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 [28] 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 [35], Teradata [45], Oracle [31], IBM DB2 [36], and MS SQLServer [30]. However, none of these systems, with the partial exception of Teradata, supports *snapshot semantics*, an important class of temporal queries. Given a temporal database, a non-temporal query Q interpreted under snapshot 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 snapshot semantics is known as *snapshot-reducibility* [29, 43].

Example 1.1 (Snapshot 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 a 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 (specialized) SP worker is in the factory at any given time. This is checked by evaluating the following query under snapshot semantics.*

Q_{onduty}: `SELECT count(*) AS cnt FROM works WHERE skill = 'SP'`

Evaluated under snapshot semantics, a query returns a snapshot (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) Snapshot aggregation

$Q_{skillreq}$

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

(c) Snapshot difference

Figure 1: Snapshot 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 snapshot semantics that we are aware of do not return results for gaps (tuples marked in red) and, therefore, violate snapshot-reducibility. Teradata [45, 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 snapshot bag difference (**`EXCEPT ALL`**).

Proceedings of the VLDB Endowment, Vol. 12, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3311880.3311882

Example 1.2 (Snapshot Bag Difference)**.** *Consider again Figure 1. Relation* `assign` *records machines (mach) that need to be 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 snapshot 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 snapshot 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 snapshot semantics queries over bag relations, users currently resort to manually implementing such queries in SQL which is time-consuming and error-prone [40].

We address the above limitations of previous approaches for snapshot 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. Previous works on *sequenced semantics* [7, 16, 18] also aim to support snapshot-reducibility. However, these approaches focus on *change preservation* (i.e., preserve intervals from the input of a query), whereas we instead focus on a unique encoding. 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 *snapshot K-relations* as a generalization of snapshot 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 snapshot semantics over snapshot 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 snapshot semantics over period K-relations into SQL.
- We implement our approach as a database middleware and present optimizations that eliminate redundant coalescing steps. We demonstrate experimentally 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, 23]. Many temporal query languages including TSQL2 [39, 41], ATSQL2 (Applied TSQL2) [8], IXSQL [29], ATSQL [9], and SQL/TP [47] support snapshot semantics. In this paper, we provide a general framework that can be used to correctly implement snapshot semantics over period set and multiset relations for any language.

Interval-based Approaches for Snapshot Semantics. In the following, we discuss interval-based approaches for snapshot 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 snapshot 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 snapshot 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 [37] (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* [45] is a commercial DBMS that supports snapshot 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

Table 1: Interval-based approaches for snapshot semantics.

Approach	Multisets	AG bug free	BD bug free	Unique encoding
Interval preservation [9] (ATSQL)	✓	×	×	×
Teradata [45]	✓	×	N/A	×[1]
Change preservation [16, 18]	×	×	N/A	×
TSQL2 [39, 41, 43]	×	N/A	N/A	✓
ATSQL2 [8]	✓	N/A	×	×
TimeDB [44] (ATSQL2)	✓	N/A	×	×
SQL/Temporal [42]	✓	×	×	×
SQL/TP [47][2]	✓	✓	✓	×
Our approach	✓	✓	✓	✓

change in the input, i.e., sequenced semantics. 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 solves the AG bug. Furthermore, we provide a unique encoding and support bag semantics in addition to set semantics. *TSQL2* [39, 41, 43] implicitly applies coalescing [10] to produce a unique representation. Thus, it only supports set semantics, and it does not support aggregation. Snodgrass et al. [42] present a validtime extension of *SQL/Temporal* and an algebra with snapshot 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* [44] is an implementation of ATSQL2 [8]. It uses a semantics for bag difference and intersection that is not snapshot-reducible (see [44, 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-snapshot Temporal Approaches. Non-snapshot temporal query languages, such as IXSQL [29] and SQL/TP [47], do not explicitly support snapshot semantics. Nevertheless, we review these languages here since they allow to express queries with snapshot semantics. SQL/TP [47] introduces a point-wise semantics for temporal queries [12, 46], 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 snapshot semantics. However, snapshot 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, 33] and aggregation [5, 32, 34]. Some exceptions supporting multiple operators are [13, 18, 26]. 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 snapshot 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. [48] 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] which removes redundant coalescing steps, we remove unnecessary K-coalescing steps and, similar to [48], we use OLAP functions for an efficient implementation.

Temporality in Annotated Databases. Kostiley et al. [27] 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 – Snapshot K-relations. As an *abstract model* we use snapshot 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 *snapshot K-relations* [21], which are snapshot 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 snapshot query Q over a snapshot 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 snapshot $\mathbb{N}$-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

[1]Optionally, coalescing (`NORMALIZE ON` in Teradata) can be applied to get a unique encoding at the cost of loosing multiplicities.
[2]Snapshot semantics can be expressed, but this is inefficient.

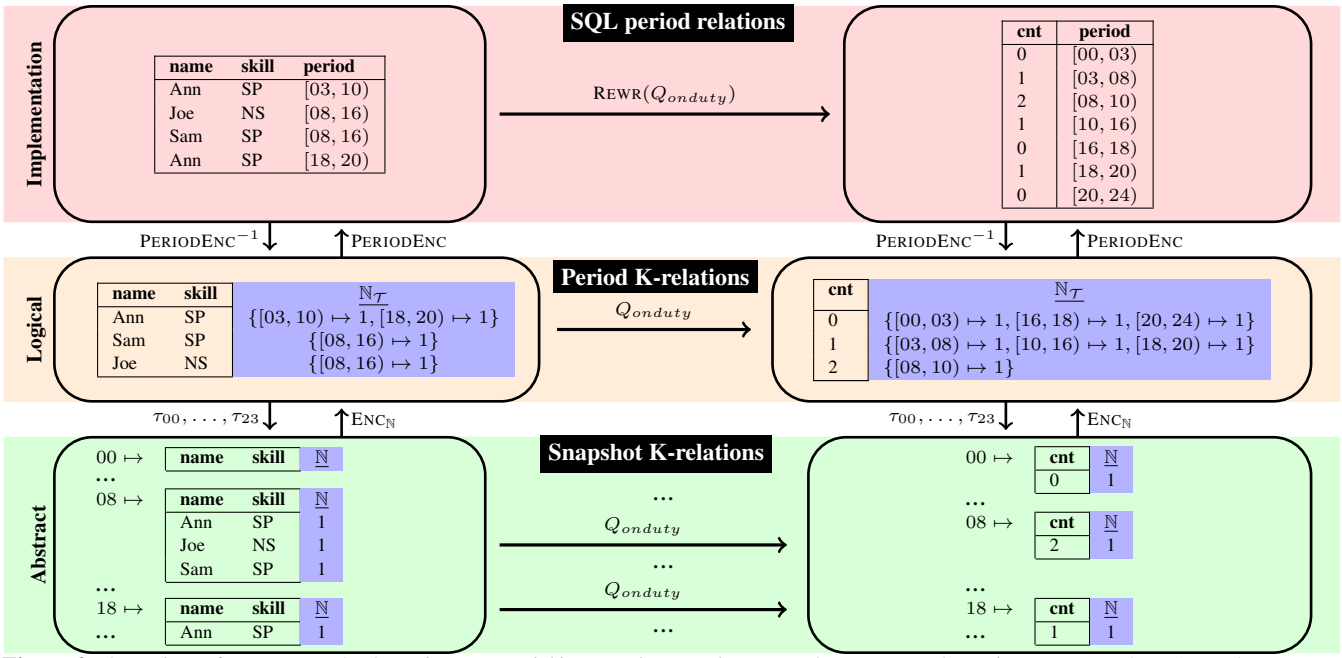


Figure 2: Overview of our approach. Our *abstract model* is *snapshot K-relations* and nontemporal queries over snapshots (snapshot semantics). Our *logical* model is *period K-relations* and queries corresponding to the abstract model's snapshot queries. Our *implementation* uses *SQL period relations*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

an interval-based model to correctly encode snapshot 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 snapshot K-relation R is (i) unique and (ii) snapshot-equivalent to R. Furthermore, (iii) queries over encodings are snapshot-reducible.

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* ENC$_\mathbb{N}$ *from snapshot $\mathbb{N}$ to $\mathbb{N}_\mathcal{T}$-relations and the time slice operator τ_T which restores 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 snapshot 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 (PERIODENC) together with a rewriting scheme for queries (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 snapshot semantics as a new language feature in SQL and rewrites snapshot queries into SQL queries over SQL period relations. That is, we directly evaluate snapshot queries over data stored natively as period relations.

4. SNAPSHOT K-RELATIONS

We first review background on the semiring annotation framework (K-relations). Afterwards, we define *snapshot K-relations* as our abstract model and snapshot semantics for this model. Importantly, queries over snapshot 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 [21], 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 [37] ($\mathcal{RA}^+$) 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 [21]. 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:*

$$\sigma_\theta(R)(t) = R(t) \cdot \theta(t) \qquad (\textit{selection})$$
$$\Pi_A(R)(t) = \sum_{u:u.A=t} R(u) \qquad (\textit{projection})$$
$$(R \bowtie S)(t) = R(t[R]) \cdot S(t[S]) \qquad (\textit{join})$$
$$(R \cup S)(t) = R(t) + S(t) \qquad (\textit{union})$$

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 [21].

Definition 4.2 (Homomorphism)**.** *A mapping $h : K_1 \to K_2$ 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 Snapshot K-relations

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

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

For instance, a snapshot $\mathbb{N}$-relation is shown in Figure 2 (bottom). Given a snapshot K-relation, we use the *timeslice operator* [24] to access its state (snapshot) at a time point T:

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

The evaluation of a query Q over a snapshot database (set of snapshot relations) D under *snapshot semantics* returns a snapshot 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, snapshot temporal queries over snapshot 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 (Snapshot Semantics)**.** *Let D be a snapshot K-database and Q be a query. The result $Q(D)$ of Q over D is a snapshot 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 snapshot $\mathbb{N}$-relation shown at the bottom of Figure 2 and the evaluation of Q_{onduty} under snapshot 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 snapshot K-relations, we study representation systems that consist of a set of representations $\mathcal{E}$, a function $\text{ENC} : \mathcal{E} \to \mathcal{DB}_{\mathbb{T},K}$ which associates an encoding in $\mathcal{E}$ with the snapshot K-database it represents, and a timeslice operator τ_T which extracts the snapshot at time T from an encoding. If ENC is injective, then we use $\text{ENC}^{-1}(D)$ to denote the unique encoding associated with D. We use τ to denote the timeslice over both snapshot 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* [22] (written as $D_1 \sim D_2$) if they encode the same snapshot 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 ENC preserves the snapshots of the input.

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

1. $\text{ENC}(E) = \text{ENC}(E') \Rightarrow E = E'$ *(uniqness)*
2. $\tau_T(Q(E)) = Q(\tau_T(E))$ *(snapshot-reducibility)*
3. $\text{ENC}(E) = D \Rightarrow \tau_T(E) = \tau_T(D)$ *(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 snapshot 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 $I \cap I' = \emptyset \wedge \neg adj(I, I')$. 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 $\mathbb{T} = \{00, \ldots, 23\}$. 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 we omit intervals that 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, 38]. 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.

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.

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.

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:

$$CPI(\mathcal{T}) = \{[T_b, T_e) \mid T_b <_\mathbb{T} T_e \wedge T_b \in CP(\mathcal{T}) \wedge (T_e \in CP(\mathcal{T}) \vee T_e = T_{max}) \wedge \nexists T' \in CP(\mathcal{T}) : T_b <_\mathbb{T} T' <_\mathbb{T} T_e\}$$

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:*

$$\mathcal{C}_K(\mathcal{C}_K(\mathcal{T})) = \mathcal{C}_K(\mathcal{T}) \quad \text{(idempotence)}$$
$$\mathcal{T}_1 \sim \mathcal{T}_2 \Leftrightarrow \mathcal{C}_K(\mathcal{T}_1) = \mathcal{C}_K(\mathcal{T}_2) \quad \text{(uniqueness)}$$
$$\mathcal{T} \sim \mathcal{C}_K(\mathcal{T}) \quad \text{(equivalence preservation)}$$

Proof Sketch. We provide the proofs for all lemmas and theorems in a technical report [19], but show sketches for important proofs here. Equivalence preservation and uniqueness follow from the fact that annotation change points are defined based on snapshots. Uniqueness plus equivalence preservation implies idempotence. □

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 snapshot 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 snapshot 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.*

Proof Sketch. We prove that $K_\mathcal{T}$ obeys the laws of a commutative semiring based on K being a semiring and Lemma 6.1. □

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 $K_\mathcal{T} \to K$. Since semiring homomorphisms commute with queries [21], $K_\mathcal{T}$ 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.*

Proof Sketch. Proven by substitution of definitions and by regrouping terms using semiring laws. □

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 Snapshot K-relations

We now define a bijective mapping ENC_K from snapshot 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 ${\text{ENC}_K}^{-1}$ (the inverse of ENC_K) form a representation system for snapshot K-relations. Intuitively, $\text{ENC}_K(R)$ 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 snapshot K-relation, ENC_K is a mapping from snapshot K-relations to $K_\mathcal{T}$-relations defined as follows.*

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

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

Lemma 6.4. *For any semiring K, ENC_K is bijective.*

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

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

Based on these properties of ENC_K and the fact that the timeslice operator over $K_\mathcal{T}$-relations is a homomorphism $K_\mathcal{T} \to K$, our main technical result follows immediately. That is, the set of $K_\mathcal{T}$-relations equipped with the timeslice operator and ${\text{ENC}_K}^{-1}$ is a representation system for positive relational algebra queries ($\mathcal{RA}^+$) over snapshot 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 $(\mathcal{DB}_{K_\mathcal{T}}, {\text{ENC}_K}^{-1}, \tau)$ is a representation system for $\mathcal{RA}^+$ queries over snapshot K-relations.*

Proof Sketch. We have to show that conditions (1), (2), and (3) of Definition 4.5 hold. 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 homomorphisms commute with $\mathcal{RA}^+$-queries [21, Proposition 3.5]. □

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, 20]. For instance, the difference operator on $\mathbb{N}$ relations corresponds to bag difference (SQL's **EXCEPT ALL**). Geerts et al. [20] 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 operation 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.*

Proof Sketch. We first prove that K being naturally ordered implies that $K_\mathcal{T}$ is naturally ordered where $k \preceq_{K_\mathcal{T}} k'$ is $\tau_T(k) \preceq_K \tau_T(k')$ for all time points T. Then we show constructively that for any k and k', the set $\{k'' \mid k \preceq_{K_\mathcal{T}} k' + k''\}$ has a smallest element wrt. $\preceq_{K_\mathcal{T}}$. These two condition are the only requirements for a semiring to have a well-defined monus. □

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: $\tau_T(k) -_K \tau_T(k')$. 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.*

Proof Sketch. Proven by substitution of definitions. □

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

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

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

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 snapshot $\mathbb{N}$-relations and $\mathcal{RA}^{agg}$ queries using aggregation according to Definition 7.1.*

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

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 snapshot 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.

$$ \begin{array}{ccc} R & \xrightarrow{\text{PERIODENC}} & R' \\ {\scriptstyle Q}\downarrow & & \downarrow{\scriptstyle Q' = \text{REWR}(Q)} \\ Q(R) & \xleftarrow{\text{PERIODENC}^{-1}} & Q'(R') \end{array} \quad (1) $$

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 $SCH(R) = \{A_1, \ldots, A_n\}$. The schema of* PERIODENC(R) *is $\{A_1, \ldots, A_n, A_{begin}, A_{end}\}$. Let R' be* PERIODENC(R) *for some $\mathbb{N}_\mathcal{T}$-relation.* PERIODENC *and its inverse are defined as follows:*

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

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

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

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:*

$$ \mathcal{C}(R) = \text{PERIODENC}(R') $$

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

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 [12,47]. 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*

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

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

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

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

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

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

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

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:

$$\mathcal{N}_G(R_1, R_2)(t) = split(t, R_1, EP_G(R_1 \cup R_2, t))$$

$$EP_G(R,t) = \bigcup_{t' \in R : t'.G = t.G \wedge R(t') > 0} \{t'.A_{begin}\} \cup \{t'.A_{end}\}$$

$$split(t, R, EP) = \sum_{t' : I(t) \subseteq I(t') \wedge I(t) \in EPI(t, EP)} R(t')$$

$$EPI(t, EP) = \{[T_b, T_e) \mid T_b <_\mathbb{T} T_e \wedge T_b \in EP \wedge (T_e \in EP \vee T_e = T_{max}) \wedge \nexists T' \in EP : T_b <_\mathbb{T} T' <_\mathbb{T} T_e\}$$

The use of the PERIODENC and PERIODENC^{-1} mappings in the definitions of the coalesce and split algebra operators is only 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.

Note that the rule for $count(*)$ is a necessary preprocessing step that takes precedence over the rule for general aggregation.

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

$$\text{REWR}(Q_{onduty}) = \mathcal{C}({}_{A_{begin},A_{end}}\gamma_{count(A)}(\mathcal{N}_\emptyset(\Pi_{1 \to A, A_{begin}, A_{end}}(\text{REWR}(Q_1)) \cup \{(null, 0, 24)\}, \text{REWR}(Q_1))))$$

$$\text{REWR}(Q_1) = \mathcal{C}(\sigma_{skill=SP}(works))$$

Subquery $\text{REWR}(Q_1)$ filters out the second tuple from the input (see Figure 2). The split operator is then applied to the union of the result of $\text{REWR}(Q_1)$ 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 $\text{REWR}(Q_{onduty})$. 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.*

Proof Sketch. Proven by induction over query structure. □

9. IMPLEMENTATION

We have implemented the encoding and rewriting introduced in the previous section in a middleware which supports snapshot multiset semantics through an extension of SQL. To instruct the system to interpret a subquery using snapshot semantics, the user encloses the subquery in a `SEQ VT (...)` block. We assume that the inputs to a snapshot 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 have proven that this Lemma also holds for monus [19]. 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 $+_{K_\mathcal{P}}$, $\cdot_{K_\mathcal{P}}$, and $-_{K_\mathcal{P}}$. 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 further discussion of this optimization and an example see [19, Appendix C].

We developed an optimized implementation of multiset coalescing using SQL analytical window functions, similar to set-based coalescing in [48], 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 declarations (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 snapshot queries with our approach over three DBMSs and compare it against native implementations of snapshot 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 two 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 [25]. 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. In our technical report we also use a real-world *Tourism* dataset (835k records).

Workloads. To evaluate the efficiency of snapshot queries, we created a workload consisting of the following 10 queries expressed over the employee dataset. `join-1`: salary and department for each employee using a join between the department and salary tables. `join-2`: salary and title for each employee using a join between the salary and title tables. `join-3`: department of employees that manage a department and earn more than $70,000 using a join between the manager and salary tables and a selection on attribute salary.`join-4`: gather all information for each manager using joins between the tables managers, salary, and employees. `agg-1`: average salary of employees per department using the result of query `join-1` with a subsequent aggregation per department. `agg-2`: average salary of managers using a join between the manager and salary tables with a subsequent aggregation without grouping. `agg-3`: number of departments with more than 21 employees. This query has no join but two aggregations, one to compute the number of employees per department and a second one to count the number of relevant departments. `agg-join`: names of employees with the highest salary in their department. This query consists of a 4-way join where one of the inputs is the result of a subquery with aggregation. `diff-1`: employees that are not managers using a difference operation between two tables. `diff-2`: salaries of employees that are not managers by computing the difference between a table and a subquery with a join. 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 snapshot queries we are aware of) and evaluated these queries under snapshot semantics. Note that some of these queries use the **`ORDER BY`** clause that we do not support for snapshot queries. However, we can evaluate such a query without **`ORDER BY`** under snapshot semantics and then sort the result without affecting what rows are returned. The number of rows returned by the Employee and TPC-H queries are shown in Table 2. The SQL code and more detailed descriptions of these queries are provided in [19, Appendix B].

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 snapshot semantics (only available as a virtual machine); and a commercial DBMS, *DBY*, without native support for snapshot semantics. We used our approach to translate snapshot 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 snapshot 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 snapshot multiset semantics for difference and aggregation, and many systems do not support snapshot semantics for these operators at all. *DBX-Nat* and *PG-Nat* both support snapshot aggregation, however, their implementations are not snapshot-reducible. *DBX-Nat* does not support snapshot 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.

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).

Table 2: Number of query result rows

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

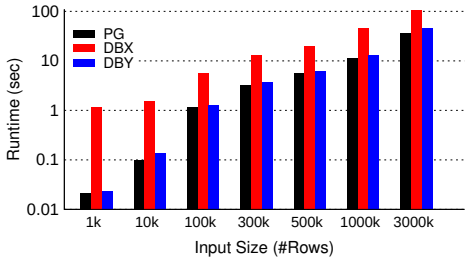


Figure 5: Multiset coalescing for varying input size.

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 snapshot 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.

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

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

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

10.3 Snapshot Semantics - Employee

Table 3 provides an overview of the performance results for our snapshot 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 snapshot semantics on all systems by several 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 snapshot 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 snapshot 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 Snapshot Semantics - TPC-BiH

The runtimes for TPC-H queries interpreted under snapshot 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 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 snapshot 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 snapshot 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 snapshot 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 snapshot 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, 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 bi-temporal 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^{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] A. Dignös, B. Glavic, X. Niu, M. H. Böhlen, and J. Gamper. Snapshot semantics for temporal multiset relations (extended version). Technical Report CoRR, `https://arxiv.org/pdf/1902.04938`, 2019.

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

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

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

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

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

[25] 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.

[26] 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.

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

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

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

[30] Microsoft. SQL Server 2016 - temporal tables. `https://docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables`, 2016.

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

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

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

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

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

[36] 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.

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

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

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

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

[41] 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.

[42] 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.

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

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

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

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

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

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

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