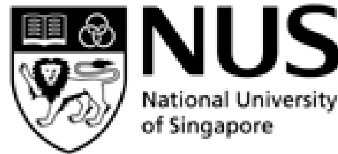
THE NATIONAL UNIVERSITY
of SINGAPORE



School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TRA7/21

Temporal Keyword Search with Aggregates and Group-By

Qiao Gao, Mong Li Lee and Tok Wang Ling

July 2021

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

Mohan KANKANHALLI
Dean of School

Temporal Keyword Search with Aggregates and Group-By

Qiao Gao (✉) Mong Li Lee Tok Wang Ling
National University of Singapore
{*gaoqiao, leeml, lingtw*}*@comp.nus.edu.sg*

Abstract. Temporal keyword search enables non-expert users to query temporal relational databases with time conditions. However, aggregates and group-by are currently not supported in temporal keyword search, which hinders querying of statistical information in temporal databases. This work proposes a framework to support aggregate, group-by and time condition in temporal keyword search. We observe that simply combining non-temporal keyword search with aggregates, group-by, and temporal aggregate operators may lead to incorrect and meaningless results as a result of data duplication over time periods. As such, our framework utilizes Object-Relationship-Attribute semantics to identify a unique attribute set in the join sequence relation and remove data duplicates from this attribute set to ensure the correctness of aggregate and group-by computation. We also consider the time period in which temporal attributes occur when computing aggregate to return meaningful results. Experiment results demonstrate the importance of these steps to retrieve correct results for keyword queries over temporal databases.

Keywords: Temporal Keyword Search, Aggregates and Group-By, Semantic Approach

1 Introduction

Keyword query over relational databases has become a popular query paradigm by freeing users from writing complicated SQL queries when retrieving data [19]. Recent works in this area are focusing on efficiency of query execution [20], quality of query results [11], and expressiveness of keyword query [9,23].

Temporal keyword search enriches the query expressiveness by supporting time conditions in keyword query and makes data retrieval on temporal databases easier. The corresponding SQL query will be automatically generated by the keyword search engine where temporal joins as well as time conditions are associated with the correct relations [9]. However, the aggregate and group-by haven't been supported in current temporal keyword search, which hinders users from querying the statistical information over time. We observe that simply combining non-temporal keyword search with aggregates, group-by, and temporal aggregate operators may lead to incorrect aggregate results. The reason is that when multiple relations are involved in a keyword query, there might be some data duplication involved in the intermediate relations of temporal joins and leading to incorrect results of accumulative aggregates, such as SUM and COUNT.

Employee

Eid	Ename	DOB	Employee _Start	Employee _End
E01	Alice	1985-03-12	2017-01-01	now
E02	Bob	1978-05-20	2017-05-01	now
E03	John	1990-10-15	2018-01-01	2018-10-31

Project

Pid	Pname	Budget	Did	Project _Start	Project _End
P01	Healthcare with AI	300k	D01	2017-07-01	2018-12-31
P02	KWS	250k	D01	2017-07-01	2018-06-30
P03	Smart City with AI	400k	D02	2018-01-01	now

Department

Did	Dname
D01	CS
D02	IS

EmployeeSalary

Eid	Salary(per month)	Salary_Start	Salary_End
E01	3k	2017-06-01	2018-03-31
E01	3.5k	2018-04-01	2018-09-30
E01	4k	2018-10-01	now
E02	4k	2017-05-01	2018-12-31
E02	4.5k	2019-01-01	now
E03	3k	2018-01-01	2018-10-31

WorkFor

Eid	Did	WorkFor_Start	WorkFor_End
E01	D01	2017-01-01	2017-08-31
E01	D02	2017-09-01	now
E02	D01	2017-05-01	2018-02-28
E02	D02	2018-03-01	2018-04-30
E02	D01	2018-05-01	now
E03	D01	2018-01-01	2018-10-31

ParticipateIn

Eid	Pid	ParticipateIn _Start	ParticipateIn _End
E01	P01	2017-07-01	2018-12-31
E02	P01	2018-01-01	2018-12-31
E02	P02	2017-07-01	2018-06-30
E03	P02	2018-01-01	2018-06-30
E01	P03	2018-01-01	2018-06-30
E02	P03	2018-10-01	now
E03	P03	2018-07-01	2018-09-30

Fig. 1. Example company temporal database.

Example 1 (Incorrect aggregate results). Consider the temporal database in Fig. 1. Suppose we issue a query Q_1 ={Employee SUM Salary Project AI DURING [2018]} to compute the total salary of employees participating in AI projects in 2018. Multiple projects match the keyword "AI" ($P01$ and $P03$), and two employees $E01$ and $E02$ participated in these projects in 2018. As such, the salary of $E01$ and $E02$ will be duplicated in the join sequence relation, and simply applying existing temporal aggregate operators, e.g., those proposed in [3,6,7,13], on the intermediate relation of joins will give incorrect total salaries for the employees.

The work in [22] first highlights the problem of incorrect query results in non-temporal keyword search with aggregates and group-by, and uses Object-Relationship-Attribute (ORA) semantics [21] to remove the duplicate data when relations capturing n-ary (n>2) relationships are joined. However, this work cannot handle all the data duplication occurred in temporal keyword search, since they are more prevalent due to the repeating attribute values and relationships over time, and the join between temporal and non-temporal relations.

In this work, we propose a framework to process temporal keyword queries involving aggregate functions, group-by and time conditions. Our framework utilizes ORA semantics to identify object/relationship type and attributes, and remove data duplication in the intermediate relation. Further, frequent data updates over time may lead to fine-grained temporal aggregate results that are not meaningful to users. Hence, we support aggregates over user-specified time units such as year or month in the keyword query. Finally, a temporal attribute may have an inherent time unit, e.g., monthly salary or daily room rate. Our framework provides an option to compute meaningful accumulative sum over such attributes by weighting its value with its duration. Experiment results indicate the importance of these steps to return correct and meaningful results for keyword queries over temporal databases.

2 Preliminaries

Temporal Keyword Query. We extend the temporal keyword query defined in [9] to allow aggregate functions and group-by as follows:

$< Q > ::= < basic_query > [< groupby_cond >][< time_cond >]$

where $< basic_query >$ is a set of keywords $\{k_1\ k_2 ... k_i\}$, and each keyword can match a tuple value, a relation or attribute name, or an aggregate MIN, MAX, AVG, SUM or COUNT; $< groupby_cond >$ is a set of keywords $\{k_{i+1}\ k_{i+2} ... k_j\}$ such that k_{i+1} is the reserved word GROUPBY, and the remaining keywords match either a relation or attribute name; $< time_cond >$ contains two keywords $\{k_{j+1}\ k_{j+2}\}$ such that k_{j+1} is a temporal predicate like AFTER or DURING [2], and k_{j+2} is a closed time period $[s, e]$*.

Temporal ORM Schema Graph. [21] proposes an Object-Relationship-Mixed (ORM) schema graph to capture the ORA semantics in a relational database. Each node in the graph is an object/relationship/mixed node comprising an object/relationship/mixed relation and its component relations. An object (or relationship) relation stores all the single-valued attributes of an object (or relationship) type. A mixed relation stores an object type and its many-to-one relationship type(s). A component relation stores a multivalued attribute of an object/relationship type. Two nodes u and v are connected by an undirected edge if there is a foreign key-key constraint from the relations in u to those in v.

In temporal databases, an object type or relationship type becomes temporal when it is associated with a time period indicating its lifespan [8]. An attribute of some object/relationship type becomes temporal when its value changes over time and the database keeps track of the changes. We extend the ORM schema graph to a temporal ORA schema graph for temporal databases.

A node with superscript T in a temporal ORA schema graph denotes it contains some temporal relations. A temporal relation R^T is essentially a relation R with a closed time period $R^T.period$ consisting of a start attribute and an end attribute. The temporal ORA semantics of a temporal relation could be identified via the relation type. A temporal object/relationship relation stores a temporal object/relationship type, while a temporal mixed relation stores a temporal object type with its non-temporal many-to-one relationship type(s), and a temporal component relation stores a temporal attribute.

Fig. 2 shows the temporal ORM schema graph for the database in Fig. 1. The object node Employee contains the temporal object relation *Employee* and its temporal component relation *EmployeeSalary*, depicting the temporal object Employee and the temporal attribute Salary respectively.

Annotated Query Pattern. We adopt the approach in [22] to generate a set of annotated query patterns to depict the different interpretations of a keyword query. Keywords are matched to relations in a temporal ORM schema graph. Aggregates and GROUPBY are reserved keywords and are used to annotate the query pattern according to the matches of the keywords that follow them. Note that the time condition in keyword query is not considered in this process.

Fig. 3 shows one of query patterns generated from the query without time condition {Employee SUM Salary Project AI} of Q_1 in Example 1. The keywords

*We use the format YYYY-MM-DD for start and end times, and allow shortened forms with just YYYY or YYYY-MM. We will convert the short form, by setting the start/end time to the earliest/latest date of given year or month, e.g., [2017,2018] to [2017-01-01, 2018-12-31].

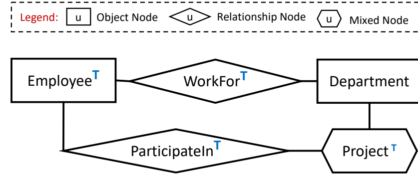


Fig. 2. Temporal ORM schema graph of the temporal database in Fig. 1.

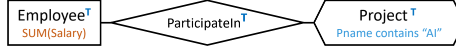


Fig. 3. Annotated query pattern generated for query Q_1.

Employee and Salary match the node *Employee* in the temporal ORM graph in Fig. 2, while keywords Project and AI match the node *Project*. Since there is a cycle in the temporal ORM graph, we have two ways to connect these two nodes: via the node *ParticipateIn* or the nodes $WorkFor - Department$, giving us two possible query interpretations. Since the keyword Salary follows the aggregate SUM, the attribute *Salary* in the node *Employee* is annotated with SUM, depicting the user's search intention to find the sum of the salaries of employees who participated in AI projects.

3 Proposed Framework

Our framework to process a temporal keyword query consists of 3 key steps:

Step 1. Generate a temporal join sequence from an annotated query pattern, and use it to compute a join sequence relation.
Step 2. Identify a subset of attributes (called *unique attribute set*) from join sequence relation and remove data duplicates on this attribute set.
Step 3. Compute temporal aggregate and group-by over the relation obtained from step 2.

3.1 Generate Temporal Join Sequence

We generate one temporal join sequence from each annotated query pattern to compute a join sequence relation. The sequence contains relations in the query pattern augmented with selections (σ) and projections (Π), if any. The selection is used to apply the time condition in query to each relevant temporal relation, while the projection is used to drop attributes irrelevant to this query. These relations are joined with natural join ($\bowtie$) or temporal join ($\bowtie^T$) [10] according to the edge connections in the query pattern. Temporal join operator is used between temporal relations to avoid incorrect join results [9].

Algorithm 1 gives the details. The select operator applies attribute value conditions on the relations in a query pattern, while the project operator filters out attributes that are not relevant to the query (Lines 1-9). The time condition in the query is applied to each temporal relation in the query pattern. Then the project operator is applied to each relation to preserve object/relationship attribute identifiers, target attributes of aggregates and group-by, and attributes

Algorithm 1: Generate Temporal Join Sequence

```
   Input: annotated pattern P, time condition TP [s,e];
   Output: a temporal join sequence J;
1  foreach node u ∈ P do
2      foreach relation R ∈ u do
3          Use select operator to apply the attribute conditions on R;
4          Let I be the set of identifier attributes for object/relationship type in R, and X,
             Y be the sets of attributes applied by aggregate function and group-by
             respectively;
5          if R is a temporal relation then
6              R = σ_{period TP [s,e]}(R);
7              R = Π_{I∪X∪Y∪R.period}(R);
8          else
9              R = Π_{I∪X∪Y}(R);
10     Let R_u be the object/relationship/mixed relation in u.
11     J_u = R_u; /* Generate join sequence J_u with relations in node u        */
12     foreach component relation R_c ∈ u do
13         if R_c has some attribute value conditions or is annotated by GROUPBY then
14             if R_c is temporal and J_u contains temporal relation then
15                 J_u = J_u ⋈^T R_c;
16             else
17                 J_u = J_u ⋈ R_c;
18 Let J = J_v for some node v ∈ P. /* Join relations from nodes in P to generate J   */
19 Add the nodes adjacent to v to a queue Q;
20 while Q.notEmpty() do
21     u = Q.poll();
22     if both J and J_u contain temporal relations then
23         J = J ⋈^T J_u;
24     else
25         J = J ⋈ J_u;
26     foreach node w adjacent to u ∈ P and J_w ∉ J do
27         Queue.add(w);
28 return J;
```

forming the time periods. We generate the join sequence J_u for relations in each node u (Lines 10-17), and join the relations based on the edges in the query pattern (Lines 18-27). For temporal join, two tuples are joined if their key values are equal and their time periods intersect. The time period in the resulting tuple is given by the intersection of the time periods in the two tuples, and the original time periods are dropped. Note that a component relation is included in the join sequence if and only if it is annotated with group-by or attribute value conditions.

For example, the temporal join sequence generated for the query pattern in Fig. 3 is $J = R_1 \bowtie^T R_2 \bowtie^T R_3 \bowtie^T R_4$ where

$R_1 = \Pi_{Eid \cup Salary \cup period}(\sigma_\varphi(Salary^T))$,
$R_2 = \Pi_{Eid \cup period}(\sigma_\varphi(Employee^T))$,
$R_3 = \Pi_{Eid \cup Pid \cup period}(\sigma_\varphi(ParticipateIn^T))$,
$R_4 = \Pi_{Pid \cup period}(\sigma_{Pname\ contains\ "AI" \cup \varphi}(Project^T))$.

The select condition "$\varphi = R^T.period\ TP\ [s,e]$" denotes to apply time condition "$TP\ [s,e]$" to time period $R^T.period$, where TP denotes a temporal predicate.

We have 3 types of attributes in the join sequence relation R_J obtained from the temporal join sequence: (1) all the key attributes of the relations in the query pattern, (2) attributes involved in the aggregate function and the group-by, and (3) time period attributes (*start* and *end*), if any. Note that attributes that

are involved only in the select conditions are not included in the join sequence relation, since they are not used subsequently.

3.2 Remove Duplicate Data

Generating the join sequence relation may introduce data duplicates on the attributes related to aggregates and group-by, which may lead to incorrect aggregate results. This is so because relations in the temporal join sequence have many-to-one or many-to-many relationships, multivalued and time period attributes. Note that a join sequence relation R_J has no duplicate tuples, but it may have data duplicates with respect to a *unique attribute set* $\mathcal{U} \subset R_J$ depicting the ORA semantics the aggregate function and group-by applied to.

We determine the attributes in $\mathcal{U}$ as follows. Let X and Y be the sets of attributes that the aggregate function and group-by are applied to respectively. Note that $Y = \emptyset$ if the keyword query does not have GROUPBY reserved word.

Case 1. X only contains the identifier attribute(s) of some object/relationship. Unique objects or relationships are identified via the values of their identifiers. From the temporal ORM schema graph, we know whether X is the identifier of a non-temporal or temporal object/relationship type. For non-temporal object/relationship type, $\mathcal{U} = (X, Y)$. Otherwise, the time period attributes *start* and *end* in R_J are included to remove duplicates, i.e., $\mathcal{U} = (X, Y, R_J.start, R_J.end)$.

Case 2. X only contains a non-prime attribute A.
Let *oid* be the identifier attribute(s) of the object/relationship type for A. If A is non-temporal, $\mathcal{U} = (oid, A, Y)$. Otherwise, A is temporal and $\mathcal{U} = (oid, A, Y, R_J.start, R_J.end)$. The reason why *oid* is added to $\mathcal{U}$ is that we need to distinguish the same attribute value of different objects/relationships.

If $\mathcal{U}$ does not contain time period attributes *start* and *end*, then two tuples in R_J are duplicates iff their $\mathcal{U}$ values are the same. Otherwise, we say that two tuples are duplicates over time iff their $\mathcal{U} - \{R_J.start, R_J.end\}$ values are the same and time periods of those two tuples intersect. Note that the duplication occurs in the intersected time period. We use the functional dependency (FD) theory to determine if there are data duplicates over $\mathcal{U}$. Let K be the key of R_J. If $K \rightarrow \mathcal{U}$ is a full FD, and not a transitive FD, then there is no data duplicate on $\mathcal{U}$. Note that when both K and $\mathcal{U}$ contain time period attributes, the FD should hold for the same time points in the corresponding time periods.

Algorithm 2 gives the details to remove data duplicates from R_J over $\mathcal{U}$. Line 2 uses projection to remove data duplicates if $\mathcal{U}$ does not contain time period attributes. Otherwise, let $\mathcal{U}' = \mathcal{U} - \{start, end\}$ be the non-period attributes set in $\mathcal{U}$. We first sort the tuples based on $\mathcal{U}' \cup \{start\}$ (lines 4-7). Then, we perform a linear scan to remove data duplication over time by merging the overlapped time periods for tuples with same values on $\mathcal{U}'$ (Lines 8-17). The time complexity of this step is $O(n * log(n))$[*], where n is the number of tuples in R_J.

[*]Projection over R_J requires $O(n)$ time, sorting the tuples requires $O(n * log(n))$ time, and linear scan to remove duplicates over time takes $O(n)$ time.

Algorithm 2: Remove Duplicate Data

```
    Input: join sequence relation R_J, unique attribute set U
    Output: Relation R_U with no duplicate data
 1  if {start, end} ⊄ R_J then
 2  |   R_U = Π_U(R_J);
 3  else
 4  |   R_U = ∅; prev = null;
 5  |   U' = U − {start, end};
 6  |   R_Π = Π_U(R);
 7  |   R_sorted = R_Π.sort(key = U' ∪ {start});
 8  |   foreach tuple t ∈ R_sorted do
 9  |   |   if prev is null then
10  |   |   |   prev = t;
11  |   |   else if prev.U' == t.U'  and prev.end >= t.start then
12  |   |   |   prev.end = max(prev.end, t.end);
13  |   |   else
14  |   |   |   R_U.append(prev);
15  |   |   |   prev = t;
16  |   if prev ≠ null then
17  |   |   R_U.append(prev);
18  return R_U
```

Example 2 (Case 1: Temporal object). Consider query {COUNT Employee GROUPBY Project DURING [2017,2018]} to count employees who participated in each project between 2017 to 2018. We generate a join sequence relation R_J from relations $Employee$, $ParticipateIn$, $Project$. Since COUNT is applied on Eid of temporal object relation $Employee$, and GROUPBY on Pid of temporal object relation $Project$, we have $\mathcal{U} = (Eid, Pid, R_J.start,\ R_J.end)$.

Example 3 (Case 2: Temporal attribute). Recall query Q_1 ={Employee SUM Salary Project AI DURING [2018]} in Example 1, and its join sequence relation with duplicated data highlighted in Fig. 4. From the temporal ORM schema graph, we know that $Salary$ is a temporal attribute of the object Employee whose identifier is Eid. Hence, $\mathcal{U} = \{Eid, Salary, Join_Start, Join_End\}$. Algorithm 2 is used to remove the data duplicates on $\mathcal{U}$.

Example 4 (Case 2: Non-temporal attribute.). Consider query Q_2 ={Employee AVG AGE() department CS DURING [2018]} where AGE() is a predefined function to compute the difference in the years between attribute DOB in *Employee* and the start year in the query period. Fig. 5 shows the join sequence relation obtained from temporal join sequence with relations $Employee$, $WorkFor$ and *Department*. The tuples with data duplicates are highlighted. Since an employee's age is computed from the non-temporal attribute DOB of the temporal object type *Employee*, we identify the unique values on $\mathcal{U} = \{Eid, DOB\}$ from the join sequence relation, i.e., {(E02,1978-05-20), (E03,1990-10-15)}. Then the age for $E02$ is 40 and $E03$ is 28. Hence, the average age is 34.

3.3 Compute Temporal Aggregates

Let $R_{\mathcal{U}}$ be the join sequence relation with no data duplicates over unique attribute set $\mathcal{U}$. If $R_{\mathcal{U}}$ does not contain time period attributes, a traditional aggregate is computed over it. Otherwise, a temporal aggregate is computed. Our

JoinSequenceRelation_1

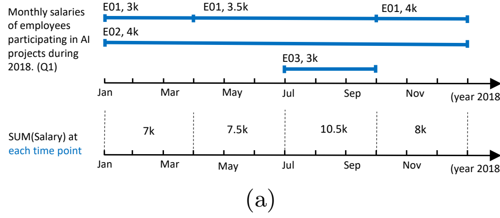
Eid	Salary(per month)	Pid	Join_Start	Join_End
E01	3k	P01	2018-01-01	2018-03-31
E01	3.5k	P01	2018-04-01	2018-09-30
E01	4k	P01	2018-10-01	2018-12-31
E02	4k	P01	2018-01-01	2018-12-31
E01	3k	P03	2018-01-01	2018-03-31
E01	3.5k	P03	2018-04-01	2018-06-30
E02	4k	P03	2018-10-01	2018-12-31
E03	3k	P03	2018-07-01	2018-09-30

Fig. 4. join sequence relation for Q_1.

JoinSequenceRelation_2

Eid	Did	DOB	Join_Start	Join_End
E02	D01	1978-05-20	2018-01-01	2018-02-28
E02	D01	1978-05-20	2018-05-01	2018-12-31
E03	D01	1990-10-15	2018-01-01	2018-10-31

Fig. 5. join sequence relation for Q_2.



(a)

SUM(Salary) at each time point

SUM (Salary)	Aggr_Start	Aggr_End
7k	2018-01-01	2018-03-31
7.5k	2018-04-01	2018-06-30
10.5k	2018-07-01	2018-09-30
8k	2018-10-01	2018-12-31

(b)

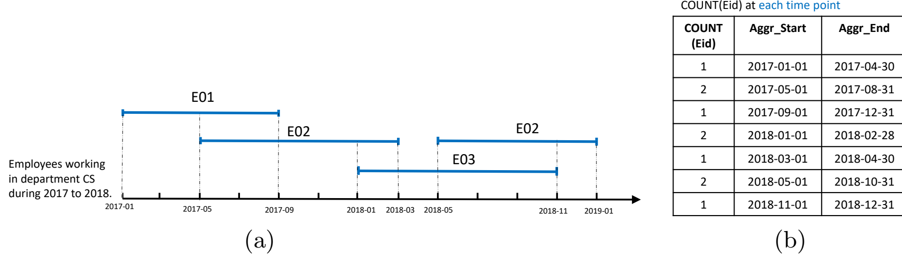
Fig. 6. (a) Sum of salaries and (b) Aggregate results at each time point for Q_1.

framework supports two types of temporal aggregate computation: (1) aggregate at each time point, and (2) aggregate over user-specified time unit. Moreover, for attributes with inherent time unit, such as monthly salary, we provide the option of computing a time-weighted sum to generate meaningful results.

Aggregate at Each Time Point. This captures the aggregate result at each time point along a timeline. A new aggregate value is computed using temporal aggregate operators [6,7] when the data is updated, and each aggregate value is associated with a non-overlapped time period indicating when the value is computed. Fig. 6(a) visualizes the non-duplicated data in $R_{\mathcal{U}}$ of query Q_1, and how the temporal aggregates at each time point are computed. The salary of employees $E01$, $E02$ and $E03$ change over time, and each change leads to a new aggregate result. The timeline in Fig. 6(a) is segmented by each salary change, and a sum is computed over the monthly salaries in each segment. Fig. 6(b) gives the aggregate results, capturing the changes in the sum of the monthly salary for employees participating in AI projects in 2018.

Aggregate over User-Specified Time Unit. Computing aggregate at each time point may lead to overly detailed results when updates are frequent, e.g., bank account balances. Thus, we provide users the option of specifying the granularity of the time unit with a reserved word YEAR, MONTH or DAY in the group-by condition. This will return one aggregate tuple per year or month or day respectively. For example, the query Q_3 ={COUNT Employee Department CS GROUPBY YEAR DURING [2017,2018]} counts employees who have worked for the CS department each year from 2017 to 2018. Two results will be returned for the COUNT aggregate: one for 2017, and another for 2018.

Fig. 7(a) gives the visualization of data used to compute the results for query Q_3. If we simply utilize the operator in [3] which applies the traditional aggregate function to tuples whose time periods overlap with the specified time unit, we



COUNT(Eid) at each time point

COUNT (Eid)	Aggr_Start	Aggr_End
1	2017-01-01	2017-04-30
2	2017-05-01	2017-08-31
1	2017-09-01	2017-12-31
2	2018-01-01	2018-02-28
1	2018-03-01	2018-04-30
2	2018-05-01	2018-10-31
1	2018-11-01	2018-12-31

Fig. 7. (a) Data visualization (b) Aggregate results at each time point for Q_3.

will get incorrect results for 2018 because the same employee E02 is counted twice, and the different work periods of employees are not taken into account (see Fig. 8(a)). Instead, we compute a summary on top of the aggregate results at each time point based on the user-specified time unit (per year/month/day). The idea is to determine the min, max and time-weighted average for tuples whose time periods overlap with the time unit. The min and max reflect the extreme aggregate values over each time unit, while the time-weighted average captures the time period of the aggregate result at each time point.

Let $\mathcal{R}_{aggr}$ be the relation of aggregate results obtained at each time point. We use the user-specified time unit to segment the timeline in $\mathcal{R}_{aggr}$ into a set of consecutive non-overlap time periods W. Then we find the tuples in each time period $[s, e] \in W$, and compute the max, min, time-weighted average over these tuples, as a summary of aggregate results over this $[s, e]$. The max and min values can be computed directly, while the average is obtained as follows:

$$AVG^T(\mathcal{R}_{aggr}, [s,e]) = \frac{\sum_{t \in (\mathcal{R}_{aggr} \bowtie^T [s,e])} t.value \times |t.period|}{|[s,e]|} \tag{1}$$

where $R_{aggr} \bowtie^T [s, e]$ are the tuples in $\mathcal{R}_{aggr}$ that overlap with time period $[s, e]$, each tuple value $t.value$ is weighted by the length of its time period $|t.period|$ to compute a time-weighted sum, and the weighted sum is divided by the length of period $[s, e]$ to compute the time-weighted average. Note that the time unit used to compute the length of $t.period$ and $[s, e]$ are the same.

Fig. 8(b) shows the results for Q_3 after computing the max, min and time-weighted average over each year based on the aggregate results at each time period in Fig. 7(b). For 2017, the min and max count of employees in the CS department is 1 and 2 respectively. The time-weighted average in 2017 is given by $\frac{1 \times 4(months) + 2 \times 4(months) + 1 \times 4(months)}{12(months)} = 1.33$ indicating that there are 1.33 full-time equivalent employees in the CS department in 2017. In the same way, we compute that there are 1.67 full-time equivalent employees in 2018.

Algorithm 3 gives the detail of the aggregate computation. The key idea is to conduct a sort-merge temporal join between relation $\mathcal{R}_{aggr}$ and the time line segmentation with given user-specified time unit w. We first generate the $startList$ of each segment and sort $\mathcal{R}_{aggr}$ and $startList$ in ascending order over start time. Since $\mathcal{R}_{aggr}$ might have some group-by attributes, its tuples are first sorted by the group-by attributes in Y and then by the start time within each

COUNT (*)	Aggr_Start	Aggr_End
2	2017-01-01	2017-12-31
3	2018-01-01	2018-12-31

(a) Incorrect results using [3]

COUNT(Eid) per year

MIN	MAX	AVG^T	Aggr_Start	Aggr_End
1	2	1.33	2017-01-01	2017-12-31
1	2	1.67	2018-01-01	2018-12-31

(b) Correct results using our approach

Fig. 8. Results of count of employees per year in Q_3.

group (lines 1-3). Then, we conduct a linear scan on tuples in $\mathcal{R}_{aggr}$, and handle those tuples belonging to the same group with function RwPerGroup() (lines 4-7). This function takes the row index i referring to the first tuple of current group as input, and return the row index of the first tuple of next group as output.

Within function RwPerGroup(), we go over each segment represented by its start time in $startList$, and find the tuples in $\mathcal{R}_{aggr}$ corresponds to each segment. First, the segment that do not overlap with the time period of current tuple t are skipped (line 12-14). Then, for the segment with corresponding tuples, we compute the max, min and time-weighted average as we introduced before on those tuples (lines 15-24). As the tuples in $\mathcal{R}_{aggr}$ are handled one by one, we will reach the end of relation $\mathcal{R}_{aggr}$ (lines 25-26) or tuples in next group (lines 27-28), which results in the exit of this function.

Aggregate over Attributes with Inherent Time Unit. Some attributes have inherent time units, e.g., monthly salary or daily room rate, and it is often useful to compute the sum of these attributes over their time unit. Such inherent time unit is obtained from the metadata of database. For instance, Fig. 1 has an attribute $Salary$ which captures the monthly salary of employees, i.e., the time unit of $Salary$ is per month.

Similar to the time-weighted average in Equation (1), we also weight the attribute values to compute the sum of a temporal attribute with an inherent time unit over a time period. Here, the weight is the ratio of the length of the attribute's time period over the inherent time unit. Given a join sequence relation with duplicates removed $R_{\mathcal{U}}$, let A_z be a temporal attribute with an inherent time unit z. The accumulated sum SUM^T over $[s, e]$ of A_z is given by

$$SUM^T(R_{\mathcal{U}}, A_z, [s, e]) = \sum_{t \in (R_{\mathcal{U}} \bowtie^T [s,e])} t.A_z \times \frac{|t.period|}{z} \tag{2}$$

Note that if we compute both SUM^T and AVG^T over the same time period $[s, e]$, then we can derive AVG^T directly as $SUM^T \times \frac{z}{|[s,e]|}$.

Consider query Q'_1 ={Employee SUM Salary Project AI GROUPBY YEAR DURING [2018]} which is similar to Q_1 but has a group-by condition with user-specified time unit YEAR. Fig. 9(a) shows the aggregate results for Q'_1 computed based on the aggregate results in Fig. 6(b). Since $Salary$ has an inherent time unit "month" (z =1(month)), we compute an accumulated SUM in 2018 based on the unique salaries in Fig. 6(a): $3k \times \frac{3(months)}{1(month)} + 3.5k \times \frac{6(months)}{1(month)} + ... + 3k \times \frac{3(months)}{1(month)} = 99k$. The result is shown in Fig. 9(b), which reflects the actual sum of salary paid to employees participating in AI projects in 2018.

Algorithm 3: Aggregate over User-Specified Time Unit

```
   Input: Aggregate results at each time point R_aggr, group-by attribute set Y, time unit w
   Output: Relation with aggregate results over user-specified time unit R_w
 1 Partition the timeline in R_aggr with w and let startList be the list of start time of each
    segment;
 2 startList.sort(asc = True);
 3 R_aggr.sort(key = Y ∪ {start}, asc = True);
 4 R_w = ∅; i = 0;
 5 while i ≠ −1 and i < len(R_aggr) do
       /* Compute aggregate over time unit w for tuples belonging to the same group   */
 6     i = RwPerGroup(R_aggr,R_w,Y,startList,w,i);
 7 return R_w

 8 Function RwPerGroup(R_aggr,R_w,Y,startList,w,i):
 9     t = R_aggr.atRow(i); grp = t.Y; j = 0;
10     while j < len(startList) do
11         s = startList[j]; e = s + |w| − 1;
           /* Skip the segment not overlapping with t.period                          */
12         while not isOverlapped([s,e],t.period) do
13             j+ = 1;
14             s = startList[j]; e = s + |w| − 1;
           /* Compute max, min and time-weighted average                             */
15         maxV = −∞; minV = +∞; sumV = 0;
16         while t ≠ null and t.Y == grp and isOverlapped([s,e],t.period) do
17             maxV = max(maxV, t.value);
18             minV = min(minV, t.value);
19             l = min(t.end, e) − max(t.start, s);
20             sumV = sumV + t.value ∗ l;
21             i+ = 1;
22             t = R_aggr.atRow(i);
23         avgV = sumV/|w|;
24         R_w = R_w ∪ {(grp, maxV, minV, avgV, s, e)}
           /* Check if tuple t still belongs to current group                        */
25         if t == null then
26             return -1;
27         else if t.Y! = grp then
28             return i;
29         else
30             j+ = 1;
31             s = startList[j]; e = s + |w| − 1;
```

MIN	MAX	AVG^T	Start	End
7k	10.5k	8.25k	2018-01-01	2018-12-31

(a) Results using our approach

SUM(Salary)	Aggr_Start	Aggr_End
99k	2018-01-01	2018-12-31

(b) Accumulated SUM

Fig. 9. Results of aggregate per year for Q_1'.

4 Experimental Evaluation

We implemented our algorithms in Python and evaluated the effectiveness of our proposed approach to process temporal keyword queries correctly. We use the temporal operators in [7] to compute the temporal join in temporal join sequence and the temporal aggregate at each time point.

We synthesized a database with temporal ORA semantics, including temporal many-to-one relationship type, temporal many-to-many relationship type, and temporal attribute, which often lead to data duplicates in the join sequence relation. Table 1 shows the schema. We have 10,000 employees, 121 departments and 5,000 projects. The average number of salaries per employee is 3.86, average

Table 1. Schema of company database

Employee(Eid, Employee_Start, Employee_End, Ename, DOB)
Department(Did, Dname)
Project(Pid, Project_Start, Project_End, Pname, Budget, Did)
EmployeeSalary(Eid, Salary_Start, Salary_End, Salary(per month))
WorkFor(Eid, WorkFor_Start, WorkFor_End, Did)
ParticipateIn(Eid, Pid, ParticipateIn_Start,ParticipateIn_End)

number of departments per employee is 1.23, average number of projects per employee is 11.26, and the average number of projects per department is 333.3.

Correctness of Query Results. For each temporal keyword query in Table 2, we examine the correctness of the temporal aggregate results before and after removing duplicate data in the join sequence relation. The correct results are obtained from data that has no duplicates. Table 3 shows the results, as well as the size of the join sequence relation in terms of the number of tuples and attributes, and the percentage of duplicate tuples. We observe that the percentage of duplicate tuples in the join sequence relations for different keyword queries ranges from 1.3% to 99.5%. The data duplication arises from repeats of the same temporal relationship (C1), join between temporal and non-temporal semantics (C2), keyword matching multiple tuples (C3), and join between multiple many-to-many and/or many-to-one relationships (C4), as we will elaborate.

Queries C1 and C2 have aggregate functions over *non-temporal* semantics. The aggregate in C1 is applied to a predefined function AGE() which computes the age of employees based on the difference between the start time in the query and the non-temporal attribute *DOB* in relation *Employee*. Since one employee could leave a department and return to the same department in 2019, so his age would occur multiple times for this department, leading to an incorrect average employee age. By removing the data duplication on unique attribute set $\mathcal{U}$ = (Eid,DOB,Did) from join sequence relation R_J =(Eid,DOB,Did,Join_Start,Join_End), we are able to obtain correct average age.

The aggregate in C2 is applied to the non-temporal object Department, and the cause of duplicate data in the join sequence relation is due to the join between the non-temporal relation *Department* and the temporal relations *WorkFor*, *Employee* and *EmployeeSalary*. A department is duplicated many times in this process, since it can have many employees with salaries more than 10000, and an employee can have many salary records. By removing data duplication on $\mathcal{U}$ =(Did) from R_J =(Did,Join_Start,Join_End), we get the correct result 121, while the incorrect result is 23739 without the duplicates removed.

Queries C3 to C6 have aggregates over *temporal* semantics. Temporal aggregate operators are used to compute aggregate at each time point, and the data duplicates over time periods would lead to incorrect results in the corresponding time periods. The reason for duplicate data in C3 is query keyword matching multiple tuples, i.e., multiple projects match the keyword "AI" with common employees, leading to 17.9% of salary values are duplicated over time. We remove the data duplication on $\mathcal{U}$ =(Eid,Salary,Join_Start,Join_End) from R_J =(Eid,Salary,Pid,Did,Join_Start,Join_End) and obtain the correct sum of salaries over time.

Table 2. Temporal keyword queries and their search intentions.

	Keyword Query	Search Intention
C1	Employee AVG AGE() GROUPBY Department DURING [2019]	Compute the average age of employees in each department in 2019.
C2	COUNT Department Employee Salary > 10000 DURING [2010,2019]	Count the departments which had employee salary larger than 10,000 during 2010 to 2019.
C3	Employee SUM salary Project AI DURING [2015,2019]	Sum the monthly salaries of employee working in AI project during 2015 to 2019.
C4	COUNT Employee Project Department CS DURING [2019]	Count the employees who participated in projects belonging to department CS in year 2019.
C5	Employee MAX Salary GROUPBY Department DURING [2010,2019]	Compute the maximum salary for employees working in each department during 2010 to 2019.
C6	COUNT Project Budget >= 2000000 GROUPBY Department DURING [2018,2019]	Count projects whose budget is larger or equal to 2,000,000 in each department during 2018 to 2019.

Table 3. Correctness of results before and after removing duplicates.

Query	Size of join sequence relation		%Duplicate	Correct before	Correct after
	#tuple	#attribute	tuples	duplicate removed	duplicate removed
C1	12295	6	1.3	N	Y
C2	23739	5	99.5	N	Y
C3	15489	10	17.9	N	Y
C4	302	6	81.8	N	Y
C5	40917	5	0	Y	Y
C6	1922	4	0	Y	Y

For query C4, the reason for its data duplicates is due to the join between many-to-many relationship type in relation *ParticipateIn* and the many-to-one relationship type in relation *Project*. Specifically, the department CS could have multiple projects involving the participation of same employees, and such employees are duplicated in the join sequence relation. By identifying $\mathcal{U}$ =(Eid,Join_Start,Join_End) from join sequence relation R_J =(Eid,Pid,Did, Join_Start, Join_End), we are able to remove then 81.8% duplicate employees over time and get the correct employee count.

Queries C5 to C6 do not have data duplicates in the join sequence relation. For query C5, the aggregate MAX is not an accumulative function, hence the results are not affected by data duplicates, even if they exist. The reason that C5 and C6 do not have data duplication is that they have identical unique attribute set $\mathcal{U}$ and temporal join sequence relation R_J. For C5, $\mathcal{U} = R_J =$ (Eid,Salary,Did,Join_Start,Join_End). For C6, $\mathcal{U} = R_J =$ (Pid,Did,Join_Start,Join_End). Therefore, there is no duplicates on $\mathcal{U}$ to be removed.

Usefulness of Aggregate over User-Specified Time Unit. When an aggregate function is applied to some temporal semantics, a temporal aggregate operator is required to compute the aggregate at each time point (recall Section 3.3). Table 4 shows the number of tuples in the result table for queries C3 to C6, and a sample of the aggregate results. The average length of the time period for the aggregate results of C3 to C6 are 1.01, 13.51, 19.65, 1.13 days, respectively. These results are rather fine-grained due to the frequent updates to the temporal database. We see that some of the results have the same aggregate values in the consecutive time periods, e.g., query C5. This is because the temporal aggregate computation follows the change preservation property [4], which implies that a new tuple will be generated in the result table based on the update even though it does not lead to a new aggregate value. Hence, we provide

Table 4. Sample results for temporal *aggregate at each time point.*

<table>
<tr><th>Q#</th><th>#Tuples</th><th>Sample aggregate results</th><th>Q#</th><th>#Tuples</th><th>Sample aggregate results</th></tr>
<tr><td>C3</td><td>1807</td><td><table><tr><th>Aggr_Start</th><th>Aggr_End</th><th>SUM(Salary)</th></tr><tr><td>2015-01-01</td><td>2015-01-01</td><td>14657665</td></tr><tr><td>2015-01-02</td><td>2015-01-02</td><td>14668065</td></tr><tr><td>2015-01-03</td><td>2015-01-03</td><td>14668938</td></tr><tr><td>2015-01-04</td><td>2015-01-04</td><td>14652725</td></tr></table></td><td>C4</td><td>27</td><td><table><tr><th>Aggr_Start</th><th>Aggr_End</th><th>COUNT(Eid)</th></tr><tr><td>2019-01-01</td><td>2019-02-01</td><td>36</td></tr><tr><td>2019-02-02</td><td>2019-02-20</td><td>36</td></tr><tr><td>2019-02-21</td><td>2019-02-23</td><td>36</td></tr><tr><td>2019-02-24</td><td>2019-02-26</td><td>37</td></tr></table></td></tr>
<tr><td>C5</td><td>427,101</td><td><table><tr><th>Did</th><th>Aggr_Start</th><th>Aggr_End</th><th>MAX(Salary)</th></tr><tr><td>D1</td><td>2010-03-16</td><td>2010-03-16</td><td>3300</td></tr><tr><td>D1</td><td>2010-03-17</td><td>2010-03-17</td><td>3300</td></tr><tr><td>D1</td><td>2010-03-18</td><td>2010-03-18</td><td>3300</td></tr><tr><td>D1</td><td>2010-03-19</td><td>2010-03-19</td><td>3300</td></tr></table></td><td>C6</td><td>9,690</td><td><table><tr><th>Did</th><th>Aggr_Start</th><th>Aggr_End</th><th>COUNT(Pid)</th></tr><tr><td>D1</td><td>2018-01-01</td><td>2018-01-01</td><td>78</td></tr><tr><td>D1</td><td>2018-01-02</td><td>2018-01-02</td><td>79</td></tr><tr><td>D1</td><td>2018-01-03</td><td>2018-01-03</td><td>79</td></tr><tr><td>D1</td><td>2018-01-04</td><td>2018-01-04</td><td>79</td></tr></table></td></tr>
</table>

an option that allows users to specify the time unit, e.g., year or month, in the keyword query's group-by condition. Then we will compute one aggregate tuple for each time unit based on the previous aggregate results at each time point.

We extend queries C3 to C6 in Table 2 with time units in the group-by condition such that the results of C3, C5 and C6 are grouped by year, and that of C4 by the month. Table 5 shows the new queries C3' to C6', as well as their statistics and sample aggregate results. We observe that the results of C3' to C6' are more meaningful and easier to understand compared to the results in Table 4. This is because the time period in the results are computed based on user-specified time unit and they are significantly longer than the time periods in the original queries C3 to C6.

Consider query C5' in Table 5. By including a GROUPBY YEAR in the keyword query, we have one tuple generated for each combination of department and year. The first tuple in results indicates that for department D1 and year 2010, the minimum of the employees' maximum monthly salary is 3300, while the maximum is 20,000. Given an employee's maximum monthly salary over time, the AVG^T gives an average by considering the duration of each maximum salary, so users will know that the time-weighted average maximum salary is 13303.44. Similarly, the first tuple in results of query C6' indicates that for department D1 and year 2018, the minimum number of projects with a budget larger than or equal to two million is 78, while the maximum is 90. By considering the duration of such projects, there are 82.7 equivalent projects with a duration of one year.

5 Related Work

Existing works on keyword search over temporal databases have focused on improving query efficiency [12,15] and identifying query interpretations [9]. The work in [12] annotates parent nodes in a data graph with time boundaries computed from its child nodes, while the work in [15] extends the traditional keyword search approach BANKS[1] with time dimension. The work in [9] uses ORA semantics to apply the time condition in a query. These works do not support aggregate functions and group-by in their temporal keyword query.

SQAK [18] is the first work that supports keyword search with aggregate functions and group-by over non-temporal relational databases. However, it does not handle the data duplication in the join sequence relation and may return incorrect aggregate results. PowerQ [22] addresses the data duplication caused

Table 5. Statistics and sample results for keyword queries involving temporal *aggregates over user-specified time unit.*

<table>
<tr><th colspan="2">Keyword Query</th><th>#Tuple</th><th>Sample aggregate results</th></tr>
<tr><td>C3'</td><td>Employee SUM salary Project AI GROUPBY YEAR DURING [2015,2019]</td><td>5</td><td>
<table>
<tr><th>Aggr_Start</th><th>Aggr_End</th><th>Min (SUM(Salary))</th><th>Max (SUM(Salary))</th><th>AVG^T (SUM(Salary))</th></tr>
<tr><td>2015-01-01</td><td>2015-12-31</td><td>14115375</td><td>16122708</td><td>15360017.46</td></tr>
<tr><td>2016-01-01</td><td>2016-12-31</td><td>15357130</td><td>16005648</td><td>15651395.31</td></tr>
<tr><td>2017-01-01</td><td>2017-12-31</td><td>15136870</td><td>15897157</td><td>15538076.16</td></tr>
<tr><td>2018-01-01</td><td>2018-12-31</td><td>15408131</td><td>16535220</td><td>16106415.25</td></tr>
</table>
</td></tr>
<tr><td>C4'</td><td>COUNT Employees Project Department CS GROUPBY MONTH DURING [2019]</td><td>12</td><td>
<table>
<tr><th>Aggr_Start</th><th>Aggr_End</th><th>Min (COUNT(Eid))</th><th>Max (COUNT(Eid))</th><th>AVG^T (COUNT(Eid))</th></tr>
<tr><td>2019-01-01</td><td>2019-01-31</td><td>36</td><td>36</td><td>36.00</td></tr>
<tr><td>2019-02-01</td><td>2019-02-28</td><td>36</td><td>38</td><td>36.25</td></tr>
<tr><td>2019-03-01</td><td>2019-03-31</td><td>37</td><td>38</td><td>37.58</td></tr>
<tr><td>2019-04-01</td><td>2019-04-30</td><td>37</td><td>39</td><td>37.93</td></tr>
</table>
</td></tr>
<tr><td>C5'</td><td>Employee MAX salary GROUPBY Department YEAR DURING [2010,2019]</td><td>22,990</td><td>
<table>
<tr><th>Did</th><th>Aggr_Start</th><th>Aggr_End</th><th>Min (MAX(Salary))</th><th>Max (MAX(Salary))</th><th>AVG^T (MAX(Salary))</th></tr>
<tr><td>D1</td><td>2010-03-16</td><td>2010-12-31</td><td>3300</td><td>20000</td><td>13303.44</td></tr>
<tr><td>D1</td><td>2011-01-01</td><td>2011-12-31</td><td>20000</td><td>20000</td><td>20000.00</td></tr>
<tr><td>D1</td><td>2012-01-01</td><td>2012-12-31</td><td>20000</td><td>21600</td><td>20935.52</td></tr>
<tr><td>D1</td><td>2013-01-01</td><td>2013-12-31</td><td>21600</td><td>21600</td><td>21600.00</td></tr>
</table>
</td></tr>
<tr><td>C6'</td><td>COUNT Project Budget >= 2000000 GROUPBY Department YEAR DURING [2018,2019]</td><td>30</td><td>
<table>
<tr><th>Did</th><th>Aggr_Start</th><th>Aggr_End</th><th>Min (COUNT(Pid))</th><th>Max (COUNT(Pid))</th><th>AVG^T (COUNT(Pid))</th></tr>
<tr><td>D1</td><td>2018-01-01</td><td>2018-12-31</td><td>78</td><td>90</td><td>82.70</td></tr>
<tr><td>D1</td><td>2019-01-01</td><td>2019-12-31</td><td>86</td><td>91</td><td>87.92</td></tr>
<tr><td>D10</td><td>2018-01-01</td><td>2018-12-31</td><td>66</td><td>87</td><td>74.15</td></tr>
<tr><td>D10</td><td>2019-01-01</td><td>2019-12-31</td><td>66</td><td>73</td><td>69.85</td></tr>
</table>
</td></tr>
</table>

by n-ary (n>2) non-temporal relationships and utilizes projection to remove duplicate data. However, PowerQ does not consider the data duplication caused by many-to-many, many-to-one relationships and time periods. Since PowerQ is the latest work supporting aggregate in non-temporal relational keyword search, our work is built on top of it to handle data duplication caused by more reasons, e.g., many-to-many relationships, many-to-one relationships and the join between temporal and non-temporal relations.

Except for querying temporal databases via keyword search as this work, a more commonly used approach is to query the database utilizing extended SQL query with the support of temporal operators, e.g., temporal join and temporal aggregate operators [14,13]. These works in this field are mainly focusing on improving the efficiency of temporal operators, which can be achieved by specially designed indices [3,13,16,24], time period partition algorithms [6,7] or parallel computation [5,17]. On the one hand, our framework could take advantage of these works to improve the efficiency of computing the temporal join sequence as well as temporal aggregate. On the other hand, our framework removes data duplication automatically and correctly, therefore avoids returning incorrect results of temporal aggregate, while querying with extended SQL requires the users noticing and manually removing the possible data duplicates, which is error-prone.

6 Conclusion

In this work, we have extended temporal keyword queries with aggregates and group-by, and described a framework to process these queries over temporal databases. Our framework addresses the problem of incorrect aggregate results due to data duplication in the join sequence relation, and meaningless aggregate

results due to updates. We use a temporal ORM schema graph to capture temporal objects/relationships and temporal attributes, and use these semantics to identify an unique attribute set required by the aggregate in the join sequence relation. We also support aggregation over user-specified time units and attributes with inherent time units to return meaningful results.

Limitations of this approach include not considering recursive relationships in database and the extraction of ORA semantics is not fully automatic. Since one keyword query may have multiple query interpretations, the future work includes to design a mechanism to rank these interpretations, and propose algorithms to share some intermediate relations when computing results for different query interpretations of the same keyword query.

References

1. B. Aditya, G. Bhalotia, S. Chakrabarti, and et.al. BANKS: browsing and keyword searching in relational databases. In *VLDB*, 2002.
2. J. F. Allen. Maintaining knowledge about temporal intervals. *CACM*, 1983.
3. M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, 2006.
4. M. H. Böhlen and C. S. Jensen. Sequenced semantics. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2018.
5. P. Bouros and N. Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *Proc. VLDB Endow.*, 2017.
6. F. Cafagna and M. H. Böhlen. Disjoint interval partitioning. *VLDB J.*, 2017.
7. 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 TODS*, 2016.
8. Q. Gao, M. L. Lee, G. Dobbie, and Z. Zeng. A semantic framework for designing temporal SQL databases. In *ER*, 2018.
9. Q. Gao, M. L. Lee, T. W. Ling, G. Dobbie, and Z. Zeng. Analyzing temporal keyword queries for interactive search over temporal databases. In *DEXA*, 2018.
10. H. Gunadhi and A. Segev. Query processing algorithms for temporal intersection joins. In *IEEE ICDE*, 1991.
11. N. Hormozi. Disambiguation and result expansion in keyword search over relational databases. In *ICDE*. IEEE, 2019.
12. X. Jia, W. Hsu, and M. L. Lee. Target-oriented keyword search over temporal databases. In *DEXA*. Springer, 2016.
13. M. Kaufmann, A. Manjili, P. Vagenas, and et.al. Timeline index: unified data structure for processing queries on temporal data in SAP HANA. In *SIGMOD*, 2013.
14. K. G. Kulkarni and J. Michels. Temporal features in SQL: 2011. *SIGMOD Record*, 2012.
15. Z. Liu, C. Wang, and Y. Chen. Keyword search on temporal graphs. *TKDE*, 2017.
16. D. Piatov and S. Helmer. Sweeping-based temporal aggregation. In *SSTD*, 2017.
17. M. Pilman, M. Kaufmann, F. Köhl, D. Kossmann, and D. Profeta. Partime: Parallel temporal aggregation. In *ACM SIGMOD*, 2016.
18. S. Tata and G. Lohman. Sqak: doing more with keywords. In *SIGMOD*, 2008.
19. J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 2010.

20. Z. Yu, A. Abraham, X. Yu, Y. Liu, J. Zhou, and K. Ma. Improving the effectiveness of keyword search in databases using query logs. *Eng. Appl. Artif. Intell.*, 2019.
21. Z. Zeng, Z. Bao, T. N. Le, M. L. Lee, and T. W. Ling. Expressq: Identifying keyword context and search target in relational keyword queries. In *CIKM*, 2014.
22. Z. Zeng, M. L. Lee, and T. W. Ling. Answering keyword queries involving aggregates and group-bys on relational databases. In *EDBT*, 2016.
23. D. Zhang, Y. Li, X. Cao, J. Shao, and H. T. Shen. Augmented keyword search on spatial entity databases. *VLDBJ*, 2018.
24. D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. On computing temporal aggregates with range predicates. *ACM TODS*, 2008.