A Data Services Composition Approach for Continuous Query on Data Streams

Guiling Wang[1,2(✉)], Xiaojiang Zuo[1], Marc Hesenius[3], Yao Xu[2], Yanbo Han[1], and Volker Gruhn[3]

[1] Beijing Key Laboratory on Integration and Analysis of Large-Scale Stream Data, North China University of Technology, No. 5 Jinyuanzhuang Road, Shijingshan District, Beijing 100144, China
wangguiling@ict.ac.cn

[2] Ocean Information Technology Company, China Electronics Technology Group Corporation (CETC Ocean Corp.), No. 11 Shuangyuan Road, Badachu Hi-Tech Park, Shijingshan District, Beijing 100041, China

[3] paluno - The Ruhr Institute for Software Technology, University of Duisburg-Essen, Schützenbahn 70, 45127 Essen, Germany

Abstract. We witness a rapid increase in the number of data streams due to Cloud Computing, Big Data and IoT development. We would like to access and share data streams using a data service approach. In this paper, we propose a flexible continuous data service model and a continuous data service composition algorithm for answering queries across data streams. Service operation instance is modeled as a view defined on data streams composed of two parts: a data part and a time synchronization part. The composition algorithm extends the traditional Bucket algorithm to find the contained rewriting of user query on views satisfying the containment relationship of both data part and time synchronization part. We also present use case and experimental studies indicating that the approach is effective and efficient.

Keywords: Data streams · Query rewriting · Data services
Service composition · Continuous query

1 Introduction

Web services technology is a general medium for sharing data and functionality and enabling cross-organization collaboration for enterprise and web systems. Data services [1] or data-providing services [2] are a kind of services that allow query-like access to an organization's data sources. Although the existing data processing framework provides composition models or query languages which allow us to retrieve desired data from multiple data sources, data services provide a flexible, controlled and standardized approach to access or query an organization's data sources without exposing its databases directly [3]. Furthermore,

Y. Cai et al. (Eds.): APWeb-WAIM 2018, LNCS 10988, pp. 106–120, 2018.
https://doi.org/10.1007/978-3-319-96893-3_9

when queries require to access data sources across organizations, several services can be composed to generate a response [4–6].

To bring the benefits of data services, we would like to access and share data streams using a data service approach. However, data streams are very different from traditional data sources. This makes the problem of data service modelling and composition challenging for accessing and sharing data streams. Firstly, unlike traditional snap-shot queries over data tables, queries over data streams are continuous. A continuous query is issued once and remains active for a long time. The answer to a continuous query is constructed progressively as new input stream tuples arrive [7]. Once executed, data services for queries on data streams need to continuously return results and consider temporal constraints. Secondly, for queries over multiple data sources, traditional data providing services are often modeled as parameterized views over data schemas [3,4]. Based on the service model, services can be composed using a query rewriting approach to answer queries over multiple data sources [3,4]. Because most of the stream query language do not support views [7], how to model data services as views over data streams is not trivial. And because queries for data streams need to be updated continuously, the traditional query rewriting approach is inapplicable to rewrite query over data streams directly.

In this paper, we introduce a data service model for continuous query over data streams, and call it "continuous data services". Service operation inputs are not modeled as fixed query conditions. They are arbitrary query conditions modeled as a set of optional attributes of the underlying data model and condition predicates. "sliding window" is introduced into the service model to describe the temporal feature of services. The instance of the service operation can be modeled as a view defined on data streams. Based on the continuous data service model, we propose a continuous data service composition algorithm for answering queries across data streams. It improve the Bucket algorithm [8] for "answering queries using views" on persistent relation data to find the contained rewriting by checking the containment relationship between time synchronization part of the query and the rewriting. We describe an implementation, a use case and provide a performance evaluation of the proposed approach.

The rest of this paper is organized as follows: In Sect. 2, we motivate the need for conjunctive queries across data streams, and discuss the underlying challenges. In Sect. 3, we describe the continuous data service model. In Sect. 4, we propose the continuous data service composition algorithm. In Sect. 5, we describe our implementation and evaluate our approach. We overview related work in Sect. 6. We provide concluding remarks and future research outlook in Sect. 7.

2 Motivation

In this section, we describe a motivating scenario we use throughout the paper. Various systems for maritime freight logistics collect data like vessel trajectories, vessel basic information and so on. Among these data sources, the data

stream `vesseltraj(mmsi, long, lat, speed)` records trajectory points of a vessel, where `mmsi` is the Maritime Mobile Service Identity, `long` and `lat` is the longitude and latitude of the vessel location, and `speed` is the vessel's speed. The relation data `vesselinfo(mmsi, imo, callsign, name, type, length, width, po-sitionType, eta, draught)`records static information of ships including the `mmsi`, the International Maritime Organization (imo) code, call sign, name, type, length, width, the Estimated Time of Arrival (eta), draught of the vessel. The relation data `vesseltravelinfo(mmsi, dest, source)` records the destination and the identification of the position message source.

These systems are subordinate to different management domains and won't expose full data access to their data sources directly. They provide access to the set of services with constraints described in Table 1. The underlying data streams of DS_1 are `vesselinfo` and `vesseltraj`. They have constraints that `mmsi` must be greater than `3000` and `speed` greater than `50 km` with a time-based sliding window of window size `5s` and slide size `1s`. The underlying data streams of DS_2 is `vesselinfo` and `vesseltraj`. The time window of the stream has window size `5s` and slide size `2s`. The underlying data stream of DS_3 is `vesseltraj`. This data stream has constraints that the `speed` must be less than `40 km` with window size `5s` and slide size `2s`. The underlying data streams of DS_4 are `vesseltravelinfo` and `vesseltraj`. This service has constraints that the `mmsi` must be less than `2000` with window size `5s` and slide size `2s`. We also express the underlying query of the services as conjunctive queries extended with time-based sliding window semantics. Note that join predicates in this notation are expressed by multiple occurrences of the same variables.

Table 1. Continuous data services in the ocean data query scenario

Service	Functionality and constraints	Formal expression of the underlying data streams
DS_1	Query on those vessels whose *mmsi* number greater than 3000 and speed greater than 50 km with a time-based sliding window of window size 5s and slide size 1s	*vesselinfo(mmsi, imo, callsign, name, type, length, width, positionType, eta, draught), vesseltraj(mmsi, long, lat, speed), mmsi* $>$ 3000, *speed* $\geq$ *50* km, *wsize(5), slide(1)*
DS_2	Query on those vessels with a time-based sliding window of window size 5s and slide size 2s	*vesselinfo(mmsi, imo, callsign, name, type, length, width, positionType, eta, draught), vesseltraj(mmsi, long, lat, speed), wsize(5), slide(2)*
DS_3	Query on those vessels whose speed is less than 40 km with a time-based sliding window of window size 5s and slide size 2s	*vesseltraj(mmsi, long, lat, speed), speed* $<$ *40* km, *wsize(5), slide(2)*
DS_4	Query on those vessels whose *mmsi* number less than 2000 with a time-based sliding window of window size 5s and slide size 2s	*vesseltravelinfo(mmsi, dest, source), vesseltraj(mmsi, long, lat, speed), mmsi* $<$ *2000, wsize(5), slide(2)*

Those services with sliding window constraints continuously push output to the service consumer once the consumer creates a connection with the service producer. The output is the query results in range of the configured window size that will be updated every slide size. So we call these services "continuous data services".

Now assume the following query asks for vessels that have outstanding speed over a defined sliding window. Note we express the query as conjunctive queries extended with time-based sliding window semantics. And note that join predicates in this notation are expressed by multiple occurrences of the same variables.

```
Q(mmsi, draught, dest, speed):-vesselinfo(mmsi, imo, callsign, name,
type, length, width, positionType, eta, draught), vesseltraj(mmsi, long,
lat, speed), vesseltravelinfo(mmsi, dest, source), speed ≥ 40 km,
wsize(5), slide(4)
```

Obviously service DS_3 is not useful to satisfy this query request, because DS_3 has information only on vessels whose speed is less than 40 km whereas we are interested in vessels which has speed greater than 40 km. Although DS_1 is relevant to user query, it only has `mmsi` information and need to retrieve destination information by invoking other service like DS_4. However, DS_1 only has information on vessels with `mmsi` greater than 3000, and DS_4 has information on vessels with `mmsi` less than 2000, meaning DS_1 and DS_4 are disjoint. So service DS_1 is also not useful to answer this user query. We are left with one possible plan to use the services to answer this query. Firstly invoke DS_2 to retrieve the list of vessels with a sliding window of window size `5s` and slide size `2s`. Then invoke DS_4 where `mmsi` is less than 2000 with a sliding window of window size `5s` and slide size `2s`. Results from both services are joint to answer `Q`. Note that the sliding window constraints of DS_2 and DS_4 is different, we also need to judge if the joint results can satisfy the query requirement. Also note that the results only vessels with `mmsi` less than 2000, which can satisfy the query is not equivalent with it. Note in this example, there is only one service composition plan satisfying the query, but there may be multiple plans in other examples.

3 Model of Continuous Data Service

3.1 Data Model

We use the synchronized relation model for describing the contents of data stream sources. The data model includes:

– `S` and $\Re$`(S)`. `S` is a tagged stream with the format of "`Tag`⟨`Attrs`⟩`ts`", where `Tag` can be either insert (+), update (u), or delete (-) and `ts` indicates the time at which the modification takes place. For detailed explanation of what is a tagged stream, please refer to [7]. Any tagged stream `S` has a corresponding time-varying relation $\Re$`(S)`. The relation is continuously modified by `S`'s tuples.

- `Attrs`. `Attrs` are the attributes of the time-varying relation $\Re$`(S)`.
- `ts`. `ts` is the time point where the relation $\Re$`(S)` is modified by the underlying `S`'s tuples.
- `sync`. `sync` synchronized stream is a special tagged stream "+⟨`timepoint`⟩`ts`", where `timepoint` represents a time point which is the only attribute of `sync`. Synchronized stream is a kind of tagged stream. So it also has a corresponding time-varying relation $\Re$`(sync)`.
- $\Re_{\texttt{sync}}$`(S)`. $\Re_{\texttt{sync}}$`(S)` is a synchronized relation of any arity. Figure 1 illustrates a synchronized stream of $\Re_{\texttt{sync}_2}$`(Vesseltraj)`. For traditional persistent data (e.g. data tables in a database), the tuples are reflected at any time. Here we denote the synchronized stream associated with the traditional persistent data as $\texttt{sync}_0$.

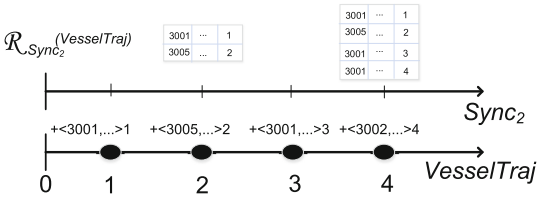


Fig. 1. A synchronized stream of $\Re_{\texttt{sync}_2}$`(Vesseltraj)`

`DataModel` of $\Re_{\texttt{Sync}}$`(S)` can be represented as a tuple: ⟨`Attrs, SyncUnits`⟩, where `Attrs` = {`attr`} is a set of attributes, `SyncUnits` is the subscript index of the synchronization stream `sync`. For example, the value of `SyncUnits` is 2 for $\texttt{sync}_2$, 3 for $\texttt{sync}_3$ and 4 for $\texttt{sync}_4$ etc.

3.2 Continuous Query Containment

Query containment and equivalence provide a formal framework to compare different queries in a data integration system. In relational databases, a query $\texttt{Q}_1$ is said to be contained in $\texttt{Q}_2$, denoted by $\texttt{Q}_1 \subseteq \texttt{Q}_2$, if and only if $\texttt{Q}_1\texttt{(D)} \subseteq \texttt{Q}_2\texttt{(D)}$ for any database instance `D`. $\texttt{Q}_1$ is *equivalent* to $\texttt{Q}_2$ if and only if $\texttt{Q}_1 \subseteq \texttt{Q}_2$ and $\texttt{Q}_2 \supseteq \texttt{Q}_1$.

In stream processing system, a continuous query over `n` tagged streams $\texttt{S}_1 \ldots \texttt{S}_\texttt{n}$ is semantically equivalent to a materialized view that is defined by a SQL expression over the time-varying relations $\Re\texttt{(S}_1\texttt{)} \ldots \Re\texttt{(S}_\texttt{n}\texttt{)}$[7]. The big difference between time-varying relations and traditional relations is that the time-varying relations have arbitrary refresh conditions. The solution is to isolate the time synchronization streams out of the continuous query expression. Then the containment relationship is tested from two aspects: (1) test data containment using traditional query containment test method, and (2) test synchronization containment.

For example, if we want to check the containment relationship of a query Q and a data service instance of DS′ like this:

$$\mathtt{Q(mmsi, draught, dest, speed)}\text{:-}\Re\mathtt{(T), I, TRAVEL, speed} \geq 40,$$
$$\mathtt{(currTime\text{-}5)} < \mathtt{TS} \leq \mathtt{currTime, sync_4}$$

and

$$\mathtt{Q_{DS'}(mmsi, speed, imo)}\text{:-}\Re\mathtt{(T), I, speed} \geq 30,$$
$$\mathtt{(currTime\text{-}5)} < \mathtt{TS} \leq \mathtt{currTime, sync_1}$$

We first test containment of time part of $\mathtt{Q_{DS'}}$ and Q. The synchronization relation part of Q (i.e. $\Re(\mathtt{sync_4})$) is contained in the synchronization relation part of $\mathtt{Q_{DS'}}$ (i.e. $\Re(\mathtt{sync_1})$). Because any tuples satisfied by the selection and projection conditions of Q also satisfied $\mathtt{Q_{DS'}}$, the data part of Q is contained in data part of $\mathtt{Q_{DS'}}$. We can conclude that Q is contained in $\mathtt{Q_{DS'}}$.

3.3 Continuous Data Service

We model a continuous service as a view defined on the underlying data streams. Any service subscribes one or multiple data streams or database tables, which is defined as Subs. Any service has zero to multiple operations in which inputs, outputs, window range, window slide size should be defined. Input and output parameters are from the attributes of the underlying synchronized relations corresponding with Subs. Every service instance publishes one tagged stream on message queue.

Such service can be expressed as follows: $\mathtt{DS} = \langle \mathtt{ID, SubS, PubS, Ops} \rangle$, where:

- ID is the unique identity of the service.
- SubS is the stream set of the service subscribed from message queue. $\mathtt{SubS} = \{\langle \mathtt{DS_{sub}, DataConstrs, TimeConstr} \rangle\}$, where $\mathtt{DS_{sub}}$ is a tagged stream defined in Section II. A Data model $\langle \mathtt{Attrs, SyncUnits} \rangle$ is corresponding with a time-varying relations $\Re(\mathtt{DS_{sub}})$. DataConstrs and TimeConstr are the constraints applied on content and time of the tagged stream.
- PubS is the stream set of the service published to message queue. $\mathtt{PubS} = \{\langle \mathtt{DS_{pub}, DataConstrs, TimeConstr} \rangle\}$, where $\mathtt{DS_{pub}}$ is a tagged stream. It is corresponding with a time-varying relation $\Re(\mathtt{DS_{pub}})$.
- $\mathtt{DataConstrs} = \{\mathtt{DataConstr}\}$, where $\mathtt{DataConstr} = \langle \mathtt{attr, condop, constant} \rangle$. attr is the attribute of $\Re(\mathtt{DS_{sub}})$ for SubS and $\Re(\mathtt{DS_{pub}})$ for PubS. condop can be one of the condition operator from $>, =, <, \geq, \neq, \leq$. constant is a constant value.
- $\mathtt{TimeConstr} = \langle \mathtt{range} \rangle$, where range is range size of the sliding window of synchronized relation. Note that tumbling window and hopping window are both a special form of the sliding window. For tumbling window, range size is equal to slide size. And for hopping window, range size is a multiple of slide size.

– $\texttt{Ops} = \{\langle \texttt{inputs}, \texttt{outputs}, \texttt{range}, \texttt{slide}\rangle\}$ is the service operations. $\texttt{inputs} = \{\texttt{input}\}$ are a set of attributes of $\texttt{S}_{\texttt{sub}}$, the corresponding condition operator $>, =, <, \geq, \neq, \leq$ and constants. $\texttt{outputs} = \{\texttt{output}\}$ are a set of output parameters of the service operation. `range` and `slide` are the time constraint of the service request. A *SyncSQL* expression can be generated from `Ops`.

The elements of the input and output set `Ops` are determined when a service is instantiated. `PubS` of a service are also determined when a service is instantiated.

Given a specific user inputs, the service has an associated instance. A service instance can also be defined as a query view on the underlying time-varying relations. We use the notation of *conjunctive queries* extended with synchronization stream to express the view. A data service $\texttt{DS} = \langle \texttt{ID}, \texttt{SubS}, \texttt{PubS}, \texttt{Ops}\rangle$ is transformed into a view:

$$\texttt{DS}(\bar{\texttt{X}})\text{:-}\Re(\texttt{S}_{\texttt{sub}_1}), \ldots, \Re(\texttt{S}_{\texttt{sub}_n}), \texttt{c}_1, \ldots, \texttt{c}_n, \texttt{tc}, \texttt{sync}_1 \cap \ldots \cap \texttt{sync}_n$$

where $\bar{\texttt{X}}$ is all the attributes from all $\texttt{S}_{\texttt{sub}}$ elements of `SubS`, $\Re(\texttt{S}_{\texttt{sub}_i})$ are the underlying time-varying relation corresponding with all the elements of `SubS`. Note that not all subscribed streams have data constraints applied on them. If $\texttt{S}_{\texttt{sub}_1}$ has no data constraint, we can add a data constraint $\texttt{c}$ on it: $\text{-}\infty \leq \texttt{c} \leq +\infty$. Thus all subscribed streams have data constraints represented as $\texttt{c}_1, \ldots, \texttt{c}_n$. `tc` is the intersection of all the window range size constraints applied on them. $\texttt{sync}_i$ is the synchronization stream applied on $\Re(\texttt{S}_{\texttt{sub}_i})$.

A service instance of $\texttt{S} = \langle \texttt{ID}, \texttt{SubS}, \texttt{PubS}, \texttt{Ops}\rangle$ can be transformed into a view like this:

$$\texttt{DS}(\bar{\texttt{X}})\texttt{inst}\text{:-}\Re(\texttt{S}_{\texttt{sub}_1}), \ldots, \Re(\texttt{S}_{\texttt{sub}_n}), \texttt{c}_1, \ldots, \texttt{c}_n, \texttt{c}_{\texttt{op}_1}, \ldots, \texttt{c}_{\texttt{op}_s}, \texttt{tc},$$
$$\texttt{sync}_1 \cap \ldots \cap \texttt{sync}_n, \cap \texttt{sync}_1 \cap \ldots \cap \texttt{sync}_t$$

$\texttt{c}_{\texttt{op}_1}, \ldots, \texttt{c}_{\texttt{op}_s}$ are data constraints from inputs of service operations. $\texttt{sync}_1 \cap \ldots \cap \texttt{sync}_t$ are synchronization stream from the time constraints of service operations. `tc` is the intersection of all the window range size constraints applied on $\Re(\texttt{Sub}_i)$ and from service operations.

4 Data Services Composition for Answering Continuous Query

When services and service instances are transformed into views on time-varying relations, given a conjunctive query `Q`, we need to find the service composition plans to answer it. The problem of answering conjunctive query using views for traditional persistent data is NP-complete [9]. Bucket algorithm or *minicon* algorithm are the approaches to drastically reduce the number of rewritings we need to consider for a query given a set of views. So we can improve the Bucket algorithm [8] or *MiniCon* algorithm [10] to find the service composition plans to answer query `Q`. Here we give the improved Bucket algorithm. The main idea of

Bucket algorithm is that we first consider each subgoal in the query in isolation, and determine which views may be relevant to that subgoal. Thus the number of query rewritings that need to be considered can be drastically reduced. In order to support finding relevant continuous data services or service instances, we improve the Bucket algorithm by adding the synchronization stream containment judgement and determining the service operation inputs and outputs after the relevant services are found.

The first step is shown in Algorithm 1. It constructs for each subgoal g in the query a bucket of relevant service or service instance atoms. In this algorithm, we check the containment relationship between the query sub-goal and the view transformed from the service or service instance.

Algorithm 1. Create buckets

Input: conjunctive query Q in two parts:
data part Q^d of the form:
$Q^d(\bar{X})$:-$\Re(R_1)(\bar{X}_1), \ldots, \Re(R_n)(\bar{X}_n), c_1, \ldots, c_n, tc$
synchronization part $Sync_Q$ of the form:
$Sync_Q = Sync_1 \cap \ldots Sync_n$;
a set of views $\mathcal{V}$ transformed from services $\mathcal{S}$ and service instances $\mathcal{S}inst$;
Output: list of buckets

1: **for** $1 \leq i \leq n$ **do**
2: Initialize $Bucket_i$ to $\emptyset$
3: **end for**
4: **for** each subgoal g_i in Q **do**
5: **for** each $V \in \mathcal{V}$ **do**
6: Let V be of the form:
$V(\bar{Y})$:-$\Re(S_1)(\bar{Y}_1), \ldots, \Re(S_m)(\bar{Y}_m), d_1, \ldots, d_m, sync_1 \cap \ldots \cap sync_m \cap sync_1 \cap \ldots \cap sync_t$
7: **if** $\Re(Sync_V) \subseteq \Re(Sync_q)$ **then**
8: **if** g_i is an element of subgoals set of V **then**
9: **if** each $x \in X_i$ is also an element of $\bar{Y}$ **then**
10: **if** the data constraints of V satisfy the data constraints of Q **then**
11: add V into $Bucket_i$
12: **end if**
13: **end if**
14: **end if**
15: **end if**
16: **end for**
17: **end for**

The second step considers all the possible combinations of services and service instances. Each combination should include one of the service or service instance atoms from every bucket. Generate the candidate composition plans by checking if each combination is satisfied (if there exists no self-contradictory in

the same combination). Keep those plans that is satisfied and delete those that is unsatisfied.

Algorithm 2. Check whether a candidate plan is equivalent

Input: candidate services and service instances composition plan $\mathtt{p}(\bar{\mathtt{Y}})$;
conjunctive query $\mathtt{Q}(\bar{\mathtt{X}})$;
a set of executable equivalent services and/or service instances composition plan `eqCompPlans`

Output: the updated result of `eqCompPlans`

Let the set of subgoals of `p` in the form of `goalsOfp`, and the subgoals of each plan `eqPlan` in `eqCompPlans` in the form of `goalsOfeqPlan`
Denote the intersection of data constraints of `p` and `Q` as $\mathtt{D} \cap \mathtt{C}$, where `D` is the data constraints set of `p` and `C` is the data constraints set of `Q`
Get all of the elements exist in set $\mathtt{D} \cap \mathtt{C}$ that don't exist in set of data constraints of `p`, denoted as $\mathtt{A} = \mathtt{D} \cap \mathtt{C} \setminus \mathtt{D}$. This set is the additional data constraints that should be added on `p` in order to be equivalent to `Q`
if $\mathtt{Q} \subseteq \mathtt{p}$ **then**
  if there exists no plan `eqPlan` in `eqCompPlans` satisfying the condition that $\mathtt{goalsOfeqPlan} \subset \mathtt{goalsOfp}$ **then**
    if there exists services (not service instance) in `p` **then**
      for each subgoal `g` of `p` **do**
        if `g` is a service **then**
          if $\mathtt{D} \cap \mathtt{C} \neq \emptyset$ **then**
            $\mathtt{A} = \mathtt{genInstance}(\bar{\mathtt{Y}} \cap \bar{\mathtt{X}}, \mathtt{A}, \mathtt{sync})$
          else
            $\mathtt{genInstance}(\bar{\mathtt{Y}} \cap \bar{\mathtt{X}}, \emptyset, \mathtt{sync})$
          end if
        end if
      end for
      if $\mathtt{A} = \emptyset$ **then**
        delete the redundant plan than `p` and add `p` into `eqCompPlans`
      end if
    else
      if $\mathtt{p} \subseteq \mathtt{Q}$ **then**
        delete the redundant plan than `p` and add `p` into `eqCompPlans`
      end if
    end if
  end if
end if

In the example explained in Sect. 2, the returned results contain only vessels with `mmsi` less than 2000, which is not equivalent with the query. In fact, the service composition plans that can answer user query can be divided into two categories: the equivalent composition plans and the contained composition plans. The former is equivalent with the query and the latter is contained in the query. There exists a maximally contained composition plan among the contained composition plans. So if a continuous query can be supported by mutiple

composition plans, we can choose the equivalent or maximally contained composition plan among the candidates.

The third step searches the equivalent service composition plans or the contained service composition plans. Take the equivalent service composition plan as the example, the basic idea is to consider each candidate composition plan $\mathtt{p}$, check if $\mathtt{p} \equiv \mathtt{Q}$ when there exists no service atom in $\mathtt{p}$. If there exists services and there exists data constraint atoms $\mathtt{C}$ and synchronization constraint atoms $\mathtt{sync}$ such that $\mathtt{Q} \wedge \mathtt{C} \equiv \mathtt{Q}$ and they can be used as the additional constraints on service when we instantiate it. The concrete steps for considering each $\mathtt{p}$ are shown in Algorithm 2.

In steps 4 and 20, when we judge the containment relationship between the plan and query, time synchronization containment relationship is checked first.

In step 5, we check if the equivalent composition plan that is more concise than the current plan $\mathtt{p}$ already exists. If it already exists, the current plan is abandoned. In steps 10 and 12, we use the additional data constraints $\mathtt{A}$ to instantiate a service. A method $\mathtt{genInstance(output, dataConstr, timeConstr)}$ is called to determine the input and output parameters of the service operation. In this method, the $\mathtt{output}$ parameter value is taken as the output parameter value of the service operation. In step 10, we take additional data constraints in $\mathtt{A}$ as the input parameter values of the service operation. The time constraints of $\mathtt{Q}$ are taken as the time constraints of the service operation. In this method, we update $\mathtt{A}$ with the unsatisfied data constraints and returned. After the loop 7, all the services in $\mathtt{p}$ are instantiated. If the attributes of all the additional data constraints are also the data attributes of $\Re(\mathtt{g_{sub}})$, it means that all the additional data constraints can be applied on the services, in other words, the services can satisfy the data constraints after instantiation. Otherwise, the services can not satisfy the data constraints and the service composition plan is abandoned.

In step 20, if $\mathtt{Q} \subseteq \mathtt{p}$, $\mathtt{Q} \supseteq \mathtt{p}$ and all atoms of $\mathtt{p}$ are service instances, delete the redundant plan than $\mathtt{p}$ (in other words, the redundant plan $\mathtt{rePlan}$ satisfying the condition that $\mathtt{goalsOfrePlan} \supset \mathtt{goalsOfp}$) from the result set and add $\mathtt{p}$ into equivalent result set.

To search the contained composition plan, if $\mathtt{Q} \supseteq \mathtt{p}$ and all atoms of $\mathtt{p}$ are service instances, add $\mathtt{p}$ into equivalent result set directly. If $\mathtt{Q}$ is not contained in $\mathtt{p}$ and sub-goals of $\mathtt{Q}$ overlap with that of $\mathtt{p}$, and there exist service atoms in $\mathtt{p}$, we should instantiate the services. Check whether all the additional constraints can be applied on the services when instantiating them. If they can't be applied, this means that the services can not satisfy the data constraints after instantiation, in other words, the plan is not executable. We omit the pseudo code of this algorithm for searching contained composition plans due to limited space.

5 Implementation and Evaluation

In this section, we first describe an implementation of our approach. Then we provide a use case and experimental evaluation.

5.1 Implementation

The architecture of our system is shown in Fig. 2. Firstly, relational databases and data stream sources should be registed and managed. When a query is posed, the query rewriter module uses the information from service registry to decide the candidate service composition plan. The service executor module is responsible for invocation and join/compose the service execution results.

Every service is implemented as a Spark Streaming job. The underlying data streams are subscribed by the service using Kafka. And the outputs of a service are published to Kafka, which can be subscribed by later services. For those Web based clients, we expose continuous data service as REST-like API over HTTP protocol based on a Web-based push technology - Sever-Sent Events (SSE) [11]. It allows the service to push query results to clients continuously. The client sends a request to a service and opens a single long-lived HTTP connection. The service then sends data continuously to the client without further action from the client.

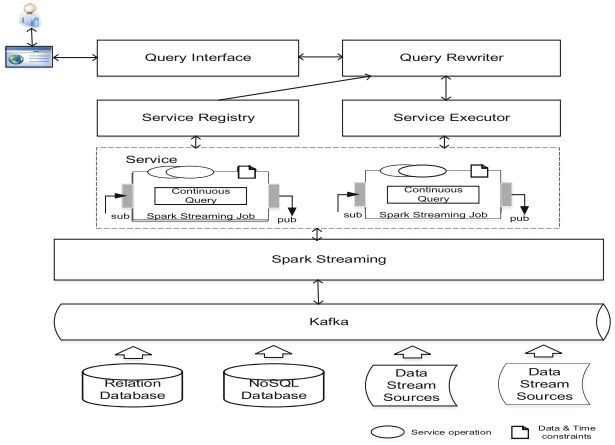


Fig. 2. Architecture of the implementation.

5.2 Case Study

In this section, we take the example introduced in Sect. 2 as the use case to introduce how our approach works.

Assume the outputs of `Ops` of an instance of $\texttt{DS}_1$ are {`mmsi, imo`} and no input parameters. `range` and `slide` are 5 and 1 separately.

This instance of $\texttt{DS}_1$ can be expressed as:

$$\texttt{DS}_1\texttt{inst(mmsi, speed, imo)}\text{:-}\texttt{T(mmsi, long, lat, speed), I(mmsi, imo, callsign, ...), mmsi} > 3000, \texttt{speed} \geq 50, 5, \texttt{sync}_1$$

In a similar way, the instance of DS_2 is:

$DS_2inst(mmsi, draught, speed)$:-$T(mmsi, long, lat, speed), I(mmsi, imo, \ldots, draught), 5, sync_2$

The instance of DS_3 can be expressed as follows:

$DS_3inst(mmsi, speed)$:-$T(mmsi, long, lat, speed), speed < 40, 5, sync_2$

Assume there is no instance for service DS_4, so it is express as:

$DS_4(mmsi, speed, dest, source, long, lat)$:-$TRAVEL(mmsi, dest, source), T(mmsi, long, lat, speed), mmsi < 2000, 5, sync_2$

Query is expressed as Sect. 2. This query has sub-goals $\Re(T)$, I and TRAVEL.

According to our algorithm, the steps to answer user query are as follows:

In the first step the algorithm creates buckets for each sub-goal of Q. The contents of bucket for sub-goal $\Re(T)$ are: DS_1inst,DS_2inst, and DS_4. DS_3inst is not in this bucket because the interpreted predicates of the view and the query are not mutually satisfiable. The contents of bucket for sub-goal TRAVEL are: $DS_4(mmsi, speed, dest, long, \ldots)$. The contents of bucket for sub-goal I are: $DS_2inst(mmsi, draught, speed)$.

In the second step of the algorithm, we combine elements from the buckets. The first combination, involving the first element from each bucket, yields the rewriting

$Q_1(mmsi, draught, speed, dest)$:-$DS_1inst(mmsi, speed, imo'), DS_4(mmsi, speed, dest', long'), DS_2inst(mmsi, draught, speed)$

However, while both DS_1inst and DS_4 are relevant to the query in isolation, their combination is guaranteed to be empty because they cover disjoint sets of vessel identifiers.

Consider the second elements in the left bucket yields the rewriting

$Q_2(mmsi, draught, speed, dest)$:-$DS_2inst(mmsi, draught, speed), DS_4(mmsi, speed, dest', lo\text{-}ng', \ldots), DS_2inst(mmsi, draught, speed)$

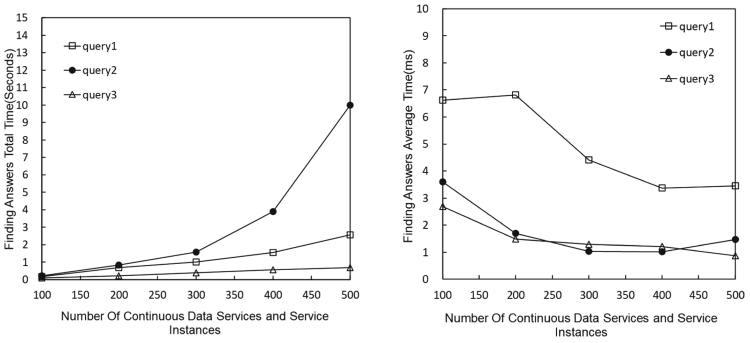
Then we remove the first sub-goal, which is redundant, and generate service instance with the additional data constraints $speed \geq 40$. The output parameters of DS_4 instance operation are set to be variables from attributes of the underlying data stream which are also in the head of Q, which is $mmsi, dest, speed$. The inputs parameters are $speed \geq 40$.

So we would obtain Q_2, which is the only contained composition plan the algorithm finds.

5.3 Experimental Evaluation

In this section, we give an experimental evaluation of our approach. The goal of the experimental evaluation is to analyze the factors that affect the performance of the service composition algorithm.

The service composition algorithm experiments were run on a computer with Intel(R) Core(TM) i5-2400 CPU 3.10 GHz and 8 GB memory. In order to experimentally evaluate our approach, we generated a set of continuous data services and service instances. We use three representative queries including the query example shown in Sect. 2. According to 80/20 rule (also known as Pareto principle), The method guarantees that the number of services and service instances that are related to user queries are about 20% of the total services and service instances generated. For each query, we generated various number of data services and data service instances from 100, 200, ... to 500. Figure 3 plots the total and average time to generate all composition plans for each query against the number of data sources. We can observe that the average generation time per composition plan is within 10ms, which is acceptable in real application.



(a) total time to generate composition plans

(b) average time to generate comosition plans

Fig. 3. Total and average time to generate compostion plans.

6 Related Work

Most of the research work on web service composition focus on traditional *Effect-Providing* services or *application-logic* services instead of *Data-Providing* services or data services. The traditional *application-logic* service composition algorithms are inapplicable and inefficient to data services that all share the same business function (i.e. data query) and have no side-effects [4].

Data integration approach is often adopted for the purpose of data services composition. Some use the query rewriting techniques as the composition algorithm [2–5,12]. Others use visual mashup languages or constructs as composition approach [13,14]. However, the data services model and composition algorithm in these work are inapplicable to data stream sources and data stream integration.

There are some related research work from data integration area such as InfoMaster [15] and Information Manifold [8]. Our work differs with these works in many ways. First, these works target toward resolving specific queries given a set of data sources, whereas in our work the focus is on constructing a composition of services that is independent of a particular input value. The composite service can be reused to answer a set of queries instead of a specific queries. Second, compared to previous query rewriting algorithms [10,16] that were proposed for the traditional static relational data model, our composition algorithm is based on data stream model. As far as we know, our continuous data service model is the first service model to support data stream query and our algorithm is the first to address the problem of composing continuous data services to support data stream integration.

There are some related research work on service modeling for data streams such as [17,18], however, the work cannot be used to solve the problem of query across various data sources directly. Some work has addressed the problem of supporting views in data stream management systems [7], however, the work is limited only to answering specific queries based on a set of data sources. Our work propose a continuous data service model which provides a flexible, controlled and standardized approach to access or query data stream. We address data stream integration problem by providing service composition approach. The composite service can access a set of conditions as input instead of limiting to answering specific queries.

7 Conclusion

In this paper, we presented an approach for conjunctive query on data streams by composing continuous data services. We introduce a flexible continuous data service model with continuous query as service operation. Service operation instance is modeled as a view defined on data streams in which the data part and time synchronization part are separated from each other. A continuous data service composition algorithm is introduced for answering queries across data streams. An experimental study is provided to evaluate the performance of our approach. As a future work, we plan to address location concerns when composing continuous data services.

Acknowledgments. This work is supported by Beijing Natural Science Foundation No. 4172018, National Natural Science Foundation of China No. 61672042, and University Cooperation Projects Foundation of CETC Ocean Corp.

References

1. Carey, M.J., Onose, N., Petropoulos, M.: Data services. Commun. ACM **55**(6), 86–97 (2012)
2. Vaculín, R., Chen, H., Neruda, R., Sycara, K.: Modeling and discovery of data providing services. In: 2008 IEEE International Conference on Web Services, pp. 54–61, September 2008

3. Barhamgi, M., Benslimane, D., Ouksel, A.M.: Composing and optimizing data providing web services. In: Proceedings of the 17th International Conference on World Wide Web, pp. 1141–1142. ACM (2008)
4. Barhamgi, M., Benslimane, D., Medjahed, B.: A query rewriting approach for web service composition. IEEE Trans. Serv. Comput. **3**(3), 206–222 (2010)
5. Zhou, L., Chen, H., Yu, T., Ma, J., Wu, Z.: Ontology-based scientific data service composition: a query rewriting-based approach. In: AAAI Spring Symposium: Semantic Scientific Knowledge Integration, pp. 116–121 (2008)
6. Zhang, F., Wang, G., Han, Y.: Automatic generation of service composition plans for correlated queries. In: 2013 10th Web Information System and Application Conference, pp. 143–149, November 2013
7. Ghanem, T.M., Elmagarmid, A.K., Larson, P.Å., Aref, W.G.: Supporting views in data stream management systems. ACM Trans. Database Syst. **35**(1), 1–47
8. Levy, A.Y., Rajaraman, A., Ordille, J.J.: The world wide web as a collection of views: query processing in the information manifold. In: VIEWS, pp. 43–55 (1996)
9. Doan, A., Halevy, A., Ives, Z.: Principles of Data Integration, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2012)
10. Pottinger, R., Halevy, A.: MiniCon: a scalable algorithm for answering queries using views. Int. J. Very Large Data Bases **10**(2–3), 182–198 (2001)
11. Hickson, I.: Server-sent events. https://www.w3.org/TR/eventsource/. Accessed 25 October 2015
12. Zhao, W., Liu, C., Chen, J.: Automatic composition of information-providing web services based on query rewriting. Sci. China Inf. Sci. **55**(11), 2428–2444 (2012)
13. Wang, G., Yang, S., Han, Y.: Mashroom: end-user mashup programming using nested tables. In: Proceedings of the 18th International Conference on World Wide Web, pp. 861–870. ACM (2009)
14. Han, Y., Wang, G., Ji, G., Zhang, P.: Situational data integration with data services and nested table. Serv. Oriented Comput. Appl. **7**(2), 129–150 (2013)
15. Genesereth, M.R., Keller, A.M., Duschka, O.M.: Infomaster: an information integration system. SIGMOD Rec. **26**(2), 539–542 (1997)
16. Levy, A.Y., Rajaraman, A., Ordille, J.J.: Querying heterogeneous information sources using source descriptions. In: Proceedings of the 22th International Conference on Very Large Data Bases. In: VLDB 1996, pp. 251–262. Morgan Kaufmann Publishers Inc., San Francisco (1996)
17. Han, Y., Liu, C., Su, S., Zhu, M., Zhang, Z., Zhang, S.: A proactive service model facilitating stream data fusion and correlation. Int. J. Web Serv. Res. (IJWSR) **14**(3), 1–16 (2017)
18. Gil, D., Ferrández, A., Mora-Mora, H., Peral, J.: Internet of things: a review of surveys based on context aware intelligent services. Sensors **16**(7), 1069 (2016)