Capítulo 4

Implementación del evaluador de consultas híbridas

Este capitulo presenta la implementación de nuestro modelo de evaluación de consultas hibridas. La implementación consiste en un evaluador de consultas que ejecuta los operadores de ventana espacial y temporal (en forma de consultas definidas) sobre un *stream* o un *LDS* de entrada. El evaluador se implementó bajo la plataforma Java, por lo tanto se presentarán las clases que lo conforman.

El resto del capítulo tiene la siguiente organización. La sección 4.1 presenta las clases que implementan el modelo de datos continuos. En la sección 4.2 y 4.3 se describen las clases que implementan los operadores de ventana temporal y espacial respectivamente. En la sección 4.4 se presenta la arquitectura del evaluador de consultas. Finalmente, la sección 4.5 presenta las conclusiones sobre nuestra implementación.

4.1 Modelo de datos continuos

La implementación de nuestro modelo de datos esta dado por las clases que se presentan el diagrama de clases de la figura 4.1. Una breve descripción de las clases es la siguiente:

- La clase `tuple` representa a las tuplas que forman una relación.
- La clase `tuple_stream` corresponde a las tuplas estampilladas que forman un *stream*
- La clase `tuple_LDS` representa las tuplas dependientes de la localización de un *LDS*.
- La clase `Timestamp` corresponde a la estampilla de tiempo que compone a las tuplas de un *stream*.
- La clase `Position` representa una coordenada geográfica que compone a las tuplas dependientes de la localización.

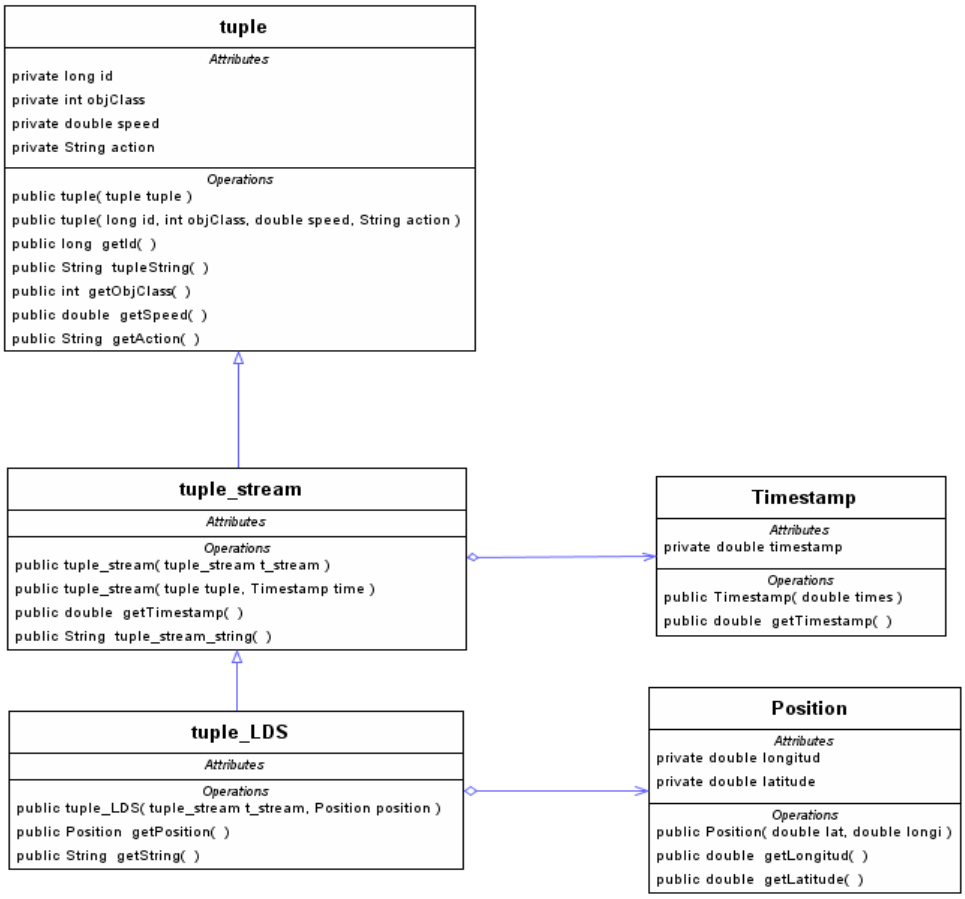

Figura 4.1. Clases del modelo de datos

4.1.1 Relación

Una relación es un conjunto de instancias de la clase `tuple` donde el esquema esta dado por sus atributos. Por lo tanto el esquema de tupla que proponemos es el siguiente:

```
tuple(id:long, objClass:int, speed:double, action:class)
```

Donde: `id` corresponde al identificador de un objeto móvil, `objClass` representa a la clase o tipo del objeto móvil, `speed` es la velocidad del objeto y `action` es la acción que realiza el objeto por ejemplo movimiento, parar, etc.

4.1.2 *Stream*

Un *stream* es secuencia de instancias de la clase `tuple_stream`. La clase `tuple_stream`, además de ser una especialización de la clase `tuple`, esta compuesta por un atributo de la clase `Timestamp` que representa a la estampilla de tiempo en que la tupla fue producida. Por lo tanto `Timestamp` solo contiene un atributo bajo el dominio `double` que representa el dominio de tiempo en forma discreta.

Cabe destacar que asumimos que los *streams* de entrada cumplen con las propiedades descritas en nuestro modelo. Es decir, que la secuencia de tupla viene con orden parcial en base a su estampilla de tiempo.

4.1.3 *Stream* dependiente de la localización (*LDS*)

El LDS es implementado como una secuencia de instancias de la clase `tuple_LDS`. La clase `tuple_LDS`, además de contar con la estampilla de tiempo de la clase `Timestamp`, cuenta con un atributo de la clase `Position`. La clase `Position`

representa el dominio espacial punto descrito en nuestro modelo y sus atributos son son: `latitude` y `longitude`.

4.2 Clase ventana temporal

La clase ventana temporal (`TemporalWindow`) es una clase abstracta que contiene los atributos y métodos comunes para los diferentes tipos de ventanas temporales definidas en nuestro modelo (ver Figura 4.2).

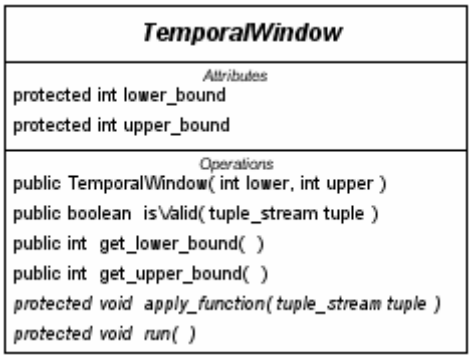

Figura 4.2. Clase `TemporalWindow`.

Atributos

- **`lower_bound`** es el límite inferior a partir del cual se va a ejecutar
- **`upper_bound`** es el límite superior hasta el cual se va a ejecutar.

Constructor

```
public TemporalWindow(int lower, int upper)
```

Inicializa los límites superior e inferior a partir de los parámetros.

Métodos principales

```
public boolean isValid(tuple_stream tuple)
```

Valida si la tupla `tuple` está dentro del intervalo definido por los límites.

```
protected abstract void apply_function (tuple_stream tuple)
```

Define la función de actualización temporal correspondiente a la semántica de cada operador de ventana temporal.

```
protected  abstract void run()
```

Define la ejecución propia de cada tipo de ventana espacial.

4.2.1 Clase *snapshot* temporal

La clase snapshot temporal (`SnapshotTemporal`) hereda de `TemporalWindow`. Implementa el operador de ventana temporal con intervalo de consumo fijo (ver figura 4.3).

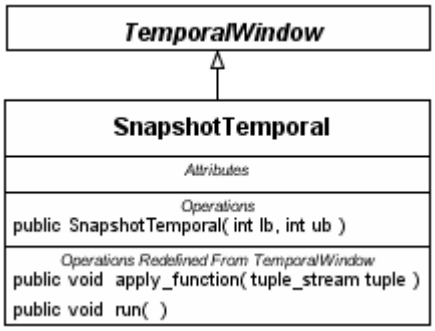

Figura 4.3. Clase `SnapshotTemporal`.

Constructor

`public SnapshotTemporal(int lb, int ub)`

Construye una ventana snapshot, con límites especificados mediante los parámetros.

Métodos principales

`public void run()`

Este método define su ejecución. Su ejecución consiste en leer el stream de entrada y validar si tuplas están dentro del intervalo fijo definido reporta la tupla. Su implementación es la siguiente:

```
tuple_stream  tuple= read_input_tuple();
if(isValid(tuple))
   write_output(tuple);
```

4.2.2 Clase *landmark* temporal

Es una subclase de `TemporalWindow` e implementa el operador temporal landmark temporal. Esta ventana se ejecuta continuamente sobre un stream, actualizando

únicamente su límite superior y almacenando los resultados de todas sus ejecuciones (ver figura 4.3).

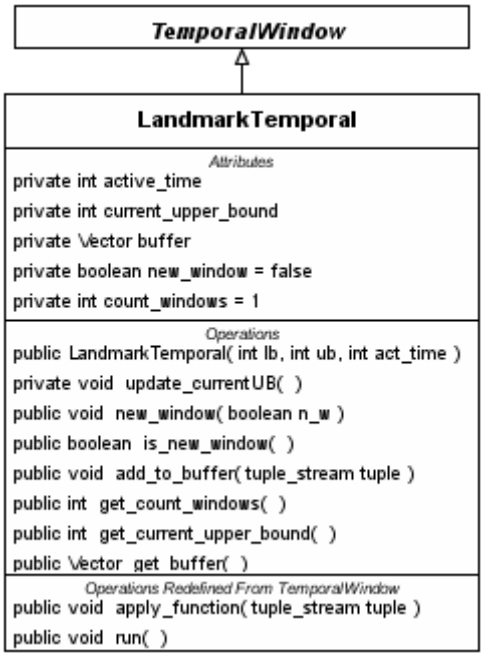

Figura 4.3. Clase `LandmarkTemporal`.

Atributos

- `active_time` es el tiempo debe trascurrir para generar una nueva ventana
- `current_upper_bound` es el límite superior dinámico actualizado en la generación de una nueva ventana.
- `buffer` es un vector que almacena las tuplas producidas desde el inicio de la ejecución.
- `new_window` es una bandera que indica si genera una nueva ventana.
- `count_window` es el contador de ventanas generadas.

Constructor

```
public LandmarkTemporal (int lb, int ub, int act_time)
```

Crea una nueva `LandmarkTemporal` cuyos límites y tiempo de activación son especificados mediante los parámetros.

Métodos principales

```
private void update_currentUB()
```

Actualiza el límite superior a partir de `active_time`.

`public void new_window(boolean n_w)`

Establece si se ha creado una nueva ventana.

`public void apply_function (tuple_stream tuple)`

Implementa la función de actualización que consiste en detectar a partir de `tuple` si se debe crear una nueva ventana y por lo tanto actualizar el límite superior, es decir incrementar el intervalo de consumo. Su implementación es la siguiente:

```
new_window(false);
if (tuple.getTimestamp() > current_upper_bound){
    new_window(true);
    update_currentUB();}
```

`public void run()`

La ejecución consiste en leer el stream de entrada, procesar las tuplas que sean válidas en el intervalo actual definido, aplicar la función de actualización para determinar en que momento se genera una nueva ventana y enviar el resultado. El resultado además de la tupla actual debe incluye el buffer acumulado. Su implementación es la siguiente:

```
if(isValid(tuple)){
   apply_function(tuple);
   add_to_buffer(tuple);
   if(is_new_window()){
       notifyNewWindow();
       for(int i=0;i<buffer.size();i++)
         write_output(buffer.get(i)); }
   write_output(tuple);}
```

4.2.3 Clase *sliding disjoint* temporal

La clase *sliding disjoint* temporal (`SlidingDisjointTemporal`)(ver figura 4.5) implementa una ventana temporal cuyo límites cambia en el tiempo sin sobre ponerse entre ellos es decir la ventana se desliza sobre la línea de tiempo durante su ejecución.

Atributos

- **`sliding`** define el tiempo de activación y la longitud de las ventanas generadas.

- **`current_upper_bound` y `current_lower_bound`** son los límites superior e inferior actuales respectivamente que se actualizan cuando se genera una nueva ventana.

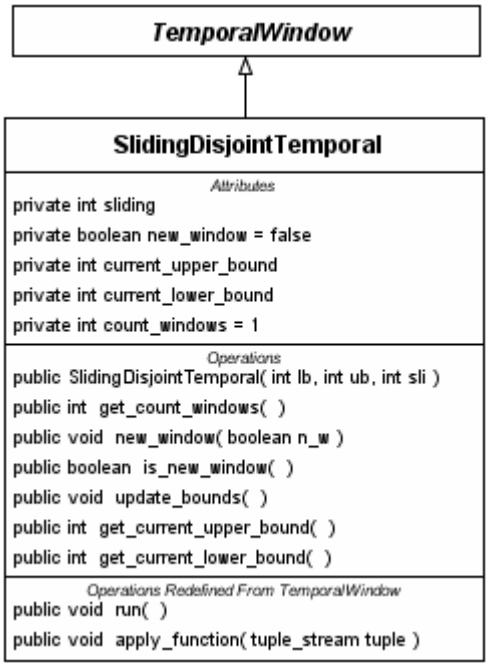

Figura 4.5. Clase `SlidingDisjointTemporal`.

Constructor

`public SlidingDisjointTemporal (int lb, int ub, int sli)`

Construye una nueva `SlidingDisjointTemporal` cuyos límites y deslizamiento son especificados mediante los parámetros.

Métodos principales

`public void update_bounds()`

Actualiza ambos límites de ejecución, por lo tanto define un nuevo intervalo.

`public void apply_function (tuple_stream tuple)`

Implementa de manera similar la función de actualización que `LandmarkTemporal` con la diferencia que para este operador se actualizan ambos límites mediante `update_bounds()`.

`public void run()`

Su ejecución es similar a la de la clase `LandmarkTemporal`, la diferencia radica en que solo envía las tuplas actuales, es decir no envía resultados previos. Por lo tanto su implementación es la siguiente:

```
if(isValid(tuple)){
   apply_function(tuple);
   if(is_new_window()){
       notifyNewWindow();
   write_output(tuple);}
```

4.2.4 Clase *sliding overlap* temporal

La clase `SlidingOverlapTemporal` (ver Figura 4.6) es una subclase de `TemporalWindow`, la cual ejecuta una serie de ventanas cuyos límites se translapan entre sí, por lo tanto múltiples ventanas se ejecuten de manera concurrente. Simula la ejecución de múltiples ventanas de manera concurrentes almacenando sus límites en vectores que trabajan como colas

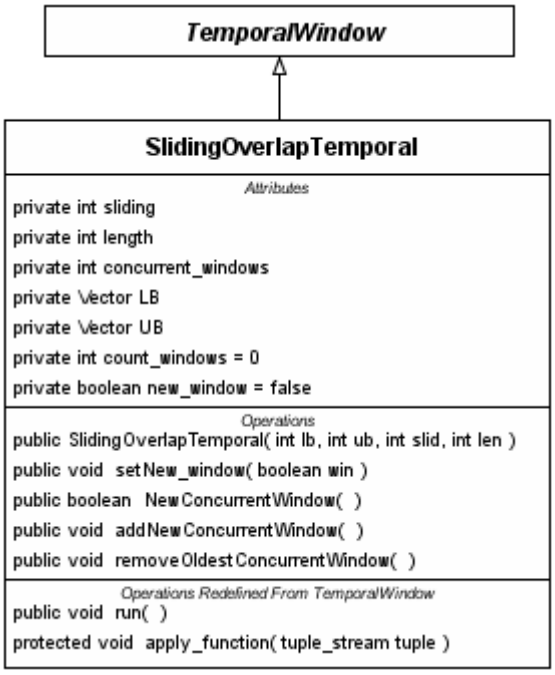

Figura 4.6. Clase `SlidingOverlap`.

Atributos

- **`sliding`** es el deslizamiento de la ventana,
- **`length`** es la longitud de las ventanas generadas,
- **`concurrent_windows`** es el número de ventanas concurrentes que se tienen durante la ejecución
- **LB y UB** son vectores que almacenan respectivamente los límites inferior y superior de las ventanas que se están ejecutando concurrentemente.

Constructor

`public SlidingOverlapTemporal (int lb, int ub, int slid, int len)`

Construye una SlidindOverlapTemporal cuyo límites de ejecución, deslizamiento y longitud de ventana son especificados como parámetros.

Métodos importantes

`public boolean NewConcurrentWindow()`

Indica si se ha generado una nueva ventana concurrente.

`public void addNewConcurrentWindow()`

Agrega una nueva ventana concurrente, es decir agrega nuevos límites a final de los vectores LB y UB.

`public void removeOldestConcurrentWindow()`

Remueve la ventana concurrente más antigua, es decir remueve los límites al principio de los vectores LB y UB.

`protected void apply_function(tuple_stream tuple)`

Básicamente se centra en la administración de los vectores que representan las ventanas en ejecución concurrente. Este método detecta cuando una tupla sobrepasa los límites de la ventana más antigua, entonces esta es eliminada. En caso contrario detecta si dicha tupla genera una nueva ventana cuyos límites se sobrepongan con otras que hay en ejecución y entonces da de alta una nueva. Su implementación es la siguiente:

```
setNew_window(false);
if (isValid(tuple)){
    if(tuple.getTimestamp() >= LB.get(0) &&
       tuple.getTimestamp() <=(UB.get(0))){
          if(tuple.getTimestamp() >=
            (((LB.get(concurrent_windows)))+sliding)
             addNewConcurrentWindow();}
          else
             removeOldestConcurrentWindow();}}
```

`protected void run()`

Implementa la ejecución de esta ventana. Lee las tuplas de entrada y notifica cuando se genera alguna nueva ventana, además envía las tuplas que corresponde a todas las ventanas concurrentes que se tengan. La codificación es la siguiente:

```
tuple_stream  tuple= read_input_tuple();
if(apply(tuple) != null){
     if(is_new_window())
          notify_new_window();
     write_to_each_window(tuple);}}
```

4.3 Clase ventana espacial

La clase ventana especial (`SpatialWindow`) es la superclase que define los atributos y métodos básicos usador por todos los tipos de ventana espacial definidos en el modelo (Ver figura 4.6).

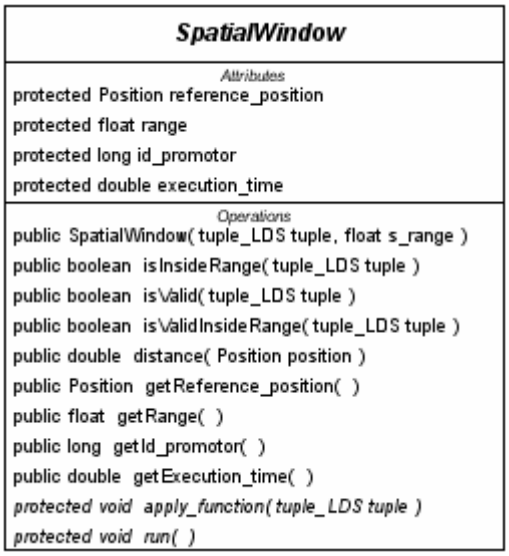

Figura 4.7. Clase `SpatialWindow`.

Atributos

- **`reference_position`** es la posición (coordenada) a partir del cual se define la ventana.
- **`range`** es el rango de alcance de la ventana.
- **`id_promotor`** corresponde al identificador del objeto móvil que ejecuta la ventana.

- **excecution_time** representa el tiempo en que se ejecutó la ventana.

Constructor

```
public SpatialWindow(tuple_LDS tuple, float s_range){
```

Recibe como entrada el rango de ventana (`s_range`) y un objeto `tuple_LDS` a partir del cual se incializan `referente_position`, `id_promotor` y `execution_time`.

Métodos principales

```
public boolean isValid(tuple_LDS tuple)
```

Verifica que una `tuple_LDS` sea valida temporalmente, es decir que su estampilla sea igual al tiempo de ejecución de la ventana.

```
public boolean isInsideRange(tuple_LDS tuple)
```

Determina si una `tuple_LDS` esta dentro del rango especial definido por la ventana

```
public double distance(Position position)
```

Obtiene la distancia entre `position` y `position_reference`. Esta es calculada usando la formula de Haversine [USCB].

```
protected abstract void apply_function (tuple_LDS tuple)
```

Debe implementar la función de actualización espacial propia de cada tipo de ventana.

```
protected  abstract void run();
```

Debe ser implementado para definir la ejecución de cada tipo de ventana espacial

4.3.1 Clase *snapshot* espacial

La clase snapshot espacial (`SnapshotSpatial`), presentada en la figura 4.8, carece de atributo extras a los heredados por la superclase `SpatialWindow`. Implementa el mismo constructor que su superclase y debido a que no tiene función de actualización el método `apply_function()` esta vacío.

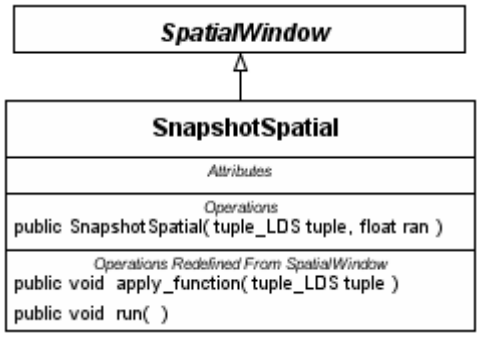

Figura 4.8 Clase `SnapshotSpatial`.

Principales métodos

`public void run ()`

Su ejecución consiste en leer las tuplas del LDS de entrada, validar las que estén dentro del espacio y tiempo definidos y enviar resultado. Su implementación es la siguiente:

```
tuple tuple= read_input_tuple();
if (isValid())
   if (isInsideRange(tuple))
       write_output(tuple);
```

4.3.2 Clase ventana espacial dinámica

La clase espacial dinámica (`DynamicSpatialWindow`) es una especialización de `SpatialWindow`. Esta clase abstracta define un conjunto de atributos y métodos para implementar una ventana espacial que evoluciona, es decir una ventana que actualiza su rango de consumo espacial a medida que se recorre cierta distancia.

Atributos

- **`active_distance`** es la distancia que el id_promotor debe recorrer para volver a ejecutar la ventana.
- **`active_window`** indica si la ventana esta activa o no.
- **`cont_windows`** indica las veces que ha sido activada la ventana.
- **`current_time`** es el tiempo actual en que se esta ejecutando la ventana, este es determinado a partir del *LDS* que se procesa.

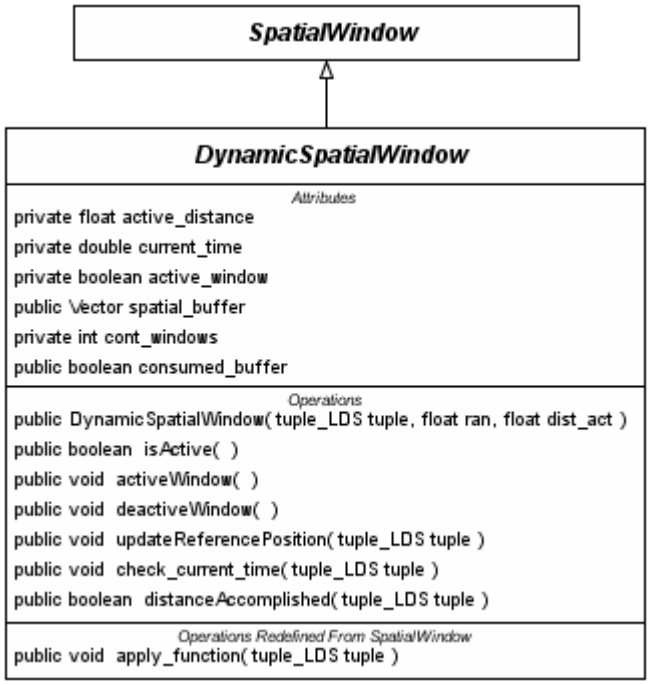

Figura 4.8. Clase `DynamicSpatialWindow.`

- **`spatial_buffer`** almacena las tuplas cuya estampilla de tiempo sea igual a `current_time,` por lo tanto es actualizado en base al tiempo de ejecución. Es utilizado con el objetivo de no perder tuplas en el procesamiento.

Principales métodos

`public boolean isActive()`

Usado para saber si la ventana esta activa en un momento dado.

`public void activeWindow()`

Activa la ventana una vez que el se haya recorrido la distancia de activación

`public void deactiveWindow()`

Este método desactiva la ventana, es decir id_promotor no ha recorrido la distancia de activación.

`public void check_current_time(tuple_LDS tuple)`

Checa si se debe actualizar el tiempo actual de ejecución (`current_time`) a partir de las tuplas de entrada, por lo tanto también actualiza `spatial_buffer`.

`public boolean distanceAccomplished(tuple_LDS tuple)`

Verifica si `id_promotor` ha recorrido la distancia de activación.

`public void apply_function(tuple_LDS tuple)`

Implementa la función de actualización que debe se aplicada a cada tupla para determinar si se ha recorrido la distancia de activación para entonces activar la ventana y actualizar la posición de referencia. Su implementación es la siguiente:

```
check_current_time(tuple);
spatial_buffer.add(tuple);
if (distanceAccomplished(tuple)){
   updateReferencePosition(tuple);
   activeWindow();}
```

4.3.3 Clase *landmark* espacial

La clase *landmark* espacial (`LandmarkSpatial`) es una especialización de la clase `DynamicSpatialWindow`. Por lo tanto, es una ventana dinámica que se activa cuando la distancia de activación es recorrida almacenando los resultados desde su primera ejecución (Ver figura 4.9).

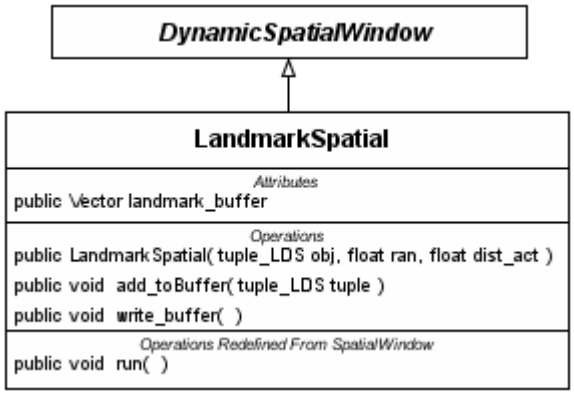

Figura 4.9. Clase `LandmarSpatial`.

Atributos

- **`landmark_buffer`** almacena las tuplas resultantes en cada activación.

Constructor

`public LandmarkSpatial (tuple_LDS tuple, float ran, float dist_act)`

Crea una ventana `LandmarkSpatial` a partir de sus parámetros con estado activo.

Métodos principales

`public void run ()`

Su ejecución consiste en leer las tuplas del *LDS* y si la ventana esta activa verificar si están dentro del rango espacial actual (agregando cada tupla a su *buffer*). Si la ventana no esta activa entonces aplica la función de actualización para verificar si `id_promotor` ha recorrido la distancia de activación y entonces activar de nuevo la ventana.

```
tuple tuple= read_input_tuple();
  if (isActive()){
     if (isValid(tuple)){
        if(isInsideRange(tuple)){
           add_toBuffer(tuple);
           write_output(tuple);}}
     else{
        send_to_view_query(getQuery(landmarkSpatial));
        for(int i=0;i<spatial_buffer.size();i++)
            if(isInsideRange(spatial_buffer.get(i))){
                add_toBuffer(spatial_buffer.get(i));
                write_output(spatial_buffer.get(i));}
        deactiveWindow();
        write_output_buffer()}}
  else
    apply_function(tuple);
```

4.3.4 Clase *sliding disjoint* espacial

La clase *sliding disjoint* espacial (`SlidingDisjointSpatial`) hereda de `DynamicSpatialWindow` (Ver figura 4.10). Es una ventana dinámica que se activa a partir de la distancia de activación consumiendo áreas disjuntas.

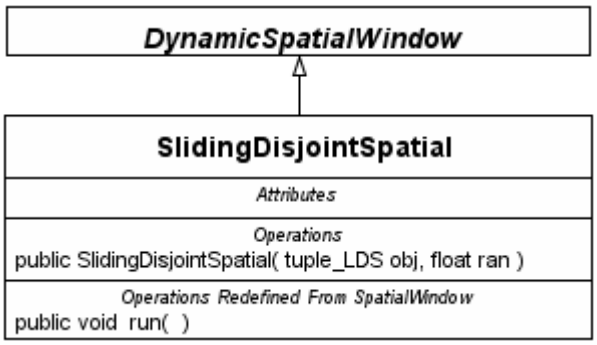

Figura 4.10. Clase `SlidingDisjointSpatial`.

Constructor

```
public SlidingDisjointSpatial (tuple_LDS obj, float ran)
```

Crea una ventana SlidingDisjointSpatial, es decir una ventana dinámica con distancia de activación = 2*ran.

Métodos principales

```
public void run ()
```

La ejecución de este operador es similar a la de `LandmarkSpatial`, excepto que este no agrega sus resultados al buffer.

4.3.5 Clase *sliding overlap* espacial

La clase *sliding overlap* espacial (`SlidingOverlapSpatial`) es una subclase de `DynamicSpatialWindow` (Ver figura 4.11). Es una ventana dinámica que se activa a partir de la distancia de activación consumiendo áreas que se traslapan entre sí.

Constructor

```
public SlidingOverlapSpatial (tuple_LDS obj, float ran,
float dist_act)
```

Construye una ventana **`SlidingOverlapSpatial`**, es decir una ventana dinámica que va a consumir áreas que se traslapan entre sí.

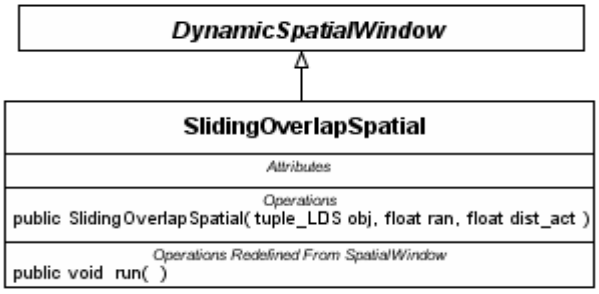

Figura 4.11. Clase SlidingOverlapSpatial

Métodos principales

```
public void run ()
```

La ejecución de este operador es idéntica a la de `SlidingDisjointSpatial`, la diferencia radica en sus distancias de activación.

4.4 Arquitectura

El evaluador tiene como objetivo procesar una consulta, a través de la ejecución de los operadores de ventana (espacial y temporal) descritos, sobre un flujo de datos (*stream* ó *LDS*) de entrada. La arquitectura de nuestro evaluador se ilustra en la figura 4.11. Sus componentes son los siguientes:

- *Stream reader* es la interfaz de entrada para las fuentes de datos, es decir es el encargado de establecer la conexión con las fuentes de datos y recibir el flujo de datos. El flujo de datos es una secuencia de instancia de la clase `tuple_stream` para un *stream* ó instancias de la clase `tuple_LDS` para una *LDS*
- *Query Listener* es la interfaz de entrada para el usuario, es decir es el encargado de recibir las consultas a ejecutar. Por simplicidad las consultas a ejecutar ya han sido preestablecidas.
- *Operador selector* es el encargado de seleccionar el operador a ejecutar en base a la consulta de entrada. Este contiene una lista con los operadores disponibles (operadores de ventana espacial y temporal).
- *Operator ejecutor* ejecuta el operador seleccionado sobre el stream de entrada y envía los resultados obtenidos. Es componente contiene la implementación de los operadores, es decir las clases descritas correspondientes a cada tipo de ventana.

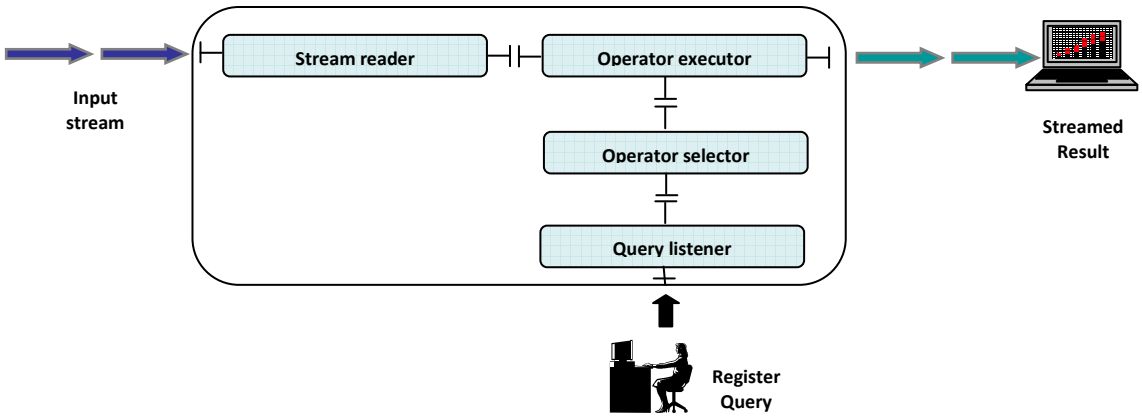

Figura 4.11. Arquitectura del evaluador de consultas.

El evaluador a través de *stream reader* acepta conexiones de fuentes de datos (clientes) y recibe sus flujos de datos producidos (*streams* o *LDSs*). Por otro lado, cuando un usuario, a través de la interfaz gráfica (*GUI*), somete consultas sobre el flujo de datos la consulta es enviada directamente al *query_listener*. El *query_listener* envía la consulta al *operador_selector* con el propósito de que este seleccione el operador adecuando para dicha consulta. *Operator_selector* informa a *operador_excecutor* el operador que debe ejecutar, este lo ejecuta directamente sobre el estado actual del flujo de datos que *stream reader* esta recibiendo de la fuente. Finalmente al aplicar el operador se envían los resultados y se muestran en la interfaz gráfica. Este funcionamiento se ilustra en la figura 4.12.

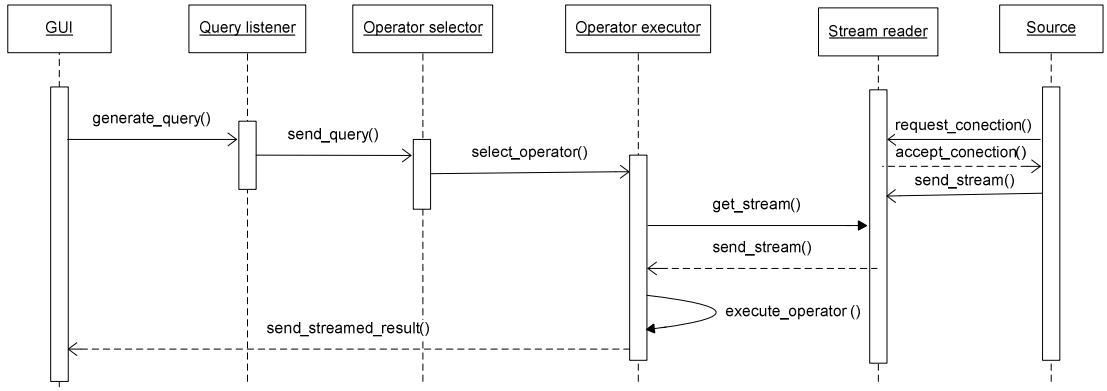

Figura 4.12 Diagrama de secuencia del funcionamiento del evaluador.

4.5 Discusión

En este capitulo se ha descrito la implementación de nuestro modelo de evaluación como clases JAVA. Se han descrito las clases que implementan a nuestro modelo de datos espacio-temporales. También se han descrito las clases que implementa a los operadores de ventana, se han descrito sus atributos y métodos más importantes (como constructores y sobre todo las funciones de actualización temporal y espacial). También se ha descrito una arquitectura para la evaluación de consultas la cual utiliza los operadores implementados. Cabe mencionar que es una primera versión de la arquitectura, ya que lo mas importante es la definición de lo operadores ya que pueden trabajar con cualquier arquitectura definida.