Fish4Knowledge Deliverable D5.2

RDF/RDMS Datastore Definition

Principal Author:	S. Palazzo, C. Spampinato, J.R. van Ossenbruggen
Contributors:	CWI, UCAT
Dissemination:	PU

Abstract: The RDF/RDMS Datastore Definition deliverable (D.5.2) of the Fish4Knowledge project aims at describing the design and implementation of a datastore schema for storing information on 1) the underwater monitoring system and on 2) the processing results in terms of fish detection, fish tracking, fish recognition, event detection and recognition. Components developed by the different partners are envisioned to inter-operate mainly by reading and writing data to a relational database conforming to this schema. In addition, an RDF schema has been defined in order to expose the project data in a Linked Data-compliant solution for Web-scale sharing of resources and experimental data as proposed in WP5.

Deliverable due: Month 3

1 Introduction

This document aims at describing the design and implementation of a datastore schema for storing information on:

- *Underwater Monitoring*. This data mainly concerns with the sites, cameras and recorded videos of the monitoring system;
- *Object Detection and Recognition*. This section describes the data related to fish detection, tracking and recognition. It also allows the storage of events that involve groups of fish, fish-fish and fish-background interactions.
- *Low Level Features*. This part aims at describing the schema for low level feature storage for supporting the description of fish, event and scene of interest.
- *Software Components*. In order to deal with different versions of the algorithms, this part of the schema aims at keeping track of which algorithm has been used for extracting specific information.

Two schema have been defined: a relational one and another one using RDF. Since the data has a clear relational structure, with little or no data integration or data heterogeneity problems, a relational approach has been chosen as the project's primary storage. Components developed by the different partners are envisioned to inter-operate mainly by reading and writing data to a relational database conforming to this schema. In addition, an RDF schema has been defined in order to expose the project data in a Linked Data-compliant solution for Web-scale sharing of resources and experimental data as proposed in WP5. This document focuses on the description of the two solutions and also on reusable components for datastore access. The relational datastore schema is described in Section 2, whose subsections discuss the single entities. The following section describes the RDF export. For each developed schema, the server name and how to access to stored data is also given.

2 Relational Datastore Schema

The huge amount of information extracted from the videos is stored in a relational database which is specifically designed to make it easy to retrieve the data typically needed to answer queries by marine biologists. The datastore schema is described by using the Entity/Relationship model. Since this model for the entire datastore would be too big to fit one page, we split it into four parts described in the following sections. Fig. 1 shows the formalism used to draw the Entity/Relationship model.

2.1 Underwater Monitoring System

The entities used for describing the underwater monitoring system, i.e. the information regarding the sites where cameras are located and the recorded videos, are: *cameras* and *videos*. The ERM schema is shown in Fig. 2.

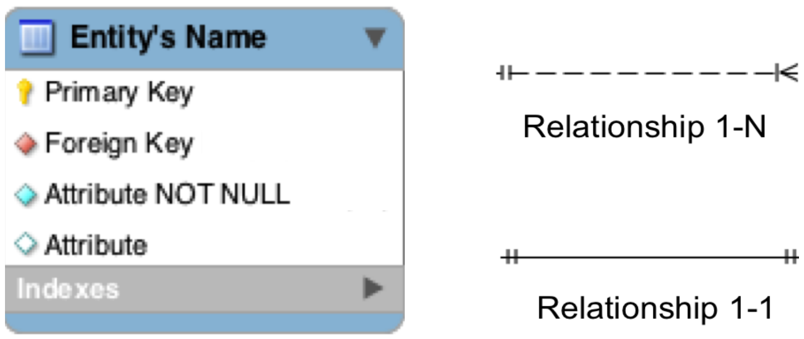


Figure 1: Entity/Relationship Formalism

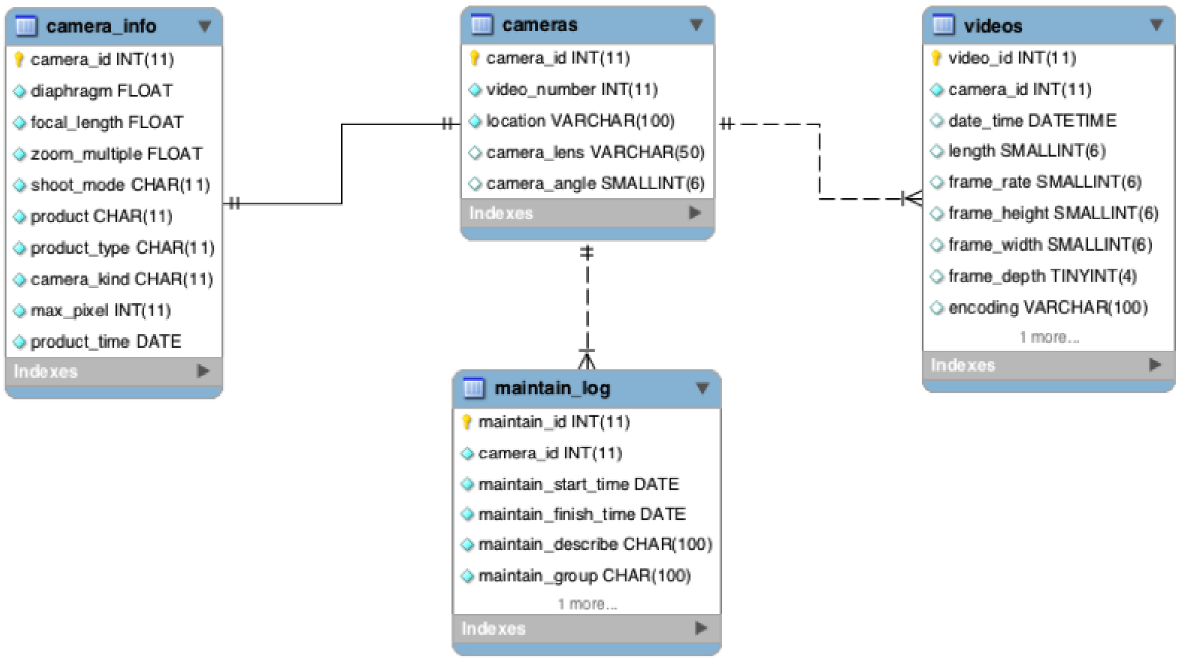


Figure 2: ERM schema for Underwater Monitoring System Information

2.1.1 Cameras

ERM Entity: (*camera*)

This entity models information about a camera, such as its location, lens, width angle. Each camera entity is identified by the name of the site (e.g. NPP-3, LanYu, etc) and by the video number (whose maximum value depends on the number of cameras available on the site).

Attributes:

- *camera_id* **(integer)**: unique identifier for a camera.
- *video_number* **(integer)**: in each location there are more than one camera. This field describes the number of camera for the specific location.
- *location* **(string)**: short name describing the geographic location of the camera.
- *camera_lens* **(string):** lens used by the camera.
- *camera_angle* **(integer)**: width angle of the camera in degrees.

ERM Entity: (*camera_info*)

This entity is meant to provide information on the cameras used for video recording. It refers to the *camera's* entity and provides the hardware specifications of the the used cameras.

Attributes:

- *camera_id* **(integer)**: foreign key to the *camera* entity.
- *diaphragm* **(float)**: diaphragm of the used camera.
- *focal_length* **(float)**: focal length of the camera.
- *zoom_multiple* **(float):** type of zoom of the camera.
- *shoot_mode* **(string)**: how the camera grabs the frames.
- *product* **(string)**: name of the used camera.
- *product_type* **(string)**: type of product.
- *camera_kind* **(string)**: the type of camera, e.g. digital camera, reflex camera. etc.
- *max_pixel* **(integer)**: spatial resolution in mega-pixel.
- *product_time* **(date)**.

ERM Entity: (*maintain_log*)
This entity describes those cameras that undergo maintenance. It describes when the maintenance starts and ends and the description of the intervention.

Attributes:

- *maintain_id* (**integer**): the primary key of the maintenance intervention.
- *camera_id* (**integer**): foreign key to the *camera* entity.
- *maintain_start_time* (**date**): Start time of the intervention.
- *maintain_finish_time* (**date):** End time of the intervention.
- *maintain_describe* (**string**): Description of the intervention.
- *maintain_group* (**string**).

2.1.2 Recorded Videos

ERM Entity: (*video*)

This entity represents the processed videos among all the available ones. It contains generic and technical information on the videos (capture date and time, length, encoding, etc). In detail, each video entity is associated to the camera from which it was captured, and contains information on the date and time at which the video starts and technical data (frame rate, size, codec, etc).

Attributes:

- *video_id* (**integer**): unique identifier for the video.
- *camera_id* (**integer**): foreign key to the *camera* entity, specifies by which this video was created.
- *date_time* (**date and time**): date and time at which the video begins.
- *length* (**integer**) : length in seconds of the video.
- *frame_rate* (**integer**): frame rate in frames/second.
- *frame_height* (**integer**): frame height in pixels.
- *frame_width* (**integer**): frame width in pixels.
- *frame_depth* (**integer**): bit width of the pixels.
- *encoding* (**string**): encoding algorithm of the video.

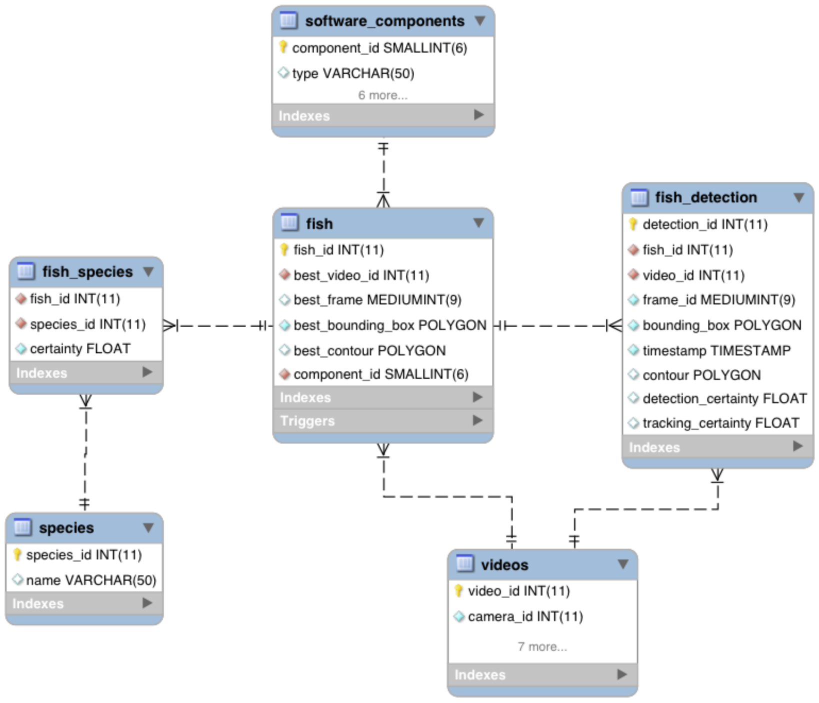


Figure 3: ERM schema for Fish Detection and Recognition

2.2 Object Detection and Recognition

The entities of this section are related to the results of video processing in terms of detection of fish and non-fish objects (secondary objects). Moreover, events that involve more fish, groups of fish, fish-background interactions and, more in general, fish non-fish objects interaction are also stored. The schema for handling fish detection, tracking and recognition is shown in fig. 3.

As mentioned before, secondary objects are non-fish elements (such as plants or other animals) which interact with fish in the context of an event (for example, feeding). These objects are stored only if an event is being detected and the database representation is the same as the one used for fish (with *secondary_objects* and *secondary_object_detection* tables). An event is the recognition of a particular behavior of a fish or group of fish, such as feeding, preying, mating, schooling. The information associated to an event is its type, the frame range in the video and the detections of both fish and secondary objects involved in the event (through the event_fish_detection and the event_secondary_object_detection tables) as shown in fig. 4.

2.2.1 Fish

ERM Entity: (*fish*)

The fish entity represents a unique fish which has been detected in a video. In other words, if a fish appears in multiple frames, this will create a single record in the database (the tracking component takes care of associating multiple detections to the same fish). To a fish we associate its species (as obtained by the recognition algorithm), its best view as bounding box (the frame coordinates $\{column, row\}$ of the four corners of the rectangle), its contour (a sequence of

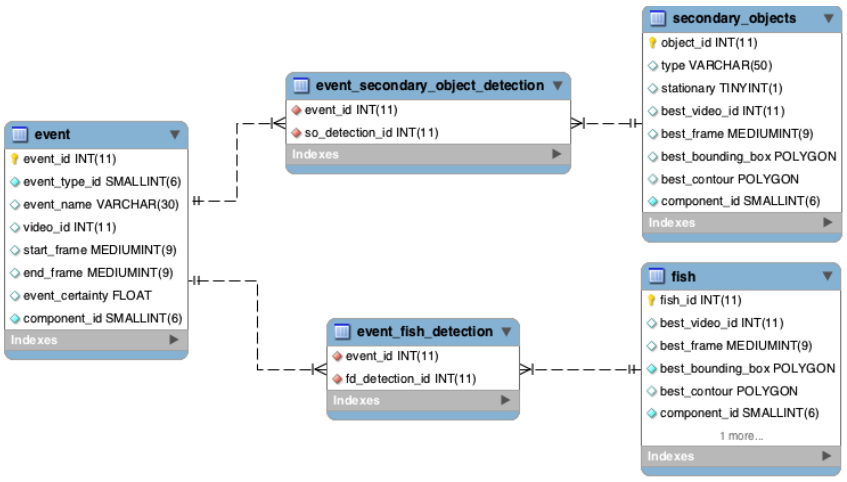


Figure 4: ERM schema for Event Detection

consecutive points of frame coordinates $\{column, row\}$), the frame number it was detected at, and the software component which detected this fish.

Attributes:

- *fish_id* (**integer**): unique identifier for the fish.
- *best_video_id* (**integer**): foreign key to the *video* entity (see below); it specified in which video we have the best view for this fish.
- *best_frame* (**integer**): frame number in the *best_video_id* video, where the best view of the fish is.
- *best_bounding_box* (**polygon**): this specifies the bounding box for the best view of the fish, as the absolute frame coordinates $\{column, row\}$ of the four vertices (corners) of the rectangle.
- *best_contour* (**polygon**): field that contains a list of consecutive points of coordinates $\{column, row\}$ defining the contour of the object (this might be subject to changes, according to performance and storage considerations).
- *component_id* (**integer**): foreign key to the *software_component* entity, identifies the component which created this record (i.e. which detected this fish).

ERM Relationship: (*fish_species*)

This relationship represents the fact that the recognition process might assign to each identified object different species with different levels of certainty.

Attributes:

- *fish_id* (**integer**): foreign key to the *fish* entity.

- *species_id* **(integer)**: foreign key to the *species* entity, identifies the species to which this fish belongs (according to the classifier).
- *certainty* **(float)**: certainty score for each specie assigned to the identified fish.

2.2.2 Detection of a fish

ERM Entity: (*fish_detection*)

Each *fish_detection* instance represents an object in a video which has been recognized as a fish by the detection algorithm. Unlike *fish*, we can have multiple *fish_detection* records referring to the same fish, if it appears in multiple frames. Each detected blob is assigned with two scores: detection_certainty and tracking_certainty. The former score describes the certainty that a detected blob is a fish and it is included between 0 and 1 (the larger, the more certain). The latter score describes the certainty that a detected blob is correctly associated to the blobs detected in the previous frames. It also ranges within 0 and 1.

Attributes:

- *detection_id* **(integer)**: unique key of the fish detection entity; it identifies each object detected by the detection algorithms.
- *fish_id* **(integer)**: foreign key to the *fish* entity, identifies the specific (i.e. unique) fish this detection refers to.
- *video_id* **(integer)**: foreign key to the *video* entity, specifies the video in which this detection was found.
- *frame_id* **(integer)**: detection frame number in the video.
- *bounding_box* **(polygon)**: this specifies the bounding box for the best view of the fish, as the absolute frame coordinates $\{column, row\}$ of the four vertices (corners) of the rectangle.
- *contour* **(polygon)**: field that contains a list of consecutive points of coordinates $\{column, row\}$ defining the contour of the object (this might be subject to changes, according to performance and storage considerations).
- *detection_certainty* **(float)**: certainty score of the detected object.
- *tracking_certainty* **(float)**: certainty score of the tracked object.
- *timestamp* **(date and time)**: estimation of the date and time of the detection performed by a software component.

2.2.3 Secondary (non-fish) objects

ERM Entity: (*secondary_object*)

This entity represents non-fish objects found in the videos which have some kind of interaction with fish, especially for the event detection component. The structure of this entity and the way detections are managed are similar to *fish* and *fish_detection*. However, consider the scenario in which a fish hides behind a rock. It might be useful to save the rock as a secondary object and to associate it to the "fish hiding" event (see below). Of course, it would be useless and unfeasible to save all detections of the rock; for this reason, secondary objects can be marked as stationary, in which case detections are not saved.

Attributes:

- *object_id* **(integer)**: unique identifier for the object.
- *type* **(string)**: name of the object (e.g. rock, plant, etc).
- *stationary* **(boolean)**: specifies if the object is stationary (if that is the case we do to save detections). By default, this value is set to false.
- *best_video_id* **(integer)**: foreign key to the *video* entity, specifies in which video we have the best view for this object.
- *best_frame* **(integer)**: frame number in the *best_video_id* video, where the best view of the object is.
- *best_bounding_box* **(integer)**: it specifies the bounding box for the best view of the object, as the absolute frame coordinates $\{column, row\}$ of the four vertices (corners) of the rectangle.
- *best_contour* **(polygon)**: field that contains a list of consecutive points of coordinates $\{column, row\}$ defining the contour of the object (this might be subject to changes, according to performance and storage considerations).
- *component_id* **(integer)**: foreign key to the *software_component* entity; it identifies the component which created this record (i.e. which detected this object).

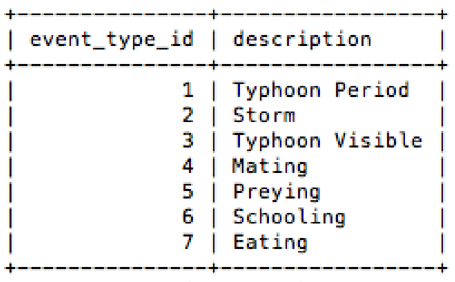
2.2.4 Detection of secondary objects

ERM Entity: (*secondary_object_detection*)

This is the equivalent of *fish_detection* for *secondary_objects*.

Attributes:

- *detection_id* **(integer)**: primary key to identify a specific detection.
- *object_id* **(integer)**: foreign key to the *secondary_object* entity, identifies the specific (i.e. unique) object this detection refers to.



```
+---------------+-----------------+
| event_type_id | description     |
+---------------+-----------------+
|             1 | Typhoon Period  |
|             2 | Storm           |
|             3 | Typhoon Visible |
|             4 | Mating          |
|             5 | Preying         |
|             6 | Schooling       |
|             7 | Eating          |
+---------------+-----------------+
```

Figure 5: List of Events in the datastore

- *video_id* **(integer)**: foreign key to the *video* entity, specifies the video in which this detection was found.
- *frame_id* **(integer)**: detection frame number in the video.
- *bounding_box* **(polygon)**: specifies the bounding box of this detection, as the absolute frame coordinates ($\{column, row\}$) of the four vertices (corners) of the rectangle.
- *contour* **(blob)**: binary field which contains a list of the points defining the contour of the object (this might be subject to changes, according to performance and storage considerations).
- *detection_certainty* **(float)**: certainty score of the detected object.
- *tracking_certainty* **(float)**: certainty score of the tracked object.
- *timestamp* **(date and time)**: estimation of the date and time of the detection performed by a software component.

2.2.5 Event types

ERM Entity: (*event_type*)

This entity lists the possible events which can be detected by analyzing the videos for fish-environment interactions. A preliminary set of events right now stored in the datastore is shown in fig. 5. Currently, we simply define the names for the possible events.

Attributes:

- *event_type_id***(integer)**: unique identifier for the event type.
- *description* **(string)**: short description of the event.

2.2.6 Events

ERM Entity: (*event*)

This entity represents an event, defined as the recognition of a particular behavior of a fish or a group of fish (e.g. eating, preying, schooling, etc). The sets of involved fish and, if necessary, secondary objects is associated to the event (via the *event_fish_detection* and *event_secondary_object_detection* entities, see below).

Attributes:

- *event_id* **(integer)**: unique identifier for the event.
- *event_type_id* **(integer)**: foreign key to the *event_type* entity, specifies the kind of event.
- *event_name* **(string)**: name of the specific event of type *event_type_id*.
- *video_id* **(integer)**: foreign key to the *video* entity, specifies in which video the event was detected.
- *start_frame* **(integer)**: number of frame in which the event began.
- *end_frame* **(integer)**: number of frame in which the event ended.
- *event_certainty* **(float)**: certainty score of the event detection algorithm.
- *component_id* **(integer)**: foreign key to the *software_component* entity, identifies the component which created this record (i.e. which recognized this event).

2.2.7 Fish detections related to an event

ERM Relationship: (*event_fish_detection*)

This relationship captures how *fish* and *event* entities are related and, more specifically, it lists all the detections of a fish involved in an event.

Attributes:

- *event_id* **(integer)**: foreign key to the *event* entity, specifies to which event this detection refers to.
- *fd_detection_id* **(integer)**: foreign key to the *fish_detection* entity, specifies the detection of a fish involved in the event.

2.2.8 Secondary object detections related to an event

ERM Relationship: (*event_secondary_object_detection*)

This relationship lists all detections of the secondary objects involved in an event. Similarly to *event_fish_detection*, it describes the relation between *secondary_object* and *event* entities. If the object is stationary and no detections of the object were saved in the time interval of the event, then an arbitrary detection of the object is assigned to the event.

Attributes:

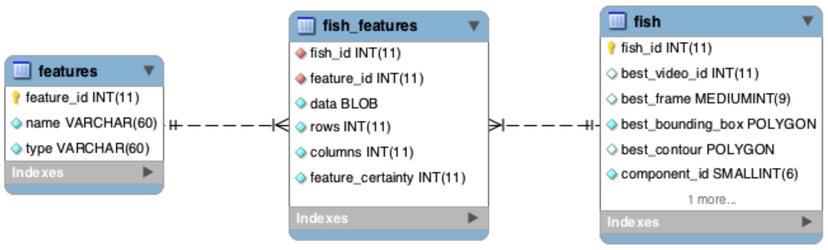


Figure 6: ERM schema for Low Level Features of Fish

- *event_id* **(integer)**: foreign key to the *event* entity, specifies to which event this detection refers to.
- *so_detection_id* **(integer)**: foreign key to the *secondary_object_detection* entity, specifies the detection of a secondary object involved in the event.

2.3 Low Level Features

These entities aim at storing the low level features extracted from fish, non-fish object, videos and frames for purposes such as fish recognition, videos classification, event classification, etc. An example of the datastore schema for storing features about fish is shown in fig. 6.

2.3.1 Fish/secondary object/video description features

ERM Entity: (*feature*)

This entity lists the possible features which can be used to describe fish and secondary objects (for classification purposes) and videos (for video-specific computed attributes). For each feature, its name and the kind of data (integer, floating point, string, ...) stored in it are specified.

Attributes:

- *feature_id* **(integer)**: unique identifier for the feature.
- *name* **(string)**: a short feature name.
- *type* **(string)**: a definition of the kind of data in which the feature is represented (the format of this string is to be specified in detail).

2.3.2 Fish description features

ERM Relationship: (*fish_features*)

This relationship describes all features computed on fish and represents the relation between entities fish and feature. In order to support matrix data, it is possible to specify the number of rows and columns this data requires. The actual data is stored in binary format, row by row (from left to right). The bit width of each element depends on the type of the feature. Given a specific fish, features can be computed from any of its detections, not necessarily the one corresponding to the best view.

Attributes:

- *fish_id* **(integer)**: foreign key to the *fish* entity, specifies on which fish this feature was computed.
- *feature_id* **(integer)**: foreign key to the *feature* entity, identifies the computed feature for a specific fish.
- *data* **(blob)**: actual data of the feature. It consists of a stream of data elements (the width of which depends on the type of feature), organized by rows, consistently with the rows and columns fields.
- *rows* **(integer)**: number of data rows.
- *columns* **(integer)**: number of data columns.
- *feature_certainty* **(float)**: certainty score of the computed feature. For some kind of features (e.g. histogram, Gabor descriptors, etc) there actually is no uncertainty, so this value is always 1. For other kinds, typically at higher-level features such as the number of fins, it actually makes sense to specify a certainty score.

2.3.3 Secondary object description features

ERM Relationship: (*secondary_object_feature*)

This relationship describes all features computed on secondary objects, in a similar way as fish_features.

Attributes:

- *object_id*(**integer**): foreign key to the *secondary_object* entity, specifies on which object this feature was computed.
- *feature_id* **(integer)**: foreign key to the *feature* entity, identifies the computed feature.
- *data* **(blob)**: actual data of the feature. It consists of a stream of data elements (the width of which depends on the type of feature), organized by rows, consistently with the rows and columns fields.
- *rows* **(integer)**: number of data rows.
- *columns* **(integer)**: number of data columns.
- *feature_certainty* **(float)**: certainty score of the computed feature (see *fish_features* entity).

2.3.4 Video description features

ERM Relationship: (*video_feature*)

This relationship contains all features computed on videos. In order to support matrix data, it is possible to specify the number of rows and columns this data requires. The actual data is stored in binary format, row by row (from left to right). The bit width of each element depends on the type of the feature.

Attributes:

- *video_id* **(integer)**: foreign key to the *video* entity, specifies on which fish this feature was computed.
- *feature_id* **(integer)**: foreign key to the *feature* entity, identifies the computed feature.
- *data* **(blob)**: actual data of the feature. It consists of a stream of data elements (the width of which depends on the type of feature), organized by rows, consistently with the rows and columns fields.
- *rows* **(integer)**: number of data rows.
- *columns* **(integer)**: number of data columns.
- *feature_certainty* **(float)**: certainty score of the computed feature (see *fish_feature* entity).

2.4 Software components

This entity aims at storing the software components used for all type of processing, from fish detection and tracking to features extraction to fish recognition to event detection and classification. There are several reasons why we might want to store results from different algorithms for detection, tracking, classification, etc. For instance, two algorithms might have different characteristics that make each of them more suitable for different kinds of queries. In order to manage results obtained by different algorithms, all data that is produced (e.g. detections, tracking, classification) is associated to the software component which created it. This causes replication of data (for example, if two detection algorithms detect the same fish, two records will be created in the fish entity, see below), but allows to keep track of the differences between the results and to let the user choose from which source to retrieve the data.

ERM Relationship: (*software_components*)

Attributes:

- *component_id* **(integer)**: unique identifier for the software component.
- *type* **(string)**: type of algorithm implemented by the component (e.g. detection, tracking, etc).
- *method* **(string)**: name of the algorithm.

- *version* **(integer)**: component version (we might want to overwrite results written by the same component but with a different version).
- *execution*_time **(integer)**: measure in $msec$ of the execution time of the algorithm.
- *settings* **(string)**: the set of parameters used by this component (e.g. thresholds).
- *input* **(string)**: input required by the component (e.g. video, fish list, etc.).
- *output* **(string)**: type of data produced by the component.

2.5 Metadata

This entity is meant to provide high level information for some fields of the databases. In detail, the entity is supposed to contain units, valid ranges, missing data symbol for fields of the database.

ERM Relationship: (*metadata*)

Attributes:

- *id* **(integer)**: unique identifier of the entity.
- *name* **(string)**: the name of an attribute of a db entity. The best practice is use to use something in the form "table.attribute". For example, if we want to provide information on the field "event_certainty" of the table events, this field should be "events.event_certainty".
- *type* **(string)**: datatype: double, integer, string, etc.
- *allowed_values* **(integer)**: valid ranges for the attribute specified in field "name".
- *unit*_time **(string)**: units of measurements.
- *missing_data_symbol* **(string)**: symbols used to describe that the information for that field is not available.
- *notes* **(string)**: additional notes.

2.6 Database Implementation

We provide two implementations of the relational datastore schema described above: MySQL and PostgreSQL. This has been done mainly in order to identify which is the best solution in terms of query optimization, available datatype for the specific data and storage size limitations. The first implementation of the relational datastore schema above described was done in MySQL. This was mainly due to the simplicity of implementation and the availability of connectors for any programming language. Afterwards, due to size limits the MySQL database was migrated to a PostgreSQL one. In fact, the effective maximum entity size for MySQL databases for Win32 w/ NTFS, Linux 2.4+ and Mac OS X, is respectively, 2TB, 4TB and 2TB, whereas the maximum entity size for PostGreSQL databases is independent from the operating and is 32 TB.

The MySQL data server has the following credentials:

```
IP address:  151.97.9.184
IP port:  3306
Database:  f4k-db
username:  f4k
username:  f4kpwd
```

The PostgreSQL server, instead, has the following credentials:

```
IP address:  151.97.6.81
IP port:  5432
Database:  f4k
username:  guest
username:  guest
```

2.6.1 C++ Interface for F4K Relational Database

To accommodate the need of the programmers to handle data in the F4K database, in a higher level than plain SQL language, a module that hides the complexity of I/O operations to and from the database has been created. The module is called *DBDispatcher* and consists of the following files:

- `DBDispatcher.h`
- `DBDispatcher.cpp`
- `DBObject.h`

These files include the functions and data variables, necessary to abstract the low level data operations and they depend on the following programs/classes that have to be already installed and configured on the computer.

- The MySQL Connector or PostgreSQL for C++ version.
- The OpenCV image processing library. The module is currently compatible with the 2.1 and 2.2 versions of the library.
- The F4K detection and tracking platform's DBObjects, in particular:
  1. `Fish.h and .cpp`
  2. `Fish_Detections.h and .cpp`
  3. `Species.h`
  4. `SWComponent.h and .cpp`
  5. `Camera.h`
  6. `Video.h and .cpp`

In the following the description of the developed classes is given.

2.6.2 Class Description

- **The Fish class**
  File: `Classes/Fish.h`
  The fish class contains variables and methods necessary to describe completely a fish in the database. The variables that a Fish object contains are:

  - `int id`. This is the fish_id associated to the detected object after being inserted in the database.
  - `int specie`. This is the fish's specie_id. Currently it is not used.
  - `int componentid`. The component_id of the software component used for fish detection
  - `int bvideo`. The id of the video which the fish belongs to.
  - `int bframe`. The number of the frame of the video that contains the best view of the fish.
  - `short bboxx`. The x coordinate of the bounding box that contains the fish's mask.
  - `short bboxy`. The y coordinate of the bounding box that contains the fish's mask.
  - `short bboxh`. The height of the bounding box that contains the fish's mask.
  - `short bboxw`. The width of the bounding box that contains the fish's mask.
  - `IplImage *bcontours`. The image representation of the contours of the best view of the fish.
  - `vector<Fish_Detection*> detections`. A vector that contains all the fish's detections.

- **The Fish_Detection class**
  File: `Classes/Fish_Detection.h`
  This class represents a detection of a fish. The variables that a Fish_Detection object contains are:

  - `int fish_id`. The fish's ID that the detection object belongs to.
  - `int frame_id`. The number of the frame of the video that contains the detection.
  - `short bboxx`. The x coordinate of the bounding box that contains the fish's mask.
  - `short bboxy`. The y coordinate of the bounding box that contains the fish's mask.
  - `short bboxh`. The height of the bounding box that contains the fish's mask.
  - `short bboxw`. The width of the bounding box that contains the fish's mask.
  - `int video_id`. The `id` of the video which the fish belongs to.
  - `long timestamp`. The timestamp of the captured frame.
  - `float detection_certainty`.

  - `float tracking_certainty`. These two parameters are used to evaluate the correctness of the detection/tracking.
  - `IplImage *bcontours`. The Image representation of the detection's mask.

- The Camera class
  File: `Classes/Camera.h`
  This class represents a capturing camera in the database. The variables that a Camera object contains are:
  - `int id`. The `id` of the the camera which is generated after the camera object is inserted in the database.
  - `int video_num`. The number of the camera that resides in a site.
  - `int camera_angle`. The camera angle in degrees.
  - `string camera_lens`. This variable describes the type of the camera.
  - `string location`. The name of site where the camera is located.

- The Video class
  File: `Classes/Video.h`
  This class contains information about a captured video. The variables that a Video object contains are:
  - `int camera_id`; The id of the camera that this video has been acquired.
  - `int video_id`; The video's ID. This number is generated after the video object is inserted in the database.
  - `long date_time`; When a video has been acquired.
  - `int length`
  - `int frame_rate`
  - `int frame_height`
  - `int frame_width`
  - `int frame_depth`
    These variables describe low level features of the video.

- The SWComponent class
  File: `Classes/SWComponent.h`
  The Software Component class is an abstract class that represents a software module used for fish detection/tracking/classification. Every software module must extend this class and implement its pure virtual methods.

- Dispatcher Initialization
  To initialize a DBDispatcher object, one must pass the following parameters in the constructor:
  1. The type of RDBMS, namely, MySQL or PostgreSQL.

2. The prefix `''tcp://''` concatenated with the IP address and the port of the database server; an example of string to be passed to the method is: `''tcp://192.168.0.1:3306''`.
3. The username.
4. The password.

So a valid `DBDispatcher object` is something like:
`DBDispatcher* dispatcher = new DBDispatcher(''MySQL'',''tcp://192.168.0.1:3306'',''username'',''password'');`

The methods interfacing to the database are included in the db variable of the `DBDispatcher object`. So to insert a software component in the database one must write:
`dispatcher→db→insertSWComponent(component);`

2.6.3 How to insert/update data

To insert data in the database the module uses the following functions:

- `void insertSWComponent(SWComponent* component);`
  This function is used to insert a software component in the DB. It is used to populate the *software_component* entity.
- `void insertFish(Fish *fish);`
  After all the necessary information to describe a fish have been acquired by the tracking and detection modules the program should create a Fish object. In this module are included data like bounding boxes of the best views, fish ids, fish detections etc..
  Fish detections are inserted by calling the `addDetection` member function of the Fish class and are all stored in a vector. Fish detections are inserted when the user calls the `insertFish` function.
- `void insertSpecies(Species* specie);`
  This function inserts a new specie in the DB.
- `void insertVideo(Video* video);`
  The insertVideo function serves the role of registering a new video in the DB. If a video with the same parameters exists the function fills all the missing member variables with the values read from the DB .
- `void insertCamera(Camera* camera);`
  The insertVideo function serves the role of registering a new camera in the database. If a camera with the same parameters exists the function fills all the missing member variables with the values read from the DB
- `void insertObject(DBObject *obj);`
  This is a wrapper function. It calls the `getObjectType` function, defined in `DBObject.h` and overloaded by all the other classes that derive the DBObject class. It can be used to replace every other insert function.

Similar methods, to the ones shown above, have been implemented for updating the database's records.

2.6.4 How to retrieve data

- `void getObjectsFromVideo(Video* video,vector<Fish*>& fish);`
- `void getObjectsFromVideoID(int id,vector<Fish*>& fish);`

These functions return a vector containing all the detected fish in a video. The video can be passed with its fish_id or by passing a Video object. For the latter the video's filename has to be decoded and a call to the insertVideo function must be made. If the video exists in the database, the Video object will be populated by the data present, else the video will be inserted with a new ID and the querying of the database will result in an empty fish vector.

3 Linked Open Data

All data described according to the relational schema will be made available through common SQL application programmer interfaces, so it will become readily available for reuse. We envision, however, that most third party applications that either need to integrate the data in our relational storage into data sets with other schemas, or applications that need to link our data to other data or *vise versa* can do so more easily if the data is also exposed as Linked Open Data (LOD) using the RDF data model. The RDF data is published in three categories. First, all relational data described above is being exposed using a direct mapping to RDF. Second, a taxonomy of all Taiwanese coral reef fish is published in SKOS. Finally, we will publish RDF links between the relational data and the SKOS taxonomy, to other relevant LOD data sets and alternative representations of the key data.

3.1 Direct Mapping to RDF

The basis of our LOD data set is a direct, one-to-one mapping from all the relational data described above to RDF, using the guidelines described in "A Direct Mapping of Relational Data to RDF" Working Draft[1] that is currently under development in the W3C RDB2RDF Working Group[2]. Note that one of the key advantages of the Direct Mapping is that the RDF Schema are directly derived from the relational schema defined in section 2. We will therefore not duplicate this information, but instead show some examples to explain the key principles. The key current Fish4Knowledge Linked Open Data addresses for the Direct Mapping are:

HTML browser entry point	`http://f4k.project.cwi.nl/lod/`	
Namespace URL	`http://f4k.project.cwi.nl/lod/`	
Namespace abbreviation	`f4k:`	`http://f4k.project.cwi.nl/lod/`
RDF browser entry point	`f4k:all`	`http://f4k.project.cwi.nl/lod/all`
SPARQL end point	`f4k:sparql`	`http://f4k.project.cwi.nl/lod/sparql`
SPARQL explorer	`f4k:snorql`	`http://f4k.project.cwi.nl/lod/snorql`

[1] `http://www.w3.org/TR/2011/WD-rdb-direct-mapping-20110324/`
[2] `http://www.w3.org/2001/sw/rdb2rdf/`

The example below shows the RDF description of camera #25, the third camera on HoBiHu harbor reef under the Direct Mapping, using Turtle as the serialization syntax. Note that by default, also the URLs of the resources which use camera #25 as an object are being returned. Below, we only included the first of these, in this video fragment #16, that was apparently shot by camera #25:

```
@prefix f4k:   <http://f4k.project.cwi.nl/lod/> .
@prefix vocab: <http://f4k.project.cwi.nl/lod/vocab/resource/> .
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix d2r:   <http://sites.wiwiss.fu-berlin.de/suhl/bizer/d2r-server/config.rdf#> .

<http://f4k.project.cwi.nl/lod/resource/cameras/camera_id=25>
      a  <http://f4k.project.cwi.nl/lod/vocab/resource/f4k.cameras> ;
      rdfs:label "camera #25 (3@HoBiHu)" ;
      vocab:cameras_camera_id "25"^^xsd:int ;
      vocab:cameras_location "HoBiHu" ;
      vocab:cameras_video_number "3"^^xsd:int .

<http://f4k.project.cwi.nl/lod/resource/videos/video_id=16>
      vocab:videos_camera_id
              <http://f4k.project.cwi.nl/lod/resource/cameras/camera_id=25> .

...
```

The next example shows an entry of the fish_detection table under the Direct Mapping. Note that — as described in section 2.2.2 — detections are identified not by a single ID but by a composite key combining the fish, video and frame IDs. Namespace declarations are omitted for brevity, and are identical to the previous example.

```
f4k:resource/fish_detection/fish_id=261208,video_id=3576,frame_id=1652
      a vocab:f4k.fish_detection ;
      rdfs:label "fish_detection fish261208/v3576/f1652" ;
      vocab:fish_detection_detection_certainty
              "0.88877"^^xsd:double ;
      vocab:fish_detection_fish_id
              <http://f4k.project.cwi.nl/lod/resource/fish/fish_id=261208> ;
      vocab:fish_detection_frame_id
              "1652"^^xsd:int ;
      vocab:fish_detection_timestamp
              "2010-12-16"^^xsd:date ;
      vocab:fish_detection_tracking_certainty
              "0.79386"^^xsd:double ;
      vocab:fish_detection_video_id
              <http://f4k.project.cwi.nl/lod/resource/videos/video_id=3576> .
```

The Turtle code above can also be accessed over HTTP by requesting the `text/turtle` MIME type on the resource's URL. For example, by using the open source `curl` command line tool, this can be done as follows:

```
shell> curl -LH "Accept: text/turtle"
         "http://f4k.project.cwi.nl/lod/resource/cameras/camera_id=25"
```

```
shell> curl -LH "Accept: text/turtle"
         "http://f4k.project.cwi.nl/lod/resource/fish_detection/fish_ \\
          id=261208,video_id=3576,frame_id=1652"
```

By using the `application/rdf+xml` MIME type, the same resources can be returned in RDF/XML notation, e.g.:

```
shell> curl -LH "Accept: application/rdf+xml"
            "http://f4k.project.cwi.nl/lod/resource/cameras/camera_id=25"
```

3.2 Taiwanese coral reef fish taxonomy in SKOS

This RDF data set will use SKOS to provide a detailed taxonomy of all coral reef fish species that live on the Taiwanese reefs. It will be based on information from authoritative sources such as the Fish Database of Taiwan by Prof.dr. K.T. Shao from the Biodiversity Research Center of the Academia Sinica in Taiwan. At the time of writing, a first, tentative version includes 28113 fish images, associated with 2893 species descriptions. These species belong to 1051 genera, 300 families, 61 suborders, 47 orders and 3 (sub)classes. (Note that the taxonomy not necessarily has a single root as fish are a paraphyletic collection of taxa, of which 3 are potentially relevant to our project).

The Fish4Knowledge SKOS taxonomy will play several roles. First, it will provide a species-centric access point to the Fish4knowledge LOD data, where the relational data described above primarily provide a detection-centric view. It will provide the most natural link target to link Fish4knowledge data to other LOD data that is species-oriented, such as the species descriptions in AGROVOC and other relevant taxonomies published by the Food and Agriculture Organization of the United Nations.

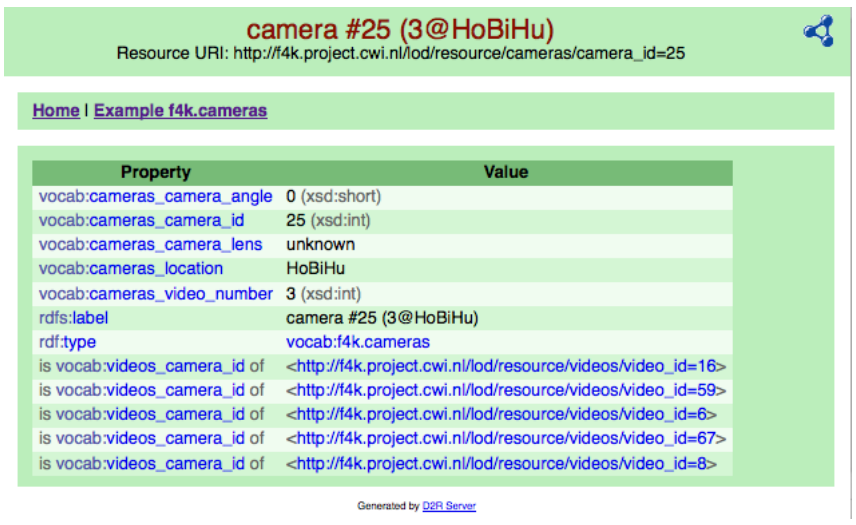
Second, it will be a means to publicly share, as LOD, the external resources that were used to train the species recognition software modules, for example by providing examples still images or contours of fish species, genera or families, or of specific features that are used, such as tail or fin shapes, or textures or color histograms etc.

Finally, the SKOS vocabulary provides an excellent means to systematically deal with the variety of names that are associated with fish species. Many species have different (accepted) scientific names, different common names in multiple languages, different transliteration of the same name, etc. Having this information in a machine-readable format such as SKOS is a key asset that can, for example, be used in the user interface to support query formulation (e.g. by using autocompletion or query expansion techniques) or in the result display (e.g. by automatically augmenting search results from the database with species names in languages appropriate to the current user).

Software used to create the taxonomy will be published as open source, along with OPM data describing the origin of the taxonomic.

3.3 Interlinking and alternative representations of Direct Mapping data

A vital part of the interlinking will be the mapping of the internal database keys of the relational species identifiers to the species that are defined as SKOS concepts in the taxonomy. As the species recognition data is not yet available at the time of writing, we will speculate on the exact nature of this mapping. However, since the species recognition will be based on



camera #25 (3@HoBiHu)
Resource URI: http://f4k.project.cwi.nl/lod/resource/cameras/camera_id=25

Home | Example f4k.cameras

Property	Value
vocab:cameras_camera_angle	0 (xsd:short)
vocab:cameras_camera_id	25 (xsd:int)
vocab:cameras_camera_lens	unknown
vocab:cameras_location	HoBiHu
vocab:cameras_video_number	3 (xsd:int)
rdfs:label	camera #25 (3@HoBiHu)
rdf:type	vocab:f4k.cameras
is vocab:videos_camera_id of	<http://f4k.project.cwi.nl/lod/resource/videos/video_id=16>
is vocab:videos_camera_id of	<http://f4k.project.cwi.nl/lod/resource/videos/video_id=59>
is vocab:videos_camera_id of	<http://f4k.project.cwi.nl/lod/resource/videos/video_id=6>
is vocab:videos_camera_id of	<http://f4k.project.cwi.nl/lod/resource/videos/video_id=67>
is vocab:videos_camera_id of	<http://f4k.project.cwi.nl/lod/resource/videos/video_id=8>

Generated by D2R Server

Figure 7: HTML rendering of `f4k:resource/cameras/camera_id=25`

the same authoritative resources that were crawled to create the SKOS taxonomy, we expect no problems in this area.

Second, we will provide links from geographical locations and event types to common LOD data sets such as DBpedia[3] and GeoNames[4]. Since the number of unique items of this type in the database is very small, we expect this mapping can be done manually without problems.

Third, we will provide links from event detections to relevant external event resources when available (e.g. typhoons or other major environmental events). Since we have precise data and location information for all detected events, we expect these links can be provided fully automatically with great reliability.

Finally, we will use the Open Provenance Model (OPM) and its associated RDF vocabularies[5] to provide machine readable, provenance metadata about our data sets. By combining the explicit data about the versions of the software components used, the timestamps of the detection and the processing work flows defined, we will strive to provide OPM data to fully describe the origin of all our data, in sufficient detail to allow full replication of it.

3.4 Implementation

All three categories will be implemented using different methods. A first prototype of the relational data under the Direct Mappings is available at the time of writing. Relational data is being exposed by running the open source D2R Server[6]. At the time of writing, we are using D2R version 0.7 alpha[7] and run it against a copy of the Postgres database described earlier in this document. The mapping table specifying the rules to map the SQL data to RDF is available from `http://f4k.project.cwi.nl/resources/d2r/f4k.ttl`.

The HTTP redirection required to conform to the LOD conventions[8] is performed by the

[3] `http://ckan.net/package/dbpedia`
[4] `http://ckan.net/package/geonames`
[5] `http://purl.org/net/opmv/ns`
[6] `http://www4.wiwiss.fu-berlin.de/bizer/d2r-server/`
[7] `http://sourceforge.net/projects/d2rq-map/files/D2R\%20Server/`
[8] `http://linkeddatabook.com/editions/1.0/`

D2R server. The general pattern is that any request for a URL starting with the pattern `http://f4k.project.cwi.nl/lod/resource/` is redirected using an HTTP "303 see other" redirect reply. The location to which is redirected depends on the HTTP `Accept:` header. For MIME types associated with an RDF serialization syntax, the redirection is to `http://f4k.project.cwi.nl/lod/data/` while for HTML related and unknown MIME types, the redirection is to `http://f4k.project.cwi.nl/lod/page/`.

For example, the URL `f4k:resource/cameras/camera_id=25/` will be directed to `f4k:data/cameras/camera_id=25/` in case the HTTP client application requests `text/turtle` as in the example above. However, if the client requests `text/html`, it will be redirected to `f4k:page/cameras/camera_id=25/`, allowing normal Web browsers to show the page in figure 7.