Hydra: A Database System with Facilities for Link Analysis

Robert Ayres
Department of Informatics and Simulation,
Royal Military College of Science,
Cranfield University,
Shrivenham, Swindon,
Wiltshire SN6 8LA
United Kingdom

Introduction

Most current systems which support link analysis are really drafting tools which allow information with a network structure to be presented in a visually attractive manner. Although such facilities are important they are often of limited utility in the initial analysis of data. Available systems also lack the features provided by conventional databases to support the general querying and modelling of information.

In this paper we present a database system which integrates support for link analysis and provides a full programming language, Hydra, in which complex queries and searches can be formulated. The advantages of this approach are as follows:

- The database programming language gives flexibility and enables the user to write programs or functions to carry out new searches on the database. The Hydra system thus provides a framework within which users can develop facilities which incorporate techniques from AI, such as case-based reasoning, or ontologically-based searches.
- It becomes possible to carry out conventional database queries alongside link searches. Such query facilities are often useful in finding entities which are starting points for further investigation through link analysis.
- A wide variety of values can be stored and manipulated — surrogates corresponding to entities (such as people or locations), numbers, text, and multimedia data such as photographs or video clips.
- Other facilities of database systems such as integrity constraints and multi-user access also become available.

Below we briefly describe the Hydra language and its facilities before presenting a graphical interface which has been developed to present query results in a more intuitive manner. This interface is little more than a presentation front end to the language but can easily be extended in power by encoding further queries or search primitives in the underlying Hydra language.

The Hydra Database Language

Hydra is a full programming language which includes all the facilities required to define, update, and query a database. It uses the functional programming paradigm in which all computation and manipulation of data is expressed in functional terms, and has a syntax similar to that of Miranda (Turner 1985). The novel feature of Hydra is that it provides facilities to carry out link-analysis queries on the database. These facilities are fully integrated into the programming language so it is possible to program new searches as well as directly using a number of built-in search primitives.

In order to ensure that any database will be amenable to link analysis the Hydra system uses a graph-based (effectively a binary-relational) data model in which application entities (such as persons or locations) and scalar values (such as strings or numbers) are represented as nodes in a directed graph. The labelled connections between these nodes represent application relationships such as the fact that Ronald is an associate of Richard, or that the age of Caroline is 29 (Figure 1). Adopting a data model of this kind ensures that link analysis can be carried out on the whole database while not losing any expressive power compared to standard relational database systems.

A Hydra database is defined by first declaring entity classes and functions corresponding to associations. The instance level is then constructed by introducing surrogates corresponding to application entities (which must begin with a capital letter) and giving definitions for the associations. A portion of the database shown in Figure 1 is built up as follows

```
entity location, person;

age      :: person -> number;
works_at :: person -> location;

create person Caroline;
create location KingGeorge;

age      Caroline = 29;
works_at Caroline = KingGeorge;
```

The first statement introduces two classes of entities

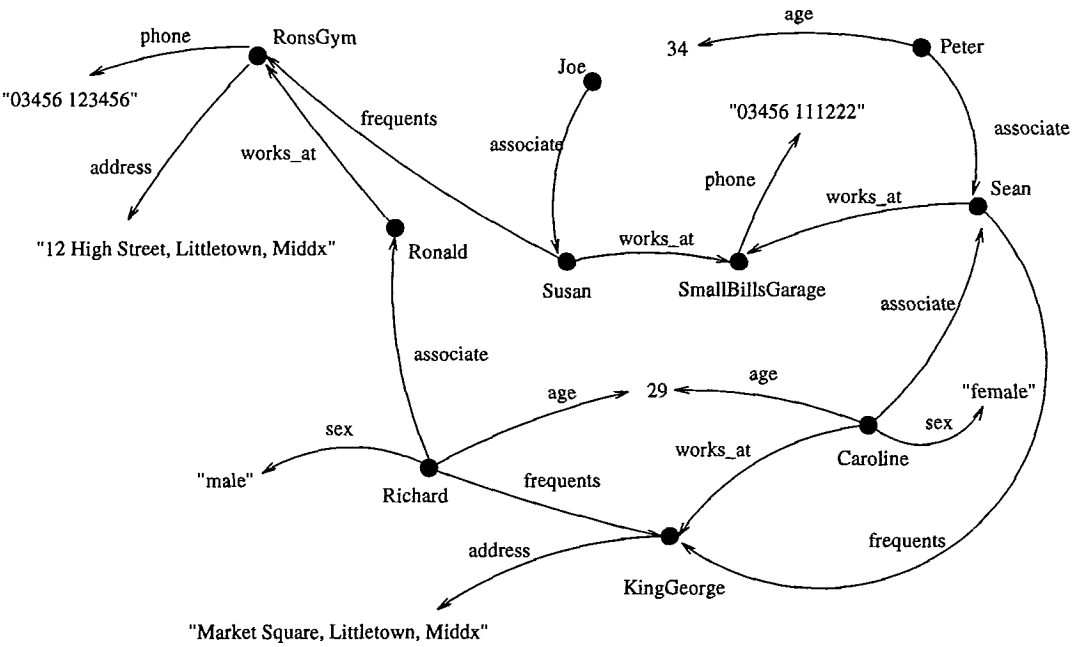

Figure 1: Instance-Level of a Criminal Intelligence Database

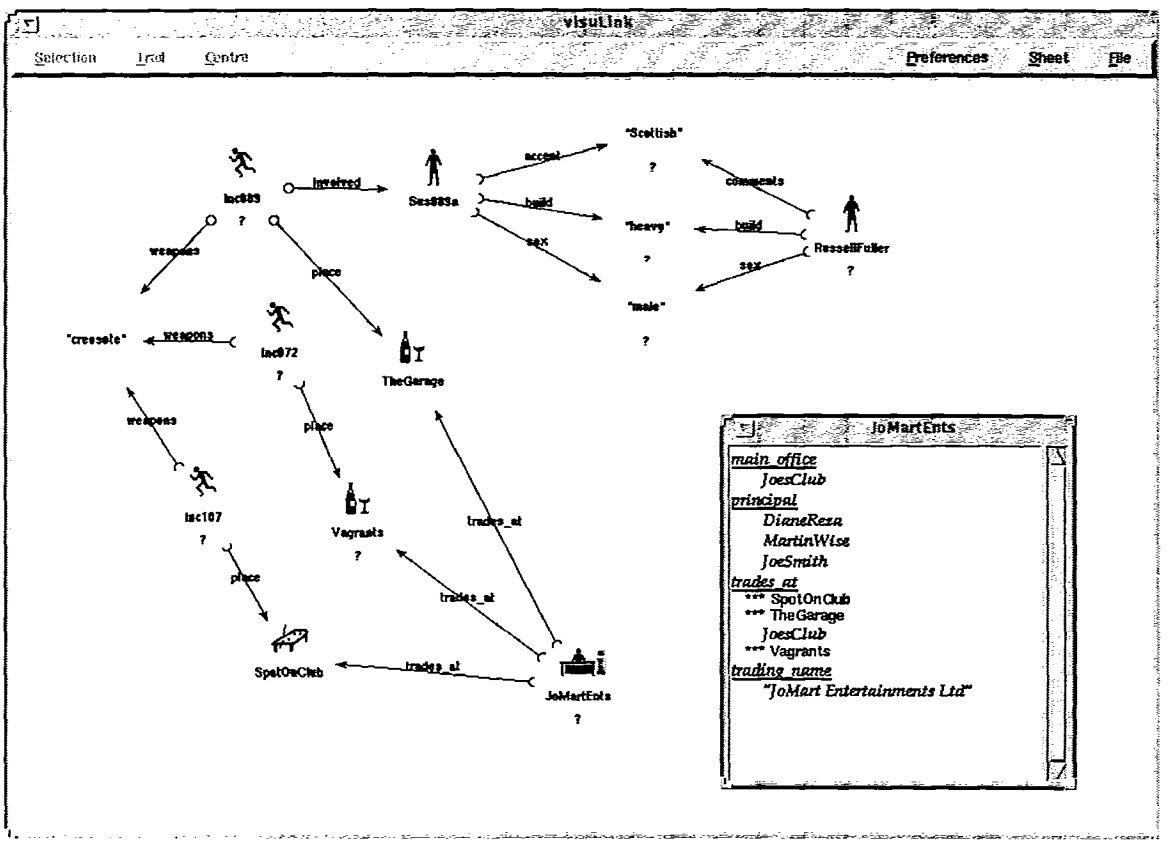

Figure 2: Portion of Database and subwindow with Information on JoMartEnts

(location and person); the following two statements declare the associations age and works_at and specify what kinds of value they can associate with each other. Two entities, represented by visible surrogates, are then introduced before defining some associations.

Hydra is a full programming language and so, unlike SQL, can be used as a general purpose programming system (though its main purpose is to enable users to express database queries). A conventional database query to find all men known to be under 30 can be made using a list abstraction (a standard construct in functional languages which is similar to the Zermelo-Frankel notation of set theory) as follows:

```
[m | m <- like ?person
   | sex m == "male" | age m < 30];
```

which can be paraphrased as *Return a list of all the m such that m is a person, the sex of m is "male" and the age of m is less than 30.* This will evaluate to the single-itemed list

```
[Richard]
```

Other queries, corresponding to the "select-project-join" style queries of SQL can easily be encoded using list-abstractions.

To find all the associations in which a value or entity may participate, one of the special link-oriented primitives is used as follows

```
from Susan;
```

This returns the list

```
[associate, works_at, frequents,
                    age, address, phone]
```

of all the associations defined on a person. Any of these associations can be used as a function and applied to the surrogate Susan. For instance

```
works_at Susan;
```

gives a list

```
[SmallBillsGarage]
```

of the locations where Susan works. Another built-in function, to, provides similar facilities to from but finds inverse associations. For example

```
to 29;
```

returns a list

```
[~age]
```

of inverse associations (whose names are prefixed with a tilde). Inverse associations correspond to following the directed arcs in the database graph "backwards". Applying the inverse function ~age to a number will find all the people with that age, so

```
~age 29;
```

gives the list

```
[Richard, Caroline]
```

of 29-year olds in the database. The use of these two built-in primitives can be combined to find everything known about a value or entity. For example

```
[(f, f Richard)| f <- append (from Richard)
                             (to Richard)
              | f Richard != [] ];
```

finds all the database functions which can be applied to Richard (append is a function to join two lists together), filters out those that give an empty result, and returns a list of pairs of the remaining functions and the result they give when applied to Richard. This effectively gives all the information directly associated with Richard in the database

```
[(age, [29]), (sex ["male"]),
          (frequents, [KingGeorge]),
                    (associate, [Ronald])]
```

This facility to find everything known about an entity can be generalised to a function known, defined as follows

```
known x = [(f,f x)| f <- append (from x)
                                (to x)
                  | f x != []];
```

Hence the expression known Richard will evaluate to the same result as the previous query. The function known has been defined in terms of the primitives from and to and provides a facility, that of finding all the information associated with an entity or value, which is not available in conventional systems. Indeed such facilities cannot be integrated into relational or object-oriented databases because these systems use data models which do not explicitly represent all the connections between database entities or values. Furthermore queries such as *Find everything known about Richard* are second order and cannot be replicated in standard database query languages which are limited to first order queries.

Another Hydra primitive trail can be used to find the connection between two database values or entities (if there is one). For example the query

```
trail 1 Richard KingGeorge;
```

will find any direct connections between Richard and the KingGeorge pub, returning the answer as a list of paths (each path represented as a sequence of nodes and links) as follows

```
[[Richard, frequents, KingGeorge]]
```

A search for longer connections can be carried out by giving a larger search depth, as in

```
trail 3 Peter KingGeorge;
```

which will return the two paths

```
[[Peter, associate, Sean, ~associate,
       Caroline, works_at, KingGeorge],
 [Peter, associate, Sean,
          frequents, KingGeorge]]
```

The trail primitive is built into the Hydra language for efficiency reasons (it could be programmed in terms of the primitives from and to but the searching would then be slower). The trail primitive automatically discards paths which contain loops or which pass through a scalar value (eg. a string or number). Scalar values

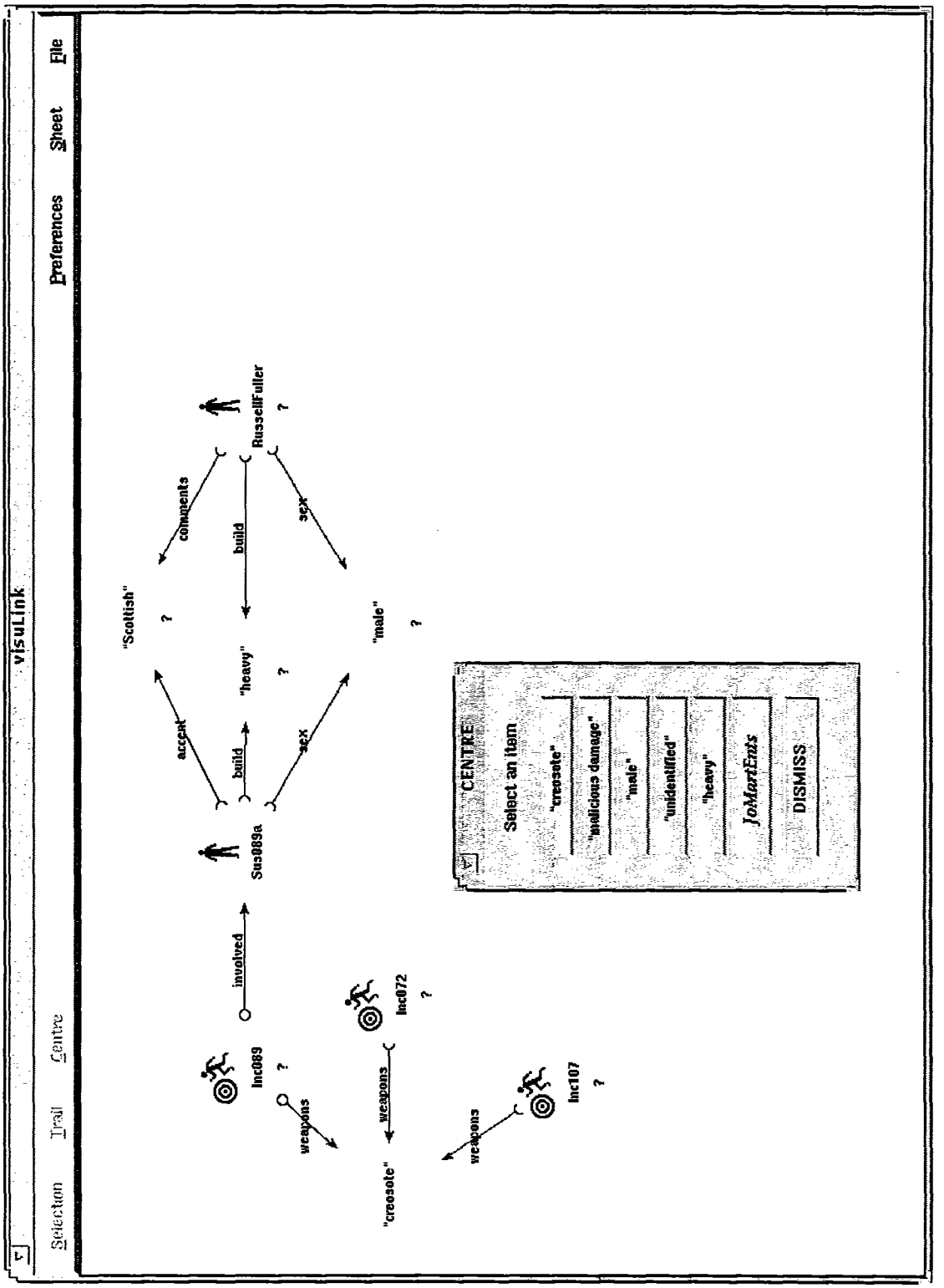

Figure 3: Canvas Showing Menu of Items Associated with Each of 3 Selected Incidents

can appear at the ends of paths but are excluded from mid-points since there would be the danger of retrieving paths of the form

```
[Bill, sex, "male", ~sex, John]
```

which are of little interest.

Given that the primitives `from` and `to` can be embedded into ordinary Hydra function definitions it is possible to program complex searches. For example it is relatively simple to develop a Hydra function `centre` which will take a search depth, a list of start nodes, and return a list of nodes which all within the search distance of each of the start nodes. Hence, given the database shown in Figure 1, the query

```
centre 2 [KingGeorge, 29, Susan];
```

will return the list

```
[Ronald, Sean]
```

of items which are all within 2 arcs distance of each of the three nodes in the original query. The `centre` function can be used to find a common association between a group of elements and can be enhanced to find the node which is, in aggregate, most closely connected to the start nodes. An example of the use of `centre` will be given in the next section.

Far more complex queries and functions can be coded in Hydra and any computable search algorithm can be implemented using its link-finding primitives.

A Graphical Interface

For most users the Hydra language has a number of drawbacks: it assumes a familiarity with functional programming and the results from queries, such as path-finding queries, are presented in a form which obscures the relationships which have been found. Furthermore a purely textual interface hinders the use of multimedia datatypes. These problems have motivated the development of a prototype graphical interface to the language. This interface exploits the graph-based data model to present data on a free-form canvas where both the connections modelled by the database and multimedia data can be incorporated (see Figure 2).

The Hydra interface initially provides the user with a blank canvas on which entities or values can be placed by clicking with a mouse and typing in the value or entity identifier. If the value or entity is present in the database a node will be placed on the canvas. Icons are used to represent entities, and the particular icon to use for any given entity is itself information which is held in the database. For example in Figure 2 a portion of a criminal intelligence database is shown where running-man icons are used to represent incident entities (that is recorded incidents such as robberies, or assaults), wine-bottle icons for nightclubs, a man-at-desk icon for a business and so on. A question mark is drawn on the canvas next to those items or values which have further information associated with them which is not shown on the canvas. Once an entity or value has been placed on the canvas, double-clicking on it brings up a sub-window which shows everything known about the item. In Figure 2 the user has double-clicked on the `JoMartEnts` entity and everything known about the entity, which is a nightclub business, is shown in the sub-window (items in the sub-window which are already on the canvas are prefixed by `***`). This information has been obtained by sending the query `known JoMartEnts` to the underlying Hydra database (making use of the `known` function described in the previous section). It is possible to select entities or values in the sub-window and place them on the canvas, at which point the interface also draws in the connecting arc. The user can drag items on the canvas to produce a more pleasing or intuitive layout than that produced by the interface's drawing routines.

Figure 2 actually shows a portion of an intelligence database where an investigating officer has started out by looking into a number of incidents where creosote has been used to vandalise premises. Having first placed the value `"creosote"` on the canvas the officer has then found that there have been three such incidents: `Inc089`, `Inc072`, and `Inc107`. A description of an unknown person, `Sus089a`, involved in incident `Inc089a` has been used to find an individual, `RussellFuller`, already in the database whose details are consistent with those of `Sus089a`.

In Figure 2 the user has also uncovered a connection between the incidents: they have all involved nightclubs run by the same company. This connection was found by using one of the facilities of the interface, `centre`. How this was done can be seen from a previous screen view, shown in Figure 3, where the user has selected three incidents (shown with target signs next to them) and used the `centre` facility on the left of the menu bar. This invokes the `centre` function described in the previous section and returns a menu of of "central" nodes (Figure 3). The user has selected the `JoMartEnts` entity and this has been placed on the canvas and its associations with other nodes drawn in to produce the state shown in Figure 2.

It is also possible to invoke the `trail` facility from the interface. Returning to Figure 2 the user has found a connection between the three incidents (they are all associated with `JoMartEnts`) and also discovered that a person whose details are already known, `RussellFuller`, has a description consistent with a suspect in one of the incidents. The obvious question to ask is whether `RussellFuller` has a connection with `JoMartEnts` — might he be a disaffected former employee for example? If a connection does exist in the data it it can be found by selecting these two entities (by clicking on them) and invoking the `trail` facility on the menu bar. This sends a `trail` query to the underlying Hydra database and the results are drawn on the canvas. Figure 4 shows the situation after this has been done and the user has further requested information on intermediate entities along the path uncovered.

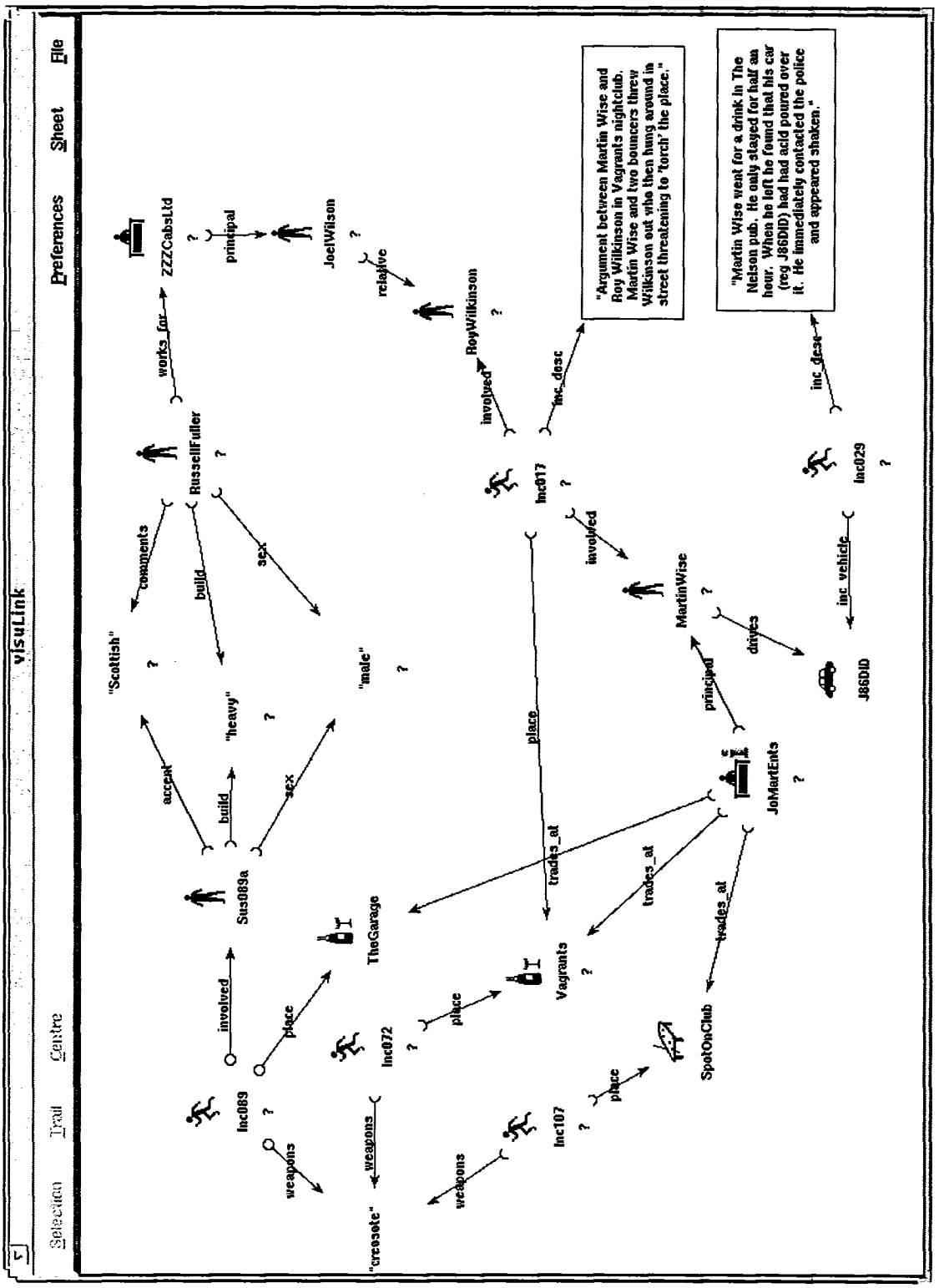

Figure 4: Database View showing Association between JoMartEnts and RusselFuller

Looking at this it appears that `RussellFuller` may have been hired, through intermediaries, to attack `MartinWise` and his business interests after the latter was involved in an altercation in a nightclub. Of course the connections uncovered do not in any way constitute evidence but they suggest a line of enquiry for the investigating officer.

Conclusion

The Hydra system demonstrates how an appropriate data model, combined with a small set of special link-oriented features, and a general purpose query language make it possible to develop a powerful system in which complex searches can be coded and in which it is possible to experiment with various AI technologies such as case-based reasoning.

Moreover these facilities can be made available through a graphical interface to users who do not need to master any programming language. The interface presented is quite generic in that it does no more than send queries (whose parameters are selected with mouse-clicks) to the Hydra system and format the results graphically. Thus the interface is easily extensible to accommodate a wide variety of search facilities once these have been coded in the Hydra language.

The Hydra system and associated interface are still under development. Future research avenues to explore include supporting certainty measures or weightings on the database associations and extending the graphical interface to make it more flexible and to allow the database to be defined and updated graphically as well as queried.

References

Turner, D. A. 1985. Miranda: a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architectures*. Springer-Verlag. Lecture Notes in Computer Science No. 201.