Semantic Indexing Based on Description Logics

Albrecht Schmiedel
Technische Universität Berlin
atms@cs.tu-berlin.de

Abstract

A method for constructing and maintaining a 'semantic index' using a system based on description logics is described. A persistent index into a large number of objects is built by classifying the objects with respect to a set of indexing concepts and storing the resulting relation between object-ids and most specific indexing concepts on a file. These files can be incrementally updated. The index can be used for efficiently accessing the set of objects matching a query concept. The query is classified, and, based on subsumption and disjointness reasoning with respect to indexing concepts, instances are immediately categorized as hits, misses or candidates with respect to the query. Based on the index only, delayless feedback concerning the cardinality of the query (upper and lower bounds) can be provided during query editing.

1 Introduction

Indexing generally involves an association between some kind of key and the actual target. The key is used to jump directly to a desired piece of information, thereby avoiding an exhaustive search through large sets of candidates. In the context of databases, keys are usually based on the set of values of a particular attribute of the objects to be indexed: if we know the value, we can move directly to the corresponding object(s).

In the following, a description logic (DL) based approach to indexing is sketched which broadens the notion of a key: instead of using attribute values, indexing elements can be arbitrary structured concepts as provided by a terminological language such as BACK (cf. [Hoppe et al.,1993]). Firstly, I will show how the construction of such an index falls out quite naturally from the normal workings of a terminological reasoner, and secondly I will discuss how such an index can be used. This approach, and an experimental implementation, is described in more detail in [Schmiedel,1993].

Basic entities:		
patient	:<	anything.
examination	:<	anything **and** **not**(patient).
observation	:<	anything **and** **not**(patient) **and** **not**(examination).
Basic relations:		
hasExam	:<	**domain**(patient) **and** **range**(examination).
hasItem	:<	**domain**(examination) **and** **range**(observation).
hasValue	:<	**domain**(observation) **and** **range**(number).
Other Primitives:		
hce	:<	examination.
bloodPressure	:<	observation.
bloodPressureSystolic	:<	bloodPressure.
bloodPressureDiastolic	:<	bloodPressure **and** **not**(bloodPressureSystolic).
normal	:<	observation.
abnormal	:<	observation **and** **not**(normal).

Table 1: Primitive concepts and roles

2 Index Construction

In a description logic such as BACK a data base is viewed as a set of distinct objects (also instances or individuals) typically representing domain entities, each of which is associated with a description.

Descriptions are terms built with

- *term-forming operators* such as **and**, **all**, **some**, etc., the logical constants provided by the language,
- *primitive concepts* and *roles* introduced by the user, and
- named *defined concepts* and *roles*.

Table 1 shows some top level primitive concepts roles for building a data base containing descriptions of patients, examinations, and observations made in

examinations[1]. Patients are related to examinations via hasExam, and examinations to observations via hasItem.

```
examSomeBpSysAbnorm   :=
  examination and some(hasItem,
                       bloodPressureSystolic and
                       abnormal).

patSomeBpAbnorm       :=
  patient and some(hasExam, examSomeBpAbormal).
```

Table 2: Defined concepts

Table 2 gives two examples for named descriptions (defined concepts) using the primitives. examSomeBpSysAbnorm is an examination which has an item which is an abnormal systolic blood pressure, and patSomeBpAbnorm is a patient which has an examination which has an abnormal blood pressure. Defined concepts are syntactic sugar for abbreviating possibly complex descriptions.

```
bloodPressureSystolic and all(hasValue, gt(140))
        =>  abnormal.
bloodPressureSystolic and all(hasValue, 110..140)
        =>  normal.
bloodPressureSystolic and all(hasValue, lt(110))
        =>  abnormal.
```

Table 3: Rules

Descriptions are also used to define rules, which are expressed as implications between two descriptions. The left hand sides of the rules shown in Table 3 are descriptions of certain sets of observations which are asserted to be in the set of normal or abnormal ones by the description on the right hand side.

Table 4 shows how data is actually entered into the system. The '::' operator is used for asserting that the description on the right hand side is true for the object referenced on the left hand side. Here, there is an object *patient1*, an instance of patient, with two examinations, *hce1* and *hce2*, both instantiating the concept hce. The keyword **closed** indicates that all fillers of the hasExam role are known, i.e. there are only two examinations. The examinations each have exactly two observations, each of which has exactly one numeric value.

Based on this type of input, the system computes

- for concepts the subsumption and disjointness relation, i.e., for each pair of concepts whether one subsumes the other or whether they are disjoint,
- for each individual the set of concepts it is (and is not) an instance of.

For our example containing three kinds of entities, patients, examinations, and observations, the result of this is illustrated in Fig. 1. Concepts (primitives marked with an asterisk) are related by subsumption links; disjointness has been left out for the sake of simplicity. The individuals at the bottom of the graph, a patient with two examinations, each of which with two observations, are linked to the most specific concepts they instantiate. For example, bpsys2 is classified under the conjunction of bPSystolic, which was explicitly told, and abnormal, due to an abnormality rule as in the example above. This leads to the classification of hce2 under examSomeBpSysAbnorm ('an examination with an abnormal systolic blood pressure') which in turn triggers the classification of patient1 as an instance of patSomeBpSysAbnorm ('a patient with an examination containing an abnormal systolic blood pressure'). Note that hce2 (patient1) was explicitly told to be only an examination (patient); the more specific concepts were derived by the system as a consequence of the role filler relations, the definitions and the rules.

patient1	::	patient **and** hasExam:closed(*hce1* **and** *hce2*).
hce1	::	hce **and** hasItem:closed(*bpsys1* **and** *bpdia1*).
hce2	::	hce **and** hasItem:closed(*bpsys2* **and** *bpdia2*).
bpsys1	::	bloodPressureSystolic **and** hasValue:130.
bpdia1	::	bloodPressureDiastolic **and** hasValue:90.
bpsys2	::	bloodPressureSystolic **and** hasValue:150.
bpdia2	::	bloodPressureDiastolic **and** hasValue:95.

Table 4: Object descriptions

In the following, two properties of description logic based systems not present in mainstream database systems play a crucial role:

- the ability to handle any degree of partial information in conjunction with an open world assumption, and
- the ability to describe individuals with complex concepts and to use these descriptions for query answering.

These two properties make it possible, for example, to remove all the information concerning observations (the shaded part in Fig. 1), but to keep all the information that was derived from observations concerning other entities. Thus, hce2 will still be known to be an instance of examSomeBpSysAbnorm, but the observations and their values from which this was derived will become unknown.

We can now define a set of individuals to be indexed (for example the set of patients), choose a set of indexing concepts (e.g., the concepts specializing patient), and store the relation which associates each indexing concept with the individuals it instantiates. This relation can efficiently be stored in two hashtables: one maps individual names to the set of most specific concepts describing them, and the other maps concept names to the set of individuals they directly instantiate, i.e. those which are not instances of any subconcept. It is also useful to store the associated cardinalities.

[1] For a more detailed description of the BACK language see [Hoppe et al.,1993].

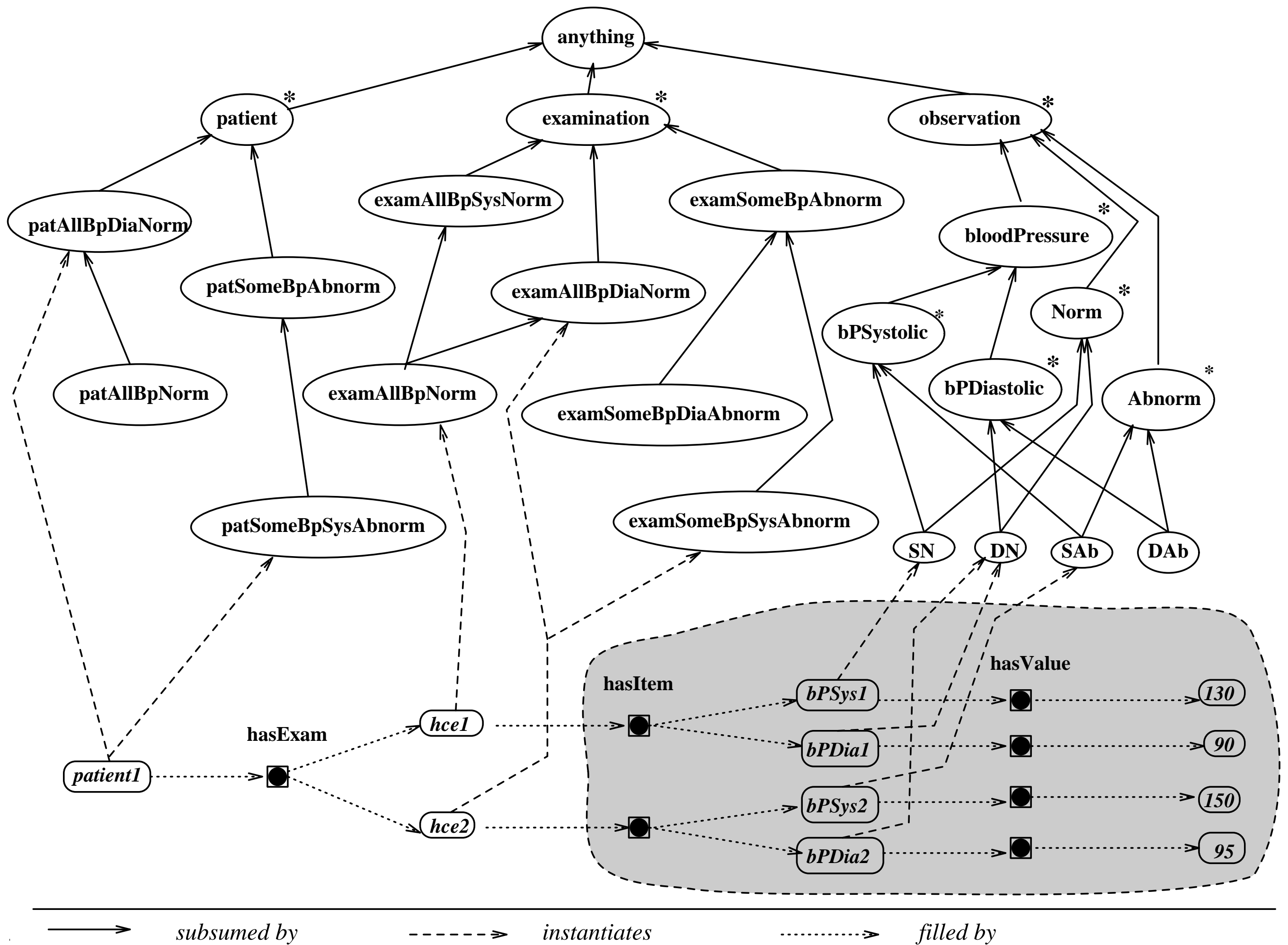

Figure 1: Example KB

3 Using the Index

Based on this stored relation and the original concept definitions, a new knowledge base can be built which contains only the classification of the individuals with respect to the indexing concepts, but lacks the full individual descriptions (see Table 5). It may thus be much smaller than the original KB. Due to the semantic properties mentioned above, it will be ignorant w.r.t to some information contained in the original KB, but it will never produce contradictory answers. This makes it useful as an index.

patient1	::	patSomeBpSysAbnorm **and** patAllBpDiaNorm.
hce1	::	hce **and** examAllBpNorm.
hce2	::	hce **and** examSomeBpSysAbnorm **and** examAllBpDiaNorm.

Table 5: Abstracted object descriptions

Queries using the index are processed in three distinct phases, each one providing progessively more information at additional costs. The first phase is designed to provide cheap and immediate feedback on the expected cardinality of the result of a query. For this only the cardinalities associated with indexing concepts need to be loaded. The query is classified, and cardinality constraints for it are computed based on the known cardinalities of indexing concepts, and their logical interrelations. Thus, the example query shown in Fig. 2 must have at least 40 instances, since there are two indexing subconcepts the cardinalities of which are added because they can be proved disjoint by the system. Similarly, there is an upper bound of 80 instances for the query, because the indexing superconcept with the least cardinality (100) has an indexing subconcept (20) disjoint from the query. Depending on this cardinality information, the user can either refine his query, specializing or generalizing it as desired, or proceed to the second phase.

The quality of the cardinality feedback depends very much on how close the query is related to already exist-

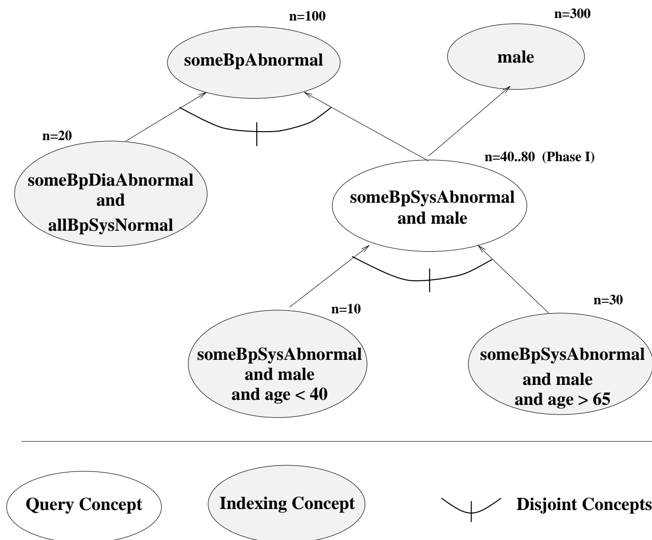

Figure 2: Approximating the cardinality of a query

ing indexing concepts. If we ask for a concept which is equivalent to an indexing one, we get the exact cardinality. If we ask for a concept which is totally unrelated to existing indexing concepts, i.e. there are no subsuming, no subsumed, and no disjoint ones, we will get a lower bound of 0 and an upper bound equal to the number of indexed instances. This means no information at all from the index. Typically, one should get something in between, some partial information.

The second phase additionally utilizes the actual extensions of indexing concepts also stored in the index. This generally results in much better cardinality estimates at the cost of having to load the instances, computing intersections and unions, etc. In case the query is a combination of indexing concepts, its exact extension (and cardinality) can be computed.

Otherwise there is a remaining set of candidates, the individuals for which the query is not known to be either true or false. In this case the index alone does not contain enough information to determine the extension of the query, and the third phase must be entered. For each candidate instance the original description must be accessed and explicitly tested against the query. After this has been done, the user can choose to declare the query as a new indexing concept, making the index more dense at that particular point in the semantic space.

4 Concluding Remarks

This semantic indexing mechanism is crucially dependent on reasoning with descriptions as provided by terminological systems. The indexing elements are potentially complex descriptions logically related by subsumption and disjointness. Note that incomplete algorithms for computing subsumption are not disastrous for indexing: they will simply result in a less informed, suboptimal index.

Compared with standard value-based indexes, this results in the following characteristics:
(1) A semantic index is inherently multidimensional since any combination of properties cast into a DL concept (i.e. an arbitrary query) can serve as an indexing element.
(2) As a structured concept the indexing elements are not just attribute values, but can be based on complex descriptions of related individuals.
(3) A semantic index as a whole is highly adaptable to patterns of usage. Indexing concepts can be added or removed at will, making it very dense and precise w.r.t to interesting sets of individuals, or very sparse in other, less interesting areas.
(4) Since the index is actually a set of partial descriptions for the indexed instances, lots of information (such as cardinality estimates) can be drawn from the index alone without accessing (possibly remote) individual descriptions at all.

These properties may turn out useful for building local information servers which cache information at various levels of completeness, depending on usage patterns.

References

[Hoppe et al., 1993] Hoppe, Th., Kindermann, C., Quantz, J.J., Schmiedel, A., and Fischer, M., BACK V5 Tutorial and Manual. KIT Report 100, Department of Computer Science, Technische Universität Berlin, Berlin, Germany, March 1993.

[Schmiedel, 1993] Schmiedel, A., Persistent Maintenance of Object Descriptions using BACK. KIT Report 112, Department of Computer Science, Technische Universität Berlin, Berlin, Germany, November 1993.