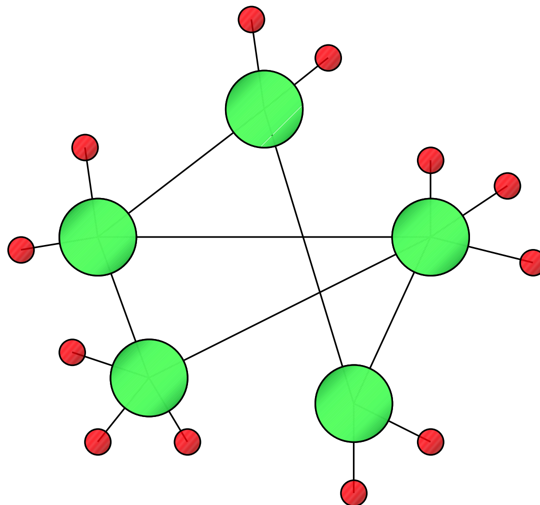
Project “Pepper”

Peer-to-Peer Architectures for
Federated Search of
Complex Digital Libraries

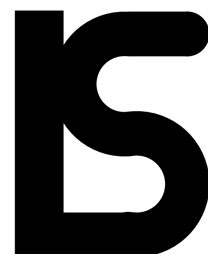
A joint DFG/NSF project



Inside PIRE: An extensible, open-source IR engine based on probabilistic logics

Henrik Nottelmann

Version 1, 2005-03-08



Contents

1 Introduction **4**

2 Probabilistic predicate logics **6**

2.1 Datalog 6

2.2 Probabilistic Datalog 7

2.3 Probabilistic Datalog++ 8

2.3.1 Aggregation operators 8

2.3.2 Computing probabilities 9

3 Document indexing **11**

3.1 Data Types and operators 11

3.2 Supported data types and operators in PIRE 12

3.2.1 Data type “Text” 12

3.2.2 Data type “Name” 13

3.2.3 Data type “Number” 13

3.2.4 Data type “Year” 13

3.3 Schemas, documents and indexing weights 13

3.4 PDatalog++ rules for indexing 14

3.4.1 Relations 14

3.4.2 Rules 14

4 Document retrieval **16**

4.1 Queries 16

4.2 Uncertain Inference 17

4.2.1 Weighted sums 17

4.2.2 Boolean-style queries 18

4.3 Probabilities of relevance 18

4.3.1 Linear functions 19

4.3.2 Logistic functions 19

4.3.3 Maximum normalisation 20

4.4 Mapping functions for PIRE operators 21

4.4.1 Data type “Text” 21

4.4.2 Data type “Name” . . . 21
4.4.3 Data types “Number”, “Year” . . . 21
4.5 PDatalog++ rules for retrieval . . . 22
4.5.1 Weighted sum queries . . . 22
4.5.2 Boolean-style queries . . . 23

5 PDatalog++ engine **24**
5.1 Relations, facts and rules . . . 24
5.1.1 Facts and rules . . . 24
5.1.2 Expressions . . . 25
5.1.3 Relations . . . 26
5.1.4 The relation base . . . 26
5.2 Formatting pDatalog++ to SQL . . . 27
5.2.1 Abstract SQL statements . . . 28
5.2.2 SQL formatter . . . 28
5.2.3 SQL formatter for pDatalog++ . . . 30
5.2.4 DB objects . . . 30
5.2.5 Examples for the formatting process . . . 30

6 PIRE implementation **34**
6.1 Methods for handling an index . . . 34
6.2 Methods for describing data types . . . 37
6.2.1 Example: `AbstractDT` . . . 38
6.2.2 Example: `TextDT` . . . 38
6.2.3 Example: `NumbertDT` . . . 39
6.2.4 Example: `YearDT` . . . 39
6.2.5 Example: `NameDT` . . . 39
6.3 Method for the IR engine . . . 39

7 Retrieval of XML documents **42**
7.1 Data structures . . . 42
7.1.1 Documents . . . 42
7.1.2 Schemas . . . 42
7.1.3 Queries . . . 45
7.2 Information retrieval with XML documents . . . 45
7.2.1 Document retrieval . . . 45
7.2.2 Document indexing . . . 46
7.2.3 XML document retrieval with PIRE . . . 47
7.3 The Indexer program . . . 47
7.3.1 Installation . . . 47
7.3.2 Usage . . . 47

8 Conclusion and outlook **49**
8.1 Conclusion 49
8.2 Outlook and further extensions 49
8.2.1 PIRE tuning 49
8.2.2 Extensional and intensional semantics 50
8.2.3 Language models 50
8.2.4 External knowledge 51
8.2.5 Ontologies and the Semantic Web 51
8.2.6 Sophisticated XML retrieval 52

List of figures **52**

List of tables **53**

Chapter 1

Introduction

This report describes PIRE, a probabilistic IR engine on a theoretically founded basis. This system has been developed as a side-product in the Pepper project.

Both document retrieval (the domain of PIRE) as well as resource selection (a closely related area) require IR engine which can easily be extended by adding new data types, indexing weights and retrieval functions. This can be best accomplished by employing a descriptive definition of the indexing and retrieval functionality, at no program code modification and re-compiling is required. Another advantage is that other IR engines could employ the same definitions. An implementation of the IR engine in the popular Java programming language would ease intregration in wide range of applications.

PIRE combines predicate logic with probability theory for defining the indexing and retrieval strategy. This unique features makes it very easy to extended PIRE new application areas. New data types or operators, weighting schemes and retrieval functions are be integrated by a new set of logical rules. Currently PIRE adopts the paradigm of Uncertain Inference and provides weighted-sum and Boolean-style queries. Optionally it can compute probabilities of relevance. Besides some glue code and a small amount of document preprocessing like splitting a document value into tokens (for which a variety of existing classes can be used), no program code has to be implemented.

In its current form, PIRE also employs this probabilistic logic for the actual indexing and retrieval tasks, which are processed in relation database systems. This can be an external database management system which is running as a server or an embedded database. In the near future, we will extend PIRE so that it can include any external knowledge into the retrieval process.

PIRE itself is entirely implemented in Java. It can be easily integrated into other applications. In the Daffodil federated Digital Library system, it already carries out the retrieval task for locally available collections as well as for ranking the merged result list.

In the last decades, a large number of experimental and productive IR engines have been developed. To our knowledge, however, none of these is completely comparable with PIRE.

One of the most prominent, and one of the oldest retrieval systems is SMART [27, 3], ranging back to the early 70s. SMART is based on the vector-space model with tf.idf indexing weights.

Lemur [1] is developed jointly by UMass and CMU [23]. This high-performance IR engine is written in C++. It supports different retrieval approaches like TF.IDF, Okapi BM25, INQUERY, language models with KL-divergence and CORI for distributed retrieval (integrating query-based sampling, resource selection and result merging). Similar to SMART and most other IR engines, Lemur only supports unstructured documents.

HyREX [2] is an XML retrieval engine which supports XIRQL [12], and thus different data types and operators. Structured documents in XML are handled in a natural way.

This report is structured as follows: The next chapter introduces the probabilistic logic which is used in

[1] http://www-2.cs.cmu.edu/~lemur

PIRE. Chapter 3 describes the document model (including data types and operators), the concept of indexing weights, the data types and operators currently provided by PIRE, and how logical rules are used for indexing. The retrieval part, including the definition of queries, the retrieval model (computation of retrieval status values and probabilities of relevance) and how rules can be used for retrieval, is explained in chapter 4. The following chapters 5 and 6 present implementation details of the components processing the logical rules, and of the IR part of PIRE. Chapter 7 describes the XML extension of PIRE which partial support for XIRQL queries. The report closes with a summary and an outlook into future extensions.

Chapter 2

Probabilistic predicate logics

This chapter describes (probabilistic) Datalog and presents its extension, called pDatalog++.

2.1 Datalog

Datalog [28] is a variant of predicate logic based on function-free Horn clauses. An atom $p(t_1,\ldots,t_n) = p(\bar{t})$ is formed by a n-ary predicate p and terms $\bar{t}$ (constants or variables for constants). A literal is either an atom $p(\bar{t})$ (positive literal) or its negation $\neg p(\bar{t})$ (negative literal). A clause $\{\neg p_1(\bar{t}), \neg q(\bar{t}), r(\bar{t})\}$ is a set of literals with one positive literal. It can be seen as a disjunction $\neg p(\bar{t}) \vee \neg q(\bar{t}) \vee r(\bar{t})$ or the equivalent rule $r(\bar{t}) \leftarrow p(\bar{t}), q(\bar{t})$. Here, the positive literal r denotes the head of the rule, the negative literals p and q form the rule body. Facts are rules with empty body.

In the remainder, we use a more technical notation for Datalog rules, which also PIRE uses internally. In particular, variables start with an upper-case letter, and constants with a lowercase letter. In addition, negated literals start with an exclamation mark. E.g., the fact that Jo is parent of Mary, that Jo is a man, and that fathers are male parents, can be expressed by:

```
parent(jo,mary).
male(jo).
father(X,Y) :- parent(X,Y) & male(X).
```

The last rule denotes that `father(x,y)` is true for two constants `x` and `y` if both `parent(x,y)` and `male(x)` are true.

An interpretation contains all facts which are considered to be true. A model of a Datalog program is an interpretation which is consistent with the given facts and rules. The semantics are defined by well-founded models [29], which are based on the notion of the greatest unfounded set. Given a partial interpretation of a program, this is the maximum set of ground literals that can be assumed to be false. As a consequence, every Datalog program has at most one model.

Negation is allowed in Datalog as long as the program is modularly stratified [26] (in contrast to Prolog). In contrast to global stratification, modular stratification is formulated w.r.t. the instantiation of a program for its Herbrand universe. The program is modularly stratified if there is an assignment of ordinal levels to ground atoms such that whenever a ground atom appears negatively in the body of a rule, the ground atom in the head of that rule is of strictly higher level, and whenever a ground atom appears positively in the body of a rule, the ground atom in the head has at least that level. In other words, no ground fact is allowed to depend negatively on itself.

2.2 Probabilistic Datalog

In probabilistic Datalog (pDatalog for short) [10], every fact or rule has attached a probabilistic weight α, prefixed to the fact or rule:

```
0.5 male(X) :- person(X).
0.8 person(ed).
```

A weight $\alpha = 1$ can be omitted. In that case the rule is called deterministic.

The intended meaning of a rule αr is that "the probability that any instantiation of rule r is true is α". This does not mean that with a probability of α the rule r is true; the probability refers to the instantiaton of the rule (where all variables are replaced by constants). Thus, the preceding example pDatalog program expresses the fact a person is male with a probability of 50%, and that Ed is a person with probability of 0.8. Thus, $Pr(\texttt{male}(\texttt{ed})) = 0.8 \cdot 0.5 = 0.4$.

Formally, an interpretation structure in pDatalog is a tuple $\mathscr{I} = (\mathcal{W}, \mu)$, where $\mathcal{W}$ is a set of possible worlds and μ is a probability distribution over $\mathcal{W}$. The possible worlds are defined as follows. Given a pDatalog program P, with $H(P)$ we indicate the ground instantiation of P [1]. Then, the deterministic part of P is the set P_D of instantiated rules in $H(P)$ having weight $\alpha = 1$, while the indeterministic part of P is the set P_I of instantiated rules determined by $P_I = \{r \colon \alpha r \in H(P), \alpha < 1\}$. The set of deterministic programs of P, denoted $D(P)$ is defined as $D(P) = \{P_D \cup Y \colon Y \subseteq P_I\}$, which is the set of classical logic programs formed by the deterministic part of P and any subset of the indeterministic part of P. Finally, a possible world $w \in \mathcal{W}$ is the minimal model [17] of a deterministic program in $D(P)$ and is represented as the set of ground atoms that are true in the minimal model (also called Herbrand model). Now, an interpretation is a tuple $I = (\mathscr{I}, w)$ such that $w \in \mathcal{W}$. The notion of truth w. r. t. an interpretation and a possible world can be defined recursively:

$$
\begin{aligned}
(\mathscr{I}, w) &\models A \text{ iff } A \in w\,, \\
(\mathscr{I}, w) &\models A \leftarrow B_1, \ldots, B_n \text{ iff } (\mathscr{I}, w) \models B_1, \ldots, B_n \Rightarrow (\mathscr{I}, w) \models A\,, \\
(\mathscr{I}, w) &\models \alpha r \text{ iff } \mu(\{w' \in \mathcal{W} \colon (\mathscr{I}, w') \models r\}) = \alpha\,.
\end{aligned}
$$

An interpretation $(\mathscr{I}, w)$ is a model of a pDatalog program P, denoted $(\mathscr{I}, w) \models P$, iff it entails every fact and rule in P:

$$
(\mathscr{I}, w) \models P \text{ iff } (\mathscr{I}, w) \models \alpha r, \text{ for all } \alpha r \in H(P)\,.
$$

In the remainder, given an n-ary atom A for predicate $\bar{A}$ and an interpretation $I = (\mathscr{I}, w)$, with A^I (an instantiation of A w. r. t. the interpretation I) we indicate the set of ground facts $\alpha\bar{A}(c_1, ..., c_n)$, where the ground atom $\bar{A}(c_1, ..., c_n)$ is contained in the world w, and $\mu(\{w' \in \mathcal{W} \colon (\mathscr{I}, w') \models \bar{A}(c_1, ..., c_n)\}) = \alpha$, i.e. $I \models \alpha\bar{A}(c_1, ..., c_n)$. Essentially, A^I is the set of all instantiations of A under I with relative probabilities, i.e. under I, $\bar{A}(c_1, ..., c_n)$ holds with probability α. For ease, we will also represent an interpretation I as a set of ground facts $\{\alpha A \colon I \models \alpha A\}$. In particular, an interpretation may be seen as a pDatalog program.

Finally, given a ground fact αA, and a pDatalog program P, we say that P entails αA, denoted $P \models \alpha A$ iff in all models I of P, $I \models \alpha A$. Given a set of facts F, with say that P entails F, denoted $P \models F$, iff $P \models \alpha A$ for all $\alpha A \in F$.

By default, facts are assumed to be independent; so the probability that two facts are true equals the product of the probabilities of the two facts. In addition, computing the probability of a disjunction requires to use the inclusion-exclusion formula.

As a consequence, the possible worlds in the example are:

[1] The set of all rules that can be obtained by replacing in P the variables with constants appearing in P, i.e. the Herbrand universe.

$$
\begin{aligned}
Pr(W_1) &= 0.2 & W_1 &:= \{\} \, , \\
Pr(W_2) &= 0.4 & W_2 &:= \{\texttt{person(ed)}\} \, , \\
Pr(W_3) &= 0.4 & W_3 &:= \{\texttt{person(ed)}, \texttt{male(ed)}\} \, .
\end{aligned}
$$

The probability of a fact is then computed by summing up the probabilities of all worlds in which the fact is true. Thus, we obtain $Pr(\texttt{male(ed)}) = 0.4$, and $Pr(\texttt{person(ed)}) = 0.4 + 0.4 = 0.8$.

Alternatively, sets of facts (e.g. all tuples in one relation) can be defined to be disjoint, which means that the probability of the conjunction equals zero. In this case, the probability of a disjunction equals the sum of the underlying probabilities. This feature will be heavily used throughout this paper.

Computation of the probabilities is based on the notion of event keys and event expressions, which allow for recognising duplicate or disjoint events when computing a probabilistic weight. Facts and instantiated rules are basic events (identified by a unique event key). Each derived fact is associated with an event expression that is a Boolean combination of the event keys of the underlying basic events. E.g., the event expressions of the subgoals of a rule form a conjunction. If there are multiple rules for the same head, the event expressions corresponding to the rule bodies form a disjunction. The probabilities of derived facts can then be computed based on the event expression and the probabilities associated with the underlying event keys.

2.3 Probabilistic Datalog++

Probabilistic Datalog is not sufficient for PIRE, as it lacks some important concepts for computing an index and for performing retrieval. Thus, we introduce its extension pDatalog++, with the following differences:

- Constants are now numbers (integers or decimals) or strings, and are optionally enclosed in `"..."` or in `'...'` (for constants which contain e.g. whitespace).
- Variables that are never used in another argument of the same rule can be replaced by _ (the underscore). This is a placeholder for a unique variable which will be ignored in the evaluation of the rule. As such, it is syntactic sugar which prevents to introduce variables which are only used once.
- SQL-like aggregation operators are introduced.
- The independence assumption can be replaced by arbitrary functions for computing the probabilities for a rule.

2.3.1 Aggregation operators

Aggregation operators like `sum` are well-known from SQL. They are also extremely useful as built-in "predicates" in pDatalog++.

An aggregation function $f : 2^D \mapsto D$ takes a bag of values in the domain D as input, and returns a single value in D. Currently, the following functions are supported:

$$
\begin{aligned}
\texttt{sum}(S) &= \sum_{s \in S} s, \\
\texttt{count}(S) &= |S|, \\
\texttt{avg}(S) &= \frac{\texttt{sum}(S)}{\texttt{count}(S)}, \\
\texttt{min}(S) &= \min\{s | s \in S\}, \\
\texttt{max}(S) &= \max\{s | s \in S\}.
\end{aligned}
$$

As the names indicate, `sum` computes the sum, `count` computes the cardinality, `avg` computes the mean average of a set, and `min` and `max` return the minimum (and maximum, respectively) value in the set S.

Aggregation functions can be embedded in pDatalog++ by using special constructs ("aggregation operator") which refer to aggregation functions. Aggregation operators have the prototypical form:

$$op(A, Y_1, \ldots, Y_y, \{p(X_1, \ldots, X_x)\}) \ .$$

Here, op denotes the aggregation function and the variable A the aggregation result. Furthermore, the Y_i are variables, p is a predicate, and each variable X_j either equals one of the Y_i, the placeholder _, or #, which denotes the argument containing the values from the domain D which have to be aggregated (thus, the # appears exactly once).

Such a literal defines the following (nameless) relation:

$$\{(a, \bar{y}) | \exists \bar{t} \colon S = \{v | (\bar{y}, \bar{t}, v) \in \texttt{p}\}, S \neq \emptyset, a = \texttt{op}(S).\}$$

Here, $\bar{y} = (y_1, \ldots, y_y)$ is bound to the variables Y_i; one aggregation value $a = a(\bar{y})$ is computed for each of these tuples $\bar{y}$. In addition, $\bar{t}$ stands for the occurrences of the underscore (the free variables), and v denotes the values which are aggregated.

Obviously, this approach ignores any probabilities, and just considers any tuple with a non-zero probability.

This is equivalent to `group by` in SQL, here:

$$\texttt{select } A, Y_1, \ldots, Y_y \texttt{ from } p \texttt{ group by } Y_1, \ldots, Y_y$$

The following real-world examples compute the document length as the sum of the corresponding term frequencies, count all documents containing the term, and compute the average document length:

```
dl(D,DL) :- sum(DL,D,{tf(D,_,#)}).
df(T,DF) :- count(DF,T,{tf(#,T,_)}).
rd('avgdl',AVGDL) :- avg(AVGDL,{dl(_,#)}).
```

2.3.2 Computing probabilities

Probabilistic Datalog is based on an independence assumption of events, so that point probabilities can be computed (without additional assumptions, it would only be possible to compute probability intervals).

When two events e_1 and e_2 are assumed to be independent, then the probability of $e_1 \wedge e_2$ can be computed as the product of the single probabilities: $Pr(e_1 \wedge e_2) = Pr(e_1) \cdot Pr(e_2)$.

In some cases, that is not enough. Thus, pDatalog++ contains the possibility to use arbitrary functions for computing the probability of facts derived by a single rule. These functions can use the probabilities of facts bound by literals, their product, and any variable occurring in the rule.

A probabilistic version of normalised tf.idf can be computed by these rules:

```
tmp_tf(D,T) :- tf(D,T,TF) & dl(D,DL) | (TF/DL).
tmp_idf(T) :- df(T,DF) & numdocs(N) | (&log(N/DF)/&log((N))).
weight(D,T) :- tmp_tf(D,T) & tmp_idf(T) | (PROB1*PROB2).
```

The part after the pipe symbol `|` denotes the function used for computing the probability of derived facts. To simplify parsing, the operators `+`, `-`, `*`, `/`, `%` (modulo n) and `^` (power) are binary, and have to be enclosed in brackets. Thus, $1+2 \cdot 3 \cdot 4$ has to be written as `(1+(2*(3*4)))`.

The first rule computes the TF part, where the variables `TF` and `DL` (bound by the two subgoals) are used for computing the TF-based probability. Similarly, the second rule computes the IDF part. Built-in functions like `log` start with an ampersand (thus, `&log`).

The last rule combines the two probabilities; where `PROB1` refers to the probability of the first literal, and `PROB2` refers to the probability of a tuple bound by the second literal. For simplicity, `PROB1*PROB2` is equivalent to `PROB` (always the product, following an independence assumption). As this is the default, it can be left out. So, these three rules are equivalent:

```
weight(D,T) :- tmp_tf(D,T) & tmp_idf(T) | (PROB1*PROB2).

weight(D,T) :- tmp_tf(D,T) & tmp_idf(T) | (PROB).

weight(D,T) :- tmp_tf(D,T) & tmp_idf(T).
```

Chapter 3

Document indexing

This chapter describes the indexing part of PIRE. First, the document model is introduced (which is very similar to the one proposed in [9]). Then, indexing weights are defined for several operators. Finally, pDatalog++ relations and rules for computing indexing weights are presented.

3.1 Data Types and operators

The key feature of this document model (in contrast e.g. to Fagin [5]) is that it also considers data types and operators.

We first assume a finite set $\mathbf{D}$ of elementary data types. Each data type $d \in \mathbf{D}$ has a domain (set of values) $dom(d)$. We denote the union of all domains as $dom(\mathbf{D}) = \bigcup_{d \in \mathbf{D}} dom(d)$, which will be used as the domain of any interpretation.

For $\mathbf{D}$, we further have a set $\mathbf{O}$ of operators (also called “data type predicates” in the Digital Library field). An operator $o \in \mathbf{O}$ defines (given an interpretation I) a binary relation $o^I \subseteq dom(d_1(o)) \times dom(d_2(o))$ with respect to two datatypes $d_1(o), d_2(o) \in \mathbf{D}$. In most cases, we have $d_1 = d_2$. On the other hand, we can assume an operator *in* comparing a date and a year.

We use a bijective mapping between operators $o \in \mathbf{O}$ and new constants $\hat{o} \in \widehat{\mathbf{O}}$ for a set of constants $\widehat{\mathbf{O}}$, which will be mapped by every interpretation onto itself. Then, these operators are combined in a ternary predicate op which allows for using operator symbols as variables for any interpretation I.

$$\mathtt{op}^I = \bigcup_{o \in \mathbf{O}} \{\hat{o}\} \times o^I.$$

In our scenario, $\mathbf{D}$ contains the data type DOCID which denotes the set of all document ids. There is no operator defined for DOCID, i.e. $\forall o \in \mathbf{O} : d_1(o), d_2(o) \neq \mathtt{DOCID}$.

For simplicity, we do not explicitly distinguish between the operators o and their constants $\hat{o}$, and use the former notation for both of them.

The predicate op will be evaluated using a specific API (called “oracle”). Internally, it can be implemented by program code or by logic programming.

Vagueness of query formulations is one of the key concepts of Information Retrieval. Thus, it is crucial that operators have a probabilistic interpretation (as proposed in [8]). Vagueness is required e.g. when a user is uncertain about the exact publication year of a document or the spelling of an author name. For a specific attribute value the vague operator yields an estimate of the probability that the condition is fulfilled from the user’s point of view — instead of a Boolean value as in DB systems. As a result, each operator and thus also op can be uncertain, i.e. that each tuple t has attached a probabilistic weight. Sometimes, we refer to this probability as

$$o(v_1, v_2) \in [0,1],$$

viewing the uncertain interpretation o^I as a function

$$o : o^I \mapsto [0,1].$$

3.2 Supported data types and operators in PIRE

Similar to [20], PIRE supports several data types.

3.2.1 Data type "Text"

For `Text`, the comparison value of a condition c is a term t. In the following definitions, $tf(t,d)$ is the term frequency (number of times term t occurs in document d), $dl(d)$ denotes the document length (number of terms in document d), *avgdl* the average document length, $|DL|$ the collection size (number of documents), and $df(t)$ the document frequency (number of documents containing term t)

Different operators are considered:

`contains`: This operator is used for experimental compatibility only, and should not be used in production environments. It applies stemming and stop-word removal, and uses a modified BM25 scheme [25] for the indexing weights, which a special kind of the popular $TF \cdot IDF$ weighting scheme (term frequency times inverse document frequency):

$$\texttt{contains}(d,t) = \frac{tf(d,t)}{tf(d,t) + 0.5 + 1.5 \cdot \frac{dl(d)}{avgdl}} \cdot \frac{\log \frac{|DL|}{df(t)}}{\log |DL|}. \tag{3.1}$$

We modified the standard BM25 formula by the normalisation component $1/\log|DL|$ to ensure that indexing weights are always in the closed interval $[0,1]$, and can thus be regarded as a probability. The resulting indexing weights are rather small; but this can be compensated by the mapping functions.

`plain`: This operator is used for experimental compatibility only. It does not employ stemming and stop-word removal.

`stemen`: This operator uses stemming (Porter stemmer for the English language) and stop-word removal. The BM25 weighting scheme is slightly different to `contains`, only $|DL| + 0.5$ instead of $|DL|$ is used. Due to this modified IDF part, it can cope with situations where $df(t) = |DL|$ and thus $\log \frac{|DL|}{df(t)} = 0$:

$$\texttt{stemen}(d,t) = \frac{tf(d,t)}{tf(d,t) + 0.5 + 1.5 \cdot \frac{dl(d)}{avgdl}} \cdot \frac{\log \frac{|DL|+0.5}{df(t)}}{\log |DL| + 0.5}. \tag{3.2}$$

`nostem`: This operator is equivalent to `stemen`, but no stemming is applied.

`stemen_tf`: This operator uses stemming (Porter stemmer for the English language) and stop-word removal. In contrast to `stemen`, normalised term frequencies are employed as indexing weights:

$$\texttt{contains}(d,t) = \frac{tf(d,t)}{dl(d)}. \tag{3.3}$$

These probabilistic indexing weights are derived from maximum likelihood estimations, where a document (actually, the content of an attribute) is considered as a bag of words, and the indexing weight of a term t is the probability that a randomly drawn token is the term t.

nostem_tf: This operator is equivalent to stemen_tf, but no stemming is applied.

3.2.2 Data type "Name"

This data type supports two Boolean operators plainname and soundex:

$$Pr(c \leftarrow d) \in \{0,1\},$$
$$Pr((author, soundex, Jones) \leftarrow (author, Johnson)) = 1.$$

3.2.3 Data type "Number"

The data type Number has the Boolean operators =, <, >, <= and >=, i.e. the indexing weight is in $\{0,1\}$.

When a user is uncertain about the exact publication year of a document and requests documents from the year 1999, a document from the year 2000 might also be relevant (although the probability is lower). Thus, vague operators ∼=, ∼< and ∼> are introduced:

$$\begin{aligned}
\sim>(v_1,v_2) &:= \begin{cases} 1-\frac{v_2-v_1}{v_1} & , \quad v_1 < v_2 \\ 1 & , \quad else \end{cases} , \\
\sim=(v_1,v_2) &:= 1-\left(\frac{v_1-v_2}{v_2}\right)^2 .
\end{aligned}$$

The vague operators are not fully implemented yet.

3.2.4 Data type "Year"

The data type Year is equivalent to Number for now, but intended for years and not for numbers.

3.3 Schemas, documents and indexing weights

Each document in PIRE adheres to a schema, which defines a list of (potentially multi-valued) attributes A_i. This document model is mapped onto our logical framework: A schema $\mathbf{R} = \langle R_1, \ldots, R_n \rangle$ consists of a non-empty finite tuple of binary relation symbols[1]. Each relation symbol R_i has a data types $d_{R_i} \in \mathbf{D}$. Then, for every interpretation $I = (\Delta, \cdot^I)$, a schema instance is a tuple $\mathbf{R}^I = \langle R_1^I, \ldots, R_n^I \rangle$, where each relation symbol R_i is mapped onto a relation instance R_i^I with the correct data types:

$$\mathbf{R}^I \subseteq \texttt{DOCID} \times dom(d_{R_i}).$$

The relations R_i can be uncertain, too, for modelling uncertain knowledge.

Informally, this is the relational model of linear schemas with multi-valued schema attributes. Each attribute A_i is modelled as binary relation R_i, storing tuples of a document id and a value for that attribute. Thus, one can be used by more than one document attribute (e.g. for multiple authors), and attributes can be left out completely (e.g. if no author is known).

The value (second argument) of a relation A_i for a specific document d is denoted by $A_i^I(d)$, abbreviated by $A_i(d)$.

Closely related to attributes and operators are indexing weights. They are the result of applying an operator $o \in \mathbf{O}$ to the content of a document attribute $A_i(d) \in dom(d_I(o))$ and a second value $v \in dom(d_2(o))$. The

[1] In principle, any arty is possible. Binary relation symbols are used to emphasise the document structure

notion of "indexing weight" typically appears in the area of text retrieval, where a document/term pair in the index has assigned a weight (derived from the index). This weight is typically stored in the index for performance reasons. We generalise this idea and call the result of an operator also an indexing weight (even in cases where it is not stored explicitly, e.g. for a data type "Year" and a less-than operator).

3.4 PDatalog++ rules for indexing

This section describes how the indexing process can be carried out via pDatalog++ rules.

3.4.1 Relations

A separate index is created for each attribute and each possible operator for this attribute, allowing for operator-specific indexing weights.

Each index is realised as a couple of relations. The names are constructed following the scheme

`[c]_[a]_[o]_[name].`

Here, `[c]` denotes the name of the collection which is indexed, `[a]` the attribute name, `[o]` the operator name, and `[name]` is the local name of the relation. The relation names are restricted to the local name in the remainder in situations where the "namespace" is clear.

Independent of the operator, each index has a ternary relation `tf` which stores the token frequencies $tf(d,t)$, where the definition of a "token" depends on the operator (e.g. terms for `Text`, a first name or a last name for `Name`, and the complete number for `Year`). This relation is created first, by splitting the document content (using program code which depends on the data type). The deterministic unary relation `docid` contains all document ids. The final indexing weights are stored in a binary relation `weight`, where the indexing weight is the probability of the corresponding fact.

Furthermore, each index has two binary relations `rd` and `idb_rd` for the "resource description". A resource description contains numerical data, e.g. parameters for the mapping functions, which are identified via textual keys. The first relation, `rd`, is set manually (as facts); the other one is defined by rules.

Some data types and operators use additional temporary relations for computing the indexing weights.

3.4.2 Rules

The following example shows how BM25 indexing weights are computed for an attribute `ti` and the operator `contains`.

In a first step, the document length (sum over the term frequencies for a fixed document) and the document frequencies (number of documents containing a fixed token) are computed:

```
coll_ti_contains_dl(D,DL) :- sum(DL,D,{coll_ti_contains_tf(D,_,#)}).

coll_ti_contains_df(T,DF) :- count(DF,T,{coll_ti_contains_tf(#,T,_)}).
```

Then, statistics are computed and stored in the resource descriptions. These statistics contain the average document length and the number of documents in the collection:

```
coll_ti_contains_idb_rd('avgdl',AVGDL) :- avg(AVGDL,{coll_ti_contains_dl(_,#)}).
coll_ti_contains_idb_rd('numdocs',NUMDOCS) :- count(NUMDOCS,{coll_ti_contains_docid(#)}).
```

With these pre-computations, BM25 indexing weights can be computed. For simplicity, the computation is split in two parts (TF and IDF) and the combined:

```
coll_ti_contains_tmp_tf(D,T) :- coll_ti_contains_tf(D,T,TF) &
                                coll_ti_contains_dl(D,DL) &
                                coll_ti_contains_rd('avgdl',AVGDL) |
                                    (TF/(TF+(0.5+(1.5*(DL/AVGDL))))).

coll_ti_contains_tmp_idf(T) :- coll_ti_contains_df(T,DF) &
                               coll_ti_contains_idb_rd('numdocs',NUMDOCS) |
                                    (&log(((NUMDOCS)/DF))/&log((NUMDOCS))).

coll_ti_contains_weight(D,T) :- coll_ti_contains_tmp_tf(D,T) &
                                coll_ti_contains_tmp_idf(T).
```

Chapter 4

Document retrieval

This chapter describes the retrieval part of PIRE. First, query syntax and semantics as well as mapping functions are defined. Then, pDatalog++ rules for actually performing retrieval are presented.

4.1 Queries

An abstract syntax is used for expressing queries, which will later be translated into sets of pDatalog++ rules. Each query refers to one schema **R**, and returns document ids.

The basic construct is a query condition. A query condition c consists of a schema attribute $A(c) \in \mathbf{R}$, an operator $o(c)$ and a comparison value $v(c)$. Of course, the data types must match, i.e. $d_{A(c)} = d_1(o(c))$ and $v(c) \in d_2(o(c))$. The abstract syntax of a query condition has the form `attribute operator value`, e.g.

$$\texttt{author soundex "nottelmann''}.$$

Queries are formed by conditions. By abuse of notation, $c \in q$ denotes that condition c occurs in query q. The result of a query is a list of document ids with probabilistic weight. Two different types of queries are supported: weighted sums and Boolean-style queries.

Weighted sums have the form $\texttt{wsum}(w(c_1)\ c_1, \ldots, w(c_n)\ c_n)$, where the c_i are conditions, and the $w(c_i) \in [0,1]$ are probabilistic weights representing the importance of the conditions. The idea is that the comparison weight for a document w. r. t. the query is the sum of the comparison weights w. r. t. the conditions c_i, weighted by the $w(c_i)$; the precise semantics are described in section 4.2.

Boolean-style queries use the Boolean operators `and` (conjunction) and `or` (disjunction) as connectors of conditions. This example query will return documents published in 2003 by a person whose name sounds like "Doe":

$$(\texttt{year} >= 2003)\ \texttt{and}\ (\texttt{author soundex}\ ''\texttt{doe}'').$$

These queries can easily be transformed into pDatalog++ rules with common head which define a unary predicate q with $q^I \subseteq \texttt{DOCID}$. The literals of these rules refer to the relation symbols defined in $\mathbf{R} \cup \{\texttt{op}\}$. The set $q^{\mathbf{R}^I}$ of answers for query q with respect to schema $\mathbf{R}^I$ contains exactly all document ids which satisfy the query.

Each condition c_i is transformed into the conjunction of two logical literals, one for the attribute alone, the other one for the operator. E.g., the condition from above is transformed into

$$\texttt{author(D,X)} \wedge \texttt{op(soundex,X,"nottelmann'')}.$$

A weighted sum query with n conditions c_i will be transformed in n pDatalog++ rules, one for each condition c_i. The weight of that rule equals the condition weight $w(c_i)$, and disjointness for the rules is assumed (so that the probabilities are summed up).

The example query is then transformed into these rules:

```
q(D) :- year(D) & op(>=,X,2003).
q(D) :- author(D) & op(soundex,X,``nottelmann'').
```

Boolean-style queries will be transformed into disjunctive form, i.e. it is a disjunction of conjunctions. Then, the transformation process is straight forward, as each pDatalog++ rule is considered as a conjunction, and multiple pDatalog++ rules for the same head predicate are considered as a disjunction. Thus, a disjunction of n conjunctions is transformed into n pDatalog++ rules. Each conjunction of m conditions is transformed into one rule with $2m$ literals.

Note, that PIRE uses a different form of pDatalog++ rules for queries, which directly use the relations in the index instead of the generic predicate op.

4.2 Uncertain Inference

Probabilistic Datalog adopts Rijsbergen's view of information retrieval as uncertain inference [30], a variant of the logical view on databases, where queries and document contents are treated as logical formulae, and a database only returns those documents d which logically imply the query q, i.e. it proves $q \leftarrow d$. One big advantage of logical models is that it easy to integrate external knowledge like a thesaurus or an ontology.

For considering the intrinsic uncertainty of information retrieval, Rijsbergen interpreted probabilistic IR as estimating the probability $Pr(q \leftarrow d)$ that the document logically implies the query. In the classical view, the formula $q \leftarrow d$ can only be true or false, i.e. $Pr(q \leftarrow d) \in \{0,1\}$. For Rijsbergen, this inference process is uncertain, thus every value in $[0,1]$ is possible.

In practice, this probability cannot be considered in the traditional sense of logical databases, where $q \leftarrow d \equiv \neg d \wedge q$. Rather, this probability has to be seen a the conditional probability $Pr(q \leftarrow d) = Pr(q|d)$.

The document model has already been defined in section 3.3: A document d contains a set of document attributes, and each of these attributes is modelled as a (potentially uncertain) binary relation. A query is either a weighted sum query or a Boolean-style query. In both cases, the query can be expressed in an abstract form or as pDatalog++ rules (where each condition is modelled as a pair of predicates, one for the attribute, the other one for the operator). Each query consists of conditions $c_i = (A(c_i), o(c_i), v(c_i))$ (attribute, operator and comparison value). In the context of uncertain inference, the probability $Pr(q \leftarrow d)$ that a document implies the query q should be computed based on the probabilities that a query implies the conditions:

$$Pr(c_i \leftarrow d) := o(c_i)(A(c_i)(d), v(c_i)).$$

4.2.1 Weighted sums

Weighted sum queries consist of a set of conditions c_i with associated weights $w(c_i) \in [0,1]$. These weights are seen as the probability $Pr(q \leftarrow c)$.

The underlying facts are defined to be disjoint, and the widely used linear retrieval function [32] is employed for computing the probability of inference:

$$Pr(q \leftarrow d) = \sum_{c \in q} Pr(q \leftarrow c) \cdot Pr(c \leftarrow d). \tag{4.1}$$

The condition weight $Pr(q \leftarrow c)$ defines the importance of the condition. It can be specified manually by the user, or derived automatically, e.g. using the term frequency in a query containing a paragraph of text.

4.2.2 Boolean-style queries

The paradigm of uncertain inference also allows for Boolean-style queries.

As for pDatalog++, an independence assumption is used for the conditions. As a result, the probabilities derived for different conditions have to be multiplied:

$$Pr((c_1 \wedge c_2) \leftarrow d) = Pr(c_1 \leftarrow d) \cdot Pr(c_2 \leftarrow d). \tag{4.2}$$

For the $\vee$ connectors, the de Morgan rules have to be used:

$$\begin{aligned}
Pr((c_1 \vee c_2) \leftarrow d) &= Pr(\neg(\neg((c_1 \leftarrow d) \wedge \neg(c_2 \leftarrow d)))) && (4.3)\\
&= 1 - Pr(\neg((c_1 \leftarrow d) \wedge \neg(c_2 \leftarrow d))) && (4.4)\\
&= 1 - (Pr(\neg(c_1 \leftarrow d)) \cdot Pr(\neg(c_2 \leftarrow d))) && (4.5)\\
&= 1 - ((1 - Pr(c_1 \leftarrow d)) \cdot (1 - Pr(c_2 \leftarrow d))) && (4.6)\\
&= 1 - (1 - Pr(c_1 \leftarrow d) - Pr(c_2 \leftarrow d) + Pr(c_1 \leftarrow d) \cdot Pr(c_2 \leftarrow d)) && (4.7)\\
&= Pr(c_1 \leftarrow d) + Pr(c_2 \leftarrow d) - Pr(c_1 \leftarrow d) \cdot Pr(c_2 \leftarrow d)). && (4.8)
\end{aligned}$$

In general, $c_1 \vee \cdots \vee c_n$ has to be evaluated using the inclusion-exclusion formula:

$$Pr(c_1 \vee \cdots \vee c_n) = \sum_{\emptyset \subset S \subseteq \{1,\ldots,n\}} (-1)^{|S|+1} \cdot \prod_{i \in S} Pr(c_i \leftarrow d). \tag{4.9}$$

4.3 Probabilities of relevance

Information Retrieval is based on the principles of uncertainty and vagueness. Thus, an IR engines does not only return a set of documents, but also "retrieval status values" $RSV(d,q)$ which describe the similarity of document d with the query q. These RSVs can then be used to rank the documents which reflects the system's confidence that the documents are good answers to the query. Different IR methods compute RSVs in different ways and different scales (e.g. $\{0,1\}$, $[0,1]$, $\mathbb{R}$).

As mentioned before, the Probability Ranking Principle (PRP) [24] gives a theoretical justification for a special case: Optimum retrieval (defined w. r. t. document representations) is given if the documents are ranked according to the probability $Pr(\text{rel}|d,q)$ that document d is relevant to a user query q ("probability of relevance").

For ad-hoc retrieval, it is sufficient to rely on the retrieval status values if the probabilities of relevance are monotonically increasing with the RSVs. Thus, little effort has been spent on approximating the relationship between retrieval status values and probabilities of relevance. Complex applications like estimating retrieval quality for resource selection (see next chapter) are based on the actual numbers (or their approximations) of the probabilities of relevance.

The basic idea is to introduce a "mapping function" for transforming $RSV(d,q)$ onto the probability $Pr(\text{rel}|d,q)$:

$$f : \mathbb{R} \mapsto [0,1], \quad f(RSV(d,q)) \approx Pr(\text{rel}|q,d). \tag{4.10}$$

This idea can be applied to a large number of retrieval functions. Here, the probabilities of inference are considered:

$$RSV(d,q) := Pr(q \leftarrow d). \tag{4.11}$$

We consider different data types and operators, which all have their own mapping function. Thus, we split the query q into sub-queries $q_{A,\hat{o}}$ which only refer to one attributes A and operator $\hat{o}$ (there can be several sub-queries referring to the same attribute/operator pair, depending on the overall query structure). The retrieval status values $RSV(d, q_{A,\hat{o}})$ for this sub-query are then converted into probabilities of relevance using an attribute- and operator-specific mapping function:

$$f_{A,\hat{o}} : [0,1] \mapsto [0,1],\ \ f_{A,\hat{o}}(RSV(d, q_{A,\hat{o}})) \approx Pr(\text{rel}|q_{A,\hat{o}}, d). \tag{4.12}$$

The overall probability of relevance is then derived by combining the probabilities of relevance for the sub-queries according to the original query structure.

We consider four different mapping functions: Pure linear functions, affine linear functions, logistic functions, and the maximum normalisation.

4.3.1 Linear functions

Obviously, the probability of relevance should be monotonously increasing with the retrieval status value (assuming that the retrieval engine provides a proper ranking). A first and very simple approximation of this relationship is a pure linear function, where the probability that a document is relevant is proportional to the retrieval status value:

$$f_{lin}(x) := c_1 \cdot x. \tag{4.13}$$

We can control the mapping function in a more flexible way by adding a second degree of freedom, arriving at an affine linear function:

$$f_{alin}(x) := c_0 + c_1 \cdot x. \tag{4.14}$$

Both linear functions have the same drawback. They do not ensure that the results are between 0 and 1 in the general case of $c_0, c_1 \in \mathbb{R}$. In other words, the result cannot necessarily be regarded as a probability.

However, linear mapping functions can be justified in the context of uncertain inference [31].

4.3.2 Logistic functions

In an ideal situation, exactly the documents in the ranks $1, \ldots, l$ are relevant, and the documents in the remaining ranks $l+1, \ldots$ are irrelevant. Let a be the RSV of the documents in rank l (i.e., the lowest RSV of any relevant documents). Thus, the relationship function should be a step function

$$f : \mathbb{R} \to [0,1],\ f(x) := \begin{cases} 1 & ; \quad x \geq a, \\ 0 & ; \quad x < a \end{cases}, \tag{4.15}$$

where x is the document's RSV.

Obviously, no information retrieval system can ensure this requirement; in general some of the top-ranked documents are irrelevant, and some of the lower-ranked documents are relevant. But, less documents with lower RSV should be relevant than documents with higher weights. In other words, the probability that any arbitrary document is relevant should decrease with decreasing RSV (see figure 4.1).

For modelling this characteristic, we want a continuous function f which approximates the discrete step function (4.15). Obviously, the pure and affine linear functions (4.14,4.13) are not appropriate.

Logistic functions have been used in different application areas within IR for quite some time, e.g. for text categorisation [13] or retrieval functions [4, 15] (logistic variant of the model proposed in [11]). As the shape of a logistic function can be seen as a continuous approximation of the discrete step function, it is a natural candidate for a mapping function.

In this work, a logistic mapping function with two degrees of freedom b_0 and b_1 is used [6, 7]:

$$f_{log} : \mathbb{R} \to [0,1],\ f_{log}(x) := \frac{\exp(b_0 + b_1 \cdot x)}{1 + \exp(b_0 + b_1 \cdot x)}. \tag{4.16}$$

Figure 4.1: Ideal and real case

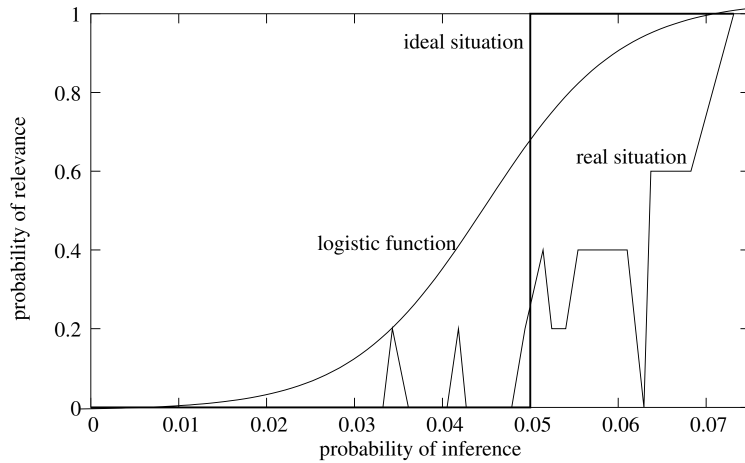
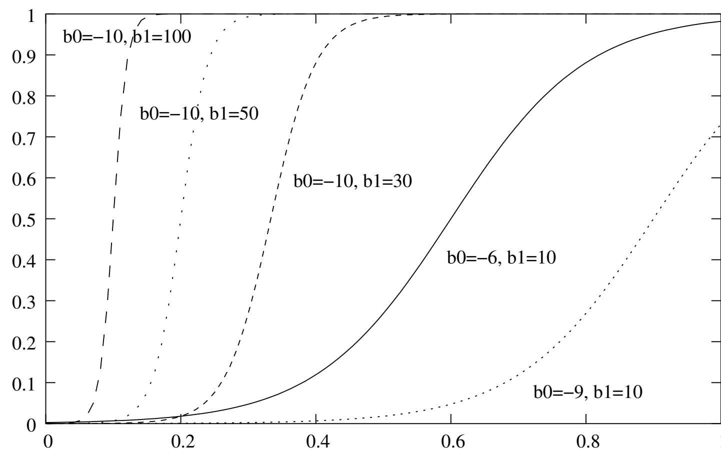


Figure 4.2 depicts some logistic functions with different parameters. One of the nice properties of logistic functions is that the result is always in $[0,1]$. In addition, a large variety of curves can be obtained by varying b_0 and b_1. The curve can be moved along the x axis by varying b_0; the slope can be adjusted by varying b_1.

Figure 4.2: Example logistic functions



4.3.3 Maximum normalisation

Another simple mapping function is to divide each value (which is the RSV of a document) by the maximum value (i.e., the maximum RSVs, which is the RSV of the top-ranked document):

$$f_{max}(x) = \frac{x}{\max x'}. \tag{4.17}$$

As a consequence, the top-ranked document has a retrieval status value of 1.

4.4 Mapping functions for PIRE operators

Several different mapping functions, sometimes in combination, are used for the PIRE operators.

4.4.1 Data type "Text"

For `Text`, the comparison value of a condition c is a term t. In the following definitions, $tf(t,d)$ is the term frequency (number of times term t occurs in document d), $dl(d)$ denotes the document length (number of terms in document d), *avgdl* the average document length, $|DL|$ the collection size (number of documents), and $df(t)$ the document frequency (number of documents containing term t)

Different operators are considered:

`contains`, `plain`: It is possible to enable a first normalisation which computes

$$\frac{Pr(q \leftarrow d)}{\max_{d'} RSV(d',q)}. \tag{4.18}$$

A logistic mapping function with the defaults $b_0 = -4$, $b_1 = 12$ is used; this can be completely disabled. Also, other parameters can be set in the resource descriptions (with the keys `b0` and `b1`).

`stemen`, `nostem`: By default, a first normalisation which computes

$$\frac{Pr(q \leftarrow d)}{\max_{d'} RSV(d',q)} \tag{4.19}$$

is performed; this can also be disabled.

In addition, another normalisation with a logistic mapping function can be enabled. The default parameters are $b_0 = -4$ and $b_1 = 12$, other parameters can be set as well in the resource description.

`stemen_tf`, `nostem_tf`: The identity function is used for the mapping function:

$$f_{\hat{\texttt{contains}}} \equiv id \text{ , } f_{\hat{\texttt{contains}}}(x) := x. \tag{4.20}$$

4.4.2 Data type "Name"

For Boolean operators like `plainname` or `soundex`, the identity function is a natural choice for the mapping function:

$$f_{\hat{o}} \equiv id \text{ , } f_{\hat{o}}(x) := x.$$

4.4.3 Data types "Number", "Year"

As for the data type `Name`, we use the identity mapping function for the Boolean operators `=`, `<`, `>`, `<=` and `>=`.

Logistic mapping functions are used for the vague operators `~<`, `~>` and `~=`.

4.5 PDatalog++ rules for retrieval

This section describes how the retrieval process can be carried out via pDatalog++ rules.

The basic idea is to split the queries into sub-queries (the actual splitting depends on the query type). For each sub-query q_i (with the sub-query id $i \geq 0$), a temporary predicate `[c]_[a]_[o]_rsv[query id]_[i]`, defined by one or more rules, stores the RSV $Pr(q_i \leftarrow d)$. The corresponding temporary predicate `[c]_[a]_[o]_prob[query id]_[i]` is used for computing the probability of relevance $Pr(\text{rel}|q_i, d)$ from the RSV $Pr(q_i \leftarrow d)$.

In a final step, the probabilities from the relations `[c]_[a]_[o]_prob[query id]_[i]` are combined in a relation `[c]_prob[query id]`.

4.5.1 Weighted sum queries

First, we present pDatalog++ rules for evaluating weighted sum queries.

As an example, we consider the query:

```
wsum(0.1 ti contains ``hello'',0.3 ti contains ``world'',0.6 ab contains ``java'')
```

Each sub-query $q_i := q_{A,\hat{o}}$ contain exactly all conditions for one attribute/operator combination, where the condition weights are normalised so that the sum of all condition weights in a sub-query equals one. Thus, we have two sub-queries:

```
q1 := wsum(0.25 ti contains ``hello'',0.75 ti contains ``world'')
q2 := wsum(1,ab contains ``java'')
```

This can easily be computed by the following pDatalog++ rules. For simplicity, we use f_{max} as a mapping function, where 0.25 and 0.33 are arbitrarily chosen maximum RSV just for illustrating the retrieval process:

```
coll_ti_contains_rsv42_0(D) :- coll_ti_contains_weight(D,'hello') | ((0.1/0.4)*PROB).
coll_ti_contains_rsv42_0(D) :- coll_ti_contains_weight(D,'world') | ((0.3/0.4)*PROB).
coll_ti_contains_prob42_0(D) :- coll_ti_contains_rsv42_0(D) | (PROB/0.25).

coll_ab_contains_rsv42_1(D) :- coll_ab_contains_weight(D,'java') | ((0.6/0.6)*PROB).
coll_ab_contains_prob42_1(D) :- coll_ab_contains_rsv42_1(D) | (PROB/0.33).
```

The relation `[c]_[a]_[o]_prob[query id]_[i]` contains the probability of relevance $Pr(\text{rel}|q_i, d)$. These probabilities of relevance for the sub-queries are then combined by a weighted sum

$$Pr(\text{rel}|q,d) = \sum_i \Big(\sum_{c \in q_i} Pr(q \leftarrow c)\Big) \cdot Pr(\text{rel}|q_i,d). \tag{4.21}$$

In this case, we compute $0.4 \cdot Pr(\text{rel}|q_1,d) + 0.6 \cdot Pr(\text{rel}|q_2,d)$.

This can easily be converted into two pDatalog++ rules, assuming disjointness of the facts in `[c]_[a]_[o]_prob[query id]_[j]` and `[c]_[a]_[o]_prob[query id]_[i]` for $i \neq j$; the result is the final relation `[c]_prob[query id]`:

```
coll_prob42(D) :- coll_ti_contains_prob42_0(D) | (0.4*PROB).
coll_prob42(D) :- coll_ab_contains_prob42_1(D) | (0.6*PROB).
```

4.5.2 Boolean-style queries

This section describes pDatalog++ rules for evaluating Boolean-style queries.

As an example,we consider the query:

```
(ti contains ``hello'' and ti contains ``world'') or ab contains ``java''
```

Each condition forms a sub-query:

```
coll_ti_contains_rsv42_0(D) :- coll_ti_contains_weight(D,'hello').
coll_ti_contains_prob42_0(D) :- coll_ti_contains_rsv42_0(D) | (PROB/0.2).

coll_ti_contains_rsv42_1(D) :- coll_ti_contains_weight(D,'world').
coll_ti_contains_prob42_1(D) :- coll_ti_contains_rsv42_1(D) | (PROB/0.4).

coll_ab_contains_rsv42_2(D) :- coll_ab_contains_weight(D,'java').
coll_ab_contains_prob42_2(D) :- coll_ab_contains_rsv42_2(D) | (PROB/0.1).
```

Again, `PROB/0.2` denotes the simply mapping function where 0.2 is the maximum RSV for this sub-query.

Then, the query is transformed into disjunctive form (the example query already is in disjunctive form). Following the standard approach of mapping disjunctive forms into pDatalog++, each conjunction is converted into one rule which combines the results for the single conditions of the conjunction:

```
coll_prob42(D) :- coll_ti_contains_prob42_0(D) & coll_ti_contains_prob42_1(D).
coll_prob42(D) :- coll_ab_contains_prob42_2(D).
```

Chapter 5

PDatalog++ engine

This pDatalog++ implementation follows the approach in [19] and operates on a relational database, i.e. pDatalog++ relations are stored in database tables, and rules are transformed into one or more SQL statements. A relational database is a good choice as it is fast, flexible, and shares the concepts with predicate logics (tables for relations). The major difference is that each database relation needs an additional column for storing the probability of the tuples. Figure 5.1 depicts the overall architecture of the pDatalog++ engine.

5.1 Relations, facts and rules

This section explains the data structures for facts and rules, for relations, and the handling of all relations (e.g. adding facts or computing rules).

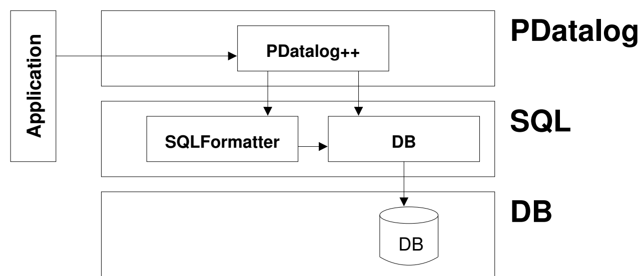
5.1.1 Facts and rules

The classes `Rule` and its sub-class `Fact` store a rule or a fact, respectively, where a fact is simply a rule with empty body. The head is stored as a `Literal` instance, the body is a list of `Literal` instances, and the function for computing the probabilities is stored as an `Expression` instance (see below). A literal is given by its name, an array of expressions, and a flag indicating if the literal is positive or negated.

Expressions are used for storing e.g. variables (class `Variable`) and constants (class `Constant`) in literals, but also in mathematical functions in a recursive way. Expression classes can be found in the packages `de.unidu.is.expression` and in `de.unidu.is.pdatalog.ds`.

The easiest way to construct a fact or a rule object is not to create it manually, but to parse a string, employ-

Figure 5.1: PDatalog++ engine architecture



ing the `de.unidu.is.pdatalog.parser.Parser` class. Note that the current implementation requires a lot of round brackets in mathematical expressions, as described in section 2.3.2.

5.1.2 Expressions

Expressions are used here for defining the function for computing the probability of facts derived by a single rule, and as the literal arguments:

DBColumnExpression denotes a database table column.

DBColExpression is equivalent to `DBColumnExpression` with the standard names of pDatalog++ fact arguments `arg0`, `arg1` etc.

DBProbExpression denotes the database table column which contains the probabilities of pDatalog++ facts.

DifferenceExpression denotes the difference of exactly two other expressions.

EqualsExpression represents the test for equality for two expressions. It is used mainly for internal purpose, e.g. for the SQL formatter (see section 5.2).

FractionExpression denotes the result of dividing one expression by another one.

FunctionExpression represents the result of applying a named function onto another expression.

LiteralExpression represents a literal (for the aggregation operators).

PlainExpression stores a plain string.

ProbExpression represents the resulting probability, i.e. `PROB`.

ProductExpression denotes the product of exactly two other expressions.

ProductNExpression denotes the product of a list of other expressions.

Str2NumExpression presents the function converting a string value into a double number. This is used e.g. for the arguments of a literal.

StringExpression stores a string in quotation marks.

SumExpression denotes the sum of exactly two other expressions.

SumNExpression denotes the sum of a list of other expressions.

VariableExpression represents a variable.

Constant is equivalent to `StringExpression` and has to be used in literals for constants.

Variable is equivalent to `VariableExpression` and has to be used in literals for variables.

Expressions have to support to methods:

substitute(Map) has to apply substitution on itself. The keys of the specified map are variables (instances of `VariableExpression`), and the values are the expressions which have to substitute the occurrences of the variable.

getSQLTemplate() returns a template for the SQL formatter for this expression, see section 5.2 for details.

5.1.3 Relations

A relation is identified by its name, and has an arity (the number of arguments). Relations are defined in the class `Relation`.

Currently there are three different kinds of relations:

`EDBRelation`: These relations are extensionally defined by facts, which are stored in a database table.

`EDBComputedRelation`: These "built-in" relations are extensionally defined by facts, which are computed on demand.

`IDBRelation`: These relations are intensionally defined by rules; the results are stored in a database table.

Built-in relations are `EqualsRelation` (name `eq`), `GreaterEqualsRelation` (name `ge`), `GreaterThanRelation` (name `gt`), `LessEqualsRelation` (name `le`), `LessThanRelation` (name `lt`) and `VagueLessRelation` (name `ge`). Note that currently not all built-in relations are defined as `EDBComputedRelation`, the system also knows the ternary relations `add`, `mult`, `div` and `sub` (the first argument always is the result), the binary `log` (the natural logarithm), the aggregation operators and `neq` (for non-equality) .

5.1.4 The relation base

The relation base provides some methods for adding facts, computing rules, performing queries, retrieving facts and removing facts. The current implementation does not compute the order in which rules have to be executed automatically, so rules have to be computed in the correct order by hand.

These methods are provided:

`add(Fact)` adds the fact to its relation.

`compute(IDBRelation, Collection)` computes the result of the specified non-recursive rules (all corresponding to the specified IDB relation). Probabilities for the same facts derived by multiple rules are combined using the inclusion-exclusion formula.

`compute(IDBRelation, Rule)` computes the result of the specified non-recursive rule (corresponding to the specified IDB relation). When facts are derived multiple times, the probabilities are combined using the inclusion-exclusion formula.

`compute(IDBRelation, Rule, boolean)` computes the result of the specified non-recursive rule (corresponding to the specified IDB relation). When facts are derived multiple times, the probabilities are combined using the inclusion-exclusion formula. A user can specify that no database index is created for the IDB relation (which typically increases performance, but takes some time to be created).

`computeDisjoint(IDBRelation, Collection)` computes the result of the specified non-recursive rules (all corresponding to the specified IDB relation). Probabilities for the same facts derived by multiple rules are added, assuming disjointness.

`computeRecursively(IDBRelation, Rule)` computes the result of the specified recursive rules (all corresponding to the specified IDB relation). Probabilities for the same facts derived by multiple rules are combined using the inclusion-exclusion formula. Non-recursive rules for the same relation have to be computed first.

`addIndex(Relation)` creates a database index for the specified relation. This typically increases performance, but takes some time to be created.

clear(String) removes all tuples from the the relation with the specified name.

query(String) returns all tuples (as `Fact` objects) matching the specified query (as a single string).

queryRelation(Relation) returns all tuples (as `Fact` objects) in the specified relation.

queryRelation(String, int) returns all tuples (as `Fact` objects) in the specified relation.

getTupleCount(Relation) returns the number of tuples in the specified relation.

The relation base keeps track of all defined relations; for that, it stores an internal list of all `Relation` objects. Thus, every relation has to be added to the relation base before it can be used, even if it already physically exists, by calling `add(relation,false)` (the `false` prevents the relation to be physically re-created). These relation list handling methods are available:

clear() removes all relations from the list, without removing them physically in the database.

containsRelation(String) returns true iff there is a relation with the specified name.

get(String) returns the relation object for the specified name.

isEmpty() returns true if there is no relation in the system.

names() returns an iterator over the relation names (as strings).

add(Relation) adds the relation to the system, and creates it physically in the database.

add(Relation, boolean) adds the relation to the system, and optionally creates it physically in the database.

remove(String) removes the relation with the specified name, and also physically removes it in the database.

size() returns the number of relations in the system.

relations() returns an iterator over all relations (as `Relation` objects).

The relation base also allows for some lower-level methods for relation handling:

existsRelation(String) returns true if the relation specified by its name physically exists.

dump(Relation) dumps the content of the specified relation to STDOUT.

perform(SQL) performs the specified abstract SQL statement.

performQuery(SQL) performs the specified abstract SQL select statement, and returns a `ResultSet`.

close(ResultSet) closes the specified result set and frees all resources.

5.2 Formatting pDatalog++ to SQL

The basic idea is to map each relation onto a table, where the relation name equals the table name. Each argument is a column, and an additional (last) column is used for storing the probabilities.

As different database management systems use slightly different SQL dialects, the concepts of abstract SQL statements is introduced. Abstract SQL statements are converted into concrete SQL statements in a database-dependent way, so that no program code has to be modified when switching to a different database.

5.2.1 Abstract SQL statements

The class `de.unidu.is.sql.SQL` is an abstraction from SQL queries. It is used in situations where different database management systems with different SQL dialects (e.g. HSQLDB vs. MySQL) are used. These abstract SQL statements are formatter later to native SQL statements, using database-dependent config files.

The bean class `SQL` stores the following values:

`insertTable` contains the table name where rows are inserted.

`select` contains the list of expressions for the "select" part.

`isDistinct` specifies if "select distinct" has to be used.

`from` contains the list of expressions for the "from" part.

`where` contains the list of expressions for the "where" part. If the list is empty or equals `null`, no "where" is used.

`group` contains the list of expressions for the "group by" part. If the list is empty or equals `null`, no "group by" is used.

`limit` contains the maximum number of rows used (if greater than zero).

`order` contains the list of list of expressions for the "order by" part. If the list is empty or equals `null`, no "order by" is used.

`orderDesc` specifies if "desc" has to be used in the "order by" construct.

5.2.2 SQL formatter

The interface `de.unidu.is.sql.SQLFormatter` provides methods for converting an abstract SQL statement into a concrete one, and sending it to a database management system. Its standard implementation is the class `de.unidu.is.sql.SQLFormatterImplementation`. It can only be instantiated by a factory.

The SQL formatters provide the following methods:

`create(String, String[], String[]))` creates a table with the specified column names and types.

`clear(String))` removes all rows from the specified table.

`remove(String))` removes the specified table.

`addIndex(String, String, String[], boolean[]))` adds a named database index to the specified table, using the specified columns (which might be marked as text columns).

`getSelect(SQL))` returns the concrete SQL query for the specified abstract SQL query.

`perform(SQL,String)` performs the specified abstract SQL statement. If the specified table name equals `null`, then the table name specified in the abstract SQL statement is used.

`performQuery(SQL)` performs the specified abstract SQL select statement, and returns a `ResultSet`.

`close(ResultSet)` closes the specified result set and frees all resources.

`getDB())` returns the database object.

`setDB(DB)` sets the database object.

The SQL formatter uses a config file (in the `conf/db`, where the file name equals the database management system identifier in the JDBC uri, followed by `.conf`) directory with templates which will be filled. For that, sometimes placeholders `foo` are defined, so that every occurrence of `${foo}` can be replaced later by another string.

These are all fields defined in the config file:

`table.remove` contains the SQL statement for removing a table, using the placeholder `table.remove.table` (the table name).

`table.create` contains the SQL statement for creating a table, using the placeholders `table.create.table` (the table name) and `table.create.args` (a comma-separated list of argument name, space and argument type).

`table.clear` contains the SQL statement for removing all rows from a table, using the placeholder `table.clear.table` (the table name).

`table.insert` contains the SQL statement for inserting rows into a table, using the placeholders `table.insert.table` (the table name) and `table.insert.select` (a “select” statement or a “values” statement, defining the row(s) to be inserted).

`table.select.select` contains the SQL “select” statements, using the placeholders `distinct` (for the “distinct” construct), `select` (for the “select” construct), `table.select.from.option` (for the “from” construct), `table.select.where.option` (for the “where” construct), `table.select.group.option` (for the “group by” construct), `table.select.order.option` (for the “order by” construct), `table.select.limit.option` (for the “limit” construct).

`table.select.distinct` contains the string used in the “distinct”.

`table.select.from` contains the “from” construct template, using `from` (the table name(s)).

`table.select.where` contains the “where” construct template, using `where`.

`table.select.where.and` contains the string used for conjunctions in “where” constructs.

`table.select.group` contains the “group” construct template, using `group`.

`table.select.from` contains the “from” construct template, using the placeholders `order` and `table.select.order.desc.option`.

`table.select.order.desc.option` contains the string used for “order by desc”.

`table.select.limit` contains the “limit” construct template, using `limit`.

`table.select.values` contains the “values” template for inserting rows as facts, using `values`.

`table.addindex` contains the SQL statement for adding an index, using the placeholders `table.addindex.table` (the table name) and `table.addindex.definition` (the definition).

`table.addindex.textcolsuffix` contains the suffix for text columns when adding an index (could be used for restricting the length of the strings considered in the index).

`str2num.start` contains the start of the SQL code converting a string into a number. It will be followed by the string in the final concrete SQL statement.

`str2num.end` contains the end of the SQL code converting a string into a number. It will be preceded by the string in the final concrete SQL statement.

`type.text` contains the SQL type of text columns.

`type.double` contains the SQL types of floating numbers.

5.2.3 SQL formatter for pDatalog++

The class `PDatalogSQLFormatter` delegates most methods to the standard SQL formatter, and provides these additional convenience methods:

`create(Relation))` creates the specified relation.

`create(int, String))` creates the specified relation.

`clear(Relation))` removes all tuples (rows) from the specified relation.

`dump(Relation))` dumps the specified relation to STDOUT.

`dump(int, String))` dumps the specified relation to STDOUT.

`dump(Relation, PrintStream))` dumps the specified relation to the specified stream.

`dump(int, String, PrintStream))` dumps the specified relation to the specified stream.

5.2.4 DB objects

The class `DB` provides an encapsulation of JDBC connections and statements. Besides some convenience methods, it also employs a connection broker for improved efficiency.

Insert and update SQL statements can be sent to the database management system with one call of the `execute(String)` method. Internally, a connection is retrieved from the pool, a statement is created, the SQL statement is sent, and everything is closed again.

For SQL select statements, the `executeQuery(String)` method returns a `ResultSet` object, which can be used for retrieving the data. The result set has to be closed after usage by calling `close(ResultSet)`.

Currently two sub-classes encapsulate specific JDBC drivers for two different database management systems:

`MySQLDB` encapsulates the popular MySQL database.

`HSQLDBEmbeddedDB` encapsulates the HSQLDB database in the mode where it keeps all data in main memory. So, it does not save anything to disk, and does not need a server, but is embedded in the Java program.

5.2.5 Examples for the formatting process

This section presents some examples for the formatting process from pDatalog++ into SQL. The MySQL database system which is able to keep relations in main memory is used for storing and processing pDatalog++.

First, two binary relations `foo` and `bar` are created, which results in creating two tables with three columns (two for the two arguments, the last one for the probabilities):

```
drop table if exists bar
create table bar (arg0 varchar(255),arg1 varchar(255),prob double)

drop table if exists foo
create table foo (arg0 varchar(255),arg1 varchar(255),prob double)
```

Inserting facts is straightforward:

```
0.3 bar('1','2').
0.5 bar('2','3').
0.7 bar('3','4').

insert into bar values('1','2',(0.3*(1)))
insert into bar values('2','3',(0.5*(1)))
insert into bar values('3','4',(0.7*(1)))
```

Processing rules is more complex. In the following, we consider two rules for the same relation `foo`, which are defined to be disjunctive:

```
foo(X,Y) :- bar(X,Y).
foo(X,Z) :- bar(X,Y) & bar(Y,Z).

drop table if exists tmp_foo
create table tmp_foo (arg0 varchar(255),arg1 varchar(255),prob double)
insert into tmp_foo select rel0.arg0,rel0.arg1,(rel0.prob) from bar rel0
insert into tmp_foo select rel0.arg0,rel1.arg1,(rel0.prob*rel1.prob)
                    from bar rel0,bar rel1 where (rel0.arg1=rel1.arg0)
insert into foo select tmp_foo.arg0,tmp_foo.arg1,sum(tmp_foo.prob)
                from tmp_foo group by tmp_foo.arg0,tmp_foo.arg1
alter table foo add index (arg0)
alter table foo add index (arg1)
drop table if exists tmp_foo
```

Here, the table `tmp_foo` stores the facts created by the two rules. Creating these two `insert into ... select` statements is rather generic and not discussed here any further. The last `insert into ... select` statement inserts the generated facts into the `foo` table, where the probabilities are summed up if the same facts occurs multiple times in the temporary table. This can happen if both rules generate the same facts; but also the same rule can generate the fact multiple times (e.g. when the facts `bar(1,a)`, `bar(a,2)`, `bar(1,b)` are available `bar(b,2)`. Finally, two database indexes are created for both arguments, allowing for faster processing, and the temporary table is removed.

When the same rules are processed without disjointness, the inclusion-exclusion formula has to be applied. This results in these SQL statements:

```
drop table if exists tmp_foo
create table tmp_foo (arg0 varchar(255),arg1 varchar(255),prob double)
insert into tmp_foo select rel0.arg0,rel0.arg1,(rel0.prob) from bar rel0
insert into tmp_foo select rel0.arg0,rel1.arg1,(rel0.prob*rel1.prob)
                    from bar rel0,bar rel1 where (rel0.arg1=rel1.arg0)
select distinct tmp_foo.arg0,tmp_foo.arg1 from tmp_foo
select tmp_foo.prob from tmp_foo where (tmp_foo.arg0='1') and (tmp_foo.arg1='2')
insert into foo values('1','2',(0.3*(1)))
select tmp_foo.prob from tmp_foo where (tmp_foo.arg0='2') and (tmp_foo.arg1='3')
insert into foo values('2','3',(0.5*(1)))
select tmp_foo.prob from tmp_foo where (tmp_foo.arg0='3') and (tmp_foo.arg1='4')
insert into foo values('3','4',(0.7*(1)))
select tmp_foo.prob from tmp_foo where (tmp_foo.arg0='1') and (tmp_foo.arg1='3')
insert into foo values('1','3',(0.15*(1)))
select tmp_foo.prob from tmp_foo where (tmp_foo.arg0='2') and (tmp_foo.arg1='4')
insert into foo values('2','4',(0.35*(1)))
alter table foo add index (arg0)
```

```
alter table foo add index (arg1)
drop table if exists tmp_foo
```

The temporary table `tmp_foo` is created as before. The `select distinct` now ranges over all possible arguments. For each argument, the probabilities are retrieved from the database and combined via the inclusion-exclusion formula. The resulting probability is inserted into the `foo` table.

The pDatalog++ engine has only basic support for recursive rules (employing a naive evaluation scheme):

```
insert into foo select rel0.arg0,rel0.arg1,(rel0.prob) from bar rel0
alter table foo add index (arg0)
alter table foo add index (arg1)
drop table if exists tmpa_foo
create table tmpa_foo (arg0 varchar(255),arg1 varchar(255),prob double)
drop table if exists tmpb_foo
create table tmpb_foo (arg0 varchar(255),arg1 varchar(255),prob double)
insert into tmpa_foo select distinct * from foo
delete from tmpb_foo
insert into tmpb_foo select rel0.arg0,rel1.arg1,(rel0.prob*rel1.prob)
                     from tmpa_foo rel0,bar rel1 where (rel0.arg1=rel1.arg0)
insert into tmpa_foo select distinct * from tmpb_foo
delete from tmpb_foo
select count(distinct arg0,arg1,prob) from tmpa_foo
delete from tmpb_foo
insert into tmpb_foo select rel0.arg0,rel1.arg1,(rel0.prob*rel1.prob)
                     from tmpa_foo rel0,bar rel1 where (rel0.arg1=rel1.arg0)
insert into tmpa_foo select distinct * from tmpb_foo
delete from tmpb_foo
select count(distinct arg0,arg1,prob) from tmpa_foo
delete from tmpb_foo
insert into tmpb_foo select rel0.arg0,rel1.arg1,(rel0.prob*rel1.prob)
                     from tmpa_foo rel0,bar rel1 where (rel0.arg1=rel1.arg0)
insert into tmpa_foo select distinct * from tmpb_foo
delete from tmpb_foo
select count(distinct arg0,arg1,prob) from tmpa_foo
alter table tmpa_foo add index (arg0)
alter table tmpa_foo add index (arg1)
delete from foo
select distinct tmpa_foo.arg0,tmpa_foo.arg1 from tmpa_foo
select tmpa_foo.prob from tmpa_foo where (tmpa_foo.arg0='1') and (tmpa_foo.arg1='2')
insert into foo values('1','2',(0.3*(1)))
select tmpa_foo.prob from tmpa_foo where (tmpa_foo.arg0='2') and (tmpa_foo.arg1='3')
insert into foo values('2','3',(0.5*(1)))
select tmpa_foo.prob from tmpa_foo where (tmpa_foo.arg0='3') and (tmpa_foo.arg1='4')
insert into foo values('3','4',(0.7*(1)))
select tmpa_foo.prob from tmpa_foo where (tmpa_foo.arg0='1') and (tmpa_foo.arg1='3')
insert into foo values('1','3',(0.38587499999999997*(1)))
select tmpa_foo.prob from tmpa_foo where (tmpa_foo.arg0='2') and (tmpa_foo.arg1='4')
insert into foo values('2','4',(0.7253750000000001*(1)))
select tmpa_foo.prob from tmpa_foo where (tmpa_foo.arg0='1') and (tmpa_foo.arg1='4')
insert into foo values('1','4',(0.19897499999999999*(1)))
alter table foo add index (arg0)
alter table foo add index (arg1)
drop table if exists tmpa_foo
drop table if exists tmpb_foo
```

The basic idea is to use two temporary tables `tmpa_foo` and `tmpb_foo`. The first one accumulates the generated facts, the second one contains newly generated facts. Then, the rule is evaluated repeatedly until the total number of facts is constant. In a final step, the probabilities are computed as described above.

Chapter 6

PIRE implementation

PIRE is a probabilistic IR engine which is based on the principles introduced in the previous chapters. This chapter describes the internals of PIRE.

As mentioned in the section 3.4.1, each attribute/operator pair has its own index, which is a non-empty set of pDatalog++ relations with some additional methods. The relations in different indexes can be distinguished by prefixing the relation name with the attribute name, if this is required by the underlying pDatalog++ engine.

PIRE supports arbitrary index types (as Java classes) as long as they implement the Java interface for the index. This allows for using different index types with the same IR engine (and the same data types, see below). Currently only one index type is available, which uses pDatalog++ not only as an interface to the other components, but also internally as it builds on the pDatalog++ engine introduced in chapter 5.

PIRE also uses classes for modelling data types. Although major parts of the data type can be expressed in pDatalog++, some parts (mainly splitting an attribute value or a comparison value into tokens) should be implemented in Java. Thus, each data type has its class, which works on a specified index.

Figure 6.1 shows the different layers of PIRE. On top, there is a thin, simple XML layer (see section 7). The actual PIRE layer contains data types and indexes (including the pDatalog-based index class). The third layer contains the pDatalog++ engine, using an abstract SQL layer, which finally accesses a relational database management system.

6.1 Methods for handling an index

The interface `de.unidu.is.retrieval.Index` is responsible for handling an IR index. It contains high-level (specialised on IR) and low-level (for arbitrary relations) methods.

As mentioned in section 3.4.1, each index uses pre-defined relations with the local names `docid`, `tf`, `weight`, `rd` and `idb_rd`, and potentially further data type-specific relations.

The higher, IR-specific level methods for creating an index are:

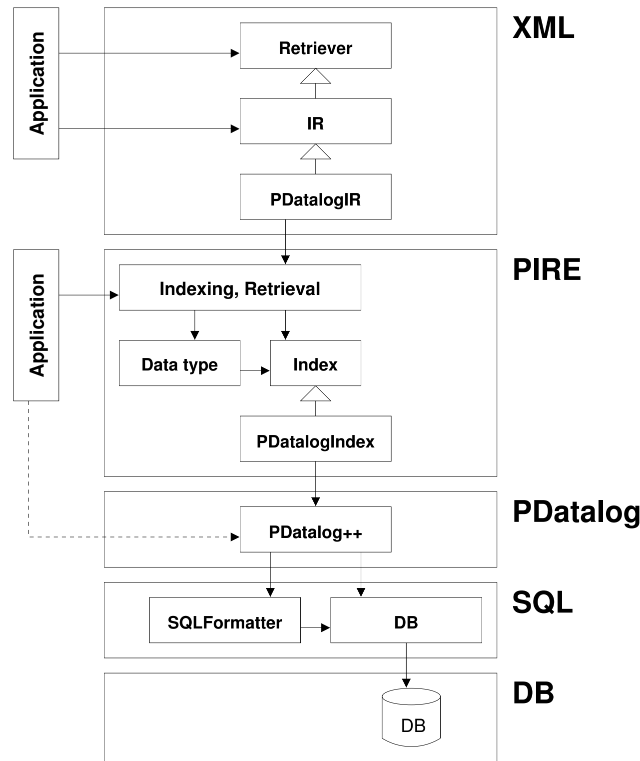
`init()` creates the pre-defined relations for this index.

`insert(String)` adds the specified document id to the relation `docid`.

`insert(String,String,String)` adds the specific document id, token and token frequency to the relation `tf`.

`computeMoments()` computes the moments of the indexing weights in the relation `expectation` (the expectation μ) and `variance` (the variance σ^2). These relations are mainly used for resource selection.

Figure 6.1: PIRE architecture



remove() removes the pre-defined relations from this index.

Computing the `weight` relation is out of the scope if the index interface, as this typically involves data type-specific knowledge. Thus, this is defined in the data type classes.

In addition, it supports retrieval on the index. One relation `[c]_[a]_[o]_rsv[query id]_[i]` is used for each sub-query. As described in section 4.5, each sub-query uses only one attribute/operator pair, and thus only one index (in which RSVs and probabilities for this sub-query are computed. These RSVs are mapped onto a probability of relevance via mapping functions (in a temporary relation `[c]_[a]_[o]_prob[query id]_[i]`). In a final step, these probabilities are combined in the relation `[c]_prob[query id]`, which is contained in a new index (used only for that purpose), and not in the attribute/operator indexes. The current interface `de.unidu.is.retrieval.Index` does not provide any method for directly transferring the data from one index into another; instead, it is required to use the same index class for every index, and moving data (if required) is left to the actual index class.

These methods are provided for performing retrieval:

initQuery(String) initialises the specified query.

getMaxRSV(String,String) returns the maximum RSV $Pr(q_i \leftarrow d)$ for the specified sub-query. This method is mainly intended for use by the data type classes.

addRule(String,String[],List) adds a rule for computing the `prob` relation to the specified list. The body is the conjunction of the specified unary relations which contain the result for a single condition.

computeProbs(String,List,boolean) applies the specified rules for the relation `prob` (where disjointness of the rules can be specified), i.e. it computes probabilities of relevance $Pr(\mathrm{rel}|q,d)$ from RSVs $Pr(q_i \leftarrow d)$, where the rules act on behalf of mapping functions.

getProbs(String,int) returns the first top-ranked document ids together with the probabilities of relevance (as a list of `de.unidu.is.retrieval.ProbDoc` instances).

closeQuery(String) frees resources acquired for processing the specified query (e.g. temporary relations).

Methods for computing RSVs are left out here as this is covered by data type classes (see section 6.2).

In addition, also computing the moments for weighted sum queries is supported:

addMomentsCondition(String,weight,Object,List) adds rules for computing the moments to the specified list, using the specified queryID, weight and comparison value.

computeMoments(String,List) applies the specified rules for computing the moments. The results are stored in the `prob` relation, with the document ids `expectation` and `variance` (i.e., by partially abusing the existing infrastructure for document retrieval).

The low-level methods allow for direct manipulation of arbitrary relations:

convert(String) converts a local relation name into an absolute one, using the collection name, the attribute and the operator name as a prefix.

convertCollection(String) converts a local relation name into an absolute one, using only the collection name as a prefix.

getRD(String) returns the resource description value for the specified key.

getRD(String,double) returns the resource description value for the specified key. If no value is the stored, the specified default value is returned.

setRD(String,double) sets the resource description value for the specified key with the specified value.

add(Fact) adds a fact to the index.

compute(Rule) evaluates the specified rule on the index.

compute(IDBRelation,Collection) evaluates the specified collection of rules with common head predicate.

computeDisjoint(IDBRelation,Collection) evaluates the specified collection of rules with common head predicate, assuming disjointness.

computeDisjoint(String,int,Collection) evaluates the specified collection of rules with common head predicate, assuming disjointness.

iterator(String) returns all facts in the specified relation.

addEDBRelation(String,int,boolean) adds an extensional relation (defined by facts) to the index.

addIDBRelation(String,int,boolean) adds an intensional relation (defined by rules) to the index.

removeRelation(String) removes the specified relation from the index.

completeRelation(String) signals that the specified relation is complete; the actual index class can perform some post-processing for efficiency improvements.

getRSVRelation(String,String) returns the name for the RSV relation, following the scheme `[c]_[a]_[o]_rsv[query id]_[i]`.

getProbRelation(String,String) returns the name for the relation with the probabilities of relevance, following the scheme `[c]_[a]_[o]_prob[query id]_[i]`.

6.2 Methods for describing data types

The data type classes cover only the parts of the retrieval process which depend on a specific data type and operator.

These methods support creating an index:

convertOperator(String) converts an operator name into an identifier (which only contains alphanumeric characters and the underscore).

addToIndex(Index,String,String,Object) adds a document/value to an index w. r. t. an operator; this method splits the attribute content into tokens (e.g. stemming and stop-word removal), calculates the token frequencies, and calls the specified index via `insert(String,String,int)` for inserting them into the `tf` relation.

computeIndex(Index,String) computes data type/operator-specific indexing weights.

removeIndex(Index,String) removes the data type/operator-specific relations from the specified index.

E.g., a simple Boolean weighting (which can be used for numbers) can be implemented using the rule

```
weight(D,V) :- tf(D,V,TF).
```

A normalised tf weighting schema requires the rules:

```
dl(D,DL) :- sum(DL,D,{tf(D,_,#)}).
weight(D,V) :- tf(D,V,TF) & dl(D,DL) | (TF/DL).
```

In addition, the rules for computing RSV for a single condition, and for probabilities of relevance for a sub-query depend on the data type and operator:

addRSVRules(Index,String,String,String,double,Object,List) adds rules for computing the RSV for the specified condition to a list. The rule head predicate is always `[c]_[a]_[o]_rsv[query id]_[sub-query id]`. Typically, it is only one rule which is added.

addProbRules(Index,String,String,String,List) adds rules for computing the probabilities of relevance for the specified sub-query to a list. The rule head predicate is always `[c]_[a]_[o]_prob[query id]_[sub-query id]`. Typically, it is only one rule which is added.

6.2.1 Example: **AbstractDT**

This class `de.unidu.is.pire.dt.AbstractDT` is the abstract super-class of all DT classes. It provides these additional methods:

getFilter(String) returns a filter object for splitting actual attribute content into tokens. The filter either has to return instances of `de.unidu.is.util.Tuple` (element 0 contains the token, element 1 the token frequency as an integer), or any object whose `toString()` method has to return the token (with frequency 1). Sub-classes have to overwrite this abstract method.

getQueryFilter(String) returns a filter object for splitting actual query condition values into tokens. The filter either has to return instances of `de.unidu.is.util.Tuple` (element 0 contains the token, element 1 the token frequency as an integer), or any object whose `toString()` method has to return the token (with frequency 1). Sub-classes have to overwrite this abstract method.

getProbsTemplate(Index,String,String,String) returns the template for computing probabilities of relevance out of the RSVs. The RSV has to be denoted as `PROB`. This method simply returns `PROB`, which refers to the identity mapping function. Sub-classes can overwrite this method.

In addition, this class provides default implementations of methods which are defined in the interface `de.unidu.is.pire.dt.DT`:

convertOperator(String) converts an operator name into an identifier (which only contains alphanumeric characters and the underscore). The default implementation simply returns the operator name, as typical operator names are also legal identifiers.

addToIndex(Index,String,String,Object) adds a document/value to an index w. r. t. an operator. It uses the filter returned by `getFilter(String)` and inserts them into the specified index.

computeIndex(Index,String) computes data type/operator-specific indexing weights. This default method uses a simply binary indexing scheme. Sub-classes can overwrite this method.

removeIndex(Index,String) removes the data type/operator-specific relations from the specified index. This default implementation does nothing.

addRSVRules(Index,String,String,String,double,Object,List) adds rules for computing the RSV for the specified condition to a list. This default implementation uses the indexing weight directly. Sub-classes can overwrite this method, but this is typically not required.

addProbRules(Index,String,String,String,List) adds rules for computing the probabilities of relevance for the specified sub-query to a list. This default implementation uses the template returned by `getProbsTemplate(Index,String,String,String)`. Sub-classes can overwrite this method, but this is typically not required.

6.2.2 Example: **TextDT**

This class defines a data type for textual content. It provides several operators which already have been defined in section 3.2.1.

By default, the RSVs are normalised (i.e., divided by the maximum RSV) before the logistic mapping function is applied. This can be enabled by `setDoScaleForBM25(true)`. However, this normalisation is only applied for the operators `stemen` and `nostem`, not for the variants with normalised tf weights.

The parameters b_0 and b_1 of the logistic mapping functions (again, only for the operators `stemen` and `nostem`) can be set as the resource description values with keys `b0` and `b1`[1]. Default values are $b_0 = -4$ and

[1]Recall that the resource description values are stored in the index.

$b_1 = 12$. The logistic mapping function has to be enabled using `setDoMapForBM25(true)`, it is disabled by default (which is equivalent to the identify mapping function).

As a byproduct, this data type computes relations `df` (document frequency) and `dl` (document length) in addition to the standard relations.

6.2.3 Example: `NumbertDT`

This class defines a data type for numbers. Binary indexing weights are used together with the identity mapping function.

This data type supports the binary operators <, <=, >, >= and = with the usual semantics and the identity mapping function.

It will also provide vague operators ∼<, ∼> and ∼= with logistic mapping functions, although this is not fully implemented yet.

6.2.4 Example: `YearDT`

This class defines a data type for numbers. Binary indexing weights are used together with the identity mapping function. Currently there is no difference to `NumberDT`.

6.2.5 Example: `NameDT`

This class defines a data type for person names with two binary operators `plainname` and `soundex`. In both cases, names are converted into lower case, split into words (tokens), and their frequencies are counted (although they are currently ignored).

The operator `plainname` directly compares the name tokens with the tokens in the query condition. The operator `soundex` compares the soundex values from the name tokens.

6.3 Method for the IR engine

Index and data type classes are combined in the PIRE main class. First, this class stores the schema (attributes with data type and operators), it keeps one index for each attribute/operator pair.

Indexing is supported by these methods.

`registerAttribute(String,String,List)` registers an attribute with a corresponding data type and a list of operators (a sub-list of the operators provided by the operator).

`initIndex()` initialises all indexes, i.e. removes old data (if necessary), and creates fresh indexes for all attribute/operator pairs.

`addToIndex(String)` adds the specified document id to all indexes.

`addToIndex(String,String,Object)` adds the specified document/value pair to all indexes for the specified attribute (i.e., for all operators which are defined on that attribute). As splitting a value into tokens, and adding tokens to the index depends on the data type and operator, it is delegated to the data type class and not to the index.

`computeIndex()` computes the indexes, basically the `weight` relations for all indexes. As this highly depends on the data type and operator, it is delegated to the data type class and not to the index.

`computeMoments()` computes the moments of the indexing weights for all indexes. The results are stored in the relation `expectation` (the expectation μ) and `variance` (the variance σ^2). These relations are mainly used for resource selection.

`removeIndex()` removes all indexes.

With these methods, creating a PIRE index has to be done in four steps:

- First, the schema has to be constructed.
- Then, the indexes have to be initialised.
- For each document, its id has to be added, and the content of each attribute (if it occurs in the document).
- Finally, the actual index (the `weight` relations) have to be computed.

For retrieval, these methods are supported:

`initQuery(String)` initialises the relations for the specified query.

`addCondition(String,String,String,double,Object)` adds the specified weighted condition (for a weighted sum query). The condition is saved and evaluated when all conditions are available, as conditions referring to the same attribute/operator pair have to evaluated together.

`addCondition(String,WeightedQueryCondition)` adds the specified weighted condition (for a weighted sum query). The condition is saved and evaluated when all conditions are available, as conditions referring to the same attribute/operator pair have to evaluated together.

`addConjunction(String,QueryCondition[])` adds the conditions of a conjunction (for a Boolean-style query in disjunctive form). These rules are stored and later evaluated together.

`computeProbs(String)` evaluates all collected rules, and computes the final probabilities of relevance.

`getProbs(String,int)` returns the first top-ranked document ids together with the probabilities of relevance (as a list of `de.unidu.is.retrieval.ProbDoc` instances).

`closeQuery(String)` frees resources acquired for processing the specified query (e.g. temporary relations).

`addMomentsCondition(String,String,String,double,Object)` adds the specified weighted condition, defined by an queryID, attribute, operator weight and comparison value, to the computation of the moments.

`addMomentsCondition(String,WeightedQueryCondition)` adds the specified weighted condition to the computation of the moments.

`getMoments(String)` returns the moments for the specified query.

Retrieving documents with PIRE is similarly easy:

- First, the query relations have to be initialised.
- Then, each condition of a weighted sum and each conjunction of a Boolean-style query has to be added.
- Once all conditions are added, the resulting probabilities of relevance $Pr(\text{rel}|q,d)$ can be computed.

- Then, the top-ranked documents can be retrieved from the `prob` relation.
- Finally, the query has to be closed. This means that the relations `rsv` and `prob` in all involved indexes have to be removed, as well as internal data about the query.

PIRE provides convenient methods for setting and retrieving values in the `rd` relation for a specified attribute and operator:

`double getRD(String,String,String` returns the value in the resource description for the specified index and key.

`void setRD(String,String,String,double` sets the value in the resource description for the specified index and key.

Finally, PIRE provides access to the underlying index objects for advanced applications:

`Index getIndex(String,String` returns the index object for the specified attribute and operator.

Thus, using PIRE from outside is fairly easy, and of course abstracts from the internals.

Chapter 7

Retrieval of XML documents

This chapter gives an overview over the implemented Java code for XML retrieval.

7.1 Data structures

This section describes the data structures used to encode documents, schemas and queries.

7.1.1 Documents

XML documents are returned in their parsed form as DOM trees, i.e. as `org.w3c.dom.Document` instances. As input parameters for methods, either `Document` instances or strings can be used (this is only important for creating a document index).

Documents are identified by a document id. This is a string of arbitrary form, e.g. a URI, a URN, a number encoded as a string, a file name, or anything else. The only requirement is that it is unique within one index.

The retrieval result is essentially a list of documents with attached weight $RSV(d,q)$ which describes the similarity between a document and the query. This can be the probability of inference $Pr(q \leftarrow d)$ in Uncertain inference, the probability of relevance $Pr(\text{rel}|q,d)$, or another similarity measure. Such a result is encapsulated in `ProbDoc` instances, which store the document id (which identifies the document itself) and the *RSV* weight. If also the XML document is required, the class `XMLDoc` can be used which combines a `ProbDoc` and a `Document` instance.

7.1.2 Schemas

Schemas define the structure of XML documents. Here, a simplified view is used, which defines the elements and attributes which are allowed as children of another element. The order and cardinality of children elements, default values for attributes etc. is not considered. Schemas are stored in `Schema` instances. Each schema contains an optional name and a reference to the root schema element.

The class `SchemaElement` contains the data used about "elements" of the schema, which can be XML elements, XML attributes and text nodes. Each schema element has a name, which can be the element name, an attributed name prefixed with `@`, or `text()` for text nodes. This naming scheme allows for direct usage in XPath expressions, which will be used for selecting parts of the XML documents. A schema element referring to text nodes also contains an optional data type name and a list of operators, which is important for indexing. Currently, there is no strict list of allowed data types or operators, it depends on the indexing/retrieval engine. Finally, a schema element contains a list of children schema elements for defining the hierarchical structure. Recursion (loops) are forbidden, thus a schema consists of a tree of schema elements. As a consequence, a schema contains a finite set of distinct XPath expressions, each

referring to one content-carrying schema element (i.e., one which does not refer to an XML element). This list can be retrieved using the method `getXPaths()` in both classes `Schema` and `SchemaElement`.

E.g., the following XML document

```
<document>
  <metadata correct=''true''>
    <title>Foo and Bar</title>
    <author>John Doe</author>
    <author>Jane Doe</author>
    <year>2004</year>
  </metadata>
  <abstract>
    The abstract ...
  </abstract>
  <fulltext>
    The main text ...
  </fulltext>
</document>
```

corresponds to the XPath expressions

```
/document/metadata/@correct
/document/metadata/title/text()
/document/metadata/author/text()
/document/metadata/year/text()
/document/abstract/text()
/document/fulltext/text()
```

Some IR systems are not capable to deal with hierarchical documents. One example is the IR engine PIRE which only allows for linear schemas. Thus, a schema definition can also contain the mapping from XPath entries (and, thus, from schema elements) onto so called "aliases", the elements of linear schemas. This is an n:m mapping, thus each XPath expression can belong to several aliases, and each aliases can contain several XPath expressions. Not every XPath expression has to belong to any alias, and the usage of aliases is not mandatory. Typically, the term "path" subsumes XPath expressions (which always start with a slash `/`) and aliases (which never start with a slash `/`).

Useful aliases for the example from above would be `title`, `author`, `abstract`, `fulltext` and `text` (containing the abstract and the full text).

The method `addAliases()` creates aliases automatically. Every XPath expression of text and XML attribute schema element nodes forms one alias, converting every slash and every minus into an underscore, converting the `@` sign into the string `@`, removing `/text()` suffixes, and ignoring the very first element name in the XPath. Thus, in our example we would derive the aliases `metadata_atcorrect`, `metadata_title`, `metadata_author`, `metadata_year`, `abstract` and `fulltext`.

The schema (using the method `usesXPathForQuery()`) defines which type of paths is used in queries (see below).

7.1.2.1 Schemas created from DTDs

One way of defining schemas is to use existing DTDs. The schema sub-class `DTDSchema` loads a DTD and extracts the children relationships from it.

Data types and operators cannot be extracted, as there is no information about them in the DTD. However, data types and operators can be added later, by manipulating the `SchemaElement` elements or adding default operators. Here, `Text` uses `stemen`, `Name` uses `plainname` and `soundex`, and the data types `Year` and `Number` use `=`.

In addition, no aliases are defined; if they are needed, they have to be specified manually or by adding all possible aliases from XPath expressions, see above.

7.1.2.2 Schemas created from HyREX DDL files

Alternatively, HyREX [1] DDL files can be used. The advantage is that it contains more information for indexing and retrieval than DTD files; in addition, many concepts are very similar for HyREX and PIRE.

This is a sample DDL file:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<hyrex
  directory="/path/to/hyrex/index"
  base="MyBase"
  class="ByClass"
  dtd="/foo/mydtd.dtd">

  <access classname="HyREX::HyPath::Document::Access::Find">
    <parameter name="expression"  value="$_[0] =~ /\/.*xml$/"/>
    <parameter name="directories" value="/path/to/xml"/>
  </access>

  <summary>
    <xslfile name="...xsl"/>
  </summary>

  <attribute name="title">
    <datatype classname="HyREX::HyPath::Datatype::Text::English">
      <parameter name="indexfilter" value="latin1_tr"/>
      <parameter name="indexfilter" value="latin1_lc"/>
      <parameter name="indexfilter" value="split2"/>
      <parameter name="indexfilter" value="stop"/>
      <parameter name="filter"      value="latin1_tr"/>
      <parameter name="filter"      value="latin1_lc"/>
      <parameter name="filter"      value="split2"/>
      <parameter name="filter"      value="stop"/>
      <query query="/document/metadata/title"/>
    </datatype>
  </attribute>

  <structure classname="HyREX::HyPath::Structure::NoStruct"/>

</hyrex>
```

In our context, `/hyrex/@directory` (the directory where HyREX stores the index on disk), `/hyrex/@dtd` (DTD file), `/hyrex/access/@classname` (Perl class name for the access method), `/hyrex/summary` (XSLT stylesheets for displaying information on web pages) and `/hyrex/attribute/datatype/parameter` (data type parameters) can be ignored.

The parts `/hyrex/@base`, `/hyrex/@class` and `/hyrex/access/parameter` are only for indexing, they are used in the indexer program (see section 7.3).

The schema class `HyREXSchema` extracts all XPaths from `/hyrex/attribute/datatype/query/@query`, and forms a new tree of `SchemaElement` elements from them. The data type definition from `/hyrex/attribute/datatype/@classname` is converted into PIRE data types: So, the HyREX data type

HyREX::HyPath::Datatype::Text::English is converted into Text, HyREX::HyPath::Datatype::Name is converted into Name, and HyREX::HyPath::Datatype::Numeric into Number. Using PIRE data type names is also allowed, although one looses compatibility to HyREX.

We extend the DDL file by optional /hyrex/attribute/datatype/predicate/@name constructs, defining operators for the schema elements. Here, plaintexten is converted into nostem, and equal into =.

The /hyrex/structure is set to NoStruct, then each attribute defines one alias.

7.1.3 Queries

Queries are identified by their query id (any string), and contain the number of documents which have to be retrieved.

The system contains different kinds of queries:

Keyword-based queries: The class KeywordQuery contains a list of keywords (when used a string, they are separated by spaces). It does not contain any reference to any path or operator.

XIRQL queries defined by a string: The class XIRQLStringQuery contains a XIRQL query string. XIRQL [12] is a XPath-like query language extended by IR constructs. XIRQL is primarily used by the HyREX XML retrieval engine [2]. In principle, any legal XIRQL query is allowed, although some XML retrieval engine (e.g. the XML extension to PIRE) only supported a limited range of queries.

XIRQL weighted-sum queries: The class WeightedSumQuery stores a weighted-sum query, encoded as a XIRQL query. An example query has one of the forms:

```
wsum(0.3,/a/b/@foo $stemen$ ``xyz'',0.7,/a/c/d/text() $plainname$ ``Doe'')

wsum(0.3,/#PCDATA $foo:stemen$ ``xyz'',0.7,/#PCDATA $bar:plainname$ ``Doe'')
```

XIRQL Boolean-style queries: The class StructuredQuery stores Boolean-style queries, the most-complex query type (from a data structure perspective). Essentially, such a query contains a query node, which itself can contain additional query nodes (forming a tree-like structure). The query nodes can be serialised into the XIRQL query format.

The following query nodes are supported:

AndQueryNode: This class models the conjunction of a list of children query nodes.

OrQueryNode: This class models the disjunction of a list of children query nodes.

QueryCondition: This class models a single query condition, consisting of a path, an operator and a comparison value.

Boolean-style queries can be brought into disjunction form, which is a conjunction of disjunctions of query conditions.

7.2 Information retrieval with XML documents

This section describes standard interfaces for indexing and retrieval (which can also be used for other IR engines), as well as the concrete implementation based on PIRE:

7.2.1 Document retrieval

The interface Retriever defines methods for XML document retrieval using the above-mentioned data structures. This interface is intended as a common gateway to different IR engines. Several implementations (e.g. for PIRE and HyREX) are available, additional ones will follow.

The interface Retriever defines the following methods:

`getSchema()` returns the schema used by this instance.

`getResult(Query)` returns the result for the specified query. The result is a (ranked) list of `ProbDoc` instances, which store the document ids together with their overall weights. An `UnsupportedQueryException` is thrown if the query format is illegal. If the index is not readable, an `IndexException` is thrown.

`getResultSummaries(Query)` returns the results as summaries for the specified query. The result is a (ranked) list of `XMLDoc` instances, which contain a `ProbDoc` instance (the document id and the document weight) as well as an XML document (which in this case contains the document summary). An `UnsupportedQueryException` is thrown if the query format is illegal. If the index is not readable, an `IndexException` is thrown. A `DocumentNotFoundException` is thrown if any of the documents cannot be returned.

`getSummary(String)` returns the summary as an XML document for the specified document id. A `DocumentNotFoundException` is thrown if the document cannot be returned.

`getSummary(ProbDoc)` returns the summary as an XML document for the specified document id, given by the specified `ProbDoc` instance. A `DocumentNotFoundException` is thrown if the document cannot be returned.

`getSummaries(List)` returns the summaries for the specified documents, given the specified list of `ProbDoc` instances (the document id and the document weight). The result is a list of `XMLDoc` instances, which contain the `ProbDoc` instance as well as an XML document (which in this case contains the document summary). A `DocumentNotFoundException` is thrown if any of the documents cannot be returned.

`getDocument(String)` returns the complete document as an XML document for the specified document id. A `DocumentNotFoundException` is thrown if the document cannot be returned.

`getDocument(ProbDoc)` returns the complete document as an XML document for the specified document id, identified by the specified `ProbDoc` instance. A `DocumentNotFoundException` is thrown if the document cannot be returned.

`getDocuments(List)` returns the complete documents for the specified documents, given the specified list of `ProbDoc` instances (the document id and the document weight). The result is a list of `XMLDoc` instances, which contain the `ProbDoc` instance as well as an XML document (which in this case contains the complete document). A `DocumentNotFoundException` is thrown if any of the documents cannot be returned.

`close()` closes the instance, which might allow for free some used system resources (e.g. closing a network connection).

7.2.2 Document indexing

The interface `IR`, a sub-interface of `Retriever` defines additional methods for XML document indexing. An implementations for PIRE is available, additional ones will follow.

The interface `IR` defines the following methods in addition to those of `Retriever`:

`registerSchema(Schema)`

`initIndex()` inits the index, i.e. it removes all old data from the index data structures.

`addToIndex(String, Document)` adds the specified XML document (with the specified document id) to the index.

`addToIndex(String, String)` adds the specified XML document (with the specified document id) to the index.

computeIndex() computes the overall index, based on the added documents. This method could e.g. compute the indexing weights based on idf values, which have to be computed once all documents are added.

removeIndex() removes the index.

7.2.3 XML document retrieval with PIRE

With the above classes and interfaces, performing document indexing and retrieval with PIRE is straightforward. It is implemented in the class `PDatalogIR`.

First, only aliases are used in paths. In other words, parts of the XML documents are mapped onto so called "attributes" (which form a linear list). So, before `PDatalogIR` is used, one has to ensure that aliases are defined for the schema.

The schema data type must be one of the PIRE data types. These XML document parts are indexed separately, and can be considered for retrieval, where only `WSumQuery` and `StructuredQuery` are allowed for queries.

Obviously, this is only a primitive form of XML retrieval, as it makes only limited usage of the document structure. A more sophisticated version is currently under development, still adhering to the same interface definitions and using the same data structures.

7.3 The Indexer program

PIRE and the XML extension can easily be used in own applications. The program `index.sh`, which calls `de.unidu.is.retrieval.Indexer`, provides a convenient front-end for the task of indexing a given collection of XML documents. As the program uses `PDatalogIR`, everything explained in section 7.2.3 is valid here as well, in particular that parts of XML documents defined by XPath expressions are mapped onto attributes.

7.3.1 Installation

You can start `de.unidu.is.retrieval.Indexer` directly, or call the `index.sh` shell script. This script is found in the `bin` directory in CVS, it will be copied to the `dist` directory automatically when `ant dist` is invoked. The shell script uses the JAR file `unidu.jar` directly, so the classpath is set automatically. It cannot be used without the JAR file.

7.3.2 Usage

The indexer programs expects some parameters:

```
-?,--help                    displays this help message
-q,--quiet                   do not output anything to STDOUT and STDERR
-l,--logfile <file>          use the specified file for the output
-u,--user <user>             use specified user name for RDBMS
-p,--password <password>     use specified password for RDBMS
-h,--host <host>             use specified host for RDBMS
-d,--db <database>           use specified database for RDBMS
-1,--ddl <ddl file>          use specified HyREX DDL file
-2,--dtd <dtdl file>         use specified DTD
-n,--name <collection name>  use specified collection name instead of
                             combining base/class from HyREX DDL
```

```
-x,--xml <dir>                  use specified directory for XML files
                                (instead of extracting it from the HyREX DDL file)
```

The user, password, host and database parameters are required, they specify where the index is created (using a MySQL database). In addition, either a HyREX DDL file or a DTD file have to be specified as a schema.

The DDL file (see section 7.1.2.2) specifies the attributes which are used for indexing and retrieval. When using structure is switched off in the DDL file, then the attribute names are used directly as aliases; otherwise, XPath expressions which are explicitly mentioned in the DDL are converted into aliases. If the DDL does not contain operator definitions (remember that this is an extension to HyREX), default operators are used. The collection name (the prefix for the MySQL tables) and the directory containing the XML files are extract from the DDL, as well as a regular expression which is used to filter documents in the directory (so that only documents matching the regular expression are indexed).

If a DTD is specified, then the collection name and the directory containing XML files (which end with `.xml`) have to be specified manually. All paths which can be created based on the DTD are used and converted into aliases, and default operators are added.

The document ids are the file names (without path and `.xml` extension is all cases).

All output to STDOUT and STDERR can be switched off by the `-quiet` option. It is also possible to specify a file where all output is written to (even if output to STDOUT/STDERR is blocked).

The program directly initialises the index, adds all documents to it, and completes the index computation. After that, the index is ready for querying.

Chapter 8

Conclusion and outlook

This chapter summaries this report, and describes some possible future work.

8.1 Conclusion

This report introduced PIRE, a probabilistic IR engine. PIRE is based on probabilistic Datalog++, which is probabilistic Datalog with aggregation, arbitrary functions for computing the probability of derived facts, and some syntactic sugar.

With pDatalog++, PIRE is built on a theoretically founded basis for information retrieval. PIRE itself is entirely implemented in Java, but makes in its current form heavy use of a relation database for the pDatalog++ part. This can be an external database management system which is running as a server (e.g., the Open Source database system MySQL[1]), or an embedded database like HSQLDB[2].

PIRE is simple to use in applications, and is already integrated in the federated Digital Library system Daffodil[3] for retrieval on locally available collections as well as for ranking the merged result list.

An additional feature, computing moments (expectation and variance) of indexing weights, allows for using the PIRE infrastructure also for the decision-theoretic framework of resource selection [21].

8.2 Outlook and further extensions

In this final section, we depict some points where PIRE can be extended in future.

8.2.1 PIRE tuning

PIRE can be easily extended towards new application areas. Besides some glue code and a small amount of document preprocessing like splitting a document value into tokens (for which a variety of existing classes can be used), everything is captured in rules. This makes it easy to integrate new data types, operators, weighting schemes and retrieval functions.

First, we plan to tune the pDatalog++ processing component to enhance speed, by improving the transformation of pDatalog++ rules into SQL statements. One problem is that one insert SQL statement is issued for every fact which is added to the system; batch insertion should be added. A second bottleneck is that database indexes are created for every intensional relation (defined by rules), which is very slow for large

[1] `http://www.mysql.com/`
[2] `http://hsqldb.sourceforge.net`
[3] `http://www.daffodil.de`

tables in MySQL. This has to be significantly improved. Furthermore, also retrieval for larger queries is getting inefficient, as every query term (or query condition) has to be handled in isolation.

PIRE can also be extended towards other directions. E.g., we did not use negation throughout this paper, but the limited usage of negation in pDatalog can quite easily be integrated into the system.

Documents could also be described by logical expressions, allowing for a richer document model. Partial representations of documents are possible when switching to four-valued probabilistic Datalog [14] (with additional probabilities for the truth values "unknown" and "inconsistent").

8.2.2 Extensional and intensional semantics

PIRE and the underlying pDatalog++ engine currently only support extensional semantics, where probabilities are directly computed when a rule is evaluated.

First, the implementation is rather inefficient when the same fact is generated several times, e.g. by different rules (see section 5.2.5). This case corresponds to a disjunction, for which the inclusion-exclusion formula has to be applied:

$$Pr(e_1 \vee \cdots e_n) = \sum_{\emptyset \subset S \subseteq \{1,\ldots,n\}} (-1)^{|S|+1} \cdot \prod_{i \in S} Pr(e). \tag{8.1}$$

Computing these sums in a single plain SQL statement is not possible, thus the pDatalog++ engine uses Java code for computing the sum, after querying the probabilities of all occurrences of the fact from the database. Although a disjointness assumption can be used for rules, a more efficient approach for computing the inclusion-exclusion formula would be helpful.

An alternative would be to switch to intensional semantics. Here, event expressions are formed from the rules and associated to each derived fact. These event expressions are Boolean combinations of event keys for the extensional facts. Probabilities for the facts are derived from these event expressions. The advantage is that multiple occurrences of the same fact can be eliminated, and disjoint events can be detected in a generic way. The disadvantage is its inefficency (although [16] describes methods for reducing the effort).

In the context of PIRE and its underlying pDatalog++ engine, the major problem is that not every SQL dialect supports the creation of event expressions in a single SQL statement. MySQL 4.1 provides a `group_concat()` aggregation operator where strings can be concatenated to form an event expression, but earlier MySQL versions and HSQLDB do not provide such a built-in construct.

8.2.3 Language models

So far, we only investigated weight sums and Boolean-style queries. The flexibility of pDatalog++ allows for further retrieval models, e.g. language models. Here, the probabilities $Pr(t|d) = Pr(t \leftarrow d)$ (the normalised frequency of term t in document d) and $Pr(t|G)$ (normalised frequency of t in the background knowledge, e.g. the complete collections) are combined with parameter λ; queries are sets of words:

$$Pr(q|d) = \prod_{t \in q} (\lambda Pr(t|d) + (1-\lambda) Pr(t|C)) \ .$$

This can easily be modelled in pDatalog++. First the probabilities $Pr(t|d)$ (stored in the relation `weight`) and the collection-specific probabilities $Pr(t|C)$ (stored in the relation `cweight`) have to be computed (the latter could also be specified manually through facts):

```
df(T,DF) :- count(DF,T,{tf(#,T,_)}).
dl(D,DL) :- sum(DL,D,{tf(D,_,#)}).
numdocs(N) :- count(N,{docid(#)}).
tfc(T,TF) :- sum(TF,T,{tf(_,T,#)}).
cl(DL) :- sum(DL,{dl(_,#)}).
```

```
weight(D,T) :- tf(D,T,TF) & dl(D,DL) | TF/DL.
cweight(T) :- tfc(T,TFC) & cl(DLC) | TFC/DLC.
```

Then, for each query term, retrieval status values have to be computed, and combined in the final result (if required, also probabilities of relevance could be computed):

```
rsv42_i(D) :- weight(D,'hello') | 0.3*PROB.
rsv42_i(D) :- cweight('hello') | 0.7*PROB.
...
rsv42(D) :- rsv42_0(D) & rsv42_1(D) & ... & rsv42_n(D).
```

8.2.4 External knowledge

One of the major advantages of logical frameworks in IR is that external knowledge can easily be incorporated into the retrieval process. For example, term associations from a thesaurus like WordNet[4] can be used for query expansion.

WordNet already contains a set of facts which can be directly be used in pDatalog. E.g. "synsets" group synonyms together:

```
s(100012748,1,'animal',n,1,67).
s(100012748,2,'animate_being',n,1,0).
s(100012748,3,'beast',n,1,4).
s(100012748,4,'brute',n,2,0).
s(100012748,5,'creature',n,1,16).
s(100012748,6,'fauna',n,2,0).
```

The first argument is the synset id, the third one the term. Thus, among other terms, "animal", "beast" and "fauna" are seen as synonyms. These synsets can be exploited for retrieval, by searching for documents which contain a synonym of a search term:

```
rsv42_0(D) :- weight(D,'hello').
rsv42_0(D) :- weight(D,T) & s(N,_,T,_,_,_) & s(N,_,'hello',_,_,_).
```

Of course, it should be possible to incorporate more complex situations as well. In general, it should be possible to state queries directly as pDatalog++ rules, which then can use any available relation.

This feature will probably be added soon. A workaround would be to use PIRE for indexing documents, and the pDatalog++ layer directly for query evaluation (i.e., pDatalog++ rules have to be created for queries manually).

8.2.5 Ontologies and the Semantic Web

A new and challenging problem is Information Retrieval in the context of the Semantic Web, where information is structured and presented employing the OWL ontology language[18]. IR plays a role in different parts of the information life-cycle:

- Wrappers extract information from web pages and provide the result in OWL, using techniques like information extraction, named entity recognition, wrapper induction etc. It should be possible to state uncertain facts, as the extraction process might not be correct.
- The resulting OWL documents have to be indexed for retrieval, providing a powerful IR query language. Both content-based retrieval as well as the structure has to be considered for querying. The result is a ranked list of individuals.

[4] http://www.cogsci.princeton.edu/~wn/

- Heterogeneity between different ontologies have to be captured via mapping rules. Also here uncertainty plays a crucial role.

A (partial) mapping from the restricted OWL Lite version, which is strongly related to the $\mathcal{SHIF}(\mathbf{D})$ description logics, onto a four-valued variant of pDatalog [14] has been proposed in [22]. This mapping also allows for extending OWL by probabilities, the resulting language is called pOWL. Four-valued probabilistic Datalog programs, finally, can easily be transformed into two-valued pDatalog.

pOWL consists of classes (unary predicates) and properties (binary predicates). pOWL assertions can easily be mapped onto pDatalog facts, and class hierarchies are transformed into rules:

```
Individual(Bob 0.7 type(Person))
Individual(Peter 0.9 value(parent Mary))
Class (Person partial 0.49 Man 0.51 Woman)
```

is transformed into:

```
0.7 Person(Bob).
0.9 parent(Peter,Mary).
0.49 Man(X) :- Person(X).
0.51 Woman(X) :- Person(X).
```

PIRE can be used, with smaller modifications, for retrieval on such pOWL data as well. We use the same techniques as for integrating external resources like WordNet: Facts and class definitions are be transformed into pDatalog facts and rules, and the results are be stored in SQL tables. Textual content (the values of a property) can be indexed using PIRE in addition. The retrieval process (i.e., the retrieval rules) then can refer to both, the IR index as well as the structural part of pOWL.

8.2.6 Sophisticated XML retrieval

Finally, XML retrieval should be supported. A primitive approach which is already implemented is to map sub-trees of the XML document (defined e.g. by XPath expressions) onto attributes. We plan to turn PIRE into a fully-fledged XML retrieval engine which explicitly takes the hierarchical structure of the documents into account. Logics seem to be an excellent starting point for this.

HyREX [2] and the XIRQL query language [12] are good starting points for defining retrieval functionality which should also be employed in the XML-enhanced PIRE version.

List of Figures

4.1 Ideal and real case 20
4.2 Example logistic functions 20

5.1 PDatalog++ engine architecture 24

6.1 PIRE architecture 35

List of Tables

Bibliography

[1] M. Abolhassani, N. Fuhr, N. Gövert, and K. Großjohann. HyREX: Hypermedia retrieval engine for XML. Research report, University of Dortmund, Department of Computer Science, Dortmund, Germany, 2002.

[2] M. Abolhassani, N. Fuhr, and S. Malik. HyREX at INEX 2003. In N. Fuhr, M. Lalmas, and S. Malik, editors, *INitiative for the Evaluation of XML Retrieval (INEX). Proceedings of the Second INEX Workshop. Dagstuhl, Germany, December 15–17, 2003*, pages 27–32, Mar. 2004. `http://inex.is.informatik.uni-duisburg.de:2003/proceedings.pdf`.

[3] C. Buckley. Implementation of the SMART information retrieval system. Technical Report 85-686, Department of Computer Science, Cornell University, Ithaca, NY, 1985.

[4] W. S. Cooper, F. C. Gey, and D. P. Dabney. Probabilistic retrieval based on staged logistic regression. In N. J. Belkin, P. Ingwersen, and A. M. Pejtersen, editors, *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Copenhagen, Denmark, June 21-24, 1992*, pages 198–210, New York, 1992. ACM.

[5] R. Fagin, P. G. Kolaitis, W.-C. Tan, and L. Popa. Composing schema mappings: Second-order dependencies to the rescue. In *Proceedings PODS*, 2004.

[6] S. Fienberg. *The Analysis of Cross-Classified Categorical Data*. MIT Press, Cambridge, Mass., 2. edition, 1980.

[7] D. H. Freeman. *Applied Categorical Data Analysis*. Dekker, New York, 1987.

[8] N. Fuhr. A probabilistic framework for vague queries and imprecise information in databases. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 696–707, Los Altos, California, 1990. Morgan Kaufman.

[9] N. Fuhr. Towards data abstraction in networked information retrieval systems. *Information Processing and Management*, 35(2):101–119, 1999.

[10] N. Fuhr. Probabilistic Datalog: Implementing logical information retrieval for advanced applications. *Journal of the American Society for Information Science*, 51(2):95–110, 2000.

[11] N. Fuhr and C. Buckley. A probabilistic learning approach for document indexing. *ACM Transactions on Information Systems*, 9(3):223–248, 1991.

[12] N. Fuhr and K. Großjohann. XIRQL: An XML query language based on information retrieval concepts. *ACM Transactions on Information Systems*, 22:313–356, 2004.

[13] N. Fuhr and U. Pfeifer. Combining model-oriented and description-oriented approaches for probabilistic indexing. In *Proceedings of the Fourteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 46–56, New York, 1991. ACM.

[14] N. Fuhr and T. Rölleke. HySpirit – a probabilistic inference engine for hypermedia retrieval in large databases. In *Proceedings of the 6th International Conference on Extending Database Technology (EDBT)*, pages 24–38, Heidelberg et al., 1998. Springer.

[15] F. C. Gey. Inferring probability of relevance using the method of logistic regression. In B. W. Croft and C. J. van Rijsbergen, editors, *Proceedings of the Seventeenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–231, London, et al., 1994. Springer-Verlag.

[16] T. Hoffmann. Effiziente wahrscheinlichkeitsberechnung für ereignisausdrücke. Master's thesis, Universität Dortmund, Fachbereich Informatik, 2004.

[17] J. W. Lloyd. *Foundations of Logic Programming*. Springer, Heidelberg et al., 2. edition, 1987.

[18] D. L. McGuinness and F. van Harmelen. OWL. Technical report, World Wide Web Consortium, 2004. `http://www.w3.org/TR/owl-features/`.

[19] H. Nottelmann and N. Fuhr. Learning probabilistic Datalog rules for information classification and transformation. In H. Paques, L. Liu, and D. Grossman, editors, *Proceedings of the 10th International Conference on Information and Knowledge Management*, pages 387–394, New York, 2001. ACM.

[20] H. Nottelmann and N. Fuhr. Decision-theoretic resource selection for different data types in MIND. In J. Callan, F. Crestani, and M. Sanderson, editors, *Recent research in multimedia distributed information retrieval. Proceedings of the ACM SIGIR 2003 Workshop on Distributed Information Retrieval, Toronto, Canada. (Lecture Notes in Computer Science, 2924).*, Heidelberg et al., 2003. Springer.

[21] H. Nottelmann and N. Fuhr. Evaluating different methods of estimating retrieval quality for resource selection. In J. Callan, G. Cormack, C. Clarke, D. Hawking, and A. Smeaton, editors, *Proceedings of the 26st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, New York, 2003. ACM.

[22] H. Nottelmann and N. Fuhr. powl lite: A probabilistic extension to owl lite based on four-valued probabilistic datalog. (Submitted for publication), 2005.

[23] P. Ogilvie and J. Callan. Experiments using the Lemur toolkit. In *Proceedings of the 2001 Text REtrieval Conference (TREC 2001)*, 2002.

[24] S. E. Robertson. The probability ranking principle in IR. *Journal of Documentation*, 33:294–304, 1977.

[25] S. E. Robertson, S. Walker, M. Hancock-Beaulieu, A. Gull, and M. Lau. Okapi at TREC. In *Text REtrieval Conference*, pages 21–30, 1992.

[26] K. Ross. Modular stratification and magic sets for Datalog programs with negation. *Journal of the ACM*, 41(6):1216–1266, Nov. 1994.

[27] G. Salton, editor. *The SMART Retrieval System - Experiments in Automatic Document Processing*. Prentice Hall, Englewood, Cliffs, New Jersey, 1971.

[28] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, Rockville (Md.), 1988.

[29] A. van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[30] C. J. van Rijsbergen. A non-classical logic for information retrieval. *The Computer Journal*, 29(6):481–485, 1986.

[31] C. J. van Rijsbergen. Probabilistic retrieval revisited. *The Computer Journal*, 35(3):291–298, 1992.

[32] S. K. M. Wong and Y. Y. Yao. On modeling information retrieval with probabilistic inference. *ACM Transactions on Information Systems*, 13(1):38–68, 1995.