Chapter 5

The HVQL Query Language

5.1 Introduction

In the previous Chapter 4 the HyperView System, a platform for implementing HyperViews has been presented. However, the Prolog encodings of graphs, queries, and rules used in the HyperView System are too low-level for the human programmer. Moreover, this encoding requires Prolog skills. A more intuitive syntax that is independent of Prolog is preferable. To this end, the HVQL query language has been developed as a syntax for

- graph literals, i.e., data and schema graphs,
- queries against the graph database,
- rules for view definitions.

The kernel of the HVQL syntax is formed by the syntax for graph patterns. Graph literals are ground (i.e., variable free) graph patterns. Queries consist of graph patterns and additional logical connectors. Rules consist of a LHS expressed as a query and an update pattern which is a graph pattern that can be annotated with reuse specifications. Rules can be seen as conditional update statements, and graph literals as (unconditional) insert statements for the graph database.

5.2 Basic Notations

A vertex identifier V is usually a Prolog atom, but in general an arbitrary Prolog term. In particular, text vertices are represented as Prolog lists containing words. Vertex identifiers are local to the cluster determined by the syntactical context. To denote a reference for a vertex V in the external cluster C, the syntax V@C is used.

The label L of a vertex V can be specified using the syntax V:L. The label may be a variable or a Prolog term. Typically it is a Prolog atom. The label of a vertex is normally used to indicate its type. Examples of valid vertices are `node1:type1`, x@g,`v234:l5`, `root@db:root_t`.

Graphs are represented in the HyperView System in a relational way by the predicates `vertex/3` and `edge/4` (cf. Section 4.1.2). The tuple (C, X, L) is a solution for `vertex(`C`,`X`,`L`)` iff there exists an L-labeled vertex X in cluster C. The goal `edge(`C`,`X`,`E`,`T`)` is true if there exists an E-labeled edge from X in cluster C to a vertex T in C. In case that $T = Y$@D, the target is a vertex Y in cluster D. Edges are denoted in HVQL within graph patterns that are discussed next.

5.3 Graph Patterns

Graph patterns are the main syntactic device to denote graphs or templates for graphs. In Figure 5.3 the EBNF grammar for graph patterns is given. A graph pattern defines a navigation from a start vertex to a destination vertex. The start vertex is provided by the syntactic context or the runtime environment. It can be overridden by a `Source` nonterminal.

The most simple graph pattern is either an explicit source pattern (nonterminal `Source`) or a single edge (`Edge`).

A source pattern P has the form S:L or S: for a vertex S and an optional label L. Since a single vertex cannot be distinguished from an edge label, the : cannot be omitted in S:. The destination vertex of P is S@C where C is the cluster of the start vertex.

An edge pattern is denoted by a Prolog term specifying an edge label E. Examples are `author`, `date`, `get_attr(Attr)`, E. For a start vertex X@C, it translates to the Prolog goal `edge(`C`,`X`,`E`,`T`)` which yields the destination vertex T@C. Using the syntax E=T', the constraint $T = T'$ is imposed which binds T to T'.

Several patterns $P_1, \ldots, P_n$ having the same start vertex can be combined using the syntax `[`P_1`,`$\ldots$`,`P_n`]`. The combined pattern translates to the conjunction of the component patterns and its destination vertex of this combined pattern is identical to its start vertex.

```
Pattern ::= Pattern "->" Pattern
          | Pattern "=>" Pattern
          | "[" Pattern ("," Pattern)* "]" ["=" Target]
          | Edge                        ["=" Target]
          | Source
Source  ::= Vertex ":" [Label]
Target  ::= Vertex [":" [Label]]
Vertex  ::= Term ["@" Cluster]
Cluster ::= Term
Label   ::= Term
Edge    ::= Term
```

Figure 5.1: Syntax for HVQL graph patterns.

To concatenate patterns and specify for instance paths, patterns can be combined using either the `->` or the `=>` operator.

In the combined pattern P`->`Q the destination vertex of P is used as start vertex of Q. For instance, `X:->a->b->c=Y` denotes a path from `X` to `Y` along three edges labeled with `a`, `b`, and `c`, respectively. Together with the `[]` operator, arbitrary trees can be constructed, e.g.,
`r:->[a=a1->[b=b1,b=b2],c= c1]`
for a tree of depth 2 with root `r`, children `a1` and `c1` of `r` and children `b1` and `b2` of `a1`. Cycles and intersections of paths can be introduced by binding the same vertex variable twice, e.g., `X:->a->b=X`.

A pattern containing only the `[]` and `->` operators can refer to a single cluster only, namely the cluster of its start vertex. To cross the boundary between clusters along an inter-cluster edge or by jumping to an global source specification Y`@`D, the concatenation operator `=>` has to be used.

In the combined pattern P`=>`Q it is assumed that the pattern P yields as destination a global reference T`@`C where $T = Y$`@`D. Then Y`@`D is taken as start vertex for Q instead of T`@`C. A typical application is in rules where an inter-cluster edge labeled `source` from the focus F of the rule to a vertex in a cluster at a lower level and a graph pattern P starting from this vertex typically comprise a pattern F`->source=>`P for the LHS of the rule. Another example is a pattern P starting from a vertex X in cluster C which is expressed by X`@`C`:=>`P.

A table with a more formal description of the implementation of queries (which subsumes the implementation of graph patterns) can be found in Table 5.1 on page Table 5.1.

5.4 Graph Literals

A graph literal is a term P`::`S of a ground (i.e., variable free) graph pattern P and the name S of the schema cluster of which the graph literal is an instance.

Figure 5.2 shows a graph literal which defines a fragment of a schema graph for journals. It is an instance of the generic (meta-)schema `schema`. The graph denoted by this pattern is depicted in Figure 5.3.

5.5 Queries

5.5.1 Syntax

The syntax for queries extends the syntax for graph patterns by introducing boolean operators `&` (conjunction) and `|` (disjunction). The EBNF grammar defining the syntax of HVQL queries is depicted in Figure 5.4.

```
[  publisher: any -> [
       name = atom,
       home = url
    ],
   journal: any -> [
       title= text,
       publisher= publisher,
       volume= volume: any -> [
           volno = int,
           issue= issue: any -> [
               issueno= int,
               year= int,
               source = issue_ref @ springer_schema
            ],
           source = volume_toc_us @ springer_schema
        ],
       source = journal_ref @ springer_schema
    ]
] :: schema
```

Figure 5.2: HVQL graph pattern of a schema fragment for scientific journals.

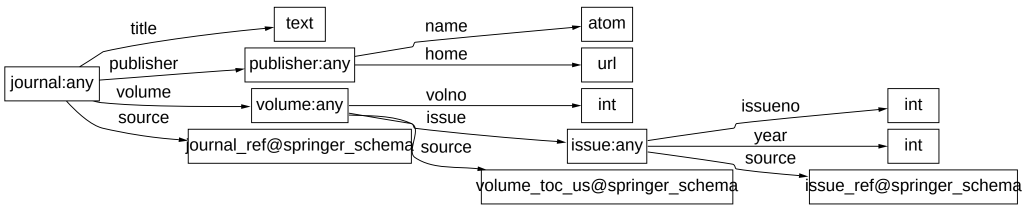


Figure 5.3: Graph denoted by the graph pattern in Fig 5.2.

As a first example we want to retrieve the IDs and names of all journals in our database. This can be denoted by the query

`JournalID: journal -> title= JournalName.`

For every node `JournalID` with label `journal` it traverses the `title`-edge leading to the name of the journal and unifies the variable `JournalName` with it. Note that HVQL adopts the Prolog convention that all capitalized identifiers denote variables. (Single quotes are used to distinguish capitalized atoms from variables).

If we want to retrieve the titles of all articles in the journal with ID `journal_42` we can use the query

`journal_42:  -> volume -> issue -> article -> title = Title.`

The query defines a path from the mentioned journal node via the edges `volume`, `issue`, `article`, and `title` to the title of an article node, which is unified with the variable `Title`.

A more complex query retrieving the titles of all articles in 1998 issues of journals on digital libraries is given in Figure 5.5. The query defines a path from a journal node to an article node, with several excursions. Excursions are denoted by sub-queries in square brackets. These sub-queries which have no navigational effect to each other or to the rest of the query. In our example they are used to bind variables such as `JournalName`, `VolumeNo`, `IssueNo` etc. on the way and to express additional constraints.

The first such constraint states that the name attribute `JournalName` of the journal node must contain the keywords `digital` and `libraries` which is denoted by the `occur`-condition. The computed edge `occur` can be materialized between any two text nodes (denoted as lists) where the words of the target node occur in that order in the source node. Other text matching operators

```
Query   ::= Query "|" Query
          | Query "&" Query
          | Query "->" Query
          | Query "=>" Query
          | "(" Query ")"                 ["=" Target]
          | "[" Query ("," Query)* "]"  ["=" Target]
          | "?" Condition                 ["=" Target]
          | Edge                          ["=" Target]
          | Source
```

Figure 5.4: Syntax for HVQL queries.

```
JournalID: journal ->
      [ name= JournalName -> occur = [digital,libraries] ] ->
      volume = _: volume -> [ volno = VolumeNo ] ->
      issue = _: issue -> [ issueno = IssueNo, year = 1998 ] ->
      article = _: article -> [ title = Title, pdf = URL ]
```

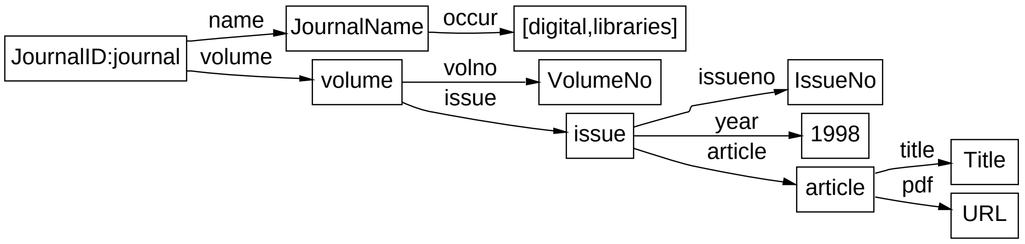


Figure 5.5: HVQL and graph representation of a query that retrieves references of articles having appeared in 1998 in journals on digital libraries.

such as regular expression matching can be added to the HyperView System in a similar way.

The second constraint fixes the year of an issue to 1998, thus filtering out all `issue` nodes not having an `year` edge with target node 1998.

For each selected article, the query returns a binding for the variables occurring in the query, namely the `JournalID`, `JournalName`, `VolumeNo`, `IssueNo`, `Title`, and `URL` of the article.

5.5.2 Semantics

A HVQL query $\langle Q \rangle$ without conjunction (&) and disjunction (|) operators corresponds to the query graph Q of a *formal* query (c.f. Definition 3.3.1). The query constraint Γ is the conjunction $\Gamma_1 \wedge \ldots \wedge \Gamma_n$ of all conditions $?\Gamma_i$ occurring in $\langle Q \rangle$. All non-variable vertex identifiers within the query together define an initial partial match m_0 of the query graph. The pre-image of this partial match is the anchor graph Q_0 of the query.

The **semantics** of a non-disjunctive and non-conjunctive HVQL query $\langle Q \rangle$ is the set of variable bindings resulting from matching the corresponding query graph Q against a given clustered data graph G in the HyperView System, such that the resulting matches are extensions of m_0 and fulfill the query constraint Γ.

A HVQL query $\langle Q \rangle$ with conjunctions can always be rewritten as a conjunction $\langle Q_1 \rangle \& \ldots \& \langle Q_n \rangle$ of non-conjunctive queries. The semantics of Q is the binding set of the union-graph $Q_1 \cup \ldots \cup Q_n$ of the query graphs defined by the queries $\langle Q_i \rangle$.

A HVQL query $\langle Q \rangle$ with conjunctions and disjunctions can always be rewritten into the form

$\langle Q_1 \rangle$ | ... | $\langle Q_n \rangle$ where all $\langle Q_i \rangle$ are non-disjunctive HVQL queries. The semantics of $\langle Q \rangle$ is then simply the union of the binding sets of $\langle Q_1 \rangle \ldots \langle Q_n \rangle$.

5.5.3 Implementation

We now discuss the concrete implementation of HVQL chosen in the HyperView System. For brevity we omit the $\langle . \rangle$ notation and use $P, Q, R, \ldots$ for HVQL queries.

The **implementation** $[[Q]]$ of a HVQL query Q is the Prolog goal into which it is compiled. This Prolog goal yields exactly the bindings defined by the semantics, but in an order defined by the implementation and the underlying semantics of Prolog. The graph-matching strategy of the HVQL implementation ensures that graph elements specified by a HVQL query are matched in textual order; in particular, vertex variables are bound from left to right. Conditions specified in a query are tested once they are reached. This allows to prune the search space as soon as all required data for evaluating a condition is available.

Each query has a start vertex and a destination vertex which are needed for the composition of sub-queries. As a default start vertex that can be overridden by the query the root node of the database graph is used.

In a conjunction P&Q both queries P and Q share the same start vertex. The semantics of P&Q is the intersection of the semantics of P and of Q. The destination vertex of the conjunction is the destination vertex of Q. The “excursion” operator `[]` is equivalent to & but uses the common start vertex also as destination vertex.

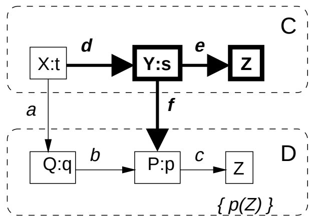
In a disjunction $P \mid Q$ the queries P and Q share the start vertex and the destination vertex of is the destination vertex of P for each solution of P and the destination vertex of Q for each solution of Q. The semantics of a disjunction is the union of the semantics of its operands. Disjunctive queries are not covered by the formalism. However, a rule having a disjunctive query as left hand side can be transformed into an equivalent set of nondisjunktive rules.

Name	**Pattern**	**Implementation**	**Condition**	**Navigation**
	P	$[[P]]$		Start $\xrightarrow{P}$ Dest
Source	$S:$	`true`		$X@C \to S@C$
	$U:L$	`vertex(`C`,`U`,`L`)`	$U \neq V@D$	$X@C \to U@C$
	$V@D:L$	`vertex(`D`,`V`,`L`)`		$X@C \to S@C$
Edge	E	`edge(`C`,`X`,`E`,`T`)`		$X@C \to T@C$
Assignment	$Q = T$	T=Y	$X@C \xrightarrow{Q} Y@D$	$X@C \to Y@D$
Constraint	?G	G	G is a Prolog goal	$X@C \to X@C$
Excursion	`[`P_1`,`$\ldots$`,`P_n`]`	$[[P_1]]$`,`$\ldots$`,`$[[P_n]]$		$X@C \to X@C$
Path	Q`->`R	$[[Q]]$`,`$[[R]]$	$X@C \xrightarrow{Q} Y@D \xrightarrow{R} Z@E$	$X@C \to Z@E$
	Q`=>`R	$[[Q]]$`,`$[[R]]$	$X@C \xrightarrow{Q} (Y@E)@D, Y@E \xrightarrow{R} Z@F$	$X@C \to Z@F$
Conjunction	Q&R	$[[Q]]$`,`$[[R]]$	$X@C \xrightarrow{Q} Y@D, X@C \xrightarrow{R} Z@E$	$X@C \to Z@E$
Disjunction	$Q \mid R$	$[[Q]]$`;`$[[R]]$	$[[Q]]$ succeeds $\wedge\ X@C \xrightarrow{Q} Y@D$	$X@C \to Y@D$
			$[[R]]$ succeeds $\wedge\ X@C \xrightarrow{R} Z@E$	$X@C \to Z@E$

Table 5.1: Implementation of queries.

5.6 Rules

Before we go into details of the HVQL representation of rules, we repeat the example of a trivial rule from Figure 4.3 and present in Figure 5.6 the HVQL representation of this rule. As we can see from the figure, the rule is denoted in the form $\langle H \rangle$ `<==` $\langle B \rangle$, where $\langle H \rangle$ contains the anchor node and the update part of the rule, and $\langle B \rangle$ the query part without the anchor node.



```
X:t -> d = Y:s ->
   [e=Z] -> f=P
   <==
   a= Q@D:q =>
   b= P:p ->
   c=Z ->
   ? p(Z).
```

```
compute_edge(C,X,d,Y):-
    vertex(C,X,t),
    edge(C,X,a,Q@D),
    vertex(D,Q,q),
    edge(D,b,P),
    vertex(D,P,p),
    edge(D,P,c,Z),
    p(Z),
    update((
        vertex(C,Y,s),
        edge(C,X,d,Y),
        edge(C,Y,e,Z),
        edge(C,Y,f,P))).
```

Figure 5.6: Example of a trivial rule shown in graph representation (left), encoded in HVQL (middle), and its Prolog translation (right).

5.6.1 Syntax

HVQL rules have the form $\langle H \rangle$ <== $\langle B \rangle$, where $\langle H \rangle$ is a HVQL pattern graph denoting the head H and $\langle B \rangle$ is a HVQL query denoting the body of a rule. The head contains an anchor node A, the primary edge, and the rest of the update part U (cf. Section 4.3).

The update part of a rule may contain reuse specifications. In HVQL, reuse specifications are indicated by enclosing graph patterns in curly brackets ($\{\ldots\}$). Such reuse patterns differ slightly from reuse graphs since they are not allowed to overlap, but need not to be complete graphs. The semantics of reuse patterns are explained in Sec. 5.6.2 below.

Using the operator ~> instead of -> inhibits the materialization of the primary edge. This can be used for simple utility rules which relay information requests to other rules.

Figure 5.7 shows the detailed EBNF syntax for rules.

```
Rule         ::=  Head "<==" Query "."
Head         ::=  Anchor ("->"|"~>") PrimaryEdge
PrimaryEdge  ::= "{" PrimaryEdge "}" [UpdateTail]
               | Edge "=" UpdateObject [UpdateTail]
Anchor       ::= Source
UpdateTail   ::= ("->"|"=>") Update
Update       ::= Update UpdateTail
               | "{" Update "}"
               | "(" Update ")"                ["=" UpdateObject]
               | "[" Update("," Update)* "]" ["=" UpdateObject]
               | Edge                          ["=" UpdateObject]
               | Source
UpdateObject ::= "{" Target [UpdateTail] "}"
               | Target
```

Figure 5.7: EBNF Syntax for rules.

5.6.2 Semantics

A HVQL rule $\langle p \rangle$ of the form $\langle A \rangle$->$\langle U \rangle$<==$\langle B \rangle$ without conjunctions, disjunctions, and reuse specifications corresponds to a formal rule p as defined in Definition 3.2.1. The LHS L of p is the union of the singleton anchor graph A and the query graph B corresponding to the rule body

$\langle B\rangle$. The RHS R is the union of L and U where U is the (incomplete) graph corresponding to the update part $\langle U\rangle$. The application constraint Γ is the query constraint of $\langle Q\rangle$ (cf. Section 5.5.2).

The semantics of $\langle p\rangle$ is then defined by the semantics of p: Let m_0 be a match for A. For each match m of L in a data graph G that extends m_0 and satisfies Γ, a fresh copy of U is instantiated with the variable bindings induced by m and is then added to G.

If we admit arbitrary HVQL queries $\langle B\rangle$ in the rule body, the semantics generalizes as follows: we instantiate the HVQL query $\langle L\rangle$= $\langle A\cup B\rangle$ corresponding to L with the variable binding for A induced by the initial match m_0. Then we pose $\langle L\rangle$ against the data graph G. For each resulting variable binding, U is instantiated and added to G as discussed before.

Now we come to HVQL rules with reuse specifications. The reuse patterns are ordered innermost first, left to right. For each match of $\langle B\rangle$ the algorithm in Figure 3.16 is applied. This means that before a copy of U is added to G, it is tried for each reuse specification K_i to match it in G. Only if there is no match, a copy of K_i is added to G. The algorithm proceeds in the specified ordering. Finally, the remainder of U is added to G.

5.6.3 Implementation

A HVQL rule of the form $\langle A\rangle$`->`$\langle U\rangle$`<==`$\langle B\rangle$ is implemented as a clause of predicate `compute_edge/4` (cf. Section 4.3). Let X@C be the anchor vertex, L the label of the primary edge, and Y@C the destination vertex of the primary edge. Then this clause has the following form:

`compute_edge(`C`,`X`,`L`,`Y`):-` $[[A\cup B]]$`, update(`$[[U]]$`).`

This clause is activated if an L-labeled edge emanating from a vertex x@c is requested and cannot be found in the graph database. In this case, the substitution $\{C=c, X=x\}$ is applied to the clause, the query $Q=[[A\cup B]]$ is solved, and for each solution of it the update $[[U]]$ is performed.

To be more exact, the update specified by U is carried out in the following way: starting from the primary edge, the update part is traversed in textual order, but handling the target of an edge before the edge itself. Targets are only inserted if a label is specified. Otherwise, it is assumed that an already existing target is referenced.

A pattern that is marked as a reuse pattern by enclosing it in curly brackets is first tried to be matched in the data graph. Reuses that are included are tried to be matched recursively. If the match fails, the elements of the reuse pattern are inserted into the data graph. To illustrate this, we introduce reuse patterns to the rule presented in Figure 5.6 so that the rule head becomes

```
X:t ->{ d = { Y:s -> [e=Z] } } -> { f=P}
```

For each binding of Z and P, it is checked whether a node `Y:s` with an e-labeled edge pointing to Z exists. If not, a vertex Y and the e-edge are created. Now it is checked whether a d-labeled edge to the matched or created Y exists and if not, this edge is created. Finally, it is tried to match a f-labeled edge from Y to P exists and if this fails, the edge is added to the graph database.

When the whole update part of the rule has been executed, the target Y of the primary edge is returned as a match for the requested edge with label L. If there is more than one match for the query part Q, backtracking will enumerate all matches and for each one, the update part will be executed and a target of the primary edge is returned. While the variable bindings within the rule will be revoked on backtracking, the newly inserted graph elements will persist since backtracking does not apply to the graph database.

5.6.4 Example

In order to provide a realistic example of a rule in HVQL notation, we use again the rule get_issue from Figure 2.8. For convenience, we repeat the graphical representation of this rule in Figure 5.8.

This rule materializes all volumes and issues of an electronic journal of the German branch of Springer in one step since its presentation on the Springer Web Site is organized by years and issues. Thus volumes have to be materialized by collecting all volume numbers occurring in the available issues. A HVQL representation of this rule is shown in Figure 5.9.

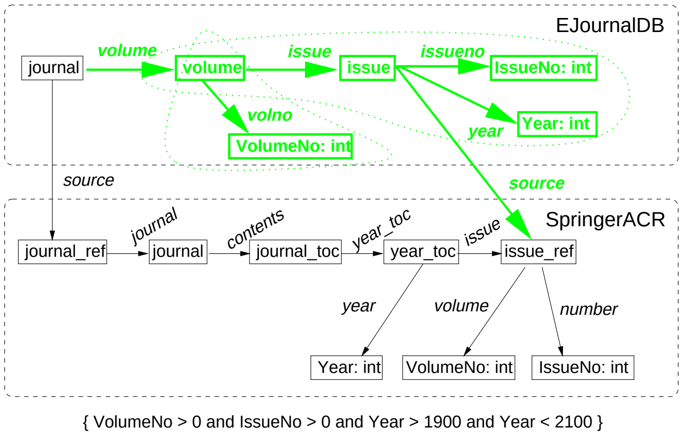


Figure 5.8: ACR rule get_issue.

```
Journal:journal-> {{volume=Volume:volume ->
        [journal=Journal, volno=VolumeNo] }->
        issue= Issue:issue->
        [volume=Volume, issueno=IssueNo, year=Year ] } ->
        source = IssueRef @ springer_acr
    <==
        source = _ @ springer_acr =>
        journal->
        contents->
        journal_toc->
        year_toc-> [year=Year-> ? (Year > 1900, Year < 2100)]->
        issue= IssueRef -> [
                volume=VolumeNo-> ? (VolumeNo > 0),
                number=IssueNo-> ? (IssueNo > 0)     ].
```

Figure 5.9: Rule get_issue in HVQL notation.

The query of this rule follows a `source` edge to the `springer_acr` cluster and there a path along edges labeled `journal`, `contents`, `year_toc`, and `journal_toc`, respectively, to a vertex having a `year` edge whose target `Year` is in the interval $[1900, 2100]$. From there, it follows an `issue` edge to an `issue` vertex having a `volume` and `number` edges which point to nonnegative integers `VolumeNo` and `IssueNo`.

The presented rule contains two nested reuse components. A volume is identified by the journal to which it belongs and its number. This is expressed by the inner reuse components. Since the condition on the volume number is formulated as excursion by using the `[]` notation, the destination of this reuse component is the volume vertex `Volume`.

An issue is identified by the volume wherein it is contained and its number. Hence the outer reuse specification includes the one for `Volume`, the issue vertex `Issue` itself, and the `issueno` and `year` edges. The the source vertex of the issue is not used for identification purposes and hence not included in the reuse component.

5.7 Meta Edges

HVQL itself does not have any aggregation operators. However, it supports the definition of meta edges which define parameterized edges that take other HVQL queries as arguments. The following query retrieves for instance the set of all authors of an article `a1` by using the meta rule `set/1`:

```
a1: article -> set({author}) = Authors
```

The HVQL compiler recognizes edge parameters enclosed in curly brackets as queries and replaces them by a query execution plan represented as a term of the form `query(`S`,`P`,`D`)` where P is a prolog goal which computes for a given start vertex S the destination vertex D.

This makes it easy to define Prolog clauses which take as input a start vertex and an edge label containing a query execution plan and use the available meta predicates of Prolog to aggregate, select, or transform the solutions of this plan and return them as target of the meta edge. A library of such meta rules is provided by module `hvs(meta_hvql)` of the HyperView System.

We conclude this section with another example. The meta-edge `maximize(`E`,`Q`)` returns all solutions for a query Q which maximize an arithmetic expression E. To retrieve volume number, issue number, and year of the *most recent issue* of a given journal `Journal`, we use `maximize/2` as in the query depicted in Figure 5.7.

```
    Journal: journal->
          maximize(VolNo,
              {volume -> [volno = VolNo ]}) ->
          maximize(IssueNo,
              {issue  -> [issueno = IssueNo ] }) ->
          year = Year
```

This query finds the volume of `Journal` with maximal `volno` and for this volume the one of its issues having maximal `issueno` and finally the `year` of this issue.

A list of the most important meta edges supported by the HyperView System is presented in Table 5.2.

Edge	**Description**
`set(Query)`	set of all results of `Query` as a list
`bag(Query)`	return the multiset of all results of `Query` as a lis
`list(Query)`	synonym for `bag(Query)`
`count(Query`	number of solutions of `Query`
`max(Expr,Query)`	return the maximum value of arithmetic expression `Expr` over all solutions of `Query`
`min(Expr,Query)`	analogous
`distinct(Query)`	return only distinct results of `Query`
`opt(Query, Default)`	return each result of `Query`, and `Default` if there are none
`not(Query)`	succeeds if `Query` has no solution
`maximize(Expr,Query)`	return the result of `Query` for which the integer expression `Expr` is maximal and not negative
`minimize(Expr,Query`	return the result of `Query` for which the integer expression `Expr` is minimal and not positive
`nth(Index, Query)`	return the `Index`-th result of `Query`
`star(Query)`	Concatenate `Query` arbitrarily often (`Query`*)
`plus(Query`	Concatenate `Query` at least once (`Query`$^+$)
`alt(Query1,Query2)`	execute `Query1`, if it does not have any solutions, execute `Query2`
`try(Query)`	execute `Query`; if it fails, ignore it
`once(Query)`	execute `Query` once

Table 5.2: A selection of Meta edges supported by the HyperView System.

5.8 HTML Edges

When HTML pages are loaded into the HyperView System, they are converted into parse trees. Each parse tree forms a cluster of the data graph which conforms to the schema cluster `html_schema`. The vertices of such a HTML cluster are labeled with the HTML tags to which they correspond. Text without markup (so-called PCDATA) is represented by lists of words. The children of a node in the parse tree are ordered textually and can be accessed via numbered edges with labels `#(`i`)`. Attributes of HTML tags are represented by edges labeled with the attribute names.

Since the parse tree does not represent the logical relations between HTML tags, the `html_schema` provides several rules which materialize such connections.

Edge	**Description**
`child`	A logical child of a node. By default, this is the child relation in the parse tree. For headers (`<h1>`,..,`<h6>`), the outermost nodes to the right of the header up to the next header of the same or higher rank are found.
`right_sibl`	Any sibling (w.r.t. `child`) textually to the right of a node
`sub`	any node which logically belongs to the source node of the `sub` edge. By default any descendent of the node in the parse tree.
`right`	Any node textually to the right of a node
`item`	Any item of a list-like node, for instance any `<li>` belonging to a `<ol>` node.
`href_resolve`	Resolves the (possibly relative) URL stored in the `href` attribute of an anchor tag `<a>` into an absolute URL.
`href_target`	Dereferences the hyperlink of an anchor tag `<a>`, loads the referenced page, and returns the root node of this page.

Table 5.3: Virtual edges defined for HTML graphs

5.9 Embedding of HVQL in the HyperView System

How are schemata and rule sets in HVQL loaded into the HyperView System? Schema clusters and the initial values of data clusters are encoded in HVQL graph literals as described in Section 5.4 and then declared as Prolog modules. These modules carry the names of the clusters (cf. Section 4.1.2). Templates for data graph and schema files are shown in Figure 5.10 and Figure 5.11, respectively. The syntax of HVQL is supported in Prolog by means of several operator declarations that have to be included in the module using the `consult/1` statement in the second line of the templates. With respect to these declarations, HVQL constructs are ordinary Prolog terms that are treated as facts asserted by the source file. However, when loading the source files, these facts are intercepted by a special predicate that translates them into the Prolog encodings required by the HyperView System. Thus graph literals are translated on the fly into sets of `vertex/2` and `edge/3` facts as discussed in Section 4.1.2.

```
:- module(<GRAPH>, []).
:- consult(hvs(hvql_syntax)).
<GRAPH_LITERAL> :: <SCHEMA>.
```

Figure 5.10: Template of a data graph file.

Schema files are distinguished from source files for data graphs in that they export the predicate `dispatch_edge/4` that is called for rule activation. A schema is itself an instance of the meta-schema called `schema`. Finally, a schema file can include or import rules of the HyperView that has the schema cluster as output cluster. Imports are specified by use of facts for the predicate `implementation/1` that forces the implementation module given as argument to be loaded.

```
:- module(<SCHEMA_NAME>, [dispatch_edge/4]).
:- consult(hvs(hvql_syntax)).
<GRAPH_LITERAL> :: schema.
<RULE>.
...
implementation(<IMPLEMENTATION_MODULE>).
...
```

Figure 5.11: Template of a schema file.

Implementation modules (see Figure 5.12) contain rule sets. These rules are included in the HyperView for the schema that imports the implementation module. Rules are treated as Prolog terms that are translated at load time into clauses of the predicate `compute_edge` as discussed in Section 4.3. These clauses are then compiled by the Prolog system into bytecode for the Prolog engine (WAM). Auxiliary predicates that are called from application constraints in rules may be included in the implementation module or imported from other modules.

```
:- module(<MODULENAME>, []).
:- consult(hvs(hvql_syntax)).
<RULE>.
...
<AUXILIARY PREDICATE>.
...
```

Figure 5.12: Template of an implementation module.

5.10 Summary

The HVQL language provides a syntax to encode graphs, queries and rules in an intuitive notation that can be translated into the Prolog encodings required by the HyperView System.

The central notion of HVQL is that of a graph pattern. Variable free graph patterns are used to denote graph literals such as schema graphs. Queries consist of nonground graph patterns that may be connected with boolean operators.

Rules are denoted by a head and a body part, the head being an update pattern and the body being a query. Update patterns are graph patterns that may contain additional reuse patterns that prevent the creation of redundant graph elements.

HVQL is extensible via so-called *meta-edges*. Meta-edges are virtual edges that take HVQL queries as arguments. The set of meta-edges currently available in the HyperView System includes various aggregation operators and a number of control and navigation operators that are not part of HVQL itself. In particular, regular path expressions are suppported this way.

For graph clusters representing HTML pages, the HyperView System provides a set of virtual edges that provide specialized navigation operators for HTML parse trees. Like meta-edges, they can be seen as an extension of the HVQL language.