WUM: A Tool for Web Utilization Analysis

Myra Spiliopoulou[1] and Lukas C. Faulstich[2] *

[1] Institut für Wirtschaftsinformatik, HU Berlin
[http://www.wiwi.hu-berlin.de/~myra], [myra@wiwi.hu-berlin.de]
[2] Institut für Informatik, FU Berlin
[http://www.inf.fu-berlin.de/~faulstic], [faulstic@inf.fu-berlin.de]

Abstract. The navigational behaviour of users in the web is essential for the providers of information, services and goods. Search engines can help a user find a provider of interest, but it is the proper organization of the provider's site that stimulates the user's propensity to consume. To verify whether the site is effectively organized, knowledge on the navigation patterns occuring during visits to the site must be obtained. Our Web Utilization Miner WUM can assist in obtaining this knowledge. WUM is a mining system for the discovery of navigation patterns. A navigation pattern is a directed graph that summarizes the traversal movements of a group of visitors and satisfies certain human-centric criteria that make it "interesting". Instead of focussing the mining process on the statistically dominant but not always interesting patterns, WUM supports the specification of *interestingness* criteria on their structure, content and statistics.
WUM provides a declarative mining language, MINT, with which the human expert can specify interestingness criteria on the fly. To discover the navigation patterns satisfying the expert's criteria, WUM exploits an innovative aggregated storage representation for the information in the web server log.

1 Introduction

Web sites are most often organized in a way the providers consider appropriate for the majority of the site's visitors. However, our knowledge of the actual navigational behaviour of the visitors is still sparse and fragmentary. Simple access statistics provide only rudimentary feedback, while studies on specific behavioural patterns, e.g. page revisits [12], are of rather ad hoc nature.

Knowledge about the navigation patterns occurring in or dominating the usage of a web site can greatly help the site's owner or administrator in improving its quality. Data mining can assist in this task by effectively extracting knowledge from the past, i.e. from the site access recordings. In [5], the term "web usage mining" is suggested to describe this type of mining activity.

To aid an expert in reorganizing a web site, a web usage miner should provide feedback on (i) access to certain nodes and paths considered of importance, (ii)

* Supported by the German Research Society, Berlin-Brandenburg Graduate School in Distributed Information Systems (DFG grant no. GRK 316).

nodes and paths preferred or avoided by the visitors and (iii) generic traversal behaviour, e.g. trends for backward moves, cycles etc. Miners generating sequential patterns that satisfy certain support or confidence thresholds are not adequate. Rather, we need a tool that can identify navigation patterns satisfying properties specified by the expert in an ad hoc manner. Those properties can concern the statistics or the contents of the pattern but may be as vague as the existence of cycles or the repeated access to some, otherwise undefined, node.

The web miner proposed in [3] simply discovers statistically dominant paths. The "WEBMINER" of [5] provides a query language on top of external mining software for association rules and for sequential patterns. However, the expressiveness of this language is restricted by the input parameters acceptable by the miner. To our knowledge, current miners do not support generic specifications on the structure of the patterns to be discovered, e.g. page revisits, cycles etc.

In this study, we present a mining mechanism that avoids those shortcomings, by incorporating the query processor to the miner. Our Web Utilization Miner WUM (`http://www.wiwi.hu-berlin.de/~myra/WUM`) employs an innovative technique for the discovery of navigation patterns. The information on web traversals is modelled according to a graph-based formalism and aggregated into a materialized view over the web log.

The expert guides the pattern discovery by means of our mining language, MINT. In MINT, she can specify the generic characteristics a pattern should possess to be of potential interest to her. Thus, only patterns having the desired characteristics are constructed, while uninteresting ones are pruned out early.

In this study, we give an overview of WUM and then focus on the functionality it offers by means of MINT. We first present the architecture of WUM. Section 3 describes the process of aggregating web log data into an Aggregated Log, on which mining is applied. In section 5 we describe our mining language MINT and its processing mechanism for the discovery of navigation patterns. The related work is discussed in section 6. We conclude in section 7.

2 The Architecture of WUM

The architecture of our Web Utilization Miner is depicted in Fig. 1. There are two major modules: the *Aggregation Service* prepares the web log data for mining and the *MINT-Processor* does the mining.

The *Aggregation Service* extracts information on the activities of the users visiting the web site and groups consecutive activities of the same user into a transaction. It then transforms transactions into sequences. Its major task is to merge and aggregate those sequences into a tree structure. This process is described informally in the next section.

MINT is the language, in which the experts specifies the interestingness criteria to be satisfied by the mining results. The *MINT-Processor* is the mining core of WUM, responsible for the discovery of navigation patterns from the aggregated information extracted by the *Aggregation Service* from the weblog. The

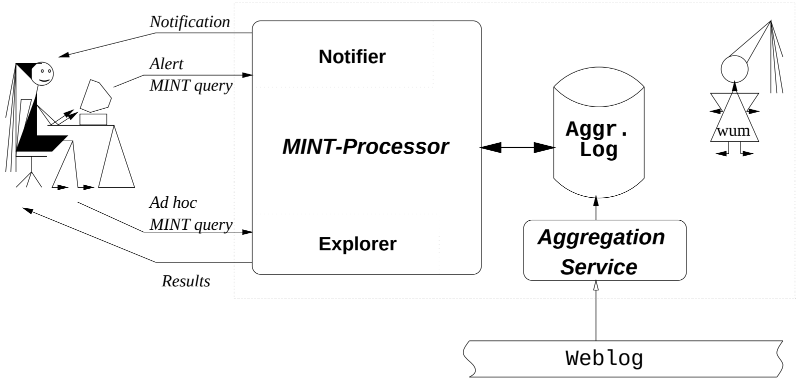

Fig. 1. The architecture of WUM

criteria specified by the expert in MINT are used by the *MINT-Processor* to prune out uninteresting patterns as early as possible.

The *MINT-Processor* can be invoked in two modes: As *Notifier*, it executes preprocessed queries periodically. As *Explorer*, it accepts ad hoc queries. The purpose of the *Notifier* is to discover whether the web access data set shows deviations from the expected usage. Hence, the "alert queries" it executes should correspond to the beliefs of the site's owner on the statistics and structure of the dominant navigation patterns. If those beliefs are not satisfied, the user should be alerted to invoke the *Explorer*, which can help in discovering the actually dominant navigation patterns.

We would like to stress here that WUM, similarly to most miners, is not an expert system. The expertise belongs to its user and is indispensable, on the one hand to describe what is interesting to her, and on the other hand to avoid misinterpretations of the results. In particular, the user should be familiar with the site's content and its intended use. She must possess some expertise on human behaviour in hypermedia environments, so that she may transform the expectations of usage into mining directives and correctly interpret the mining results; these issues are addressed e.g. in [12, 14]. Finally, she should own the appropriate background to properly interpret the statistics of the results.

3 Aggregating Visitor Transactions

The discovery of navigation patterns in WUM is performed on the basis of the information extracted from the web server log after some (currently simple) data cleaning and inserted into the "Aggregated Log". The Aggregated Log contains *aggregated* data on the visitor trails in the site, where a "visitor trail" is a sequence of page requests comprising a visitor transaction.

The Aggregated Log is built by WUM's *Aggregation Service* as explained in the following. Its activities are summarized in Fig. 2. In this figure, we use the shorthands a, b, c, d, e, f for the web pages. We list accesses of three visitors between 11:42 a.m. and 12:34 p.m.

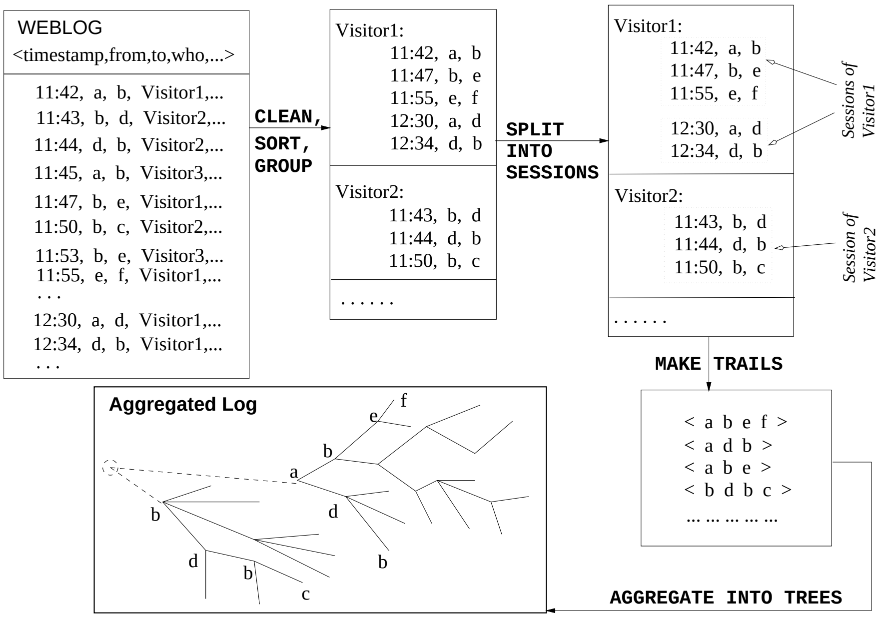

Fig. 2. Construction of the Aggregated Log

3.1 Cleaning the Log and Grouping the Entries into Transactions

The web server log is first "cleaned" by removing entries that should not be taken into account during mining. Such entries may be those recording accesses to image files [4] or accesses performed by a special group of visitors, such as robots. The types of entries to be removed by *Aggregation Service* can be interactively specified in the user interface [15].

The *Aggregation Service* must next decide whether any two subsequent accesses to web pages stem from the same visitor. Methodologies based on the usage of cookies, user registration, exploitation of knowledge on the network's topology etc have been proposed to this purpose [5, 8]. Currently, our *Aggregation Service* assumes that accesses from the same host come from the same visitor. An extended discussion on this issue can be found in [5, 11].

3.2 Identifying the Transactions of a Visitor

In the next step of the process depicted in Fig. 2, consecutive entries of a visitor are grouped into "transactions". A transaction consists of the recorded activities of the visitor within one session. Hence, the *Aggregation Service* decides here whether two subsequent entries belong to the same session of the visitor or not.

We support two criteria for grouping consecutive web page accesses of a visitor into a transaction: (i) a maximal duration or (ii) a maximal elapsed time between any two subsequent page accesses. According to the former criterion, the elapsed time between the first and last entry of a transaction may not exceed a threshold value. According to the latter criterion, two consecutive entries belong to the same transaction if the elapsed time between them is lower than a(nother) threshold. In both cases, if the latest entry occurs at a timepoint beyond the threshold value, it is assigned to a new transaction.

Other criteria for grouping entries into transactions are proposed e.g. in [4]. They can be seamlessly incorporated into the *Aggregation Service* of WUM.

After grouping entries into transactions, transactions are modelled as sequences in the next step. To do so, the *Aggregation Service* removes unnecessary information like the visitor identification and the timestamp. It then merges consecutive entries into a sequence $< fromPage_1, \ldots, fromPage_n, toPage_n >$. We call this sequence a "trail". Note that after the removal of visitor identifiers and timestamps, there are many identical trails.

3.3 Merging Trails into an Aggregate Tree

In the final step, the *Aggregation Service* merges the visitor trails along common prefixes into a tree structure, the "aggregate tree". The "Aggregated Log" is a large persistent aggregate tree composed of all trails. To bypass the fact that not all trails start at the same page, we introduce a dummy root node, from which the merged trails emanate.

A node in an aggregate tree corresponds to the occurence of a page in a trail. Common trail prefixes are identified, and their respective nodes are merged into a tree node. This node is annotated with the number of visitors having reached the node across the same trail prefix. We call this the "support" of the node.

Example 1. At the left side of Fig. 3 we show the topological graph of a tiny web site and a number of trails recorded in the web log. Along with each trail we show the number of visitors that have followed it.

In Fig. 3, `a` is the first page of trails 1, 4 and 5; by summing up the visitors having traversed them, we compute 21 as the support of `a`. In trails 1, 4, page `b` was visited after `a`; the respective trie node has a support of 11. Note that trails 2, 3, 6 starting at `b` cannot be merged with trails 1 and 4 starting at `a`.

In trails 2 and 6, page `b` has been accessed twice. When constructing the aggregate tree, we see that a total of 13 visitors accessed `b` (via trails 2, 3, 6), but only 6 of them came back to `b`, across trails 2 and 6 that have the same prefix. We must distinguish between different occurences of the same page, so that page revisits can be discovered. Hence, during trail merging, the *Aggregation Service* assigns to each page its occurence number.

The aggregate tree of Fig. 3 has a dummy node `#` as root. This convention allows us to model the Aggregated Log as a single large aggregate tree. The support of `#` is the total number of entries recorded in the weblog.

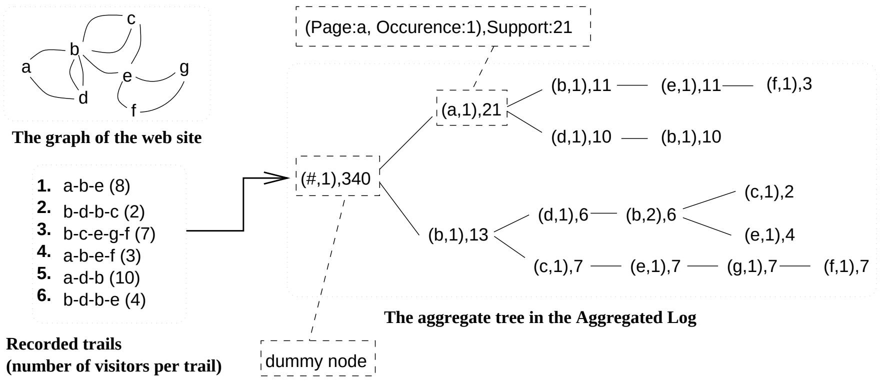

Fig. 3. Merging trails in the Aggregated Log

3.4 Navigation Patterns over the Aggregated Log

On Fig. 3 we can see that trails with different prefixes may still have pages in common. If we merge those trails at their common nodes, we construct an aggregated graph that is no more a tree: In fact, this is a "navigation pattern".

Example 2. Before introducing navigation patterns formally, let's see how they can be constructed from the Aggregated Log of Fig. 3. Suppose we are interested in the navigation patterns between `b` and `e`. To construct them, we first discover the branches of the Aggregated Log containing them. From the figure, there is one branch in the subtree rooted at `(a,1),21` and two branches rooted at `(b,1),13`. Note that the branch `(b,1),13--(d,1),6--(b,2),6--(e,1),4` reveals that there are *two* navigation patterns between `b` and `e`: one between `(b,1)` and `e,1)` and one between `(b,2)` and `e,1)`.

To produce those navigation patterns, we merge the branches at the common nodes. Then, we have to recompute the supports of the "junction" nodes that were produced after the merging. To do so, we just sum up the supports of the original nodes. Hence, the support of `(b,1)` in the first pattern (Pattern 1) is $11 + 13$, produced by summing the support of the node below `(a,1),21` and of the node root of the second subtree below the dummy root. Note that `(b,1),13` belongs to two branches but is only one node; hence, it is not used twice in the summation. The two navigation patterns are shown in Fig. 4.

Pattern 1 shows that 22 of the 24 visitors of `(b,1)` have reached `e` across different routes. 6 of them returned to `b`. Only 4 of those 6 reached `e` afterwards. This last fact cannot be deduced from pattern 1; it is assessed from pattern 2.

One might ask why we use a tree structure instead of a navigation pattern as our Aggregated Log. The reason is that trees can be built without information loss, whilst patterns cannot: In our simple example, a closer observation of Pattern 1 reveals that we lost the number of visitors that have accessed `e` after reaching `b` for the second time. Hence, navigation patterns are not explicitly stored but extracted from the Aggregated Log by the miner.

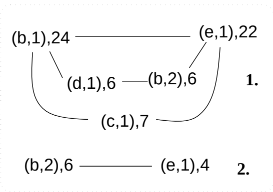

Fig. 4. Two navigation patterns over the Aggregated Log

4 Construction of the Navigation Patterns

By the end of the previous section, we have built navigation patterns by means of examples. To formally describe the notion of navigation pattern and its construction procedure, we must analyze the formalism underlying WUM [10]. Due to space limitations, we rather present a simplified construction method. In particular, instead of defining navigation patterns on the basis of the Aggregated Log, from which they are actually extracted, we define them on the basis of the trails produced by the *Aggregation Service* at an intermediate step.

4.1 Patterns and Pattern Descriptors

We denote by *Trails* the collection of trails extracted by the *Aggregation Service* from the web server log. *Trails* may contain duplicates, so it is not a set but a "bag" (synonym: "multiset"). Hereafter, we refer to the terms "bag" and "multiset" interchangeably and we use $\{\!\!\{\ldots\}\!\!\}$ as the delimiter pair for bags.

Before defining a "navigation pattern", we introduce the formal notion of descriptor for a navigation pattern.

Definition 1. *Let WPO be the set of Web Page Occurences, where a page occurence is a pair comprised of a page URL and an occurence number. Further, let $\star$ be a wildcard denoting a sequence of zero or more nodes, and $\_$ a wildcard denoting exactly one node. We denote by WPO_+ the set $WPO \cup \{\star, \_\}$.*

A "pattern descriptor" is a vector $a \in (WPO_+)^*$, such that any two $\star$-elements in a are separated by at least one page occurence from WPO.

Thus, a pattern descriptor is a vector of page occurences and wildcards. We disallow constructs like $\star\star$ which is identical to $\star$, or $\star\_\star$ which is identical to $\_\star$.

We must now specify when a trail y matches a pattern descriptor a. Intuitively, a page occurence is matched only by itself, a $\_$ is matched by one arbitrary trail node, while a $\star$ is matched by an arbitrary subtrail. We generalize this into a predicate $matches(a, y, k)$ which is true if y matches a up to position k.

Definition 2. *Let a be a pattern descriptor, whose non-wildcard elements are at positions $k_1, \ldots, k_n$. The "navigation pattern with descriptor" a is a graph $navp(a)$. It is built by discovering the trails that contain a subtrail matching a, extracting those subtrails, merging them by common prefix and at the positions $k_1, \ldots, k_n$ and summing up the support values.*

This rather informal definition indicates that the navigation pattern is defined in a constructive way. In the following, we analyze the two steps of the pattern construction process and present an algorithm for each step. Those algorithms are conceptual, though, intended to make the principles of navigation pattern construction clear. The algorithms used in WUM process the Aggregated Log instead of individual trails and are much more complicated.

4.2 The Subtrails Satisfying a Descriptor

A pattern is formed from the subtrails satisfying the predicate $matches(\cdot)$. Hence, we must detect and extract those subtrails from the trails *Trails*. To do so, we first identify the page occurences contained in the pattern descriptor. We then extract the subtrails that satisfy the pattern descriptor up to each page occurence in an iterative way. We show the algorithm in Fig. 5.

Conceptual Algorithm-1: Extracting the subtrails that match a descriptor

Input: *A pattern descriptor* a *with page occurences at positions* $k_1, \ldots, k_n$ *and a bag* X *originally set equal to Trails.*
Output: *A list of bags* $bag\_list(a)$ *of matching subtrails*
Method: *We iterate on the contents of* X *for* $i = 1, \ldots, n$.
After each iteration, X *contains the trails* x *for which* $matches(a, x, k_i)$ *is true.*

1. *From each trail* x *in* X*, for which* $matches(a, x, k_i)$ *is true, we extract the suffix* $y = a[k_i] \cdot z$.
   Those suffixes form a bag Y_i.
2. *We remove from* X *all trails that do not contribute to* Y_i.
3. *If* Y_i *(or* X*) is empty, we return the empty list and quit.*
   Otherwise:
   If $i = n$*, we are done and can go to Step 4.*
   Otherwise, we continue from Step 1 for $i = i + 1$.
4. *We return the list* $< Y_1, \ldots, Y_n >$.

Fig. 5.

By Step 1 of the first iteration, X = *Trails*. If the descriptor starts with a page occurence (i.e. $k_1 = 1$), we look for this page occurence anywhere in each trail. Otherwise, the wildcards before k_1 determine how many elements prior to $a[k_1]$ a trail may or should have. Non-matching trails are deleted from X.

By Step 1 of iteration i, X is already reduced to the trails that match the descriptor a up to position k_{i-1}. Among them, we select those matching the descriptor up to position k_i; in Step 2, we remove all other trails from X.

For each trail matching the descriptor up to position k_i, we store the suffix starting at $a[k_i]$ in a bag Y_i. Thus, Y_i contains all subtrails that start with a node referring to $a[k_i]$ and originate from trails matching the descriptor up to $a[k_i]$. Note that each trail in X contributes at most one suffix to Y_i: since $a[k_i]$ is a page occurence, it can appear in a trail at most once.

After the last step of our algorithm, X contains the subtrails that match the whole pattern descriptor. The reader may therefore wonder what we need the Y_i bags for. We explain this in a small example.

Example 3. Let *Trails* $= \{\!\{c, dac, acb, ad\}\!\}$ and let `<(a,1),*,(b,1)>` be a descriptor. We can directly see that its pattern is `(a,1),3--(c,1),2--(b,1),1`. It indicates that from the 3 visitors of `a` only 1 reached `b` via `c`, while a second one stopped at `c`.

We now use our algorithm to find the trails constituting this pattern. X is initially equal to *Trails*. In the first iteration, it is reduced to $\{\!\{dac, acb, ad\}\!\}$. $Y_1 = \{\!\{ac, acb, ad\}\!\}$. In the second iteration, $X = \{\!\{acb\}\!\}$ and $Y_2 = \{\!\{b\}\!\}$. Now, if we compute the support of each node in the navigation pattern from X, we obtain a value equal to 1 for *all* nodes. To obtain the correct support values, we must build the aggregate tree of Y_1. This is done by the algorithm presented in the next subsection.

It should be noted that the bag Y_i can be directly built from the bag Y_{i-1}, so the computation of X is actually redundant. However, a correctness proof is necessary for this simplification, which falls beyond the scope of this study.

4.3 Constructing a Pattern as a Graph

In Def. 2, we state that the navigation pattern with descriptor a is a graph $navp(a)$. Conceptual Algorithm-1 extracts the subtrails matching the descriptor's parts in an incremental way and stores them in $bag\_list(a) = Y_1, \ldots, Y_n$.

To build the graph, we first merge the subtrails in each bag Y_i on common prefix to form the aggregate tree $aggTree(Y_i)$, as described in 3.3. According to Step 1 of the Conceptual Algorithm-1, all those subtrails start at $a[k_i]$. The aggregation ensures that each element of Y_i that does not contain $a[k_{i+1}]$ but has a common prefix with another element that does is still taken into account when the statistics for the common prefix are computed.

After building $aggTree(Y_i)$, we can safely remove branches that do not satisfy the part of the descriptor between positions k_i and k_{i+1}. We then discard all subtrees rooted at $a[k_{i+1}]$: all subtrees of interest are already contained in $aggTree(Y_{i+1})$. Finally, we connect the affected parent nodes to the root of $aggTree(Y_{i+1})$, which refers to $a[k_{i+1}]$ by definition. The complete algorithm is shown in Fig. 6.

For each aggregate tree $AT_i = aggTree(Y_i)$, this algorithm gradually removes all (sub)branches that do not end at $a[k_i + 1]$. This is done at Step 3(c). Since the traversal of AT_i is postfix, we access the parent after having processed the children. Hence, if the parent of a removed node becomes a leaf, it will also be removed in a subsequent iteration.

Conceptual Algorithm-2: Building the navigation pattern as a graph.

Input: *A pattern descriptor* a *with page occurences at positions* $k_1, \ldots, k_n$
Output: *The "navigation pattern with descriptor"* a*, which is a graph* $navp(a)$
Method:

1. *We build the* $bag\_list(a) =< Y_1, \ldots, Y_n >$ *by Conceptual Algorithm-1.*
2. *We form the* $AT_1 = aggTree(Y_1), \ldots, AT_n = aggTree(Y_n)$.
3. *On each of* $AT_1, \ldots, AT_{n-1}$*, we perform the following depth-first postfix traversal:*
   (a) *We find the next node* N *of* AT_i *that is a leaf or refers to* $a[k_{i+1}]$.
   (b) *If* N *refers to* $a[k_{i+1}]$*, then*
      – *we discard the subtree rooted at* N
      – *and we link the parent of* N *to the root of* AT_{i+1}.
   (c) *We discard* N.
   (d) *We can now go to Step 3.(a) to find the next* N *within* AT_i.
4. *We return the linked graph structure as* $navp(a)$.

Fig. 6.

4.4 An Example on the Construction of Patterns

We now explain the above definitions in an example. We use the Trails in Fig. 3.

Example 4. Let $a =< (b, 1), *, (e, 1), *, (f, 1) >$ be the pattern descriptor. To build its navigation pattern, we first build the bag list $bag\_list(a) =< Y_1, Y_2, Y_3 >$ using Conceptual Algorithm-1. In the following table, we show the contents of Y_1, Y_2, Y_3, as well as the contents of X after each iteration.

On the table, we have annotated each trail with with the number of times it appears in the *Trails*. Then, a subtrail extracted from it appears the same number of times. We further omit occurence numbers for brevity. This has no side effects, because only the first occurence of b qualifies.

	Trails	$X.1$	Y_1	$X.2$	Y_2	$X.3$	Y_3
(8)	abe	abe	be	abe	e	—	—
(2)	bdbc	bdbc	bdbc	—	—	—	—
(7)	bcegf	bcegf	bcegf	bcegf	egf	bcegf	f
(3)	abef	abef	bef	abef	ef	abef	f
(10)	adb	adb	b	—	—	—	—
(4)	bdbe	bdbe	bdbe	bdbe	e	—	—

In this example, $X.1 = Trails$ because all trails contain the element b.

The aggregate trees and the output navigation pattern built according to Conceptual Algorithm-2 are shown in Fig. 7. The retained branches of the trees are enclosed in lassos.

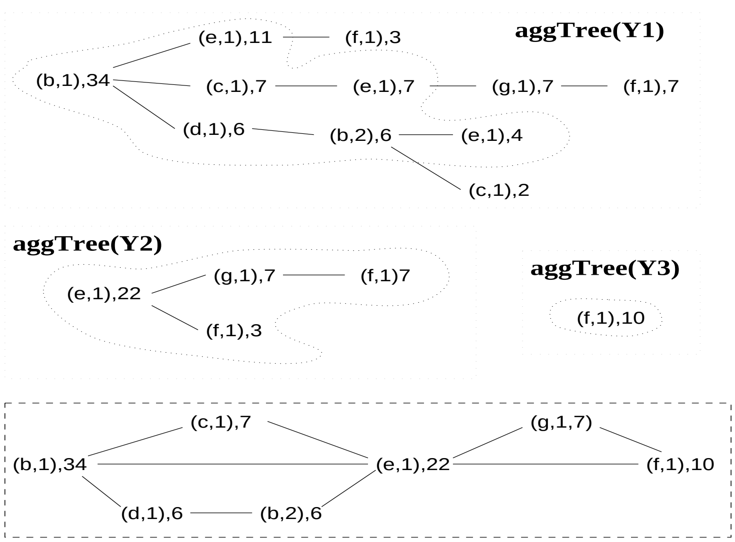

Fig. 7. The aggregate trees and result pattern for descriptor $<(b,1),*,(e,1),*,(f,1)>$

5 Knowledge Discovery Queries

Similarly to [6], we believe that good mining results require a close interaction of the human expert and the mining tool, in which the expert uses her domain knowledge to guide the miner. Therefore, WUM provides a mining query language, MINT , with which the expert can specify the subjective characteristics that make a navigation pattern of interest to her.

5.1 The Mining Language MINT

Specifying the interesting characteristics of a pattern. The notion of interestingness based on beliefs is discussed in [9]: a belief is a rule of the form $A \rightarrow B$, which is expected to be true. The same study proposes mechanisms for the verification of beliefs and the discovery of belief violations *in the context of association rules*. To our knowledge, there is no respective formalism for beliefs on sequential patterns. However, MINT allows the specification of beliefs or belief violations as predicates. Via predicates one can also specify the structure or statistics a navigation pattern should have to be of significance. Thus, besides the classical statistical thresholds, more elaborate criteria are supported.

In particular, the human expert issuing queries in MINT may specify the interesting aspects of the patterns to be discovered in terms of:

Content: The user may specify properties of the content or meta-data of pages that should appear in the navigation pattern. Occurences of pages may be also specified, if the user is interested in discovering cycles.

Statistics: These include the support of a page within the pattern to be built or within the Aggregated Log as a whole, as well as the confidence of a page with respect to a previously accessed page.

Structure: The user may specify the order of the pages, adjacency of visited pages, length of subtrails between pages etc.
For this, MINT supports the notion of "template" for a pattern descriptor. A template is a vector of variables interleaved with wildcards. During query execution, the variables are bound, resulting in one descriptor per binding.

The schema and syntax of MINT queries. The syntax of MINT reminds of SQL. However, MINT is not a conventional query language: Instead of selecting tuples, it discovers patterns. Instead of accessing a database or data warehouse, it is applied on the Aggregated Log presented in subsection 3.3.

In Fig. 8, we show the conceptual schema used for query formulation. This schema reflects our definitions: A navigation pattern is a graph, the nodes of which refer to web pages. The properties of the entities can be used in the query predicates. The primary keys are underlined.

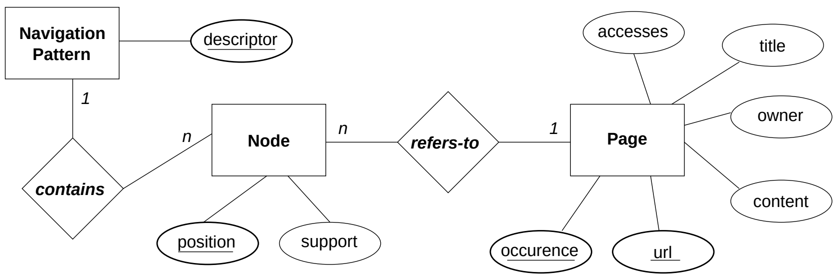

Fig. 8. The schema of the Aggregated Log (using ER-notation)

A navigation pattern is determined by its descriptor. In MINT we specify "templates" for descriptors, i.e. vectors of node variables and wildcards. The statement `TEMPLATE a*b AS t` specifies that we are interested in descriptors of the form `a*b`, where `a` and `b` are variables. Each pair of nodes that can be bound to `a` and `b` results in a descriptor and a navigation pattern satisfying it.

The above template is not very convenient. In data mining, we want to focus the discovery process by statistical thresholds and other interestingness measures. Those measures are applied on the node variables of the template. With the statement `NODE AS a b` we declare the two node variables for the template `a*b` above. According to the schema of Fig. 8, we can specify thresholds on the support of those nodes. The predicates:

```
a.support >= 1000  AND b.support / a.support >= 0.9
```

describe the usual specification of a support threshold (here: 1000 visits to this node) and of a confidence minimum (here: 0.9).

One may notice that the entity `Node` in our schema has no property "confidence", so that the confidence must be described by means of supports. We show our motivation for this decision in a small example:

Example 5. We consider pattern descriptors with three variables:

```
SELECT t
FROM NODE AS a b c, TEMPLATE a*b*c AS t
WHERE b.support >= 1000
AND c.support / b.support >= 0.8
AND c.support / a.support >= 0.6
```

The support of `b` is the summation of the supports of paths leading from `a` towards it. The same holds for `c` in the context of `a, b`. Hence, the confidence of `c` is expressed by the second predicate above. The last predicate does not express any confidence but can still be observed as a reasonable measure. Therefore, we have decided for the generic specification of any confidence-like measure by means of arithmetic operations on the support.

Beyond the specification of thresholds for statistical measures, MINT allows restrictions on the pages referenced by the node variables. Such restrictions are appropriate to focus mining to some interesting types of pages only, to refine some mining results or to drill down into the Aggregated Log information.

According to the schema of Fig. 8, we retain for a `Page` some meta-information gained from the HTML-file, the URL of the page and the total number of `accesses` to the page, independently of the trails followed to the page. Since a trail may contain backward moves to already visited pages, a page may occur more than once in a trail. As we cannot model cycles in the Aggregated Log without losing statistical information, we "flatten" cycles by attaching to each page an `occurence` number. This introduces redundancy in the schema shown in Fig. 8. In our implementation, we obviously avoid such redundancies.

5.2 Querying in MINT

A MINT query produces a set of navigation patterns extracted from the Aggregated Log according to the query predicates. In the following, we present the functionality of MINT by means of examples. To make those examples more intuitive, we introduce another Aggregated Log on a fictious site with HTML-pages on middleware. This Aggregated Log is shown on Fig. 9.

Paths between web pages. A generic question on web usage concerns the paths followed between two or more web pages. Usually, only certain types of pages are of interest, e.g. pages on Corba from our example Aggregated Log. Typically, restrictions on the statistics of those paths are also placed, e.g. to discover the paths dominantly or most rarely used. Query **Q1** below constructs the navigation patterns from a page on middleware to a page on Corba, which has been followed by more than 20% of the visitors of the page on middleware.

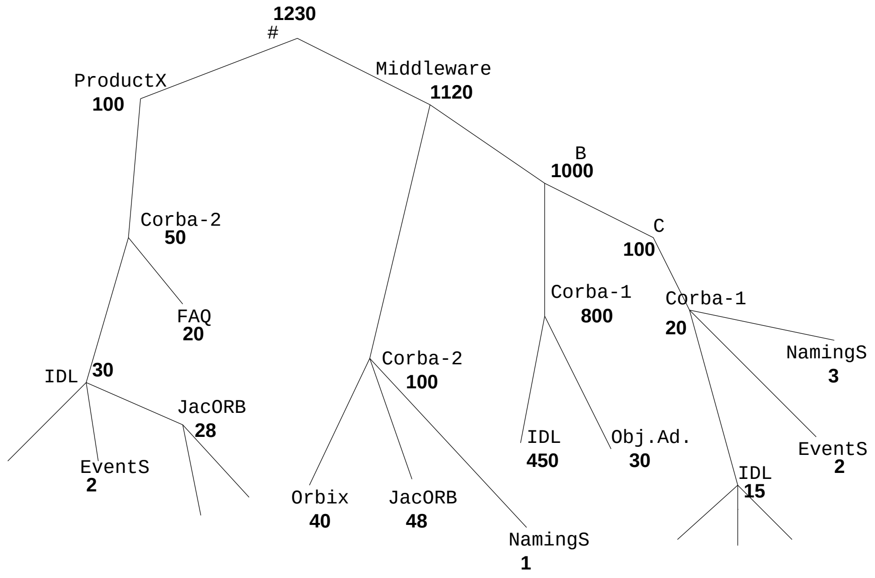

Fig. 9. The Aggregated Log of a fictious example web site

```
SELECT t
FROM NODE AS a b, GRAPH TEMPLATE a*b AS t
WHERE a.title CONTAINS "Middleware"
AND b.title CONTAINS "Corba"
AND b.support / a.support > 0.2
```

By comparing this query to the schema of Fig. 8, we notice that the `title` is a property of a page, not of the node referring to a page. So, the syntax of MINT hides part of the schema complexity from the user. The result of **Q1** is shown in Fig. 10 and explained below.

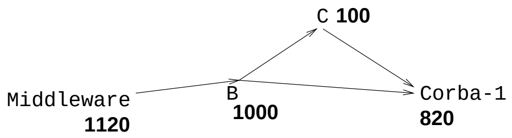

Fig. 10. The result of query **Q1**

Conceptually, the *MINT-Processor* evaluates this query as follows: All nodes in the Aggregated Log referring to Middleware are discovered. The branches from them to a page on Corba are found, thus producing two pattern descriptors, `<(Middleware,1),*,(Corba-1,1)>` and `<(Middleware,1),*,(Corba-2,1)>`. Descriptors are built incrementally; candidates without corresponding patterns are pruned out at each step.

Next, the pattern for each descriptor is constructed. Instead of applying the conceptual mechanism of section 4 on the trails, the *MINT-Processor* operates on the Aggregated Log [11]: The branches matching each descriptor are extracted from the Aggregated Log and aggregated by merging nodes belonging to common prefixes and nodes referring to the page occurences of the descriptor. The support values of the merged nodes are summed up to compute the support of the resulting node. Then the last predicate of **Q1** is evaluated.

Only the first of the two descriptors above satisfies this predicate. It produces the navigation pattern shown in Fig. 10.

Statistical thresholds. Query **Q1** contained two rather restrictive predicates on the contents of the web pages. If only statistical thresholds are considered, the number of candidate descriptors considered by the *MINT-Processor* is significantly larger. Consider **Q1'**:

```
SELECT t
FROM NODE AS a b, GRAPH TEMPLATE a*b AS t
WHERE a.support >= 1000
AND b.support / a.support > 0.7
```

This query produces the navigation pattern of Fig. 10, the navigation patterns `(Middleware,1),1120---(B,1),1000`, `(B,1),1000---(Corba-1,1),800` and some navigation patterns starting at the dummy root node and ending at the node on Middleware or at node B.

Results in Tree or Graph form. "Where do visitors of page `a` go afterwards?" Some restrictions are often placed on content and support of `a` and of the subsequent pages, as in the following query **Q2**:

```
SELECT t
FROM NODE AS a b, TREE TEMPLATE a*b AS t
WHERE a.support > 100
AND a.title CONTAINS "Corba"
AND b.support / a.support > 0.5
```

Here, we receive all URLs reached after a page `a` on Corba, which has been accessed 100 or more times, provided that those URLs have been accessed by more than 50% of the visitors of `a`. We have however specified that the template should be a `TREE` instead of a `GRAPH`. This means that the result will be an aggregate tree.

More precisely, **Q2** is executed as follows: All nodes in the Aggregated Log whose title refers to Corba-1 or Corba-2 are retrieved. All subtrees rooted at each such node are identified, their common prefixes are merged as described in section 3, and the supports of the nodes of the new tree are computed. However, differently from the approach for navigation pattern s, the leaf nodes need not be identical, nor are identical leaf nodes merged together. Rather, the last query predicate is evaluated for each branch of the tree separately.

In Fig. 11, we show the intermediate aggregate trees produced for **Q2** and the tree satisfying all query predicates. At the lower right corner we show the two navigation patterns output by query **Q2'**, which is the same as **Q2** but for using the `GRAPH` option for the template. The different results are caused by the different values of `b.support` in the descriptors for the tree and the graph.

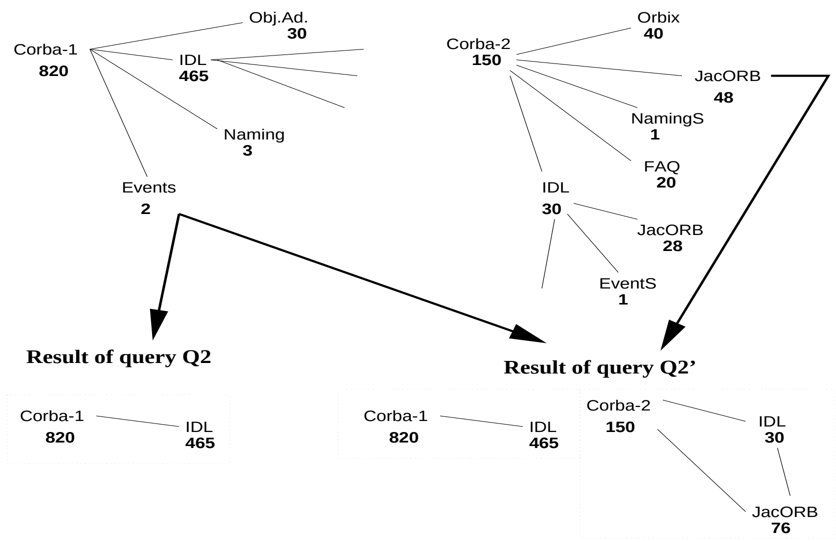

Fig. 11. aggregate trees and graph patterns as query output

The reader may have noticed that if the pages on Corba-1 had not been merged, the path from Corba-2 to IDL would have satisfied the predicates. To avoid this merging, the `TREE` template should be changed into `#*a*b`, where `#` indicates that the navigation pattern should start at the dummy root.

Revisited nodes. The study of [12] showed that 58% of accesses in the studied domain were node revisits, also called "recencies". For a given site, this statement can be verified by issuing a query that sums up the support of all revisited nodes and compares it to the total number of accesses. Recall that this number is the support of the dummy node ^. This test is performed in the following query **Q3**:

```
SELECT SUM(a.support) / ^.support
FROM NODE AS a, TREE TEMPLATE #*a
WHERE a.occurence = 2
```

Note that the occurence number must be exactly 2. Otherwise, nodes revisited more than once would contribute to the support summation multiple times.

As can be seen from this query, conventional aggregation of nodes is also permitted. For a template with multiple variables, e.g. `#*a*b`, grouping would also be of interest: A clause "`GROUP BY a`" would compute the summation of the supports of all distinct values of `b` within the context of a given `a` value. The full syntax is given in the Appendix.

The study of [12] also states that recencies to any *specific* node do not exceed 19% of the total accesses. To find nodes revisited more often than that we can issue query **Q4**:

```
SELECT b
FROM NODE AS a b, TREE TEMPLATE #*a*b
WHERE a.url = b.url
AND a.occurence = 1
AND b.occurence = 2
AND b.support / a.support > 0.19
```

5.3 Complexity of the Mining Process

The *MINT-Processor* is responsible for identifying common patterns in the large aggregate tree of the Aggregated Log, merging them to aggregate graphobjects, computing the node supports and evaluating the query predicates. Any navigation patterns not satisfying the predicates should be recognized and filtered out as soon as possible.

This mechanism can be quite expensive: A query asking for all navigation patterns containing nodes with supports beyond a threshold will imply considering all possible variable bindings. We are working on optimization techniques to alleviate this problem. One of them is to retain for each web page a list of the nodes corresponding to it, thus trading speed for space.

Still, our *MINT-Processor* is de facto faster than a miner operating on web log data [5,3], because of the off line aggregation of weblog information into the Aggregated Log. Moreover, our mechanism can exploit restrictive predicates to avoid building uninteresting navigation patterns. A miner not accepting ad hoc predicates needs a postprocessing module that removes all unnecessarily computed patterns.

6 Related Work

The subject of discovering navigation patterns in web logs has been addressed in the studies of [3, 5, 13]. The "Footprints" tool of [13] focusses on the visualization of frequently accessed patterns and on the identification of pattern *types* that may be of importance [14]. Our work on MINT is complementary to Footprints, since we are interested in ways of formally describing such types.

The studies of [3, 5] focus on pattern discovery. They are both based on the adaptation of an existing miner to the particular problem. More precisely, a preprocessing algorithm groups consecutive page accesses of the same visitor into a transaction, according to some criterion. Then, a miner for association rules or sequential patterns is invoked to discover similar patterns among the transactions. In [5], the association rules' miner has been further customized to guarantee that no feasible patterns are erroneously skipped.

This approach has some drawbacks. First of all, the generic characteristics that make a navigation pattern interesting are different than those making an association rule interesting, and which are studied e.g. in [9]. Interestingness in the mining of sequential patterns is expressed in rather simple criteria on the support/confidence and length of the patterns [7]. Hence, the miners used in [3,5] do not allow an expert to guide the mining process to discover e.g. node revisits, patterns of unexpectedly low support etc.

This problem is alleviated in [5] by the query language, with which the expert can give instructions to the miner. The language of [5] is syntactically similar to MINT but, in contrast to MINT, it is only a front end to an existing miner. Thus, instructions that cannot be understood by the miner must be postponed to a postprocessing phase: the interestingness of some plans cannot be tested prior to this phase. This decreases the overall system performance.

A performance drawback of miners on sequential patterns [1, 7] is caused by activating them over the whole set of transactions. By aggregating transactions into an Aggregated Log, WUM guarantees a performance improvement at least linear to the degree of similarity between transaction prefixes in the original log.

Our work is conceptually related to the study of [6] for text mining. A query language is introduced to drive a miner that discovers association rules in document collections. This miner is applied on *aggregated data*, organized in a trie structure [2] in a way similar to our Aggregated Log. The performance gains of using such an aggregated representation are demonstrated in [6].

However, the exploitation of the aggregated information in [2] is different than in WUM: Association rules have no order, so any rule can be discovered by traversing a single subtree of the trie. Web navigation patterns have order, so that the discovery of all matching branches of the large aggregate tree in our Aggregated Log are necessary. Thus, the *MINT-Processor* uses a much more complicated technique than a simple trie traversal.

7 Conclusions

We have presented the Web Utilization Miner, our system for the discovery of interesting navigation patterns in a web site. We have primarily focussed on the MINT query language and its execution mechanism. MINT can be used by a human expert to guide the mining system by specifying the generic structural and statistical characteristics that make a navigation pattern interesting.

WUM is intended for web site authors and administrators who are trying to improve the organization of their web documents and adapt it better to the needs of the information consumers. This applies to research and educational institutions, but also to other organizations, e.g. public authorities providing citizens with environmental information.

We have thus far implemented the *Aggregation Service*, the MINT processor and the visualization tool for presenting the discovered navigation patterns to the user. We currently experiment with WUM on the usage of educational hypermedia. Our ongoing research focusses on extensions for MINT, on optimization techniques to reduce the complexity of pattern discovery, on the formulation of beliefs for navigation patterns and on the design of a template generator.

Acknowledgment. We would like to thank Karsten Winkler, diplom student in the Humboldt-University Berlin for his work on the implementation of WUM and the design of the user interface and the persistent storage manager. We also wish to thank the anonymous reviewers for many useful comments.

References

1. Rakesh Agrawal and Ramakrishnan Srikant. Mining sequential patterns. In *ICDE*, Taipei, Taiwan, Mar. 1995.
2. Amihood Amir, Ronen Feldman, and Reuven Kashi. A new and versatile method for association generation. *Information Systems*, 22:333–347, 1997.
3. Ming-Syan Chen, Jong Soo Park, and Philip S Yu. Data mining for path traversal patterns in a web environment. In *ICDCS*, pages 385–392, 1996.
4. Robert Cooley, Bamshad Mobasher, and Jaideep Srivastava. Grouping web page references into transactions for mining world wide web browsing patterns. Technical Report TR 97-021, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, USA, June 1997.
5. Robert Cooley, Bamshad Mobasher, and Jaideep Srivastava. Web mining: Information and pattern discovery on the world wide web. In *ICTAI'97*, Dec. 1997.
6. Ronen Feldman, Willi Klösgen, and Amir Zilberstein. Visualization techniques to explore data mining results for document collections. In *KDD'97*, pages 16–23, Newport Beach, CA, Aug. 1997. AAAI Press.
7. Heikki Mannila and Hannu Toivonen. Discovering generalized episodes using minimal occurences. In *KDD'96*, pages 146–151, 1996.
8. Peter Pirolli, James Pitkow, and Ramana Rao. Silk from a sow's ear: Extracting usable structures from the web. In *CHI'96*, Vancouver, Canada, Apr. 1996.
9. Avi Silberschatz and Alexander Tuzhilin. What makes patterns interesting in knowledge discovery systems. *IEEE Trans. on Knowledge and Data Eng.*, 8(6):970–974, Dec. 1996.
10. Myra Spiliopoulou. The laborious way from data mining to web mining. *submitted*, June 1998.
11. Myra Spiliopoulou, Lukas Faulstich, C., and Karsten Winkler. Discovering Interesting Navigation Patterns over Web Usage Data. *submitted*, 1998.
12. Linda Tauscher and Saul Greenberg. Revisitation patterns in world wide web navigation. In *CHI'97*, Atlanta, Georgia, Mar. 1997.
13. Alan Wexelblat. An environment for aiding information-browsing tasks. In *Proc. of AAAI Spring Symposium on Acquisition, Learning and Demonstration: Automating Tasks for Users*, Birmingham, UK, 1996. AAAI Press.
14. Alan Wexelblat. *Footprints: History-Rich Social Navigation.* PhD thesis, MIT Media Laboratory, Dec. 1997.
15. Karsten Winkler. Realization and testing of a data mining environment for the discovery of user navigation patterns in a web site. Master's thesis, Institut für Wirtschaftsinformatik, HU Berlin, Oct. 1998. to appear on german.

A The Syntax of MINT 1.0

Query structure

```
query::= 'SELECT' selectList
         fromClause [whereClause]
         [groupClause [havingClause]]
```

Specifying the query output

```
selectList ::=  ['DISTINCT'] derivedColumn (',' derivedColumn)*
derivedColumn ::=  (valExpr|aggrExpr) ['AS' columnName]
aggrExpression ::= aggrOp '(' ['DISTINCT'] (valExpr|varName) ')'
aggrOp ::=  'AVG' | 'MAX' | 'MIN' | 'SUM' | 'COUNT'
```

In the current version, aggregates are not implemented, so the `groupClause` and the `havingClause` are not supported yet.

Specifying the query template

```
fromClause ::=  'FROM' nodeRef ',' (templateRef)*
nodeRef ::= 'NODE' 'AS' nodeVar*
templateRef ::= [templateType] 'TEMPLATE' template
                ['AS' templateVar]
templateType ::= 'GRAPH' | 'TREE'
template ::= ['#'] [wildcards] (nodeVar [wildcards])*
wildcards ::=  '_' | ('_')* | '*'
```

In the current version only a single template variable is allowed. The default template type is `GRAPH`. The type `TREE` is not supported yet.

Introducing predicates

```
whereClause ::=  'WHERE' condition ('AND' condition)*
condition ::= stringCondition | numericCondition
stringCondition ::= columnReference stringCompOp stringLiteral
columnReference ::=  [templateVar '.'] nodeVar '.' columnName
stringcompOp ::=  '=' | 'CONTAINS' | 'STARTSWITH' | 'ENDSWITH'
numericCondition ::= numericExpr numericCompOp numericLiteral
numericExpr ::=  [numericExpr ('+'|'-')] term
term ::=  [term ('*'|'/')] factor
factor ::=  [('+'|'-')] primary
primary ::=  numericLiteral | columnReference
```

In the current version, numeric expressions involve at most two column references and are always linear.

Aggregating and grouping results

```
groupClause ::=  'GROUP' 'BY' groupExpr (',' groupExpr)*
groupExpr ::=  nodeVar | columnRef
havingClause ::=  'HAVING' condition ('AND' condition)*
```