Large-Scale Information Filtering Systems

Fatma Ozcan

May 9, 2000

University of Maryland, College Park

Outline

- Introduction -- information filtering systems
- Index Structures for Information Filtering under the Vector Space Model, by Yan and Garcia-Molina (ICDE 1994)
- Self-Adaptive User Profiles for Large-Scale Data Delivery, by Cetintemel, Franklin and Giles (ICDE 2000)
- Efficient Filtering of XML Documents for Selective Dissemination of Information, by Altinel and Franklin (VLDB 2000)

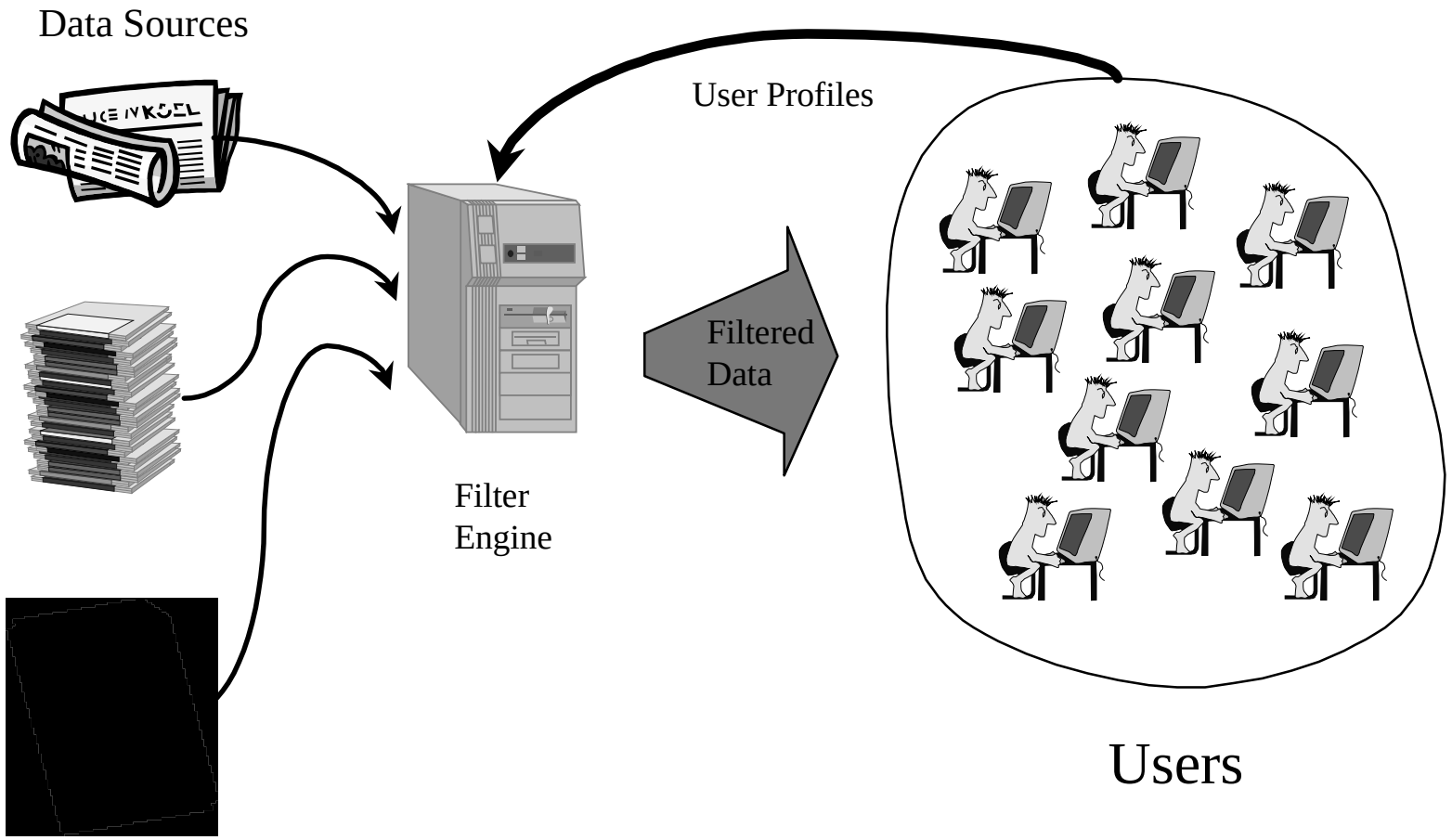
Motivation

- Information is increasingly becoming available in large volumes in electronic form, *high information generation rate*
- Users need mechanisms to keep themselves apprised of new documents that are relevant to their interests
- Efficiency is a major concern in a large-scale information filtering system

Information Filtering Systems

- Users' information needs are specific and change slowly w.r.t. the information generation rate
- Information filtering systems make use of techniques from two research areas, *information retrieval and user modeling*
- User interests are modeled via *user profiles*
- System evaluates user profiles as new information arrives, and *selectively* directs information to interested users

Information Filtering System



Index Structures for IF (YGM94)

- Uses Vector-Space Model to represent documents and profiles
- To compute the vector representation of the document
  - Words that belong to the *stop-list* are removed
  - Stemming is performed
  - The weight of a term is the multiplication of the term frequency factor (*tf*) with the inverse document frequency factor (*idf*)

YGM-94, cont.

- Profiles are represented similarly
- The document and profile vectors are assumed to be normalized by their lengths
- In order to calculate the similarity between a document and a profile, the *cosine measure* is used, that is

$$sim(D, P) = D.\ P =$$

YGM-94, cont.

- In a filtering systems ranking does not make much sense, how many of those documents should be sent to a user?
- Instead, a user may provide some kind of absolute *relevance threshold*
- Given a profile *P* and a relevance threshold θ, a document *D* is *relevant* to *P* if *sim(D, P)* > θ

Brute Force (BF) Method

- Profiles are stored sequentially on disk
- When a document arrives, first its vector representation is calculated
- Then, each profile is examined in turn
- A document is relevant to a profile if the cosine measure is greater than the relevance threshold associated with each profile

Data Structures

- For each profile the following information is stored; a profile-identifier, the length (number of terms in the profile), the (term, weight) pairs, and the relevance threshold
- An inverted index is kept for *profiles*
- For each term *x*, profiles that contain *x* are collected to form an inverted list
- The mapping from terms to their locations in the inverted index is implemented as a hash-table, called the *directory*

Data Structures, cont.

- The *directory* is assumed to fit in memory, while the inverted index is kept on disk
- Two arrays, *THRESHOLD* and *SCORE*
  - *THRESHOLD* contains the relevance threshold for each profile
  - *SCORE* contains the similarity score computed for each profile

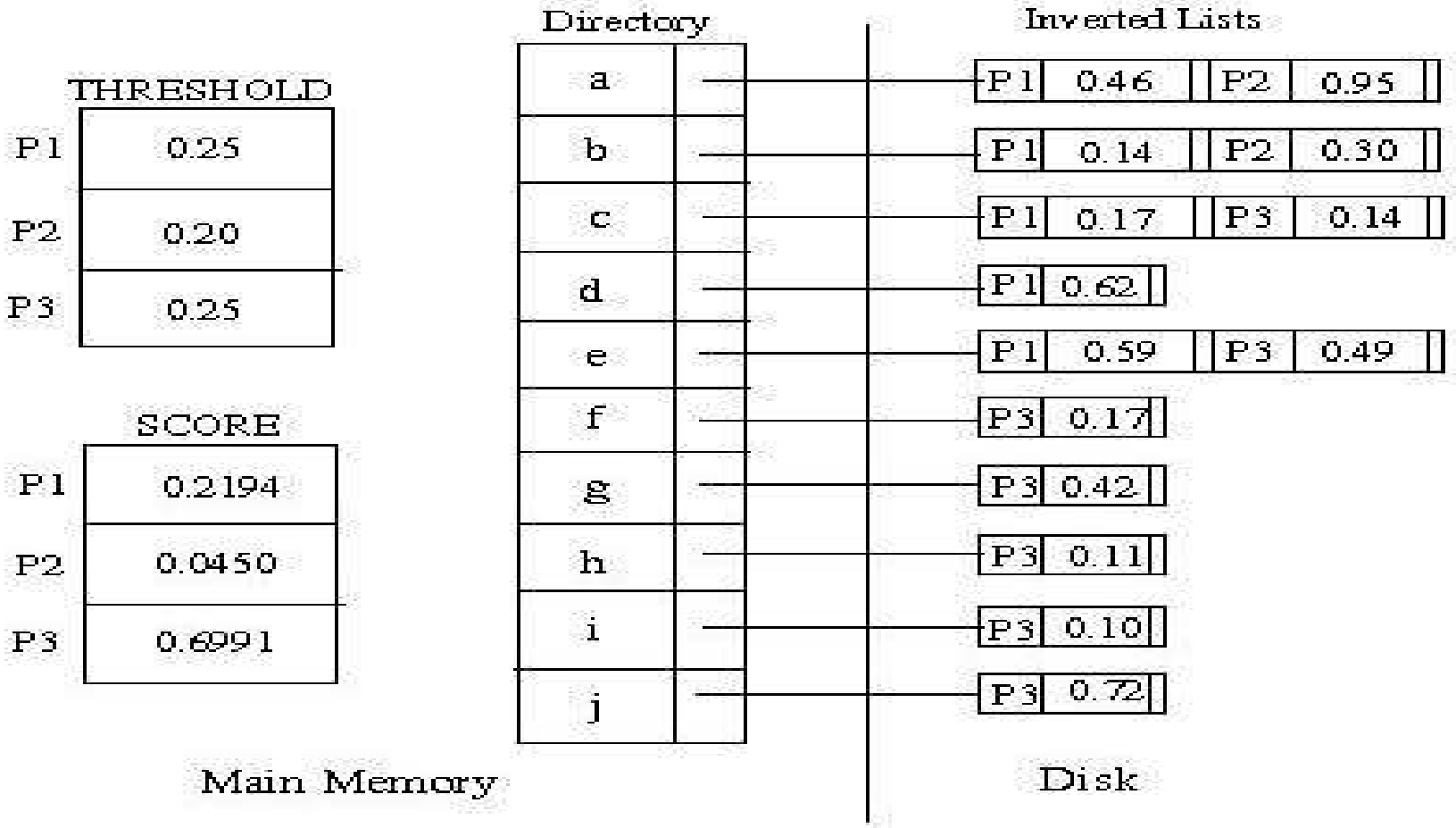
Profile Indexing (PI) Method

- When a document arrives, the *SCORE* array is initialized to 0's
- For each *x with weight w in D*, the directory is used to locate and retrieve *x*'s inverted list
- Each profile *P* in the list is processed
- Let the weight of *x* be *u* in *P*, *SCORE[P]* is incremented by $w * u$

PI Method, cont.

- After all document terms are processed, a profile whose *SCORE* entry is greater than its *THRESHOLD* entry matches the document
- This method greatly reduces the number of profiles examined for a given document

Yan & Garcia-Molina (ICDE'94)



Selective Profile Indexing (SPI) Method

- If we can determine that some terms are insignificant, that is their weights are not large enough by themselves to exceed the relevance threshold, then we do not have to store them in the inverted list
- However, an insignificant term together with a significant one may exceed the threshold, hence we need to store the insignificant term-weight pair with the significant term

SPI Method, cont.

- Given a profile vector P, a sub-vector P_s is *insignificant* at a threshold θ, if for any document D, $sim(D, P) \leq \theta$
- Which sub-vector should be used in case there are many insignificant sub-vectors?

  $\Rightarrow$ Use the one that contains the most *low-idf* terms

SPI Method, cont.

- Given a profile vector P, a sub-vector P_s is *most insignificant* at a threshold θ if it has the largest number of lowest *idf* terms among the insignificant sub-vectors at a threshold θ
- **Theorem** : For any P and any D, $||D||=1$, we have $sim(D, P) \leq ||P||$

SPI Method, cont.

- For each profile, the most insignificant sub-vector at a given threshold is found
  - The terms are sorted by *idf* and as many terms as possible are included
- The profile is then posted in the inverted list of the significant terms
  - In each posting, the insignificant terms and their weights are included, i. e. they are replicated in the lists of all significant terms

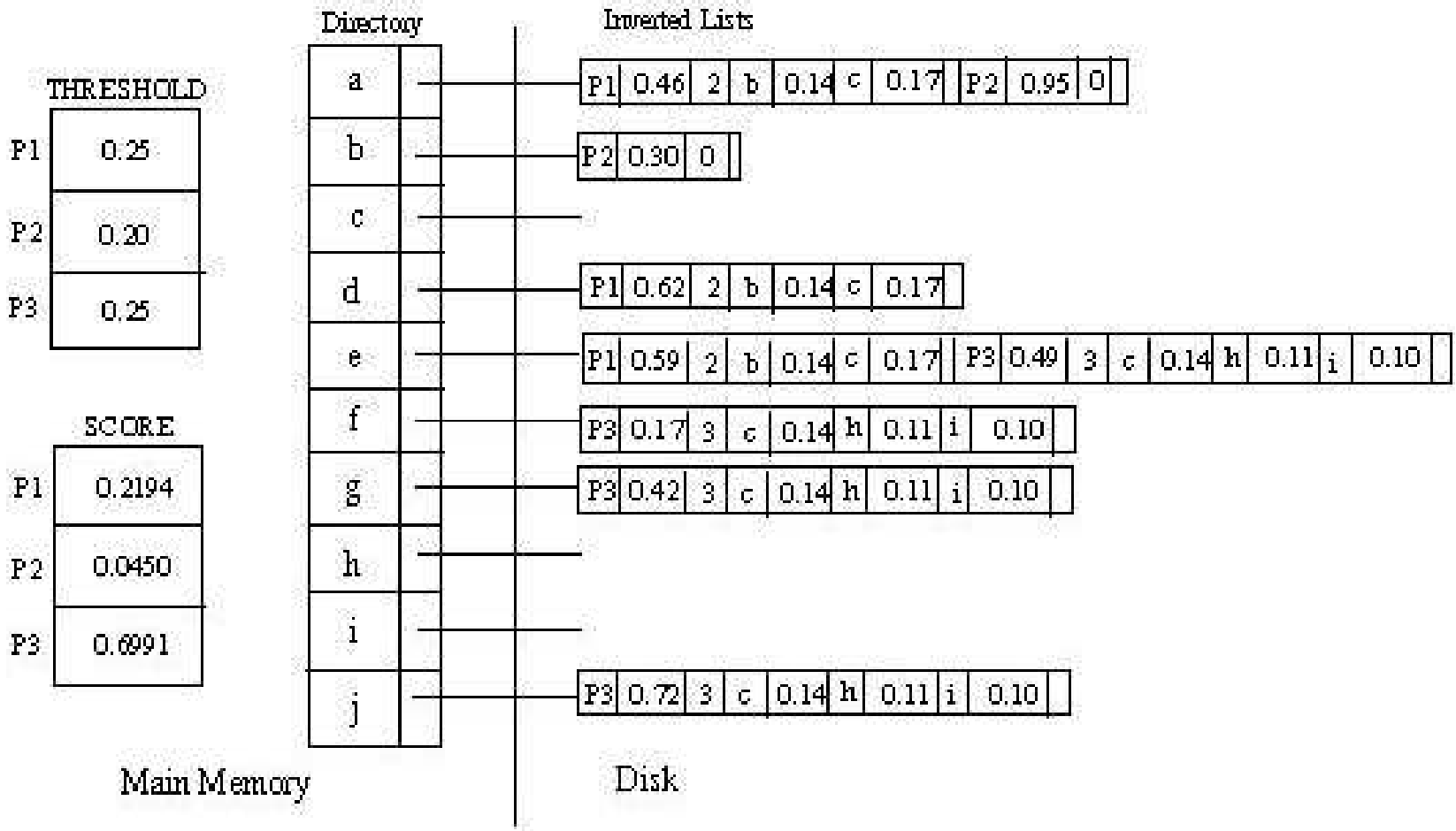
SPI Method, cont.

- When a document arrives, its vector representation is constructed
- Then, *SCORE* is initialized to all 0's
- Then, the inverted list of each term is retrieved by indexing the *directory*
- Suppose we are processing the term *x* with weight *w*
- We increment *SCORE[P]* by *w *u*

SPI Method, cont.

- Then, there are two possible cases
  - *SCORE* entry is zero: We increment *SCORE[P]* by all insignificant term weight * document term weights
  - *SCORE* entry is non-zero, meaning that the contribution of the insignificant terms have already been added
- A profile matches a document if *SCORE[P] > THRESHOLD[P]*

Yan & Garcia-Molina (ICDE'94)



Self-Adaptive User Profiles(CFG'00)

- Publish-subscribe models and other forms of push-based data delivery are becoming popular
- They require the ability to identify user interests, to target the right information to the right user
- The quality of user profiles is a key for push-based systems
- Single vectors are insufficient for adequately modeling user interests, whereas multiple independent vectors result in redundancy

Self-Adaptive User Profiles, cont.

- Uses a multi-modal representation of user profiles; a profile is represented as a collection of (*inter-related)* clusters of user interests
- The algorithm automatically and dynamically adjusts the number and the content of clusters
- Incremental algorithm: it receives user feedback one at a time and modifies the user profile accordingly
- Effectiveness, measured as *precision* and *recall*, is the primary metric

Self-Adaptive User Profiles, cont.

- The vector space model with *stop-list* removal and *stemming* is used to represent docs & profiles
- The cosine measure is used to calculate similarity
- The weight of term *t* in document *d* is given by

$$w_{t,d} = tf_{t,d} * log_2 (N / df_t)$$

$tf_{t,d}$: *the frequency of term t in document d*

df_t : *number of documents that contain term t*

N : total number of documents

Self-Adaptive User Profiles, cont.

- Rocchio relevance feedback is used to adjust profiles, that is

- A purely incremental feedback that updates a profile (query) for each individual document judgment is used

Overview of Multi-Modal (MM) Approach

- MM represents a user profile *P* as a set of *profile vectors* p_1, p_2,…,p_n, where each p_i is a list of (term, weight) pairs
- The number of profiles, *n*, and the size of a profile vector, *m*, change over time based on user feedback
- A single-pass, non-hierarchical clustering algorithm is used to construct user profiles

Overview, cont.

- Maintain clusters of document vectors, where each cluster is stored as a single *representative vector*
- The first document vector is assigned as the first cluster
- When a document is processed, its similarity with all existing clusters are calculated
- If its similarity to the closest cluster is less than a threshold (δ), then the document is used to initiate a new cluster

Overview, cont.

- If the similarity is greater than δ, then the document is incorporated into that cluster and the cluster representative is repositioned
- The influence of the new document is controlled by a parameter λ
- Two similar profile vectors may be merged into one to avoid redundancy
- A profile vector maybe deleted to adjust to changing user interests

Details of the MM Approach

- When a new relevance judgment , f_d, is received, MM first identifies the profile vector p_{act} that is most similar to v_d
- There is only one p_{act} that is active at any time
- If sim(p_{act} , v_d) < δ, then v_d is inserted into P as a new profile vector
- If sim(p_{act} , v_d) > δ, then v_d is incorporated into p_{act} as follows

$$p_{act} = (1-\lambda) * p_{act} + \lambda * f_d * v_d$$

Details, cont.

- If P is empty when feedback is received, MM checks the sign of f_d
  - If f_d is negative, it is simply ignored
  - Else, v_d is inserted into P as a new profile vector

  δ (*threshold parameter*) is between 0.0 and 1.0, and controls the number of profile vectors

  λ (*adaptability parameter*) is between 0.0 and 1.0, and controls the rate at which the active profile vector is moved

Adjusting the profile size

- The merge operation checks if a merge is possible between p_{act} and other profile vectors
- The profile vector closest to p_{act}, p_c is identified
- If sim(p_{act}, p_c) > δ, then p_{act} and p_c are merged
- p_c is pushed towards or pulled away from p_{act}
  - strength(p_c)/ *strength*(p_c)+*strength*(p_{act}) is used instead of λ
- A single merge at a time, no cascaded merges

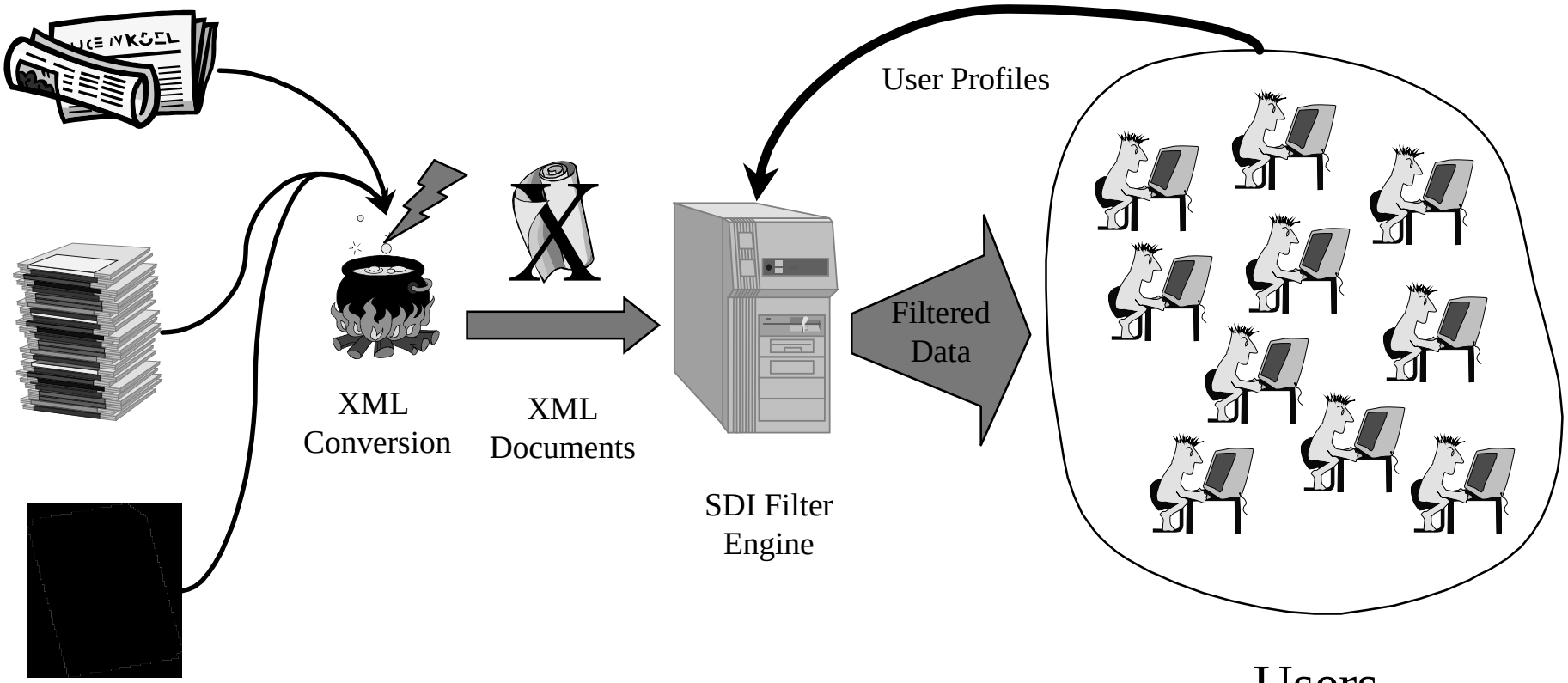
Deleting profile vectors

- Each profile is given a *strength* value (1.0) when the vector is created
- This value is updated each time a document is incorporated into the profile vector
- Strength modifications are performed using a simple exponential decay function where a positive constant, *c*, controls decay rate
- If the strength of a profile vector drops below a certain *deletion threshold*, then the vector is removed from the profile

XFilter (Altinel & Franklin 00)

- Simple text-based information filtering systems suffer from limited *expressiveness* of user interests
- *Efficiency* is a major concern for Internet-scale SDI systems
- XML has emerged as a standard information exchange mechanism over the Internet
- XML provides a base for more expressive profile models that take into account the schema information that are represented in the tags

Altinel & Franklin (VLDB'00)



XFilter, cont.

XML Document

```
<?xml version="1.0"?>
<catalog>
  <product id="Kd-245">
    <name> "Color Monitor" </name>
    <price currency = "USD">
      <msrp> 310.40 </msrp>
     <wholesale>257.8</wholesale>
    </price>
    <notes> Hottest Product> </notes>
  </product>
</catalog>
```

Corresponding DTD

```
<!ELEMENT catalog (product+)>
<!ELEMENT product (name, price, notes*)>
<!ELEMENT product id ID#REQUIRED>
<!ELEMENT price (msrp, wholesale?)>
<!ATTLIST price currency #REQUIRED>
<!ELEMENT msrp (#PCDATA)>
<!ELEMENT wholesale (#PCDATA)>
<!ELEMENT notes (#PCDATA)>
```

XFilter, cont.

- User profiles are represented as queries using the XPath language
- XPath is a standard language for specifying path expression over XML data
- XPath treats an XML document as a tree of nodes
- A query path expression consists of a sequence of one or more *location steps*
- Parent/child (/), ancestor/descendant (//), filters

XFilter, cont.

- Major components of the system are: ***1-)*** *an event-based XML parser for documents,* ***2-)*** *an XPath parser for user profiles,* ***3-)*** *the filter engine and* ***4-)*** *the dissemination component*
- The process of checking profiles is driven by an event-based XML parser
- The filter engine maintains an inverted index, called the *Query Index*, which is used to match documents to individual XPath queries

XFilter, cont.

- XFilter converts each XPath query into a *Finite State Machine*
- The events that drive the state transitions of this FSM is generated by the XML parser
- A profile matches a document when the final state of its FSM is reached
- The *Query Index* is built over the states of the XPath queries

Query Index

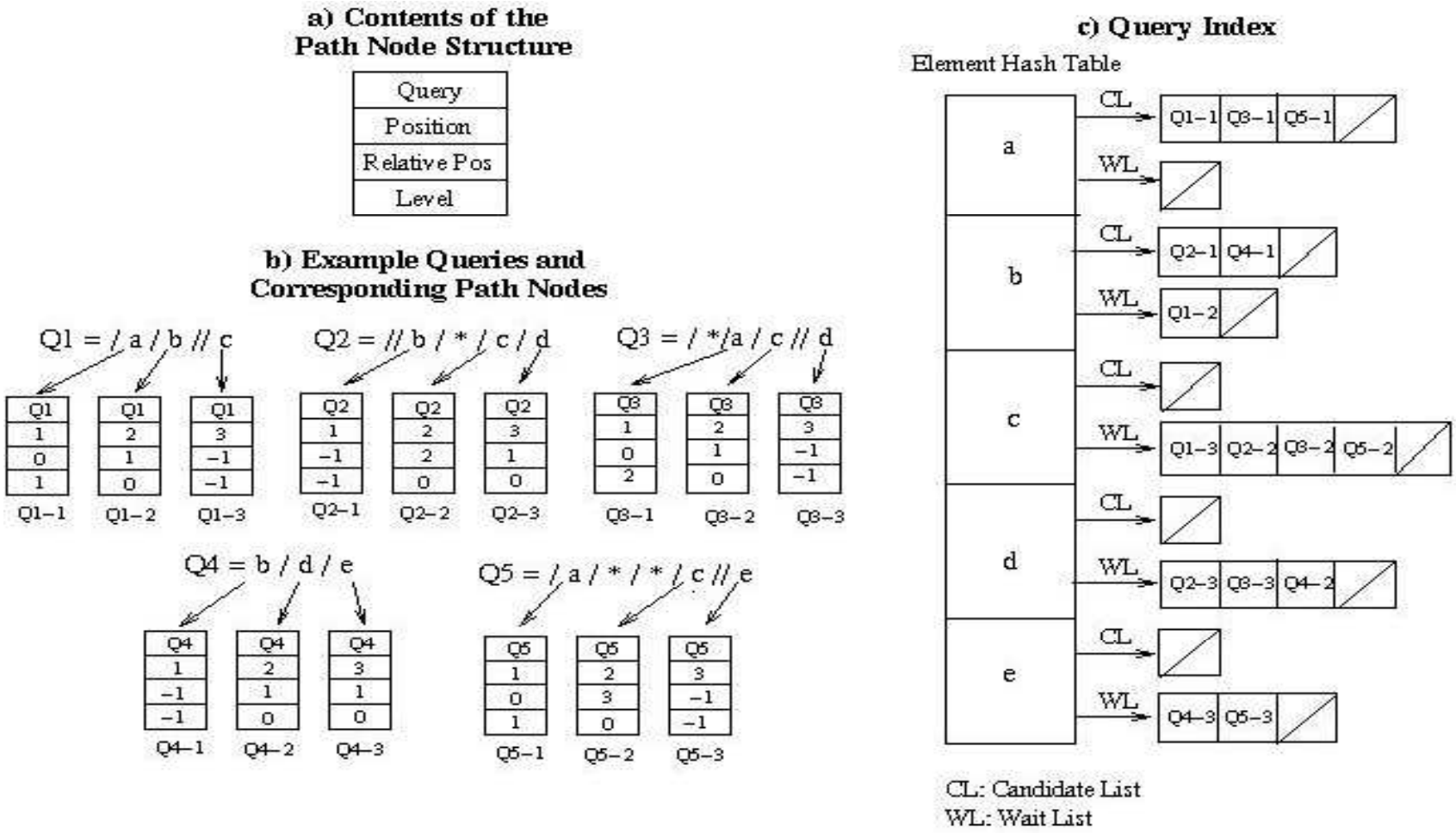
- Each XPath query is decomposed into a set of *path nodes*
- A path node contains the following information
  - **QueryId**: A unique identifier for the path expression
  - **Position**: A sequence number that determines the location of the path node in the order of the path nodes for the query
  - **RelativePos**: An integer that describes the distance in the document levels between this path node and the previous path node

Query Index, cont.

- **Level** : An integer that represents the level in the XML document at which this path node should be checked
- **Filters**: Stored as expression trees pointed to by the path node
- **NextPathNodeSet:** Pointer(s) to the next path node(s) of the query to be evaluated

- The *Query Index* is organized as a hash table based on the element names that appear in the XPath expressions
- Each entry in the *Query Index* has two lists of path nodes: the *Candidate List* and *Wait List*

Altinel & Franklin (VLDB'00)



Query Index, cont.

- The current node of each query is placed on the *Candidate List*
- All the path nodes representing future states are stored in the *Wait Lists*
- A state transition is recognized by promoting a path node from the *Wait List* to the *Candidate List*
- The initial distribution of the path nodes to these lists have an important effect on the performance of the system

XML Parsing and Filtering

- The event-based API reports parsing events directly to the application through callbacks
- XFilter has three callback functions
  - *Start Element Handler*: Passes in the name and the level of the element, as well as attributes. Performs a level and an attribute filter check. Can cause a state transition.
  - *End Element Handler:* The corresponding path node is deleted from the *Candidate List*.
  - *Element Characters Handler:* The data associated with the element is passed in. Performs a content filter check and can also cause a state transition

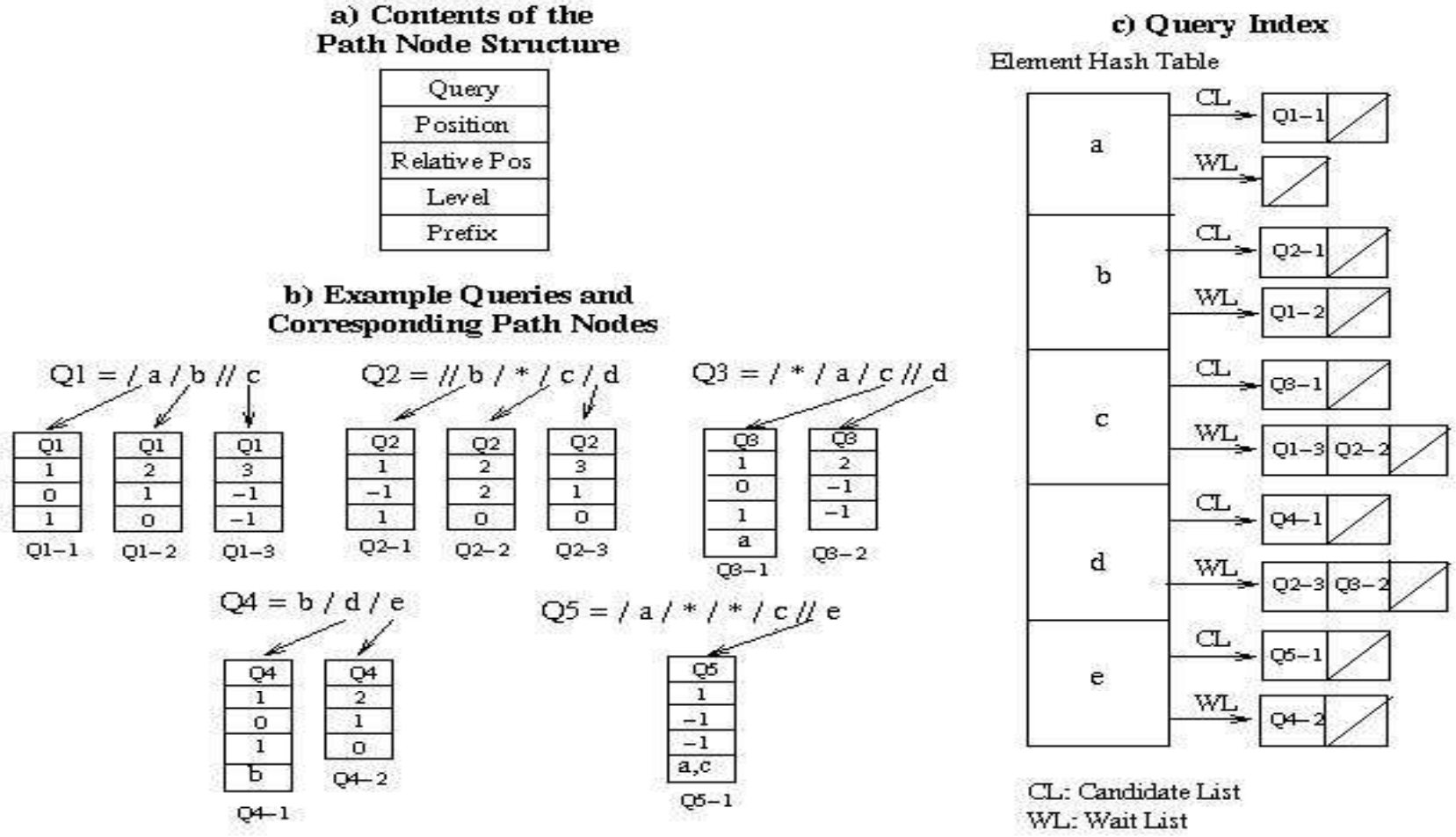
Algorithms

- *Basic*:
  - Place the first path node of each XPath query in the *Candidate List*
  - May produce highly skewed *Candidate List* lengths
  - First elements in the queries are likely to have poorer selectivity
- *List Balance*:
  - Attempts to balance the number of path nodes that are initially placed on each *Candidate List*

Algorithms, cont.

- When a query is to be added to the index, the element node whose entry in the index has the shortest *Candidate List* is chosen as the “pivot” node
- The “pivot” node is inserted into the *Candidate List*
- The portion of the FSM that precedes the “pivot” node is represented as a *prefix* that is attached to the node
- When the “pivot” node is activated, the prefix of the query is evaluated as a precondition
- The tradeoff is the additional work of checking prefixes

Algorithms, cont.

- *Prefiltering*:
  - The idea is to eliminate from consideration, any query that contains an element name that is not present in the input document
  - Each query is assigned a “key”
  - An occurrence table is built during a first parse of the input XML document
  - The occurrence table contains an element name and the list of queries whose keys are the same
  - If all the element names are found in the occurrence table, the query passes to the second step

Altinel & Franklin (VLDB'00)