A Markov Decision Processing Solution to Natural Language Querying of Online e-Commerce Catalogs: The EQUIsearch Agent

Barry G. Silverman[1], Mintu Bachann[2], Khaled Al-Akharas[2]
1- Dept. of Systems Engineering, University of Pennsylvania, 2- Equalfooting.com
contact: barryg@seas.upenn.edu
February 2001

ABSTRACT

A long-standing problem facing the field of Natural Language Query (NLQ) is that NLQ has been largely an academically appealing concept that isn't able to scale up to the complexities and performance demands of large-scale online catalogs (e.g., e-commerce shopping sites). Instead, the major relational database management systems and their applications rely on Conceptual Query (CQ) at best, and more often they use simple conjunctive keyword search for relational shopping catalogs. This paper expresses the NLQ of relational, online catalogs as an MDP problem that can be solved via policy-iterative, dynamic programming methods. Our research further delineates how to reduce the computational complexity of the MDP problem so as to obtain solutions in near real-time. Finally, this article concludes with a demonstration of the technique as a complement to CQ in a real world, large-scale relational catalog and with presentation of statistical tests that verify that NLQ scales up with reasonable response times and with better precision and quality than can be obtained with CQ approaches alone. The result is that NLQ complements CQ and improves search of online shopping sites.

1) Introduction

This article describes a bot or agent that uses restricted natural language to help users search product catalogs in large scale market exchanges. Market exchanges are sites that offer a virtual integration across multiple vendors' catalogs so that users can browse, search, bid, finance, buy, and/or ship products. Market exchanges are a major form of e-commerce in both the business to consumer and business to business segments: e.g., see Amazon.com, Priceline.com, or Ariba.com among many others.

Despite some recent setbacks, the rapid rise of e-commerce is impressive – over half of today's 80 million web users shop for or buy products online, and business to business purchasing is expected to rapidly eclipse that level [1]. However, in the rush to provide online presence, many e-market sites have been built quickly, with little infrastructure and capabilities needed to run such an e-business. As we all know only too well, browsing, searching, and buying via online web catalogs can be a time consuming, frustrating task. For example, [1] reports that over 80% of web shoppers have at some point left e-markets without finding what they want and that 23% of all attempted e-shopping transactions end in failure. Four of the top five failure modes are search-related (i.e., page loading times, couldn't find product, system crashed, had to call customer service) although some of the blame needs to be shared by other causes as well, a few of which are internet delays, overall website design, and reliability.

In general, several challenges confront one attempting to support search across product catalogs. First, is the question of the incompleteness and inconsistency of the product descriptions across sellers. Some descriptions include partial parameter information (e.g., 2”, black) while others focus on how the product might be used, and still others are highly terse and omit most details. Second is the difficulty of matching the buyer’s search terms to the wording in the descriptive fields. Where exact term matches don’t exist, one must consider issues related to word stemming, spelling errors, abbreviations, synonyms, and related problems. A third challenge is that numeric attributes are poorly and incompletely represented in these descriptions, yet some users will want to search by size, weight, height, voltage, and many other quantitative parameters. Unfortunately, the item description fields do not permit structured search of attribute information. For example, if searching the item description field for 2” brushes, a keyword search engine doesn’t know this is a width attribute and it would return hits on any occurrence of 2 and brushes. Thus the engine will return 2 ½ inch brushes, brushes made at 2 Downing Lane, and so on. Finally, there is the challenge of trying to figure out the user’s intent and underlying search goal.

Earlier we mentioned that most users encounter search failure and frustration due to sites having poor search and being unable to properly support them. Research shows that about 15% of online search failures are due to spelling errors and another 40% are due to customers using different terms from those in the website (e.g., patching vs. concrete): e.g., see [1-4]. Because they can’t interpret the meaning of the users’ query, search engines typically are tuned to bring back innumerable hits of everything even remotely relevant, often burying the best choice deep within the list (or omitting it altogether) and providing little help for then searching just the returned set of items. In short, when users get in trouble, most shopping sites provide little in the way of “trouble management”.

In this article, we describe progress to date with a search agent that uses a customized form of limited natural language to try and determine “what the user means”, not just “what they said.” This is also variously called either “conceptual search” or “natural language querying (NLQ)” of databases and it is intended to provide “trouble management” capability for market exchange (or single company) sites that don’t currently have it. Actually, [26] points out some interesting distinctions between conceptual query (CQ) vs. natural language query, where the former tend to deal with term semantics and ambiguities, while the latter attempt to parse the entire query string and may even engage in interactive conversation about the query to confirm its interpretation. According to [26], CQ languages [e.g., 27-29] can be improved by adding a parser with a grammar restricted to the domain of the database. Such parsers have less-than-general interpretation power, but still can offer much needed trouble management improvements and in helping the interface conform to the user’s language. Several systems purport to provide these extensions such as [26, 30-32], and some of them offer formal slotted grammars (with semantically typed slots) for merging the natural language approach atop conceptual querying.

Proponents of pure CQ in turn argue that the natural language extensions tend to be impractical, and indeed none of the NLQ systems just cited have been evaluated on a large scale. They argue that the “natural language problem” is too difficult and remains unsolved on any reasonable scale. As a result, most large scale relational database management products (eg, Oracle, Sybase, DB II, or Alta Vista Search Engine) that are widely used by market exchanges and other large scale e-commerce sites, include CQ features for those who choose to deploy them, but exclude NLQ. There are some smaller (less formally defined) NLQ systems that appear to work atop specific databases or environments. One such example is Microsoft English Query [5] which applies natural language searching ideas to catalogs built atop their own proprietary software products (i.e., SQL Server and ASP or COM), or EasyAsk which likewise only works on Microsoft NT machines [6]. Also, there is a push to add natural language self-help or chatterbots to many e-commerce websites, but these bots handle site navigation and document retrieval issues, and they are entirely incapable of processing catalog or database search requests, a fact that adds fuel to the arguments that only conceptual query is scalable: e.g., see [7-10]. In sum, there are no examples of NLQ working in large scale e-commerce catalog shopping sites, and one is tempted to believe the proponents of CQ rather than NLQ.

Specifically, there are three hypotheses that are worth examining and that we will attempt to explore in this paper:

- **H1**: While it may be work as a novelty item in specialty applications, NLQ is not scalable to the complexity of databases found in large scale market exchanges.
- **H2**: Even if NLQ could be scaled up to work in market exchanges, its query parsing and processing time would be unacceptable compared to the best CQ approaches.
- **H3**: Since natural language is so ambiguous and difficult to parse, the effectiveness or quality of the results from NLQ will never equal that of CQ.

Creating an NLQ agent that is scaleable, fast, and effective would be a useful complement to many of the catalog CQ products that already exist. This article explains our research toward this end, and concludes with empirical tests of these hypotheses to allow the reader to determine what progress has been made to date. Briefly, our approach is similar to [26] in that we both use parsers, information extraction approaches, and semantically-typed grammar slots that are domain specific. But that's where the similarities end, and unlike systems such as [26], our approach uses Markovian decision processing rather than formal logic. Also, we have done away with some of the less practical NLQ proposals in the literature, such as conversational feedback and explanation of the translation of the query to be user prior to executing the query. Instead we converse with the user in the fashion that CQ systems use -- after the query via the display either of hits or of pre-canned, limited explanations of query failures. In the end, we will test the three hypotheses via a relatively large scale implementation, and we'll examine results from several timing and effectiveness evaluation tests as well.

1.1) Definition of an Online Shopping Catalog

Before going further, it is worth elaborating on the definition of an online catalog and some of the search challenges it represents. An e-commerce catalog is the heart of a shopping site and it holds information on all the products one can buy at that site. The catalog is browsable like other website contents, but unlike the other contents, it usually is stored in a relational database product as are the transactions such as bid, buy, ship, etc. On the face of it, the reason a relational database is used is that the catalog is far more structured than the HTML documents typically found on a website. However, the degree of structure is relative, and most product catalogs do not have nearly enough structure for search to work at its optimum.

We start by describing the general structure of product catalogs and the search challenges they pose, and then proceed to a discussion of the actual catalog instantiated for our testing. The basic logical structure of a product or item catalog may be described as four sets of Relations, **R = {R1, R2, R3, R4},** three of which we discuss here. The fourth concerns the fields that are used in processing transactions:

Product Hierarchies (R1) – These are the fields supplied by the website when users try to browse the catalog. As a result, **R2 =** $<\mathbf{C_1, \ldots, C_N}>$ is a tuple of multi-level trees that taxonomize the contents of the catalog and whose leaf nodes point to actual products in the catalog. When merging each of the $\mathbf{C_N}$ catalogs of multiple vendors on a dynamic, continuing basis, market makers encounter almost overwhelming taxonomic challenges to constructing R1, which in turn pose browse and search design concerns.

Product Descriptions (R2) – This relation is a tuple, **R2 =** $<\mathbf{D_1, \ldots, D_K}>$, of k=1,K free text descriptions that suppliers create and which hold the information about products that suppliers believe users need. The D_K are the document fragments utilized by search engines to support user queries and product search commands. Interestingly, while catalogs include a field called "product name", most suppliers omit this field believing the "description" field is sufficient. When merging 100,000s and 1,000,000s of products, market-makers have no resources to supply this or other missing information, or to standardize free text descriptions the suppliers provide in the D_K. This makes for some formidable search engine design challenges – ones the field has yet to entirely overcome.

Product Attributes and Values (R3) – The catalogs also hold numerous parameters about each product (e.g., color, weight, length, manufacturer, price, condition (new, used), availability, reviews, etc.). This is the information that is used to support parameter search and sorts such as by price or size or location. A universal feature of catalogs is that there are many products, the product lines are continually changing, and each of the 1,000s of leaf node categories of products has a different set of attributes. As a result, attributes are not stored as fields of a table. Rather, the m=1,M categories of attribute names are stored as data items as are their value settings. Thus **R3 =** $<<\mathbf{A_{31}, V_{31}}>, \ldots, <\mathbf{A_{3M}, V_{3M}}>>$ where A and V are vectors of a-v pairs that are equi-length for any given category**.** This and large numbers of unfilled in attribute fields are stumbling blocks few DSS designer have yet to fully eliminate in trying to support attribute or parameter search.

Shopping catalogs often include many dozens of tables (some have 1,000s) in order to support all the services and user support. In order to improve runtime performance, a denormalized field called a "munge" is often used as the target for the search engine. This munge places into its sub-fields copies of each of the searchable fields of the catalog such as item name, item ID, category name and ID, model or part number, description, price, maker, condition, and all attribute-value pairs. In effect the munge is like a document on each product in the catalog. Since most search algorithms can't infer the sub-fields to search, they will search the entire munge. For example, keyword search will search across all sub-fields of the full munge for a strict match on terms, while CQ will traverse the same ground but with the ability to look for synonyms, alphabetically similar terms, and related conceptualizations. NLQ, in turn, is the only search strategy that infers the labels or field names of each token in the query string and, hence, can then send the (conceptual) search to the precise "subfields" of the catalog in which the query's tokens should exist provided they are in the database. In general, however, the challenges mentioned at the outset of this section cast into doubt the ability of any of these search strategies to be entirely effective in product catalogs.

1.2) Measuring Query Effectiveness (recall, precision)

When one talks about improving online search, it is important to have a way of measuring progress and of calibrating improvements. Ideally, any measure should be both scientifically grounded and reflect improvement from the user's perspective. Fortuanately, there are several decades of information retrieval "effectiveness" research to turn to for suitable metrics, though that literature is by no means settled on what is the best metric [35-39]. There, effectiveness is defined as a measure of the ability of the system to satisfy the user in terms of the relevance or pertinence of items retrieved. Pertinence is assumed to mean 'aboutness' and 'appropriateness', that is, a document is ultimately determined to be pertinent or not by the user (or us). It is helpful at this point to introduce the ubiquitous 'contingency table' shown in Table 1.

Table 1 – "Contingency Table" For Deriving Effectiveness Metrics

	Pertinent (**P** = RP + NP)	Irrelevant (**I** = RI + NI)
Retrieved (**R** = RP + RI)	**RP** Benefit (B1) "Hits"	**RI** Cost (C1) Type I errors
Not-Retrieved (**N** = NP + NI)	**NP** Cost (C2) Type II errors	**NI** Benefit (B2)

Total Catalog = RP + RI + NP + NI

A large number of measures of effectiveness can be and are derived from this table. To list but two of the most common ones:

- Precision is a measure of how well a system finds ONLY pertinent documents on a searched for query. Thus precision is the ratio HITS:(True HITS + False Positives)

PRECISION = RP/(RP+RI)
= Pr(P|R) = conditional probability of being pertinent given its retrieved

- Recall is a measure of how well an information search and retrieval system finds ALL pertinent documents on a searched for topic, even to the extent that it includes some irrelevant documents. Thus, Recall is the ratio HITS:(True HITS + False Negatives)

RECALL = RP/(RP+NP)
= Pr(R|P) = conditional probability of being retrieved given its pertinent

Most of the information retrieval literature defines precision and recall as the simple ratios defined here. Swets in [37] objected to these ratios as failing to be grounded in statistical decision theory. He redefined them as approximations to conditional probabilities, which is the second line of each equation above. This is a useful step, and following this idea, we have introduced the notion of Type I (false positive) and Type II (false negative) errors into Table 1. Swets also introduced the idea of benefits and costs and we reproduce that in Table 1, however, neither Swets nor the ensuing literature have generated a suitable way of measuring these monetarily. So, most evaluations stick with the recall and precision metrics.

The Cranfields Tests [35,36] in the 1960's claimed to find an inverse relationship between recall and precision. As recall rate went up, precision rate fell; as precision rate rose, recall rate went down. However, these co-variances are only loosely connected and are largely a pragmatic effect. In theory, it is not necessary and one could devise a search algorithm that maximizes hits while minimizing both Type I and II errors. As a result, a number of researchers have attempted metrics that are composites and serve as a single measure of effectiveness: e.g., see [38, 39] for a survey. The simplest of these is just [SUM = RECALL + PRECISION], although this has no theoretical basis to it. Several others do exist that have theoretical rigor, however, those either appear computationally over-complex, or they tend to ignore Recall, or at least they omit consideration of the False Negative rate: e.g., see [38, 39].

We are interested in a simple composite with some degree of rigor to it. Another field that has some metrics we find suitable is that of statistical quality control. There they measure defect rates and both Type I and II errors. The metric that seems most suited to us is one where utility of the search approaches unity as the error rate falls to nil:

$$\begin{aligned}\text{QUALITY} &= 1.0 - \{\text{ERROR RATE}\}\\ &= 1.0 - \left\{\frac{\text{FALSE POSITIVES + FALSE NEGATIVES}}{\text{HITS + FALSE POSITIVES + FALSE NEGATIVES}\}}\right\}\\ &= 1.0 - \{(\text{RI} + \text{NP})/\,(\text{RP} + \text{RI} + \text{NP})\}\end{aligned}$$

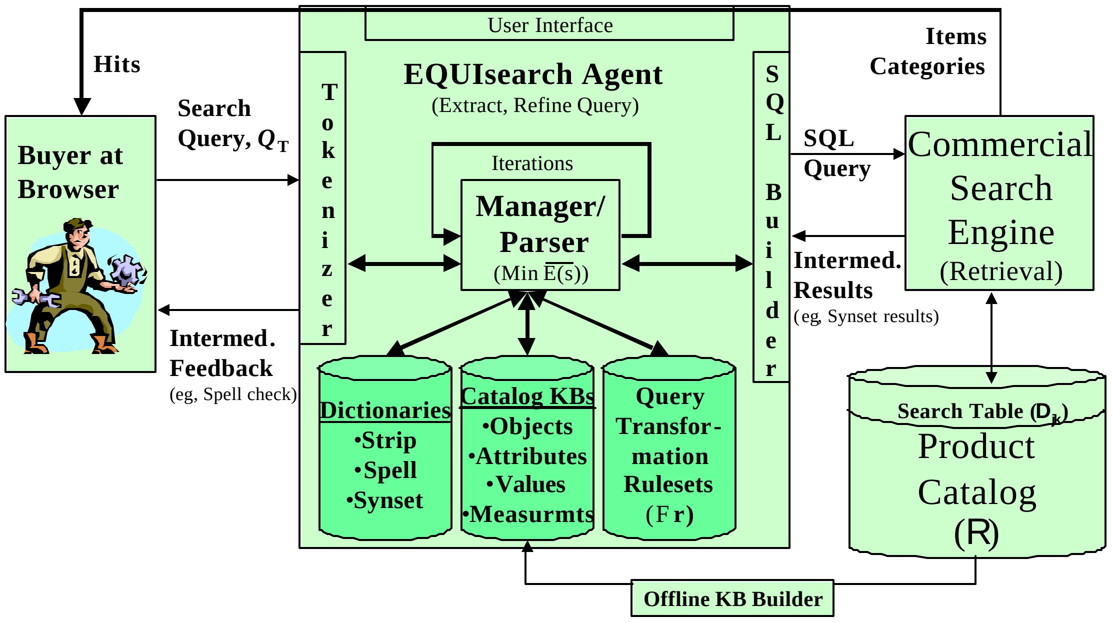
We purposely label this as quality, since we also plan to use it as the surrogate of the net-benefits (benefits – costs) to the user of the search engine. Finally, it will serve a third purpose, that of being the reward value we seek to maximize within the Markov Decision Process, a topic taken up in the next section.

2) <u>System Architecture and Algorithm</u>

This section begins with an overview of how the agent is configured and proceeds to elaborate the algorithms used to try and improve search quality. Specifically, the search agent in this article tokenizes an initial search string or phrase typed by the user at their browser (left side of Figure 1), and then acts as an intelligent interpreter for traditional search engines, such as Intermedia from Oracle or the Alta Vista Search Engine (AVSE) on the right of the Figure. As a language or meaning “broker,” the agent's intelligent interpretation efforts iteratively label each token of the query and subsequently modify these labels by applying a sequence of transformation rules which attempt to reduce the remaining ambiguities and residual errors left in place by the previous rules. In this regard, the Search Agent is a markovian decision process or a finite state automaton, technology that is commonly used in message understanding and in language understanding, but which is relatively untested for the searching of electronic commerce catalogs over the web.

For this to work, first, one must accept that from the agent's perspective, each search string or query produced by a different user is a partially observable instantiation of the current state of the world. The agent's job, then, is to discover the meaning of each query, by parsing it within a limited sub-grammar. This morphosyntactic parsing is accomplished by attempting to label the state of each term in the query, where state is defined within the sub-grammar. Second, for this to work, one must have the sub-grammar. The sub-grammar we created here is called "OAV Triplet Grammar." OAV triplets, or object-attribute-value triplets, captures the underlying meaning of searches in product catalogs even though product catalogs may contain many tables of features and parameters of a given product. Specifically, in product catalog domains, searches are usually for objects with attributes of a certain value. This is something of a noun phrase that includes adjectives. For example, some triplets in various orderings might be: 'mini sized amps' or 'hammer colored red'. The order of the triplets is a task for the agent to discover. Thus a common order variant occurs where the O is sought initially and then the AVs are subsequently used for comparison and search refinement (bolt cutter followed by size, price, and availability). Even more common are searches for one or more V of a given type of O where the A is suppressed (eg, AA Eveready batteries, ½" no.8 slotted screws, or desk chair).

Figure 1 – Overview of The EQUIsearch Agent as a Meaning Translator In the Interface Between Users and Traditional Search Engines



In addition to the rules of the sub-grammar, one obviously must also derive the vocabulary for any given instantiation or catalog. In shopping domains, the vocabularies can be derived for objects and another for attribute-value pairs by crawling the fields of the relational catalog database and by extracting all unique terms (item names are object name, attributes are parameter names, and parameter settings are attribute values). We show this tool for crawling the catalog and extracting the KB values at the base of Figure 1. Using such a tool, one can readily construct the sub-grammar and its vocabulary for any given electronic catalog. The result is stored in KB lookup tables (base of agent in Figure 1) so it can be utilized by transformation rules of the phrase parser to infer and insert O, A, and V labels onto the various terms of any given search phrase. Also at the bottom right of the agent box are shown other rules (Φ_r) that are used to infer OAV meanings from queries.

It is not sufficient that the vocabulary extracted from the catalog provides 100% coverage of the lexicon of a given shopping site if the users of the site are unaware of that lexicon or prone to misusing it. To help overcome such semantic impasses, the tokens in the query string are first stemmed, subsequently expanded for synonyms, and ultimately checked for spelling (bottom left of the agent in Figure 1). This is supported by adding the dictionaries and KBs across the base of Figure 1. Each of these items across the base of the agent requires major investment by a given e-commerce site as the lexicons are often unique and poorly covered in off-the-shelf dictionaries. These items must be developed and adapted to each new catalog application. So a generic contribution is not only their contents, but also the infrastructure to acquire and maintain them in each new application. To the extent these steps can be automated, that is a direct payoff in manpower reduction and time to market with better search capability.

2.1) How the EQUIsearch Agent Manager Works

We make the assumption that a Markov decision process is suitable to analyze the semantics and morphosyntactics of the user's query. Specifically, Markov decision processes (MDPs) model decision theoretic planning problems in which an agent must make a sequence of decisions to maximize its expected utility given uncertainty in the effects of its actions and its current state. At any moment in time, the agent is in one of a finite number of states (s=1,S) and must choose one of a finite set of actions (a=1,A) to transition to the next state. More specifically, optimizing a Markov decision process was defined in [11-14] as a dynamic programming problem that maximizes expected, discounted rewards across future periods as follows:

$$\text{Max } V^* = E\left[\sum_{t=1}^{T} \delta^t U(s_t, a_t) \right] \qquad (1)$$

where,

V^* = optimum value point (in terms of "quality" as defined earlier)

E[ ] = expected value of the discounted future reward over iterations t=1,T

δ^t = discount factor ($0<\delta^t<1$, but in short horizon problems let $\delta^t=1$)

U( ) = reward function or utility (QUALITY) from selecting action a_t at state s_t

Often this formulation is expanded to a “value iteration” formulation where one loops across iterations (t=1,T) and for each iteration, one then loops across all states (s=1,S) to find the action or set of actions that maximizes both current and future rewards so as to avoid local optima: e.g., see Howard(1960), Monahan(1982), or White(1987). This expansion is captured by finding the maximal value of the following function after testing all possible actions, a=1,A:

$$Z_t(s,a) = U(s_t, a_t) + \delta \sum_{s_{t+1}=1}^{S} \pi(s_t, a_t, s_{t+1})\, V_{t-1}^{*}(s_{t+1}) \qquad (2)$$

where,

$\pi(s_t, a_t, s_{t+1})$ = the transition probability of being in state s_{t+1} immediately after taking action a_t from state s_t

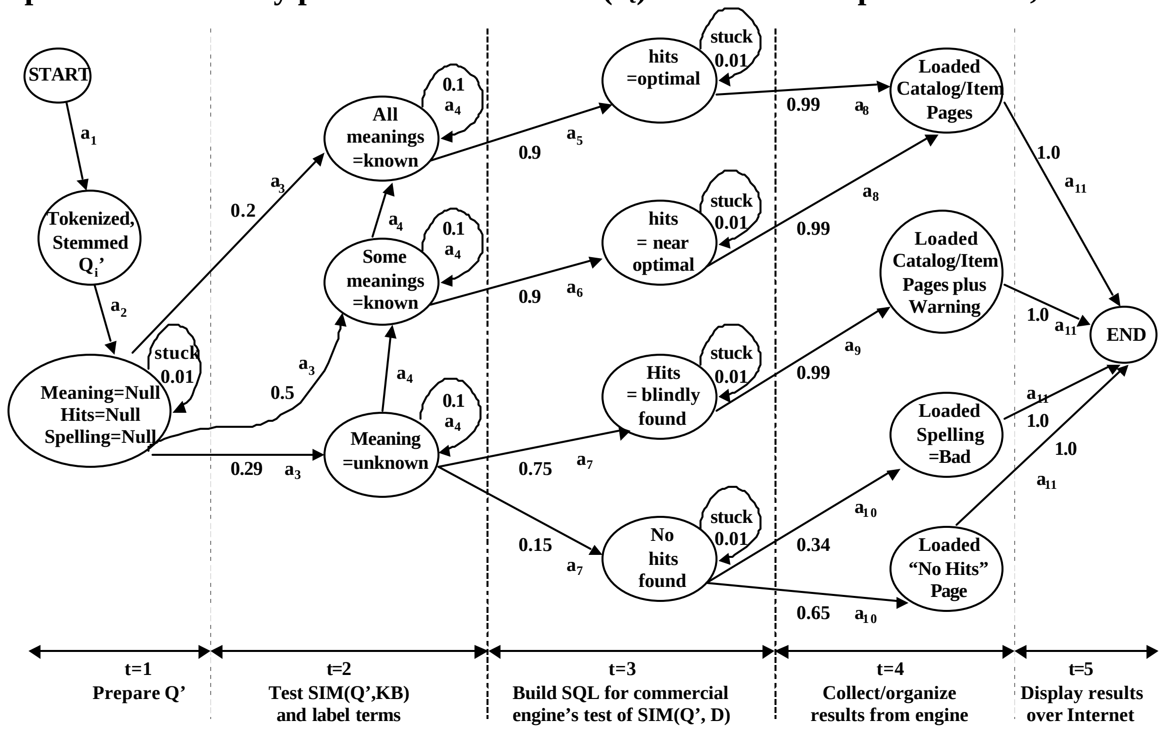
$V_{t-1}^{*}(s_{t+1}) = Z_{t-1}(s_{t+1}, \mathrm{argmax}_a(Z_{t-1}(s,a)))$ where argmax_a finds the maximal quality action

Thus, recursive equation (2) summarizes the standard computable “value iteration” formulation of the dynamic programming optimization of a Markovian decision process. It is necessary to adapt this into a formulation that can be solved in catalog search domains. To do so, we first must define the permissible states and actions, the reward function, the transition probabilities, and the other terms of the equation within the context of catalog search problems.

Figure 2 displays one of many possible orderings of the permissible states, actions, and transition probabilities. Actually, these are illustrative and somewhat of a simplification, since there is more looping possible between iteration levels (t) on the diagram than is actually plotted. Also, the transition probabilities displayed here are approximations for a particular market exchange. They must be computed anew for each application based on frequencies in transaction log files from that application’s catalog.

According to the Markov Decision Process theory, we must compute the expected utility function or a set of decision triggers at each state (the policy table) that drives the agent to seek the optimal path through the state transition network. That is, we want the agent to chose argmax_a for each state such that it tries to optimize utility. In theory, the agent could follow this utility maximizing scheme to achieve the desired argmax result. That is the “policy iteration” approach within dynamic programming. However, the computational complexity of dynamic programming updates of partially observable MDPs is at least P-complete which is intractable for large problems unless a number of restrictions are imposed. In practice we have determined the “steady state” policy set for the agent: i.e., a “policy matrix” that the agent uses to select specific actions in a given order when the query is in a given state as portrayed in Figure 2. This approach allows the MDP to be independent of initial state of the query and it is widely used to reduce the computational complexity of Markov Decision Processes: e.g., see [12-16]. Another simplification we make is that since reward assignments are implicit within policy iteration, we do not bother to have the agent compute them for each query, a change that has no impact on finding the optimal argmax resultant. Finally, the number of iterations are bounded by the number of rules that are authored in Φ_r, a number that is often less than a few dozen, and many of which are lumped for runtime purposes into a few iterations at most.

Figure 2 - **Permissible States and Actions of the Agent when viewed as a Markov Decision Process (S=15, A=11). NOTE: This view displays the computationally optimized stationary policies for all actions (a_t) and transition probabilities, π.**



ACTIONS LEGEND (a_t):
a1 = tokenize, strip, and stem a new user query, Qi, from the server
a2 = pop a token from the Qi' string
a2'= return for another token if Qi' not empty
a3 = test the token against the OAV triplet vocabulary stored in the KBs
a4 = test unknown-meaning tokens against rules in F_r
a5 = build SQL query (conjunctive keyword search of munge on stemmed tokens expanded by synonyms and with OAV field assignments for all tokens)
a6 = build SQL query (conjunctive keyword search of munge on stemmed tokens expanded by synonyms and with OAV field assignments as known)
a7 = build SQL query (conjunctive keyword search of munge on stemmed tokens expanded by synonyms. This is also known as V1.3 search)
a8 = collect query results in the form of returned category and item IDs & names, and load display pages
a9 = collect query results in the form of returned category and item IDs & names, and load display pages along with a warning that meaning was not determined, and blind keyword search was used
a10 = invoke spell checker, and if spelling is wrong then load spelling suggestions, else load no hits page
a11 = push display pages to web server for user viewing

Before turning to that evaluation topic, it is probably worth it to provide a little more detail about the agent's actions during each of the iterations.

2.2) Agent Actions in Iteration 1 (t=1)

Action 1 of Iteration 1 consists of the standard morphosyntactic extraction steps intended to transform the raw search string, Q, into a set of stripped and stemmed tokens that form the transformed query string, Q'. To facilitate the transformation process the agent first tokenizes the terms in the search phrase, Q. Next it eliminates stop or strip words (e.g., "find me a", "get all", and opening and closing quotes) by comparing the search string to a negative dictionary or a strip word list. Finally, the agent applies a stemming algorithm uses Porter [17] modified to also strip out all numerals. The result of these steps, as already mentioned, is the query string the agent labels as Q'. The stripped numerals are returned to the query during the SQL build step as will be explained below.

In order to perform subsequent actions on the tokens within the transformed query, the agent next defines a query string Q'(λ, h, Spelling) and its facets or tags as:

Q =	the user's original query string
Q' =	the query string after being transformed by the agent
I =	number of tokenized terms in the search string
λ =	the $\lambda=1,\Lambda$ meaning tags that a query or a term, i, might assume after transformation
h =	the counter of the H hit tags of a term (initially NULL, then transitioning to one of 4 possible labels shown in the t=3 column of Figure 2)
Spelling =	a flag that assumes the value of NULL initially, and then Bad or Good as a result of action a10 in t=3

When discussing the entire query, Q'(λ, h, Spelling), we say that $\Lambda=4$, and the five tags are NULL, UNKOWN, SOME KNOWN, ALL KNOWN. Here NULL is a assigned at the outset, whereas UNKNOWN is a specific label or tag that implies the string has failed all attempts to label it's tokens. The SOME or ALL KNOWN tags are string status indicators based on individually testing each token in the string. To assign meaning tags to individual tokens, we refer to the ith token and its facets as $Q'_i(\lambda$, h, Spelling). In this case, $\Lambda=5$, and the five permissible token level tags are NULL, UNKOWN, OBJECT, ATTRIBUTE, or VALUE.

2.3) Agent Actions in Iteration 2 (t=2)

The primary agent activity in this iteration is to test the individual tokens of the query string against the knowledge bases that hold the catalog's objects, attributes, and values expressed within the vocabulary of the sub-grammar.

The agent then proceeds to display all categories containing hits from keyword searching of the KB fields. This is defined as a match between the i= 1,I terms in the query string (Q'_i) and their counterparts somewhere in the j=1,J terms in the kth knowledge base (KB_{jk}) being searched. Here there are three KBs, one each for objects (k=1), attributes (k=2), and values (k=3). If the match is exact, the similarity (Sim) equals unity and the pointer to the KB or product is returned.

Let us state this as:

$$\text{Sim}(Q, KB_k) = \{ \sum_{i=1}^{I} \text{Sim}(Q_i , KB_{ji}) \}/ I \qquad (3)$$

Subject to,

$$\overline{\varepsilon(s)} = \{ \sum_{i=1}^{I} e_i(s)\}/ I$$

$e_i(s) = \{ 1- \mathrm{Sim}(Q_{is}', KB_{jk}') \}$

$Q_{is}' = \Phi_r(Q_i')$

Stopping rule: $\overline{\varepsilon(s)} < \beta_1$ or, $\mathrm{TIME} = \beta_2$

Where,

I = number of terms in the search string
s = current state counter for the query string
s-1 = prior state before latest rule transformations

$\mathrm{Sim}(Q_{is}', KB_{jk})$ = score of the i^{th} query token against the j=1,J terms of the kth Knowledge Base. Score is (0,1) depending on whether a match occurs. A perfect match occurs where $\mathrm{Sim}(Q_{is}', KB_{jk}) = 1$.
Φ_r = application of one of a number of possible miscellaneous rules that changes the meaning, λ, of the query string by tagging or labeling some terms within it (see action a4 below).
β = threshold for error allowed -- generally $e_t(s)$ is an on-off function or (0,1)
TIME = maximum search time before the web-logic engine will terminate the request

Action a4 – Run Miscellaneous Rules (Φ_r):

Testing the $\mathrm{Sim}(Q_i', KB_{jk})$ corresponds to action a3. However, one can add all kinds of domain-specific rules for any given catalog that will provide further transformations of the individual tokens within a query string in order to increase the likelihood that $\mathrm{Sim}(Q_i', KB_{jk})$ will be non-zero. In the hardware domain some of these pertain to recognizing and labeling the notation. For example, a number followed by ‘ is feet or “ is inches, while a number followed by lb is a weight. Likewise, the term “made by” is a two value predicate often preceded by an object and superseded by a manufacturer (so C-H Cutler Hammer or just Hammer can be discerned as the object or the maker). Or other rules might be added about two measures linked together, like ½ x 8 for screws. One of the advantages of the rule-based approach is that rules can be added on the fly, and as more get added, the better the search engine becomes. Thus one can peruse domain content sources and manually extract rules over time, or one can try to deploy machine learning approaches and add newly discovered patterns as convenient.

2.4) Agent Actions in Iteration 3 (t=3)

The primary activity in this iteration is for the agent to use the object, attribute, and value tags plus findings from rules in t=2 to construct an intelligent SQL search statement that the commercial engine can use to test the query string against the catalog itself. Any token that is tagged retains its tag and the commercial engine uses this to restrict the fields it searches. In many cases numerals have already been returned to the search string and labeled or tagged by rules. Where all tokens in the string are tagged, the agent calls this the optimal search, and performs action a8. Even in this case, the tokens are expanded via synonyms since the users' meaning might be broader than their term usage (e.g., a search for "safety glove" includes "protective mitten", "work glove", and so on).

Where the set of tokens and numerals in the query are only partially tagged, action a8 constructs a SQL statement that searches the specific fields for the tagged tokens, and searches the "munge" or entire search table for the remaining tokens. Again, all tokens are expanded via synonyms. For those unfamiliar with the jargon, "munge" is a construct provided by the commercial database systems. It is a join of all the database fields that one wishes to search into a single de-normalized table. Typically, one places quite a few fields into this munge such as product ID, name, description (short and long), and select attributes such as manufacturer name, price, and so on. When tokens are labeled, only the respective fields of the munge table (also called the search policy table) are queried. Otherwise, keyword matching is attempted for all expanded, untagged tokens across the full range of fields in the munge. If none of the tokens in the query string are tagged at all, the agent calls this blind search of the munge, and invokes a7 which attempts a conjunctive search on the tokens and their expansion sets.

Our concept of how the commercial engine uses the SQL query built and invoked by a5, a6, or a8 is as follows. We believe the commercial search engine sorts through and finds all categories containing hits from searching of the munge fields. This is defined as a match between the i= 1,I terms in the query string (Q'_i) and their counterparts somewhere in the m=1,M fields in the nth database record (D_{mn}) being searched. If the match is exact, the similarity (Sim) equals unity and the product ID and its browse tree category ID are returned. A perfect match occurs where Sim(Q', D) = 1. In general (non-catalog) web searching, most engines will return partial matches often by reducing the threshold to some reasonable number (Sim(Q, D)<1), but they will sort the documents in descending score so those closest to 1.0 will appear first. Also, when searching on the web, S(Q, D) is most often computed as a weighted dot product of the respective term vectors, Q_i and D_{mn}. In catalog search, something equivalent to this can be used depending on the preferences of the market exchange or site proprietor. In our experience a strict keyword match is often utilized (albeit on synonyms), where the boolean AND is assumed between all the terms of the vector Q'_i. Thus, a variant of equation (3) is often utilized in the form shown here:

$$\left.\begin{aligned}
&\mathrm{Sim}(Q, D_k) = \left\{ \sum_{i=1}^{I} \mathrm{Sim}(Q'_i, D_{mn}) \right\} / I \\
&\text{Subject to,} \\
&\overline{\varepsilon(s)} = \left\{ \sum_{i=1}^{I} e_i(s) \right\} / I \\
&e_i(s) = \{ \mathrm{Sim}(Q_{is-1}', D_{mn}') \} \\
&\text{Stopping rule: } \overline{\varepsilon(s)} < \beta_1 \text{ or, TIME } = \beta_2
\end{aligned}\right\} \quad (4)$$

Where,

I = number of terms in the search string

s = current state counter for the query string

s-1 = prior state before latest rule transformations
D_{mn} = the m=1,M field in the nth "record" being searched.
β = threshold for error allowed - generally $e_t(s)$ is an on-off function or (0,1)
TIME = maximum search time before web-logic engine terminates the request

2.5) Agent Actions in Iteration 4 and 5 (t=4,5)

At this juncture, the agent is seeking to load the appropriate display pages to send to the user. Under a8, the agent is fairly assured that a reasonably relevant response has been located (expectation is that "precision" is fairly high) and, under a11, the user is sent a summary of the number of hits sorted by browse tree category. The user can inspect these through subsequent browsing interactions.

By contrast, if a9 was just run, the agent realizes that a blind, conjunctive keyword search of the munge just transpired and that any hits may or may not contain the relevant products. Thus precision is likely to be low and recall fairly high (many hits returned). So the agent prepares a warning to send at the top of the page it displays to the user as follows: "We have been unable to locate your exact item (possibly due to mislabeling in our catalog). You might find your item in the following near hits:" The rest of the page then contains a listing that is identical in format to that of the high precision case – i.e., a summary of the number of hits sorted by browse tree category.

When the commercial engine's SQL search of the catalog is completed and there is no direct hit in the database, we assume that the stopping rule in (1) has failed such that

Stopping rule: $e_t >= \beta_1$ or, TIME $> \beta_2$

In this case, our agent branches between one of two actions. The agent first checks for mis-spelled words (action a10). This can be set to happen earlier in the chain (e.g., in iteration 1), however, our experience to date indicates this is a fruitful place for this action to occur. If mis-spellings are detected, they are flagged and a display page is sent to the user requesting them to accept the suggested or alternate corrections, or to add their own (a11). The spell checker we use is a component (source code) purchased from Sentry Software [18]. For each market exchange or proprietary application, one must extend its spelling dictionary with a number of local terms, manufacturer mis-spellings lists, and commonly used acronyms. Once the user corrects their spelling error, the agent process begins anew back at Iteration 1.

If there is no spelling error found, action a11 prints the message: "We have found 0 items for your (search string)." And it provides a number of standard hints for better searching of the catalog, and a request that they try again in the box at the base of the page.

3) Comparison Testing of EQUIsearch's Capabilities

Earlier we said that EQUIsearch is complementary to commercial search engines for online catalogs. To illustrate that point, this section provides an empirical evaluation of query performance both with and without the EQUIsearch Agent for a major commercial search engine relevant to e-commerce catalogs. In general, search engine performance can be measured in terms of four methods: retrieval effectiveness metrics (e.g., recall, precision), user satisfaction measures, transaction log analysis, and the critical incident technique. Silverman et al [34] examines the last three of these. In the current study, we provide a comparison based on the metrics enumerated in earlier Section 1.2.

3.1) Test of the Scaleup Hypothesis (H1)

In order to fully test the hypothesis that NLQ can scale up, we ideally should demonstrate EQUIsearch's deployment and operation in a number of e-commerce websites and online catalogs. To date we have only deployed it at a single market exchange, so the results of this section are an "existence proof". A valid question is: can we duplicate this deployment for a variety of catalogs, or if not, what constrains the scale-up? We return to this question in the conclusions, and focus attention here on the details of the existence proof.

Specifically, the existence proof website is EqualFooting.com (www.equalfooting.com), a B2B online marketplace for the "maintenance, repair, and operations" (MRO) sector: e.g., see [28]. This means basically EqualFooting sells industrial and construction supplies – something like a Home Depot for small contractors only with an order of magnitude more products than Home Depot offers. The company's official launch date was February 2000 and by June 2000 they were handling one million hits per day (by about 23,000 separate users daily). Also, at this writing their catalog integrates almost 450,000 products offered by over 2,000 sellers, although they project an order of magnitude growth in catalog size by year's end.

The catalog is stored internally in an Oracle database on dual processors to balance user load and to support Oracle Parallel Server. A hidden mirror site exists on the opposite side of the country to further address redundancy and load issues. At the front end is an application server (WebLogic) and an Enterprise Java Beans implementation of about ½ million lines of code, as of this writing, that performs all the functions of the website and that connects the users' web-browsing clients to the Oracle parallel servers holding the catalog. The application server consists of another 5 or 6 parallel machines with a load balancing capability. The EQUIsearch agent is deployed on the parallel application server machines as the sole searching interface to the catalog. Users at this website now utilize the NLQ via EQUIsearch, that in turn interfaces with Oracle and its CQ search technology to represent the user's interests.

The CQ capabilities within Oracle are known as the *inter*Media search engine. It should be noted that the conceptual search features of a product such as *inter*Media are not immediately usable at a given shopping site. For example, in order to make the production version of the Equalfooting.com database conceptual-search capable required significant investment and effort. We took advantage of that prior investment in order to quickly assemble the benchmark for the current study, but it might be interesting to readers if we briefly delineate the tasks that were involved in setting up conceptual search for a production catalog. The tasks were to create the three dictionaries always needed for conceptual search – spelling, stripping, and synonyms – as these are domain-specific items. Growing the thesaurus involved many false starts and deadends. For example, it is tempting to try and use an existing general purpose thesaurus such as WordNet from Princeton. This includes 95,000 words and all their synonyms, however, this thesaurus brings back too many synonyms, many of which are inappropriate (racial slurs, curses, body parts, religious terms, etc.). Plus most of the specialty terms of a given domain are omitted (e.g., chain saw, Phillips head, and safety gloves are among the 1,000s of items found in a hardware catalog). Instead, based on search log analyses, we assembled our own thesaurus for about 1500 terms critical to this domain with five synonyms for each. The second dictionary task was to assure the database would have a spell checker (Oracle doesn't ship with this functionality), so we purchased and installed one separately [18]. Although it came with 100,000 words it was necessary to embellish the spell checker's dictionary by adding: (1) the top 1,000 mis-spelled words from the user search logs and their corrections, (2) proper names of all manufacturers and suppliers (the spell-checker assumes initially that all proper names are errors) and how they might be mis-spelled, and (3) many 100s of acronyms with proper spelling (e.g., CD, DVD, HVAC, etc.). The third and final dictionary task was to massage the stop word list for the current domain as some generic stop words are relevant here and others are unique.

For a site that already has prepared all three dictionaries required by CQ (i.e., strip, synset, spell), the extra effort to add EQUIsearch is rather straightforward. The steps are to:

1) run a catalog crawler that extracts all the lexicon
2) construct the lexical knowledge bases (objects and attribute-value pairs)
3) author the relevant rule sets and encode them in the Φ_r
4) complete any interface code needed in the SQL builder of the agent to adapt it to the requirements of the CQ technology (*inter*Media in this case)

We performed these steps manually for the test catalog during the Fall 2000, and the agent was deployed as of November 17, 2000. It has been in continuous operation (24 hours a day, 7 days/week) since that time as the sole searching interface that users interact with. A new goal to enhance the portability of EQUIsearch is to construct automated tools that help us to do each of the steps more easily, both for helping to maintain the current site, and for deploying EQUIsearch for new catalogs and exchanges.

3.2) Test of the Timing Hypothesis (H2)

Once the agent went live in the online catalog that users access via the web, we no longer had access to it for scientific inquiry, and we were not allowed to alter the settings of that site, turn on and off the agent (it stays on all the time on that site), or run studies. Instead, for further testing we were given access to an alternative online catalog – the developers' benchmark catalog. The benchmark catalog is a smaller version of the actual production site (172,000 vs. 450,000 records), and it exists on a relatively slow server on the intranet that is accessed by 50 or so developers from their workstations. The production database by contrast is hosted on multiple high speed, parallel servers on the internet.

Both of these catalogs are in Oracle, and whether on the inter- or intra-net, EQUIsearch accesses Oracle via the WebLogic application server. Thus timing tests should be viewed for their relative outcomes. In general, although the server we had access to is significantly slower than the bank of production site machines, the internet has several orders of magnitude more traffic and far greater latency, so the timing test results will be a couple of seconds faster on average than actual user times.

Timing test results for noun (item name or synonym of a name) search with and without the agent are displayed in Tables 2a and b. The left column of that table displays the search term, while the next column shows the total hits retrieved. Here, we are

Table 2 –Timing and Effectiveness Statistics for Item (Noun) Search
(a) Without the Agent

	With out Agent											
Search String	TOTAL RTRVD	RETRVD & PERTNT	RETRVD & IRRLVT	NOT RTRV & PERTNT	NOT RTRV & IRRLVT	Quality	Recall	Precis ion	Actual	Elapsed time (ms)		
										# 1	# 2	Avg(ms)
aircompressor	178	88	90	0	171891	0.5	1.0	0.5	88	571	561	**566.0**
ballast	738	3	735	0	171331	0.0	1.0	0.0	3	431	441	**436.0**
blower	480	364	116	0	171589	0.8	1.0	0.8	364	390	400	**395.0**
brad	298	159	139	0	171771	0.5	1.0	0.5	159	441	431	**436.0**
brush	792	660	132	0	171277	0.8	1.0	0.8	660	341	341	**341.0**
cabinet	668	436	232	0	171401	0.7	1.0	0.7	436	350	340	**345.0**
calculator	78	49	29	0	171991	0.6	1.0	0.6	49	361	371	**366.0**
chipper	497	290	207	0	171572	0.6	1.0	0.6	290	421	401	**411.0**
chisel	456	428	28	0	171613	0.9	1.0	0.9	428	300	310	**305.0**
cleaner	1685	438	1247	0	170384	0.3	1.0	0.3	438	521	531	**526.0**
compressor	178	88	90	0	171891	0.5	1.0	0.5	88	571	561	**566.0**
connector	2815	1182	1633	0	169254	0.4	1.0	0.4	1182	1292	1282	**1287.0**
cupboard	668	436	232	0	171401	0.7	1.0	0.7	436	351	361	**356.0**
cutter	18339	2519	15820	0	153730	0.1	1.0	0.1	2519	811	811	**811.0**
drill	10787	7745	3042	0	161282	0.7	1.0	0.7	7745	3084	3134	**3109.0**
fans	480	364	116	0	171589	0.8	1.0	0.8	364	400	410	**405.0**
glove	1303	1122	181	0	170766	0.9	1.0	0.9	1122	551	551	**551.0**
grinder	497	290	207	0	171572	0.6	1.0	0.6	290	421	401	**411.0**
hammer	701	586	115	0	171368	0.8	1.0	0.8	586	430	430	**430.0**
ladder	178	112	66	0	171891	0.6	1.0	0.6	112	451	461	**456.0**
mallet	701	586	115	0	171368	0.8	1.0	0.8	586	450	420	**435.0**
mitten	1303	1122	181	0	170766	0.9	1.0	0.9	1122	551	551	**551.0**
nail	298	159	139	0	171771	0.5	1.0	0.5	159	441	431	**436.0**
paper	677	257	420	0	171392	0.4	1.0	0.4	257	390	390	**390.0**
pipe	8945	2488	6457	0	163124	0.3	1.0	0.3	2488	791	761	**776.0**
processor	78	49	29	0	171991	0.6	1.0	0.6	49	380	381	**380.5**
saw	1458	598	860	0	170611	0.4	1.0	0.4	598	391	391	**391.0**
screw	7677	5769	1908	0	164392	0.8	1.0	0.8	5769	651	651	**651.0**
snapper	18339	2519	15820	0	153730	0.1	1.0	0.1	2519	811	831	**821.0**
soap	1685	438	1247	0	170384	0.3	1.0	0.3	438	531	530	**530.5**
tube	8945	2488	6457	0	163124	0.3	1.0	0.3	2488	771	761	**766.0**
Average						0.6	1.0	0.6				601.2
Standard Deviation						0.2	0.0	0.2				507.2

(b) With the Agent

	With Agent											
Search String	TOTAL RTRVD	RETRVD & PERTNT	RETRVD & IRRLVT	NOT RTRV & PERTNT	NOT RTRV & IRRLVT	Quality	Recall	Precis ion	Actual	Elapsed time (ms)		
										# 1	# 2	Avg (ms)
aircompressor	88	88	0	0	171981	1.0	1.0	1.0	88	391	391	**391.0**
ballast	3	3	0	0	172066	1.0	1.0	1.0	3	290	280	**285.0**
blower	364	364	0	0	171705	1.0	1.0	1.0	364	321	311	**316.0**
brad	159	159	0	0	171910	1.0	1.0	1.0	159	320	320	**320.0**
brush	660	660	0	0	171409	1.0	1.0	1.0	660	250	260	**255.0**
cabinet	436	436	0	0	171633	1.0	1.0	1.0	436	290	280	**285.0**
calculator	49	49	0	0	172020	1.0	1.0	1.0	49	291	291	**291.0**
chipper	290	290	0	0	171779	1.0	1.0	1.0	290	330	320	**325.0**
chisel	428	428	0	0	171641	1.0	1.0	1.0	428	260	270	**265.0**
cleaner	438	438	0	0	171631	1.0	1.0	1.0	438	351	361	**356.0**
compressor	88	88	0	0	171981	1.0	1.0	1.0	88	390	400	**395.0**
connector	1182	1182	0	0	170887	1.0	1.0	1.0	1182	561	561	**561.0**
cupboard	436	436	0	0	171633	1.0	1.0	1.0	436	290	290	**290.0**
cutter	2519	2519	0	0	169550	1.0	1.0	1.0	2519	351	351	**351.0**
drill	7745	7745	0	0	164324	1.0	1.0	1.0	7745	1352	1332	**1342.0**
fans	364	364	0	0	171705	1.0	1.0	1.0	364	331	301	**316.0**
glove	1122	1122	0	0	170947	1.0	1.0	1.0	1122	420	420	**420.0**
grinder	290	290	0	0	171779	1.0	1.0	1.0	290	311	321	**316.0**
hammer	586	586	0	0	171483	1.0	1.0	1.0	586	300	310	**305.0**
ladder	112	112	0	0	171957	1.0	1.0	1.0	112	321	331	**326.0**
mallet	586	586	0	0	171483	1.0	1.0	1.0	586	301	311	**306.0**
mitten	1122	1122	0	0	170947	1.0	1.0	1.0	1122	391	391	**391.0**
nail	159	159	0	0	171910	1.0	1.0	1.0	159	310	310	**310.0**
paper	257	257	0	0	171812	1.0	1.0	1.0	257	291	291	**291.0**
pipe	2488	2488	0	0	169581	1.0	1.0	1.0	2488	400	390	**395.0**
processor	49	49	0	0	172020	1.0	1.0	1.0	49	291	301	**296.0**
saw	598	598	0	0	171471	1.0	1.0	1.0	598	301	291	**296.0**
screw	5769	5769	0	0	166300	1.0	1.0	1.0	5769	460	450	**455.0**
snapper	2519	2519	0	0	169550	1.0	1.0	1.0	2519	380	370	**375.0**
soap	438	438	0	0	171631	1.0	1.0	1.0	438	360	380	**370.0**
tube	2488	2488	0	0	169581	1.0	1.0	1.0	2488	391	391	**391.0**
Average						1.0	1.0	1.0				373.8
Standard Deviation						0.0	0.0	0.0				190.6

only discussing the timing statistics in this table, so we will focus on the last three columns of the table for now and return to a discussion of the rest of the table in the next section. In that regard, the rows of Table 2a display 31 keyword searches without the NLQ agent, below which are two rows of various summary statistics. Some of these keywords are actual item names (e.g., hammer, glove), while others are synonyms (e.g., mallet, mitten). Table 2b displays the results for the same set of search strings, but this time with the NLQ agent running. By inspection, the agent or NLQ search is faster than CQ search (373 vs. 601 milliseconds averaged over two runs per query). In this case the delay is reduced by about 1/3rd of a second on average. However, these differences aren't noticeable and one may reasonably claim that NLQ is on a par with CQ for noun search.

Next we compared the timing for NLQ vs. CQ in the case of searches with both nouns and adjectives (objects and attribute-values) as Table 3 depicts. The format is identical to that in Table 2. Two differences from Table 2 are that we only used 17 search strings (rows) here, however, we have again repeated each search 2 times so that the timing statistics are for the average across the 34 replications. The reader should not attempt to compare the times between Tables 2 and 3 since they were collected on different days and at different times of the workday. It is may be partially a nuance of the varying server load and intranet latencies that make the results in Table 3a seem better than in Table 2a even though this second test involves a more complex search string. Also, a factor may be the number of hits found, and since the Table 3a searches are more precise they tend to return less hits.

Examining the response times within Table 3 alone, one can see that NLQ is far slower than CQ on this more complex type of search (1.8 vs. .5 seconds on average). The absolute difference is still within the toleration level of most shoppers, and we view this as a reasonable price to pay. One can reasonably argue that the two sample means are on a par (in fact a t-test of the sample means indicates no difference at the alpha = .05 level). Still, it is instructive to discuss the differences a little further.

It should surprise no one that the NLQ is faster than CQ in noun search and slower in noun-adjective search. In noun or object search, CQ must dynamically sort and index the entire munge (many sub-fields de-normalized) while NLQ need only sort through the Item-Name field since it has narrowed the search by labeling the token. Likewise, in object-attribute search, CQ behaves the same as before, but NLQ's SQL statements cause the Oracle engine to sort an additional set of fields (all those with attributes) and then do a soft join and eliminate items not in all parts of the join. That is, both sets of results reflect the fact that the processing time is dominated by operations that must be carried out by the Oracle *inter*Media product, not by the Markovian processes of the NLQ agent. In general, the number of operations required by Oracle (or any relational database sorting process) is on the order of O(n Log (n)). While this iteration may be solved in log-linear time and thus seems faster than that for the MDP (which is P-complete on n), in fact there is a gross imbalance in "n" used for complexity during the MDP and during the actions for sorting of the catalog. In the former n is on the order of 10E2 or less for number of action-state pairs in the stationary policy set, while in the catalog n is typically on the order of 10E6 to 10E7 product records to be sorted. The only exception to this is that the spelling checker typically has on the order of n = 10E6 terms. Thus the commercial search engine is typically the slowest process, and when NLQ can narrow the search it will beat CQ.

Table 3 – Timing and Effectiveness Statistics for Noun and Adjective Queries

(a) Without the Agent

Search String	TOTAL RETRVD	RETRVD & PERTNT	RETRVD & IRRELVT	NOT RTRV & PERTNT	NOT RTRV & IRRLVT	Quality	Recall	Precision	Actual	Time
										2 Run Avg (ms)
16 oz hammer	28	23	5	0	172041	0.8	1.0	0.8	23	**490.5**
10 inch nail	47	38	9	0	172022	0.8	1.0	0.8	38	**601.0**
bolt cutter	33	28	5	0	172036	0.8	1.0	0.8	28	**461.0**
copier paper	55	55	0	0	172014	1.0	1.0	1.0	55	**626.0**
cotton glove	265	57	208	0	171804	0.2	1.0	0.2	57	**565.5**
crimped brush	120	79	41	0	171949	0.7	1.0	0.7	79	**360.5**
cutter wheel	16	12	4	0	172053	0.8	1.0	0.8	12	**425.5**
leather glove	334	84	250	0	171735	0.3	1.0	0.3	84	**510.5**
nail hammer	17	12	5	0	172052	0.7	1.0	0.7	12	**531.0**
pipe clamp	460	239	221	0	171609	0.5	1.0	0.5	239	**440.5**
power cord	56	27	29	0	172013	0.5	1.0	0.5	27	**751.5**
protective apron	175	68	107	0	171894	0.4	1.0	0.4	68	**766.0**
roofing nail	9	9	0	0	172060	1.0	1.0	1.0	9	**405.5**
safety tape	95	56	39	0	171974	0.6	1.0	0.6	56	**521.0**
steel brush	338	166	172	0	171731	0.5	1.0	0.5	166	**385.5**
tape measure	368	162	206	0	171701	0.4	1.0	0.4	162	**601.0**
white paper	64	28	36	0	172005	0.4	1.0	0.4	28	**375.5**
wire connector	398	60	338	0	171671	0.2	1.0	0.2	60	**891.0**
Average						0.6	1.0	0.6		539.4
Standard Deviation						0.3	0.0	0.3		147.3

(a) With the Agent

Search String	TOTAL RETRVD	RETRVD & PERTNT	RETRVD & IRRELVT	NOT RTRV & PERTNT	NOT RTRV & IRRLVT	Quality	Recall	Precision	Actual	Time
										2 Run Avg (ms)
16 oz hammer	24	23	1	0	172045	1.0	1.0	1.0	23	**661.0**
10 inch nail	38	38	0	0	172031	1.0	1.0	1.0	38	**510.5**
bolt cutter	28	28	0	0	172041	1.0	1.0	1.0	28	**365.5**
copier paper	54	54	0	1	172014	1.0	1.0	1.0	55	**376.0**
cotton glove	57	57	0	0	172012	1.0	1.0	1.0	57	**590.5**
crimped brush	79	79	0	0	171990	1.0	1.0	1.0	79	**335.5**
cutter wheel	12	12	0	0	172057	1.0	1.0	1.0	12	**315.5**
leather glove	84	84	0	0	171985	1.0	1.0	1.0	84	**656.0**
nail hammer	12	12	0	0	172057	1.0	1.0	1.0	12	**386.0**
pipe clamp	239	239	0	0	171830	1.0	1.0	1.0	239	**320.0**
power cord	27	27	0	0	172042	1.0	1.0	1.0	27	**2944.0**
protective apron	68	68	0	0	172001	1.0	1.0	1.0	68	**4992.5**
roofing nail	9	9	0	0	172060	1.0	1.0	1.0	9	**321.0**
safety tape	56	56	0	0	172013	1.0	1.0	1.0	56	**11797.0**
steel brush	166	166	0	0	171903	1.0	1.0	1.0	166	**2663.5**
tape measure	162	162	0	0	171907	1.0	1.0	1.0	162	**5363.0**
white paper	12	12	0	16	172041	0.4	0.4	1.0	28	**485.5**
wire connector	60	60	0	0	172009	1.0	1.0	1.0	60	**555.5**
Average						1.0	1.0	1.0		1868.8
Standard Deviation						0.1	0.1	0.0		2956.7

Since the average search in shopping sites tends to be about 2.3 words long, many of the queries are in fact just a single noun. In many of the other cases, the impact is not overly severe in absolute terms and use of load shedding, parallelism, and prudent algorithms should lead to search times of no more than a couple of seconds (excluding latency) for a majority of the queries. Obviously, queries with large numbers of hits in the database can take longer, as can queries with multiple spelling errors.

Also, there are some problematic queries. The worst example in Table 3 is “safety tape”, which on the syntactic level seems almost identical to “cotton glove” and which returns comparable numbers of hits, yet the response time is about 20 times as long.

These types of problem queries require individual investigation to learn where the trouble arises. Sometimes it is due to poor specification in the synonym list which causes the query to be expanded extensively, thereby leading to a lengthy join and eliminate process. Other times it is due to one of the tokens appearing in a large portion of the records of the database (e.g., a number like “2”). As these problems are discovered, workarounds are developed, often in the form of fixes to one of the dictionaries or KBs or as new rules for the agent.

3.3) Tests of the Quality Hypothesis (H3)

Proofs of NLQ scaleup and timing parity of NLQ with respect to conceptual query (CQ) would be of minimal value if the effectiveness of the search were not improved by an NLQ agent. Table 4 depicts the results of the same 31 item name searches of the benchmark catalog as used in earlier Table 2 -- column 1 shows the search strings. The first 31 rows have the agent turned off, so this is strictly conceptual search using Oracle *inter*Media. As in the real system, Oracle is using stemming, the 8,000 word synonym list, and conjunctive search. The second group of 31 rows of the table are for the identical search strings, but with the NLQ agent turned on, as the second column indicates. The 3rd column shows the total items retrieved, which is further broken into the retrieved items that are pertinent and irrelevant in the subsequent 2 columns. The 6th and 7th columns show statistics on the non-retrieved items. The last three columns of the table compute the metrics mentioned at the outset – the commonly used recall and precision, plus our merged “quality” metric. At the end of each group of 31 rows, one can view the Mean and Standard Deviation for those searches.

One can see by inspection of column 6 that in this test there are no false negatives (not retrieved but pertinent) left behind by either query method. This has the effect of driving Recall to unity, and of leaving Quality and Precision effectively equal. Also, one can see by inspection that on average the agent’s Quality is 0.90 as compared to 0.63 for the average of the 31 searches with the conceptual query method.

In a similar vein, Table 5 presents the quality and effectiveness statistics for the 21 attribute-object query strings of earlier Table 3. Here again we see the quality improves from .6 on average without the agent to .93 when the agent is engaged. The results are not uniformly better. In a few cases the agent does no better than the CQ search, and in two cases (both involving “paper”) it even does worse than CQ search in terms of having some relevant items left not-retrieved. Upon close inspection, one can attribute these to the missing data problem alluded to in the introduction to this paper. For example, consider “white paper” for which the agent retrieved 12 pertinent items, but for which the CQ search found the other 16 that are in the catalog. For those 16 items, their color attribute is missing, and the CQ agent matched them from a description of the product contained in the “short description” in the munge. These kind of outcomes are not overwhelming in the benchmark catalog though, and on average the agent is out-performing the CQ search. However, in catalog sites where missing data is an overwhelming problem, one can expect CQ search to begin to approach the quality and precision of NLQ search unless the effort is made to clean the catalog. One would expect this type of effort is necessary anyways for a variety of reasons.

3.4) Other Results

We would be entirely remiss if we ended without also pointing out the fundamental advantage of adding a parser to any search engine – it can parse free-form sentences so long as they are written in the lexicon (or synonym set) of the catalog. As Table 4 shows, in fact, EQUIsearch processes whole sentences at about the same speed as it does the noun-adjective phrases. Further, one can compare the hits to earlier Table 3 and notice it suffers no drop off in quality or precision. Table 4 also shows what is well known: absent a parser, CQ search is unable to strip enough of the words to make sense of the queries and it returns no hits for all but one of the sentences.

Table 4 – Results of Full Sentence Queries With and Without EQUIsearch Agent

	With Agent												Without Agent
Search String	TOTAL RETRVD	RETRVD & PERTNT	RETRVD & IRRELVT	NOT RTRV & PERTNT	NOT RTRV & IRRLVT	Quality	Recall	Precision	Actual	Elapsed time (ms)			Retrieved
										#1	#2	Avg	
show me all power cords	27	27	0	0	172042	1.0	1.0	1.0	27	1682	1612	**1647.0**	0
list me cotton gloves	57	57	0	0	172012	1.0	1.0	1.0	57	391	381	**386.0**	0
show me gloves made of leather	84	84	0	0	171985	1.0	1.0	1.0	84	441	411	**426.0**	0
i want to buy leather gloves	84	84	0	0	171985	1.0	1.0	1.0	84	400	500	**450.0**	0
show me steel brushes	166	166	0	0	171903	1.0	1.0	1.0	166	2654	2644	**2649.0**	0
i want to buy a crimped brush	79	79	0	0	171990	1.0	1.0	1.0	79	370	371	**370.5**	0
let me see leather gloves if any with you	84	84	0	0	171985	1.0	1.0	1.0	84	400	410	**405.0**	0
list me some good cotton gloves	57	57	0	0	172012	1.0	1.0	1.0	57	371	381	**376.0**	0
let me see some leather gloves	84	84	0	0	171985	1.0	1.0	1.0	84	400	400	**400.0**	0
list me nails for roofing	9	9	0	0	172060	1.0	1.0	1.0	9	261	261	**261.0**	0
list me all wheel cutters	12	12	0	0	172057	1.0	1.0	1.0	12	300	300	**300.0**	0
show me bolt cutter	28	28	0	0	172041	1.0	1.0	1.0	28	321	331	**326.0**	0
list wire connectors	60	60	0	0	172009	1.0	1.0	1.0	60	360	380	**370.0**	11
list brush made of steel	166	166	0	0	171903	1.0	1.0	1.0	166	2644	2664	**2654.0**	0
Average						1.0	1.0	1.0				787.2	
Standard Deviation						0.0	0.0	0.0				860.9	

4) Results Analysis and Concluding Remarks

This paper has presented an MDP algorithm for an NLQ agent that can complement and improve search engines for online e-commerce shopping catalogs. The results show that the agent improves the quality and precision of the search with no significant overall impact on response time. These results hold true for a large scale example deployment.

Several lessons learned are worth discussing further:

- **Scaling Up NLQ** – The result to date is encouraging. The agent worked and is continuing to work at a relatively large-scale market exchange, which previously operated with CQ search alone. We hope to deploy the EQUIsearch agent at further sites in the future as part of the effort to address this line of investigation.
- **CQ Paves the Way for NLQ** – Relational DBMSs ship with CQ features, however, the three standard dictionaries (strip, synonym, and spell) need to be enabled and domain-filled before CQ can work for a given shopping catalog. The good news is that these are the same three dictionaries that NLQ needs, and if a site has already

developed them and deployed CQ, then there is only minimal extra effort to also deploy NLQ.

- **Automatic KB Extraction Tools Would Further Ease NLQ Adoption** – For the current deployment of EQUIsearch, we manually extracted the lexicon and built the KBs and other interfaces. However, as the catalog is altered over time, or as new ones are attempted, the EQUIsearch agent's KBs must be modified and updated. For these purposes, we are already developing a suite of automated KB extraction and instantiation tools, plus other maintenance features (e.g., SQL interfaces to a range of commercial DBMS products).
- **Data Cleansing Obstacles Remain for Any Search Method** – Possibly the most serious obstacle to scale up in any unified shopping catalog is the numerous typoes, missing item name and other data, and poor quality of attribute information. This obstacle is not unique to NLQ search, but equally plagues the CQ and conjunctive keyword searching methods as well.
- **Speed Differences are a NonIssue For Most Cases** – It seems that NLQ is faster for noun search, but slower than CQ for noun-attribute pair searching. However, the average search involves just over 2 terms per query, so noun searching is a major mode of user query. But in either case, the time differences are not statistically significant, although precision and quality improvements are being achieved for all types of search.
- **NLQ Agent Provides Precision and Quality Improvement** – The results to date reflect about a 50% improvement in quality and precision when NLQ is added to the CQ capability. This means that users experience noticeably shortened retrieval sets, and that the items retrieved include far less false positives. In addition there are fewer false negatives or relevant items omitted. An implication for the effectiveness studies and metrics literature is that recall and precision aren't always inversely coupled. Certainly for introducing an innovation such as NLQ, it is possible to shift precision in the positive direction without adversely affecting recall.
- **NLQ Agent Offers Parsing Services CQ Can't Provide** – Many shopping sites are starting to add chatterbots like AskJeeves that provide navigation help and answer site or content questions in natural-like language. The results to date indicate that users like this type of self-service help, and that when it is present, they build up a higher expectation that the catalog search will behave in a similarly naturalistic way and they no longer limit their queries to the short keyword format (2.3 words on average) [4]. They pose English-like sentences and questions to the catalog search engine and it seems they are adversely affected by the inability of CQ search engines to parse their questions. As the testing here demonstrated, NLQ search in general, and EQUIsearch in particular is able to parse and reliably answer full sentences and questions in the domain of the catalog.

To close, we believe this research demonstrates that NLQ search is able to compete with industrial strength CQ search. It holds its own in terms of timing as scaleup occurs, and it improves quality, precision, and parsing capability. For catalogs that are already adding CQ search, there is not much added effort to also include an NLQ agent. It thus seems likely that many catalogs will begin to add NLQ search in the near future in the effort to improve the online shopping experience.

ACKNOWLEDGEMENT

A project like this involves the help of many hands. We thank Raja Balasubramanian and Raghava Vemula for flashes of occasional programming brilliance and for being so patient throughout the implementation and benchmarking tests; Danny Zhao and his database crew for guiding us through the maze of schemas and versions; and Gary Vaughn, Angela Zaev and the rest of the catalog team for being remarkably responsive.

REFERENCES

1. Anon., "Winning the Online Consumer: Insights Into Consumer Behavior," Cambridge: Boston Consulting Group, March 2000, www.bcg.com.
2. Pollock, A., Hockley, A., "What's Wrong with Internet Searching," D-Lib Magazine, Ipswich, UK: BT Laboratories, March 1997, avail. at www.dlib.org/dlib/march97/bt/03pollock.html
3. Neilsen, J., "Search and you *may* find" Alert Box, July 15, 1997, avail. at http://www.useit.com/alertbox/9707b.html
4. Hagen, PR, Manning, H, Paul, Y, "Must Search Stink?", Cambridge: Forrester Research, June 2000. (www.forrester.com)
5. Blum, A., "Add Natural Language Search Capabilities to Your Site with English Query," Microsoft Interactive Developer, April 1998, www.microsoft.com/Mind/0498/equery.htm
6. Anon., "Revolutionizing the Search for Products at E-Commerce Sites," Littleton: Easy Ask Inc., March 2000, www.easyask.com
7. "BotSpot Categories - Chatter Bots," http://www.botspot.com/search/s-chat.htm
8. "Virtual Personalities", http://www.vperson.com/sapphire2000/index.html
9. http://www.askjeeves.com/
10. http://support.dell.com/us/en/askdudley/
11. Bellman, R., "A Markovian Decision Process," J. Math. And Mech., v.6, 1957, pp 679-93.
12. Howard, R, Dynamic Programming and Markov Processes, New York: John Wiley, 1963.
13. White, C., "Procedures for the Solution of a Finite Horizon, Partially Observable, Semi-Markov Optimization Problem," Operations Research, v. 24, 1976, pp. 348-58.
14. Monahan, GE, "A Survey of Partially Observable Markov Decision Processes: Theory, Models, and Algorithms," Management Science, v.28,n.2, 1982, pp. 2-16
15. Littman, ML, Cassandra, AR, Kaebling, LP, "Efficient Dynamic Programming Updates in Partially Observable Markov Decision Processes," Dec. 1995.
16. Bernstein, DS, Zilberstein, S, Immerman, N, "The Complexity of Decentralized Control of Markov Decision Processes," Proc. 16th Conf. Uncertainty in Artificial Intelligence, Stanford, California, July 2000.
17. Porter, MF, "An Algorithm for Suffix Stripping," *Program*, v.14, n.3, July 1980, pp.130-137.

18. Sentry Spelling Checker avail at http://www.wintertree-software.com/dev/ssce/java/index.html
19. Anon., Oracle 8i *inter*Media Text 8.1.6 – Technical Overview, Redwood City: Oracle, 2000.
20. Anon., DB II and the Universal Database (UDB), Yorktown Hgts: IBM, 2000
21. Anon., SQL Server 2000 Database Management System, Redmond: Microsoft, 2000
22. AltaVista Search Engine (AVSE) version 3.1, available from www.altavista.com
23. Anon., "Datasheets on Frictionless Search Engine," 2000, avail. at http://www.frictionless.com/solutions/datasheetform.html
24. William S. Cooper, "On Selecting a Measure of Retrieval Effectiveness," Journal of the American Society for Information Science, v.24, 1973, pp. 87-100, 413-424.
25. Dagobert Soergel, "Is User Satisfaction a Hobgoblin?," Journal of the American Society for Information Science, v.27, July-August 1976,pp. 256-259.
26. Tonta, Y., "Analysis of Search Failures in Document Retrieval Systems: A Review," Public Access Computer Systems Review, v. 3, no.1 (1992): 4-53.
27. Owei, V., "Natural Language Querying of Databases: An Information Extraction Approach in the Conceptual Query Language," IJHCS, v. 53, 2000, pp. 439-92.
28. Chang, T., Sciore, E., "A Universal Relation Data Model with Semantic Abstraction," IEEE Trans. KDE, v.4, 1992, pp. 23-33.
29. Chan, HC, Wei, KK, Siau, KL, "User Database Interface: The Effect of Abstraction Levels on Query Performance," MIS Quarterly, v.17, 1993, pp. 441-64.
30. Jarke, M, Gallersdorfer, R, et al., "Concept-Base – A Deductive Object Base for Meta Data Management, "J. of Intell. Inf.Sys., v.4, 1995, pp. 167-92.
31. Cowie, J, Lehnert, W., "Information Extraction," CACM, v. 39, 1996, pp. 80-91.
32. Vilian, M., "inferential Information Extraction," in MT Pazienza (ed.), Information Extraction: Toward Scalable, Adaptable Systems, Berlin: Springer, 1998, pp. 95-119.
33. Dekleva, SM, "Is Natural Language Querying Practical?" Data Base, v. 25, 1994, pp. 24-36.
34. Silverman, BG, Bachann, M, et al, "Buyer Decision Support Systems and Search Agents for eCommerce Websites," (submitted for pub.), http://www.seas.upenn.edu:8080/~barryg/BDSS.html
35. Cleverdon, C. W., and E. M. Keen. 1966. Factors determining the performance of indexing systems, volume 1: design, volume 2: test results. Cranfield, England: Aslib Cranfield Research Project.
36. Sparck Jones, K. 1981. The Cranfield tests. In: Information Retrieval Experiment, K. Sparck Jones (ed.). London: Butterworths: pp. 256-284.
37. Swets, JA, "Information Retrieval Systems," SCIENCE, July 19, 1963, pp. 245-50.
38. van Rijsgergen, CJ, "Information Retrieval, New York: _______, 2nd ed.,1979.
39. Yee & Layne, Modern Information Retrieval, New York: , 1998.