Searching the Web More Effectively With Multiple Simultaneous Queries

Francisco Corella and Karen Lewison

October 2008

Patent Pending

Abstract

We describe a Web search facility that reduces the time and effort that it takes the user to home in on the desired results for difficult search problems. When the user enters a query the search facility anticipates possible follow-up queries, issues them immediately, and allows the user to browse the search results of the original query and these additional queries simultaneously. Additional queries may include a respelling of the original query, related queries, and/or subqueries. (By subquery we mean a query consisting of a subset of the search terms of the original query.) We describe a parallel algorithm that efficiently produces an optimal set of subqueries and their results in the important special case where the original query has zero results; although it is rare for a query that targets the Web at large to have no results, the zero-result case is important for local searches, queries that target a particular site, and intranet searches.

We have built a prototype of such a search facility as a client-side script, implemented on the Adobe Flex platform, and thus running on the Flash plug-in, that accesses the Yahoo search engine via the Yahoo Astra Web API library. The Yahoo search engine has not been modified for this purpose, so our innovations are implemented entirely on the client side. We point out, however, that it would be beneficial to transfer parts of the implementation to the server side, and explain how this could be done. The prototype only handles purely conjunctive queries, but we also describe a method for handling general Boolean queries, and we describe an extension of the parallel zero-result algorithm to the general case. The prototype is available for use at noflail.com.

1. Introduction

A search engine with a good page-ranking algorithm can be extremely effective at locating a desired page or set of pages among the billions of pages of the World Wide Web. The button labeled *I'm Feeling Lucky* on the Google Home Page is no doubt meant to advertise that it is not unreasonable to expect the desired result to be the very first result listed.

From time to time, however, every user runs into a difficult search problem. The user may not know what terms are used in a target document to describe a given concept, or it may be hard to think of a combination of terms that will unearth a page from under a mass of more highly ranked pages. When this happens, the unfortunate user may issue one after another many queries with slightly different terms. Then the user may perhaps go back to earlier queries to dig deeper into the result set of each of them. This can be very time consuming and frustrating; informally, we refer to it as *flailing*.

To help the user in these difficult cases, we propose a new kind of search facility that takes on the drudgery of issuing many different queries and eliminates the delays incurred in going back and forth between their result sets. When the user issues a query, the facility anticipates possible follow-up queries, runs them automatically, in parallel, and allows the user to browse the results of the original query and these additional queries simultaneously.

We have implemented a prototype of such a search facility as a client-side script that runs within the user's browser and accesses the search engine through a Web API. It is built on Flex [1] and thus uses the Adobe Flash plug-in. It interacts with the Yahoo search engine using the Yahoo Astra Web API library [2]. It is available for use at noflail.com.

The fact that the prototype relies on Flex and the Yahoo Astra library is by no means essential. Other search engines expose Web APIs that we could perhaps have used, although we have not researched this. JavaScript, Silverlight or other client-side platforms could have been used instead of Flex. Furthermore, the fact that the new functionality is implemented entirely on the client side is due to the fact that we do not have access at this time to the server-side internals of a search engine; otherwise it would be beneficial to implement some of the new functionality on the server side. This is discussed in more detail below, in Section 6.4.

The remainder of this white paper is organized as follows. Section 2 examines three kinds of possible additional queries. Section 3 describes a method for issuing such additional queries and making their results available to the user simultaneously. Section 4 describes a parallel algorithm that identifies and runs an optimal set of *subqueries* in the special case where the original query has zero results. Section 5 describes a method, not included in the prototype, for handling general Boolean queries, i.e. queries that combine conjunction, disjunction and negation. Section 6 contains additional remarks, including an explanation of how some of the new functionality could be moved to the server side and why this would be beneficial. References can be found in Section 7.

2. Selection of additional queries

Today, most search engines suggest queries to the user. Suggested queries fall into two categories: *respelled queries*, and *related queries*.

A respelled query is formed by correcting one or more misspellings found in a user query. For example, the query *madnna lirics* results in the suggestion: *Did you mean: madonna lyrics*?

Related queries are suggested differently by different search engines:

- Ask suggests related queries as the user is typing a query into the query box. The suggested queries are possible completions of the partially typed query.
- Microsoft's Live Search, on the other hand, suggests related queries after the user has submitted her query. A list of suggested queries is included in the top-left corner of each page of results. These suggested queries may or may not be completions of the user's query.
- Google and Yahoo suggest related queries both before and after submission of the user's query. Those suggested before submission are often, but not always, query completions. The help page that describes the *Search Assist* feature of Yahoo [3], makes a distinction between two different kinds of related queries, both of which are shown as the user is typing her query; but it is not clear what the precise difference is, and the Yahoo Astra API [2] provides only one kind of related queries.
- Cuil has a rich set of features involving related queries [4]. Before submission, it suggests query completions and sometimes a direct link to a site. After submission, it may suggest related queries in a top-left panel of each page of results, sorted by categories; these queries are not used as stand-alone queries; they are instead appended to the original query. After submission, Cuil may also suggest related queries that narrow the original query and are shown as tabs across the top of the page. This has similarities to our approach, discussed below in Section 3.

Our prototype suggests related queries as the query is being entered. We use for this purpose the related queries provided by the Yahoo Astra API. If the API made a distinction between completions and other related queries, we would prefer to use completions only, to give the user more control over the suggestions being displayed as the query is typed.

The key point we want to make, however, is what the prototype does once the user has submitted the query. As soon as the results of the query are available, the prototype issues additional queries without user intervention. These additional queries are processed while the user is beginning to inspect the results of the original query. The results of the additional queries become available within a few seconds and the user can then browse simultaneously the results of the original query and those of the additional queries.

As additional queries we use the two kinds of queries that today's browsers use as suggestions, viz. respelled queries and related queries, plus a third kind, viz. *subqueries*. By *subquery* we mean a query formed by taking the conjunction of a subset of the terms of the original query. (We assume for the time being that the original query itself is purely conjunctive. General Boolean queries are discussed below in Section 5.) It is important, however, not to overwhelm the user, nor the search engine, with too many additional queries. The number of related queries is potentially unlimited; the Yahoo API that we are using provides up to 50. The number of subqueries is limited, but exponential in the number of terms of the original query. As a practical heuristic, our prototype does the following:

(1) The prototype asks if there is a suggested respelling of the query. If the Yahoo API provides one, the prototype uses it as an additional query.

(2) The prototype requests up to 7 related queries. If the Yahoo API returns any related queries, the prototype uses them as additional queries, and uses no subqueries.

(3) If the Yahoo API returns no related queries, and the original query has more than 1 but no more than 7 terms, the prototype uses subqueries as additional queries. Specifically, it uses the subqueries obtained by removing exactly one of the terms of the original query. (To see the result sets of other subqueries, with fewer terms, the user can use any of the subqueries listed in the left-hand side as a follow-up query, as described in Section 3, and repeat the process as needed; an alternative strategy would be to show more subqueries at once for queries that do not have too many terms.)

Related queries are typically available when the original query is a *single-concept query*, consisting of a word or phrase (quoted or unquoted) that refers to one concept. On the other hand, they are typically not available when the user is looking for Web pages that mention two or more concepts, and thus enters a *multi-concept query*, consisting of two or more unrelated words or phrases. In this case subqueries fill the void and can be very useful. An alternative strategy, however, is discussed in Section 6.2.

3. Running queries and showing results

In this section we describe a possible implementation of a search facility that runs multiple queries and shows their results simultaneously to the user, as demonstrated in our prototype. The section is best read while experimenting with the prototype at noflail.com.

When the prototype is launched, it presents to the user an initial page. This page has a text input field or query box where the user enters a query, and a button to submit the query. Suggestions are made to the user as the query is being typed, as discussed above. Once the user has submitted the query, the search facility advances to the main page and remains at the main page thereafter. The main page has the same query box and button for entering and issuing queries, but they are located at the very top of

the page. Underneath, most of the real estate of the page is taken up by three boxes or panels, which are resizable along the horizontal axis.

The central panel is (initially) wider and starts out showing the results of the query that the user has submitted. A user who encounters the new search facility and has no time or inclination to explore its new features can ignore the other two panels and use the central panel together with the query box and button exactly as she would use a traditional search engine interface. Thus a traditional search engine could upgrade to a search facility like the one we are describing without confusing its users.

The right-hand side panel is intended for sponsored links.

The left-hand side panel is the main new feature. It shows a list of queries, comprising, first, the original query entered by the user, and below it, the additional queries, if any, described above in Section 2. If there is a respelled query, it is shown immediately after the original query. The entry for each query indicates the type of query (whether it is the original query, the respelled query, a related query, or a subquery) and the cardinality of its result set.

At any point in time one query in the left-hand side list is selected, and the central panel shows the corresponding result set. The user can change the selection, and thus the result set shown in the central panel, by clicking on a different left-hand side query. Once the focus has been brought to the left panel, either by clicking or by tabbing, the keyboard up/down keys can also be used to move the selection around and thus flip through the result sets.

Within each result set, the user can move back and forth using a page menu located at the bottom of the central panel. The current page of each result set is retained in browser memory, and thus *there is absolutely no delay* when switching from one result set to another by clicking on a query in the left panel or using the up/down keys. The absence of delay is particularly striking when the up/down keys are used to flip through the result sets by moving the left-panel selection.

At any time, the user can take any of the queries in the left panel and issue it as the next query. Two mechanisms are provided for doing this: the user can double-click on the desired query, or she can drag it from the left panel to the query box at the top of the page. In either case the effect is the same as if the user had typed the query into the query box and submitted it.

The interactions between our prototype and the Yahoo search engine are as follows. When the user submits a query, the prototype sends three requests to the search engine:

(1) A request for the first 10 results of the query,

(2) A request for a respelling of the query (actually, for any number of respellings, but only one respelled query is ever produced), and

(3) A request for up to 7 related queries.

The requests are sent in parallel, and the responses to the requests may come back in any order, with event listeners dispatched as the responses arrive.

As soon as the results of the user's original query arrive, they are displayed to the user in the central panel.

The respelling request may or may not return a result. If it does, the result is a respelled query, and we send a subsequent request to the Yahoo search engine for the first 10 results of this query. We could do this as soon as the respelling result arrives, but we prefer to wait until the results of the original query have arrived and been shown to the user, if this has not yet happened.

Similarly, after the result of the request for related queries has arrived, but not before the results of the original query have been displayed to the user, we send requests to the search engine, in parallel, for the

first 10 results of each of those queries, if any. If there are no related queries, requests for subquery results are sent instead.

At this point, the results of the original query have been displayed and the results of the additional queries have been requested. While the user inspects the results of the original query, those of the additional queries arrive within a few seconds. As the results of each additional query arrive, an entry for the query is created and inserted at an appropriate position in the left-hand side list. The user can immediately click on that entry to see the results of the query in the central panel; there is no need to wait for the results of all the other additional queries to have arrived.

It is useful to compare the additional queries shown in the resizable left panel to the Cuil tabs [4] that narrow the user's original query. Both share an advantage over the links to related queries shown by Live Search, Google and Yahoo, viz. they allow the user to switch from one related query to another with a single click. But there are important differences:

- All the additional queries in the left panel are issued simultaneously and automatically, so that their results are ready when the user wants them. A tab query, on the other hand, is issued when the user clicks on the tab, and the user has to wait for the result.
- The result sets of all the additional queries (more precisely, the current page of each result set) are kept in browser memory, and the user switches from one to the other with no delay. The result sets of the tab queries, on the other had, are not retained in the browser. When the user returns to a tab, the result set has to be fetched again from the Web server. The user has to wait for the first page of the result set to arrive, and then has to navigate back to the page the she was visiting earlier.
- Showing related queries as tab labels severely limits the number of related queries that can be shown simultaneously.
- Whereas the additional queries in the left panel can be respellings, related queries of any kind, or subqueries, tab queries are always more specific than the original query, and thus can provide no help when the original query misses its target.

4. The zero-results case

It is rare for a query that targets the Web at large to have no results, due to the large number of pages indexed by search engines and the presence of pages whose content is randomly generated verbiage. But a search can be restricted to a particular site by using a search term with the prefix *site:*, and such a search can as easily have results as not have them. Zero results also occur frequently in local search queries (not implemented in the prototype), and in queries run against search facilities of bounded scope, such as an intranet search engine.

When a query has zero results, the set of subqueries having one fewer term than the original query is suboptimal. The optimal set of subqueries in this case coincides with the set of subqueries that would be involved in providing a cooperative response [3] to the zero-response query. It consists of the most specific subqueries (subqueries with the most terms, a.k.a. maximal subqueries) that produce results and the most general subqueries (subqueries with the fewest terms, a.k.a. minimal subqueries) that do not produce results. The latter identify, in a sense, the cause or causes of failure, while the former identify the minimal relaxations of the query contraints that produce results.

The cited work [3] on cooperative responses used a simple sequential algorithm to find the minimal zero-result subqueries and the maximal non-zero-result subqueries. Our prototype uses instead a parallel algorithm that takes advantage of the inherent parallelism available in the search engine API.

The algorithm (meta-)searches the graph of subqueries of the original query (including the original query itself, which may be considered a subquery of itself). Starting from the original query, which has zero results, the algorithm traverses the entire subgraph of zero-result subqueries. When it finds a zero-result subquery, the algorithm looks for additional ones by launching in parallel the subqueries that have one fewer term. It does this while other subqueries may still be running. But it never runs a subquery before it has determined that all subqueries with one more term have zero results. This saves unnecessary work and also guarantees that the non-zero-result subqueries that are encountered in the process (as subqueries with one fewer term than some zero-result subquery) are precisely the ones with the most terms; they can thus be immediately inserted at their proper position in the left-hand side panel. When a zero-result subquery is encountered, on the other hand, it is not immediately placed in the left panel; it is inserted instead in a list of zero-result subqueries, and any subqueries currently in that list whose terms are a superset of those of the new query are removed from the list. At the end of the process the list contains the zero-result subqueries with the fewest terms, and the whole zero-result list is appended to the list of non-zero-result subqueries in the left-hand side panel. (The process ends when there are no pending requests to the search engine.)

5. Further work: general Boolean queries

So far we have only been concerned with purely conjunctive queries. These are the only queries currently handled by our prototype. In this section we explain how to handle general Boolean queries, i.e. queries that combine conjunction, disjunction and negation, assuming that an API is available for a search engine that can process such queries. For the sake of brevity we shall use algebraic notation, with + denoting disjunction, juxtaposition denoting conjunction, ¬ denoting negation, and letters *A*, *B*, *C*, ... denoting search terms or Boolean expressions. It will be convenient to think of disjunction and conjunction as n-ary operators, e.g. to think of *A*+*B*+*C* as a ternary operator with three operands *A*, *B* and *C*.

When the user's original query is a Boolean query, there is no problem in using a respelled query and any number of related queries, as described above, as additional queries. The question that arises is what to use instead of the conjunctive subqueries when no related queries are available. In this section we specify queries that are syntactically derived from the original query and can be used in that case.

5.1 Purely disjunctive queries

Consider first the case of a purely disjunctive query, such as *A*+*B*+*C* where *A*, *B* and *C* are atomic search terms, i.e. single words or quoted phrases. At first glance, it seems that no syntactically derived quey could add any information to the result set produced by *A*+*B*+*C*, since this result set includes all the results of *A*, *B*, and *C*. A closer look, however, shows that this is not so. All desired results may be in the set, but the interesting ones may be buried below millions of uninteresting ones, and thus practically invisible. (Actually, it seems that most search engines don't let users retrieve more than the first 1000 results of a query, so any result ranked beyond 1000 is completely invisible.)

If, for example, *C* is a very common term, results containing *A* or *B* may be crowded out of the visible portion of the result set. So queries that omit *C* are definitely of interest as additional queries. But which queries that omit *C* should be included? *A*, *B*, *A*+*B*, *AB*? The conjunction *AB* seems irrelevant, because the user has not expressed interest in the joint presence of A and B in the same document. The disjunction *A*+*B* has the same problem discussed above for *A*+*B*+*C*: results with one of the terms *A* or *B* could crowd out results with the other term from the result set. And all results in the result set of *A*+*B* are also in the result set of *A* or in the result set of *B*. So it seems that the additional queries should be *A* and *B*.

Again, however, a closer look shows that this it not the solution. Even though all the results of *A*+*B* are also results of *A* or results of *B*, the result set of *A*+*B*, as presented to the user, carries important additional information, namely the relative ranking of the results of *A* and *B* with respect to each other. Thus, as additional queries that omit *C*, it would seem best to include *A*, *B* and *A*+*B*. The complete set of syntactically derived additional queries would then be *A*, *B*, *C*, *A*+*B*, *A*+*C*, and *B*+*C*. These are the purely disjunctive queries whose set of terms are the subsets of the set of terms of *A*+*B*+*C*. We refer to them as the *disjunctive subqueries* of *A*+*B*+*C*.

But now we face a familiar problem: the number of disjunctive queries is exponential in the number *N* of terms of the original query. As in the case of a conjunctive query, the simplest solution to this problem is to include only the subqueries with *N-1* disjuncts. The user can then double click on one of them, or drag one of them to the query box, to run it as the next query and thus obtain result sets of subqueries with *N-2* disjuncts; and so on. In the example, the subqueries with *N-1=2* disjuncts are the disjunctive subqueries *A*+*B*, *A*+*C*, and *B*+*C*.

5.2 AND/OR queries

We consider now the case of a query that combines conjunction and disjunction, such as AB(C+D+E), or *A*+*B*+(*CDE*), or *A*(*B*+(*C*(*D*+*E*))), or *A*+(*B*(*C*+(*DE*))). We propose the following rules for handling such queries:

(1) It the top-level operator is conjunction, handle the query as if it was a purely conjunctive query, with nested disjunctions treated as atoms. For example, treat *AB*(*C*+*D*+*E*) as *ABX*, with *X*=*C*+*D*+*E* treated as atomic.

(2) Symmetrically, if the top-level operator is disjunction, handle the query as if it was a purely disjunctive query, with nested conjunctions treated as atomic. Thus, treat *A*+*B*+(*CDE*) as *A*+*B*+*X*, with *X*=*CDE* treated as atomic.

(3) Provide the user with a mechanism, such as clicking on a button or a link, for transforming the query by distributing conjunction over disjunction, e.g.:

AB(C+*D*+*E*) -> *ABC*+*ABD*+*ABE*

or:

A+(*B*(*C*+(*DE*))) -> *A*+*BC*+*BDE*

When the transformation is applied, replace the original query in the query box with the transformed query, generate a new set of syntactically-derived additional queries and make their results available to the user.

The transformation must distribute a top-level conjunction or a conjunction nested under a top-level disjunction, but not a conjunction nested under a disjunction which is itself nested under a conjunction:

A(*B*+(*C*(*D*+*E*))) -> *AB*+*AC*(*D*+*E*)

The user will be able to apply the transformation repeatedly to reach disjunctive normal form:

A(*B*+(*C*(*D*+*E*))) -> *AB*+*AC*(*D*+*E*) -> *AB*+*ACD*+*ACE*

The motivation for not going to disjunctive normal form at once is that the intermediate steps may be of more interest to the user than the normal form.

(4) Also provide a symmetric mechanism for distributing disjunction over conjunction, which lets the user reach conjunctive normal form in some number of steps.

5.3 Negation

We propose to handle negation by using De Morgan's laws to push the negation operator to the bottom of the Boolean parse tree, so that it only applies to atomic terms, e.g.:

$A\neg(B+(C(D+E)) \rightarrow A\neg B\neg(C(D+E)) \rightarrow A\neg B(\neg C+\neg(D+E)) \rightarrow A\neg B(\neg C+(\neg D\neg E))$

We can then proceed as above, treating negated terms as atoms.

Note that some queries with negation are “unreasonable” and should be rejected. They can be characterized by the fact that, when put in disjunctive normal form, they have one or more disjuncts where all the terms are negated. They are also precisely those queries that cannot be expressed using the set of operators AND, OR, ANDNOT rather than the set AND, OR, NOT. Providing the user with the former set of operators rather than the latter is one way of avoiding such unreasonable queries.

5.4 General Boolean queries with zero results

The case where a Boolean query produces a zero result requires again special treatment.

If a Boolean query has a zero result and its top operator is conjunction (after pushing negation to the bottom as described in the previous section), it can be handled as described in Section 4, with nested disjunctions and negations treated as atoms.

If, on the other hand, the top operator is disjunction (again after pushing negation to the bottom), then it can be handled by the following modification of the method of Section 4.

The query can be viewed as a disjunction of conjunctions, each conjunction having one or more conjuncts. These conjuncts may or may not be atomic, but they are treated as atomic. The same conjunct may appear in more than one conjunction, and this is duly recognized. Consider the set of all the conjuncts appearing in one or more of the disjuncts, and the graph of all the subsets of this set. Each node in the graph represents a query. In particular, there is a node in the graph representing each of the disjuncts of the user's query; we refer to these nodes as the *maximal nodes of interest*. We refer to a node that has fewer conjuncts than one of the maximal nodes of interest as a *node of interest*.

Nodes that are not of interest are not directly relevant to the user's query and should not be used as additional queries. For example, given the query *AB+CD*, the maximal nodes of interest are those representing *AB* and *CD*, and the other nodes of interest are those representing *A*, *B*, *C* and *D*. Nodes such as the one representing *AC* are not of interest. *AC* is not directly relevant to the query, because the user has asked for pages that contain *A* and *B*, or pages that contain *C* and *D*, but has expressed no particular interest in pages that contain *A* and *C*.

The maximal nodes of interest represent zero-result queries. To obtain the syntactically-derived additional queries we propose to use the algorithm of Section 4, but starting with the maximal nodes of interest rather then with the top node of the graph, which is not of interest. Thus, the algorithm starts by launching in parallel the queries corresponding to the children of the maximal nodes of interest (i.e. those with one fewer conjunct than a maximal node of interest). Proceeding as in Section 4, the algorithm discovers all zero-result nodes of interest, and the maximal (as sets of conjuncts) non-zero-result nodes of interest. It inserts the queries represented by the latter into the left panel as it encounters them. Then, when done, it adds the queries represented by the minimal (as sets of conjuncts) zero-result nodes of interest.

The syntactically-derived queries produced by this algorithm are the same whose result-set cardinalities were used in the cited work [3] to provide a cooperative response to a general Boolean query with zero results, except that this algorithm omits the conversion to disjunctive normal form that took place in

[3]. It is up to the user to convert the query to disjunctive normal if desired, or to apply one or more times the above-mentioned transform that distributes conjunction over disjunction, without necessarily reaching normal form.

6. Additional remarks

6.1 Multiple respelled queries

When multiple words seem to be misspelled in the original query, or multiple respellings are possible for the same word, it might be desirable to give the user a choice of respelled queries among the additional queries. Currently search engines propose only one respelled query that includes all the identified misspellings. The Yahoo API could in principle return multiple respelled queries, but only produces one in actuality.

6.2 Related queries vs. subqueries

Our prototype uses subqueries as additional queries only if no related queries are available. However, it might be desirable to offer subqueries even when related queries are available. Both kinds of related queries could be listed together in the left panel. Alternatively, the two kinds of additional queries could be kept separate by placing a Flex TabNavigator component with two tabs inside the left panel. The tabs would be labeled *Related Queries* and *Subqueries*, and the user could click on the desired tab to see the corresponding set of additional queries. The respelled query, if available, could be included in either tab, or it could be taken out of the left panel and shown by itself below the query box.

6.3 Avoiding rerun of follow-up query

In our prototype, when the user double-clicks on an additional query or drags it to the query box, the query is rerun. This is unnecessary, since the result set of the query is available and is unlikely to have changed. It is done in our prototype mostly as a programming shortcut, and it might be better to avoid it.

6.4 Moving functionality to the server side

In our prototype the invention is implemented on the client-side, with no modification of the server-side search engine code. But this is just a matter of expediency, due to the fact that we do not have access at this time to the server-side internals of a search engine.

In the future, we hope to be able to collaborate with search engine implementors to move some of the new functionality to the server side. Once the server receives the user's query, it has all the information it needs to determine the set of additional queries. Therefore it can send the list of additional queries and the first 10 results of each additional query to the client without waiting for any additional input from the client. The client is still responsible for showing the list of queries and their results to the user, and downloading additional pages of results for each query as the user requests them.

Furthermore, a query and its subqueries have overlapping sets of terms. When a group of queries have overlapping sets of terms, it is possible to generate their result sets together in one pass through the index, and this is more efficient than computing their result sets separately. A server-side implementation could adapt the algorithms described above to take advantage of this observation and make more efficient use of server resources.

7. References

[1] Adobe Flex 3, http://www.adobe.com/products/flex/.

[2] Yahoo! Astra Web APIs, http://developer.yahoo.com/flash/astra-webapis/.

[3] Yahoo Search Assist Help Page, http://help.yahoo.com/l/us/yahoo/search/basics/basics-27.html.

[4] Cuil Features, http://www.cuil.com/info/features/.

[5] F. Corella, S. J. Kaplan, G. Wiederhold and L. Yesil, Cooperative Responses to Boolean Queries, in Proceedings of the First International Conference on Data Engineering, p. 77-85, ISBN:0-8186-0533-2.