A parallel algorithm for computing cooperative responses through a Web API

Francisco Corella
Pomcor
fcorella@pomcor.com

Karen P. Lewison
Pomcor
kplewison@pomcor.com

ABSTRACT

Hundreds of new search engines have recently appeared on the Web. Many of them consist of a search front-end that accesses one or more search back-ends through a Web API. Typical API latencies range from one tenth of a second to one or two seconds. Such latencies are acceptable for running a single query, but make it impractical to compute a cooperative response, which requires running many subqueries, using the sequential algorithms that have been proposed so in the literature for that purpose. The front-end, however, can compute a cooperative response by exploiting the inherent parallelism of a Web API. We describe a parallel algorithm that computes cooperative responses and prove its soundness, completeness and non-redundancy. We provide performance results obtained from an implementation of the algorithm in Noflail Search, a search front-end available at noflail.com.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; H.3.5 [**Information Storage and Retrieval**]: On-line Information Services—*Web-based services*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms

Algorithms, Performance

Keywords

Cooperative responses, cooperative answering, query relaxation, Web search, search engine, parallel algorithms.

1. INTRODUCTION

A *cooperative response* to a query is an indirect response that is more helpful to the user than a direct response would be. Interest in cooperative responses arose in the context of natural-language query answering, but carried over to formal-language query answering [14, 13, 11, 3, 15]. In a natural language setting, cooperative responses may correct false presuppositions, anticipate follow-up queries and provide information not explicitly requested by the user. In a formal language setting, they may fill the void created by the absence of results for a query, suggesting follow-up queries that would produce results, and explaining the failure to produce results by listing queries that are more general than the direct query but also fail to produce results.

Much work on cooperative answering has been done over the last thirty years. Researchers have investigated the use of various forms of domain knowledge to produce cooperative responses, including integrity constraints [9] and taxonomies [7] in a deductive database setting, integrity constraints plus completeness assertions [16], type-abstraction hierarchies [2], and a variety of other meta-data [12]. They have used methods ranging from heuristics [9] to explicit user preferences [5] to on-the-fly machine learning [18] in order to choose the most appropriate information to include in a response. They have constructed intensional [17] responses, qualified responses [8], and responses to Web search queries that point to relevant know-how through hyperlinks [1]. They have also studied the computational complexity of computing certain cooperative responses [10]. More recently, query relaxation has been studied in the context of searching XML data [4]. An overview of early work can be found in [6].

Much further work remains to be done, however, and recent developments in the field search and information storage and retrieval provide great opportunities for such work. We are witnessing today a multi-faceted transformation of the field. Aspects of this transformation include the development of large-scale repositories of information and knowledge, both structured and unstructured (Wikipedia, UMLS, the "commercial ontology" announced by Hakia, etc.); fast-paced innovation in the area of user interfaces for search; the birth of a search ecosystem where independently developed search components can be integrated through Web APIs; and the emergence of hundreds of new search engines. This calls for further work on semantics-based cooperative answering and on cooperative search interfaces; and such work is facilitated by the availability of Web APIs in the search ecosystem.

As a step towards bringing cooperative answering into the new search ecosystem we have implemented cooperative responses in Noflail Search, a search front-end available at

noflail.com that obtains search results from the Bing™API[1]. Few queries on the Web have zero results, because there are many pages with large lists of words, so one may question the usefulness of cooperative responses for Web searches. They are very useful, however, when search is restricted to a particular Web site or a particular geographic location.

Noflail Search has a user-interface feature designed to reduce the time and effort it takes to solve a difficult search problem. The user can collect queries in a left panel and browse the result sets of muliple queries at once. This makes it easy to conduct a manual breadth-first search of those result sets, something hard to do in traditional search engines. The cooperative answering feature is integrated with this user-interface feature. When a query fails to produce results, Noflail Search generates a list of subqueries (queries with fewer terms) that it inserts in the left panel. The list comprises the most specific subqueries that produce results, i.e. those whose set of terms is maximal, and the most general subqueries that fail to produce results, i.e. those whose set of terms is minimal. This list of subqueries is the cooperative response to the query. The user can immediately browse the result sets of the subqueries with results, which are prefetched during the computation of the cooperative response.

Although we hope to use semantics in the future to improve the generation of cooperative responses, we do not do so at this time. In fact, there is nothing new in the list of subqueries that we provide as a cooperative response. It is actually the same list that was provided in [3]. But we have successfully addressed a new problem, one that arises from the fact that Noflail Search is a search front-end, which takes advantage of the new search ecosystem by obtaining search results from a back-end through a Web API.

It typically takes a few tens of a second to obtain a response from a Web search API. Such a latency is acceptable for providing a direct response to a query, but makes it impractical to compute a cooperative response to a failing query using the sequential algorithms that have been proposed so far in the literature for that purpose.

A search front-end, however, can compute a cooperative response by exploiting the inherent parallelism of a Web API. Multiple subqueries can be submitted simultenously to the API, which processes them in parallel, independently of each other, as if they had been submitted by independent front-ends. We have developed a parallel algorithm that takes advantage of this. In Section 2 we give a description the algorithm, for conjunctive queries, in the form of pseudo-code for event listeners. In Section 3 we prove that the algorithm is sound and complete, as well as non-redundant in the sense of [10]. In Section 4 we discuss how the algorithm could be extended to handle Boolean queries. In Section 5 we provide performance results and discuss the complexity of the algorithm. In Section 6 we recapitulate and suggest directions for future work.

2. DESCRIPTION OF THE ALGORITHM

[1]Bing is a trademark of Microsoft Corporation. We are not affiliated with Microsoft.

There are two methods for implementing a search front-end: the traditional, server-based method, and a more modern, client-based method.

In the server-based method, the user's browser, running on the client machine, submits a query to the Web site of the search front-end, which includes a server farm dedicated to processing queries. The front-end site relays the query to the back-end site, an independent Web site, requesting the first page of results. When the response arrives from the back-end site, the front-end site creates an HTML file containing the results and returns it to the browser.

In the client-based method, the search front-end runs on the client machine. It may run within the user's browser, if implemented in Javascript, Flash, Flex (an alternative programming platform for Flash Player) or Silverlight. It may also run under the client machine OS as an independent application separate from the browser. Noflail Search is a client-based search front-end that runs within the browser, implemented in Flex. The client-based front-end submits queries directly to the back-end site through the Web API, obtains results from the back-end, and presents the results to the user. There is a front-end site, but its only role is to download the front-end code to the user's machine.[2]

The parallel algorithm shows the subqueries that make up a cooperative response to the user as it finds them (at least those with results). This is easiest to implement in a client-based search front-end, with the algorithm implemented in the client. The same functionality, however, could be achieved in a server-based search front-end with the algorithm implemented in the server, using frames, or polling from the client; we leave the details to the reader.

In this section we describe the algorithm as it applies to purely conjunctive queries, i.e. queries consisting of a conjunction of search terms. Boolean queries are discussed in Section 4.

The algorithm is event-driven. An *event handler*, or *event listener*, is invoked whenever a response to a query or subquery arrives from the back-end, and does the following:[3]

1. If the response corresponds to a query submitted by the user, the event listener checks the cardinality of the result set. If it is positive, it shows the first page of results to the user. If it is zero, it lets the user know that there are no results, and it calls the routine *processRoot* of Figure 1.

2. If the response corresponds to a subquery, the event listener calls the routine *processNode* of Figure 2 and passes to it two parameters: the subquery, which is assigned to the parameter `curNode`, and the cardinality of the result set of the subquery, which is assigned to the parameter `cardinality`. The value of `curNode` during an invocation of `processNode` will be called the *current subquery* or *current node* of the invocation.

[2]In the case of Noflail Search, search ads are relayed through the front-end site.

[3]The same or different handlers may be used for queries and subqueries.

There are no threading or parallel-processing constructs in `processRoot` and `processNode`. The parallelism of the algorithm takes place in the back-end, not the front-end. The back-end runs multiple subqueries simultaneously, but in the front-end different invocations of the subroutines are not expected to overlap with each other. In fact, there must not overlap with each other. No special precautions need to be taken to ensure that they do not overlap if the front-end is implemented in JavaScript or ActionScript, which are single-threaded; but mutual exclusion may have to be enforced in other cases.

The user's query and its subqueries form a graph $(G, \leq)$ isomorphic to the powerset of the set of search terms of the user's query minus the empty set. We shall refer to query and subqueries as *nodes* of the graph, and to the user's query as the *root* of the graph, which we call r. The *parents* of a node n are the successors of n for the ordering relation, i.e. the minimal nodes greater than n, i.e. the subqueries obtained by adding one more search term to the search terms of n. The *children* of n are the predecessors of n, i.e. the maximal nodes less than n, i.e. the subqueries obtained by removing one search term from those of n. Nodes can be represented in programming languages as bitmaps of size N, where N is the number of search terms of the user's query.

A query that fails to produce results (whether the user's query or a subquery) is called a *failing query*, a query that produces results, a *succeeding query*. We call G' the subgraph of G consisting of those nodes all of whose parents are failing queries. G' comprises the failing queries and the maximal succeeding queries. We shall see that the algorithm discovers and traverses G'. It does not construct or traverse the entire graph G.

Figures 1 and 2 describe the routines *processRoot* and *processNode* using pseudo-code. In the pseudo-code we use shorthands to refer to functionality that is implemented by bitmap manipulation if nodes are represented by bitmaps: `>>` tests that the left operand is strictly greater than the right operand in the graph ordering; the `foreach...` constructs are loops over bitmaps of length N generated by turning on (to generate parents) or off (to generate children) one bit of a given bitmap of length N; and `root` is a global variable whose value is a bitmap of length N all of whose bits have value 1, representing the root r of G.

The routine `processRoot` is straightforward. It initializes a hash table[4] `failed` and an array `minFailures`. The hash table `failed` is used to remember which subqueries fail to produce results. Undefined hash-table values are equivalent to false. The array `minFailures` collects failing subqueries, but subqueries are removed from the array when found not to be minimal. At line 105, the routine initializes `pending` to 0. The variable `pending` is an up-and-down counter that keeps track of the number of subqueries being processed by the back-end. Finally, the loop at lines 106–109 launches the subqueries with one fewer term than the user's query, and increments `pending` accordingly.

The routine `processNode` decrements `pending` at line 202.

[4]In ActionScript we implement a hash table as an Object.

```
101    procedure processRoot() {
102        failed = new HashTable();
103        failed[root] = true;
104        minFailures = new Array();
105        pending = 0;
106        foreach (node in children(root)) {
107            launch(node);
108            pending++;
109        }
110    }
```

Figure 1: Pseudo-code for *processRoot*

Then it checks if the current subquery has results. If so, as shown in section 3, it must be a maximal succeeding subquery. The current subquery, with the cardinality of its result set, is reported to the user as part of the cooperative response at line 204 by calling `showSuccess`. (In Noflail Search the subquery is inserted in the left panel with the cardinality of its result set.) If the current subquery does not have results, this fact is recorded at line 207, then the current subquery is added to the end of the array `minFailures` at line 213 after removing from the array any subqueries greater than the current subquery.[5]. Then the children of the current subquery are generated in the loop at lines 214–227. Those whose parents have all failed are launched, and `pending` is incremented accorgindly. Finally, at line 228 we use `pending` to check if the computation has ended. If so, the loop at lines 229–231 reports the elements of `minFailures` to the user as part of the cooperative response. (In Noflail Search this is done by inserting them in the left panel after the maximal succeeding subqueries, each with its zero result-set cardinality.) An indication that the cooperative response computation has ended is then provided to the user at line 232.

3. PROPERTIES OF THE ALGORITHM

In this section we prove that the algorithm is *sound* and *complete*. As in [10], we also prove a few *non-redundancy* properties.

Soundness has two parts: (I) every subquery reported to the user by a call to `showSuccess` is a maximal succeeding subquery; and (II) every subquery reported to the user by a call to `showFailure` is a minimal failing subquery

Completeness also has two parts: (I) every maximal succeeding subquery is reported to the user by a call to `showSuccess`; and (II) every minimal failing subquery is reported to the user by a call to `showFailure`.

We prove two non-redundancy properties concerning the user interface: (I) no succeeding subquery is reported twice by `showSuccess`; and (II) no failing subquery is shown twice by `showFailure`. We show as well three non-redundancy properties concerning the back-end interface: (III) no subquery is launched twice; (IV) no query is launched after a lesser failing subquery has been launched; and (V) no succeeding subquery is launched after a greater succeeding sub-

[5]In ActionScript, `remove(minFailures,k)` is implemented as `minFailures.splice`(k,1)

```
201  procedure processNode(curNode, cardinality) {
202    pending--;
203    if (cardinality > 0) {
204      showSuccess(curNode, cardinality);
205    }
206    else {
207      failed[curNode] = true;
208      for (k = minFailures.length; k >= 0; k--) {
209        if (minFailures[k] >> curNode) {
210          remove(minFailures,k);
211        }
212      }
213      push(minFailures,curNode);
214      foreach (node in children(curNode)) {
215        allParentsFailed = true;
216        foreach (parentNode in parents(node)) {
217          if (!failed[parentNode]) {
218            allParentsFailed = false;
219            break;
220          }
221        }
222        if (allParentsFailed) {
223          launch(node);
224          pending++;
225        }
226      }
227    }
228    if (pending == 0) {
229      for (k = 0; k < minFailures.length; k++) {
230        showFailure(minFailures[k]);
231      }
232      showDone();
233    }
234  }
```

Figure 2: Pseudo-code for *processNode*

query has been launched.

Notice that the fact that the algorithm is non-redundant does not mean that it is optimal. One possible improvement, in particular, comes to mind. When searching the Web at large, rather than a particular site or location, it is rare to find combinations of ordinary words that produce zero results. Most failures to find results are due to a phrase that cannot be found or a non-existent word, i.e. to failing search terms. It may therefore be advantageous to check the leaves of the graph first, remove any that fail, and they run the algorithm starting from the root of the remaining graph.

A run of the algorithm consists of an invocation of the routine `processRoot` followed by a number of invocations of the routine `processNode`. We number these invocations starting from 0: invocation number 0 is the invocation of `processRoot`, invocations 1, 2, ... are invocations of `processNode`. We call I the set of invocation numbers; I is finite, but this needs to be proved. Let $I^* = I \setminus \{0\}$.

For every $i \in I^*$ let c_i be the current subquery during invocation number i,[6] i.e. the value of `curNode` during the invocation; let c_0 be r. For every $i \in I$ let L_i be the set of subqueries launched by invocation number i; for $i = 0$, the subqueries are launched at line 107 of `processRoot`; for $i > 0$ they are launched at line 223 of `processNode`. For every $i \in I$ let p_i be the value of `pending` at the end of invocation number i.

We use the word *launch* to refer to an execution of line 107 by an invocation of `processRoot` or an execution of line 223 by an invocation of `processNode`. We say that the back-end exhibits *liveness* if every launch has a response, which in turn causes an invocation of `processNode`, i.e. if

$$(\forall i \in I)(\forall n \in L_i)(\exists j \in I, j > i)(n = c_j). \qquad (1)$$

We do not assume liveness, but we use it as a hypothesis of the completeness results.

We do have to assume that there are no extraneous invocations of `processNode`, which implies

ASSUMPTION 1. *For every $i \in I^*$ there exists $j < i$ such that $c_i \in L_j$.*

and

ASSUMPTION 2. *For distinct $i, i' \in I^*$ there exist distinct launches l and l' of the subqueries c_i and $c_{i'}$ respectively, such that l takes place during an invocation $j < i$ and l' takes place during an invocation $j' < i'$.*

We prove first a few lemmas.

LEMMA 1. *For every $i \in I$, $L_i \subseteq G'$.*

PROOF. Let $n \in L_i$. If $i = 0$, n is a subquery launched by `processRoot` at line 107; n is the value of `node`; therefore by the loop condition at line 106, it is a child of r. Thus n has a single parent that is a failing subquery, and $n \in G'$. If $i > 0$, then n is a subquery launched by `processRoot` at line 223; n is the value of `node`, and line 223 is executed only if the loop at lines 216–221 has verified that all parents of n have failed according to the hash table `failed`; and it is clear that the hash table is only assigned the value true for a query or subquery if it has indeed failed. Therefore $n \in G'$. □

LEMMA 2. *For every $i \in I$, $c_i \in G'$.*

PROOF. If $i = 0$, $c_0 = r \in G'$. If $i > 0$, $c_i \in G'$ follows immediately from assumption 1 and lemma 1. □

LEMMA 3. *If the back-end exhibits liveness, for every $n \in G'$ there exists $i \in I$ such that $n = c_i$.*

[6]Recall that, when i is positive, invocation number i is an invocation of `processNode`.

PROOF. Assume that the back-end exhibits liveness, i.e. that the liveness condition (1) holds.

Then reason by contradiction: let S be the set of nodes $n \in G'$ for which there is no $i \in I$ such that $n = c_i$, and assume that $S \neq \emptyset$.

Let m be a maximal element of S. If $m = r$, then $m = c_0$, a contradiction. Otherwise, let P be the set of parents of m. Either m is a child of r and $P = \{r\}$ or $r \notin P$; we consider in turn these two cases.

Assume m is a child of r. When `proccessRoot` is invoked, the loop at lines 106–109 launches all the children of the root. Therefore $m \in L_0$ and, by (1), there exists $i \in I$ such that $m = c_i$. Hence $m \notin S$, a contradiction.

Now assume $r \notin P$. Since $m \in G'$, every element of P is a failing query. Since m is a maximal element of S, the elements of P are not in S; hence every element of P is of the form c_j for some $j \in I$. Let $J = \{j \in I \mid c_j \in P\}$.

Consider an invocation j for $j \in J$, which, since $r \notin P$, is an invocation of `processNode`. Since $c_j \in P$ is a failing query, the value of `cardinality` is 0. Therefore the condition of line 203 fails and branch 207–226 of the conditional is taken. In particular, line 207 is executed, assigning true (the value of the constant `true`) to the key c_j of the hash table `failed`; and after that value is assigned, it is never removed.

Now consider more particularly the last of those invocations, i.e. invocation k where k is the greatest element of J. Within branch 207–226, the loop 214–226 is executed, with one iteration of the loop for each child of c_k. Thus in one iteration of the loop the value of `node` is m. In that iteration, the nested loop 216–221 checks the value `failed(parentNode)` when `parentNode` takes the value c_j for each $j \in J$, and finds it to be true in call cases, inclucing the case $j = k$ because line 207 has already been executed. Therefore line 223 is executed and m is launched. Thus $m \in L_k$ and, by the liveness condition (1), there exists $i \in I$ such that $m = c_i$. Hence $m \notin S$, a contradiction. □

The next two lemmas are used in proofs by contradiction, hence their peculiar hypotheses.

LEMMA 4. *If i and i' are distinct elements of I^* such that $c_i = c_{i'} = n$, there exist distinct elements j and j' of I such that $j < i$, $j' < i'$, and $n \in (L_j \cap L_{j'})$.*

PROOF. By assumption 2 and the observation that the same subquery is never launched twice within a given invocation. □

LEMMA 5. *If j and j' are elements of I such that $L_j \cap L_{j'} \neq \emptyset$, with $j < j'$, there exists $k \in I$, $k < j$, such that $c_k = c_{j'}$.*

PROOF. Let $j, j' \in I$ with $j < j'$ and let $n \in L_j \cap L_{j'}$. Since n is launched by invocations j and j', n is a child of both c_j and $c_{j'}$. We cannot have $j = 0$ because n would then be a child of r and we would have $c_j = c_{j'} = r$ and $j = j' = 0$. Consider invocation j, an invocation of `processNode`. It launches n at line 223 during an execution of the loop 214–227. Therefore, at loop 216–221, it must have found the hash table `failed` to have the value true for every parent of n, including $c_{j'}$. There must have been an invocation k, with $k < j$ that has set `failed` to true for $c_{j'}$, and an invocation of `processNode` can only assign true to `failed` for the current query. Hence $c_k = c_{j'}$. □

LEMMA 6. *For all i, i' in I, if $c_i = c_{i'}$ then $i = i'$.*

PROOF. By contradiction. Let

$$S = \{h \in I \mid (\exists h' \in I, h' > h)(c_h = c_{h'})\}$$

and assume that $S \neq \emptyset$. Let i the smallest element of S, and let $i' \in I$, $i' > i$ be such that $c_i = c_{i'}$. We must have $0 < i < i'$ for otherwise $c_i = c_{i'} = r$ and $i = i' = 0$. By lemma 4, there exist distinct $g, g' \in I$ such that $g < i$, $g' < i'$ and $L_g \cap L_{g'} \neq \emptyset$. Let j and j' be the smallest and largest element, respectively, of $\{g, g'\}$. We have $j \leq g < i$. By lemma 5, there exists $k \in I$, $k < j$, such that $c_k = c_{j'}$. We have $k < j < j'$. From $k < j'$ and $c_k = c_{j'}$ it follows that $k \in S$. But $j < i$, a contradiction. □

LEMMA 7. *For all $n \in G$ and $i \in I$, if $n > c_i$ then there exists $j \in I, j < i$ such that $n = c_j$.*

PROOF. By induction on i. Assume that the proposition is true for every $j < i$, and let us prove that it is true for i. If $i = 0$ it is vacuously true because there is no $n \in G$ such that $n > c_0 = r$. Assume that $i > 0$ and let $n \in G$ be such that $n > c_i$. Let P be the set of parents of c_i. Consider invocation i, an invocation of `processNode`. Every $m \in P$ has been found to have failed by the loop 216–221, and therefore has been launched by an earlier invocation, so if $m \in P$ there is some $k < i$ such that $m = c_k$. If $n = m \in P$ the proposition is therefore proved by letting j be k. If $n \notin P$, there must exist $m \in P$ such that $n > m$. Let again $k < i$ be such that $m = c_k$. By induction hypothesis there exists $j < k < i$ such that $n = c_j$ and the proposition is also true in this case. □

LEMMA 8. *For all $i, j \in I$, if $c_i > c_j$ then $i < j$.*

PROOF. Assume $c_i > c_j$. By lemma 7, there exists $k < j$ such that $c_k = c_i$. Then by lemma 6, $i = k < j$. □

LEMMA 9. *For every $i \in I$, $p_i = \left|\bigcup_{j \leq i} L_j \setminus \{c_j\}_{j \leq i}\right|$*

PROOF. By induction. □

LEMMA 10. *For every $i \in I$, if $\{c_j\}_{j \leq i} \neq G'$, then $p_i > 0$.*

PROOF. Let $i \in I$ be such that $\{c_j\}_{j \leq i} \neq G'$. Let $S = G' \setminus \{c_j\}_{j \leq i}$ and let n be a maximal element of S. Let P

be the set of parents of n; $P \subseteq \{c_j\}_{j \leq i}$. Let $J = \{j < i \mid c_j \in P\}$, and let k be greatest element of J. Since $n \in G'$, all the elements of P, and c_k in particular, are failing queries. Consider invocation number k. If $k > 0$, it is an invocation of `processNode`, which finds that all parents of n have failed and therefore launches n. If $k = 0$, it is an invocation of `processRoot`, which launches all the children of $r = c_k$ including n. In either case, $n \in L_k$. Since $k < i$, n is an element of $\bigcup_{j \leq i}$ without being an element of $\{c_j\}_{j \leq i}$. Therefore, by lemma 9, $p_i > 0$. □

We can now prove the desired results.

THEOREM 1. (Soundness, part I) *Every subquery reported to the user by a call to* `showSuccess` *is a maximal succeeding subquery.*

PROOF. Let i be an invocation of `processNode` that executes line 204. Recall that the values of `curNode` and `cardinality`, which do not change during the invocation, are the current subquery c_i and the cardinality of its result set. The subquery reported by `showSuccess` is the value of `curNode`, c_i; and since line 204 is executed, `cardinality` must have a positive value. Hence c_i is a succeeding subquery. By lemma 2, $c_i \in G'$; and by definition of G' all the parents of c_i are failing queries. Therefore c_i is a maximal succeeding subquery. □

THEOREM 2. Completeness, part I *If the back-end exhibits liveness, every maximal succeeding subquery is reported to the user by a call to* `showSuccess`.

PROOF. Assume that the back-end exhibits liveness and let n be a maximal succeeding subquery. By definition of G', $n \in G'$ and by lemma 3, there exists $i \in I$ such that $n = c_i$. Since n succeeds and $c_0 = r$ fails, $i > 0$, and invocation i is an invocation of `processNode`. During invocation i, the value of `cardinality` is positive, hence line 204 is executed. And the subquery reported to the user by `showSuccess` at line 204 is the value of `curNode`, which is $c_i = n$. □

THEOREM 3. (Soundness part II, completeness part II, and non-redundancy part II) *The subqueries reported to the user by calls to* `showFailure` *are the minimal failing subqueries, and each is reported only once.*

PROOF. Consider the function $f : i \in I \mapsto c_i$. By lemmas 2, 3 and 6, f is a bijection from I onto G'. I being therefore finite like G', let k be its greatest element.

All calls to `showFailure` are made at line 230 of `processNode`, within the loop 229–231. This loop runs when `pending` has the value 0 at line 228, and the value of `pending` does not change between line 228 and the end of the invocation. By lemma 10, this can only happen during the very last invocation, invocation k; by lemma 9, `pending` does have the value 0 at the end of invocation k, and the loop is executed during that invocation. The loop reports to the user the subqueries collected in the array `minFailures`. The array `minFailures` is modified by those invocations of `processNode` where the current subquery is a failing subquery. Each of those invocations adds the current subquery and removes any subqueries greater than the current subquery. Therefore every minimal failing subquery is added to the array and never removed; and by lemma 8, every non-minimal failing subquery is removed from the array. Thus at the end of the computation, and hence when the loop is executed during the last invocation, invocation k, the array contains exactly the minimal failing subqueries. By lemma 6 each subquery is added only once to the array, and hence it is reported only once by the loop. □

THEOREM 4. (Non-reducancy, part I) *No succeeding subquery is reported twice by* `showSuccess`.

PROOF. A succeeding subquery n is reported by an invocation of `processNode` only if n is the current subquery during the invocation. But by lemma 6, there is no more than one such invocation. □

THEOREM 5. (Non-redundancy, part III) *No subquery is launched twice.*

PROOF. Subqueries are launched by loop 106–109 of `processRoot` and loop 214–227 of `processNode`. Each loop is executed at most once per invocation, and launches each query at most once. It remains to show that a subquery is not launched by two invocations. Reasoning by contradiction, assume that there exist $j, j' \in I$ such that $L_j \cap L_{j'} \neq \emptyset$, with $j < j'$. Then by lemma 5 there exists $k \in I$, $k < j$, such that $c_k = c_{j'}$. But this contradicts lemma 6. □

THEOREM 6. (Non-redundancy, part IV) *No query is launched after a lesser failing subquery has been launched.*

PROOF. Let $i, j \in I$, $m \in L_i$, $n \in L_j$, $m < n$.[7] We are going to prove that $j < i$.

If $j = 0$, n is a child of $r = c_0 = c_j$. If $j > 0$, n is launched by loop 214–227 of `processNode` during invocation j and is therefore a child of the current subquery during the invocation, c_j. Thus in both cases n is a child of c_j. Similarly, m is a child of c_i.

Since $m < n < c_j$, m is not a child of the root, hence c_i is not the root, and m is launched by loop 214–227 of `processNode` during invocation i. The loop verifies that all the parents of m have failed. From $m < n < c_j$ it follows that c_j is greater than some parent of m, which is recorded as having failed in the `failed` hash table and must thus have been the current subquery c_k of an earlier invocation k, $k < i$. From $c_j > c_k$ it follows by lemma 8 that $j < k$. Hence $j < i$. □

THEOREM 7. (Non-redundancy, part V) *No succeeding subquery is launched after a greater succeeding subquery has been launched.*

[7] m plays the role of "lesser failing subquery", but we do not actually need to assume that it fails.

PROOF. In fact, no subqueries that are less than a succeeding subquery are launched, since loop 106–109 launches children of the root, and loop 214–227 launches subqueries only after verifying that all their parents are failing queries. □

4. BOOLEAN QUERIES

We briefly describe now a possible extension of the algorithm to provide coopoerative responses to Boolean queries. This extension has not been implemented yet in Noflail Search.

Consider first the case where the query is in disjunctive normal form (DNF). If the query fails, a cooperative response should consist of the maximal succeeding subqueries and the minimal failing subqueries of each of the disjuncts of the DNF, as in [3]. The algorithm can be modified to produce these subqueries. The modified algorithm starts by launching in parallel the children of all the DNF disjuncts. Later, a child n of a failing subquery is launched only after all parents of n that are descendents of at least one of the DNF disjuncts have been found to be failing subqueries.

Now consider the case where the query is not in DNF. In [3], the query was simply converted to DNF. This makes sense for bibliographic search, where queries often have few answers, but not necessarily for Web Search, where result sets are huge and result ordering is important.

Consider for example the query $A(B+C)D$, where A, B, C and D are atomic search terms, additive notation is used for disjunction, and multiplicative notation is used for conjunction. Assume that the query fails only because D has no results. If the query is converted to DNF, then two queries are offered as follow-up queries: AB and AC. If the query is not converted, then one query is offered as a follow-up query: $A(B+C)$.

Suggesting AB and AC is not equivalent to suggesting $A(B+C)$. One one hand, if the user is only allowed to see the first 1000 results of a result set, the two result sets of AB, AC carry more information than the single result of $A(B+C)$. On the other hand, inspecting two result sets takes more effort than inspecting one, and the relative ranking of results of AB compared to results of AC is lost when the $A(B+C)$ is multiplied out.

This suggests not converting to DNF, or giving the user control over conversions.

5. PERFORMANCE RESULTS AND COMPLEXITY

Table 2 shows performance results for a sample of queries; the queries themselves are listed in table 1. The last two columns of table 2 show the time taken by the parallel computation and the time that would have been taken by issuing sequentially the same subqueries that are issued in the parallel computation.

As discussed in the introduction, it is difficult to find interesing queries with zero results on the Web at large, but failing queries can easily be found when targetting a particular site. Queries marked "(Site)" in table 1 are restricted to the site myrecipes.com. Queries marked "(Web)" are not restricted. Restriction to a site does not seem to have a major or even a predictable impact on performance. The last six queries illustrate this by comparing restricted to unrestricted performance for three queries that have the same graph of failing subqueries when restricted and not restricted.

1	paella mussels escargots (Site)
2	paella clams mussels sardines (Site)
3	paella mussels clams peas sardines (Site)
4	paella mussels clams peas sardines cauliflower (Site)
5	paella mussels clams peas sardines escargots (Site)
6	paella mussels clams nosuchingredient (Site)
7	paella mussels clams nosuchingredient (Web)
8	paella mussels clams peas nosuchingredient (Site)
9	paella mussels clams peas nosuchingredient (Web)
10	paella mussels clams peas nosuchi1 nosuchi2 (Site)
11	paella mussels clams peas nosuchi1 nosuchi2 (Web)

Table 1: Queries

Query Num.	Num. Terms	Subq. Issued	Max. Parall.	Time (ms)	Seq. Time
1	3	4	3	981	2108
2	4	7	4	819	2516
3	5	15	6	4076	15393
4	6	31	10	4879	26149
5	6	47	14	6824	47431
6	4	8	4	2067	4650
7	4	8	4	3353	4879
8	5	16	5	2340	7178
9	5	16	7	3776	12897
10	6	48	15	7452	51577
11	6	48	14	5578	43984

Table 2: Performance

Table 3 shows the cooperative response for query no. 5, as an example. Table 4 shows a trace of the parallel computation for query no. 3 (the trace for query no. 5 is too long).

paella mussels clams peas	7
mussels sardines	1
peas sardines	1
paella sardines	0
mussels peas sardines	0
clams sardines	0
escargots	0

Table 3: Cooperative response for query no. 5

Godfrey [10] showed that computing the minimal failing subqueries or the maximal succeeding subqueries is intractable, because the worst-case number of either kind of subqueries grows exponentially with the number N of search terms. The run-time of the parallel algorithm is exponential for that same reason.

The performance benefit of the algorithm is that it harnesses the parallelism of the Web API and the power of the backend in order to reduce the latency seen by the user (the UI latency), and thus makes it practical to provide cooperative responses.

This benefit can be quantified in terms of complexity, in a

Time	Event	Bitmap	Latency	Card.
0	Launch	01111		
0	Launch	10111		
1	Launch	11011		
1	Launch	11101		
1	Launch	11110		
627	Response	01111	627	0
647	Response	11011	646	0
665	Launch	01011		
675	Response	11110	674	7
749	Response	11101	748	0
765	Launch	11001		
765	Launch	01101		
2770	Response	10111	2769	0
2791	Launch	10101		
2791	Launch	10011		
2791	Launch	00111		
2846	Response	01101	2081	0
2872	Response	01011	2207	0
2894	Response	11001	2129	0
2914	Launch	01001		
3244	Response	10101	453	0
3315	Response	10011	524	0
3330	Launch	10001		
3410	Response	01001	495	1
3502	Response	00111	711	0
3515	Launch	00101		
3515	Launch	00011		
3746	Response	10001	416	0
3891	Response	00101	376	0
4052	Response	00011	537	1

Table 4: Computation trace for query no. 3

somewhat unorthodox way, by observing that the fraction of the UI latency attributable to front-end computation is very small. If we deem it negligible, then the UI latency is worst-case linear in N, because the depth of the graph is linear in N, and the graph is traversed in parallel.[8]

While the parallel algorithm keeps the UI latency low, it does not reduce the consumption of back-end resources, and this may require setting an upper limit on the number N of search terms for which cooperative responses can be provided. Notice, however, that cooperative responses are only computed for queries that fail, which are rare, and the back-end can amortize the cost of computing cooperative responses for failing queries over the mix of failing and succeeding queries.[9]

[8]Notice, though, that the front-end computations that we are neglecting are sequential and grow exponentially with the number of search terms.

[9]As mentioned above, the back-end for Noflail Search is the Bing API provided by Microsoft. We discussed the resource-consumption issue with Alessandro Catorcini of Microsoft. Because failing queries are very rare, he allowed us to use cooperative respones without setting any limit on N, and even offered to help if we had problems with any countermeasures that might be in place against denial-of-service attacks. We are very grateful for this. We decided nevertheless to set an upper limit on N: Noflail Search provides coopertive responses only for queries having up to six search terms.

6. CONCLUSION

We have described a parallel algorithm for computing cooperative responses to failing conjunctive queries through a Web API. The algorithm takes advantage of the inherent parallelism of the API and makes it practical to provide cooperative responses in spite of the considerable latency of the API. The algorithm has been implemented on the search front-end Noflail Search, available at noflail.com, where cooperative responses are integrated with a new user-interface feature that lets the user browse multiple result sets at once. We have presented a sample of performance results showing how the algorithm reduces the latency of computing cooperative responses to queries with 3 to 6 search terms from a range of 2 to 50 seconds to a range of 1 to 7 seconds. We have provided pseudo-code for the algorithm and proved soundness, completeness and non-redundancy results. We have discussed how the algorithm could be extended to handle general Boolean queries.

The current surge of innovation in search and data mining presents many opportunities for further work on cooperative responses. New user-interface paradigms for search are emerging. Cooperative responses should be integrated with these paradigms and may in turn suggest new ones. The availability of large-scale repositories of semantic information may offer new perspectives for investigating the use of semantics for query relaxation and detection of false presuppositions. We believe that cooperative responses have an important role to play in the search revolution.

7. REFERENCES

[1] F. Benamara and P. Saint-Dizier. Advanced relaxation for cooperative question answering. In M. T. Maybury, editor, *New Directions in Question Answering*, pages 263–274. AAAI Press, 2004.

[2] W. W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, and C. Larson. Cobase: A scalable and extensible cooperative information system. *J. Intell. Inf. Syst.*, 6(2/3):223–259, 1996.

[3] F. Corella, S. J. Kaplan, G. Wiederhold, and L. Yesil. Cooperative responses to boolean queries. In *ICDE*, pages 77–85. IEEE Computer Society, 1984.

[4] G. Dong, X. Lin, W. W. 0011, Y. Yang, and J. X. Yu, editors. *Advances in Data and Web Management, Joint 9th Asia-Pacific Web Conference, APWeb 2007, and 8th International Conference, on Web-Age Information Management, WAIM 2007, Huang Shan, China, June 16-18, 2007, Proceedings*, volume 4505 of *Lecture Notes in Computer Science*. Springer, 2007.

[5] T. Gaasterland. Cooperative answering through controlled query relaxation. *IEEE Expert*, 12(5):48–59, 1997.

[6] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. *J. Intell. Inf. Syst.*, 1(2):123–157, 1992.

[7] T. Gaasterland, P. Godfrey, and J. Minker. Relaxation as a platform for cooperative answering. *J. Intell. Inf. Syst.*, 1(3/4):293–321, 1992.

[8] T. Gaasterland and J. Lobo. Qualifying answers according to user needs and preferences. *Fundam. Inform.*, 32(2):121–137, 1997.

[9] A. Gal and J. Minker. Informative and cooperative

answers in databases using integrity constraints. In *Natural Language Understanding and Logic Programming Workshop*, pages 277–300, 1987.

[10] P. Godfrey. Minimization in cooperative response to failing database queries. *Int. J. Cooperative Inf. Syst.*, 6(2):95–149, 1997.

[11] J. M. Janas. On the feasibility of informative answers. In *Advances in Data Base Theory*, pages 397–414, 1979.

[12] M. Kao, N. Cercone, and W.-S. Luk. Providing quality responses with natural language interfaces: The null value problem. *IEEE Trans. Software Eng.*, 14(7):959–984, 1988.

[13] S. J. Kaplan. Designing a portable natural language database query system. *ACM Trans. Database Syst.*, 9(1):1–19, 1984.

[14] R. M. Lee. Conversational aspects of database interactions. In S. B. Yao, editor, *VLDB*, pages 392–399. IEEE Computer Society, 1978.

[15] A. Motro. Query generalization: A method for interpreting null answers. In *Expert Database Workshop*, pages 597–616, 1984.

[16] A. Motro. Seave: A mechanism for verifying user presuppositions in query systems. *ACM Trans. Inf. Syst.*, 4(4):312–330, 1986.

[17] A. Motro. Intensional answers to database queries. *IEEE Trans. Knowl. Data Eng.*, 6(3):444–454, 1994.

[18] I. Muslea. Machine learning for online query relaxation. In W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, editors, *KDD*, pages 246–255. ACM, 2004.