The CompleteSearch Engine: Interactive, Efficient, and Towards IR&DB Integration

Holger Bast   Ingmar Weber

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{bast,iweber} at mpi-inf dot mpg dot de

ABSTRACT

We describe *CompleteSearch*, an interactive search engine that offers the user a variety of complex features, which at first glance have little in common, yet are all provided via one and the same highly optimized core mechanism. This mechanism answers queries for what we call *context-sensitive prefix search and completion*: given a set of documents and a word range, compute all words from that range which are contained in one of the given documents, as well as those of the given documents which contain a word from the given range.

Among the supported features are: (i) automatic query completion, for example, find all completions of the prefix "seman" that occur in the context of the word "ontology", as well as the best hits for any such completion; (ii) semi-structured (XML) retrieval, for example, find all email-messages with "dbworld" in the subject line; (iii) semantic search, for example, find all politicians which had a private audience with the pope; (iv) DB-style joins and grouping, for example, find the most prolific authors with at least one paper in both "SIGMOD" and "SIGIR"; and (v) arbitrary combinations of these.

The prefix search and completion mechanism of CompleteSearch is realized via a novel kind of index data structure, which enables subsecond query processing times for collections up to a terabyte of data, on a single PC. We report on a number of lessons learned in the process of building the system and on our experience with a number of publicly used deployments.

1. INTRODUCTION

We start right away by explaining CompleteSearch's central *context-sensitive prefix search and completion* mechanism, which will be the basis for everything else in this paper. This mechanism solves instances of a non-standard range-searching problem, proposed in [5] and [6]. We first give a formal definition of the problem, and then explain it by a number of examples. Section 2 will summarize the basic idea of the index data structure behind CompleteSearch. Section 3 will give a detailed account of the features provided by CompleteSearch. Section 4 will comment on related work. In Section 5 we give implementation details and report on some of the lessons learned in the 2 1/2 years of our work on this system. Section 6 compiles a wish list of things that are still left to do. Several live demos of the CompleteSearch engine are available under `http://search.mpi-inf.mpg.de`.

DEFINITION 1. *For a given collection of documents, with a unique id for each document and a unique id for each of the words used in the collection*[1], *a* context-sensitive prefix search and completion query *is a pair* (D, W), *where* D *is a* set *of document ids and* W *is a* range *of word ids. To process the query means to compute* a ranked list of all pairs (d, w), *where word* w *occurs in document* d, d *is from* D *and* w *is from* W.

Definition 1 can be understood in a number of ways. In the following we will give *three* interpretations: one from an *IR perspective*, one from a *DB perspective*, and one from a *theorist's perspective*. The ranking mechanism will be explained in more detail in Section 2; for now, let us just take it for granted.

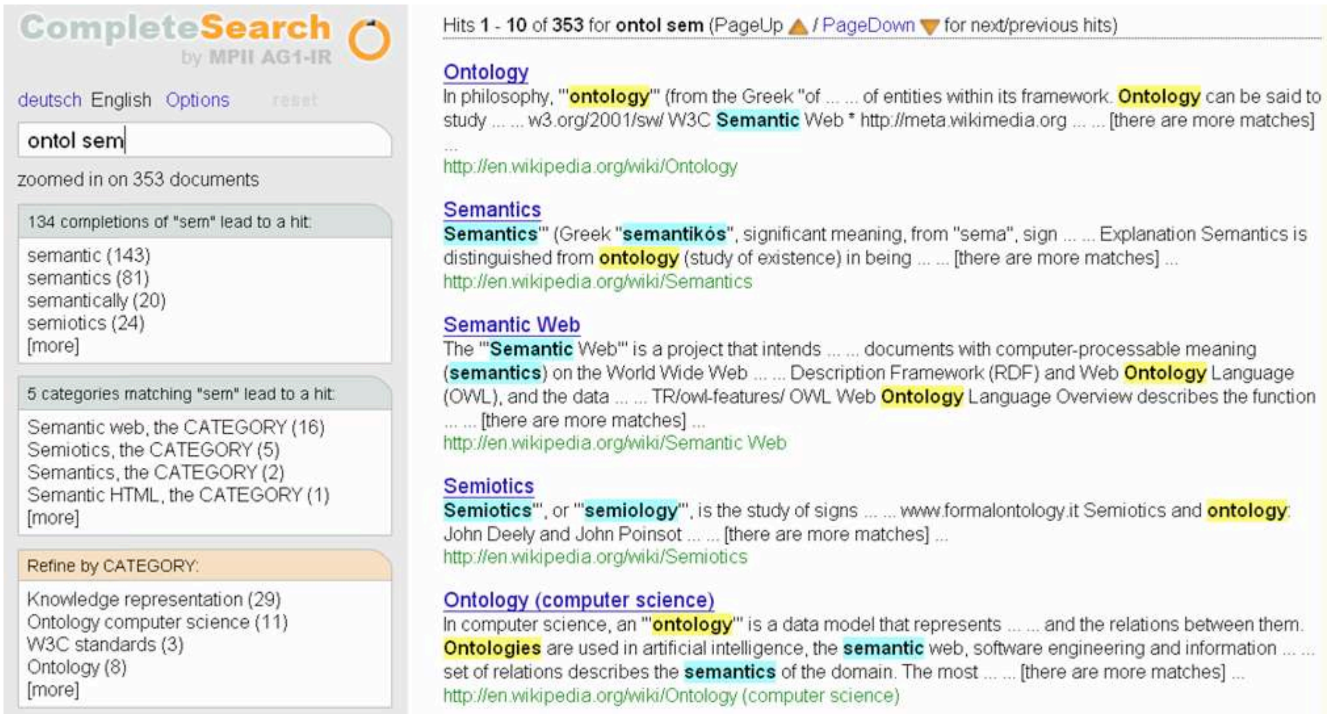
IR perspective

The original use of Definition 1 was for the following interactive *autocompletion* feature. Imagine a user of a search engine typing a query. Then with every letter being typed, display completions of the last query word that would lead to good hits, as well as the best hits for any of these completions. Here is an example. Consider a user searching the English Wikipedia (one of our demo collections), having typed his or her query to the point `ontol sem`. Then the input set D would be the set of ids of documents containing a word starting with[2] `ontol` (like `ontology`, `ontological`, etc.), and the input range W would be the range of ids of words starting with `sem`.

The output set would then consist of pairs (d, w), where word w starts with `sem` and occurs in a document d that

[1]Different occurrences of the same word have the same id.

[2]By default, CompleteSearch assumes an implicit `*` at the end of each query word, because that is the desired behavior in most cases. Exact-word matches can be enforced by ending a word with a `$`



ontol sem

zoomed in on 353 documents

134 completions of "sem" lead to a hit:
semantic (143)
semantics (81)
semantically (20)
semiotics (24)
[more]

5 categories matching "sem" lead to a hit:
Semantic web, the CATEGORY (16)
Semiotics, the CATEGORY (5)
Semantics, the CATEGORY (2)
Semantic HTML, the CATEGORY (1)
[more]

Refine by CATEGORY:
Knowledge representation (29)
Ontology computer science (11)
W3C standards (3)
Ontology (8)
[more]

Hits 1 - 10 of 353 for ontol sem (PageUp ▲ / PageDown ▼ for next/previous hits)

Ontology
In philosophy, '''ontology''' (from the Greek "of ... ... of entities within its framework. Ontology can be said to study ... ... w3.org/2001/sw/ W3C Semantic Web * http://meta.wikimedia.org ... ... [there are more matches] ...
http://en.wikipedia.org/wiki/Ontology

Semantics
Semantics''' (Greek "semantikós", significant meaning, from "sema", sign ... ... Explanation Semantics is distinguished from ontology (study of existence) in being ... ... [there are more matches] ...
http://en.wikipedia.org/wiki/Semantics

Semantic Web
The '''Semantic Web''' is a project that intends ... ... documents with computer-processable meaning (semantics) on the World Wide Web ... ... Description Framework (RDF) and Web Ontology Language (OWL), and the data ... ... TR/owl-features/ OWL Web Ontology Language Overview describes the function ... ... [there are more matches] ...
http://en.wikipedia.org/wiki/Semantic Web

Semiotics
Semiotics''', or '''semiology''', is the study of signs ... ... www.formalontology.it Semiotics and ontology: John Deely and John Poinsot ... ... [there are more matches] ...
http://en.wikipedia.org/wiki/Semiotics

Ontology (computer science)
In computer science, an '''ontology''' is a data model that represents ... ... and the relations between them. Ontologies are used in artificial intelligence, the semantic web, software engineering and information ... ... set of relations describes the semantics of the domain. The most ... ... [there are more matches] ...
http://en.wikipedia.org/wiki/Ontology (computer science)

Figure 1: A screenshot of our search engine for the query `ontol sem` searching the English Wikipedia. The list of completions (on the left) and hits (on the right) are updated automatically and instantly after every keystroke, hence the absence of any kind of search button. The number in parentheses after each completion is the number of hits that would be obtained if that completion where typed. As completions, CompleteSearch offers ordinary words as well as phrases, subwords, and category names, if appropriate. The "Refine by" box gives a breakdown of the 353 hits by the most prominent categories in that set. This box is produced proactively without any special action on part of the user, by launching the query `ontol sem cat:` in the background; this is explained in Section 3.5.

also contains a word starting with `ontol`. Top ranked words would be `semantic` and `semiotics`,[3] but not for example `semiconductor`, which, although it is one of the most frequent words starting with `sem` in Wikipedia, does not occur prominently in the context of words starting with `ontol`. Top ranked documents would be an article about ontologies in general, an article about the meaning and origin of the word semantics, and an article about the semantic web. See Figure 1 for a screenshot of our search engine in action for that query.

DB perspective

Definition 1 could also be viewed as the problem of computing what could be called a *half join*. For example, consider a collection of computer science articles (another one of our demo collections), and assume that each article contains special words of the form `<category name>:<category instance>` (the colon serves to distinguish these words from ordinary words), for example, `conference:vldb`, or `author:jon_kleinberg`, or `year:2006`. Observe that by adding these special words, we implicitly create a table with the schema (conference, author, year, publication). We will see in Section 2, that our index data structure actually stores the columns of this table in contiguous memory, just by the way it works.

Now consider the *two* queries `conference:sigir author:` and `conference:sigmod author:`. According to Definition 1, the first query produces a list of authors who have published at SIGIR, along with the corresponding publications. Similarly, the second query produces a list of authors who have published at SIGMOD, along with the corresponding publications. Now let us us intersect the two lists of authors, that is, the lists of (ids of) *completions* of the two queries. Note the duality to the archetypical search engine operation of intersecting lists of (ids of) *documents*. The intersection of the two lists of completions gives us the list of all authors, who have published at both SIGIR and SIGMOD, and the two lists of documents provide the witnesses of these facts. That is, with *two* of our context-sensitive prefix search and completion queries we have effectively computed a *self-join* on the table which he have implicitly created by the addition of the special words. In Section 3.4, we explain how to generalize this to arbitrary joins. Note that the information required to process this kind of query is spread over several documents, which is something standard IR-style keyword search cannot handle. For example, the query `conference:sigir conference:sigmod author:` would not match any document, because no document is a SIGIR paper and a SIGMOD paper at the same time.

[3]Semiotics, or semiology, is the study of signs and symbols and how meaning is constructed and understood.

If we prepended `ir db integration` to the two queries above, we would obtain the join table restricted to documents matching this query, and we would obtain a list of authors which have published at both SIGIR and SIGMOD about the topic of IR&DB integration. This is a first example of how CompleteSearch can combine IR-style with DB-style querying. We will elaborate on this in Section 3.4. In Section 3.5, we will show how to take away from the user the burden of having to know the syntax of the special words.

Theorist's perspective

From an efficiency point of view, it is instructive to view Definition 1 as formalization of a non-standard, 1 1/2-*dimensional range-searching problem*.

The one-dimensional range-searching problem that corresponds to Definition 1 would be, given a word range, to find all word-in-document pairs with a word from that range. Such range queries can be processed efficiently using a B-tree like data structure, but they would not give us the *context-sensitivity* that is so essential for all the features of CompleteSearch. Sorting the result pairs by document id is an efficiency problem already for the one-dimensional case. As we will see in Section 2, our index data structure manages to avoid sorting.

The multi-dimensional range-searching problem that corresponds to Definition 1 would be, given a d-tuple of ranges, to find from a given collection of d-tuples all tuples matching all d given ranges. This is a much researched, very hard problem [13]. All the data structure we know of that solve this problem directly, without solving the corresponding d one-dimensional subproblems first, have a space consumption on the order of $N^{1+d'}$, where N is the size of the collection, and d' grows fast with the dimensionality d of the query [3] [1] [11].

In the interactive scenario we consider, where the partial query is processed after every keystroke, the computation for a d-dimensional query can reuse the result from the preceding $(d-1)$-dimensional query. This naturally gives rise to the 1 1/2-dimensional range searching problem captured by Definition 1. For example, the sorted list of document ids for the query `ontol sem` would serve as input set D for the query `ontol sem search`. Also, many of the proactively launched background queries discussed in Section 3 just modify an existing query by changing its last word or by appending a prefix. Note that problems of join ordering and query plan optimization do not arise in our interactive setting, because there is no choice here but to evaluate the query in a strict order from left to right.

2. THE HYB DATA STRUCTURE

The central completion mechanism of the CompleteSearch engine makes use of a novel kind of index data structure, called HYB, that was first presented in [6]. In this section, we briefly present the main ideas behind HYB, to the extent that it will be useful for understanding what follows in this paper.

The basic unit of processing of HYB is a *block*. Each block corresponds to a range of words, and these ranges form a partitioning of the set of all words. In the simplest conceivable setting, without ranking and without positional information, a block consists of all pairs (w, d), where word w occurs in document d, and w is from the word range pertaining to that block. Each block is sorted by document id.

For example, assume we have 10 documents overall with ids 3, 5, 6, 7, 8, 9, 11, 12, 13, 15, which contain words A, B, ..., Z, and the four words A, B, C, and D are contained in the following manner:

A	:	3, 5, 6, 8, 9, 11, 12, 15
B	:	5, 11
C	:	3, 7, 11, 13
D	:	3, 8

Then a block of HYB for the word range A - D would correspond to the sequence of pairs

3	3	3	5	5	6	7	8	8	9	11	11	11	12	13	15
A	C	D	A	B	A	C	A	D	A	A	B	C	A	C	A

HYB consists of a collection of such blocks, one for each range from some partitioning of the full range of words (we comment on appropriate partitionings below), with every word-in-document pair being stored in exactly one block.

It is proven both theoretically and empirically in [6] that these blocks can be compressed extremely well (at least as well as the lists of an inverted index). Note that if we picked one block for each single word (that is, the word ranges corresponding to the blocks would all be singletons), HYB would degenerate to a (compressed) inverted index. If, in the other extreme, we picked a single large block for the range of all words, HYB would degenerate to a (compressed) representation of the original documents. HYB stands right in the middle between these two extremes, as a hybrid between the two, hence its name.

It is shown in [6] that blocks should be chosen of about equal volume, where volume means number of pairs (w, d), and that for optimal space efficiency and query processing time, this volume should be chosen as a small fraction of the total number of documents. CompleteSearch performs two basic operations on the blocks of HYB: intersection with a sorted list of document ids, which is fast because the blocks are sorted by document ids; and intersection with a list of word ids, which is fast, because the word ids from a block come from a relatively small range. For each query, CompleteSearch always processes *at least one full block* of HYB, and for most queries it actually processes *exactly one block*.

For example, consider the query `ontol sem search`, and assume that the sorted list of ids of documents matching `ontol sem` has already been computed (right after the last letter of that query was typed). This list would then be intersected with the sorted list of document ids from the block(s) containing all occurrences of the word `search` (considering only those document ids where the corresponding word id matches `search`). If this is just a single block, which it will be if the beginning of the last query word, `search` in this case, is specific enough, we obtain the sorted list of documents ids for `ontol sem search` in linear time, without having to sort or merge.

In a full-blown index, the blocks of HYB are augmented by parallel lists of word positions (needed for phrase and proximity search) and scores (needed for ranking), both of which are compressed, too.

The block structure of HYB has a variety of advantages. It is simple. It can be compressed provably well. It enables a processing of the prefix search and completion queries according to Definition 1 by mere sequential access, without the need for sorting or other non-linear operations (more about this in Section 5.1). In particular, tables created by inserting special words with a common prefix, as described in the introduction, will have their columns stored in contiguous memory. Finally, HYB can be combined with techniques for top-k retrieval [10] [4]; however, we have not fully exploited that potential yet, see Section 6.

Given the simple structure of HYB, ranking of the result lists is relatively straightforward, and easy to customize too. For each word-in-document pair in the index, we have a precomputed score. This could be a BM25 score reflecting term and inverse document frequency [20], or it could be a bit vector reflecting whether that word appears in the title or is set in a particular font, or it could be any combination of these. Whenever we process two result lists (forming their union or their intersection, depending on the query), we simply aggregate scores according to some user-definable function, for example, sum the BM25 scores.

Table 1 repeats the main performance figures from [6], which show that HYB compresses indeed as well as an inverted index, yet can process context-sensitive prefix search and completion queries according to Definition 1 by an order of magnitude faster.

Collection	Method	space	avg query	99%-ile
Homeopathy (450 MB/pos)	INV	70 MB	0.033 secs	0.384 secs
	HYB	62 MB	0.003 secs	0.026 secs
Wikipedia (7.4 GB/pos)	INV	2.2 GB	0.171 secs	2.272 secs
	HYB	2.0 GB	0.055 secs	0.492 secs
Terabyte (426 GB/nopos)	INV	4.6 GB	0.581 secs	16.83 secs
	HYB	4.9 GB	0.106 secs	0.865 secs

Table 1: Index size, average query time, and 99%-ile of query times, of our block data structure (HYB) versus the inverted index (INV), on three test collections. The parenthesis below each collection name specifies the raw size of the collection and whether the index was built with positional information or not.

3. COMPLETESEARCH'S FEATURE SET

In this section, we give an account of CompleteSearch's feature set. We focus on the most important features, trying to emphasize the diversity of functionality supported. The main message of this section is that we get all this functionality via one and the same mechanism, namely our context-sensitive prefix search and completion, by only adding suitable words to the documents. Note the relevance to a web scenario, where users have control over their documents, but not over the search engine.

Some of the features discussed below were already anticipated in [6]. The faceted-search feature was first presented at [7]. The DB-style join feature is new and came as a pleasant surprise for us when working on the original submission of this paper. As we will see in Section 3.4, the fact that HYB stands right in the middle between a representation by document and a representation by word is especially crucial for the join queries.

For the complete set of features, which also includes proximity search, boolean OR, and negation, check out the online help of our live demos under `http://search.mpi-inf.mpg.de`.

3.1 Context-sensitive autocompletion search

This is the feature we already discussed in the introduction for the query `ontol sem`, namely to compute all completions of the last query word, `sem` in this case, which would lead to good hits together with the preceding part of the query, `ontol` in this case, as well as the best such hits. This feature is useful in a variety of ways. It saves typing. It spares the user the experience of overspecifying the query, when already a (much) shorter query would give the desired result. It helps the user exploring formulations used in the collection, substantially reducing the amount of guess work required. Note that without the context-sensitivity this feature would lose most of its worth; for example, there are thousands of completions of `sem`, but only few that make sense in the context of words starting with `ontol`. If the index has been built with positional information, the user may also require that completions of `sem` come right after a word starting with `ontol` (phrase search), or within a window of, say, 10 words (proximity search).

Over the last two years, related autocompletion features have been added to a number of desktop and web search engines, for example, Apple's Spotlight (`www.apple.com/macosx/features/spotlight`), Google Suggest (`labs.google.com/suggest`), and Alltheweb Live Search (`livesearch.alltheweb.com`). We remark that a prototype of our engine already existed when Google Suggest and Apple Spotlight were launched. We also remark that any of these services offers only a much simplified feature compared to our Definition 1. Google Suggest, for example, completes from a list of popular queries, which is algorithmically easy (ordinary search in a sequence of strings) compared to our context-sensitive full-text completion queries.

More value is added to the autocompletion feature, if we augment the index by certain subwords and phrases. Then, for example, `sear` can also complete to `livesearch`, and `max` can complete to the phrase `max planck institute`. Check out our demos for more examples.

3.2 Structured search in XML documents

Any kind of full-text index with support for proximity search can be easily extended to take advantage of semi-structured text, by which we here mean text enriched with XML tags. A generic way is to add all XML tags as special words (that is, recognizable as such), for example, `tag:email` or `tag:subject`. It is then straightforward to extend the proximity operator such that a word is considered close to a particular tag if and only if it occurs between a corresponding tag pair. In CompleteSearch, the proximity operator is denoted by `..` (two dots). An example query would be `tag:email..tag:subj..dbworld`, which would retrieve all email messages (tagged as such) mentioning (a word starting with) `dbworld` in their subject line.

This simple trick supports a subset of the *XPath* query syntax, called NEXI [22]. XML support has been added in a similar way to the TopX engine [21], which we will briefly discuss in Section 4. Note that CompleteSearch permits free mixing of queries using tag information with any of the other query types.

3.3 Semantic search

Here is a feature which we would not get from an ordinary full-text search index, but which we get easily from a mechanism solving instances of the problem from Definition 1. The feature we are going to describe would otherwise require an ad hoc solution and efficiency would be non-trivial to achieve.

Consider the Wikipedia collection, and assume we have tagged all mentionings of a *politician*.[4] Then create a copy of each such mentioning, prefixed by a unique category identifier, for example, wherever Tony Blair is mentioned, add the word `politician:tony_blair`.

This gives us a basic question answering facility. For example, the question *Which politician had a private audience with a pope?* could be formulated as the query `audience pope politician:`. Our context-sensitive prefix search and completion mechanism according to Definition 1 would then compute a ranked list of completions of `politician:` which occur in the context of `audience pope`.

In fact, we can even take away from the user the burden of having to know the special syntax `politician:` by launching appropriate queries in the background. Such behavior of a search engine is called *proactive*. More details on this are given in Section 3.5.

3.4 DB-style joins

The previous section showed that questions such as *Which politicians had an audience with the pope?* can be dealt with if only we have the necessary semantic annotation. Matters become more complicated when, as is often the case, the information required to answer the question is *spread over several pages*. For example, to find an answer to the question *Which German chancellors had an audience with the pope*, it might be essential to combine information from the following two pages: one page about Angela Merkel, mentioning that she is the current German chancellor (but not that she met with the pope), and another page about the current pope having met Angela Merkel. For this example, assume that in both documents, Angela Merkel has been correctly tagged as a politician, as described in the previous section.

What we need then is DB-style *join* functionality, and it came as a surprise for ourselves that we can also reduce this operation to an instance of the problem from Definition 1. For the above question, all we have to do is launch the two queries `german chancellor politician:` and `audience pope politician:` and intersect the two lists of *completions*. (Note the duality: the archetypical search engine operation is to intersect lists of *documents*.) This will give us a list of names of politicians with both properties, as well as a list of document pairs witnessing this fact. These witnesses will exactly be the combination of the results for the first query, proving that the particular politician is or was German chancellor, and the second query, proving that the particular politician had an audience with the pope.

As we explain next, it is not hard to generalize this to arbitrary joins. Given any table named ABC with attributes (corresponding to columns) $attr\_1$ up to $attr\_n$, create a special document for each row of the table as follows. First add the name of the table as `table:ABC` and then the attribute-value pairs as `attr_1:<val_1>` up to `attr_n:<val_n>`, where `<val_i>` is the entry for attribute i in the considered row.

If we then want to compute the inner join with table XYZ on attribute $attr\_k$, we launch the queries `table:ABC attr_k:` and `table:XYZ attr_k:` as prefix search and completion queries according to Definition 1. For the two result sets, we then intersect the lists of matching *completions* (not documents). These completions are then exactly the matching attribute-value pairs for the join attribute. To obtain the corresponding rows of the join result table efficiently, note that whenever we are intersecting lists of word ids with HYB, we are actually handling *pairs*, and we also have the corresponding document id at hand (and vice versa when we are intersecting lists of document ids). This way we can easily obtain the corresponding document ids, which correspond to the matching rows in both tables. By slightly modifying the intersection routine to output NULL when a word id is present in only one of the two lists, we can use the same procedure to compute left, right or outer joins as well.

Since the special words for a particular attribute of any such table share a common prefix, they will be stored in consecutive locations by HYB, and will either form their own block, or be part of a single block. This allows for an efficient processing of join queries. Note that the complex problem of *join ordering* does not occur in our interactive setting, because the fact that we want results after every keystroke demands an evaluation of the query in a strict order from left to right.

3.5 Proactive search

The fanciest and most well-meant features are bound to be left unnoticed if the query syntax is too complicated. This applies to user interfaces in general, but especially to search engines, where users expect to get from a vaguely felt search desire to relevant results without much trouble or special instructions.

Here the completion mechanism of CompleteSearch comes in handy again. Assume the user is looking for politicians and has started to type `politic`. Assume that for each document categorised under *Politicians*, we have added the special word `cat:politician`. Then the answer to the query `cat:politic` will give us all category names starting with `politic`. CompleteSearch can be configured to launch these queries, with `cat:` prepended to the last word, automatically, and thus inform the user whenever there is category information relevant to one of the query words typed. Note that we get context-sensitivity for free here. If, for example, the user has typed `altruist politi`, the category *Politicians* will be suggested to the user only if there exists a page mentioning a politician and also containing a word starting with `altruist`.

[4]How we obtain such a tagging is an issue orthogonal to the aspects considered in this paper. For our Wikipedia demo we currently use the following simplistic approach: Wikipedia has category information for most pages. In particular, all politicians' pages are in the category *Politicians*. Most mentionings of a politician will link to that politician's page, which gives us the desired information for tagging.

With category information of the kind `cat:...` added to the index, we get yet another feature, known as *faceted search* [7]. Namely, whenever a query `<query>` is being considered, CompleteSearch also launches the query `<query> cat:` in the background. According to Definition 1, the completions for this background query will be a ranked list of categories that occur in the context of `<query>`, which gives nothing else but a breakdown of the hits for `<query>` according to whatever category information has been added to the index. This feature lets the user freely alternate between directory-style browsing and keyword search, which has been shown to enhance user experience and retrieval quality significantly [14]. For an example, see Figure 1, which shows a breakdown of the hits for `ontol sem` by Wikipedia categories, for example, *Knowledge Representation* and *W3C standards.* Optionally, the displayed hits can also be *grouped* with respect to these categories.

Note that we could get this feature by an ad hoc add-on to whatever system we have at our disposal. With an underlying DBMS, for example, this feature is a just matter of suitable SQL queries, however at the price of a very substantial loss in efficiency. With CompleteSearch we merely have to add the right words to the index and rewrite queries in an appropriate way. It turns out that the queries for realizing faceted search are among those which yield the worst-case behavior for both our block index HYB and the inverted index INV. However, HYB beats INV by an order of magnitude also in this worst case; performance figures for a large collection (Wikipedia) are provided in [7].

4. RELATED WORK

The QUIQ engine [15] is another recent attempt to integrate IR and DB functionality into a single system in a uniform manner. QUIQ is built on top of a DBMS, partly motivated by their focus on *dynamic updates* (to which we give only relatively little attention, see Section 6). Like CompleteSearch, QUIQ makes extensive use of the idea to "map non-text data to pseudo-keywords that cannot be confused with actual keywords of text".

The *TopX* engine, developed by our colleagues at MPII [21], combines search in semi-structured (XML) data with techniques for top-k retrieval, with a strong focus on the latter. As explained in Section 3.2, CompleteSearch supports exactly the same subset of XPath queries as TopX. Like QUIQ, TopX is built on top of an off-the-shelf DBMS (Oracle).

The *HySpirit* system [12] was designed for "hypermedia retrieval integrating concepts from information retrieval and deductive databases." The system is based on a probabilistic model of Datalog. Like CompleteSearch, it can combine ranked retrieval with database queries. Like QUIQ and TopX, HySpirit is built on top of a DBMS.

Our work on CompleteSearch addresses some of the issues and challenges raised in a recent overview paper by Chaudhuri, Ramakrishnan, and Weikum [8]; see also [9]. Our central completion mechanism might be viewed as an instance of the "storage-level core system with RISC-style functionality" argued for in [8]. We certainly agree with their point of view that an integrated IR&DB (or DB&IR) system should *not* be built on top of an SQL-engine or a vanilla B-tree implementation, for reasons of efficiency. Table 2, which will be discussed in Section 5, gives a simple confirmation. Flexible scoring and ranking and high-performance query processing, the first two items on the requirement list of [8], are at the core of the design of CompleteSearch.

For a more thorough overview of the area of IR&DB-integration, we refer the reader to the SIGMOD'05 panel discussion [2], in particular its references. A classification of existing schemes according to criteria such as integration architecture and general approach is attempted in [19].

We discuss work related to aspects which are not yet adequately addressed by CompleteSearch in our Conclusions, Section 6.

5. LESSONS LEARNED

We give an account of the main implementation issues and a brief overview of the system's architecture. Several instances of CompleteSearch are up and running and publicly used, and we report on some experiences with our users.

5.1 Locality of Access

Efficiency was of utmost importance to us in the design and implementation of the central prefix search and completion mechanism of CompleteSearch. To achieve that, the following three aspects turned out to be most essential: (i) that access to the data is *as sequential as possible*, (ii) that *as little data as possible* is processed per query, and (iii) a very careful, hardware-aware (yet portable) implementation. We elaborate on each of these three aspects in the following.

It is a truism that sequential access to data is faster than random access. For a typical disk, average seek time is 5 milliseconds versus an average transfer rate of 50 Megabytes per second. But even when the data is entirely in main memory, sequential access is up to 100 times faster than random access. This factor tends to be smaller for complex applications (or programs in higher-level languages, see the next but one paragraph), but when other factors of inefficiency are eliminated it plays a crucial role. Indeed, our first[5] index data structure, presented in [5], was theoretically close to optimal in that we could prove its query processing time to be asymptotically bounded by the size of the output. Yet, our follow-up scheme from [6], without this theoretically desirable property, but highly optimized for locality of access, beats the scheme from [5] by a factor of more than 5.

To reduce the amount of data that have to be read from disk and processed per query, the HYB index makes extensive use of compression. Further, it is one of the distinguishing features of HYB that the index data is laid out such that the processing of a query requires mere *scans* of portions of the data. In particular, no sorting or other non-linear or non-local operations of large portions of the data are required. We also make use of techniques for top-k retrieval [10] [4], though not (yet) exploiting their full potential; see Section 6.

Concerning implementation, C++ was the programming language of choice. It is often debated how much faster an implementation in C++ really is compared to, say, a program written in JAVA, or queries to a DBMS like Oracle or MySQL. Indeed, anecdotal evidence as well as a study by Prechelt [18] have it that the choice of programming language does not make much of a difference for the average

[5]Note that the work on [5] predated that of [6] by almost a year, the two articles just happened to be accepted for publication at about the same time.

C++	JAVA	MySQL	Perl
1800 MB/s	300 MB/s	16 MB/s	2 MB/s

Figure 2: Average processing rate (in Megabytes per second) for four different programming languages/environments scanning an ordinary (in-memory) array of 10 million 4-byte integers, measured on a Linux PC with two 3 GHz Intel Xeon processors and 4 GB of main memory. The rate for C++ is close to the 2 GB/s memory bandwidth specified for that machine.

program. However, when it comes to algorithms highly optimized for sequential access to data, the difference is enormous. See Table 2, where for a simple scanning task, C++ wins over JAVA by factor of 6, over a MySQL application by a factor of more than 100, and over a scripting language like Perl by a factor of almost 1000.

We made extensive use of *templating*, to reduce the code complexity without compromising instruction-cache efficiency (few instructions in the inner loops) and branch predictability (no conditionals in the inner loops, wherever possible).

5.2 An interactive web-application

Building an interactive web application like CompleteSearch that is supposed to display its GUI via any standard web browser is a very challenging task.

It starts with the design, which is all but obvious. The completion server necessarily has a non-negligible start-up cost and cannot be started from scratch for every query, but has to run as a background process continuously. But letting the client's web browser communicate with a program on a remote computer is a security problem. We solved this by a three-step approach: the web page displayed to the client contains JavaScript code, which for each user action triggers the loading of a special web page via an AJAX protocol. This web page is dynamically created via PHP, in particular taking care of the communication with the completion server, and generating the HTML as well as the JavaScript code.

The advantage of this approach is that no installation or special software is required on the side of the user; any standard web browser will do. Nor can any firewall settings be a problem: if web browsing works, CompleteSearch works too. The price for this is complex code on three different machines (completion server, web server, client machine) which interacts with each other in a non-trivial manner, and can be hard to debug. Missing standards and inconsistencies concerning the way web browsers process JavaScript, render a complex layout, or deal with the the history (back button) are a constant source of trouble.

5.3 User Feedback

The CompleteSearch engine would not be close to what it is today without the feedback of our users. In this section we report on some of the main lessons we learned from this feedback loop.

The first users were ourselves. When starting the project 2 1/2 years ago, we first wrote a prototype (in Perl) to see the search engine in action, on a real collection. Many of the features were born in that way, and a number of features which we deemed interesting at first were discarded in that process.

One of the lessons we had to learn was that the vast majority of (our) users is not willing to read even the tiniest bit of documentation before using a search engine, not even if the search does not give the expected results. Actually, we anticipated this to some extent, and tried to keep the user interface intuitive and simple right from the beginning. And after all, the whole approach of CompleteSearch is a proactive one: display completions, hits, refinements, alternatives, etc. as the user is typing. If he or she opts to ignore this information, the basic functionality of a search engine is still there.

But the following surprised us: below the search field we put a very short note saying "Type `?` for help", and the mechanism was such that typing `?` at any point in the query would instantaneously display a few sentences on the most important advanced operators which can help improve search results. Well, hardly any user ever pressed the `?` key, let alone read the help information. After this experience we abandoned all our plans for more elaborate help pages, feedback forms, etc. and focused on making our whole system as proactive as possible.

6. THINGS TO DO

As explained in Section 5.1, we have made strong efforts, in algorithmic design as well as in implementation, to limit the sheer amount of data that has to be processed per query and to make access to it almost exclusively sequential. This gives us subsecond query times (suitable for an interactive system) for collections up to a terabyte in size, on a single PC.

However, we have not yet fully exploited the potential of *top-k retrieval techniques* [10] [4] for further reducing the amount of data that has to be scanned, especially for large word ranges. This is work in progress.

An aspect to which we have paid little attention so far, is the question of how to deal with *dynamic updates*. So far, our philosophy has been to split large collections into several parts, and to rebuild partial indices from scratch when it becomes necessary. In the IR world this is actually considered one of the most effective ways of updating [16]. Still there is work to do for us here, especially in automating this process.

We have pointed out the applicability of our Definition 1 to a variety of query types, both IR and DB-style, and also combinations of the two. Our account in Section 3 is somewhat by example. It looks like there is an interesting, non-trivial theoretical connection between Definition 1 and standard concepts from IR and DB to be worked out.

Finally, there is the issue of distributing our indices over several machines, to be able to scale up to not just millions but billions of documents. Preliminary work in this direction made us optimistic that standard techniques for distributing very large corpora by document [17] would also work for CompleteSearch.

Acknowledgments

We thank the anonymous referees for their useful and inspiring comments.

7. REFERENCES

[1] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *41st Symposium on Foundations of Computer Science (FOCS'00)*, pages 198–207, 2000.

[2] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the DB/IR panel at SIGMOD 2005. *SIGMOD Record*, 34(4):71–74, 2005.

[3] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *18th Symposium on Principles of database systems (PODS'99)*, pages 346–357, 1999.

[4] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-Top-k: Index-access optimized top-k query processing. In *32nd International Conference on Very Large Data Bases (VLDB'06)*, pages 475–486, 2006.

[5] H. Bast, C. W. Mortensen, and I. Weber. Output-sensitive autocompletion search. In *13th Symposium on String Processing and Information Retrieval (SPIRE'06)*, pages 150–162, 2006.

[6] H. Bast and I. Weber. Type less, find more: Fast autocompletion search with a succinct index. In *29th Conference on Research and Development in Information Retrieval (SIGIR'06)*, pages 364–371, 2006.

[7] H. Bast and I. Weber. When you're lost for words: Faceted search with autocompletion. In *SIGIR'06 Workshop on Faceted Search*, pages 31–35, 2006.

[8] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 1–12, 2005.

[9] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning RISC-style database system. In *26th International Conference on Very Large Data Bases (VLDB'00)*, pages 1–10, 2000.

[10] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[11] P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Srivastava. Two-dimensional substring indexing. *Journal of Computer and System Science*, 66(4):763–774, 2003.

[12] N. Fuhr and T. Rölleke. HySpirit — A probabilistic inference engine for hypermedia retrieval in large databases. In *6th International Conference on Extending Database Technology (EDBT'98)*, LNCS 1377, pages 24–38, 1998.

[13] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[14] M. Hearst, A. Elliott, J. English, R. Sinha, K. Swearingen, and K.-P. Yee. Finding the flow in web site search. *Communications of the ACM*, 45(9):42–49, 2002.

[15] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ engine: A hybrid IR DB system. In *19th International Conference on Data Engineering (ICDE'03)*, pages 741–, 2003.

[16] N. Lester, J. Zobel, and H. E.Williams. In-place versus re-build versus re-merge: Index maintenance strategies for text retrieval systems. In *27th Australasian Computer Science Conference (ACSC'04)*, 2004.

[17] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In *29th Conference on Research and Development in Information Retrieval (SIGIR'06)*, pages 348–355, 2006.

[18] L. Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, 2000.

[19] S. Raghavan and H. Garcia-Molina. Integrating diverse information management systems: A brief survey. *IEEE Data Engineering Bulletin*, 24(4):44–52, 2001.

[20] S. E. Robertson, S. Walker, M. M. Beaulieu, M. Gatford, and A. Payne. Okapi at TREC-4. In *4th Text Retrieval Conference (TREC'95)*, pages 73–96, 1995.

[21] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for TopX search. In *31st International Conference on Very Large Data Bases (VLDB'05)*, pages 625–636, 2005.

[22] A. Trotman and B. Sigurbjörnsson. Narrowed extended XPath I (NEXI). Available at `http://www.cs.otago.ac.nz/postgrads/andrew/2004-4.pdf`, 2004.