Handbook 0.1 (work in progress)

Author: Edward C. Zimmermann <edz@nonmonotonic.net>

I. Background / History

Isearch was an open-source text retrieval software first developed in 1994 as part of the Isite Z39.50 information framework. The project started at the Clearinghouse for Networked Information Discovery and Retrieval (CNIDR) of the North Carolina supercomputing center MCNC and funded by the National Science Foundation to follow in the track of WAIS (Wide Area Information Server) and develop prototype systems for distributed information networks encompassing Internet applications, library catalogs and other information resources. From 1994 to 1998 most of the development was centered on the Clearinghouse for Networked Information Discovery and Retrieval (CNIDR) in North Carolina and BSn in Germany. By 1998 much of the open-source core developers re-focused development into several spin-offs. In 1998 it became part of the Advanced Search Facility reference software platform funded by the U.S. Department of Commerce.

Isearch was widely adopted and used in hundreds of public search sites, including many high profile projects such as the U.S. Patent and Trademark Office (USPTO) patent search, the Federal Geographic Data Clearinghouse (FGDC), the NASA Global Change Master Directory, the NASA EOS Guide System, the NASA Catalog Interoperability Project, the astronomical pre-print service based at the Space Telescope Science Institute, The PCT Electronic Gazette at the World Intellectual Property Organization (WIPO), [[Linsearch (a search engine for Open Source Software designed by Miles Efron), the SAGE Project of the Special Collections Department at Emory University, Eco Companion Australasia (an environmental geospatial resources catalog), the Open Directory Project and numerous governmental portals in the context of the Government Information Locator Service (GILS) GPO mandate (ended in 2005). A number of sites worldwide continue (despite development long ago suspended) continue to use Isearch in their production systems.

One of the main split-offs was the (closed source and proprietary) IB engine developed by Bsn. With some new algorithms it was deployed in a large number of high profile projects ranging from news search for Wirtualna Polska (one of the largest and most known Web portals in Poland); genomic search for the Australian National Genomic Information Service's human genome project (and its eBiotechnology workbench split-off); the D-A-S-H search portal against racism, antisemitism and exclusion (funded within the framework of the action program "Youth for tolerance and democracy - against right-wing extremism, xenophobia and anti-Semitism", the YOUTH program of the European Community and with additional support from the German Federal Agency for Civic Education); the e-government search (Yeehaw) of the U.S. State of Utah to agronomic cooperation across the Mediterranean region.

Its radical approach and re-think of search was even on display at the ISEA2008: 14th International Symposium on Electronic Art in a collaboration with the Dutch design cooperative Metahaven: https://isea-archives.siggraph.org/art-events/metahaven-exodus-cross-search/. In the words of the project "*Exodus is the compound name for a 'research engine' into algorithms and visual strategies for searching the internet, revealing the structural properties of web content and its inherent distribution of influence. Exodus promotes bridging behaviour across the web's new borders of power.*"

Handbook 0.1 (work in progress)

Development of IB halted in 2011 as its main developers moved on to other projects. While still being deployed by a number of sites it was no longer updated or actively maintained. The software sat idle in the attic for 10 years.

Now through the generous support of the Nlnet Foundation and the European Union’s Next Generation Internet (NGI) initiative its being reborn,open-sourced and renamed as the re-Isearch engine (as a tribute to its roots).

II. Motivation. What does re-Isearch offer?

Organizations of all sizes and within all industries generally distribute their corporate knowledge amid a variety of heterogeneous database applications: from customer relationship systems, staff directories, content management systems (CMS), electronic document and records management systems (EDRMS) to library catalogs.

Mainstream search engines are about finding any information: "a list of all documents containing a specific word or phrase”. Because of this, search engines paradoxically return both too much information (i.e. long lists of links) and too little information (i.e. links to content, not content itself). The re-Isearch engine is, by contrast, about exploiting document structure, both implicit (XML and other markup) and explicit (visual groupings such as paragraph), to zero in on relevant sections of documents, not just links to documents.

Our goal is to provide information search and retrieval services to text, data (a large number of standard types such as numerical, ranges, dates etc), images/video/audio, geographic information, network objects and databases to zero in on and retrieve relevant information.

Key Features and Benefits:

- ➔ Cost effective access to a heterogeneous mix of XML and other data of any shape and size. Allows for the rapid creation of scalable (XML) warehouses.
- ➔ All the capabilities you can ever expect in an enterprise search solution and then some: including phrase, boolean, proximity, wildcard, parametric, range, phonetic, fuzzy, thesauri, polymorphism, datatypes (including numeric, dates, geospatial, ranges etc.) and object capabilities.
- ➔ Relevant ranking by a number of models including spatial score for geospatial queries, date, term frequency, match distribution etc.
- ➔ Object oriented document model: Supports W3C XML, ISO Standard 8879:1986 SGML and a wide range of common file (such as Word, Excel, RTF, PDF, PostScript, HTML, Mail, News), citation (such as BibTex, Endnote, Medline, Papyrus, Refer, Reference Manager/RIS, Dialog, etc), scientific, ISO and industry formats including standards such as USPTO Green Book (patents), DIF (Directory Interchange Format), CAP (Common Alerting Format) and many more.
- ➔ Automatic structure recognition and identification for "unstructured" textual formats (e.g., such as, alongside metadata, lines, sentences, paragraphs and pages in PDF documents).
- ➔ Sophisticated extendable type system allowing for numerical, date, geospatial and other search strategies, including external datastores and brokers parallel to textual methods: "Universal Indexing".
- ➔ Synchronized information: As soon as context its indexed (appended) its available. Functional Append/Delete/Modify and transaction-consistent revision information deliver consistence and up-to-date information without the time-lag typical of many search engines.

Handbook 0.1 (work in progress)

- ➔ True search term highlighting (exactly what the query found, structure etc.) including Adobe Acrobat PDF Highlighting.
- ➔ Extendable/Embeddable/Programmable: Java, Python, Tcl, C++ and other other language APIs.
- ➔ Support for a number of information retrieval protocols including ISO 23950 / ANSI NISO Z39.50, SRW/U and OpenSearch.
- ➔ Runs on a wide range of hardware and operating systems.
- ➔ Easy to maintain, tiny, scalable and fast. Energy efficient: One can start off with inexpensive low-power hardware (our Blau Appliance, for example, draws only 10 watts of power and is sufficient to handle several concurrent user sessions searching GB of data and still deliver search performance measured in fractions of a second.).
- ➔ Does not demand advance setup or pre-processing.
- ➔ Unlike most search engines it is not based on "Inverted file indexes". Because of the limitation of "inverted indexes" most search engines typically index text (excluding common words and long terms as "stop words") and only a fixed and limited number of (defined at indexing time), additional fields (since they are expensive). Out aim is to provide unlimited query flexibility without having to know in advance what questions users are going to ask.
- ➔ Unlike databases we don’t require conversion into a "common format" with a schema set in concrete.
- ➔ Unlike XML databases we support also non-hierarchical structures and overlap.
- ➔ Unlike search engines we can index all elements, their structure, and their contents. This means that one can quickly evaluate text queries, structural queries, and queries that combine both text, objects (numerical, geospatial etc.) and structural constraints (e.g.,find diagram captions that mention engine in articles whose title contains Airbus).
- ➔ Virtual "indexes" allow for the design of logically segmented information indexes and fast on-demand search of arbitrary combinations thereof. Via the field and path mapping architecture this can be implemented completely transparent to search.
- ➔ Index collection binding: multiple indexes can be imported into an index. This allows for the custom creation of indexes on the basis of a large catalog of indexes— highly relevant to publishers as their customers tend to subscribe to only a sub-set of products (e.g. journals).
- ➔ Full ability to search specific structure/context in information without even knowing their details (such as tag or field names).
- ➔ User defined "search time" unit of retrieval: the structure of documents can be exploited to identify which document elements (such as the appropriate chapter or page) to retrieve. No need for intermediate documents or re-indexing.
- ➔ No need for a “middle layer” of content manipulation code. Instead of getting URLs from a search engine, fetching documents, parsing them, and navigating the DOMs to find required elements, it lets you simply request the elements you need and they are returned directly.
- ➔ "Any-to-Any" architecture: On-the-fly XML and other formats.

The default modus is to index all the words and all the structure of documents. It provides powerful and fast search without prior knowledge about the content yet enables arbitrarily complex questions across all the content and from different perspectives. Not bound by the constraints of "records" as unit of information, one can immediately derive value from content with the flexibility to enhance content and the application incrementally over time without "breaking anything".

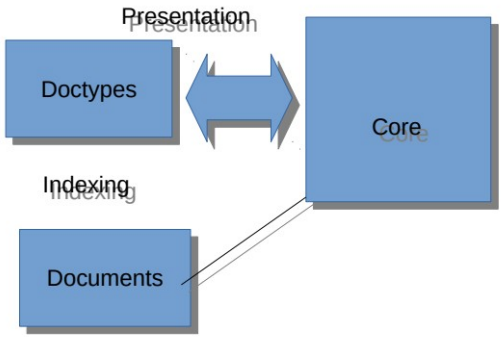
Handbook 0.1 (work in progress)

III. Design

The **Core** engine contains all the classes and methods (C++) to ingest documents, parse them, create indexes, store objects and provide search thereto. On a higher metalevel we have the engine kernel which provides the core indexing, search, retrieval and presentation services, manages objects and dispatches to handlers.

Datatypes (object data types handled polymorphic to text)

Data types are handled by the core and extendable within code. Many of these are well known from standard schemas such as string, boolean, numerical, computed, date and many more motivated by user needs such as phonetic types. Milestone 1 (CORE) shall contain a large assortment including the possibility to install a callback and some local types. A list and some documentation of available types is available at runtimes as the system contains a rudimentary self documentation of the installed handlers.

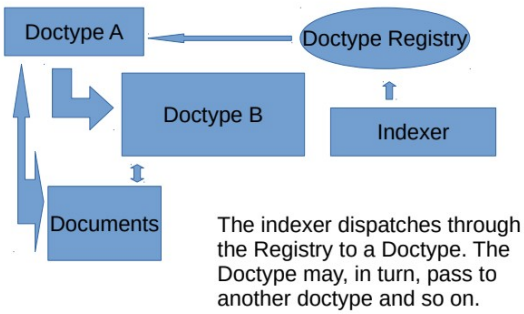


Object Indexes (data type handlers)

The indexer supports a number of datatypes. For example: strings (full text), numerical (IEEE floating), computed numerical, range of numbers, date, date-range, Geospatial n-ary bounding coordinates etc. This is handled by a data type registry. All data is stored as string (the octets defining the words/terms, e.g. xs:string or xs:normalizedString in XML schema jargon) and, if specified or determined, the specific object type. These different data types have their own for-purpose indexing structures.
Some have also very special functions. The datatype TTL, for example, is a numeric computed value for time-to-live in seconds. The same structures are used for another datatype called „expires“ but with the time-to-live associated with the record to define record expiration. There are also a number of datatypes that use hash structures such as sound or phonetic hashes which have proven useful in name searches. All datatypes carry their own most basic documentation into the registry.

Doctypes (Document handlers)

Services to handle the various document formats (ingest, parse, recognize start and end of records with multi-record file formats, recognize start and end of fields, decode encodings, convert and present) are handled by a so-called “doctype” system. These doctypes are built upon a base DOCTYPE class. They are managed and dispatched by a registry. All doctypes carry their own basic documentation (and tree heirarchy) when they register into the registry. We have both a growing collection of “built-in” types (provided with the core engine and covered by the same license and conditions as the rest of the engine) and “plugin-in” types. The latter are loadable at run-time. These loadable plug-ins can handle all or just partial services and are not just descendents of a built-in document handler class but also can pass control to other types not immediately in its class



path. Milestone 1 shall contain some 60 formats including several multi functional that transparently use confiigurable external conversion tools—for example OCR to process scanned documents or an image captioning tool to process photographs alongside the embedded metadata is readily implemented. All parsers have some self documentation available at runtime of their options and class tree.

Field Unitification (indexing/search)

Since the engine is designed to support a wide range of heterogenous documents and record formats a facility was developed to allow for name, resp. query path alignment. Defined by a configuration file, field names (and paths) can to be mapped during indexing to alternative names. This is intended to handle the issue of semantically equivalent field (or path) contents having syntactically different names (or paths). It also allows one to skip fields (for example P=<Ignore>). These settings may be defined unique to doctype, to doctype instance or to database (e.g. the search indexing target).

Ranking (search results)

The set of elements (records) on a search response tends to be sorted by scores—but there are a number of other sorting methods available. Scores are normalized by a number of methods (that impact the sort). The standard methods in the core engine are NoNormalization, CosineNormalization (Salton Tf-idf), MaxNormalization (weighted Cosine), LogNormalization (log cosine score), BytesNormalization (normalize frequency with a bias towards terms that are closer to one another) and CosineMetricNormalization.

Scores and ranking can also be boosted by a number of progressions: Linear, Inversion, LogLinear, LogInversion, Exponential, ExpInversion, Power and PowInversion.

Both sorting and normalization algorithms are user extendable and designed for customizations.

Queries can also weight various matches or field results to give them more importance (or less). There are also methods to cluster or shift position in results ranking with hits (matches) that are closer to one another („magnitism“).

Presentation (retrieval)

For presentation the engine uses something called “Record Syntax” to define the response syntax either through reconstruction, reconstitution or transformation. Like datatypes and doctypes it too is handled by an extensible registry. These are defined by internally registered OIDs. A ITU-T / ISO/IEC object identifier (OID) is an extensively used identification mechanism. It is the job of the DOCTYPE subsystem (using the doctype associated with the record whose data the response contains either partially or in full) to build the appropriate reponse as requested. It is typically built as a reconstruction using the indexed structure and addresses of content. This allows one to control the final reconstruction to exclude sensitive information that might have been in the original record but to be excluded from some presentations.

Handbook 0.1 (work in progress)

IV. SearchIB Query Language

All parts of the query language are case insensitive apart from terms. Fields (paths) are case insensitive. While XML, for example, is case-sensitive, we are case insensitive. Should the same element name have different semantics (generally a very bad practice) by case in the source it needs to be converted (e.g. with a prefix).

Queries to the engine are done by a number of means: *1)* RPN expressions *2)* Infix (algebraic) *3)* Relevant feedback (a reference to a fragment) *4)* so called **smart** plain language queries *5)* C++ *6)* Via a bound language interface in Python or one of the other SWIG supported languages (Go, Guile, Java, Lua, MzScheme, Ocaml, Octave, Perl, PHP, R. Ruby. Scilab. Tcl/Tk).

Smart queries try to interpret the query if its RPN or Infix or maybe just some terms. The logic for handling just terms is as-if it first searched for a literal expression, if not then tying to find the terms in a common leaf node in a records DOM, if not then AND‘d (Intersection of sets), if not then OR‘d (Union) but reduced to the number of words in the query.

Example: Searching in the collected works of Shakespeare:

(a) `rich water`

It find that there are no phases like “rich water” but in the `The Life of Timon of Athens' it finds that both the words “rich” and “water” are in the same line: “.. And rich: here is a water, look ye..”. The query is confirmed as `"rich" "water" PEER` (see binary operators below)

(b) `hate jews`

It finds that there are no phrases like “hate jews” but in the `The Merchant of Venice' we have in act lines that both talk about “hate” and “jews”. The smart query gets confirmed as `"hate" "jews" || REDUCE:2` (see unary operators below)
with the result “.. I hate him for he is a Christian ..” found in PLAY\ACT\SCENE\SPEECH\LINE

(c) `out out`

It finds a number of lines where “out out” is said such as “Out, out, Lucetta! that would be ill-favour'd” in `The Two Gentlemen of Verona'. The confirmed query is: `“out out”`.

While „Smart“ searches is fantastic for many typical use cases (and why we developed it), power users tend to want to perform more precise queries. The implemented infix and RPN languages support fields and paths (like Xquery) as well as a very rich set of unary and binary operators and an assortment of modifiers (prefix and suffix). Since the engine supports a number of data types it includes a number of relations /.<,>,>=,<=,<> whose semantics of depends upon the field datatype. These operators are all overloaded in the query language.

Terms are constructed as **`[[path][relation]]searchterm[:n]`**

Example: `design`

This is the most basic search. Example `“design”` to find “design” anywhere. Or `title/design` to find “design” just in `title` elements. Since paths are case-insensitive there is no difference between `Title/design`, `tITle/design` and `title/design`.

Handbook 0.1 (work in progress)

`Example: title/design`

Paths can be from the root '\' (e.g. *record**metadata*\title) or from any location (e.g. title or metadata\title)). Paths can be specified with '/' or '\' but one needs in the `field/term` format to be quite careful (with quotes) to delineate the field from the term.

Another way to approach the problem of searching an element (RPN):

Is to use special unary operator called WITHIN:path. `Searching Design WITHIN:title`
is equivalent to title/Design but often more convenient, especially where for the element we have a path. Instead of `\\A\\B\\C\\D\\E/"word"` (since we need to escape the \) we can use the expression: `"word" WITHIN:/A/B/C/D/E`

Note: the expressions:

```
term1 term2 && WITHIN:title
```

and

```
term1 WITHIN:title term2 WITHIN:title &&
```

yield the same results as they are both looking for term1 and term2 in the title element BUT they have different performance. The first expression can be slower since it builds the set for term1 and the set for term2 and then reduces the intersection to a new set that contains only those hits where term1 and term2 were within the title element. The second is taking an intersection of two potentially smaller sets. It is generally faster and better.

Term and field paths support "Glob" matching

We support left (with some limitations in terms) and right (without limitations) truncated search as well as a combination of * and ? for glob matching. This can be universally applied to path (fieldname) and with some limitations to searchterm. These can be connected by more than a dozen binary as well as a good dozen unary relational operators.

Wildcard	Description	Example	Matches
*	matches any number of any characters	Law*	Law, Laws, or Lawyer
?	matches any single character	?at	Cat, cat, Bat or bat
[abc]	matches one character given in the bracket	[CB]at	Cat or Bat
[a-z]	matches one character from the (locale-dependent) range given in the bracket	Letter[0-9]	Letter0, Letter1, Letter2 up to Letter9
{xx,yy,zz}	match anyof "xx", "yy", or "zz"	{C,B}at	Cat or Bat

Example: `t*/design would if there was only one field starting with the letter t and it was title search for title/design.`

Glob in path names is quite powerful as it allows one to narrow down search into specific elements.

For every hit we can also examine where it occurred. Example: Searching for
Example: `PLAY\ACT\SCENE\SPEECH\LINE`

V. Query Operators (IB Language)

The re-Isearch engine has been designed to have an extremely rich and expressive logical collection of operators. Some operators can, however, be quite expensive. The complement, for example, of a set with a single result in a large dataset is large. Search time is directly related to the time to build the result set.

Binary Operators

Polymorphic Binary Operators	
>	Greater than
<	Less than
<=	Less than or equal to
>=	Greater then or equal to

Long Operator Name	Sym	Description
OR	||	Union, the set of all elements in either of two sets
AND	&&	Intersection, the set of all elements in both sets
ANDNOT	&!	Elements in the one set but NOT in the other
NOTAND	!&	As above but operand order reversed
NAND	&!	Complement of AND, elements in neither
XOR	^^	Exclusive Union, elements in either but not both
XNOR	^!	Exclusive not OR, complement set of XOR
PROX:num		PROX:0 := ADJ. PROX:n := NEAR:n
NEAR[:num]		matching terms in the sets are within num bytes in the source
BEFORE[:num]		As above but in order before
AFTER[:num]		As above but in order after
DIST[>,>=,<,<=]num		Distance between words in source measured in bytes. (and order)
num > 1: integer for bytes. As fraction of 1: % of doc length (in bytes).		
NEIGHBOR	.~.	

Handbook 0.1 (work in progress)

PEER	.=.	Elements in the same (unnamed) final tree leaf node
PEERa		like PEER but after
PEERb		like PEER but ordered after
XPEER		Not in the same container
AND:field		Elements in the same node instance of field
BEFORE:field		like AND:field but before
AFTER:field		like AND:field but after
ADJ	##	Matching terms are adjacent to one another
FOLLOWS	#>	Within some ordered elements of one another
PRECEDES	#<	Within some ordered elements of one another
PROX		Proximity
FAR		Elements a "good distance" away from each other
NEAR	.<.	Elements "near" one another.

Since the engine produces sets of result we can also combine two sets from different database searches into a common set as either augmentation or by performing some operations to create a new set.

Operators that can act on result sets from searching different databases (using common keys)	
JOIN	Join, a set containing elements shared (common record keys) between both sets.
JOINL	Join left, a set containing those on the right PLUS those on the left with common keys
JOINR	Join right, a set containing those of the left PLUS those on the right with common keys

Unary Operators

Operator	**Sym**	**Description**
NOT	!	Set compliment
WITHIN[:field]		Records with elements within the specified field. RPN queries "term WITHIN:field" and "field/term" are equivalent. (for performance the query "field/term" is preferred to "term WITHIN:field")
WITHIN[:daterange]		Only records with record dates within the range
WITHKEY:pattern		Only records whose key match pattern
SIBLING		Only hits in the same container (see PEER)
INSIDE[:field]		Hits are limited to those in the specified field
XWITHIN[:field]		Absolutely NOT in the specified field
FILE:pattern		Records whose local file path match pattern
REDUCE[:nnn]		Reduce set to those records with nnn matching terms This is a special kind of unary operator that trims the result to metric cutoff

		regarding the number of different terms. Reduce MUST be specified with a positive metric and 0 (Zero) is a special case designating the max. number of different terms found in the set.
HITCOUNT:nnn		Trim set to contain only records with min. nnn hits.
HITCOUNT[>,>=,<,<=]num		As above. Example: HITCOUNT>10 means to include only those records with MORE than 10 hits.
TRIM:nnn		Truncate set to max. nnn elements
BOOST:nnn		Boost score by nnn (as weight)
SORTBY:<ByWhat>		Sort the set "ByWhat" (reserved case-insensitive names: "Key", "Hits", "Date", "Index", "Score", "AuxCount", "Newsrank", "Function", "Category", "ReverseHits", "ReverseDate", etc.)

SortBy:<ByWhat>

The SORTBY unary operator is used to specify a specific desired sort of the result set upon which it applies. It is used to sort (ByWhat keywords are case insensitive) by

- ➔ “Score” → Score (the default). The score, in turn, depends upon the selected normalization algorithm.
- ➔ “Key” → Alphanumeric sort of the record key.
- ➔ “Hits” → Number of hits (descending)
- ➔ “ReverseHits” → Number of hits (ascending)
- ➔ “Date” → Date (descending)
- ➔ “ReverseDate” → Date (ascending)
- ➔ “Index” → Position in the index
- ➔ “Newsrank” → Newsrank: a special mix of score and date
- ➔ “Category” → Category
- ➔ “Function” → Function
- ➔ “Private1” , “Private2” .. → A number of private sorting algorithms (installable) that use private data (defined by the algorithm) to perform its sort. This is quite site specific and “out-of-the-box” is not defined.
  - ➔ If the private sort handler is not installed (defined) it defaults to sorting by score (ascending).

Queries can also weight various matches or field results to give them more importance (or less). There are also methods to cluster or shift position in results ranking with hits (matches) that are closer to one another („magnitism“).

RPN vs Infix

Internally all expressions are converted into a RPN (Reverse Polish Notation) and placed on stacks.
In reverse Polish notation, the operators follow their operands; for instance, to add 3 and 4 together, one would write 3 4 + rather than 3 + 4. The notation we commonly use in formulas is called “infix”: the binary operators are between the two operands—and unary before—and groups are defined by parenthesis. RPN has the tremendous advantage that it does not need parenthesis or other groupings. It also does not need to worry about order and precedence of operations. In infix expressions parentheses surrounding groups of operands and operators are necessary to indicate the intended order in which operations are to be performed. In the absence of parentheses, certain precedence rules

determine the order of operations (such as in the grade school mantra “Exponents before Multiplication, Multiplication before Addition”).

While Infix is generally more familiar (except perhaps users of HP Scientific calculators) with a bit of practice the RPN language is generally preferred for it clarity and performance.
Result sets from searches on terms on the stack are cached to try to prevent repeated searches. Caches may also be made persistent by using disk storage. This is useful for session oriented search and retrieval when used in stateless environments.

Since it is a true query language it is possible to write extremely ineffective and costly queries. There is a facility to give search expressions only a limited amount of „fuel“ to run. One can also explicitly select to use these partial results.

Example RPN	Example Infix
title/cat title/dog || title/mouse ||	title/cat || title/dog || title/mouse
	title/(“cat” || “dog” || “mouse”)
speaker/hamlet line/love AND:scene	(speaker/hamlet and:scene line/love)
out spot PEER	out PEER spot
from/edz 'subject/"EU NGI0"' &&	from/edz && 'subject/"EU NGI0"'

Comparison to CQL

CQL query consist, like the IB language, of either a single search clause or multiple search clauses connected by boolean operators. It may have a sort specification at the end, following the 'sortBy' keyword. In addition it may include prefix assignments which assign short names to context set identifiers.

Re-Isearch Expression	CQL Expression
dc.title/fish	dc.title any fish
dc.title/fish dc.creator/sanderson OR	dc.title any fish or dc.creator any sanderson

Handbook 0.1 (work in progress)

Programming Languages

Since the internal representation of a query is a RPN stack and we have a number of programming interfaces (C++, Python, Java, PHP, Tcl, etc.) we have the tremendous power to express and store what we wish. At current we don't have a SQL or SPARQL interface but it should be relatively easy for a contributor to write in Python (or one of the other languages).

Python

In Python one can build a query like:

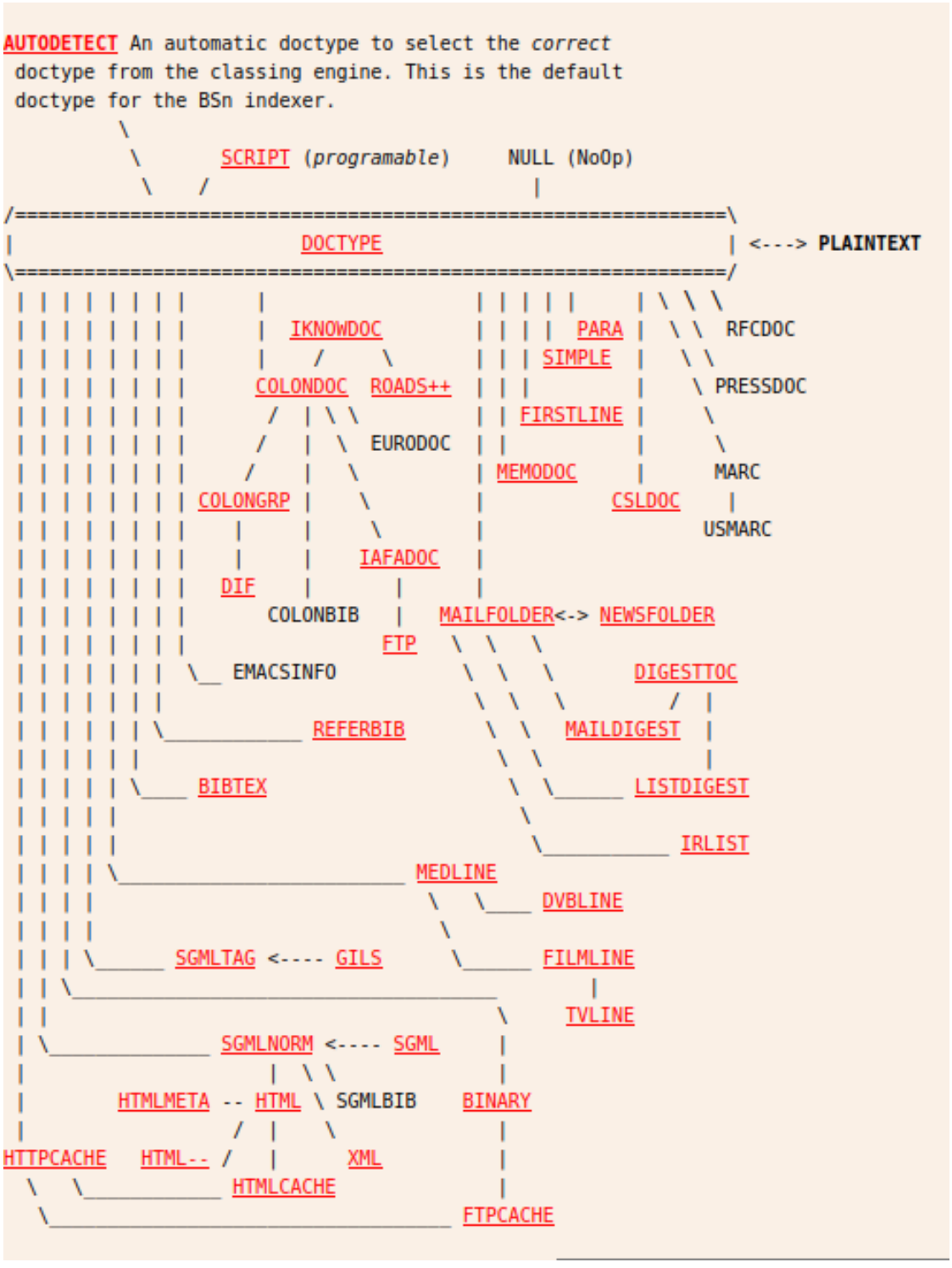
```
query = "beschaffungsmanagement:3 OR beschaffungsmarketing:3 OR beschaffungsmarkt:3
OR beschaffungsplanung:3 OR beschaffungsprozesse:3 OR (deterministische:3 FOLLOWS
beschaffung:3) OR einkaufspolitik:3 OR (stochastische:3 FOLLOWS beschaffung:3) OR
strategien:2 OR strategie OR (c:3 FOLLOWS teilemanagement:3) OR
beschaffungsmarktforschung:3 OR (double:4 FOLLOWS sourcing:4) OR (global:4 FOLLOWS
sourcing:4) OR (modular:4 FOLLOWS sourcing:4) OR (multiple:4 FOLLOWS sourcing:4) OR
(single:4 FOLLOWS sourcing:4) OR sourcing:3 OR methoden:2 OR methode OR lieferant:3
OR lieferanten:2 OR logistikdienstleister:3 OR rahmenvertraege:3 OR tul:4 OR
spediteur:3 OR spediteure:2 OR spediteuren:2 OR spediteurs:2 OR stammlieferant:3 OR
vertraege:3 OR vertrag:2 OR vertraegen:2 OR vertrages:2 OR vertrags:2 OR
zulieferpyramide:3 OR partner:2 OR partnern OR partners OR beschaffungskosten:3 OR
einkaufscontrolling:3 OR einkaufsverhandlungen:3 OR incoterms:3 OR
wiederbeschaffungszeit:3 OR zahlungskonditionen:3 OR konditionen:2 OR kondition OR
einfuhr:3 OR einfahre:2 OR einfahren:2 OR einfahrend:2 OR einfahrest:2 OR
einfahret:2 OR einfahrt:2 OR einfuehrt:2 OR einfuehre:2 OR einfuhren:2 OR
einfueren:2 OR einfuehrest:2 OR einfuehret:2 OR einfuhrst:2 OR einfuhrt:2 OR
eingefahren:2 OR einzufahren:2 OR eust:4 OR einfuhrumsatzsteuer:3 OR inbound:3 OR
jis:4 OR (just:3 FOLLOWS in:3 FOLLOWS sequence:3) OR jit:4 OR (just:3 FOLLOWS in:3
FOLLOWS time:3) OR sendungverfolgung:3 OR stapler:3 OR staplern:2 OR staplers:2 OR
we:4 OR wareneingang:3 OR wa:4 OR warenausgang:3 OR wareneingangskontrolle:3 OR
zoll:3 OR zoelle:2 OR zoellen:2 OR zolles:2 OR zolln:2 OR zolls:2 OR gezollt:2 OR
zolle:2 OR zollen:2 OR zollend:2 OR zollest:2 OR zollet:2 OR zollst:2 OR zollt:2 OR
zollte:2 OR zollten:2 OR zolltest:2 OR zolltet:2 OR zollware:3 OR transport:2 OR
transporte OR transporten OR transportes OR transports"
db_path="/var/opt/nonmonotonic/NEWS";
pdb = IDB(db_path); ## Open the index
squery = SQUERY(query); # Build the query
irset = pdb.Search(squery, ByScore); # Run the query
## The resulting irset is a set which we can perform operations upon or combine with another irset
## from another search using the operators in the tables above—for  example Or, Nor, And, ….
irset = irset1.And(irset2); ## This is like irset = irset1 AND irset2
```

As one can see it is relatively straightforward to build alternative query languages to run.

Handbook 0.1 (work in progress)

Doctypes

This is an old map of the doctypes (not much has changed)



Handbook 0.1 (work in progress)

```
The current collection (June 2021):
Available Built-in Document Base Classes (v28.5):
        AOLLIST           ATOM     AUTODETECT       BIBCOLON
         BIBTEX         BINARY            CAP       COLONDOC
       COLONGRP       DIALOG-B            DIF        DVBLINE
        ENDNOTE      EUROMEDIA       FILMLINE    FILTER2HTML
    FILTER2MEMO    FILTER2TEXT     FILTER2XML      FIRSTLINE
            FTP           GILS        GILSXML        HARVEST
           HTML         HTML--      HTMLCACHE       HTMLHEAD
       HTMLMETA     HTMLREMOTE       HTMLZERO        IAFADOC
       IKNOWDOC         IRLIST        ISOTEIA     LISTDIGEST
     MAILDIGEST     MAILFOLDER        MEDLINE           MEMO
        METADOC       MISMEDIA     NEWSFOLDER         NEWSML
            OCR            ODT        ONELINE       OZSEARCH
        PAPYRUS           PARA            PDF      PLAINTEXT
             PS          PTEXT            RDF       REFERBIB
            RIS        ROADS++         RSS.9x           RSS1
           RSS2     RSSARCHIVE        RSSCORE           SGML
       SGMLNORM        SGMLTAG         SIMPLE           SOIF
         TSLDOC            TSV        XBINARY        XFILTER
            XML        XMLBASE        XPANDOC      YAHOOLIST
External Base Classes ("Plugin Doctypes"):
  RTF:               // "Rich Text Format" (RTF) Plugin
  ODT:               // "OASIS Open Document Format Text" (ODT) Plugin
  ESTAT:             // EUROSTAT CSL Plugin
  MSOFFICE:          // M$ Office OOXML Plugin
  USPAT:             // US Patents (Green Book)
  ADOBE_PDF:         // Adobe PDF Plugin
  MSOLE:             // M$ OLE type detector Plugin
  MSEXCEL:           // M$ Excel (XLS) Plugin
  MSRTF:             // M$ RTF (Rich Text Format) Plugin [XML]
  NULL:              // Empty plugin
  MSWORD:            // M$ Word Plugin
  PDFDOC:            // OLD Adobe PDF Plugin
  TEXT:              // Plain Text Plugin
  ISOTEIA:           // ISOTEIA project (GILS Metadata) XML format locator
records
```

AUTODETECT:

Lets start off the with AUTODETECT type as it's often the go-to default. Class Tree:

DOCTYPE

v

AUTODETECT

Handbook 0.1 (work in progress)

AUTODETECT is a special kind of doctype that really isn't a doctype at all. Although it is installed from the viewpoint of the engine as a doctype in the doctype registrary, it does not handle parsing or presentation and only serves to map and pass responsibility to other doctypes. It uses a complex combination of file contents and extension analysis to determine the suitable doctype for processing the file.

The identification algorithims and inference logic have been designed to be smart enough to provide a relatively fine grain identification. The analysis is based in large part upon content analysis in contrast to purely magic or file extension methods. The later are used as hints and not for the purpose of identification. These techniques allow the autodetector to distinguish between several different very similar doctypes (for example MEDLINE, FILMLINE and DVBLINE are all based upon the same basic colon syntax but with slightly different features). It allows one to index whole directory trees without having to worry about the selection of doctype. It can also detect many doctypes where there are, at current, no suitable doctype class available or binary files not probably intended for indexing (these include misc files from FrameMaker, SoftQuad Author/Editor, TeX/METAFONT, core files, binaries etc). At current ALL doctypes available are identified. For doctypes that handle the same document formats but for different functions (eg. HTML, HTML-- and HTMLMETA) given that being logical does not mean it can read minds. For these one must specify the document parser or the most general default parser would be chosen (eg. HTML for the entire class of HTML files).

Should the document format not be recognized by the internal logic it then appeals to, should it have been built with it (its optional) libmagic. That library has a user editable magic file for identification. If the type is identified as some form of "text", viz. not as some binary or other format, then it is associated with the PLAINTEXT doctype.

Since it has proved accurate, robust and comfortable it is the default doctype.

Options in .ini:

```
[General]
Magic=<path>     # Path of optional magic file
ParseInfo=[Y|N]  # Parse Info for binary files (like images)

[Use]
<DoctypeClass>=<DoctypeClassToUse> # example HTML=HTMLHEAD
<DoctypeClass>=NULL  # means don't index <DoctypeClass> files
```

Handbook 0.1 (work in progress)