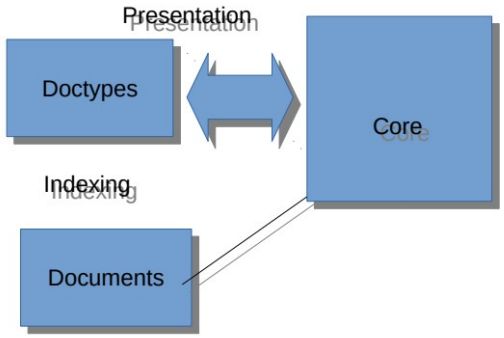
Whitepaper v.2.91 (work in progress)

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

Architecture (Brief sketch to understand the engine):

The **Core** engine

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.



Basic Text Search Algorithm

The engine for text does not use a conventional word-level inverted index.

Normative inverted index structures are composed of two elements: the vocabulary and the occurrences. The vocabulary is the set of all different words in the text. By its very nature it needs to limit the length of possible words. For popular inverted indexes this is typically some length under 255 (some even under 64). For each word in the vocabulary the index stores the documents which contain that word (and perhaps its location either on word or block level). These data structures tend to be quite compact. Using document addressing indexes can require only 20-40% of text size. Since frequent occurances demand disproportionate space (and also slow the system) they tend to use stopword lists (or frequency limits) to exclude frequent words from the index. Word addressing also significantly increases size so many system keep to document addressing and build individual inverted indexes for each field to be searched in structured documents. Word addressing and especially individual inverted indexes have a serious impact on indexing performance so it is generally best practice with inverted indexes to try to keep these to a minimum.

Limiting the length of words in the vocabulary might make perfect sense for the english language but in some languages like German with its myriad of compound words it can place limitations. With 80 characters the German word „Donaudampfschifffahrts-elektrizitätenhauptbetriebswerkbauunterbeamtengesellschaft“, the 63-character „Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz“ or even the 36-character „Kraftfahrzeug-Haftpflichtversicherung“ surpases the length of the most popularly cited long word in the English language „antidisestablishmentarianism“. These are all dwarfed by the 189819 character „methionylthreonylthreonylglu…“

Excluding stopwords too can make sense but sometimes these frequent words are significant. Beyond the often cited „To be or not to be“ we have in some mixed language text words that can fall through the cracks. Words that one might argue are frequent in one but of profound significance in others are not rare. The word „war“ (in German equivalent to English „was“ which itself in German is the equivalent of „what“) for example. We also have words that are commonly found in stopword lists that might be extremely significant, for example „IT“ (commonly used for information technology), „WHO“ (which can also refer to the World Health Organization), etc.

Limiting the structure to a few fields can in many use cases make sense but part of the ambitions of the NoSQL movement are to not require any predefined schemas and to be able to search for things that one did not design apriori for. Demanding a choice on structure seem to go against the

desires for a highly flexible data model that doesn't need, as with relational databases, to be immediately defined.

So with an aim to allowing for utmost freedom to work with a collection of hetrogeneous structured and "unstructured data" **we choose another technological approach**. Instead of building a graph layer on-top of an inverted index, we implement an algorithm inspired by Gaston H. Gonnet and Ricardo A. Baeza-Yates' NEW INDICES FOR TEXT: PAT TREES AND PAT ARRAYS. Text input is viewed as a semi-infinite string (stream of characters). This delivers immediately an address for every word (the address encodes both the identity of the file input and its offset in the file) and does not need to maintain a catalogue of terms (index). The disadvantage is its poor I/O. The documents need to be opened and read for each work during search. We address this with a simple yet powerful trick: we cache in a kind of index the first n-characters of unique words and map that to adresses in our table that organizes terms to addresses in the individual records. This affords a search speed nearly en-par with an inverted index but without its significant downsides. Performance speed is effectively I/O constrained, limited primarily by computer memory and disk storage speed-- faster RAM and faster SSDs lead to faster performance more than faster CPUs which allow for clusters to be built with low cost/low power CPUs. Since we are only caching the first n-characters we can index words of unlimited length. Since we have a fixed length we can map the SIS cache directly into virtual address space and perform „in-memory" binary search. Since we map into an index of addresses we have no limitations on the frequency of terms. The space overhead is just its address (either 32-bit or 64-bit encoded). When the term length exceeds the length of the cache we open the file and read the term. To keep these system calls to a minimum we keep a cache of open file handles and memory map pointers. This typically results in the terms often being already in memory. Since we store field information in yet another structure which just carries sorted start-end addresses we have no significant overhead for additional fields and can support complex structures and even overlap. While field search performance has a penalty compared to building an inverted index for each field since we have a complete address map we can walk through all the trees and at search time explore interdepencies at will. This lets us, for example, search for term in the same unnamed attribute (field) instance without knowing its path.

Since we are indexing records as parts of collections literally viewed as an infinite stream of characters, what constitutes a record and what can constitute a unit of retrieval can be quite dynamic. Allowing member records of a collection to have a shared root key we can find both collections as a search response but also digging deeper into the structure of the search (where the terms are located in a record) find relevant sub-elements. This allow for the development of search interfaces and applications that break with the mould of record or document level retrieval.

Datatypes (object data types handled polymorphic to text)

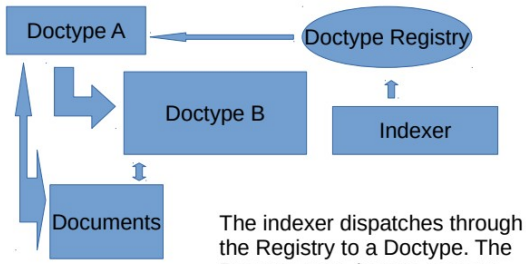
Data types are handled by the core and extendable within code. Many of these are well known from standard schemas such as string, boolean, numerical, computed, date and many more motivated by user needs such as phonetic types. Milestone 1 (CORE) shall contain a large assortment including the possibility 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)



The indexer dispatches through the Registry to a Doctype. The Doctype may, in turn, pass to another doctype and so on.

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 multifunctional 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.

Doctype functions

A doctype needs to parse a document into records. Frequently documents consist of a single record but many don‘t. Many doctypes perform also an examination of the document or record to see if they might want to pass it to another doctype with more intimate knowledge about the format it thinks it may have detected. One doctype AUTODETECT is only about auto-detecting the input format. Doctypes exploit the registry to pass the input to another class. Somewhere down the chain a doctype takes responsibility—if only to process metadata should it fall throught the raster. The individual records are in turn parsed into their fields or attribute containers. It is also the job of a doctype to be able to read and decode the text stream in a record. For HTML, for example, this means decoding the HTML character entities. Some types may need to perform pre-processing and create an intermediate. SGML, for exmple, needs to be normalized. PDF needs to have its text rendered.

During search the doctypes continue to have jobs. They must be able to read the documents and prepare fragments to be compared in a suitable manner. With HTML this means, among other things, with its HTML character entities decoded and text normalized. It is also responsible for retrieval of information (document, record or parts thereof) or what we call presentation.

Whitepaper v.2.91 (work in progress)

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, fieldnames (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 syntatically 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).

Query Engine (search)

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. 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).

While „Smart" searches is for many normal users, power users tend to want to perform more precise queries. The iimplented nfix 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]. We support right and left truncated search as well as a combination of * and ? For glob matching. This can be applied to path (fieldname) and searchterm alike. These can be connected by more than a dozen binary as well as a good dozen unary relational operators.

Internally all expressions are converted into a RPN (Reverse Polish Notation) and placed on stacks. 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.

Ranking (search results)

The set of elements (records) on a search response tends to be sorted by scores. There are a number of algorithms that tend to be used for scoring. 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. There is also normalization using a date bias (Newsrank).

Scores and ranking can also be boosted by a number of progressions Linear, Inversion, LogLinear, LogInversion, Exponential, ExpInversion, Power and PowInversion.
Scoring algorithms are user extendable as 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.

OIDs (ITU-T / ISO/IEC object identifiers)

We use OIDs (Object Identifiers) throughout instead of ad-hoc names for a number of request and response features to allow for easier integration with compliant systems. Example: For PDF we have the octet stream: 1.2.840.10003.5.109.1 while for SUTRS (effectively plain structured text) we have 1.2.840.10003.5.101

This is in the tree 1.2.840.10003 registered with ISO as the Z39-50 standard from ANSI (which has been also adopted by ISO 23950). The sub-node 5 is for Record Syntaxes (presentation). 101 is for SUTRS. 109 is for MIME types with 109.1 for PDF, 109.2 for Postscript, 109.3 for HTML etc.
The use of the Z39.50 OID space is for rational, practical and historical reasons. Firstly ISO 23950 (identical in text to ANSI/NISO Z39.50-1995 except for certain style discrepancies between ISO and ANSI standards) remains the lingua franca for library systems, OPACS (Online Public Access Catalogues) and information federations (such as the various Geospatial data clearinghouses) worldwide. When we designed the standard within the ZIG (Z39.50 Interest Group) many of the features where designed to go beyond the library paradigm and for use in general search and retrieval within a range of federated architectures.

Whitepaper v.2.91 (work in progress)

IB 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: title/design`

Paths can be from the root ‘/’ (e.g. *record/metadata/*title/) or from any location (e.g. title/ or metadata/title/)). To escape the special ‘/’ we use ‘\’, e.g \/ to have ‘/’ without its special significance.

Whitepaper v.2.91 (work in progress)

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`.

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`

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.

Whitepaper v.2.91 (work in progress)

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 OR title/mouse OR	title/(“cat” or “dog” or “mouse”)
speaker/hamlet line/love AND:scene	(speaker/hamlet and:scene line/love)
out spot PEER	out PEER spot
from/edz 'subject/"EU NGI0"' AND	from/edz AND 'subject/"EU NGI0"'

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

Whitepaper v.2.91 (work in progress)

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	.~.	
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 names: Date, Score, Hits, etc.)

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

Whitepaper v.2.91 (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:

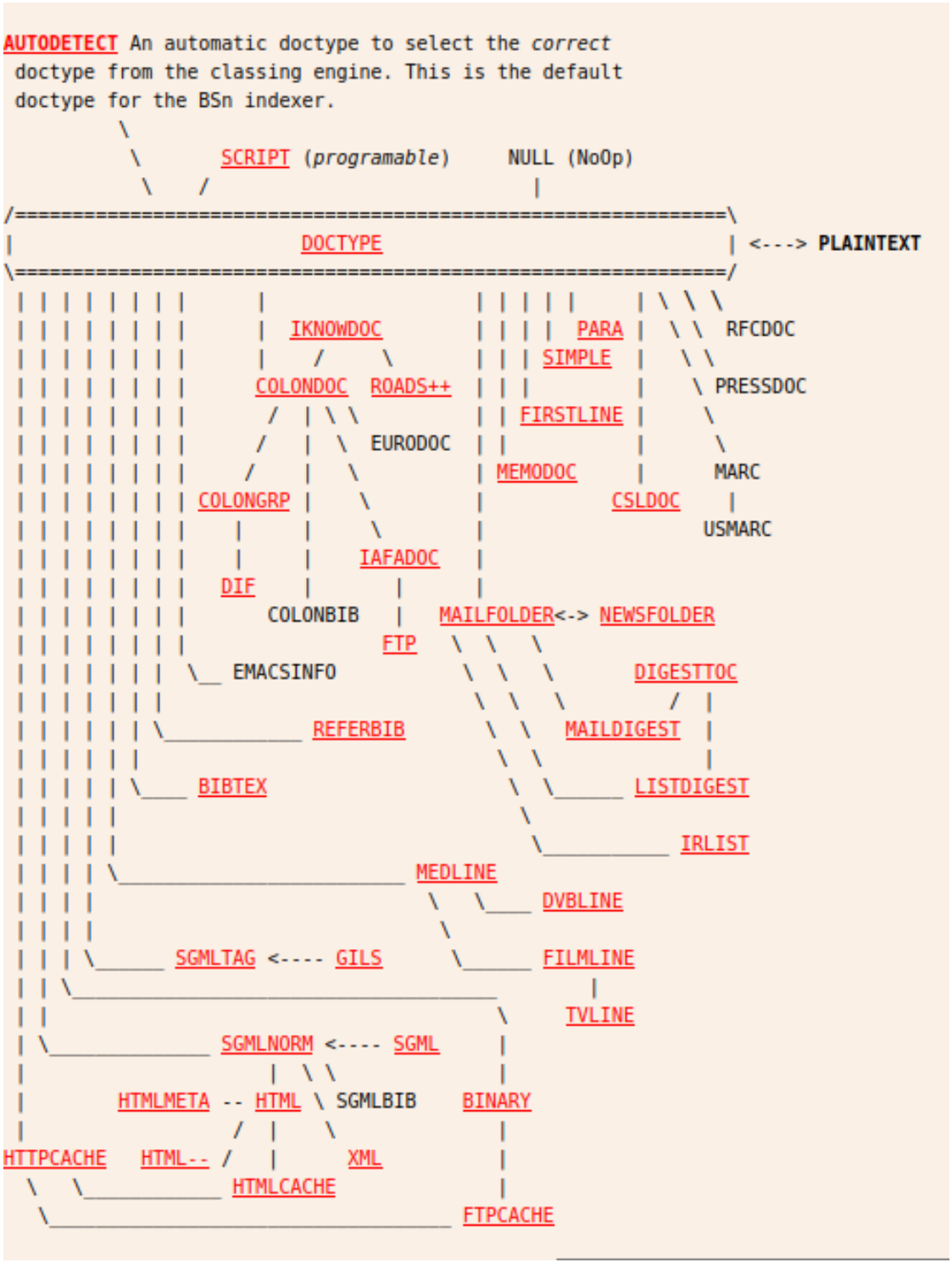
```
query = "beschaffungsmanagement:3 OR beschaffungsmarketing:3 OR beschaffungsmarkt:3
OR beschaffungsplanung:3 OR beschaffungsprozesse:3 OR (deterministische:3 FOLLOWS
beschaffung:3) OR einkaufspolitik:3 OR (stochastische:3 FOLLOWS beschaffung:3) OR
strategien:2 OR strategie OR (c:3 FOLLOWS teilemanagement:3) 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.

Whitepaper v.2.91 (work in progress)

Doctypes

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



Whitepaper v.2.91 (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
```

Lets start off the with AUTODETECT type as it’s often the go-to default. 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.