1.3 Query Languages: Keyword query languages

"Keyword-based query language" refer to query languages that use (relatively) unstructured bags of words that the user deems signifcant as queries or a part thereof. A typical characteristic of keyword query languages is the implicit conjunctive semantics, that is, by default answers must contain all words in a query.

The motivation behind keyword-based query languages is to enable casual users to construct queries and obtain useful results without having to undergo training in a query language, without having to know the underlying structure of the data and even without having a clear understanding of the data structures. Another advantage of keyword query languages is that they allow for querying over heterogeneous data, that is, data with different underlying schemas.

Keyword queries are not only easy to construct, but also have proven to be surprisingly effective in helping the user to localize relevant information. Consequently, keyword querying has become an established technique for finding information that virtually all Web users are familiar with. Keyword search is used in a wide variety of applications and domains, in Web search engines such as Google and Yahoo! which allow for general internet search over various types of documents as well as in more specialized contexts and domains. For example, the query "XML Web" entered into Google yields a lists of Web pages in which the terms occur. On the shopping site Amazon and the auction site Ebay the same query "XML Web" results in a list of products available on the sites and on the social networking site Facebook, the search results for the same query "XML Web" contain relevant user groups, events, user profile add-ons and users who are interested in the Web and XML.

Since keyword-based querying is established and is effective in querying the Web in a variety of domains, it is a promising and worthwhile approach to achieve non-expert querying of XML and RDF data. Various keyword query languages have been proposed. Tis section presents the main different approaches and gives an overview over the different paradigms and concepts. Keyword querying in relational databases (see e.g. [112]) is a related topic not be treated in this section.

1.3.1 Characteristics of keyword query languages

Three different types of keyword-based query languages for structured data can be distinguished:

1) Keyword-only query languages, the simplest variant. Here, queries consist of a number of words, which are usually matched on the textual content of nodes in an XML or RDF document. In some cases, the keywords may also be matched against node or, in the case of RDF, edge labels, but generally, the query makes no reference to the structure of the data.

Most keyword query languages for XML and RDF presented in the following fall into this category, for example XKeyword [15], [111], XRank [106] and XK-Search [196].

2) Label-keyword query languages, for example XSearch [64], where a query term is a label-keyword pair of the form l:k. The term matches data where a node with the label l has textual content, either directly or through a descendant node, matching the associated keyword k. It is thus possible to indicate some structure in the query.

Depending on the query language, either the label or the keyword may be optional, meaning that query terms may be of the form :k, l: or l:k. For example, the query "title:Web" applied to the data for Figure XX, an excerpt from a fictional bibliographical XML database , matches nodes 3 and 17, while

the query ":Web", a keyword-only query in which no label is speciï¬ed, matches nodes 3, 17 and 26, since it does not impose constraints on the node label.

While queries of keyword-only languages match both on labels and content, queries of label-keyword languages make it possible to express both, the association between label and keyword and the kind of element (node label or node content) a keyword refers. Thus, label-keyword query languages allow for more precise queries.

3) Keyword-enhanced query languages like for instance Schema-Free Query [133] and XPath/XQuery Full Text (FT) integrate simple keyword querying in traditional query languages like XPath. They allow for the specification of the structure to the degree that it is known and make it possible to use keyword-querying where it is not, thus offering a smooth transition between keyword querying and traditional querying.

Since traditional query languages offer ways to specify queries when the user lacks knowledge about the data structure, for example through regular path expressions in XPath, the question arises how traditional query languages and Keyword-enhanced query languages can be differentiated. The answer is pointed out in [88] and [178]: Regular path expressions are useful if the the structure is incompletely known to the user, but are not practical if the user has no knowledge of the structure at all. Thus, keyword querying aims at accommodating the lack of knowledge about the structure in a more fundamental and comprehensive way than traditional query languages.

A second, complementary characteristic of keyword query languages is how they are realized:

1) The majority of keyword query languages are implemented as stand-alone systems that handles the query evaluation and determining, where applicable, rankings of the answers.

2) Another group of keyword query languages translate the keyword queries into a traditonal query language and thus outsource the query evaluation.

3) Keyword-enhanced query languages combine conventional query languages like XPath or XML-QL with keyword-querying techniques and thus build on existing systems.

1.3.2 Determining the Returned Answers

Querying consists in both, selecting data and determining from those data the answers to return.

In keyword querying the Web with search engines, some structural information may be taken into account in selecting data and ranking answers, for example by assigning different scores depending on whether a keyword occurs in the title or is printed in big or bold text [36], but the structure of a document plays no role in determining the answers to return. There are two reasons for this: Efficiency and the generally small size of Web pages

1. Efficiency: Search engines process an hudge amount of data (in July 2008, Google stated their link processing system had found more than 1 trillion individual links, although not all pages found are indexed http://googleblog.blogspot.com/2008/07/we-knew-Web-was-big.html). Retreving and indexing structural information would increase both the data and the processing load, decreasing the efficiency of search.

2. Small size of Web pages: Web pages are typically of comparable small sizes, what makes it practicable to return answers consisting of whole Web pages.

In contrast, with keyword-search against RDF or XML documents, one might query a single big document (represents for example a bibliography or an address book) containing thousands of entries or more, making it possible to efficiently query (some of) the document structure, but preventing to return the whole document as answer.

With the domain-specific keyword querying of limited, homogeneous data like on shopping Web sites querying only serves one single task like finding products. There are only a few types of objects (like books and DVDs), and all answers have a limited set of attribute (like for books "title", "author" and "price", and for DVDs "title", "price" and "region code"). Thus the answers to return from the data selected by a query can be pre-defined.

In contrast, with keyword-search against RDF or XML documents, queries might serve all kind of purposes, making it necessary to determine from the queried document and the data selected by a query what is to be returned as answer.

Since XHTML is an application of XML and XHTML files are not data-centric but document-centric, methods for determining answers from the queried document and the data selected have to perform well on both, data and document-centric XML files. Indeed, a truly versatile keyword-based query language for XML should ideally yield useful results for both kinds of XML files, as well as any that might combine aspects of both.

Simple keyword queries can be formalized as follows: A keyword query $K = \{w_1, \ldots, w_k\}$ consisting of k keywords (conjunction is assumed here and in the following) matched on an XML data set T selects a set of node sets $L = \{L_1, \ldots, L_k\}$ where each $L_i = \{v_1, v_2, \ldots\}$ consists of all nodes v_i which contain some w_j.

Applied to the data in Figure XX, the keyword $w_1 = Smith$ in the query $K = \{Smith, Web\}$ matches the content of node 11 which is the last name of one of the authors of an article, that is $L_1 = \{11\}$, while the keyword $w_2 = Web$ matches the titles of both articles, that is, $L_2 = \{3, 14\}$.

Returning only the matched nodes as answers would not provide useful information, nor would returning the whole document which might contain many more irrelevant entries. Given the nature of the data and the query, the user can be assumed to be interested in receiving information about articles whose data contain his search terms "Smith" and "Web". This means that the "article" nodes are the nodes of subtrees which constitute meaningful semantic entities that should be returned, as a whole or in part, as answers to the query.

A common approach to determining answers in keyword-based query languages is first to find a most specific node of the document queried which is an ancestor to at least one match instance of each keyword and to construct a return value, or answer, from this node, for example the subtree of which it is the root. The idea behind this is that the ancestor-descendant relationship indicates a strong semantic connection, especially when the distance between ancestor and descendant is small. Thus, a node which is the closest common ancestor of a set o instances of each keywords is assumed to encode the most specific concept in the document queries that the keyword matches have in common.

In the case of tree-shaped XML data, this is deï¬□ned as the Lowest Common Ancestor (or LCA) [108], a common concept in graph theory which takes a match set S and computes the lowest node that has all nodes in Si among its descendants. Depending on the application and specific algorithm,

$S = \{S_1, S_2, \ldots\}$ may contain either all answer sets that cover the query exactly, i.e. contain one matched node for each keyword (â^¥Siâ^¥=â^¥Kâ^¥)(WTF?), or allow the answer sets to contain more than one match instance for each keyword.

In the example considered above, assuming the latter case, there are three answer sets $S_1 = \{11, 3\}$, $S_2 = \{11, 14\}$ and $S_3 = \{11, 3, 14\}$ with the respective LCA nodes $LCA(S_1) = 2$, $LCA(S_2) = 1$ and $LCA(S_3) = 1$. Intuitively, the subtree rooted at node 2 constitutes the domain of the best answer of the query and is preferred to the other answers which are not specific enough and either is not be returned as an answer or else is returned as answer of lower rank.

A relevant answer is not always contained in the LCA subtree. For example, the label-keyword query K = {author:Doe,author:Smith} yields node 5 as the LCA. This answer is not necessarily informative since it cointains only the authors' names, while it can plausibly be assumed that the user is interested in further information like the titles of the articles they coauthored.

The following shows how a few keyword query languages among other automatically determine answers from keyword matches. The problems they face are: avoiding false positive and false negative, ranking (if provided by the language), and returning informative answers.

1.3.3 Examples of Keyword Query Languages

In the following, we present three examples of Keyword Query Languages: XKeyword, XSearch, and XRank. XKeyword requires the users not only to specify selections with keywords, but also the answers (containing matches of the selection keywords). XSearch frees the user of specifying answers. XRank also frees users of specifying answers and in addition has a sophistcated answer ranking imnspired from PageRank.

XKeyword

Unlike almost all later approaches to XML keyword querying which can be applied only to tree-shaped data[1], can be used on XML graphs and does not require the XML data to have one common root node.

As in most other keyword query languages, a Xkeyword queries is a sequence of keywords, expressing a conjunction. Keywords are matched both on node labels and on textual content. To achieve semantically meaningful answers, the XML schema graph is grouped by the query author into possible return types, *target objects*, which are annotated with their relationships to other target objects. For example, a target object of type *article* could consist of article, author and title nodes and stand in a *contained in* relation to a target object of type *proceedings*.

The system stores the XML data in a relational database as a set of connection relations and an inverted index that indicates for every keyword the elements in which it occurs. Queries are then processed by retrieving the relevant objects for the keywords and generating minimal cycle-free subgraphs that contain all input keywords. These, in turn, can be mapped onto subtrees of the target object graph, yielding the answers. The answers are generated in parallel, meaning that smaller results (size being measured as edge distance between keyowrd matches) are generated first,

[1] Project Pages: http://db.ucsd.edu/XKeyword/

resulting in a ranking of the results reflecting the following principle: "Trees of smaller sizes denote higher association between the keywords, which is generally true for reasonable schema designs."

The results can be be displayed in a list or as a *presentation graph* that can be navigated through clicking and expanding results. Trivial and duplicate matches are hidden and answers are grouped by their schema.

XSearch

XSearch is a label-keyword query language with ranking for XML. Search terms can be of the form $l:$, $:k$ or $l:k$. A term $l:k$ matches a node if the node has label l and a descendant in whose content k occurs. All terms are optional unless prefixed with "+".

Matched nodes are grouped into entities according to an **interconnection** (binary, reflexive and symmetric) relationship. Two nodes nodes v_1 and v_2 are **interconnected** if, except possibly v_1 and v_2, no two distinct nodes on the paths from v_1 or v_2 to their LCA (that is, no two distinct nodes except possibly v_1 and v_2 on the shortest undirected path beween v_1 and v_2 have the same label). An answer set contains at most one match for each keyword, exactly one match for each + prefixed keyword, in the query.

An answer set is said to be **connected** if

- either it contains a node, the **star center**, that is with all other nodes in the set - the answer set is said to be **star interconnection-related**)
- or if all its nodes are pairwise inconnected – the answer set said to be **all-pairs interconnection-related**).

In a query, the type of connection condition, star interconnection-related or all-pairs interconnection-related, may be chosen depending on the data. An answer is defined as an answer set that fulfills the selected connection constraint.

Consider the query $K=\{+last:Smith, +:Web\}$ evaluated on the example on Figure XX. As in the similar query before, there are two answer sets, $L_1=\{11\}$ and $L_2=\{3,17\}$. Since every answer set must have the cardinality of the query and every keyword must be matched, the answer sets are $S_1=\{11,17\}$ and $S_2=\{11,3\}$. The shortest undirected path between nodes 11 and 3 contains every node label only once which means that nodes 11 and 3 are interconnected. In contrast, nodes 17 and 3 are not interconnected since nodes 2 and 16, which both lie on the path between the respective nodes and the LCA node ``bib'' have the same label, ``article''. The only answer to the query is thus $S_1=\{11,3\}$. The interconnection relation in this case avoids grouping matches together that belong to different articles as simple LCA-based grouping does.

Since there are only two elements in each S_{i} in the previous example, the interconnected nodes are both star-related and all-pairs related. However, since star-relatedness is a relaxation of the constraint of all-pair relatedness in the sense that for a set of nodes to be all-pairs related, every node has to be a star center, this is not always the case as illustrated by the following example:

The query $K=\{+last:Smith, +last:Doe, +year:2005\}$ yields the answer set $S_1=\{11, 8, 4\}$. Nodes 11 and 8 are not interconnected since the path between them passes two nodes with label ``author''. Node 4 however is interconnected with both 11 and 8. Consequently, $S_1=\{11, 8, 4\}$ is not a query answer if all-pairs interconnection-related is used, but it is according to star interconnection-related.

Grouping using all-pairs interconnection-related answer sets is thus more restrictive than star interconnection-related answer sets which in turn is more restrictive than requiring an answer set to consist of the nodes of a tree rooted at the lowest common ancestors of nodes matching keywords in the query.

The above example also illustrates that all-pairs interconnection can lead to false negatives, since S_1 is a valid answer to the query K.

Additionally, both types of interconnection semantics are subject to false positives when node labels are different but refer to similar concepts. In Figure \ref{nxmltreealt}, the query $K=\{+:Smith, +:2003\}$ yields an answer set $S_1=\{11, 18\}$ with $LCA(S_1)=1$. The document root node 1 is wrongly returned as a result of the query since ``article'' and ``book'' are different labels but represent semantically related entities.

XSearch ranks answers using the vector space model and the tf-idf measure applied at the granularity of individual nodes. Other factors considered in ranking are the distance between the nodes in the answer set, and the number of pairs of nodes in an answer set that stand in an ancestor-descendant relationship (since this relationship generally indicates a strong connection).

XSearch is implemented using inverted indices for keywords and labels, an interconnection index and a path index. The interconnection index contains the pre-computed interconnection relation for all pairs in a document, while the path index stores for each keyword the paths by which it can be reached, which allows to compute answers in ranking order when only the subtree size and the number of ancestor-descendant pairs are considered. The interconnection index which stores the interconnection relationships between nodes can be pre-computed or generated incrementally online as queries are evaluated.

The approach of XSearch can be used with other (binary, refelxive and symmetrical) relationships beween nodes than the above-mentioned interconnection relationship, thus making it possible to adapt to the semantics of specific applications. Furthermore, the approach of XSearch, can according to its authors, be adapted to graph-shaped (instead of tree-shaped) documents, as resulting from ID-IDREF references and links in HTML and in XML docuements, or as RDF graphs.

Xrank

Xrank, as the name suggests, puts a bigger focus on result ranking and employs more refined ranking techniques than the above-mentioned approaches. XRank allows for querying a mix of (graph-shaped, i.e. containing hyperlinks) XML and HTML documents. When querying XML, answers are XML fragments, when querying HTML, answers are whole documents, that is, XRank behaves like a traditional Web search engine when querying HTML documents. XRank is one of the few query languages presented here that assume document-centric XML and emphasizes data grouping.

XML query evaluation in XRank proceeds by first matching the keywords of the query on the content of the nodes of the XML document. A node set *S* is defined as a set of nodes in the content of whih at least one instance of a query keyword occur. Content is defined as usuall via the ancestor-descendant relationship, ignoring Hyperlinks.

An answer to a query is a node *n* in an answer set *S* such that the content of *n* (defined as usual through the descenfant relationship) includes occurrence of all query keywords, after excluding the

occurrences of the keywords in descendants of n that already contain occurrences of all the query keywords. In other words, an answer is a node in an answer set which is an LCA of occurrences of all keywords having no descendants that also are LCAs of (another set of) occurrences of all the keywords.

As an example, consider the query $K=\{XML, Web\}$ evaluated on the data in \ref{nxmld2}. The keyword match sets are $L1=\{13,15\}$ and $L1=\{2,12,13,16,19\}$. Based on this, some exemplary answer sets are $S1=\{13\}$, $S2=\{15,16\}$, $S3=\{13,12\}$ and $S4=\{15,19\}$.

S1 consists only of one node which contains all keywords, meaning that the LCA of S1 is identical with its single element, node 13. Since this node is the LCA node and does not have any children, it is an answer.
Similarly, node 14, the LCA of $S2$, is also an answer.
In contrast, $S3$ and $S4$ both have as LCA node 11 which is an ancestor of nodes 13 and 14, that, as explained, are themselves LCA nodes. $k2=Web$ has an occurrence contained by 11 which is not part of an LCA, namely in node 12. However, there is no match of $k1=XML$ in a descendant of node 11 which is not also contained in an LCA. Therefore, node 11 is no answer.

The intent behind ruling out "non-minimal" LCAs is to have answers as specific as possible. However, since document-centric XML represents a cohesive text where structurally close elements can be expected to be strongly interrelated in their topic, it might be of interest to accept answers that are not as specific as possible.

The XRank grouping mechanism is susceptible to the same types of false positives are LCA and interconnection semantics (in presence of synonym or near-synonym labels) are, that is, unrelated entities may be grouped together. Note that XRank also accept disjunctive queries.

XRank uses three criteria, specificity, keyword proximity and the connections between elements, for ranking answers. *Specificity* refers to the distance between the matched leaf nodes and the answer node, while *keyword proximity* means the distance between the keyword matches themselves. Specificity --vertical distance-- and keyword proximity --horizontal distance-- thus combine into a two-dimensional proximity metric. Finally, a variant of Google's PageRank, ElemRank, is used to let the links between elements factor into a result node's ranking value.

ElemRank is an adaptation of PageRank that takes specific characteristics of XML data into account, namely the bi-directional propagation of ElemRanks through links, the aggregation semantics for reverse containment relationships and the distinction between containment links and hyperlinks. While hyperlinks are ignored when matching keywords, they are considered when calculating ElemRanks. Since containment links, the parent-child relationship between XML elements, represent a stronger relation than hyperlinking through e.g. IDREFs, the two are handled separately with the propagation of ElemRank value between elements connected by containment edges being bi-directional. Additionally, the ElemRank of a node is defined as the sum of the ElemRanks of its children, meaning that the ranking values of an entity's subparts in turn combine into that entity's ranking value.

The ranking value of each instance of a keyword match is then calculated as its ElemRank value, scaled by a decay factor that is inversely proportional to the distance between the result node and the keyword match.

Finally, the ranking value of the result tree is the sum of the ranking values of the contained keyword occurrences multiplied by a measure of keyword proximity which is based on the size of the smallest text window containing all matches.

If a keyword has several occurrences in the subtree governed by the answer node, the value of the node with the highest ElemRank value is used.

In summary, the criterion of specificity is realized as the decay scaling factor where the decay increases as the distance between a keyword occurrence and the result node grows, meaning that the ElemRank calculated from the link connections between the elements becomes smaller. The keyword proximity criterion on the other hand is represented as the scaling factor of the overall ranking value of the result, here, a bigger distance between the keyword occurrences corresponds to a lower scaling factor.

Answers are computed through an evaluation of Hybrid Dewey Inverted Lists. Dewey ID enumeration is a system to enumerate nodes in an XML tree. A Dewey ID is a vector that summarizes the path from the root node of a document to a node. Figure \ref{nxmld} shows the data from the previous figure enumerated using Dewey IDs. In this enumeration scheme, the ID of an ancestor node is a prefix of the IDs of its descendants. The Lowest Common Ancestor of a set of nodes can thus be easily computed by determining the longest prefix shared by all nodes' Dewey IDs. For example, $S2=\{15,16\}$ and $S4=\{15,19\}$, using Dewey enumeration, become $S2=\{0.1.2.2.0,0.1.2.2.1\}$ and $S4=\{0.1.2.2.0,0.1.2.3.1\}$ and this information suffices to compute the respective LCA nodes, 0.1.2.2 and 0.1.2.
This property allows for the efficient computation of answers in a single pass when all keywords are stored in an inverted list associated with their Dewey ID.

The authors present two techniques for computing top-k answers without first generating all answers, then ranking them.

Further keyword query languages are XKSearch[2] and XSeek[3]. Presentations and comparisons of these languages are given in *QuoVadis*.

References:

Klara Weiand, Tim Furche, and Francois Bry. Quo Vadis, Web Queries? Proc. International Workshop on Semantic Web Technologies (Web4Web), 2008), http://www.pms.ifi.lmu.de/publikationen/index.html#PMS-FB-2008-7

Yi Chen and Wei Wang 0011 and Ziyang Liu and Xuemin Lin. Keyword search on structured and semi-structured data Proc. SIGMOD Conference, pages 1005-1010, 2009, http://www.cse.unsw.edu.au/~weiw/project/keyword_sigmod09_tutorial.pdf

[2] Project Pages: http://db.ucsd.edu/People/yu/xksearch/index.htm
[3] Project Pages: http://xseek.asu.edu/intro/Home.htm

1.4 Engineering: Inverted files, Distributed Hash Tables (DHT), BigTable, MapReduce for Network Analysis

1.4.1 Inverted files

An inverted file is a data structure for full text search. An inverted files consists of an index mapping terms (like word, expressions, numbers, etc.) to data items (like documents or more generally files) they occur in. Inverted files are widespread in document retrieval systems and search engines.

One distinguishes between record level and word level inverted index. A **record level inverted index** or **inverted file index** or **inverted file** maps a term t to the set of data item i_k and its frequency f_k in each data item $\{(i_1, ..., (i_n, f_n)\}$ it occurs in. A **word level inverted index** or **full inverted index** or **inverted list** maps a term t not only to each data items and its frequency in each data item but also to the various positions in each data items it occurs in $\{(i_1, f_1, p_1, p_{k1}), ...(i_n, f_n, p_{kn})\}$ where the positions p_j are word or term counts, not characters or bit counts, so as to make it possible to determine term adjacencies that are needed, for example for phase querying (such as seaching for documents with occurrencves of "Air France" and not of "Air" and of "France").

In building a record level or word level inverted index, a **forward index** is first built from each document, that is a list of the words, if needed with their positions, occurring in the document. Generating an inverted file from forward indices is called **inversion**. Different algorithms for inversion have been conceived.

An important issue is to maintain indexes as the document collection is modified. Algorithms of different kinds have been divised for this, too.

When the document collection considered is hudge, as it is the case for general puropose Web search engines, the inverted indices have to be distributed. This issue is addressed in the next section. Distribution can be based on documents (as Google uses) or on terms.

References:

Justin Zobel, Alistair Moffat. "Inverted Files for Text Search Engines". ACM Computing Surveys (New York: Association for Computing Machinery) 38 (2): 6, 2006 "In this tutorial, we explain how to implement high-performance text indexing."

1.4.2 Distributed Hash Table (DHT)

A Distributed Hash Tables (DHT) is a storage data structure similar to a standard hash table: it uses a hash function to efficiently map values (like file names or file contents) to keys, the keys serve as indices of an arry-like data structure where the values are stored, a value can be retreived using its key for a look-up in this data structure. DHTs differ from hash tables by storing values on several computers, or nodes, organized as a network. Responsibility for maintaining the mapping from keys to values is distributed among the nodes in such a way that a change in the participating nodes causes minimal disruption.

DHTs are

- **decentral** (there is no central coordination of the network),
- **scalable** (they can handle large networks possibly with millions of nodes), and
- **fault tolerant** (they can handle continual node arrivals, node departures, and node failures).

The first DHT (CAN, Chord, Pastry, and Tapestry) have all been developped in 2001. DHTs have been motivated by peer-to-peer file sharing systems of the first generation such as Napster, Gnutella, and Freenet. Instead of DHT, Napster had a central index (what made it vulnerable to denial-of-service attacks and lawsuits), Gnutella used flowed queries, that is, broadcasting of queries to the whole network, Freenet used a key-based routing (bringing files with similar keys the same nodes) and could not always returned the files sought for.

DHTs are used to build among others peer-to-peer file sharing systems (like BitTorrent and the Coral Content Distribution Network), cooperative web caching systems and instant messaging systems.

A DHT consist in

- an abstract keyspace,
- a keyspace partitioning scheme, and
- an overlay network connecting the nodes

The abstract key space is for example the set of the 128-bit strings or the set of 160-bits strings. A DHT uses a hash function for maping a value (a file name or a file content) to an element of the key space called the value's key.

The keyspace partitioning scheme maps the keys to the nodes of the network. Most DHTs use some variant of consistent hashing for this purpose.

Consistent hashing maps a value to a key which is a (real) angle or, equivalently, to a point on a circle. Each node is randomly assigned such a key called its identifier (this assignement does not have to follow the physical topology). The "keyspace distance" $d(k_1, k_2)$ between two keys k_1 and k_2 or between a node identifier k_1 and a key k_2 is their angle. A nodes n with identifier id_n "owns", that is assigned and stores, the values with keys k such that $d(id_n, k) < d(id_m, k)$ for every other node $m = n$ with identifier id_m. That, is the circular keyspace is partitioned into contiguous segments, or slots, whose middles are the node identifiers. The addition or removal of a node n can be performed by modifying only the set of keys owned by the nodes with adjacent identifiers adjacents to that of n, that is, keeping local the network reorganization.

The overlay network consist in the nodes with their routing tables, that is, the set of links to a node's neighbours. The neighbours are defined in a certain way which determine the so-called **network topology** which must fullfil the following condition:

- For any node n and for any key k, either n owns k or n has a link to a node m
- the identifier id_m of which is closer to k (for the keyspace distance) than
- the identifier of n.

A message (like `get(k, value)` or `store(k, value)`) is forwarded from node to node as follows: If a node n does not owns k, then it forwards the message to the one of its neighbour the identifier of which is closest to k (for the keyspace distance). This form of routing is called key-based routing.

The choice of a network topology is a trade-off between minimizing the outdegrees of the nodes and minimizing the number of hops on any route. These two goals contradict each other: The smallest outdegree (1) yields routes of as many hops as there are nodes; the shortest routes (of length one) requireconnecting every node to every other node. The following combinations are possible, the third

is coomonly retained even though it is sub-optimal:

- Outdegree O(1), route length O(n)
- Degree O(log n), route length O(logn / log log n)
- Degree O(log n), route length O(logn)
- Degree O($\sqrt{n}$), route length O(1)

Reliability can be enhanced by redundantly assigning each key to several nodes. Also, instead of forwarding messages `put(k, value)`, it is possible to determine (as described above) which node owns k and later send the data (value) to that node.

References

Moni Naor and Udi Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. Proc. SPAA, 2006. http://www.wisdom.weizmann.ac.il/~naor/PAPERS/dh.pdf

Gurmeet Singh Manku. Dipsea: A Modular Distributed Hash Table. Ph. D. Thesis (Stanford University), August 2004. http://www-db.stanford.edu/~manku/phd/thesis.pdf

Gurmeet Singh Manku. Routing Networks for Distributed Hash Tables. Proc. 22nd ACM Symposium on Principles of Distributed Computing (PODC), 2003
http://www.cs.stanford.edu/~manku/papers/03podc-dht.pdf

1.4.3 Big Table

BigTable is database system built on Google porgramms (such as theGoogle File System (GFS) and the Chubby Lock Service). It is currently not distributed or used outside of Google. Google offers access to BigTable as part of the Google App Engine, a platform for developing and hosting web applications in data centers managed by Google. Thus, BigTable is an essential component of Google's cloud computing technology. BigTable, or similar database systems, are used in all sclaing Web systems like Amazon, Yahoo, etc.

> *"BigTable is a system for storing and managing very large amounts of structured data. The system is designed to manage several petabytes of data distributed across thousands of machines, with very high update and read request rates coming from thousands of simultaneous clients."*
>
> Jeff Dean (cited by Tim O'REilly
> in his blog post of the 3rd of May 2006, "Database War Stories #7: Google File System and BigTable"
> http://radar.oreilly.com/archives/2006/05/database-war-stories-7-google.html)

BigTable is used by Google applications, among other by the computing platform MapReduce for distributed computing with large data sets on clusters of computers, Google Reader, a Web-based aggregator of Atom and RSS feeds, Google Maps, Google Earth, Google Books search, Blogger.com, Google Code hosting, the digital network Orkut, and YouTube.

BigTable store tables, or relations, consisiting in a finite number of tuples (or rows). BigTable does not require all tuples of a table to have a same fixed number of columns. BigTable supports fast access to both, table columns and rows, that is, it can be seen as both a row-oriented and a column-oriented database system.

Each table has multiple dimensions, one of which for time used in versioning. Tables are optimized for the Google File System (GFS) by being split into so-called tablets, that is table segments of about 200 megabytes. The Google File System (GFS) is a kind of Distributed Hash Table (DHT).

When a tablet size reach a given limit, the tablet is compressed (using the publicized algorithm BMDiff and the secret algorithm Zippy, avariation of LZO).

The locations of the tablets in the GFS are recorded as database entries in special tablets called "META1" tablets. META1 tablets are found by querying the single "META0" tablet, which has a machine for itself (since it is repeatedly queried by clients looking for the location of data).

Like GFS's master server, the META0 server is generally no bottleneck since the processor time and bandwidth necessary to discover and transmit META1 locations is minimal and clients intendsively cache locations to minimize queries.

> *"[BigTable] is designed to scale to hundreds or thousands of machines, and to make it easy to add more machines the system and automatically start taking advantage of those resources without any reconfiguration."*
>
> Jeff Dean (cited by Tim O'Reilly
> in his blog post of the 3rd of May 2006, "Database War Stories #7: Google File System and BigTable"
> http://radar.oreilly.com/archives/2006/05/database-war-stories-7-google.html)

References:

Jeff Dean. BigTable: A System for Distributed Structured Storage Video of a talk given at the University of Washington, 2006 *http://www.uwtv.org/programs/displayevent.aspx?rID=4188*

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. Proc. Seventh Symposium on Operating System Design and Implementation (OSDI), 2006. *http://labs.google.com/papers/bigtable.html*

Amazon's Dynamo, a "highly available key-value storage system", is similar to Google's BigTable. It has a very similar data model (rows with arbitrary attribute/value pairs), no fixed schema (of what columns are allowed in a row), each row may have different columns/attribute sets.

Dynamo is used by Amazon for implementing the shop as well as many of the Amazon Web Services (like Amazon's SimpleDB http://aws.amazon.com/simpledb/).

Dynamo has a very simple query language, inspired from, but quite different of, SQL.
* charges PER RESOURCE USE (CPU, bandwidth, storage) HOW TO REPHRASE THIS IN A SENTENCE?

Reference

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. (2007) Dynamo: Amazon's Highly Available Key-Value Store. Proceedings of the 21st ACM Symposium on Operating Systems Principles, pages 205-220. *http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf*

1.4.4 MapReduce for network analysis

MapReduce is a programming framework (developped at Google) for data intensive computing motivated by the following problem: The index size of common Web search engines being around 10 billion pages, neither inverted lists, nor the computation of PageRank are feasible even on the fastest super-computers available: This would require about four months just to sequentially read "the Web" (that is, the part of the Web a search engine considers); the Web grows faster than the speed of (single core) CPUs increases.

The idea of MapReduce is to perform text extraction and the PageRank computation on massively parallel computer clusters.

Google throws several millions of computers (with fairly standard components) at the problem with a share-nothing architecture, only connected via high-speed network. With only about 1000 machines, "the Web" can be read in about 3 hours.

The chhallenges in conceibving MapReduce were

- to figure out how to execute *data intensive* tasks massively parallel,
- to find a proper computing model for *data intensive* parallel processing,
- to provide for an automated workload distribution and management (rather than an explicit thread management and communication as, for example, in Java), allowing adaptation to changing a environment (due to failures that change the overall workload), minimizing the network communication (by minimizing the size of the intermediary results), and partitioning data so as to maximize locality (for common tasks).

MapReduce is inspired by the map and reduce functions of functional programming.

Excursus: Principles of Data Centrix Parallel Computing

Parallelism in imperative languages is mostly *explicit*. It is expressed for example in Java through threads and synchronisation. This is very convenient if parallelism is mostly to achieve so-called *task parallelism*. An example of task parallelism is a GUI and an application logic processed in parallel so as to increase responsiveness by avoiding idleness when waiting for user interaction.

Now, with the advent of multi-core processors and clusters, parallelism is becoming possible to all. *Data parallelism* can be striven for for performances. Examples of data parallelism are: highly parallel global processing units (GPUs) for applying filters to bitmaps, matrix operations in scientific computing, and PageRank computations. The classic approaches to imperative parallelisation fail: The programming is much to complex and therefore too expensive.

A solution is to exploit so-called data parallelism. This apporoach, which is that of MapReduce, is inspired from functional programming. It consists in applying (side-effect free) operations to large amounts of data, to automatically manage the data partitioning, distribution, and result aggregation. There is a trade-off, though: The approaches restricts how data (and thus "threads" called in this context "workers") may interact. Implicit data-centric parallelisation is becoming increasingly common (e.g., Chapel (Cray's FORTRAN successor) and Fortress (Sun's FORTRAN successor) both use it for example for matrix and vector processing.

Tasks well suited for data parallel processing are characcterized as follows:

- large amount of (input or intermediary) data (e.g., large array)

- same processing/function applied to each data element (e.g., array cell)
- no dependencies between the processing/function applied to different data elements (that is, communication between tasks is not needed)

A simple architecture for data parallel processing is the Master and Worker architeture:

A Master (or Controller)
- initializes the processing and partitions the data
- distributews to each worker its workload
- receives (and possibly aggregates) the results delivered by the workers

Each Worker
- receives its workload, process data independently from any other worker, and returns a result to the master

The model can be recursively continued, workers being masters of their own workers, yieding a tree-like processing structure.

Example: Method for approximating PI

Principle:
Given a circle with radius r inscribed in a square, we can observe that
- the area of the square $As = 4r^2$
- the area of the circle $Ac = \pi r^2$
- and thus $\pi = Ac/r^2 = 4\,Ac/As$

PI can be "approximated" by the following algorithm
- randomly generate P points in the square
- count the number of those points in the circle
- $\pi \approx 4\frac{IN}{P}$

This can be computed after the Master - Worker architecture as follows:

MASTER:
1. sends to each worker the number of random points to generate within the square

WORKER:
2. generates the given number N of random points within the square
3. determines the number of points within the circle
4. sends this number to the master

MASTER:
5. Computes the sum IN of the numbers returned by the workers
6. Computes P the total number of points generated (that is, product of N by the number of workers)
7. Computes the approximation of PI = 4 * IN / P

The failure of a number of workers (to performed the task they are assigned) does not result in a failure of the general vomputation. All needed is that, in computing P at step 6 the master subtracts the number failed workers.

1.4.4.1 The principle of MapReduce

MapReduce is a programming framework for implicit data parallelims on massively parallel architectures inspired from list processing in functional programming

(LISP, SML, Haskell)
"In Lisp, a map takes as input a function and a sequence of values. It then applies the function to each value in the sequence. A reduce combines all the elements of a sequence using a binary operation. For example, it can use "+" to add up all the elements in the sequence."

[Dean and Ghemawat, 2004]

What are the traits of functional programming relevant tp understanding MapREduce?

- Functional operations are side-effect-free, hence their computation order does not matter

  Example (in SML):

  ```
  fun f(l: int list) = sum(l) + mul(l) + length(l)
  ```

  The computation order of sum, mul, length does not affect the result l

- Functional operations do not modify data structures, but instead create new ones

  Example (in SML):

  ```
  fun naive_append(x, lst) = reverse(x :: reverse(lst) )
  ```

- Functions can be used as arguments of other, so-caled higher-order functions:

  Example (in SML):

  ```
  fun DoubleApply(f, x) = f(f x)
  ```

  - *whatever f does to its argument x, DoubleApply(f, x) does it twice*
  - *DoubleApply requires its first argument to be a unary functions with the same parameter and return type, that is, DoubleApply has the type ('a -> 'a).*

Let us recall examples of list processing with the higher-order functions map and reduce in functional programming:

- Functions can be used as arguments of other, so-caled higher-order functions:

- **map** creates a new list by applying a function f to each element of an input list
  In SML:
  ```
  map(f list): ('a -> 'b) -> ('a list) -> ('b list)
  fun map f [] = []
    | map f h::t = f(h) :: map(f, t)
  ```

- **fold:** moving over an input list, fold applies a function f to each list element and to an accumulator, yielding a new value for the accumulator

- passes this value of the accumulator for the application of the function f to the next list element
  Here, the order of the elements in the list does matter. Therefore one distinguishes between two variants of fold, foldl (for fold left) and foldr (for fold right):

  In SML:

```
foldl f initial list: ('a * 'b -> 'b) -> 'b -> ('a list) -> 'b
fun foldl f a []      = a
  | foldl f a (x::xs) = foldl f (f(x, a)) xs

foldr f initial list: ('a * 'b -> 'b) -> 'b -> ('a list) -> 'b
fun foldr f a []      = a
  | foldr f a (x::xs) = f(x, (foldr f a xs))
```

  Example (in SML):

  Implementations of sum (sum of the elements of a list), mul (multiplication of the elements of a list), length (of a list) using foldl:

```
fun sum(list)    = foldl (fn (x,a) => x+a) 0 list
fun mul(list)    = foldl (fn (x,a) => x*a) 1 list
fun length(list) = foldl (fn (x,a) => 1+a) 0 list
```

  Given a list of numbers, generates a list of partial sums like for example:
  `[1,4,8,3,7,9] -> [0,1,5,13,16,23,32]`

```
fun partsum list = foldl (fn (x, (a::as)) => (x+a::a::as)) [] list
```

Restricted to homogeneous types, fold it is called REDUCE in Lisp, Python, Ruby and Javascript. In SML, the type of reduce would be:

```
reduce f initval list: ('a * 'a -> 'a) -> 'a -> ('a list) -> 'a
```

In SML, the implementation of map is basically a left-to-right traversal of the list. Other traversals are possible as well since there are no no side effects: An application of the parameter function f to a list element does not affect an application of f to another list element.

It is therefore possible to reorder and even to parallelize the application of f. This can be achieved with a master-worker architecture as follows:

1. MASTER: distributes the function f and elements of the list to n workers
2. WORKER: each worker applies f to the list element it has received and sends the result to the master
3. MASTER: collect the results and bring them (in the right order) into a list

Note that the approach works with less workers than list elemnts: All needed in such a case is to reassign a tak to worker once it is idle agin up till all list elements have been processed.

How to implement REDUCE (or fold)? Let us restrict ourselves to "grouping operations" um, avg, mult, ad length. In such cases, we can again partition the task into sub-tasks (think of the divide-and-conquer scheme). The sub-tasks are independent of each other. Since the grouping operations are

associative, the order in which the sub-tasks are performed doesn't matter. It is therefore possible to collects and combine the results of the sub-tasks in whatever order they are delivered.

1.4.4.2 Google's MapReduce: Imperative Key-Value List-Processing

MapReduce is the name of an approach, of a C++ library developed by Google. It has nowadays many other implementations including an open source one called Apache Hadoop (cf. http://hadoop.apache.org/).

Its original use case was indexing of "the Web" (that is, billions of documents, peta-bytes of data). It is combined at Google with a parallelization infrastructure that allows s seamless use on clusters from 1 to 10^7 ciomputers. It hides details of parallelization, data distribution, load balancing, and fault tolerance. The price for these additional features is that it is slightly more restricted than its general model.

At the core of MapReduce are two functions, MAP and REDUCE, that have to be written by the users of the MapREduce library. They both operate on (key, value) pairs.

MAP takes an input pair and returns a set (or a list) of intermediate (key, value) pairs. The MapReduce library groups all intermediate values by key, and deliver as results sets of pairs each associating one key with the set (or list) of values for these key.

REDUCE takes a pair of intermediary key and associated set (or list) of values for these key and "merges" these values to form a possibly smaller set of values, usually just one AGGREGATE value.

Examples

Input of MAP: (URL, contents) --- URL serves as key, value is full content of document

```
                    |          Output MAP          |      Output REDUCE
Word Count          |         ("the", "5")         |     ("the", 1048576)
Word Occurrence     |  ("the", "www.example.org/" |  ("the", <list of URLs>)
Site Term Frequency| ("lmu.de", ("the", 2), ...) | ("lmu.de",
                    |                              | ("the", 7800), ...)
Next Word Probab.   | ("purple", "cow")            | ("purple", ("cow", .01),
                    |                              | ("color", .1), ...)
```

Word Count: (how often occurs "the" in total over all documents)
Word Occurence: (in which documents does "the" occur)
Site Term Frequency: (frequency per site of the terms)
Next Word Probability: (probability distribution for next word per word)

In SML the signatures of MAP and REDUCE and the full mapReduce operation would be as follows:

In SML:

```
map: ('k1 * 'v1) -> (('k2 * 'v2) list)
reduce: ('k2 * ('v2 list)) -> ('v2 list)

mapReduce: (('k1 * 'v1) -> (('k2 * 'v2) list))     (* signature map *)
        -> (('k2 * ('v2 list)) -> ('v2 list))      (* signature reduce *)
```

```
        -> (('k1 * 'v1) list)       (* input list of (key,value) pairs *)
        -> (('k2 * ('v2 list)) list) (* output list of
                                      * (key, set of values) pairs *)
```

It is worth noting the difference to the functional programming models map and fold:

- (key, value) pairs are used in place of list elements
- intermediary values are "implicitly" grouped by keys between MAP and REDUCE
- intermediary and final keys-value pairs are assumed to have the same type -- THIS ALSO THE CASE IN FUNCTIONAL PROGRAMMING, ISN'T IT?

Example: Word Frequencies and Counts in Web Pages

```
- Input: (URL, contents) --- URL serves as key, value is full content of document
  * e.g.,                        ("document1", "to be or not to be")
         -> MAP           -> ("to", 1),     ("be", 1),     ("or", 1), ...
         -> (impl.) GROUP -> ("to", (1,1)), ("be", (1,1)), ("or", (1)), ("not", (1)),
         -> REDUCE        -> ("to", 2),     ("be", 2),     ("or", 1),   ("or", 1)

map(String input_key, String input_value):
   // input_key: document URL; input_value: document contents
   for each word w in input_value:
       EmitIntermediate(w, "1");

reduce(String key, Iterator intermediate_values):
   // key: a word; intermediate_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
      result += ParseInt(v);
  Emit(AsString(result));
```

- just pseudo-code, but real code isn't much more complex
  * additional mostly some control code for partitioning and load balancing

Reference:

LÃ¤mmel, R. 2007. Google's MapReduce programming model -- Revisited. Sci. Comput. Program. 68, 3 (Oct. 2007), 208-237. DOI= http://dx.doi.org/10.1016/j.scico.2007.07.001
http://www.cs.vu.nl/~ralf/MapReduce/paper.pdf

Example: Computing a Word Co-Occurrence Matrix

- Input: (URL, contents) --- URL serves as key, value is full content of document
- Output: word co-occurrence matrix =
  square matrix M over set of unique words in the corpus (set of a all documents)
  cell M(i,j) = number of times the i-th word co-occurs with the j-th word
  (in some window of m words)

- **Naive: Pair-by-Pair**

```
map(String url, String document)
  for all w IN document:
    for all u IN NEIGHBORS(w):
      EmitIntermediate((w,u), 1);
```

```
reduce(WordPair p, Iterator intermediate_counts)
      int sum = 0;
  for all v in intermediate_counts:
    sum += ParseInt(v);
  Emit(p, sum);
```

- difference to word count is only that a pair of words is used as intermediary key
- emits large number of intermediate key-value pairs

For corpora of ~2 mio. documents (6 GB of data) and window size m = 2 2.6 billion intermediate key-value pairs (size ~31 GB) are emitted. This takes about 62 minutes on 20-machine cluster.

The apporach causes far too much network traffic

- **Improved: Word-by-Word**

The idea is to emit only one co-occurrence vector per word and document which contains the co-occurrence counts for all words

```
map(String url, String document)
  initialize(HashMap H);
  for all w IN document:
    for all u IN NEIGHBORS(w):
      H{u} := H{u} + 1;
    EmitIntermediate(w, H);
reduce(Word w, Iterator cooccurrence_vectors)
  initialize(HashMap HFinal);
  for all H IN cooccurrence_vectors:
    merge(HFinal, H);
  Emit(w, HFinal);
```

For corpora of ~2 mio. documents (6 GB data) and a window size m = 2 653 million intermediate key-value pairs (size ~48.1 GB) are emitted. THis takes about 11 minutes on 20-machine cluster

- An optimization is possible since the matrix is diagonal.

Reference:

Jimmy Lin. (2008) Scalable Language Processing Algorithms for the Masses: A Case Study in Computing Word Co-occurrence Matrices with MapReduce. Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP 2008). http://hcil.cs.umd.edu/trs/2008-28/2008-28.pdf

Example: Google's Indexing Pipeline

- Input: (URL, contents) --- URL serves as key, value is full content of document
- Output: inverted index, i.e., (term, list of occurrences)

- consists of a number of composed MapReduce tasks for example for determining the canonical URL of Web documents, for extracting link information (Web graph) from Web documents, or for extracting word occurrence from Web documents

References

Jeffrey Dean and Sanjay Ghemawat. (2004) MapReduce: Simplified Data Processing on Large Clusters. Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004), pages 137-150. http://labs.google.com/papers/mapreduce.html

A FEW PICTURES OF THW COMPUTATION LOADS UNDER MAPREDUCE WOULD BE GOOD HERE. I DO NOT KNOW WHICH ONES TO CHOOSE AND HOW TO COMMENT.

1.4.4 Computing PageRank with MapReduce

In implementing a graph algorithm on MapReduce, one has find an efficient "distributed" graph representation. Standard graph libraries use pointers, and thus require shared memory. On MapReduce, a relational representation with explicit keys would be preferable.

Candidates for a relational graph representations are: adjacency matrix, adjacency list over enumeration of nodes. Considering that very large graphs are usually very sparsely populated (that is, almost all cells are 0), asn adjacency list is the best representation. Note that for undirected graphs, only the upper (or lower) diagonal matrix need be represented.

Example: sparse matrix representation

A sparse matrix can be respesented as follows

```
1: 3, 18, 200
2: 6, 12, 80, 400
3: 1, 14
...
```

This representation is to be understood as follows: The edges are
1->3, from 1->18, from 1->200, from 2->6, ...

Example: Distributed Single-Source Shortest Path (SSSP)

Problem: From a givenstart node compute the shortest distance
(or shortest path) all other nodes

Standard solution: Dijkstra's Algorithm

```
function Dijkstra(Graph, source):
    for each vertex v in Graph:           // Initializations
        dist[v] := infinity               // Unknown distance function from source to v
        previous[v] := undefined          // Previous node in optimal path from source
    dist[source] := 0                     // Distance from source to source
    Q := the set of all nodes in Graph
    // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:                 // The main loop
        u := vertex in Q with smallest dist[]
        if dist[u] = infinity:
            break                         // all remaining vertices are inaccessible
        remove u from Q
```

```
            for each neighbor v of u:         // where v has not yet been removed from Q.
                alt := dist[u] + dist_between(u, v)
                if alt < dist[v]:             // Relax (u,v,a)
                    dist[v] := alt
                    previous[v] := u
        return previous[]
```

The MapReduce solution is similar to a breadth-first search from the start node

- shortest distance to start node := for start node: 0
  for all nodes n with an edge from some other set of nodes S:
  min(dist(m): m IN S
- MAP: receives (node n, (dist(n), children(n))) where children(n) = {m: (n,m) edge}
  emits (p, D+1) for all p IN children(n).
- REDUCE: receives (node n, (distances for n))
  emits (node n, minimum distance)
- iterate MAP/REDUCE until no changes in the distances
  One needs to copy over the children lists and already computed distances.
  An external terminator is needed.
- +1 can be replaced with positive weight (if desired)

This algorithm is less efficient than Dijkstra's algorithm because it explores all paths in parallel (the Disjktra's algorithm explores only the minimum-distance paths). But this algorithm can be efficiently parallelized.

One can compute PageRanks in a similar manner as single-source shortest paths are computed above.

Recall:

(Simplified) PageRank computation

```
- For page A
     PR(A) = (1-a) + a (PR(T1)/C(T1) + ... + PR(Tn/C(Tn)))
  where T1, ..., Tn are pages linking to A
        C(P) the cardinality (out-degree) of page P
        a the damping factor (jump to "random" page)
- Basic iterative computation:
  * Initialize: two matrices, current and next, holding the
                PageRank for the current and next "round"
                current := initial PR values
  * DO  iterate over all pages -> distribute PR/n from "current" to "next"
                                  to all {P1 ... Pn} linked pages
        current := next, next := empty_table();
    WHILE NOT converged
```

Distributed PageRank computation:

- Observe: (a) Each row of "next" depends on "current" but on no other row of "next".
  => Each row can be processed in parallel.
  (b) out-degrees are very small compared to overall number of pages
  -> very sparse matrix
  -> with suitable (e.g., adj. list) representation sufficently small size

- Algorithm:

```
* Phase 1:
  (URL, page content) -MAP-> (URL, (inital-PR, list-of-linked-URLs))
                             (no REDUCE)
* Phase 2: (URL, (current-PR, linked-URLs)
          -MAP->
              (URL, linked-URLs)        <-- needed for next iteration
              (URL', current-PR/|linked-URLs|) for each URL' IN linked-URLs
          -GROUP->
              (URL, (linked-URLs, PR-fragments))
          -REDUCE->
              (URL, (sum(PR-fragments), linked-URLs))
       * stop if converged otherwise go to Phase 2
```

Efficent implementations of MapReduce require appropriate data representations and data structures making possible independent computation steps.

The blog post http://michaelnielsen.org/blog/?p=534 gives a (single machine) MapReduce implementation of PageRank in Python. This program should work with fairly large Web graphs on an average PC.

1.4.A2 ASIDE: Open-Source MapReduce --- Apache's Hadoop

Hadoop is an open source implementation of MapReduce of the Apache foundation. It includes a distributed file system called HBase similar to BigTable. It is implemented in Java, but can to some extend also distribute non-Java programs. It is extensively used by Yahoo! and it is supported by IBM. It rather easy to set up and the most accessible MapReduce implementation available.

Hadoop has been named after the stuffed elephant of a child of Doug Cutting, Hadoop's main developer.

Using Hadoop in Java:

```
WordCount.java
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text,
IntWritable> {
      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

      public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output,
Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
```

```
          word.set(tokenizer.nextToken());
          output.collect(word, one);
        }
      }
    }

    public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable,
Text, IntWritable> {
      public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
IntWritable> output, Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
          sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
      }
    }

    public static void main(String[] args) throws Exception {
      JobConf conf = new JobConf(WordCount.class);
      conf.setJobName("wordcount");

      conf.setOutputKeyClass(Text.class);
      conf.setOutputValueClass(IntWritable.class);

      conf.setMapperClass(Map.class);
      conf.setCombinerClass(Reduce.class);
      conf.setReducerClass(Reduce.class);

      conf.setInputFormat(TextInputFormat.class);
      conf.setOutputFormat(TextOutputFormat.class);

      FileInputFormat.setInputPaths(conf, new Path(args[0]));
      FileOutputFormat.setOutputPath(conf, new Path(args[1]));

      JobClient.runJob(conf);
    }
}
```

1.4.5 Discussion: MapReduce: A major step backwards?

MapReduce has been a hype mostly due to its intensive use at Google. Recently, in 2008-2009, MapREduce has experienced a a backlash from the database community. The criticisms are as follows:

MapReduce uses a so-called "Schema later" or "Schema never" approach, but the experience (in databases) has sohwn that "schemas are good". Indeed, schema, like programming types, ensure that assumptions made on data are met by the actual data. Furthermore, schemas make possible data type specific optimizations of data storage and data access.

MapReduce mixes schema with application, nbut the experience (in databases) is that "separation of the schema from the application is good". Indeed, such a separation allows schema to be discovered (or queried) and data to be re-used more easily.

MapReduce is a low-level record manipulation paradigm, but the experience (in databases) is that languages that, like Codasyl (the widespread database access language before the relational era) and MapReduce describe how to access the data, are inferior to more (high-level or declarative) languages like SQL that describe what data ones wants.

MapReduce suffers from immature implementations characterized by:

- always initialy scanning all data rather than using indexes
- no consideration of skew (some keys have much more values, what makes the reduce phase last longer)
- forces materialization of intermediary keys rather than leaving a choice (because a push communication from MAP to REDUCE instances is not possible)

MapREduce offers a much more limited expressiveness than SQL due to its fixed grouping operations and limited analytical features.

MapReduce re-invents the wheel as many of the above-mentioned deficiencies of MapReduce already have been solved in the past in the research on distributed databases.

Reference

DeWitt, D. MapReduce: A major step backwards. The Database Column, 2008. http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html

What is your opinion? Ours is as follows:

Some of the critique is over the top

- MapReduce does address data skew (with its COMBINERS), though it needs user intervention.

Most of the critique aims at a wrong setting, for example:

- Is indexing really an alternative for billions of documents and pera-bytes of data?
- Have parallel DB proven to be as scalable? Even the largest Teradata (the most used parallel database system) is significantly below 1000 nodes.

The simplicity of MapReduce is often an advantage because it is simple enough to be quickly understood, easliy deployed and to have easy strategies for failure recovery.

As usual, the right tool has to be chosen for the right task. MapReduce is appropriate for certain tasks characterized by very large, volatile data the indexing of which would be too expensive or too hard, and data that might significantly increase. MapReduce has proven itselve to scale well and easily, this is not the case of database systems. Meaningful applications of MapReduce are therefore Web index, log analysis, and PageRank computation.

If the data is more controlled, rarely updated, and/or of smaller size, and if a dozen of machines then the classical database technology a better choice because if its indexing, declarative query languages, index and query optimizers, and also because if its maturity.

MapReduce and the database technology have been copmatred on benchmarks:

Pavlo, A., Paulson, E., Rasin, A., Abadi, D., DeWitt, D. J., Madden, S., Stonebraker, M. 2009. A Comparison of Approaches to Large-Scale Data Analysis. Proc. of ACM SIGMOD Symposium on Management of Data. http://database.cs.brown.edu/sigmod09/
http://database.cs.brown.edu/sigmod09/benchmarks-sigmod09.pdf

The rough results of this investigation is that the database technology is often faster. However, the test used only 100 nodes and the tasks PageRank computation GIVE FIGURES?, MapREduce is competitive.

A piossible future might be in the integration of MapREduce with the database technology, or in or more expressive MapReduce extensions such as Microsoft's Dryad REFERENCE?.