**Physical**

1. Firstly, you need some servers. The amount will depend on how much content you plan to index
2. You'll need an internet connection
3. You should have a firewall
4. You should have a Web application firewall to protect your data
5. You'll need a load balancer for your application servers - both the web publisher and possibly your database servers

**Applications - the search web application**

* Note: most people think that search and ranking happen when you do a search. Its actually already done. To make search so fast, the search layer just returns the next 10 results from a list starting at position 1, and the only logic applied is (a) geo-specific pages are removed, (b) QDF checks pull in items based on a date and time range and (c) real-time data is pulled from a queue which has it ready to go
* Essentially you want to break the HTML user interface from the business logic and the databases (the index)
* I would imagine a 3 tier where you have a very basic Apache (or IIS) web application that lets a user enter a search query and then it fires it the search application which sends the results to another 2nd User interface. That way should there be peak demand you can basically disconnect the user operation servers from the functioning search server
* The business logic layer - the "search" should simply have 2 functions
* Function A - strip the search string in a query of non-value words (a, the, an, and, it, is) where each word is searched and a second query where its searched in different %'s of completeness. E.g. a search for "how much does an iPhone cost" would [much or does or iphone or cost] and also [much or does or iphone] and [much or does or cost]
* The query is then handed to the next available search parser. The search parser connects to the next available database server. this is where load balancing comes in - if any single server becomes non-responsive, the load balancer just removes it from the queue and connects the next available healthy server
* The parser requests the top 10 results with those words and is given a list of key IDs which each match a URL
* The magic of rank order isn't established during a search - search is the final leg.

**Applications - Crawl List**

1. You need a URL list, which you would need to mark with last crawled dates, time to crawl/page speed, robots and meta index allowed status
2. You would also want to store a rank and geo-location and a language identifier
3. Your crawl servers do the clever work
4. The crawl list is made up from a multitude of sources including: crawling and processing files and pages found on a server, inside a file (i.e. links from other pages), from URL's accessed by a browser, toolbar app, or a submit page on your search app
5. Lastly, you would store a list matrix of the pages that link to this page and a value of those pages which is in turn based on the number of links they have. you might also build in some kind of penalty/spam feature at this point
6. You would then have a realtime source list - e.g. news sites that you either allow a direct URL submit (e.g. look at pingomatic for WordPress: [Ping-o-Matic!](http://pingomatic.com/))
7. Remember submitted URLs save time - so if you have a trust indicator or protocol established, then trusted URLs would be auto-crawled super faster - because there is less time spent on discovery. Once you crawl it, you mark it as crawled, thus if the url is found during another process,, well, unless its refresh-by date hasn't passed it doesn't need to be recrawled.
8. You would then have a scheduler which would be a view or table of that data in terms of which pages and domains are prioritized first
9. You would need multiple crawlers if you were trying to index the web or a large piece of it
10. some servers will work on a dedicated list - when the list is complete they go back to the top
11. Other servers will kick off a crawl based on minutes hours or days - that way highly popular/authoritative or frequently used content is kept very fresh, and new pages are added and indexed quickly
12. Fringe content on lower authority sites could take weeks/months to be crawled. And that's a bit of a cheat but it works.

**Crawl and processing**

* Each page then needs to be requested. The date stamp includes a start and end timestamp
* Again, load balancing is helpful here because it can route the reply from the crawled server to a different process, essentially a HTTP file receiver. Because there are many slow servers and because many other servers become slow - due to hosting too many sites, network clogging, bad architecture, you don't want the crawl requester to be delayed - needs to send out billions of URLs across to web servers as requests. By divorcing the request and receive function you remove bottlenecks. The file receiving servers can accept lots of slow responsive servers because the data demand is low. Other servers can be separated for highly responsive servers because they will quickly reach maximum connection counts. Again, load balancers will distribute that between those servers based on weighting, round-robin, health checking
* Another crawler will use good old-fashioned server directory querying to see if the webserver exposes files and folder lists and it will create pseudo URL's and add them to the crawl list
* I would guess that once the page is saved in a caching or staging area, multiple processes grab it rather than one large sequential processor. Sometimes you can see the impact of this when during one search, a result contains more information in its "snippet" than it did in a slightly earlier search
* When a page is downloaded, either wholly or in part after a set timeout based on time and retries., all of the applications that scan, process, and index pages get stuck in
* A language scanner will determine the language based on a dictionary lookup
* the page is largely stripped of scripts and HTML formatting. The ones that are kept are ones you need for the purposes of determining relevancy. This might probably include the page title, the content (which is all text that isn't very script)
* A key scanner type is one that would look for code and security flaws, malware and the like
* I would imagine you would also have a spam scanner which request a copy and scan for very spammy type things
* Another parser would build a searchable page - content stripped out
* Each application works independently and some take longer than others (although this really is split second stuff)
* Also, any javascript, non-html files, images etc automtically get blocked - even if the remote/client web server sends them - they routed to a different port which is essentially a Null or digital black hole.
* I would imagine the servers that receive files will apply a process to present the HTML in a format that's low but provides the same basic minimum viable product to each sniffer or parser. For example -if it receives the <body> chunk - it might replace complicated words with simplified language. For example, it might replace every instance of motor-car, automotive car and car with just "car"
* Again instead of just a queue, I would image that load balancer are able to set ordering based on the importance
* You then have very big servers that you deploy either quickly behind or very far ahead of the curve of the growing amount of data you're planning to index (again, modeling on Google). These just process all of the lower-triage pages that arent flagged as highly important. Thus you can manage the frequency of very important content very acutely. If your QDF, news, twitter feed - high volume content represents 10% of content and then suddenly spikes, you just take another 5% of your "generalist servers" and route them until that spike point is over. If the total time to refresh is on average 3 weeks and then one day you notice that the longest refresh was 5 weeks, then you just need to add in an extra 20% capacity at some point. You're not adding it in immediately because your instant indexing servers have auto-scale
* Auto-scale gives elasticity to ride out demands - and that's why you break components into non-asynchronous processes that don't rely on each component to finish. You build out work queues based on the smallest set of procedures that have to be completed in the same order, the very minimalist list you can.
* Another example would be a URL sniffer that also requests a copy of the page. Firstly, it will take the URLs and then add that pages KID to records of the URL's it links to, based on say an industry-agreed standard like nofollow, which will prevent that KID from registering as a link. The sniffer will also look at the anchor text in that link. Whether or not that page has that text, the sniffer will record it in an index of relevance. A separate thread or process will query based on a synonym dictionary
* and then magic happens - an indexer will then list that urls Primary Key in all of the lists. Imagine a list called "iPhone" and in it are every page in the world with the word iPhone, a count of how often it appears, what countries its relevant to, what language it is. based on whatever signals your algorithm needs, you then immediately score that page
* The page is also added to another similar-sounding/meaning indices
* Remember, at this point, your index copy of the web is just hundreds of single word tables: car, book, iphone, pen, mouse, dell, server, kemp, load, balancer
* Some indices will span spelling mistakes and if a query for balancer comes in, it runs off the same one as balancing or you can keep them separate and score the pages higher or lower depending on which version of that word is more or less relevant. I would imagine Google started with this and now have a hybrid version where some are collapsed (joined together) and others are separated.

**The results application**

Each query processed will be assigned to a result set. the input into this is the query phrase, the country and whatever personalization settings it needs to know. The results from the index are then re-prioritised accordingly.

**Ads Server**

An ads server will list the KID's of any ads and the targeted keywords and negatives. When the query is passed to the results server, the ads server will get a copy of that request and give HTML back to the Result Application Web server which gives a HTML page back to the user with the organic and paid results. All of that is done in 0.0001 seconds or faster if possible