Combining Systems and Databases: A Search Engine Retrospective

1 author:

Eric Brewer
University of California, Berkeley
241 PUBLICATIONS **18,321** CITATIONS

From: Readings in Database Systems, Fourth Edition.
Joseph M. Hellerstein and Michael Stonebraker eds, MIT Press, Cambridge, MA, 2005.

Combining Systems and Databases: A Search Engine Retrospective

Eric A. Brewer
University of California at Berkeley

Although a search engine manages a great deal of data and responds to queries, it is not accurately described as a "database" or DMBS. We believe that it represents the first of many application-specific data systems built by the systems community that must exploit the principles of databases without necessarily using the (current) database implementations. In this paper, we present how a search engine should have been designed in hindsight. Although much of the material has not been presented before, the contribution is not in the specific design, but rather in the combination of principles from the largely independent disciplines of "systems" and "databases." Thus we present the design using the ideas and vocabulary of the database community as a model of how to design data-intensive systems. We then draw some conclusions about the application of database principles to other "out of the box" data-intensive systems.

1 Introduction

Search engines (SEs) are arguably the largest data management systems in the world; although there are larger databases in total storage there is nothing close in query volume. A modern search engine handles over 3 billion documents, involving on the order of 10TB of data, and handles upwards of 150 million queries per day, with peaks of several thousand queries per second.

This retrospective is based primarily on almost nine years of work on the Inktomi search engine, from the summer of 1994 through the spring of 2003. It also reflects some the general issues and approaches of other major search engines — in particular, those of Alta Vista, Infoseek and Google — although their actual specifics might differ greatly from the examples here.

Although queries tend to be short, there are more than ten million different words in nearly all languages. This is a challenge for two reasons. First, implies tracking and ranking ten million distinct words in three billion documents including the position and relative importance (e.g. title words) of every word. Second, with so few words per query, most queries returns thousands of hits and ranking these hits becomes the primary challenge.

Finally, search engines must be highly available and fresh, two complex and challenging data management issues. Downtime contributes directly to lost revenue and customer churn. Freshness is the challenge of keeping the index up to date with the data, nearly all of which is remote and relatively awkward to access. Most data is "crawled" using the HTTP protocol and some automation, although there is also some data exchange via XML.

Despite the size and complexity of these systems, they make almost no use of DBMS systems. There are many reasons for this, which we cover at the end, but the core hypothesis here is that, looking back, search engines should have used the *principles* of databases, but not the artifacts, and that other novel data-intensive systems should do the same (covered in Section 8).

These principles include:

Top-Down Design: The traditional systems methodology is "bottom up" in order to deliver capabilities to unknown applications. However, DBMSs are designed "top down", starting with the desired semantics (e.g. ACID) and developing the mechanisms to implement those semantics. SEs are also "whole" designs in this way; the semantics are different (covered below), but the mechanisms should follow from the semantics.

Data Independence: Data exists in sets without pointers. This allows evolution of representation and storage, and simplifies recovery and fault tolerance.

Declarative Query Language: the use of language to define queries that says "what" to return not "how" to compute it. The absence of "how" is the freedom that enables powerful query optimizations. We do not however use SQL (a DBMS artifact), but we do use the structure of a DBMS, with a query parser and rewriter, a query optimizer, and a query executor. We also define a logical query plan separate from the physical query plan.

The fundamental problem with using a DBMS for a search engine is that there is a semantic mismatch. The practical problem was that they were remarkably slow: experiments we performed in 1996 on Informix, which had cluster support, were an order of magnitude slower than the hand-built prototype, primarily due to the amount of specialization that we could apply (see Section 8). Most modern databases now directly support text search, which is sufficient for most search applications, although probably not for Yahoo! or Google.[1]

The semantics for a DBMS start with the goals of consistent, durable data, codified in the ideas of ACID transactions [GR97]. However, ACID transactions are not the right semantics for search engines.

First, as with other online services, there is a preference for high availability over consistency. The CAP Theorem [FB99,GL02] shows that a shared-data system must choose at most two of the following three properties: consistency, availability, and tolerance to partitions. This implies that for a wide-area system you have to choose between consistency and availability (in the presence of faults), and SEs choose availability, while DBMSs choose consistency.[2] In addition, the index is *always* stale to some degree, since updates to sites do not immediately affect the index. The explicit goal of freshness is to reduce the degree of inconsistency.

Second, SEs can avoid a general-purpose update mechanism, which makes isolation trivial. In particular, queries never cause updates, they are all read only. This implies query handling (almost) never deals with atomicity, isolation, or durability. Instead, updates are limited to atomic replacement of tables (covered in Section 5.2), and only that code deals with atomicity and isolation. Durability is even easier, since the SE is never the master copy: any lost data can generally be rebuilt from local copies or even recrawled (which is how it is refreshed anyway).

We start with an overview of the top-down design, followed by coverage of the query plan and implementation in Sections 3 and 4. Section 5 looks at updates, Section 6 at fault tolerance and availability, and Section 7 at a range of other issues. Finally, we take a broader look at data-intensive systems in Section 8.

2 Overview

In a traditional database, the focus is on a general-purpose framework for a wide variety of queries with much of the effort expended on data consistency in the presence of concurrent updates. Here we focus on supporting many concurrent read-only queries, with very little variation in the range of queries, and we focus on availability more than consistency.

These constraints lead to an architecture that uses an essentially static database to serve all of the read-only queries, and a large degree of offline work to build and rebuild the static databases. The primary advantage of moving nearly everything offline is that it greatly simplifies the online server and thus improves availability, scalability and cost.

We believe that most highly available servers should follow this "snapshot" architecture — the server uses a simple snapshot of the data, while most work, such as indexing, can be done offline without concern for availability. For example, any work done offline can be started and stopped at will, has a simple "start over" model for recovery, and in general is very low stress to modify and operate since these efforts are not visible to end users.

2.1 Crawl, Index, Serve

The first step is to "crawl" the documents, which amounts to visiting pages in essentially the same way as an end user. The *crawler* outputs collections of documents, typically a single file with a map at the beginning and thousands of concatenated documents. The use of large files improves system throughput, amortizes seek and directory operations, and simplifies management.[3] The crawler must keep track of which pages have been crawled or are in progress, how often to recrawl, and must have some understanding of mirrors, dynamic pages, and MIME types.

The *indexer* parses and interprets collections of documents. Its output is a portion of the static database, called a *chunk*, that reflects all of the scoring and normalization for those documents. In general, the goal is to move work from the (online) web servers to the indexer, so that the servers have the absolute minimum amount of work to do per query. For example, the indexer does all of the work of scoring, generating typically a single normalized score for every word in every document. The indexer does many other document analyses as well: determining the primary language and geographical region, checking for spam, and tracking incoming and outgoing links (used for scoring). One of the more interesting and challenging tasks is to track all of the *anchor text* for a document, which is the hyperlink text in *all other* (!) documents that point to this document.

Finally, the *server* simply executes queries against a collection of chunks. It performs query parsing and rewriting, query optimization, and query execution. Since the only update operation is the atomic replacement of a chunk (covered in Section 5), there are no locks, no isolation issues, and no need for concurrency control for queries.

2.2 Queries

Conceptually, a query defines some words and properties that a matching document should or should not contain. A *document* is normally a web page, but could also be a news article or an e-mail message. Each document is presumed unique, has a unique ID (`DocID`), a URL and some summary information.

Documents contain *words* and have *properties*. We distinguish words from properties in that words have a

1: As an aside, the databases were also very expensive. However, as we were among the first to build large web-database systems, we were charged per "seat", which in the fine print came down to distinct UNIX user IDs. But *all* of the end users were multiplexed onto one user ID, so this was quite reasonable! Later the database companies changes the definition of "user" and this trick was no longer valid.

2: Wide-area databases vary in their choice between availability and consistency. Those that choose availability operate some locations with stale data in the presence of partitions and generally have a small window that is stale (inconsistent) during normal operation (typically 30 seconds) [SAS+96]. Those that choose consistency must make one side of a partition unavailable until the partition is repaired.

3: In theory, a DBMS could be used for document storage, but it would be a poor fit. Documents have a single writer, are only dealt with in large groups, and have essentially no concurrent access. See the Google File System [GGL03] for more on these issues.

Property	Meaning
lang:english	doc is in english
cont:java	contains java applet
cont:image	contains an image
at:berkeley.edu	domain suffix is berkeley.edu
is:news	is a news article

Table 1: Example Properties

score (for this document) and properties are boolean (present or absent in the document). Table 1 lists some examples. A query *term* is a word or a property.

Simple queries are just a list of terms that matching documents must contain. Property matching is absolute: a matching document must meet all properties. Word matching is relative: documents receive relative scores based on how well they match the words.

Complex queries include boolean expressions of terms based on AND, OR and NOT. Boolean expressions for properties are straightforward, but those for words are not. In particular, the expression (NOT *word*) should not affect the scoring; it is really a property.

We cover scoring in more detail in the appendix, but for now we will use simple definitions. A query is just a set of terms:

$$\text{Query } Q \equiv \{w_1, w_2, \ldots, w_k\} \tag{1}$$

The *score* of a document *d* for query *Q* is the sum of an overall score for the document and a score for each term in the query:

$$\begin{aligned} \text{Score}(Q, d) &\equiv \text{Quality}(d) \\ &\quad + \sum_i \text{Score}(w_i, d) \end{aligned} \tag{2}$$

The quality term is independent of the query words and reflects things like length (shorter is generally better), popularity, incoming links, quality of the containing site, and external reviews. The score for each word is a determined at index time and depends on frequency and location (such as in the title or headings, or bold).

There are some important non-obvious uses for words. In general, any property of a document that is not boolean is represented by a *metaword*. Metawords are artificial words that we add to a document to encode an affine property. For example, to encode how frequently a document contains images (rather than just yes or no), we add a metaword whose score reflects the frequency. You can use this trick to encode many other properties, such as overall document quality, number of incoming or outgoing links, freshness, complexity, reading level, etc. Implicitly, these metrics are all on the same scale, but we can change the weighting at query time to control how to mix them.

Document table, *D*, about 3B rows

DocId	URL	Date	Size	Abstract

Word table, about 1T rows:

WordID	DocId	Score	Position Info

Property table, about 100B rows:

WordID	DocId

Term table, *T*, about 10M rows:

String	WordID	Stats

Figure 1: Basic Schema

3 Logical Query Plan

Given this simplified scoring, we turn to how to map a query into a query plan. This section looks at the logical query plan and the next section looks at the physical operators and plan implementation.

In the original development of this work, we were not cognizant that we were defining a declarative query language and that it should have a query plan, an optimizer, and a rewriter, and that we should cleanly separate the logical and physical query operators and plans. We did know we needed a parser. The absence of this view led to a very complicated parser that did ad-hoc versions of query rewriting and planning, and some optimization. The use of an abstract logical query plan is one of the important principles to take from database systems, and hence we retrospectively present the work based on a clean logical query plan.

For simplicity, we will limit the schema to three (large) tables: document info, word data and property data. Figure 1 shows the schema. Tables that we ignore include those for logging (one row per search), advertising, and users (for personalization); we talk about some of these in Section 7.

To simplify dealing with words and properties, we conceptually use an integer key for each word, the WordID. The term table, *T*, maps from the string of the term to the WordID for that word, and also keeps statistics about the word (or property). The stats are used for both scoring and to compute the *selectivity* for query optimization.[4] The simplest useful stat is the number of rows in the table, which tells you how common the term is in the corpus; high counts imply high selectivity and lower scores (since the word is common). Note that the

4: Selectivity is the fraction of the input that ends up in the output, and is thus a real number in the interval [0,1]. Ideally, a query plan should apply joins with low selectivity first, since they reduce the data for future joins. With multi-way equijoins (and semijoins), this is less important since we aim to do them all at once. Of great confusion to many is that *high* selectivity numbers imply the operation is *not* very "selective" in the normal English usage of the word.

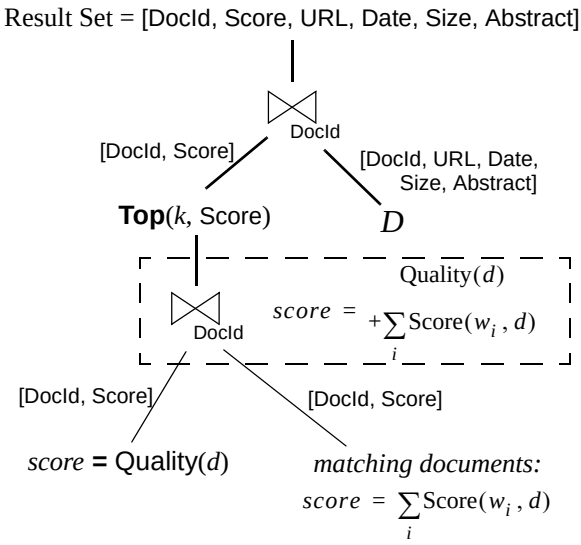

Figure 2: The General Query Plan
After finding the set of matching documents and their scores, the **Top** operator passes up the top *k* results (in order) to an equijoin that adds in the document information.

term table is only used during query planning and is never referenced in the query itself.

Figure 2 shows the general plan for all queries. The fact that all queries have essentially the same plan is a significant simplification over general-purpose databases, and a key design advantage. The equijoins ($\bowtie_{\text{DocId}}$) join rows from each side that match on DocId, resulting in an output row that has the union of the columns. We refer to the top equijoin as the "document join," since it joins the top results with their document data.

The `Top`(*k*, *column*) operator returns the top *k* items from the input set ordered on *column*; it is not a common database operator, although it appears in some extensions to SQL and in the literature [CG99,CK98]. The input to `Top` is the set of fully scored documents, which combines the document quality score with the sum of the word scores from the matching documents. `Top` can be implemented in O(*n*) time using insertion sort into a *k*-element array.[5]

The next task is to produce the set of matching documents and their scores. Applying the top-down principle, we design a small query language for this application, rather than using SQL. Here is the BNF for one possible query language:

```
expr:  expr AND expr
     | expr OR expr
     | expr FILTER prop
     | word
```

5: Insertion sort is normally O(*n lg n*), but since we only keep a constant number of results, *k*, we have a constant amount of work for each of the *n* insertions.

Operator	Meaning
e AND e	Equijoin with scoring
e OR e	Full outer join with scoring
e FILTER p	Semijoin: filter e by p
p AND p	Equijoin without scoring
p OR p	Full outer join without scoring
NOT p	s antijoin p (invert the set s)
NOT e	s antijoin e (invert, omit score)

Table 2: Logical Operators

```
prop:  prop AND prop
     | prop OR prop
     | NOT prop
     | NOT expr
     | property
```

There are seven corresponding logical operators as shown in Table 2. In this particular grammar, *expr* nodes have scores and *prop* nodes do not. The only way to join an *expr* and a *prop* is through the `FILTER` operator, which filters the word list on the left with the property list on the right. Note that the logical negation for an *expr*, `NOT e`, is a *prop* and not an *expr*. This is because there is no score for the documents not in the set. This implies that it is not possible to ask for "-foo" as a top-level query (i.e. the set of all documents that do not contain "foo"). In practice, we actually do allow properties and negated expressions as top-level queries, which are useful for debugging. Note that the `Top` operator (with query optimization) saves us from having to return (nearly) the whole database.

Normal queries are just a sequence of words with the implicit operator `AND` between them. For example, the query:

```
san francisco
```

maps to `(san AND francisco)`. For properties:

```
bay area lang:english
```

maps to `((bay AND area) FILTER lang:english)`, which is the set of english-language documents that contain both words. A minus sign preceding a word normally means negation, so that:

```
bay area -hudson
```

maps to `((bay AND area) FILTER NOT hudson)`.

More complex queries usually come from an "Advanced Search" page with a form-based UI, or from a test interface that is amenable to scripting. We will use the parenthesized representation directly for these queries.

4 Query Implementation

Given the scoring functions and the logical operators, we next look at query optimization and the map-

ping of logical operators onto the physical operators. We start by defining the physical operators and then show how to map the logical operators and some possible optimizations. We finish with parallelization of the plan for execution on a cluster.

4.1 Access Methods and Physical Operators

There is really only one kind of access method: sequential scan of a sorted *inverted index*, which is just a sorted list of all of the documents that contain a given term. For properties, this is just the sorted list of documents; for expressions we add the score for each document. A useful invariant is to make expressions a subclass of properties, so that an expression list can be used for any argument expecting a property list. For example, this means we do not need a separate negation operator for expressions and properties. We cover the physical layout of the tables in Section 4.3, when we discuss implementation on a cluster.

An unusual aspect of the physical plan is that we cache all of the intermediate values (for use by other queries), and do not pipeline the plan. Caching works particularly well, since there are no updates in normal operation (updates are covered in Section 5). Given that we keep all intermediate results, there is no space savings for pipelining. Pipelining could still be used to reduce query latency, but we care more about throughput than latency, and throughput is higher without pipelining, due to lower per-tuple overhead and better (memory) cache behavior. Thus, we increase the latency of a single query that is not cached, but reduce the average latency (with caching) and increase throughput. [6]

Because the lists are sorted, binary operators become merging operations: every join is a simple (presorted) merge join. In fact, there is no reason to do binary operators: every join is a multiway merge join. The use of multiway joins is a win because it reduces the depth of the plan and thus the number of caching and scan steps (remember that intermediate results are not pipelined). In addition, it is useful to move negation into the multiway join as well, since the antijoin is a simple variation of a merge join. A consequence of this is that for every input to the multiway join, we add a boolean argument to indicate the positive or negative version of the input.

This leads us to have only four physical operators:

`OR`(e_1, e_2, ... e_k)$\rightarrow$ *expr*

Compute the full outer multiway join, with scoring. We have left out the boolean flags; we will use "¬*e*" as the input when we mean the negation.

`ORp`(p_1, p_2, ... p_k)$\rightarrow$ *prop*

Multiway full outer join without scoring.

`ANDp`(p_1, p_2, ... p_k)$\rightarrow$ *prop*

Multiway inner join without scoring

`FILTER`(e_1, e_2, ... e_k)(p_1 ... p_n) $\rightarrow$ *expr*

Multiway inner join with scoring for the expressions and no scoring for the properties.

Most queries map onto a one-deep plan using `FILTER`. It is essentially an `AND` of all of its inputs, with only the expressions used to compute the score. It implements (e `AND` e) if there are no properties. Although it could also subsume (p `AND` p), it is better to use `ANDp`, since the latter returns a property list rather than an expression list which avoids space for unused scores. Figure 3 shows some example queries with their logical and physical plans.

One nice property of using multiway joins is that it mitigates the need for estimating selectivity. Selectivity estimation is normally needed to compute the size of an input to another operator; increasing the fan in of an (inner) join limits the work to the actual size of the smallest input and thus decreases the need for estimates. For example, for `FILTER` and `ANDp`, the output is limited by the size of the smallest input (lowest selectivity). Thus selectivity only matters when we cannot flatten a subgraph to use a multiway join.

4.2 Query Optimizer

The optimizer has three primary tasks: map the logical query (including negations), exploit cached results, and minimize the number of joins by using large multiway joins. As expected these optimizations often conflict, leading us to either heuristics or simple models (as done in traditional optimizers). The basic heuristic is to focus first on caching, second on flattening (using larger multiway joins), and third on everything else.

The focus on caching all subexpressions leads to the atypical decision of using a top-down optimizer [Gra95], rather than the bottom-up style that is standard for traditional databases [Sel+79]. Although either could be made to work, the top-down approach makes it easy to find the highest cached subexpression: we simply check the cache as we expand downward. The bottom-up

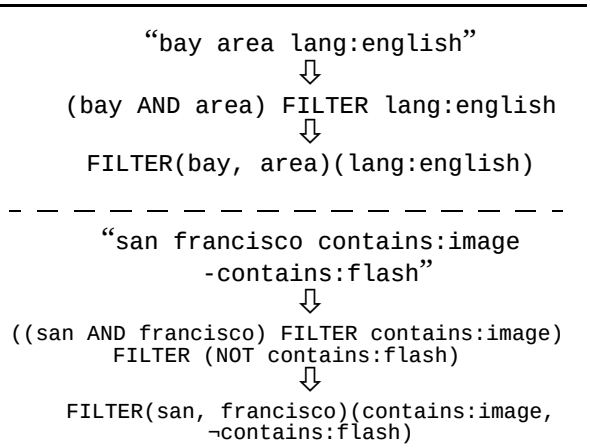

Figure 3: Two example queries and their logical and physical plans

6: In theory, we could choose to not cache some intermediate results that we believe unlikely to be used again, and pipeline the results, but this is not worth the extra complexity.

approach implies building many partial solutions as part of dynamic programming that are unnecessary if they are part of a cached subexpression: that is, you build up best subtrees before you realize that they are cached. Both approaches typically require building partial solutions and exploring parts of the space that are not used for the final plan.

The basic mapping of logical operators is straightforward, with the only subtlety being how to map negations:

```
a AND ¬b ⇨ FILTER(a)(¬b)

¬a AND ¬b ⇨ ANDp(¬a, ¬b)

a OR ¬b ⇨ OR(a, ¬b)

¬a OR ¬b ⇨ ORp(¬a, ¬b)
```

Note the use of `ANDp` and `ORp` when the output is a property and not an expression.

Given a correct tree of physical operators, the next step is to optimize it, which consists mostly of flattening the tree to use fewer but wider joins. The first step is to flatten all chains of pure AND or pure OR, making liberal use of the commutative and associative properties. For example:

```
(((a AND b) AND c) OR d)
  ⇨ OR(AND(a, b, c), d)

FILTER(a, FILTER(b)(c))(d)
  ⇨ FILTER(a, b)(c, d)
```

We can do more complicated forms of flattening by using DeMorgan's Law, which allows us to convert between and AND and OR operations. The basic conversion is:

```
a AND b ⇨ ¬(¬a OR ¬b)

a OR b ⇨ ¬(¬a AND ¬b)
```

An example use for flattening:

```
a AND b AND NOT (c OR d) ⇨
  AND(a, b, AND(¬c, ¬d)) ⇨
  FILTER(a, b)(¬c, ¬d)
```

However, we have to be careful to keep score information when applying DeMorgan's Law. For example, is `(a OR ¬b)` an expression or a property? If it is an expression, not all of the elements of the set have scores and we must make some up (typically zero). If it is a property, then we forfeit the scoring information (from `a`) for later expressions. Either policy can be made to work, although more flattening is possible when treating this case as a property, since otherwise `(a OR ¬b)` ≠ `¬ANDp(¬a,b)`.

The current heuristic is to flatten completely before looking for cached subexpressions, which is part of the general philosophy of using canonical representations for trees to ensure that only one form of a subexpression could appear in the cache. As a consequence, for a large multiway join, we must look for subsets of the terms in the cache. We first look for the whole *k*-way join, then for each *k-1* subset, then each *k-2* subset, until we get to individual terms. Thus, if `FILTER(a,b,e)` and `FILTER(d,f)` were in cache (but larger subsets were not), we might map:

```
FILTER(a, b, c, d, e, f) ⇨
  FILTER(FILTER(a,b,e), c, FILTER(d,f))
```

One useful aspect of the cache is that we can keep the size of the cached set as part of its metadata, which allows us to exploit selectivity. In particular, given two overlapping sets in cache that each represent *k* terms, we choose the one with the smaller set size, since it is probably more selective. In the above example, if `FILTER(a,c,d)` was also in cache, we would select between it and `FILTER(a,b,e)` based either on selectivity, or the degree of caching of the remaining terms, or both. It is in the exploration of the remaining terms that the top-down approach may explore parts of the plan space that are not used.

As with traditional databases there are many other possible optimizations with increasing complexity, which we ignore here. To give the flavor of these, consider in the example above if `c` was not in cache, but `FILTER(c)(e)` was, then the latter could be used instead, since `e` is part of the larger conjunctive join, and ANDing it twice won't affect the final set. However, `FILTER(c,e)()` is not an acceptable replacement, since the score for `e` would be counted twice.[7]

4.3 Implementation on a Cluster

Once we have the optimized tree, we must map the query onto the cluster. The approach we take is to exploit symmetry, which simplifies design and administration of the cluster. In particular, the bulk of every query goes to every node and executes the same code on different data, as in the SPMD model from parallel computing [DGNP88].

From the database perspective, this means a mixture of replication for small tables and horizontal fragmentation (also known as "range partitioning") for large tables. In particular, the document, word and property tables are all horizontally fragmented by `DocID`, so that a self-contained set of documents (with a contiguous ranges of `DocIDs`) resides on each node. This structure simplifies updates to documents, and also makes it easy to mix nodes of different power, since we can give more powerful nodes more documents. The term table, which maps term strings to WordIDs, is replicated on each node, and we use global values for the WordIDs, so that WordIDs can be used in physical queries instead of the strings.

An important point is that the `DocID` is essentially random relative to the URL (such as a 128-bit MD5 or CRC), which means that documents are randomly spread across the cluster, which is important for load balancing and caching.

7: Under overload conditions, this might be an acceptable replacement, since it will be a small reranking of the same set of documents, and the scoring function is always magic to some degree anyway.

The word and property tables are (pre)sorted by WordID for that node's set of DocIDs, so that they are ready for sorted merge joins. We maintain a hash index on WordID for these tables to locate the beginning of the inverted index for each term. It is somewhat easier to think of the word and property tables as sets of "sub-tables", one for each WordID. This is because each sub-table is independently compressed and cached, as described below; the hash index on WordID is thus really in index of the sub-tables, and keeps track of whether or not they are in memory.

Initially, using a load balancer, a query is routed to exactly one node, called the *master* for that query. Most nodes will be the master for some queries and a *follower* for the others. The master node computes and optimizes the query plan, issues the query to all other nodes (the followers), and collates the results. Each follower computes *its* top *k* results, and the master then computes the top *k* overall. Finally, the last equijoin with the document table, *D*, is done via a distributed hash join with one lookup for each of the *k* results (which may also be cached locally). This is really a "fetch matches" join in the style of Mackert and Lohman [ML86], which means that you simply fetch the matching tuples on demand rather than doing any kind of movement or repartitioning of the table.

Using the master to compute the query plan has some subtle issues. The primary advantage is that the plan is computed only once, and all followers execute only physical queries. However, the cache contents are not guaranteed to be the same, since different nodes may have different amounts of cache space. In practice, the cache size is always proportional to the size of the fragment, so the contents usually agree, but not always. For example, a cache entry would be larger than usual if that node has more occurrences of a particular word, which may force something else out; however, since documents are spread randomly this effect tends to even out. Nonetheless, a follower may have to recompute something that the master expected to be in cache.

4.4 Other Optimizations

In addition to traditional database optimizations, search engines exploit some unusual tricks that merit discussion. We cover three of them here.

One of the most important optimizations is compression of inverted indices. Although compression has been covered in the literature [CGK01], it is not widely used in any major DBMS. It makes more sense for a search engine for a few reasons. First, there is no random access for these sub-tables, they are always scanned in their entirety as part of a sorted merge join. (The exception is the document table, which is random access and is not compressed.) Second, there are no updates to the tables, only whole replacement, so there is no issue of how to update a compressed table. The simplest good compression scheme is to use relative numbering for `DocIDs`, since they are in sorted order and the density may be high. This requires many fewer bits than the 32+ it would require to store the whole `DocID`. Similarly, it is important to make good use of all of the scoring bits, which can be done by a transformation of the scoring function.

It turns out that the compression not only increases the effective disk bandwidth, but also the cache size. By keeping the in-memory representation compressed, we increase the cache hit rate at the expense of having to decompress the table on every use. This turns out to be an excellent tradeoff, since modern processors can easily do the decompression on the fly without limiting the off-chip memory bandwidth, and the cost of a cache miss is millions of cycles (since it goes to disk).

A related optimization is preloading the cache on startup. This turns out to be pretty simple to do and greatly reduces the mean-time-to-repair for a node that goes down (for whatever reason). In the case of a graceful shutdown of the process, the node can write out its cache contents, and even reuse the memory via the file cache when the new process starts. For an unexpected shut down, the process can use an older snapshot of the cache, but will have to page it in, which is still faster that recomputing it (which requires reading all of the constituent tables). The primary limitation is that the snapshot must match the current version of the database; both are marked with version numbers for this reason.

Finally, the use of a master node enables a powerful kind of optimization based on the classic A* search algorithm (from AI) [RN02], which employs a conservative heuristic to prune the search space. In particular, instead of simply sending out the query, the master executes the query locally first (which adds some latency), and computes its top *k* local results, which it will have compute at some point anyway. The score of the k^{th} local result is a conservative lower bound on the scores of the overall top *k* results. In particular, followers need not pursue any subquery that cannot beat this lower bound. For example, for a term in a multiway join, there is typically some score below which you need not perform the join, since even with the best values for the other terms the end score will not make the top *k*. Similarly, by keeping track of the best score for the whole table, we may be able to eliminate whole terms.

5 Updates

Although search engines are clearly read mostly, at some point we actually need to update the data. One huge benefit of the top-down strategy is that we can exploit our complete control over the timing and scope of updates.

We follow a few basic principles for updates. First, nodes are independent, so we can update one node without concern for the impact on other nodes. Replicas are clearly an exception to this, since they must be updated together, but their group is independent of other groups. Second, we only update whole tables and not individual rows. This means that we never insert, update or delete a row, and that we need at most one lock for the whole

table. Third, updates should be atomic with respect to queries; that is, updates always occur between queries.

To simplify updates, we define a chunk to be the unit for atomic updates. Earlier we mentioned that the tables are partitioned by `DocID` among the nodes, but it is more accurate to say that the databases are partitioned into chunks, and that a node contains a contiguous range of chunks. Each chunk is a self-contained collection of documents with their word and property tables.

In practice, it is useful to split the cluster into multiple databases, called *partitions*. This allows each partition to have its own policies for replication and freshness. A query still goes to all nodes (of all partitions), and the `DocIDs` and `WordIDs` are still globally unique. For replicated partitions, which normally have two replicas, each node has the same chunks as it replica(s), and only one member of the replica group receives any given query in the normal case.

The next two sections looks at the creation and installation of chunks, and the following two look at more complex types of updates.

5.1 Crawling and Indexing

The first step for an update is to get the new content, which is usually done by crawling: visiting every document to verify that we have the current version, and retrieving a new version if we do not. Indexing is the process of converting a collection of documents into a chunk, and includes parsing and scoring, and the management of metadata, such as tracking incoming and outgoing links.

It is easiest to think of a chunk as a *range* of `DocID` values, which means that a chunk does not have a specific size per se, but rather an average size. This definition simplifies the addition and removal of documents from a chunk, since there is no effect on neighboring chunks. As a database grows, the average chunk size will grow until it reaches some threshold at which point it may be split into two or more chunks.

The simplest kind of crawl simply refreshes all of the documents in one chunk, and then reindexes them. Some documents may have to be recrawled multiple times if their site is down, or they can be left out for this version and recrawled next time, although eventually they are permanently removed.

The refresh rate is a property of the partition, and thus a property of all of its chunks. News partitions may be updated every fifteen minutes, while slow-changing content, such as home pages, may be refreshed every two weeks (or longer).

Document discovery, which is the process of finding new documents for the database, is primarily a separate process, although outgoing links are the main source of new documents. A separate database tracks metadata about all of the sites, including new links and global properties about spam sites, mirrors, paid content, etc. New documents can be added to existing chunks when they are next refreshed, or may be added to a new chunk in a separate partition, called the "new" partition.

Each chunk has a version number that is unique and monotonically increasing, typically a sequence number. The version number is used for cache invalidation, content debugging, and data rollback.

5.2 Atomic Updates

Once we have a new chunk, we need to install it atomically. Conceptually, this is done by updating a version vector [Cha+81], with one element for each chunk.

In the absence of caching this is trivial: it is sufficient to close and reopen the corresponding files. It is slightly better to open the new files first, which allows the existing queries to finish on the old version, while new queries go to the new version. When the last pre-update query completes, the old version files can be closed (and later deleted).

With caching, we must also invalidate the cache entries for the old version. The simplest implementation of caching uses a separate cache for each chunk, in which case we can just invalidate the whole cache for that chunk. This works pretty well; other chunks keep their caches intact and the overall performance impact is thus limited. Alternatively, caches can be unified for all of the chunks on a node, which improves performance, but chunk replacement invalidates the whole cache.

The replacement of a specific chunk does not require the node to be stopped. Rather by using UNIX signals, we can use a management process to install chunks remotely. We also use signals to initiate a rollback to the previous version of a chunk. With some automation, the management process can update all of the chunks in a smooth rolling upgrade, and likewise update all of the nodes. Updating chunks incrementally limits the impact of rebuilding the cache, since most of the cache remains intact; this makes it feasible to keep up with the ongoing load during an update.

5.3 Real-time Deletion and Updates

So far, we have said that we do not do updates to individual records. This is not strictly true, but is the right overall view, since the mechanism described here is relatively heavyweight. There are some occasions where it is useful to update a specific document immediately. For example, a document known to be illegal may need to be removed immediately upon discovery. For this purpose, we add a mechanism for real-time deletion, which also enables real-time updates.

The general approach to deletion is to *add* a row that means "item deleted" that we can then use as the right-hand side of an antijoin to cull the document from a set. For real-time deletion, we add a very small table (usually empty) to every chunk, which contains the list of deleted documents. It is a property table, where the property it represents is "has been deleted", and we apply it as a filter to every query. Since we add this filter before optimization, it will be optimized as well. In the normal case the top-most operator is already a `FILTER`, and the optimizer can just add the inverse of this table as an extra property. Thus to delete a document in real

time, we simply add a row to this special table, and then atomically update the whole table (as with regular updates).

Given this mechanism, we can also do real-time updates. An update involves inserting the new version into a different chunk (usually in the "new" partition), and deleting the old version. Just doing the insert is not sufficient, since the master will see both versions, and may return both or the even just the old one (if it thinks they are duplicates).

5.4 System-Wide Updates

Occasionally, we perform updates that affect all of the nodes. The most common example is a change to the scoring algorithm, which makes the old scores incomparable with the new scores. Similarly, we may change the schema or the global ID mechanism. In such cases, we need to ensure that masters only use compatible followers.

The approaches to this are covered better elsewhere [Bre01], but the easiest solution is to update all of the nodes at once. By staging the updated versions ahead of time (i.e. loading them onto the disks in the background before the update), and using some automation, it is possible to update all of the nodes at once with less than a minute of downtime. The cold caches will perform poorly until they warm up, but since this kind of update is only done when the load is low, this is not a problem in practice.

6 Fault Tolerance

The primary goal of fault tolerance for search engines is high availability. We use a variety of techniques and optimizations to achieve this, few of which are novel, but together form a consistent strategy for availability.

The first task is to decide exactly what needs to be highly available, since there is always a significant cost to provide it. First, the snapshot approach means that all of the indexing and crawling process is independent to the server and thus need not be highly available. The only fault tolerance requirement for these elements is idempotency, to ensure that we can simply restart failed processes.

In addition, most documents are not worth replicating for high availability. In fact, most documents will never appear in a search result at all, but alas we cannot reliably predict which these are (or we would keep zero copies). Thus some partitions are replicated and some are not, and faults in non-replicated chunks or nodes simply reduce the database size temporarily. However, the use of pseudo-random `DocIDs` means that we lose a random subset of the documents in a partition, rather than, say, all the documents from one site. A typical policy might replicate popular sites and paid content.

6.1 Disk Faults

The most common fault is a disk failure, either of a block or a whole disk. A block fault only affects one chunk, but a disk failure might affect more than one. In both cases, new copies of the chunks can be loaded onto other blocks or disks in the background, and then atomically switched in. Note that chunks are never updated in place even in normal operation, so the replacement chunk is really just an atomic update to the same version. Nodes are limited by disk seeks, not space, so there is always plenty of free space for staging. In fact, given that space is cheap and staging areas are useful, it is worthwhile to cluster the active chunks onto contiguous tracks, which reduces the seek time during normal operation; other parts of the disk are used for staging.

Failed disks are left in active nodes until some convenient time, typically the scheduled maintenance window for that node. We replace whole nodes only, and then sort out the failed disks offline. This simplifies the repair process, as we always have spare nodes ready to swap in, which are then loaded with the proper chunks and put back online. Originally, we used RAID to hide disk faults, as most DBMSs do, but found this to be expensive and unnecessary, and those disks still needed some process for replacement.

For replicated chunks, if this node is the secondary, nothing special happens during recovery. If it is the primary, than the other replica becomes the primary and handles the queries until the local copy is restored. For caching purposes, it is best to have only one replica handle queries in the normal case (the primary), with the other replica idle. For load balancing, each member of a replica group will be the primary for some chunks and the secondary for others. The are lots of ways to determine which node should be the primary by default, but any simple (uniform) function of the chunk ID suffices.

6.2 Follower Faults

For node failures, we separate the case of followers from that of masters. A failed follower takes down all of its chunks. A master will detect this failure, if it doesn't already know, via a timeout. It will then either continue without the data in the unreplicated case, or contact the secondary in the replicated case. An important optimization is to spread the secondary copies across the partition, so that we spread out the redirected load that occurs during a fault [Bre01]. This can be done by "chained declustering" [HD90], but there are many suitable placements. For example, a typical partition might have ten nodes, 2-way replication, and nine primary chunks per node. Ideally, the nine secondaries that match the nine primaries for a given node, should be on nine different nodes, so that after a failure we have evenly spread out the load for the secondaries. Thus a replicated partition should have more nodes than the degree of replication, and a enough chunks per node to enable fine-grain load balancing after a failure.

Failed nodes are typically replaced later the same day, but they can be replaced at any time. The risk is that the secondary might fail before then.

6.3 Master Faults

Since masters are interchangeable, the basic strategy is to reissue the query on a different master. Originally, the master was also the web server, which meant that its failure was externally visible. A layer-7 switch [Fou01] can hide failed nodes for new queries, but it typically cannot reissue the outstanding queries at the time of the failure. For that, we depended on the end-user to hit reload, which they are remarkably happy to do.

The current approach separates the web server from the master, and the web server detects the failure and reissues failed queries to a new master (much like the relationship between masters and followers). This "smart client" approach [C+97] is strictly better for two reasons. First, the retry is transparent to the end user, much like a transactional queue [BHM90]. Second, it allows us to reissue the query to a different data center, which facilitates global load balancing and disaster recovery (covered below). The web servers are often owned by partners and are thus located in other data centers anyway. They use a client-side library within the web server to execute search queries, and the recovery and redirection code is part of this library.

6.4 Graceful Degradation

An important challenge for Internet servers that is not typically present for DBMSs is that of overload. There are many documented cases of huge load spikes due to human-scale events such as earthquakes or marketing successes [Mov99,WS00]. These spikes are too large for overprovisioning, which means we must assume that we will be overloaded and must degrade the quality of service gracefully. Overload detection is based on queue lengths: when queues become too long, the system enters overload mode until they drop below some low-water mark.

The details are beyond the scope of this paper, but there are two basic strategies that we use for graceful degradation (see [Bre01]). The first and simplest is to make the database smaller dynamically, which we can do by leaving out some chunks. This both reduces processing time per query and increases the effective cache size for the remaining data. Each chunk we take out increases our effective capacity by some amount, and we can continue this process until we are no longer saturated.

Second, we can decline to execute some queries based on their cost, which is a form of admission control. The naive policy simply denies expensive queries, such as those with many search terms. A more sophisticated version denies queries probabilistically, so that repeated queries will eventually get through, even if they are expensive.

6.5 Disaster Recovery

Disaster recovery is the process of recovering a whole data center, which might take considerable time, but should be very rare. So far we have not had any disasters, although we have moved data centers on multiple occasions, while keeping the system up, and thus know that our approach works.

The basic strategy is to combine master redirection and graceful degradation. When a data center fails or becomes unreachable, the client-side library in the web server will detect that the master has failed and will retry another master, probably in the same data center. At some point it will give up on that data center and try an alternate. The number of data centers varies, but the range is 2-10. Important partitions must be replicated at multiple data centers, in addition to local replication.

Although redirection is sufficient for a single query, it would not work in aggregate without automatic graceful degradation. If we simply redirect all queries from one data center to another, the new target will likely be overloaded. (At low load times, it would probably be fine.) Thus, we depend on graceful degradation to increase the capacity of the new data center to handle the load of both centers. Unlike a traditional load spike, which is relatively short lived, this state may persist for a while. Although it is possible to add some real capacity on short notice, full capacity may require major repairs or even the provisioning and setup of new space.

7 Other Topics

In this section we briefly visit a range a search engine challenges that differ from traditional database systems.

7.1 Personalization

Although personalization has become an important part of the web experience, e.g. "My Yahoo!", there is no equivalent in other media and thus search engines were the first systems to run into the problems of large-scale personalization. The first such site was the HotBot search engine, which (originally) allowed users to customize the search interface.

There are two general approaches: cookies and databases. In the cookie approach, user data is stored in a "cookie" and parsed as part of each visit, while the database approach stores only the user ID in the cookie, which it then uses to retrieve the appropriate row from a table. Although the cookie approach appears simpler, it suffers from two serious problems: the data is distributed and generally unreachable, which hinders analysis, and it is difficult to evolve the schema.

Essentially the cookie approach requires that all current and previous schema *overlap in time*, since there is no way to update the schema for a user until they next visit. For example, if the schema has gone through six versions, the current system must be able to handle cookies that use all six schemas, since which version a user follows depends only on the time of *their* last visit (from a given browser), which can be any time in the past. Given the large population of users, every schema will have some number of representatives. This can be

addressed with version numbers (stored in the cookie), but remains awkward.

Although we used a DBMS to manage user data, it is actually a mediocre approach, primarily due to cost, complexity and availability. Indeed, there has been substantial work on how to solve this problem more directly, including some support in Enterprise Java Beans (backed by a database), the use a highly available cluster hash table [GBH+00,Gri00], and a new framework specifically for session-state management [LKF04]. Like the search engine itself, this component requires only a single query plan, in this case just a highly available hash table lookup (no joins, ranges, or projections).

7.2 Logging

Search engines, like other large-scale Internet sites, create enormous logs, often over 100GB per day. These logs are used primarily for billing advertisers, but also for improving the quality of the search engine, and debugging. (These are not the kind of logs used for durability in a DBMS.) Log management systems have become their own class of data-intensive systems, and they also do not fit well on top of existing databases. Although this material is covered much better by Adam Sah [Sah02], who worked on the original Inktomi log manager, it is worth some discussion here.

The two primary issues are 1) DBMSs traditionally do not handle large-scale real-time loading of data, and 2) the query language really needs to support regular expressions, relative timestamps, and partial string matches, none of which fit well within SQL. In addition, log records have a different and far simpler update model: logs are append only and log records are (generally) immutable. The concurrency control and fault tolerance decisions are thus quite different from a DBMS.

However, database principles and the top-down approach still apply, and in fact are the right approach. The log system has its own query language and its own optimizations, including compression, caching (of reverse DNS lookups), and parallelization.

7.3 Query Rewriting

As in DBMSs, query rewriting is a powerful and useful tool [PHH92,SJGP90]. In our case, there are two primary values. First and most important, it provides the easiest way to customize a query for a given user or population. For example, for users known to speak a certain language (based on their ISP for example), a rewritten query might increase the ranking of documents in that language or even filter the results for only that language. Similarly, personalization can be used to customize queries for a given user based on collections (e.g. more emphasis on news), topic, complexity, geographical location, etc.

Second, query rewriting is a clean way to encode the *context* of the query. An important direction for search engines is to provide different results based on the context of the query. For example, a query issued from a page about semiconductors that contains the word "chip" probably refers to semiconductors rather than corn chips or the TV show *Chips*. Rewriting the query to include a few terms about the context (with low weight) is one easy way to disambiguate an otherwise ambiguous query.

7.4 Phrase Queries

So far, we have only covered the simplest kinds of queries, those based on words and properties. However, the relative positions of words within a document are of great value for improved ranking. For example, searching for "New York" really should give much higher scores to documents in which the two words are adjacent and in the correct order. There are two general approaches to this problem: tracking proximity and tracking exact word positions.

Proximity techniques boost the scores of documents that have the words "near" each other, but not necessarily adjacent. This is a long-standing technique in information retrieval [Sal89], and there are many approaches. One typical one is to break a document into "pages" of some size and use one bit per page to track which pages contain a given word. "Nearness" is then defined by how many pages contain both words (which is just a bitwise AND). This requires building the bitmaps for every word/document pair, and then matching bitmaps once you know that document contains multiple words from the query.

The second approach is "phrase searching" in which the engine actually tracks every position of every word in every document. Remarkably, current search engines actually do this! Phrase queries are significantly more complex, as you need to do what amounts to a nested merge join for every word in the query. For example, given the sorted lists of positions for the words "New" and "York", you join them using an "off by one" equijoin: output a tuple exactly if the position of "New" is one less than the position of "York". The multiway join for phrases is analogous. Overall, the best ranking occurs by mixing the results of regular scoring, proximity boosts, and phrases.

8 Discussion and Conclusions

Up to now, the focus has been covering the design of a search engine from the perspective of a database system. In this section, we argue that is the right approach for other top-down data-intensive systems, and that such systems should employ the principles of databases if not the artifacts. We cover a few other example systems, each of which is a poor fit for existing databases, and yet a good fit for the principles.

First, it is worth summarizing why Informix did 10x worse than the hand-built search engine in 1996. Informix was among the best choices for a search engine at the time, and we in fact used it for other parts of the system, particularly personalization. It had cluster support and seemed to do a reasonable job with caching; it was also viewed as the best "toolbox" database, which is

what we needed. The basic issue was *over generalization*, which presumably might limit modern DBMSs as well. Here is a partial list of the optimizations that account for the 10x difference: no locking, a single hand-optimized query plan, multiway joins, extensive compression, aggressive caching, careful data representation, hand-written access methods, single address space, and no security or access control (handled by the firewall). The representation of indexed text in mid-90's databases was typically 3x larger than the raw text; Inktomi and Alta Vista drove this number to well below one, which accounts a significant fraction of the overall performance gain, since this directly affects the number and size of I/O operations, and the hit rate of the cache. Finally, even if modern databases solved all of these problems, which they do not, the designers of the next big data-intensive system will surely find some mismatches, and will also have to apply the principles rather than the artifacts.

For the first example of such a system, we return to the logging system, discussed in Section 7.2. The best solution [Sah02] is a top-down design with data independence and a declarative language. Although based on Postgres, it is a large deviation from a traditional DBMS, as it includes Perl in the query language for string handling, strong support for loading data in real time, and changes for high availability. Predecessors, in fact, were not based on Postgres at all and used the file system for storage.

Another search-related example is the Google File System [GGL03], which is a distributed file system optimized for large files, constrained sharing, and atomic append operations. It is a top-down design driven by the need to handle more than one billion documents and millions of files; in particular, in handles all of the files used by the crawling and indexing systems. It has a relatively clean semantics for its important operations (concurrent append in particular), and support for high availability and replication. Although "navigational" rather than query based, it fits the top-down model proscribed here.

A more remote example is the Batch-Aware Distributed File System (BAD-FS) [BT+03]. This is a file system for large wide-area I/O intensive workloads such as cluster-based scientific applications. It is a top-down design with a simple declarative query language, which allows the scheduler to optimize communication, caching and replication by controlling both the placement and scheduling of jobs. Although not described this way, it has the usual phases: a parser, query planner and optimizer, and an execution engine. It also provides a variation of views. As with SQL, the declarative nature is critical for enabling optimizations. This project exhibits the proposed methodology in part because it has members from both the database and systems communities.

Although harder to show, many other systems fit this model of applying the principles without the artifacts. These include workflow systems, which have a query language and data independence, XML databases, and the emerging field of bioinformatics. All of these systems have top-down designs that do not map well on SQL and existing database semantics. The most common approach is to "make" them fit, however awkward that may be. A clean top-down design, as in the case of logging above, would lead to different implementation that is simpler, cleaner, and presumably more reliable and a better fit.

In the end, the hope is that projects on the "systems" side will benefit from top-down thinking, well-defined semantics, and declarative languages that leave room for optimization. Conversely, the hope on the "database" side would be for more modular and layered designs that are more flexible than current (monolithic) designs, and thus more useful for new kinds of systems. It is not clear that such layering is possible, but there is some evidence in the form of Berkeley DB and some of the novel uses of Postgres, such as the logging system.

Acknowledgments: We would like to thank Joe Hellerstein, Adam Sah, Mike Stonebraker, Remzi Arpaci-Dusseau, and many great Inktomi employees including Brian Totty, Paul Gauthier, Kevin Brown, Doug Cook, Eric Baldeschweiler, and Ken Lutz.

[Apa01] The Apache Web Server. http://www.apache.org.

[BEA01] *The BEA WebLogic Server Datasheet*. http://www.bea.com

[BHM90] P.A. Bernstein, M. Hsu and B. Mann. "Implementing Recoverable Requests Using Queues." *Proc.of ACM SIGMOD*. Atlantic City, NJ. 1990.

[Bre01] E. Brewer. "Lessons from Giant-Scale Services." *IEEE Internet Computing* 5(4): 46-55, April 2001. http://www.cs.berkeley.edu/~brewer/papers/GiantScale.pdf

[BT+03] J. Bent, D. Thain, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. "Explicit Control in the Batch-Aware Distributed File System." *Proc. of SOSP 2003*. October 2003.

[C+97] C. Yoshikawa *et al.*, "Using Smart Clients to Build Scalable Services," *Proc of the.Usenix Annual Technical Conference*. Berkeley, CA, Jan. 1997.

[CG99] S. Chaudhuri and L. Gravano. "Evaluating Top-k Selection Queries." Proc. VLDB Conference, 1999. http://citeseer.nj.nec.com/chaudhuri99evaluating.html

[CGK01] Z. Chen, J. Gehrke, and F. Korn. "Query optimization in compressed database systems." *Proc. ACM SIGMOD 2001*. http://citeseer.nj.nec.com/chen01query.html

[Cha+81] D. Chamberlin *et al.* "A history and evaluation of System R." *Communications of the ACM*, 24(10), pp. 632–646, October 1981.

[CK98] M. J. Carey and D. Kossmann. "Reducing the braking distance of an SQL query engine." In *Proceedings of the 24th VLDB Conference*, pp. 158–169, New York, NY, August 1998. http://citeseer.nj.nec.com/carey98reducing.html

[DGNP88] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. "A single-program-multiple-data computational mode1 for epex/fortran." *Parallel Computing*, 5(7), 1988.

[FB99] A. Fox and E. A. Brewer. “Harvest, Yield, and Scalable Tolerant Systems.” *Proc. of HotOS-VII*. March 1999.

[FGCB97] A. Fox, S. D. Gribble, Y. Chawathe and E. Brewer. “Scalable Network Services” *Proc. of the 16th SOSP*, St. Malo, France, October 1997.

[Fou01] Foundry Networks ServerIron Switch. http://www.foundrynet.com/

[GBH+00] S. Gribble, E. Brewer, J. M. Hellerstein, and D. Culler. “Scalable, Distributed Data Structures for Internet Service Construction.” *Proc. of OSDI 2000*, October 2000.

[GGL03] S. Ghemawat, H. Gobioff, and S.-T. Leung. “The Google File System.” *Proc of the SOSP 2003*. October 2003.

[GL02] S. Gilbert and N. Lynch. “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.” *Sigact News*, 33(2), June 2002.

[GR97] J. Gray and A. Reuter. *Transaction Processing*. Morgan-Kaufman, 1997

[Gra95] G. Graefe. “The Cascades framework for query optimization.” *Data Engineering Bulletin*, 18(3):19–29, September 1995.

[Gri00] S. Gribble. *A Design Framework and a Scalable Storage Platform to Simplify Internet Service Construction*. Ph.D. Dissertation, UC Berkeley, September 2000.

[GWv+01] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. “The Ninja Architecture for Robust Internet-Scale Systems and Services.” *Journal of Computer Networks*, March 2001.

[HD90] H. I. Hsiao and D. DeWitt. “Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines.” *Proc. of the 6th International Data Engineering Conference*. February 1990.

[Her91] Maurice Herlihy. *A Methodology for Implementing Highly Concurrent Data Objects*. Technical Report CRL 91/10. Digital Equipment Corporation, October 1991.

[LAC+96] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day and L. Shrira. “Safe and efficient sharing of persistent objects in Thor.” *Proc. of ACM SIGMOD*, pp. 318–329, 1996.

[LKF04] B. C. Ling, E. Kiciman, A. Fox. Session State: Beyond Soft State *Proceedings of Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.

[LML+96] R. Larson, J. McDonough, P. O’Leary, L. Kuntz and R. Moon. Cheshire II: Designing a Next-Generation Online Catalog. *Journal for the American Society for Information Science*. 47(7), pp. 555–567. July 1996.

[ML86] L. F. Mackert and G. M. Lohman. “R* optimizer validation and performance evaluation for local queries.” *Proceedings of SIGMOD 1986*, pp. 84–95, 1986.

[Mov99] MovieFone Corporation. “MovieFone Announces Preliminary Results From First Day of Star Wars Advance Ticket Sales.” Company Press Release, *Business Wire*, May 13, 1999.

[PAB+98] V. S. Pai, M. Aron, G. Banga, M, Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. “Locality-Aware Request Distribution in Cluster-based Network Servers.” *Proc. of ASPLOS 1998*. San Jose, CA, October 1998.

[PDZ99] V. S. Pai, P. Druschel, and W. Zwaenepoel. “Flash: An efficient and portable Web server.” *Proc. of the 1999 Annual USENIX Technical Conference*, June 1999.

[PHH92] H. Pirahesh, J. M. Hellerstein, and W. Hasan: “Extensible/Rule Based Query Rewrite Optimization in Starburst.” *Proc. of SIGMOD 1992*. pp. 39-48. June 1992.

[RN02] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 2002.

[Sah02] A. Sah. “A New Architecture for Managing Enterprise Log Data.” *Proc. of LISA 2002*. November 2002.

[Sal89] G. Salton. *Automatic Text Processing: The transformation, analysis, and retrieval of information by computer*. Addison-Wesley, 1989.

[SAS+96] J. Sidell, P. M. Aoki, A. Sah, C. Staelin, M. Stonebraker, and A. Yu. “Data Replication in Mariposa.” *Proc. of the 12th International Conference on Data Engineering*. February 1996.

[SBL99] Y. Saito, B. Bershad and H. Levy. “Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-based Mail Service.” *Proc. of the 17th SOSP*. October 1999.

[Sel+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and G. T. Price. “Access path selection in a relational database system.” *Proc. of SIGMOD 1979*. Boston, MA. pp. 22–34. June 1979.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. “On rules, procedure, caching and views in data base systems.” *Proc. of the 1990 ACM SIGMOD International Conference on Management of Data*. June 1990.

[WCB01] M. Welsh, D. Culler and E. Brewer. “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services.” *Proc. of the 18th SOSP*. October, 2001.

[WS00] L. A. Wald and S. Schwarz. “The 1999 Southern California Seismic Network Bulletin.” *Seismological Research Letters*, 71(4), July/August 2000.

[ZBCS99] X. Zhang, M. Barrientos, J. B. Chen, and M. Seltzer. “HACC: An Architecture for Cluster-Based Web Servers.” *Proc. of the 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.

Appendix: Scoring

In this section we present a simple but representative scoring algorithm. Most of the research for current search engines is on improving the scoring algorithms or adding new components to the scoring systems, such as popularity metrics or incoming link counts.

We define a query as a set of words and their corresponding weights (W_i):

$$\text{Query } Q \equiv \{[w_1, w_2, \ldots, w_k], W_i\} \tag{3}$$

The score of a document for query Q is the weighted sum of an overall score for the document and a score for each word in the query:

$$\begin{aligned} \text{Score}(Q, d) \equiv\ & c_1 \text{Quality}(d) \\ & + c_2 \sum_i W_i \text{Score}(w_i, d) \end{aligned} \tag{4}$$

The document quality term is independent of the query words and reflects things like length (shorter is better), popularity, incoming links, quality of the containing site, and external reviews.

The use of weighted sums for scoring is very common in information retrieval [Sal89] and this one is loosely based on Cheshire II [LML+96]. It has several advantages over more complex formulas: it is easy to compute, it can represent multiplication by using logarithms within components (commonly done), and the weights can be found using statistical regression (typically from human judgements on relevance). To simplify query execution, we define:[8]

$$\text{Score}(w_i, d) \equiv 0 \qquad \text{if } w_i \notin d \tag{5}$$

We don't actually require that $\sum W_i = 1$ and it useful to modify the weights individually at query time. Since we only care about the relative scoring within one query, there is no particular meaning to the sum of the weights. Nor do the words need to be unique; in fact, entering the same word twice usually gives it twice the weight.

The word score can be further broken down:

$$\text{Score}(w_i, d) \equiv c_3 \cdot f(w_i, d) + c_4 \cdot g(w_i) + c_5 \tag{6}$$

where f captures the relevance of the word in this document, and g captures the properties of the word in the overall corpus. For example, the specific version from Cheshire II is essentially [LML+96]:

$$\begin{aligned} \text{Score}(Q, d) \equiv\ & -0.0674 \sqrt{length(d)} \\ & + \frac{1}{M} e \sum_{i=1}^{M} \begin{pmatrix} 0.679 \cdot \log Freq(w_i, d) \\ +0.223 \cdot \log IDF(w_i) \end{pmatrix} \end{aligned}$$

The top term is Quality(d) and the bottom term is the weighted sum, with even weights, of equation (6), where $f \equiv \log Freq(w_i, d)$ is the log of the count of w_i in d, and $g \equiv \log IDF(w_i)$ is the log of the *inverse document frequency* of w_i, which is one divided by the fraction of documents in which this word appears.

The scoring for `AND` and `OR` is trivial: just sum up the scores for the matching words. For example, `(a AND b)` has the same score as `(a OR b)`, although the `AND` will usually return fewer documents.

8: Words that are in "anchor text" that point to the document are considered part of the document.