Multi-tier Architecture for Web Search Engines

Knut Magne Risvik
Overture Services AS
P.O.Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
knut.risvik@overture.com

Yngve Aasheim
Overture Services AS
P.O.Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
yngve.aasheim@overture.com

Mathias Lidal
Overture Services AS
P.O.Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
mathias.lidal@overture.com

Abstract

This paper describes a novel multi-tier architecture for a search engine. Based on observations from query log analysis as well as properties of a ranking formula, we derive a method to tier documents in a search engine. This allows for increased performance while keeping the order of the results returned, and hence relevance, almost "untouched". The architecture and method have been tested large scale on a carrier-class search engine with 1 billion documents. The architecture gives a huge increase in capacity, and is today in use for a major search engine.

1. Introduction

Search engines are becoming a very important application for navigating the web. Search engines face huge challenges trying to keep up with the web dynamics. The web is growing exponentially, and it is becoming more and more dynamic. Risvik and Michelsen[11] discusses the dynamics of the web, and the challenges it imposes on the large search engines.

The ability to consistently deliver excellent result relevance is probably the single most important factor for user satisfaction in a web size search system.

When we disregard the fixed costs of maintaining a huge index, the cost of search execution is close to linear to the size of the index searched; doubling the amount of information in the index will generally double the cost of query execution.

Search engines of this size are usually distributed applications, using replication and partitioning to scale to the desired number of documents. We intend to utilize properties of query distributions and ranking formula contributions to optimize which nodes in the mesh to use for each query.

2. Preliminaries

Most practical and commercially operated Internet search engines are based on a centralized architecture that relies on a set of key components, namely Crawler, Indexer, Searcher and Dispatcher. This architecture can be seen in systems including AltaVista [2], Google[3], and the FAST Search Engine [1], and is illustrated in Figure 1.

Definition 1 Crawler. *A crawler is a module aggregating documents from the World Wide Web in order to make them searchable. Several heuristics and algorithms exist for crawling, most of them based upon following links in hypertext documents.*

Definition 2 Indexer. *A module that takes a collection of documents or data and builds a searchable index from them. Common methods are inverted files, vector spaces, suffix structures and hybrids of these.*

Definition 3 Searcher. *The searcher is working on the output files from the indexer. The searcher accepts user queries from the dispatcher (defined below), executes the query over its part of the index, and returns sorted search results back to the dispatcher with document ID and the relevance score (defined below).*

Proceedings of the First Latin American Web Congress (LA-WEB 2003)
0-7695-2058-8/03 $17.00 © 2003 IEEE

Definition 4 Dispatcher. *The dispatcher receives the query from the user, compiles a list of searchers to execute the query, sends the query to the searchers and receives a sorted list of results back from each searcher. For each result it receives a unique document ID, and the relevance score. The hits from the searchers are then merged to produce the list of results with the highest relevance scores for presentation to the user.*

Search engines generally operate by producing a result set for each query (a set of documents that matches the query), and then ranks the documents by using a formula to compute a relevance score for each entry in the result set with respect to the query being executed and sorting the documents by their relevance score.

Definition 5 Relevance score *Upon a given query, q, the search engines give back a sorted list of results. Each document in the result has a relevance score $R_{d,q}$ (Relevance of document d with respect to query q). The relevance score can be further broken down into a static component Rs_d that is independent of the query, and a dynamic component $Rd_{d,q}$ dependent on the query. The total relevance score formula is then*

$$R_{d,q} = Rs_d + \alpha Rd_{d,q} \quad (1)$$

The constant α is used to balance the weight of the static and the dynamic relevance scores.

For a web search engine, utilizing link structure and HTML semantics, the relevance score is usually derived from several different contributing attributes:

- Static relevance score for the document (link cardinality, page quality).
- Superior parts of the document. (Titles, metadata, document headers)
- Authority (external references, and the "level" of these).
- Document statistics (by e.g. term frequency in the document, global term frequency and term distances within the document).

3. Related Work

From several studies of query logs ([12], [5]) and also supported by our own analysis presented in this paper, there is a highly skewed distribution of unique queries on a typical websearch engine. Both [12] and [5] reports that the top 25 queries count for more than 1% of the total query volume.

A vast amount of work has been conducted on cache algorithms for web search engines, and for information retrieval systems in general. In [6] we see a study of different cache-replacement schemes for web search engines, testing both standard LRU, segmented LRU and a predictive system based on session analysis named PDC (Probability Driven Cache).

The results are promising, one sees more than 50% hit ratio of the caches, clearly being a very important optimization tool for any serious web search engine.

On a different perspective, there has been several studies on how to partition a large dataset in an information retrieval system and to provide some sort of a broker algorithm to select a subset of these partitions for searching with the intent of providing an approximation of the search results from searching the entire collection.

In [7] and in particular [9] we see studies using locality analysis of the query log to compute a partial replica of the entire collection and to use a broker (to multiple InQuery servers) server to select the replica. In [8] we see the same technique applied on a large dataset.

4. This Work

In this work we propose an architecture for a scalable web search engine that uses multiple log and relevance analysis to build tiers of documents. The architecture is novel to prior work in the sense that each tier is disjoint from the others (as opposed to partial replication where the smaller sets are subsets of the main index). Furthermore we deploy a fallthrough algorithm to allow for queries to "fall" from one tier to another based on analysis of the query and the results from the given tier.

We run experiments on 3 different architectures inspired by results from partial replication and query cache analysis as well as the ranking function itself. The experiments provide evidence that these architectures offer a significant increase in query capacity over regular caching.

5. Target Architecture

The analysis described in this paper are all based on the same conceptual architecture. We will describe the architecture in two levels, first the cluster concept, then the tiering concept.

5.1. Basic Elements

The basic element in the target architecture is a search node. A search node holds a partition, an index for a fraction of the entire database, and allows for searches in that partition.

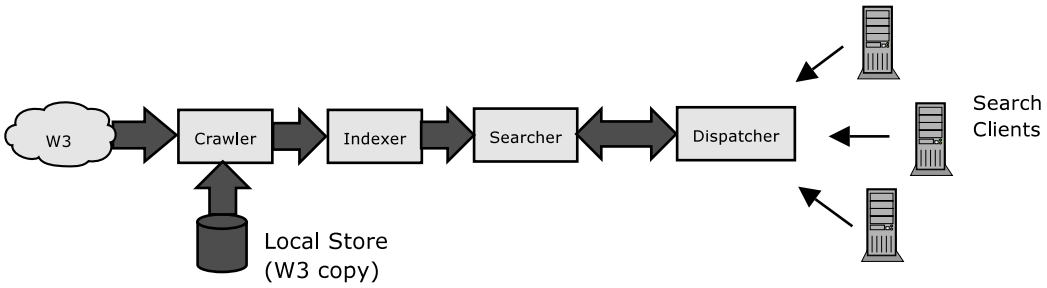


Figure 1. Search Engine reference model

Documents	QPS	disk KB	disk TPS	CPU idle
3M	900	25	2000	45%
4M	557	34	1720	41%
5M	434	40	1560	42%
6M	352	50	1350	43%
7M	365	58	1450	35%
8M	311	66	1315	37%
9M	292	73	1310	35%

Table 1. Data volume performance

5.2. Performance for different data volumes

In this paper we will use the FAST Search engine kernel. This engine has performance characteristics that, within certain boundaries, can be considered linear with the number of documents indexed per node. Some performance numbers are shown in Table 1, and illustrated in Figure 2, where QPS denotes the number of queries per second, disk KB denotes the number of kilobytes read per query, disk TPS indicates the number of disk transactions per second, and CPU idle is the percentage of CPU slices idle during testing.

All these tests were conducted on a dual CPU (2x2.4Ghz) Pentium 4 with 1GB of main memory and 12 x 36 GB disk drives. The benchmarking was done with 50 parallel clients, and maintaining average response time of a query below 0.5 seconds, the standard customer SLA (service level agreement).

Letting I denote the entire database, a search node S_j typically holds I_j. Now, letting $QPS(S_j)$ denote the query capacity of search node S_j in queries per second and $|I_j|$ denote the size of partition I_j, and furthermore assuming that the average length of queries are constant, the following relation is given:

$$QPS(S_j) \propto 1/(|I_j|) \quad (2)$$

That is, the query capacity (QPS) is inversely proportional to the size of the partition, as showed in Figure 2 (a). This means that, given a fixed index-size the total query capacity is proportional to the number of search-nodes. The FAST Search kernel employs several types of performance optimizations that is required for this approximation to be applicable.

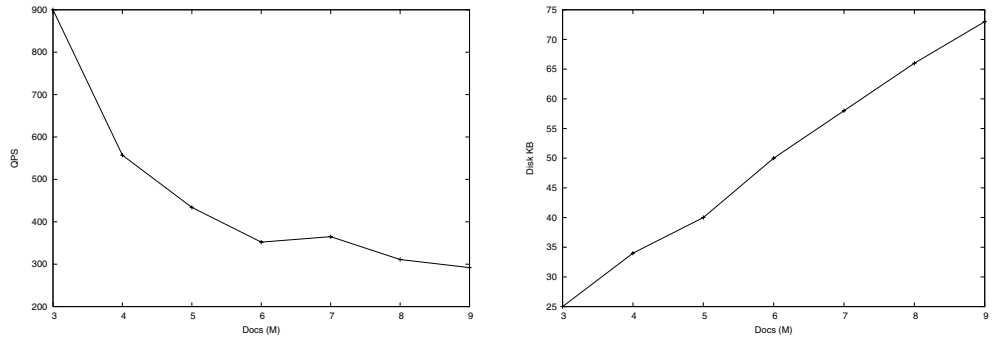
5.3. Basic Scalability

Now, let a cluster be a collection of search nodes that are grouped in rows and columns. Partitioning and replication are then used to create linear scalability in size and query rate. The basic principles of this architecture is also covered by [10].

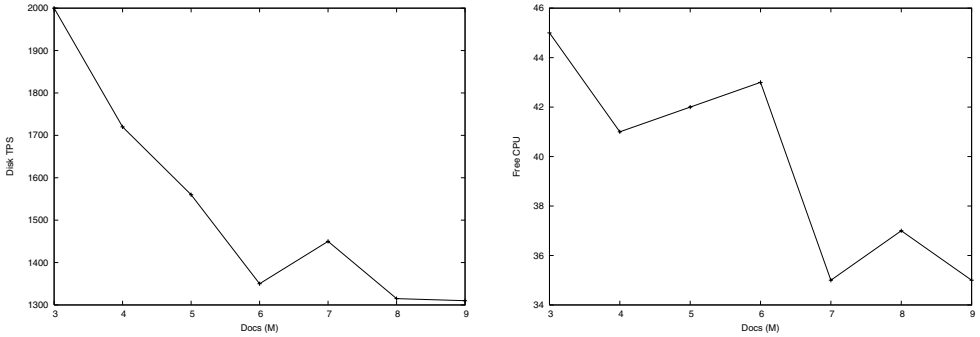
5.3.1. Data volume scaling by partitioning Let each search node hold a partition of the index, I_j of the entire index I. Each search node queries its partition of the index upon a request from a dispatcher. The dispatcher sends an incoming query to a set of search nodes such that all partitions $I_1 \ldots I_n$ are asked. The results are merged and a complete result from the cluster is generated. We use the notion of a *row* to name a set of search nodes making up for all partitions of the entire index. Thus, by partitioning the data, we create scalability in data volume, as illustrated in Figure 3. Given Equation 2, and assuming that we split the data into n partitions, the search time on each node would be $O(|I|/n)$. n is selected based upon hardware and software criteria, but varies between 1M and 20M documents.

A limiting factor here is the merging of results that needs to take place in the dispatcher. A heap-based algorithm typically gives $O(mlogn)$ to merge m results from n sources. m is typically low due to the interactive nature of a search system.

5.3.2. Performance scaling by replication By replicating each of the search nodes, we are able to increase the query processing rate for a given partition of the index. Letting S_i^j denote a search node in an nxm cluster, all search nodes $S_i^1 \ldots S_i^m$ holds index partition I_i. Thus, the dispatcher can rotate



(a) QPS (b) KB



(c) TPS (d) % Free CPU-time

Figure 2. Performance metrics for different document sizes

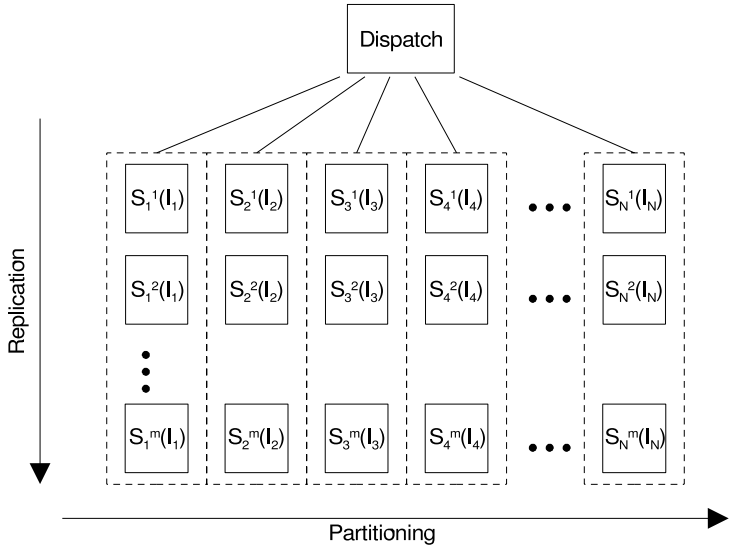


Figure 3. Cluster overview

between m nodes for each index partition when selecting a set of search nodes to handle an incoming query.

The dispatcher handles this by a round-robin fashion algorithm. By adding rows, performance will increase proportionally to the the capacity of a single row.

The entire architecture in Figure 3 illustrates the organization into rows and columns, and the role of the dispatcher.

5.4. Tiered Architecture

Based on the search node clusters described above, we now intend to build a tiered architecture of search nodes.

Conceptually, the tiered architecture groups documents into tiers $T_1 \ldots T_m$. Each query starts off in tier 1, and a fallthrough algorithm FTA selects whether the query shall continue execution into consecutive tiers and how results are to be merged.

Thus, there are three elements to the tiered architecture:

Definition 6 Tier mapping. *Given a set of documents D, the function $\mathcal{F}$ takes a document D_i with a vector of properties, $\vec{P_{D_i}}$ and maps it into a given tier T_j.*

Furthermore, the algorithm to define the fallthrough needs to be defined:

Definition 7 Fallthrough - *FTA. Each query starts off in the lowest tier. The fallthrough algorithm, FTA, determines the path of the query in the set of tiers based on criteria such as relevance scores and number of hits in the result set. The FTA is responsible for determining how many results from each tier can be used before the next tier must be consulted.*

When the FTA decides that a new tier must be consulted, the relevance score for the new results will be considered when merging results to produce the final list of results to be presented to the user.

In Figure 4 an example is illustrated. Given that the documents can be viewed in a two dimensional feature space, we can illustrate the document space as in the first figure. Then tiering takes place by applying $\mathcal{F}$ to map each document to the desired tier. Then mapping can be done into search nodes as shown.

Figure 4 also illustrates that documents can be divided into sections for tiers. For instance, one could define tier 1 to be the superior context (titles and anchors) of all documents, and tier 2 to be the body context of a small selection of the documents. Thus, a search node will hold data for multiple tiers, but without duplicating the data stored.

The tiered architecture has an impact on performance when the FTA is able to reduce the number of search nodes involved in a query, and the appearing results have little or no deviation from what you would get by searching all nodes. Of course, this implies an interactive system, where the top 10–100 results are the interesting ones.

6. Query locality

We aim to understand the locality of queries and to estimate the confidence of these locality figures. By locality we mean to what degree the same queries are received multiple times, both over short and long time-periods. Provided a significant number of queries are duplicates, we can build a tiering mechanism that would have a significant positive impact on performance of the entire search system. The queries we examine are gathered from the query log of alltheweb.com. The query log is a log showing the query-string, along with the date and time of the query, and some more information which we do not use at this time.

6.1. Query Distribution

First we study the query distribution. By folding the queries into unique queries, and sorting the bins based on query frequency, we get a sorted distribution of queries.

The queries we examine are collected from the queries executed by users of alltheweb.com. We analyze five different query-sets, collected from the following time-periods:

- 3. of January 2002
- 15. - 21. (3. week) of January 2002
- 1-31. of January 2002
- 11. of February 2002
- 11. of March 2002

These query-sets enable us to analyze the queries from both overlapping and disjoint time-periods, which should give a good indication of the level of locality of the queries. These querysets are summarized in Table 2, showing the total number of queries, the number of unique queries and the number of queries made up of the 1 million most popular unique queries.

We now want to analyze how the most popular queries do in the overall frequency picture. Here we

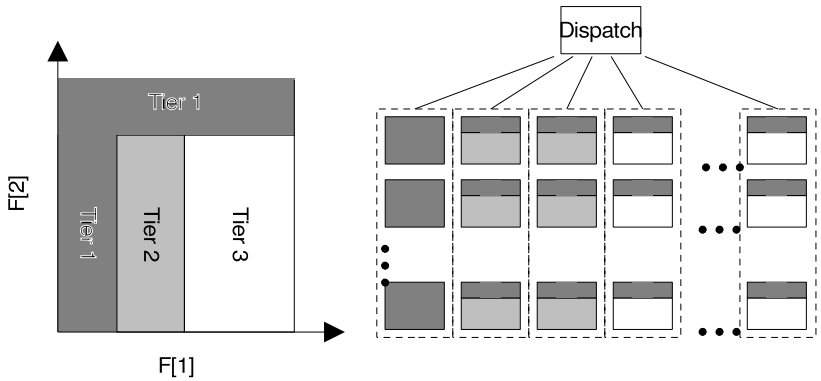


Figure 4. Mapping from a document space into tiers of search nodes

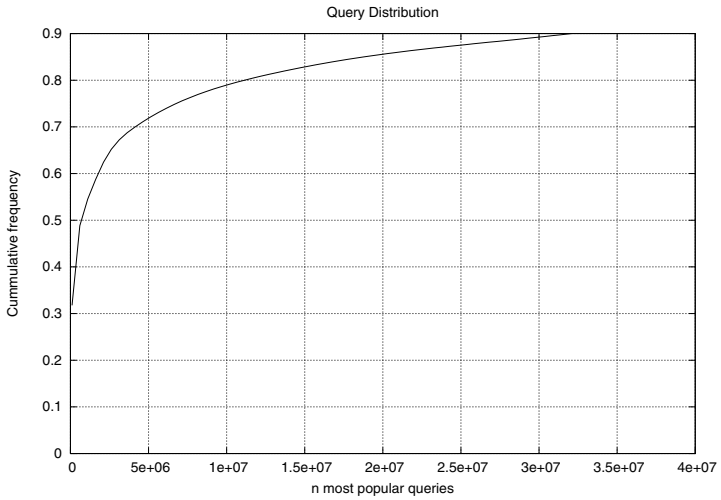


Figure 5. Query frequency distribution

used the largest query-set, from January 2002. We collected the cumulativ efrequency of the n most popular queries. The distribution of this frequency is shown in Figure 5.

We observe that the query distribution is strongly logarithmic, thus we have an exponential gro wth in the number of queries to hold a given percentage of the total queries.

This distribution is fairly constant over time, as sho wn in Figure 6 which sho ws the frequency of the 1 million most popular queries per week for the first half of 2002.

F urthermore, we count the number of unique queries (i.e. queries that are only ask ed once), and w e find that 41,748,880 queries out of the 573,394,981 queries recorded in January only occur once. This is 7.28% of the queries.

6.2. Temporal o v erlap

Here we examine the overlap betw een queries from different time-periods, focusing on the 1 million most popular queries.

Letting Q_1 and Q_2 denote the set of queries for each consecutive period, the overlap is computed as:

$$O(Q_1, Q_2) = \frac{|(Q_1 \bigcap Q_2)|}{|(Q_1 \bigcup Q_2)|)} \quad (3)$$

Now, letting O_n denote the overlap of the n most frequent queries from Q_1 and Q_2, we compute the overlap for different number of unique queries.

First we examine the overlap betw een the query-sets from January. This will give an indication of how representative the results of a single day or week is. Next we examine the overlap betw een the three different days, as well as the overlap betw een the queries from Jan-

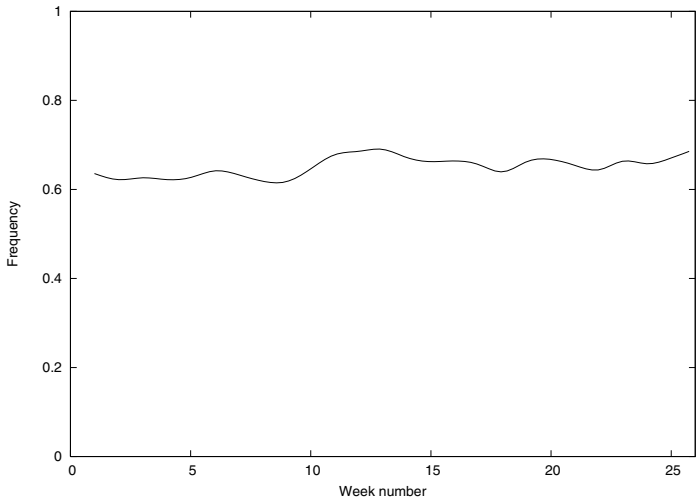


Figure 6. Frequency of top 1 mill. queries first half of 2002

Time period	Num. queries	unique queries	num. queries within top 1 million
3. Jan 2002	19 846 936	4 560 172	15 239 298 (76.8%)
15.21. Jan 2002	132 741 183	22 590 276	83 108 836 (62.6%)
Jan. 2002	573 394 981	81 721 575	307 434 094 (53.6%)
11. Feb 2002	20 062 879	4 300 090	15 678 972 (78.14%)
11. Mar 2002	20 714 809	4 311 001	16 518 754 (79.74%)

Table 2. Summary of querysets

uary and those from one day in February and March. This will give an indication of how much the queries differ over time. Table3 shows the overlap between the different query-sets.

6.3. Locality

These results show that there is a high degree of repetition in the queries, with only 7.28% of the queries being asked just once. There is also a very high temporal overlap, with 80 - 85% overlap on single days in different months, when looking at the top 1 million queries. These queries account for 75-79% of all queries these days.

7. Applications of the tiered architecture

We have evaluated three different architectures and document distribution functions, $\mathcal{F}$, and different fallthrough algorithms. For the different architectures, we compute the quality of the architecture as two numbers based upon the difference between its search results compared with the search results from a predefined reference system, typically a single-tier system.

The Fallthrough algorithm is basically equal for all three architectures, and consist of 5 parameters:

- *Hitlimit* - Specifies the maximum number of hits to be used from a tier before fallthrough to the next tier is forced.
- *Per centlimit* - The maximum percentage of the hits from this tier that may be used before fallthrough to the next tier is forced.
- *R anklimit* - Used together with the *T ermranklimit*. See *Termranklimit*.
- *Termranklimit* - If the relevance score of the hit being considered is less than the *R anklimit* plus this value times the number of terms in the query, then fallthrough to the next tier is forced.
- *MinUsableHits* - The number of hits that must pass the abovecriterias for a given tier in order not to do an immediate fallthrough to the next tier. This number is typically the number of results that is presented to the user on a result page. The rationale for using this rule is that if it is known

	1 day (Jan)	1 day (Feb)	1 day (Mar)	1 week	1 month
1 day (jan)		82.04%	80.70%	85.06%	84.53%
1 month	86.65%	84.72%	84.53	92.54%	

Table 3. Temporal overlap

that we will have to fall through in order to produce the number of hits most often requested, we should perform the fallthrough immediately. We do not want to apply this rule on a non-constant value (such as the number of hits really requested by the user), in order to ensure that we produce consistent results.

```
boolean FallThrough(Result sr,
                    FallthroughConfig c,
                    int hitNo,
    int queryTerms) {

    // Verify HitLimit rule
    if (c.hitLimit < hitNo)
        return true;

    // Verify PercentLimit wrt. requested hit
    // and MinUsableHits parameter
    if (max(hitNo, c.minUsableHits) >
       ((r.numResults * c.PercentLimit) / 100))
        return true;

    // Verify RankLimit and TermRankLimit wrt.
    // requested hit and MinUsableHits parameter
    if (r.Result[max(hitNo,c.minUsableHits)].
       relevanceScore <
       (c.rankLimit + c.termRankLimit *
        queryTerms))
        return true;

    return false;
}
```

The parameter values for the basic FTA is defined in Table 4.

For the multi-tier concept to work, we need to have higher performance on tier 1, which is going to execute all queries, than on tier 2 which will only execute those queries that fall over from tier 1. Tier 3 will, in turn, only execute queries that fall over from tier 2. The extra capacity on the lower tiers may be achieved either by replicating the columns on this tier, or by reducing the number of documents on each node, in order to achieve higher throughput on each node. From an operations point of view, the latest option is more flexible, hence this is the preferred approach.

As already stated in this paper, the performance of a node is considered to be linearly dependent on the number of documents on the node. For the tests runs being described in this paper, we had 1.5M documents indexed on each tier 1 node, 6M documents on each tier 2 node and 10M documents index on each tier 3 node.

7.1. Evaluation

To evaluate the different multi-tier configurations, we randomly select 100000 unique queries from the list of queries entered by users of FAST Web Search. The selected queries are then executed twice, once towards the multi-tier configuration we want to evaluate and once towards the reference system. The results from the two runs are compared to extract information about the quality of the multi-tier configuration.

We use two metrics to indicate the quality of search results in comparison with the reference system:

- **Different First hit**. The percentage of queries that return a different hit in the first position.
- **Different Top 10**. The percentage of queries for which there are one or more differences within the first ten results.

With an ideal solution, those percentages are zero or close to zero.

Combined, those two numbers are believed to give good knowledge about key attributes of the systems relevance penalty with respect to the reference system.

Furthermore, we measure the total system query capacity to understand the performance impact of the multi-tier architecture. This is done by executing a standard query log (typically 100K to 1M queries) onto the search cluster and measure the query production rate at the standard SLA. Then the speedup ratio between the reference system and the candidate system is computed.

The experiments are conducted on a sample collection containing 1 billion documents. The number of documents to be index on each tier have been predefined, and are the same for all three configurations to be tested.

Tier jump	Hitlimit	P ercen tlimit	Ranklimit	T ermranklimit	Minhits
1–2	1000	10	200	0	0
2–3	8100	30	0	0	100

Table 4. Basic FTA

Tier 1	30M documents
Tier 2	360M documents
Tier 3	610M documents

T able 5. 1D Multi-tier config

Tier 1	30M documents (5M tierlocked)
Tier 2	360M documents
Tier 3	610M documents

T able 6. 1.5D Multi-tier config

7.2. 1-dimensional multi-tier configuration

This is a plain three-level multi-tier configuration, with all documents being distributed by the static relevance score. The 30M top documents are then mapped to tier 1, the 360M next documents to tier 2, etc. The breakdown of documents into tiers is shown in Table 5.

The system is illustrated in Figure 7, using the standard FTA as defined in Table 4.

The reference system chosen is a plain single-tier configuration, thus searc hing all nodes simultaneously.

One significant problem with this configuration is that static relevance is only a part of the formula for determining the relev ance score of a hit. This means that a document with low static relev ance score can still be a good hit for certain queries. The next configuration is an attempt to reduce this problem.

7.3. 1.5-dimensional multi-tier configuration

Again, this is a plain three-level multi-tier configuration. Before doing tier distribution, w e runa query log with the 1M most common queries for a period in time. To avoid delaying the indexing process in the production system, this query log was executed tow ards the previous generation index executed on the previous index. The first 20 documents returned for those queries will be indexed on the first tier.

The remaining documents are distributed according to the static relevance score, so the tier breakdown is as shown in Table 6. The system is illustrated in Figure 8, and was tested using the FTA as defined in Table 4.

Tier 0	1B documents - superior context
Tier 1	30M documents - body context
Tier 2	360M documents - body context
Tier 3	610M documents - body context

T able 7. 2D Multi-tier config

Again, the reference system is a plain single-tier configuration.

7.4. 2-dimensional multi-tier configuration

The tier distribution for this configuration is identical to the tier distribution for the 1.5-dimensional multi-tier configuration. The difference is the way the information is searched. With this configuration, w e search the information in the high-value con textsfor all documents first. If we need more hits, we con tinue to search the full index, using a multi-tier configuration, removing duplicates from the returned results.

Obviously, searching the high-value contexts for all documents will consume its fair share of resources available on the second and third tiers. This mean that we hav e capacity for processing fewer queries on those tiers when searching the full index on those nodes.

The system has tier distribution as sho wn in T able 7, and it is illustrated in Figure 9. As shown, Tier 0 is introduced as the lay er of superior contexts.

The FTA for this system is sligh tly modifiedfrom the original one (Table 4). The modified FTA is sho wn in Table 8.

The reference system defined for the other multi-tier configurations can not be used directly for this configuration, since this multi-tier configuration not directly emulates a plain single-tier solution. We still want to use a single-tier system, but w e no w need to searc h high-value contexts for all nodes first, then merge in results from a search in the full-text index.

This change in the reference system naturally changes the order of the results produced b y the search engine. How ever, manual verification b y editors as well as theoretical studies indicate that, given the w ay w e currently compute the relev ance score,

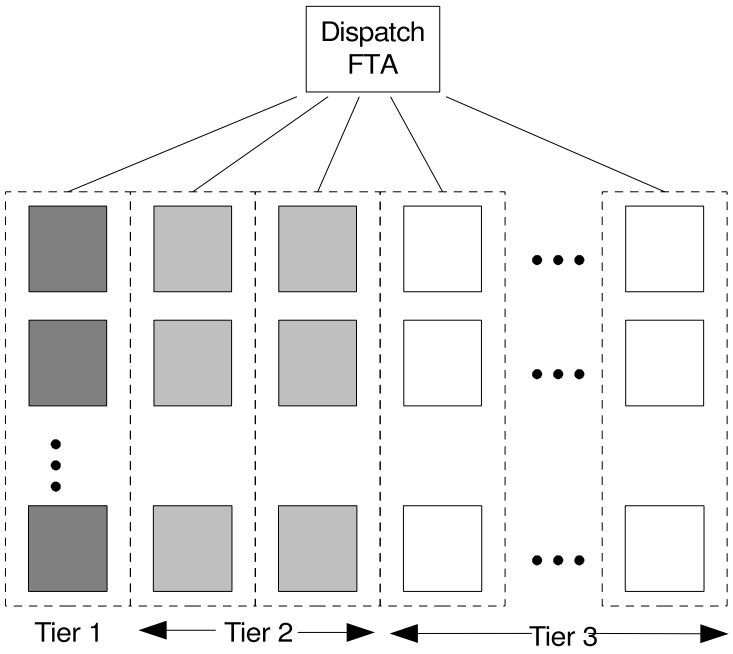


Figure 7. 1-dimensional MT system

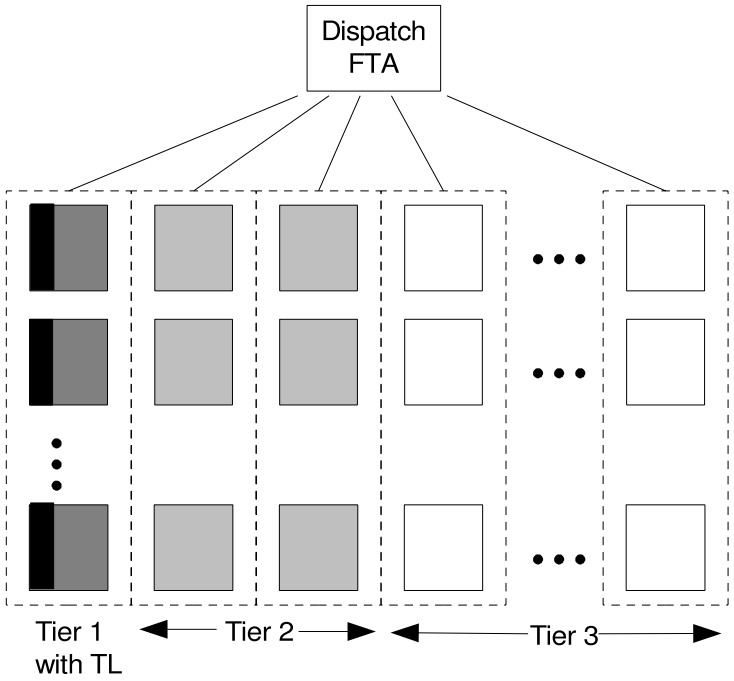


Figure 8. 1.5-dimensional MT system with tierlocking

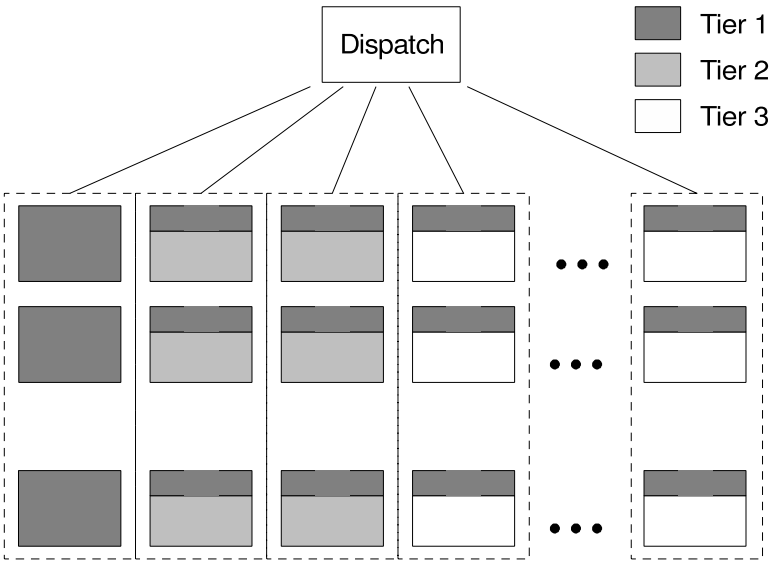


Figure 9. 2-dimensional MT system

Tier jump	Hitlimit	Percentlimit	Ranklimit	Termranklimit	Minhits
(0+1)–2	8100	50	200	0	100
2–3	8100	70	200	0	100
3–4	8100	50	0	0	100

Table 8. 2D FTA (Note that 0 and 1 are searched in parallel.)

this change more often improve the relevance than reduce it.

7.5. Results

A summary of the experiments is shown in Table 9. The 1D Multi-tier solution heavily relies upon high covariance between the rank of a search result and the static relevance score of the documents in the search result. This assumption is often wrong for web search queries are for most cases wrong, thus we will see big deviations from the reference system. This is mainly due to the nature of static relevance score, which is mostly based upon link cardinality. In [4] the link structure of the web is discussed.

Moving on to the 1.5D architecture, we utilize the locality within query logs to bias the creation of the first tier. This model will guarantee that the most popular queries (which accounts for a very high total number of queries) always gets the "correct" results compared with the reference system. This is clearly indicated on the results. However, the queries falling out of the "popular" group are still subjects for the same treatment" as with the 1D system.

The key observation from the 2D system is that the usage of more document properties, and another dimension in the tier space has a much higher covariance with the actual ranking function. So by letting the relevance score space be divided into static relevance score and the binary selection of superior and non-superior context, the approach has a relevance that is very appealing given the performance capabilities. Also, the use of a reference system that has proved superior to the other reference systems by editorial inspection is promoting this as a very promising architecture. The end result is more than a doubling of the performance without much sacrifice in relevance

The performance of the different configurations, as stated in the QPS ratio column of Table 9 have been measured by benchmarking the different systems to extract information about the highest possible throughput that can be achieved while complying with the standard SLA.

System	Different #1	Different #10	QPS ratio
1D Multi-tier	26.8%	62.1%	2.78
1.5D Multi-tier	15.4%	45.8%	2.71
2D Multi-tier	4.36%	17.7%	2.02

Table 9. Summary of results

8. Capsule summary of the FAST Search Engine

The FAST Search Enginewas originally developed for the AllTheWeb(http://www.alltheweb.com) search destination site. The engine is also used on a wide variety of enterprise search solutions. Performance was the key driver for how the search engine was designed, and it is utilizing a unique mix of inverted postings and suffix structures for achieving the performance and scalability discussed in this paper.

Fast Search & Transfer (FAST) (http://www.fast.no/) is a Norwegian company that has been operating in the web search and corporate search segments for six years. After this paper was drafted, the web search product was sold to the American company Overture. Overture is currently in the process of being acquired by Yahoo.

9. Conclusive Remarks and Future Work

We have shown how multi-tier architecture can be used to achieve "super-linear" scaling of web size search engines. Sample systems are able to achieve more than double the capacity with a relatively small relevance penalty.

The testing has been restricted to a handful of configurations, but the results are still very promising. Moving along, it will make sense to do much more extensive analysis to determine the dimensions of the document space, and possibly one could use mining techniques to derivethe document to tier mapping rules and fallthrough algorithms.

In order to efficiently design and tune a multi-tier search architecture, we aim to build a simulation tool that can simulate different fallthrough algorithms and different tier separation rules. Evaluationof possible performance benefits and relevance penalties with different tier sizes is equally interesting.

For a configuration beingused for production purposes, one would want a system that continuously monitor the number of queries that get a relevancy penalty as described in this document.

References

[1] The alltheweb search engine. (www.alltheweb.com).

[2] The altavista search engine. (www.altavista.com).

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[4] A. B. et. al. Graph structure in the web. In *Proceedings of the Ninth International World Wide Web Conference (WWW9)*, 2000.

[5] B. J. Jansen, A. Spink, and T. Saracevic. Real life, real users, and real needs: a study and analysis of user queries on the web. *Information Processing and Management*, 36(2):207–227, 2000.

[6] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the Twelfth International World Wide Web Conference (WWW2003)*, 2003.

[7] Z. Lu. Scalable distributed architectures for information retrieval TITLE2:. Technical Report UM-CS-1999-049, , 1999.

[8] Z. Lu and K. S. McKinley. Searching a terabyte of text using partial replication TITLE2:. TechnicalReport UM-CS-1999-050, , 1999.

[9] Z. Lu and K. S. McKinley. Partial collection replication versus caching for information retrieval systems. In *Research and Development in Information Retrieval*, pages 248–255, 2000.

[10] K. M. Risvik, T. Egge, B. Svingen, and A. Halaas. Search engine with two-dimensional linear scalable parallel architecture. International Patent PCT/NO99/00155, 2000.

[11] K. M. Risvik and R. Michelsen. Search engines and web dynamics. *Computer Networks (Amsterdam, Netherlands: 1999)*, 39(3):289–302, 2002.

[12] C. Silverstein, M. Henzinger, J. Marais, and M. Moricz. Analysis of a very large alta vista query log.Technical report, SRC Technical Note, 1998.