Maguro, a system for indexing and searching over very large text collections

Knut Magne Risvik, Trishul Chilimbi, Henry Tan, Chris Anderson, Karthik Kalyanaraman

{knutmr, trishulc, htan, chand, karthk}@microsoft.com

Microsoft Corporation

ABSTRACT

Maguro is a system for efficiently searching very large collections of text content of up to 1 trillion documents at low cost. Search engines span across content that is very dynamic and highly augmented with metadata to the tail content of the web. A long tail distribution of content calls for different trade-offs in the design space for good efficiency across the entire index range. Maguro is designed for the long tail of content with less dynamics and less metadata, but very good cost efficiency. Maguro is part of the serving stack in Bing and allows us to scale the index significantly better.

1. INTRODUCTION

Designing an efficient index serving system to comprehensively index the entire World Wide Web is extremely challenging. Given the versatility of the content on the web, it makes sense for Bing to use a portfolio of technologies for efficient and cost effective index serving for different subsets of the web. The head of the web has extreme dynamics in both content and references, and the amount of information external to the page itself (links, anchors, tweets, likes, shares, etc.) are in many cases vastly larger than the content itself. The number of external signals to determine the actual search engine ranking of a head object is large, and the requirements to handle these dynamics drives the index serving system design. Moving into the web's extremely long tail of content, very few signals outside the content itself exists, and the scale of the web tail drives the requirements for us to build a very efficient index serving system for this content.

The storage hierarchy economics implies that we need to utilize hard drives to handle the web tail in an efficient manner. Maguro is designed to index any atom from a web page (word, n-gram, tuple, and feature) and to distribute the index of each atom across hard disks and machines to optimize the retrieval cost, hence enabling a flexible model for trading off efficiency for retrieval features at fine granularity. Maguro's use of hard drives enables it to index more documents per machines as well as a large selection of compound atoms, such as n-grams and tuples. In addition, Maguro's atom partitioned serving architecture reduces the number of machines that need to participate in every query as compared to a document partitioned serving architecture that involves all machines[13][16][14]. Finally, Maguro takes advantage of its phrase-based indexing with n-gram and tuple based query formulation to additionally reduce both the number of machines that must participate in each query as well as the computation needed to answer the query. Maguro achieves 12x cost efficiency improvement over Bing's baseline search engine that serves head content on the same hardware, and over 40x improvement, when run on hardware with a faster network interconnect. By reducing the number of machines that participate in computing the query results, Maguro is able to demonstrate better latency scaling characteristics and enables providing fast and consistent query response even when indexing and serving a large collection of tail documents.

The rest of the paper is organized as follows. Section 2 talks about overall search engine architecture, introduces the funnel architecture for how we design search engines and helps motivate the Maguro system design based on hardware and scale trends. Section 3 presents the overall Maguro system architecture along with descriptions of the key building blocks. Section 4 provides experimental evaluation of the Maguro system. Finally, Section 5 covers related work, and Section 6 concludes the paper. Protecting Bing's business interests requires us to reduce some level of detail and omit additional information that may have been appropriate. Examples of this include the details of the hash function used, query formulation algorithm, posting list compression, and the omission of absolute numbers from the graphs in Section 4.

2. BACKGROUND

A search engine [13][4][3] is a very complex system operating with multiple large components. Overall, one model of a search engine can be broken down into:

1) Selection and crawling. Using a map of the webgraph to select what documents to schedule for fetching, and a crawler to efficiently download and discover these.
2) Index building by inverting the content of documents into efficient inverted indexes for searching.
3) Matching and ranking documents upon user queries.
4) A search engine user interface to handle the dialog with the user.

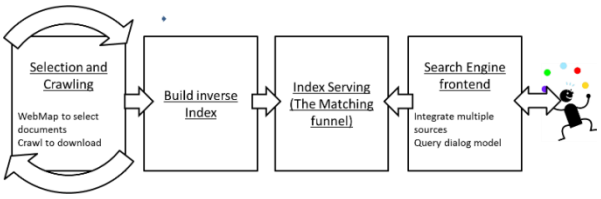

Figure 1: Anatomy of a search engine

2.1 THE MATCHING FUNNEL

We break matching and ranking down into multiple phases that are sequenced in a funnel. A rational breakdown can be:

1) L0 – Matching. Given a query, find all candidate matches for the query.
2) L1 – Preliminary ranking and pruning. The number of matches from stage 1 can be very large, so stage 2 is using efficient and preliminary ranking to prune away matches that are unlikely to be part of the final answer.
3) L2 – Final ranking and sorting. Based on the pruned results from stage 2, use a full featured ranking system to derive final scores and sort the matches.

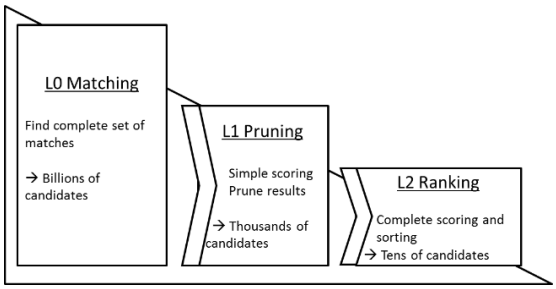

Figure 2: Search matching funnel

This is illustrated above, and Maguro covers both L0 and L1 in this model, and uses pre-computed scores to prune the lists in L1.

2.2 HARDWARE TRENDS

Running a search engine at extreme scale implies a relentless focus on cost efficiency. To the extent possible, datacenters are made up of commodity components. When considering the index serving problem at large scale, recent hardware trends open up and suggest specific architectural directions. In particular we take into account trends in 3 major areas:

1) *HDD cost trends*: HDD storage cost is significantly lower than other storage systems, and currently trending at around $0.05/Gbyte. The technology is expecting to be able to increase the density of storage at least 50x, so there seems to be no immediate physical barrier to be hit here. The challenge is to scale throughput and latency, which is more stagnant.
2) *The evolution of SSD*: Flash-based storage has rapidly entered the stage as a viable storage layer. Since 2007 the cost per GB of SSD has dropped from 120x the cost of HDD to around 25x the cost of HDD, and it has huge advantages in latency of access. Though, there are challenges in lifetime management and write cycles handling.
3) *10Gbit commodity networks*: With the introduction of 10Gbit networking, the bandwidth between machines enables remote storage to be as fast as local storage. Architecturally, this opens up the ability to do more global computation across machines, and this is a key element in the scalability of the Maguro serving platform.

3. MAGURO SYSTEM OVERVIEW

Maguro breaks the searchable index into disjoint segments that are queried in parallel. These segments can be selected based on different criteria such as language, region, topic or static rank. Inside a segment, the index is again distributed onto leaf nodes of computing. At the top, a *CorpusRoot* orchestrates querying across segments, while inside a segment, the *SegmentRoot* handles query execution and result computation. Each leaf node contains an *L01Matcher* component and an *L2Ranker* component that implements the search funnel. Overall the architecture can be illustrated as below.

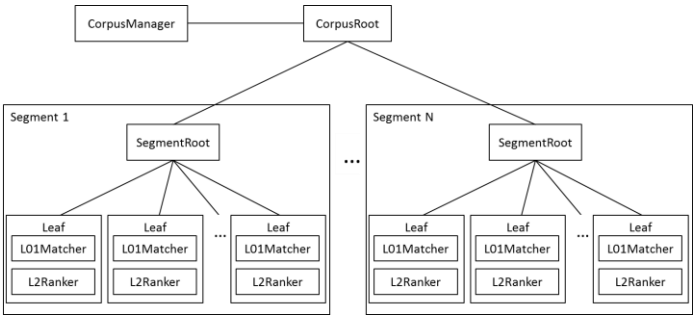

Figure 3: Maguro architecture overview

3.1 Basic Data Model and atom-based indexing

The fundamental data model for Maguro is a mapping from an atom to a list of documents containing the atom called a posting list, e.g:

$$(atom \rightarrow \{(docid, attributes)\}$$

Documents to be indexed are represented as stream of tokens, e.g (w_1, w_2, w_3, …), and Maguro generates atoms from the following logic:

1) All unigrams/words in the stream are atoms.
2) All n-grams (up to a given n) are considered as atoms.
3) A selection of all n-grams (beyond length n) are considered as atoms.
4) A selection of known n-grams of arbitrarily length are considered as atoms.
5) A selection of all tuples (w_i, w_j) are considered as atoms.

The selection process is based on several a-priori distributions from the document corpus and query logs. From the candidate set above, Maguro generates a set of atoms (a_1, a_N) for each document, as a mix of unigrams along with n-grams and tuples that we collectively refer to as phrases. The primary location for the posting lists is on hard-drive, and hence Maguro is more sensitive to the length of each posting list, rather than the total space used for all postings. This makes it feasible to generate posting lists for a large number of atoms and potentially reduce the number of IOPS and the length of each transfer due to richer atoms. For example, the posting list length of n-gram "*ab*" is typically 1/1000 times smaller than the length of the posting lists for its constituent unigrams, *a* and *b*.

The posting lists are generated as the join of documents for a given atom:

$$P = (atom \rightarrow \{(docid, score)\})$$

Here, the score is a pre-computed ranking score, *Rank_L1(query, doc)* with atom itself as the query. We refer to this score as the *L1 Score*. The total index is then the union of all posting lists for all atoms.

3.2 Partitioning the index by atoms

Maguro distributes the index across machines inside a segment by the hash value of each atom. This is similar to global indexing schemes described extensively in the literature [13][16][14]. Hence for a given doc, the postings will be scattered over potentially all the machines inside a segment, implying the need for query execution to coordinate fetching results from the machines.

Partitioning the index by atom instead of document is driven by the desire to utilize hard-disk efficiently. For a given query, the number of IO requests with this scheme is linear to the number of atoms in the query, as compared to a document partitioning model that will issue IO requests for each atom on each machine. Hence if we have a query of 10 words, which is quite feasible after query augmentation and analysis, each segment in an atom partitioned configuration will do O(10) IO operations. If the data was partitioned by documents, and assuming that N machines are needed to index the entire document corpus, we are looking at significantly more IO operations O(10*N). The amortized amount of IO operations in document partitioned models is prohibitive, and hence most search engines will use DRAM or SSD to store such an index, where the IOPS capacity is way higher. Typically, this implies that each node has many fewer documents (say 20M), and a significantly wider fan-out. While the throughput of such a DRAM/SSD based index is expected to be higher, the impact of wide fanout is higher latency variations, and typically this is compensated with large capacity buffers to have less latency flux. Latency is an extremely important element of large-scale online services, and as we will show later in this paper, the lower fanout enables us to run with smaller capacity buffers, making the disk based index very attractive.

3.3 Efficient atom lookup

The search index on each machine comprised of inverted posting lists is organized as a multi-level hash-based index. We use a custom hash function that is fast to compute and has almost zero collisions even when indexing trillions of atoms, while still using a small number of bits for space efficiency. The hash-structure is organized in three levels across DRAM, SSD, and HDD with the most popular atoms in DRAM and the least popular ones on HDD. Atom popularity is determined by occurrence frequency in query logs and the document corpus. When N machines participate in a Maguro segment, the amount of atoms accessible without going to HDD is proportional to N x (DRAM size reserved + SSD size reserved), which is sufficient to ensure that the vast majority of query atoms can be efficiently looked up without accessing HDD. This lookup only provides summary statistics about the atom and requires an additional HDD lookup to retrieve the posting list.

Maguro uses such a hash-based structure because efficient atom lookup is crucial to our phrase-based query formulation where a large candidate set of atoms are examined prior to selecting the subset of atoms whose posting lists need to be examined and intersected. Since not all phrases (n-grams, tuples) are present in our index we need to quickly determine if the atom exists in the index along with other information such as occurrence frequency and score distribution that is important for effective query formulation.

3.4 Handling long posting lists

Maguro's efficiency primarily arises from using HDD to index a much larger number of documents per machine. In addition, since it uses a global indexing scheme [13][16][14] each machine's inverted posting list comprises all documents indexed by the segment. Consequently, posting lists, especially for popular atoms, can grow extremely large and require special consideration for efficient processing. This issue is partially addressed by aggressively indexing and using phrases (n-grams, tuples) in query formulation, which have significantly smaller posting lists as compared to unigrams. In addition, Maguro tiers posting lists based on a posting's pre-computed L1 score. High scoring postings are placed in tier 0, the next set of postings are placed in tier 1, and finally, tier 2 is a catch-all tier that contains all remaining postings. Query planning takes this tiering of posting lists into account while formulating the query. Tier 0 postings are always examined by all queries and the size of tier 0 is tuned to strike a balance between HDD IOPS, transfer latency, and L1 score cutoff for query relevance. Tier 2 postings are too long to store on a single machine and are distributed across all machines in the segment. They are also too long to transfer in a timely manner over the network and are locally intersected against rarer query atoms with shorter posting lists. The results of this intersection are communicated over the network and aggregated. Cascading query execution that we describe later can choose to use tier 1 and occasionally

tier 2 postings when there are insufficient high scoring results from examining only tier 0 postings.

3.5 Query planning

The sharding of the index data across machines calls for more logic in coordinating the query execution. Maguro builds a query plan upon receiving an augmented and parsed query. A query plan is a hierarchical set of instructions that each leaf node is capable of executing. The results of the execution will be sent to a different machine or kept on the local machine for further processing. The instructions use the notion of a sequence as the result of the instruction, which is an addressable array structure. Maguro has an instruction set with the following basic operations:

Operation	Semantics
ATOMSEQ(a)	Generate a result sequence for the atomic unit a. Results returned sorted by document with approximate ranking (L1 doc ranks).
NEARSEQ(a_1, a_2)	Generate a result sequence for the tuple of atoms a_1 and a_2 that are near each other (predefined to be k). Results returned sorted by document with approximate ranking.
ADJSEQ(a_1,a_2, …a_N)	Generate a result sequence for the exact phrase composed of the atoms a_1,..a_N.
AND(s_1, s_2, …, s_N, ∂)	Perform a ranking intersection of all sequences in the argument list, computing the rank as $\partial(R(s_1), R(s_2), \dots, R(s_N))$ where ∂ would default to the sum.
SOFTAND(s_1, s_2, …, s_N, ∂)	Perform a ranking soft intersection of all sequences in the argument list, computing the rank as $\partial(R(s_1), R(s_2), \dots, R(s_N))$ where ∂ would default to the sum. The intersection softly converts into a union operator when stream hits natural boundaries due to tiering posting lists.
OR(s_1, s_2, …, s_N, ∂)	Perform a ranking union of all sequences in the argument list, computing the rank as $\partial(R(s_1), R(s_2), \dots, R(s_N))$ where ∂ would default to the max over the set.
ANDNOT(s_0, s_2, …, s_N)	Perform a disjunction of the first sequence and the N-1 following atomic sequences, not impacting ranking already found in s0.
RELAXEDAND(s_1, s_2, …, s_N , m, ∂)	Perform a relaxed AND where m of N atoms need to be present for the intersection to be considered "successful".
FILTER(s_1, s_2, …, s_N , ∂)	Filter a set of sequences based on document attributes ∂, like language, spam or porn.
WAND(s_1, w_1, s_2, w_2,…, s_N, w_N, ∂)	Perform a ranking intersection with weights on each atom and a threshold ∂, to trigger a "successful" match.

In addition, each instruction also carries metadata about what machine to execute the instruction on, and expected external consumers of the resulting sequence.

Since the query from an end user has no strict Boolean language or constructs in the common case, we need to reformulate the query into a stricter model. Operators like *RelaxedAnd* are helpful to enable a bit more "fuzziness" in the evaluation of the query, and this has positive impact in overall relevance especially for a tail index aiming at good recall [5].

3.6 Query execution

The query execution performance of Maguro is strongly correlated to the ability to use n-gram and tuple atoms in the query. The actual formulation of the query for execution on a Maguro segment is currently based on different aspects:

1) **Signals from query analysis and augmentation.** Bing uses multiple techniques to detect entities, indicate phrase and tuple connections in the query, as well as tag it with classification metadata. This is a rich set of rules, and Maguro uses these signals to formulate parts of the query terms into n-gram and tuple atoms.
2) **Atom statistics itself**. Upon dictionary data extraction in Maguro, we have available statistics about the frequency and score of each atom. This data is stored in the index along with the atom and used to make late-binding decisions about where n-gram and tuple formulations make sense.
3) **Network topology and load state on nodes**. Since queries can be hierarchically evaluated and shipped across the network between machines, we also take the load and topology into account when planning execution. This enables us to dynamically make trade-offs between minimal execution time and avoiding resource congestion. The load and resource state data is continuously updated, and available to any node doing the query planning.
4) **Cascading operations.** Maguro has the capability of cascading query operations through early short-circuit of Boolean expressions. For instance can a query "foo bar" be executed as "ngram(foo,bar) OR

(term(foo) AND term(bar))", where we only evaluate the OR after knowing the statistics of the ngram(foo,bar)."

The leaf machine nodes in a Maguro segment have 3 roles, namely *SegmentRoot*, *L01Matcher* and *L2Ranker*. For L01Matcher, each node has an atom-based partition of the entire reverse index. For the L2Ranker, we partition the ranking data needed across the segment of machines with a uniform distribution and place this data on SSD. Hence, each machine has a range of documents, $D_i..D_j$, for which it has ranking information. illustrates this along with our query execution workflow.

Query execution takes place as follows:

1) Any machine can be a *SegmentRoot*, and for each query a random machine is selected. The query is routed down to one of the *SegmentRoot* machines, and the machine computes the query plan as described above.
2) During query execution, the root machine that is the *SegmentRoot* selected in step 1 for the query calls out to the *L01Matchers* holding posting lists for each atom in consideration in the query plan, or it even sends partial query plans out to other nodes for execution. A map of all machines in the segments (with partition and load information) is locally available in any machine to guide routing.
3) Eventually, the root machine assembles the final result set.
4) The top N results are selected for second-level ranking, and again the map is consulted to find the required *L2Rankers* to do the final ranking of the top N results.
5) The re-ranked results are aggregated and returned to the calling service.

This process is done in parallel on each segment, and is illustrated in steps below:

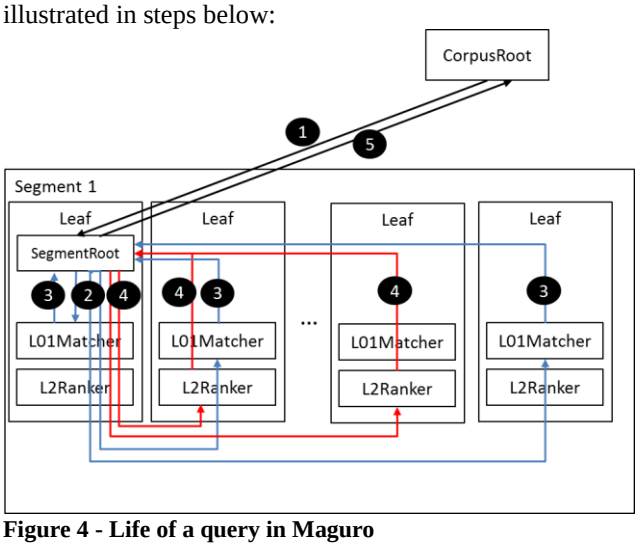

Figure 4 - Life of a query in Maguro

3.7 Fault-tolerant serving operations

Fault-tolerant operations are crucial to running an online search engine. It is also important to have stable performance and fast failovers. The failure scenarios to handle within a Maguro segment are as follows:

1) **Machine failure.** A machine(s) is failing due to software or hardware problems.
2) **Network partition.** A machine is only reachable from a subset of the expected network, hence the network has an error.
3) **Load hotspot occurring on machine.** Machines fail queries or are very slow in answering queries due to high load.
4) **Slow or partially failing machine.** Disk errors, or general outlier characteristics occur on a machine.
5) **Planned maintenance.** Updating a service requires it to gracefully finish its current tasks and shut down. Also merging, updating and OS upgrades have this same code path.

All machines in a Maguro segment have a map of what it perceives as the state of each machine in the segment. This map is used both for marking failed nodes and partitions as well as for dynamic query planning and load balancing. The map holds the following information for all nodes in the segment:

Field	Description
Node	Network address of the node
State	Overall state of machine (UP, DOWN, PROBATION)
L01Load	Load information for the L01 processes on the node
L2Load	Load information for the L2 processes on the node
L01Partitions	List of partitions for L01Matching on this node
L2Partitions	List of partitions for L2 ranking on this node
CPU state	Total load state for CPU
Disk state	Total load state for disks on this machine
Network	Total load state for network on this machine
Connection Status	Immediate status of the remote connection

Failures are detected by multiple mechanisms:

1) **Peer-based watchdogs.** Each machine in a segment is responsible for watching a set of other machines by monitoring heartbeat requests from other machines. Lack of response will have a machine moved into PROBATION state. If it is a

temporal failure and does not recur, it will be moved back into UP state. Otherwise it will be marked as DOWN by the watchdog and eventually that state is shared.

2) **Absolute watchdogs.** Maguro operates in the Microsoft-based cluster management system [10] that also holds dedicated watchdogs; these will ping and send sanity RPC requests to machines and trigger DOWN state settings if no response occurs.
3) **Failure detection in calling remote machines.** Every call to a remote machine for query execution can fail, and repeated failures causes a machine to be first placed in a probation state.
4) **Self-detection**. Machines can also mark themselves as bad from internal failures and errors detected in serving software or through periodic integrity checks being run.

When a machine is marked as DOWN, it can return to an UP state by self-fixing and transitioning through PROBATION mode. When a machine registers that there is consensus in the map that it is DOWN, it can run a set of verifications about its ability to serve. If these pass, it can register itself into PROBATION mode. When the probation state is registered across the segment, other machines will probe connectivity and serving availability and gradually form a consensus that the machine is ok, and can get back into the UP state. A machine with permanent failure due to hardware failure(s) will be fixed by replacing with a spare machine. Once the repair process is completed the new machine will be tagged as UP.

The map that carries the machine states is updated across the segment by means of two mechanisms. In regular update mode, a simple gossip protocol is used where each node shares its current map with others. Each receiving node will update its own map based on the receiving maps by accumulating load and serving state information across the segment machines. This protocol will never reach consensus on load information, but it will give a balanced view of load across the segment machines, so that a query plan will never generate amplifications around hotspots. In addition, a subset of machines in a segment form a Paxos protected quorum [11] to make consistent decision about state changes (DOWN to PROBATION, PROBATION to UP).

3.8 Continuous Index Updates

Maguro generates a relatively large index through exploding the number of atoms indexed per document. Maguro Index Build employs a true delta update mechanism. There is a continuous stream of new and updated documents coming in from the crawl and document processing systems. The documents classified as tail documents are accumulated and processed in a batch for processing efficiency. The computational nature of building index updates maps very well to the computing paradigm championed in Cosmos/Scope [7] and the Dryad engine [9].

3.8.1 Building the index

The continuous stream of tail documents flowing from the crawl and document processing subsystem are assigned and accumulated into the appropriate Maguro segment streams (Queues) in Cosmos. When the count of unprocessed documents for a segment exceeds a threshold, an index build job (SCOPE job) is spawned to process the accumulated documents to build index updates for the segment.

An index build job processes a batch of documents and produces the following index update artifacts which constitutes a mutation to the index served by the segment.

1) Inverted Index updates: These constitute updates to the inverted index that is used to match user queries against documents served by the segment. Every document is processed to produce a set of hit records {Atom Bucket, Atom Id, Document Id, Ranking payload} for the document. Since Maguro employs an atom partitioned inverted index within each segment, the hit records are grouped first by the atom bucket, then grouped by the atom Id, ordered by the Document Id and output into a set of inverted index update files (1 per atom bucket).

2) Forward index updates : These constitute updates to the per document index used for final ranking (L2 ranking) for the set of candidate documents that matched the user query. Every document is processed to produce a set of per document index records {Document Bucket, Document Id, per document index blob}. Since this index is partitioned by document within each segment, the document records are grouped by document bucket, ordered by document id and output into a set of forward index update files (1 per doc bucket).

3) URL to Document Id Mapping: Since Maguro is a large scale index (dealing with potentially a trillion documents), it is expensive for the index build sub system to assign a globally unique document id to every document. The Document Id employed by an Index Build Job is typically only guaranteed to be temporally unique (perhaps within the context of a single job/mutation). Each serving segment in Maguro employs its own scheme of document id assignment to identify a document uniquely in its index. The index build job also outputs the URL→Document Id map employed in the mutation which is used by the segment nodes to remap the index build document

ids to the actual serving document ids for the documents that got processed in the mutation.

3.8.2 Updating serving index

Each Index node in the Maguro segment monitors a special location in cosmos to check if there are any new updates produced for its segment. When new updates are available, each node downloads the portion of the updates that concern that node (based on the atom bucket and document bucket assignment to segment nodes). The serving nodes download the updates and also transform them by reassigning document Ids to facilitate efficient index merge.

Periodically the serving nodes are instructed to merge the updates accumulated into the index served by the node. For document level consistency the merge is constrained to include only the set of index build updates that are present on all the nodes in that segment. Once the set of updates for a round of update merge has been established, the index update reduces to an n-way merge of the served index files with the update files.

Listed below is an outline of the update merge process.

1) Merge URL → Document Id map in the served index with updates to the Map that were computed as a part of the transformation process and build a bit vector (D) of documents deleted from the index. This reduces to a simple n-way merge of the map files with conflict resolution handled as follows
    a. If there are multiple entries for a URL then the highest document id wins and the bits for other document ids for the document are set in the deleted documents bit vector (D).
2) Inverted Index Merge is an n-way Merge of the posting lists from the base index and the inverted index update files. The postings for document ids that have bits set in deleted document bit vector (D) are dropped from the merged index.
3) Forward index is also an n-way merge of the served index file with forward index update files. The per document records for document ids that have bits set in deleted document bit vector (D) are dropped from the merged index.

3.9 Index Growth

The number of documents indexed by a Maguro machine is determined by local HDD capacity. The number of machines (N) in a Maguro segment is determined by the QPS and serving latency SLA (Service Level Agreement) requirements as these dictate the HDD IOPS and network bandwidth that the segment must support. Consequently, index growth is achieved by adding more segments to the system architecture rather than adding more machines to a segment. Adding more machines to a segment while keeping the number of documents indexed per machine constant will increase serving latency as more postings must be examined to return relevant results. The downside of adding entire segments rather than increasing the size of a segment is that index growth in not very granular, but this is not a significant concern for the tail.

3.10 Discussion

Maguro achieves its cost-efficiency goals for serving a large tail document corpus through a well-designed and synergistic combination of techniques that exploit low HDD costs along with availability of SSD and high speed commodity networks. The use of HDD allows each machine to index and serve a significantly larger amount of documents. This also forces the index to be partitioned across machines by atoms rather than documents due to HDD IOPS considerations as described earlier. Since atoms are partitioned across machines, multi-word queries require postings lists to be transferred across the network and intersected and benefit from a high-speed network. The use of HDD also permits indexing n-grams, tuples, and phrases in addition to unigrams and allows query planning to leverage this to reduce the number of remote posting lists that need to be transferred over the network for a multi-term query. Posting lists contain pre-computed ranking scores and this enables tiering posting lists based on these scores. This use of tiering for posting lists cuts down on the average quantity of remote postings that need to be transferred over the network. Pre-computing and storing ranking scores in the inverted index trades off experimental agility for execution efficiency and is an appropriate design point for indexing the web's long tail of content. Finally, separating the index data used for final L2 ranking and placing this on SSD frees this ranking from HDD IOPS constraints and increases the number of documents that can be ranked.

4. EXPERIMENTAL EVALUATION

Maguro indexes tail documents and is part of the serving stack in Bing. In this section we evaluate the serving efficiency and scaling characteristics of Maguro by comparing it against the baseline serving system, which is a conventional document partitioned, RAM-based architecture, that indexes and serves head documents.

4.1 Methodology and Metrics

Search is a large-scale online distributed system and both query latency, and throughput, measured in queries per second (QPS), are key performance metrics. Query latency has strict SLA requirements (average latency, 95% latency, etc.) that must be satisfied both for good user experience and to meet contractual obligations. Hence the performance objective is to maximize the system throughput in terms of QPS while successfully meeting all query latency SLAs. However, this QPS metric and overall system cost is highly sensitive to the configuration of the search engine in terms of how many million documents are indexed by a single machine. To account for this, we use *DQ*, which is the

product of QPS and millions of documents indexed per machine, as our search efficiency metric. Computing *DQ* for a document partitioned architecture is trivial as queries are sent to all machines in parallel. However, for an atom partitioned architecture only a subset of machines participate in each query so the effective *DQ* of a machine in an atom partitioned segment is computed as ((*Documents indexed by the segment*) x (*QPS of the segment*))/(*Number of machines in the segment*).

Search relevance as measured by normalized discounted cumulative gain (*NDCG*) is another key search engine metric [11]. For relevance measurements, Maguro is queried in parallel with the baseline system serving head documents, results from both tiers are aggregated, and the top scoring results across these tiers are surfaced. All measurement reported here are for Maguro configurations where the *NDCG* as measured on multiple collections of search queries used by the Bing relevance team is either at parity or better than the baseline system when Maguro results are not surfaced.

We perform all measurements using a custom capacity and latency testing tool to send several millions of user queries taken from query logs to production beds serving the baseline index and the tail index served by Maguro, that were temporarily removed from the active serving rotation.

4.2 Comparative search efficiency

We compare the search efficiency of Maguro measured in DQ against the baseline system that indexes and serves head documents. We note that the comparison is likely biased against the baseline system due to availability and usage of many more signals for head documents as compared to tail documents that must be taken into consideration while matching and ranking documents, though the magnitude of improvement suggests that this is a second order effect.

The first set of comparisons measure the performance of the Maguro search engine running on identical hardware as the baseline system. Next, we measure the efficiency of the Maguro system running on machines with a fast network interconnect. In addition, for both sets of tests we also evaluate a Maguro configuration where use of n-grams and tuples in query formulation is disabled. Figure 5 shows the results of these measurements. The results indicate that Maguro shows over 12x efficiency gains over the baseline system running on identical hardware. Using n-grams and tuples in query formulation contributes a little bit less than 3x to this overall efficiency gain. On hardware with a fast interconnect, Maguro's efficiency improvements dramatically increase to over 40x the baseline system, with usage of n-grams and tuples in query formulation again contributing a bit less than 3x to the overall efficiency gain. The faster network interconnect permits increasing the QPS the system can sustain while still meeting its latency SLA.

Maguro's efficiency arises from using hard disks to significantly increase the number of documents that can be indexed and served per machine and the atom partitioned serving architecture reduces the number of machines that need to participate in every query. Maguro's phrase-based indexing and query formulation further reduces both the number of machines that must participate in each query as well as the computation needed to answer the query.

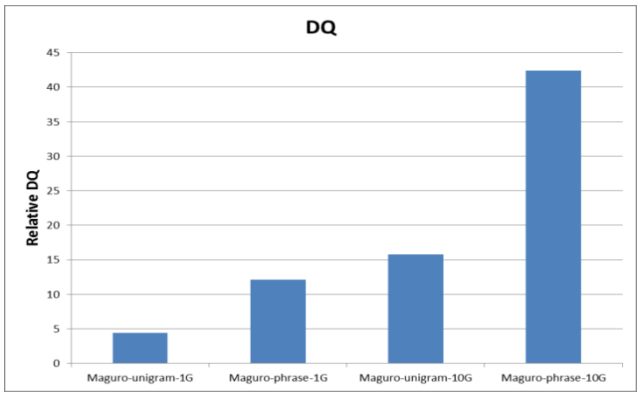

Figure 5: Maguro performance characteristics

Figure 6 shows the index size comparison at different scales between Maguro's phrase based index and a unigram index for the same set of documents. Maguro is selective about indexing n-grams and tuples as described in section 3 but despite this the index size is over 20x larger. This index size increase is possible because Maguro stores its index on hard disk (several terabytes per machine), and is justified because the use of n-grams and tuples provides almost a 3x improvement in serving efficiency.

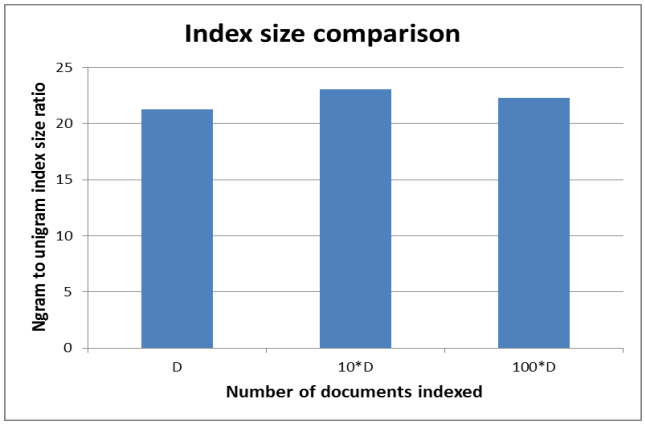

Figure 6: Maguro index size

4.3 Serving latency characteristics

Search is an online distributed system and users expect quick responses to their queries. In addition, the user experience is not just influenced by the average query response latency but it is also sensitive to higher percentile latencies, such as the 95% latency. Providing low latency response even at the 95% is extremely challenging for large distributed systems where many machines must participate in computing the response. In addition, for a consistent user experience the

gap between the average latency and 95% latency should not be too large.

Figure 7 shows the 95% latency comparison between the baseline search engine and Maguro at three different scale points. Each scale point corresponds to system configurations that index the same number of documents. The first data point corresponds to a single machine configuration for the baseline system indexing several million documents. Since the smallest Maguro configuration is an atom-sharded segment that involves many more machines, the 95% latency is around 70% higher than the baseline single machine system. The next data point corresponds to the baseline system indexing all the head documents. Maguro uses an identical number of machines (as the first configuration) to index these amount of documents and as a result its 95% latency is comparable to the first configuration. However, since the baseline system now has to send the query to many more machines, its 95% latency is now three times higher than the first configuration and also much higher than the Maguro system indexing the same number of documents. Finally, the third data point corresponds to the full Maguro system indexing all the tail documents. The increase in 95% latency for Maguro is modest even at this scale and still lower than the 95% latency of the baseline system in the second configuration where the number of documents indexed is significantly smaller. The baseline system does not have sufficient machine capacity to index and serve this amount of documents. But even if it were capable, the 95% latency would be much higher due to the large number of machines required.

Maguro's 95% latency is lower than the baseline system primarily because a smaller number of machines are involved in responding to individual user queries. This reduction in machine fan-out arises from indexing a larger number of documents per machine by leveraging hard disk and from Maguro's phrase-based atom partitioned architecture where only machines than contain query atoms participate in computing the query response. This low machine fan out also enables Maguro to scale out and index a large number of documents without significantly impacting the 95% latency.

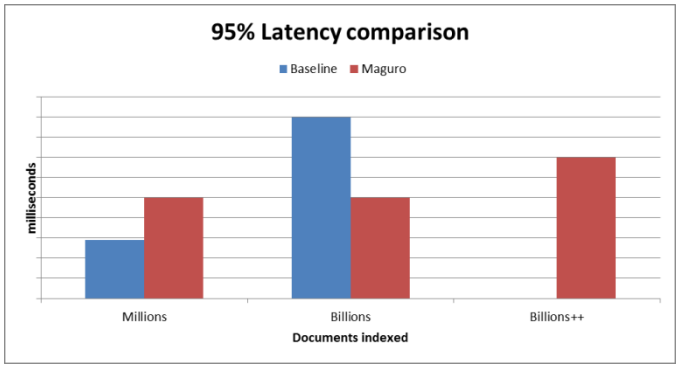

Figure 7: 95% Latency scaling

Figure 8 indicates the ratio of 95% latency to average latency as the load on the system is increased for system configurations corresponding to the second data point in Figure 7. Maguro, despite being a hard disk based index, has a reasonable 95% to average latency ratio. In addition, due to its comparatively lower machine fan-out, Maguro has stable 95% latencies that increase at a lower rate than the average latency. As a result, the ratio of 95% to average latency decreases as the load on the system is increased. By contrast, the baseline index serving system shows more conventional behavior where the ratio of 95% to average latency grows with increasing load.

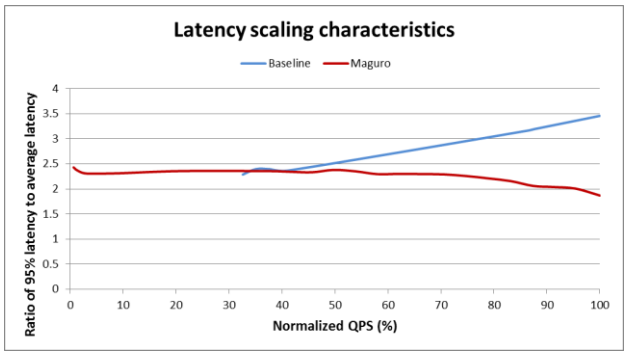

Figure 8: Latency scaling with increasing load

5. RELATED WORK

Discussions around methods for distribution of inverted posting lists have been studied for many years. Conclusions vary quite a bit, there are several papers that conclude on that term-sharded models are superior on performance [18]. Though, the commercial scale of search engines and the impact on commodity hardware usage and datacenter network models makes it hard to rely too much on these results.

The overall aspects of a search engine architecture is described in several keynote talks, and also described in [3].

6. CONCLUSIONS

Maguro is designed for efficiently searching very large collections of text content of up to 1 trillion documents at low cost. It achieves this by using hard disks to significantly increase the number of documents that can be indexed and served per machine. In addition, it employs an atom partitioned serving architecture to reduce the number of machines that need to participate in every query. Finally, Maguro's phrase-based indexing and query formulation uses a large selection of n-grams and tuples to additionally reduce both the number of machines that must participate in each query as well as the computation needed to answer the query.

Maguro achieves 12x cost efficiency improvement over Bing's baseline search engine that serves head content on the same hardware, and over 40x improvement, when run on hardware with a faster network interconnect. By reducing the number of machines that participate in computing the query results, Maguro is able to demonstrate better latency scaling characteristics and enables providing fast and consistent query response even when indexing and serving a

large collection of tail documents. Maguro is part of the serving stack in Bing and allows us to scale the index significantly better.

7. ACKNOWLEDGEMENTS

Special thanks to Daniel Yuan, Madan Musuvathi, Reddy Duggempudi, Vishesh Parikh, Tanj Bennett, Richard Kasperski, Dexin Zhu, Lin Song, Nan Zhang, Jinru He and Bing Shi who made several contributions to the Maguro system. In addition, we would like to thank Qi Lu, Harry Shum, and Chad Walters for their support throughout the project.

8. REFERENCES

[1] Badue C., Baeza-Yates R., Ribeiro-Neto B., and Ziviani N. 2001. Distributed query processing using partitioned inverted files. In G. Navarro, editor, *Proc. String Processing and Information Retrieval Symp.*, IEEE Computer Society, Laguna de San Rafael, Chile, 10–20.

[2] Baeza-Yates, R., Ribeiro-Neto, B. 1999. *Modern information retrieval*. ACM Press, New York, NY.

[3] Barroso, L. A., Dean, J., & Holzle, U. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro*, *23*, *2*, 22–28.

[4] Bing L. 2011. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*, Springer Berlin Heidelberg.

[5] Broder, A., Carmel, D., Herscovici, M., Soffer, A., Zien, J. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the CIKM*, 2003.

[6] Chaiken, R., Jenkins, B., Larson P., Ramsey, B., Shakib, D., Weaver, S., Zhou J. SCOPE: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow. 1, 2.* August 2008.

[7] Greenberg, A., Hamilton, J., Jain N., Kandula, S., Kim, C., Lahri, P., Maltz, D., Patel, P., Sengupta, S. VL2: A scalable and flexible data center network. *Communications of the ACM*, March 2008.

[8] Greenberg, A., Lahri, P., Maltz, D., Patel, P., Sengupta S. Towards a next generation data center architecture: scalability and commoditization. *Proc. ACM workshop on Programmable routers for extensible services of tomorrow.*, 2008.

[9] Isard, M., Mihai, B., Yuan, Y., Birell, A., Fetterly, D. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. *Proc. Of ACM Eurosys*, 2007.

[10] Isard, M. 2007. Autopilot: Automatic Data Center Management, in *Operating Systems Review*, 41, 2 (April 2007), 60-67.

[11] Jarvelin, K., Kekalainen, J. Cumulated gain-based evaluation of IR techniques. In *ACM Transactions on Information Systems 20(4).* 2002.

[12] Lorch J. R., Adya A., Bolosky W. J., Chaiken R., Douceur J. R., Howell J. 2006. The smart way to migrate replicated stateful services, In *Proceedings of ACM Eurosys*.

[13] Manning C. D., Raghavan P., Schutze H. 2008. *Introduction to Information Retrieval*, Cambridge University Press.

[14] Marin, M., Gomez, C., Gonzalez, S., Costa, G.V. 2008. Scheduling Intersection Queries in Term Partitioned Inverted Files, 14th European Conference on Parallel and Distributed Computing (EuroPar 2008), LNCS, Springer (Aug. 26-29), Spain.

[15] Melink, S., Raghavan, S., Yang, B., Garcia-Molina, H. 2001. Building a distributed full-text index for the web. *ACM Trans. Inf. Syst.* 19, 3, 217–241.

[16] Moffat, A., Webber, W., Zobel, J. 2006. Load balancing for term-distributed parallel retrieval. In *Proceeding of SIGIR 2006: 29th annual international ACM SIGIR conference on Research and development in information retrieval*, 348–355.

[17] Moffat, A., Webber, W., Zobel, J., Baeza-Yates, R. 2007. A pipelined architecture for distributed text query evaluation. *Information Retrieval*, 10, 3, 205-231.

[18] Ribeiro-Neto, B., Barbosa, R. Query performance for tightly coupled distributed digital libraries. *In Proc. ACM Digital Libraries*, June 1998.

[19] Tomasic, A., H. Garcia-Molina, H. 1996. Performance issues in distributed shared-nothing information-retrieval systems. *Information Processing & Management*, 32, 6, 647–665.