Optimal Aggregation Policy for Reducing Tail Latency of Web Search

Jeong-Min Yun[1], Yuxiong He[2], Sameh Elnikety[2], Shaolei Ren[3]

[1]POSTECH [2]Microsoft Research [3]Florida International University

ABSTRACT

A web search engine often employs partition-aggregate architecture, where an aggregator propagates a user query to all index serving nodes (ISNs) and collects the responses from them. An aggregation policy determines how long the aggregators wait for the ISNs before returning aggregated results to users, crucially affecting both query latency and quality. Designing an aggregation policy is, however, challenging: Response latency among queries and among ISNs varies significantly, and aggregators lack of knowledge about when ISNs will respond. In this paper, we propose aggregation policies that minimize tail latency of search queries subject to search quality service level agreements (SLAs), combining data-driven offline analysis with online processing. Beginning with a single aggregator, we formally prove the optimality of our policy: It achieves the offline optimal result without knowing future responses of ISNs. We extend our policy for commonly-used hierarchical levels of aggregators and prove its optimality when messaging times between aggregators are known. We also present an empirically-effective policy to address unknown messaging time. We use production traces from a commercial search engine, a commercial advertisement engine, and synthetic workloads to evaluate the aggregation policy. The results show that compared to prior work, the policy reduces tail latency by up to 40% while satisfying same quality SLAs.

1. INTRODUCTION

A web search engine often employs partition-aggregate architecture, forming a distributed system with aggregators and index serving nodes (ISNs). The web index is large, containing information on billions of web documents, and is typically document-sharded among hundreds of ISNs [7, 12]. Such a system includes aggregators that merge results collected from ISNs (e.g., a single aggregator (Figure 1(a))).

An *aggregation policy* determines how long the aggregator waits for its ISNs before sending the results back. Designing a good aggregation policy is crucial: It directly impacts quality and responsiveness, which are among the most important metrics in web search. For example, the aggregator can choose to wait for responses from all ISNs, returning complete results, or it may wait for only a fixed time duration, returning partial results from a subset of ISNs. There is a clear trade-off: The longer the aggregator waits, the more results it collects, improving quality but degrading responsiveness. The objective of the aggregation policy can be expressed as reducing a high latency percentile while meeting one or more quality constraints. Reducing a high latency percentile, i.e., tail latency, is more suitable than average latency to ensure *consistently* low latency, which is important to attract and retain users [4, 17, 25, 27, 33].

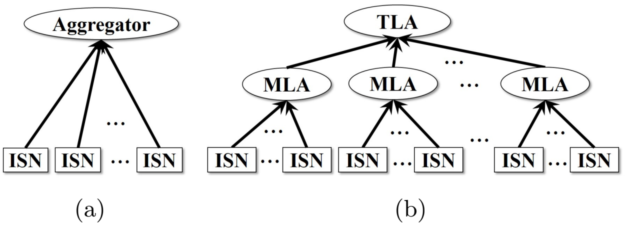


Figure 1: Web search engine architecture: (a) single-level aggregation, (b) two-level aggregation.

Designing an effective online aggregation policy is challenging: Aggregators do not know when ISNs will return their results and their response times vary significantly. For example, different web search queries exhibit highly variable demand: the 99-th percentile query execution time is often a factor of 10 larger than the average [13, 22]. Even for the same query, the response times of different ISNs could vary widely [5]: It may take only 10 ms to collect responses from a majority of ISNs but require another 200 ms to receive the remaining few.

Prior work provides aggregation heuristics that reduce the tail latency compared to simply waiting for all responses, but they miss significant potential latency improvement. Instead of using heuristics, we take a different approach: Imagine if the aggregator knows the future, in particular when each ISN will respond. We study this case to derive an offline policy, which decides when to return the results on a per-query basis. We further show that this offline policy is optimal, forming a clear latency target that an online policy strives to achieve.

However, in practice, the aggregator must make online decisions without knowing the future information (e.g., when

SIGIR'15. August 09 - 13, 2015, Santiago, Chile.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3621-5/15/08:.$15.00.
DOI: http://dx.doi.org/10.1145/2766462.2767708.

ISNs will respond), which invalidates offline policies and makes the problem challenging. When an online aggregation policy decides to terminate a query and returns its partial results, it makes a bet that waiting for the remaining responses to arrive will take too much time. However, all the remaining responses might arrive immediately afterwards, indicating that search quality was degraded with only a marginal latency reduction.

Inspired by the offline policy, we develop an online policy that uses two thresholds, one for query quality and the other is a time threshold. Only if a query meets or exceeds the quality threshold at the time threshold, the aggregator terminates the query. We employ offline analysis of the query logs to compute the quality and time thresholds.

We prove a surprising result: The aggregation policy is optimal, providing the same results as the offline optimal policy. It minimizes the tail latency of queries while meeting their quality service level agreements (SLAs). In contrast, prior work does not identify the optimal structure of the online policy, or compare to the offline optimal solution. They employ heuristics to reduce tail latency indirectly, for example, by minimizing latency variance [20].

Next, we study multiple levels of aggregators, which is an important practical case yet largely unaddressed in prior work. There is no heuristic specifically designed for multiple aggregators. Figure 1(b) shows a cluster with two-level aggregation, containing a single top-level aggregator, called TLA, and several mid-level aggregators, called MLAs. Microsoft Bing search employs the two-level aggregation architecture [1]. This architecture fits naturally with the hierarchy of data center networks, where MLAs collect responses from ISNs located in the same rack and TLA collects responses from the MLAs, reducing inter-rack communication. Moreover, MLAs can reduce the processing load of the top aggregator, for example by re-ranking results and removing near duplicates [34].

Multilevel aggregation is significantly more challenging than single-level aggregation because the aggregator's decisions at the different levels are coupled and cannot be optimized independently. For example, an aggressive TLA policy (e.g., terminating queries based on a lower time or quality threshold) needs to be combined with a more conservative MLA policy to meet the quality SLAs. Finding a good policy requires exploring various combinations of aggregation decisions at different levels and deciding the parameters used at each level jointly, which significantly increases the design space and computation complexity. Further, messaging time between aggregators (including both processing and transmission time) may or may not be known online.

We extend our policy to address the challenges for multilevel aggregation and prove its optimality when messaging times between aggregators are known. For unknown messaging times, we show that the policy performs very close to the offline optimal using empirical evaluation. To the best of our knowledge, this policy is the first one to solve the multilevel aggregation problem.

We assess our techniques using query logs from a production cluster of a commercial search engine, a commercial advertisement engine, as well as using synthetic workloads derived from well-known distributions. We evaluate the policy extensively in both single-level and two-level aggregation clusters with different quality and latency targets, varying the number of ISNs, latency distributions. We conclude our policy consistently outperforms the state-of-the-art techniques. In particular, the results show, to meet the target quality constraints, on a 44-ISN search cluster with a single aggregator, we reduce the 95-th tail latency by 36% (from 59 ms to 38 ms) compared with the best existing work. The results also show similar savings for two-level aggregation.

The key contributions of this work are the following: (1) We characterize the workload to outline how a good aggregation policy should behave. (2) We develop an optimal aggregation policy, combining both data-driven offline analysis with online processing, to minimize a crucial metric — tail latency of search queries — subject to quality constraints. (3) We generalize our policy to address multilevel aggregation, which has practical importance but has not been addressed in prior research. (4) We conduct extensive evaluation using production traces as well as various synthetic workloads to assess the benefits of the proposed aggregation policy and show that it outperforms existing solutions.

2. PROBLEM FORMULATION

Aggregation policy assigns an online algorithm for each aggregator to decide when it returns its response.

Latency. Web search requires *consistently* low latency to attract and retain users [4, 17, 25, 27, 33], for which average latency is not a suitable metric due to potentially very large variation in latency. Thus, service providers often target to reduce high percentile latency of web search queries, which is also called *tail latency* [13, 19, 31]. We define *query latency* as the latency measured at top aggregator.

Quality. To quantify search quality, we define the *utility* of a query as the fraction of ISNs from which the TLA receives results and returns to the user. This definition is also used in prior work [20]. Lower utility indicates the query is more likely to miss important answers, thus resulting in a lower quality. A utility of 1 is the best baseline case: The aggregators receive responses from all ISNs.

We use both average utility and tail utility, which are two important metrics for web search SLAs [2, 19]. The average utility is the mean utility of all queries, and tail utility is high-percentile utility of queries, ensuring consistently high quality results for the search queries.

Optimization Objective. The objective of an aggregation policy is to minimize query tail latency of a large-scale web search engine, while satisfying average and tail utility constraints. This is a common objective in many prior studies [11, 20, 22] as well as in real systems. For example, the optimization criteria of Bing search is to minimize the 95-th tail latency subject to an average utility of 0.99 and 95-th tail utility of 0.95.

Given an aggregation policy $\mathcal{A}$ and a set of n queries, we denote the latency and utility of query i by t_i and u_i, respectively, for $i = 1, 2, \cdots, n$. The k-th percentile latency for the set of n queries, denoted by $t_{k\%}$, is the $\lceil kn \rceil$-th largest latency among these n queries. Likewise, we can compute the h-th percentile utility, denoted by $u_{h\%}$, as the $\lceil hn \rceil$-th lowest utility. Note that both latency and utility are affected by the aggregation policy $\mathcal{A}$. Thus, our tail latency minimization (called TLM) problem is formulated as:

$$\mathsf{TLM}: \quad \text{minimize}_{\mathcal{A}}\ t_{k\%} \tag{1}$$

$$s.t., \quad \frac{1}{n}\sum_{i=1}^{n} u_i \geq \bar{u}_{\text{avg}} \text{ and } u_{h\%} \geq \bar{u}_{h\%}, \tag{2}$$

where (2) specifies the average and tail utility constraints.

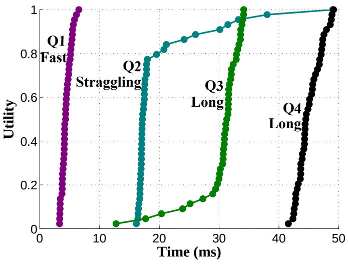


Figure 2: Progress of 4 sample queries.

Table 1: Aggregation Policy Comparison.

	Complete Fast	Terminate Straggling	Complete Long	Inputs		Multi level
				Time	Utility	
Wait-All	✓	✗	✓	✗	✗	✗
Time-Only	✓	✓	✗	✓	✗	✗
Utility-Only	✗	✓	✗	✗	✓	✗
Time-Utility	✓	✓	✗	✓	✓	✗
Kwiken	✓	✓	✗	✓	✓	✗
FSL	✓	✓	✓	✓	✓	✓

Production Deployment Environment. When an aggregator decides to return partial results, there is an implementation choice to either abort query execution on the ISNs that have not yet responded or to let them complete. The later is more common in practice as observed in both commercial search engines, e.g., Bing [20], and the well-known open source search software Apache Lucene [18]. Supporting query abort has a cost and comes with limited benefits.

Supporting abort messages and interrupting running queries increases search engine implementation complexity, making logging, testing, debugging more difficult in addition to the message processing overhead of the abort messages. Also, the computational resource savings from aborting a late query are often negligible. More specifically, as query processing on ISNs has a timeout limit (e.g., $\leq$ 200 ms in Bing search [19]), no query takes longer processing time than the timeout limit. Moreover, to meet an average utility of 0.99, at most 1% of ISNs could terminate processing earlier using an abort message, making the overall computational resource savings limited. Finally, aborting late messages also has little impact on the waiting time (and thus the latency) of other concurrent queries, because commercial search engines are provisioned with additional capacity to ensure that ISNs operate at low to modest loads with a very small query waiting time [21].

Therefore, we present and evaluate our aggregation policies based on systems without abort messages. However, as the performance impact of abort messages is small, these policies are also well applicable to systems with aborts.

3. INTUITIONS

We derive the key intuitions of our aggregation policy using query workload characteristics. Then, we discuss what a good policy should do and what existing policies miss.

Workload Characterization. The execution time distribution of web search queries has a high variability: Prior work shows that the 99-th percentile query execution time is a factor of 10x larger than the average, and even 100x larger than the median [13, 22]. In addition, for processing a single query, some ISNs may take much longer than others even if they all use the same server hardware due to several well-known factors, such as document partitioning scheme, correlation between query term frequencies and inverted list sizes [5].

Figure 2 shows utility progress for a few examples from Bing. We categorize queries into three types: (1) *Fast* query for which the responses from all ISNs arrive quickly, e.g., Q1 in Figure 2. (2) *Straggling* query for which most of the responses arrive quickly but a few ISNs take much longer to respond, e.g., Q2 in Figure 2. (3) *Long* query for which most responses take a long time, e.g., Q3 and Q4.

Key Intuition. Our policy identifies query type and decides which queries among the three types to complete or terminate by considering the tradeoff of latency and utility:

Complete fast queries. Since they do not impact tail latency even if we wait for all responses, we complete them with full utility.

Terminate straggling queries. We reduce latency with little utility loss. For any given utility target, we can only tolerate a limited utility loss. For example, for an average utility of 0.99, only 0.01 utility loss is allowed. We allocate the allowed utility loss to straggling queries to maximize latency reduction.

Complete long queries. As their responses from ISNs arrive late but altogether at TLA, terminating them early can cause major utility loss. Thus, we let long queries complete without affecting the tail latency, because a latency percentile target allows for a few slow queries without penalty (e.g., for 95-th latency, 5% slow queries are allowed), which we call *tail latency slackness.*

The distinction among fast, straggling and long queries is relative, depending on the selection of time and utility thresholds obtained by offline analysis of query workloads.

Existing Policies. Table 1 compares six aggregation policies with respect to their actions on the three types of queries, inputs, and support for multilevel aggregations.

Wait-All waits for every ISN before returning the results. The query latency is dominated by the slowest ISN, resulting in a high latency. *Time-Only* terminates a query upon a timeout threshold [22]. It terminates both long and straggling queries without differentiation. *Utility-Only* terminates a query if it already received a given percentage of responses, even including fast queries whose full completion does not affect tail latency. These three policies miss either time or utility to differentiate among query types.

Time-Utility terminates a query when its timeout threshold is triggered and the query utility reaches a threshold. It is better than Time-Only, but it still early terminates long queries after they receive the threshold percentage of responses. *Kwiken* [20] terminates a query if it runs a fixed time interval either after a utility threshold or exceeding a time threshold. It selects these thresholds by minimizing latency variance among all responses. Although minimizing latency variance may indirectly reduce tail latency, it can degrade utility significantly. For example, in Figure 2, Q2 and Q3 have similar variances in terms of latency from their ISNs. However, terminating the straggling query Q2 causes little utility loss, while terminating a long query Q3 results in a large utility loss. Although Time-Utility and Kwiken

Algorithm 1 FSL: Online processing

```
Input: time threshold $t^*$ and utility threshold $u^*$
for each query do
    if all the responses are returned within $t^*$ then
        Do nothing
    else if utility $u \geq u^*$ after the aggregator waits for $t^*$ then
        Early terminate this query
    else
        Run query until completion (or system failure timeout)
    end if
end for
```

use both time and utility for aggregation, neither of them directly optimizes for tail latency by exploiting tail latency slackness to obtain high utility for long queries.

Furthermore, none of the existing policies address multi-level aggregation, which is common in practice, and is much more challenging than a single level. In comparison, our policy takes effective actions for Fast, Straggling and Long queries, so we name it FSL. FSL exploits time and utility, and addresses single and multiple aggregation levels.

4. SINGLE-LEVEL AGGREGATION

This section presents the design and analysis of our policy FSL (for fast, straggling and long queries) for a single aggregator. FSL decides when to terminate a query by combining both online processing and offline analysis. For online processing, FSL jointly considers the time and utility of a query. In particular, we define a time threshold t^* and a utility threshold u^*. Based on whether a query is completed by t^*, and its utility at time t^*, we categorize the running query as fast, straggling or long, and take actions accordingly. The offline processing calculates the optimal values of t^* and u^* according to the workload information, utility target, and type of tail latency we seek to minimize. As response time distributions of interactive services vary slowly over time [20, 24], we trigger offline processing periodically to update the parameters for online processing. The latency can also be monitored online to trigger offline analysis whenever needed. To adapt to varying system load, we can construct a table that maps each load (e.g., query arrival rate) to its time and utility threshold values through offline analysis of the corresponding query logs; online processing simply applies the proper parameters by monitoring the load. FSL is *not* another heuristic: We derive an offline optimal algorithm with complete future information in Section 4.3 and prove that the online algorithm FSL is optimal, performing as well as offline optimal without knowing future responses of queries.

4.1 Online Processing

Online processing of FSL takes two runtime properties of a query — time and utility — as inputs, and it decides when to terminate query under execution. In particular, as described in Algorithm 1, if the aggregator does not receive all query responses after the query has been sent to the ISNs for time t^*, FSL checks the progress of the query: if the utility of the current query is higher than or equal to u^*, then terminate the query at time t^* (for reducing tail latency with a small utility loss); otherwise, let the query run till completion (to avoid significant utility loss).

Intuitively, FSL uses t^* and u^* to differentiate the three types of queries and take actions accordingly. (1) If a query completes by t^*, the query is fast: FSL executes fast queries till completion. (2) If a query has received most of the responses by t^*, the remaining utility of the query is marginal and the query is straggling: FSL early terminates straggling queries. (3) If responses are only received from no or a few ISNs at time t^*, the query is long: FSL executes long queries till completion, avoiding a significant utility degradation. Moreover, to optimize a tail latency target, e.g., 95%, the longest 5% of queries have no effect. Thus, by carefully choosing the values of t^* and u^* through offline analysis (as shown in Section 4.2), FSL allows long queries to complete, avoiding utility degradation without affecting the desired target tail latency.

In practice, system failure timeouts are set at the aggregators to prevent them from waiting forever in the case of server/network failures. Upon this timeout, FSL returns the collected results, just like all other aggregation policies. In addition, when there is only a tail utility constraint without average utility constraints, FSL can return the results of long queries after reaching the target tail utility rather than query completion.

4.2 Offline Processing

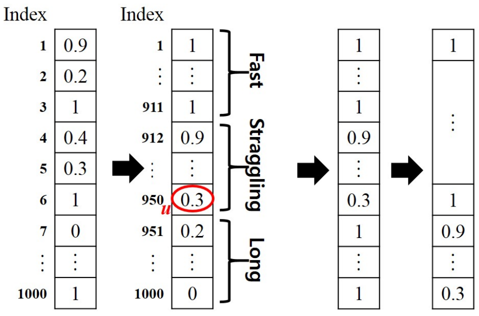
Offline processing includes a search algorithm to find the optimal time and utility thresholds. Algorithm 2 presents its formal description.

Inputs and outputs. FSL takes the workload data, utility constraints and tail latency objective as inputs. The workload data includes receiving time of each ISN responses for every query in the training set, which is the duration from aggregator sending the query to receiving its response. The output of FSL is the optimal time and utility threshold (t^*, q^*) that minimizes the k-th percentile latency subject to all utility constraints.

High-level description. The search algorithm iterates over all possible values of time thresholds. For each time threshold t, it calculates the optimal value of utility threshold u to maximize query utility (average and/or tail) while producing k-th percentile latency of t. Then among all the t values, we select the smallest one which meets the utility constraints, producing the optimal k-th latency subject to utility constraints. The corresponding (t, q) values are the optimal time and utility thresholds (t^*, q^*).

Step 1. Given a step size δ, there are a set of candidate time thresholds $\{\delta, 2\delta, ..., m\delta\}$, where m is decided by the maximum latency t_{max} of all responses. The tail latency of FSL can exceed the (theoretically) true optimal by at most δ. By reducing δ, our result approaches arbitrarily close to the theoretically optimal, while the search complexity increases, trading off algorithm complexity for accuracy. For web search, δ =1 ms is sufficiently accurate.

Step 2. For any possible time threshold t, we compute the corresponding utility of each query. We sort the queries according to a descending order of the utility values. Figure 3(a) shows an example for a set of queries with sorted utility. The queries at the top of the list are fast queries: they complete before time threshold t and each gets a full utility of 1. At the very bottom are long queries, whose utility is affected significantly by early termination at t. Between fast and long queries are straggling queries, which, if early terminated, incur smaller utility loss than long queries



(a) Construct B (b) Construct C

Figure 3: Example of the matrix construction of B and C at time threshold t, where $k = 95$ and $n = 1000$.

would. Such a list of utility values for a given time threshold t corresponds to column t/δ of matrix B in Algorithm 2.

A research question here is how we differentiate straggling versus long queries, which also relates to how we compute the utility threshold u. Given a latency objective k, (e.g., k=95 for optimizing 95-th latency), we know that we can disregard the latency of less than $(100 - k) = 5$ percent of queries (i.e., classify these queries as long queries). We choose the long queries to be those with the highest loss in utility if we early terminate them, i.e., the bottom $(100 - k)$ percent of queries in the list. The remaining queries are straggling queries to be early terminated, losing less utility in total. Here, the utility threshold u is equal to the utility value of the $(100 - k)$-th percent query from the bottom. As shown in Figure 3(a), suppose that there are $n = 1000$ queries in total, u corresponds to the utility value of the $n * k/100 = 950$-th query from the top. The beauty of differentiating straggling and long queries in this way is that (1) it allows us to obtain the highest utility gain for any given latency target, and (2) during online processing, it does not require future response times of ISNs to differentiate straggling versus long queries.

Step 3. We update the resulting utilities of queries according to online processing: the utility values of fast and straggling queries remain unchanged, while the utility values of the long queries become 1 since these queries will run till completion. An example is shown in Figure 3(b).

In the case where long queries are returned due to system failure timeout, their utility values are set to the utility of the query at the timeout instead of a full utility of 1.

Step 4. We find the smallest termination time threshold t value that meets all quality SLAs.

Time complexity. Let r represent the number of ISNs, m the number of time threshold candidates, and n the number of queries. Step 1 takes time $n * r$ to compute t_{max}. Step 2 takes $O(rnm)$ for the matrix construction of A, and takes $O(n\log(n)m)$ for the sorting to compute B. Step 3 take $O(n\log(n)m)$ for sorting, and other operations have lower asymptotical order. Step 4 performs a search over all entries of the table with cost of $O(nm)$. Therefore, the overall time complexity of the offline processing of FSL is $O((rn + n\log(n))(t_{max}/\delta)$. The offline processing is an efficient polynomial time algorithm for any given value of δ. For 10,000 queries, 1,000 ISNs, $t_{max} = 350$ ms, and $\delta = 1$ ms, we compute this in 43 seconds on a desktop PC.

Algorithm 2 FSL: Offline Processing

```
Input:
  • time step size $\delta$ and log data $X = [X_{i,r}]$ for $n$ queries: $X_{i,r}$
    is the response time for $r$-th ISN and $i$- th query
  • constraints: average utility $u_A$, $k_T$-percentile utility $u_T$
  • optimization objective: latency percentile $k$
Procedure:
  Step 1. $t_{max} = \max_{i,r} X_{ir}$; $m = \lceil \frac{t_{max}}{\delta} \rceil$
  Step 2. Construct $A = [A_{i,j}] \in \mathbb{R}^{n\times m}$: $A_{i,j}$ represents the util-
    ity of $i$-th query when terminating it at time $j \times \delta$; construct
    $B = [B_{i,j}] \in \mathbb{R}^{n\times m}$: The $j$-th column of $B$ is sorted results
    (in descending order of the utility) of $j$-th column of $A$.
  Step 3. Construct $C = [C_{i,j}] \in \mathbb{R}^{n\times m}$: set $C_{i,j} = B_{i,j}$ for
    $i \leq \lfloor \frac{k\cdot n}{100} \rfloor$ and $C_{i,j} = 1$ otherwise; construct $C' \in \mathbb{R}^{n\times m}$ by
    resorting each column of $C$ in descending order of the utility.
  Step 4. Find time threshold.
// For the $k_T$-percentile utility constraint $u_T$
    Set $t_T = j_T \times \delta$, where $j_T = \min\{j | C_{\lfloor \frac{k_T n}{100} \rfloor, j} \geq u_T\}$
// For the average utility constraint $u_A$
    Set $t_A = j_A \times \delta$, where $j_A = \min\{j | \frac{\sum_i C'_{i,j}}{n} \geq u_A\}$
Output: time threshold $t^* = \max\{t_T, t_A\}$ and utility threshold
    $u^* = B_{\lceil \frac{kn}{100} \rceil, t^*/\delta}$
```

4.3 Optimality Proof

FSL is an online policy without future information: it does not know when the remaining responses will be returned by ISNs. In Theorem 1, we compare FSL with an optimal offline policy, which assumes complete information. We show that FSL performs as well as the optimal offline policy — FSL is an optimal online policy achieving the minimum tail latency subject to utility constraints.

We first define an *optimal offline policy*, which decides termination time of each query by assuming full knowledge of the query as well as the entire query set. Each query may have its own termination time. A simple brute-force algorithm to compute such an optimal solution is as follows. For each query i, its candidate termination time t_i is one of the response times of all ISNs, and hence there are r choices (assuming that there are r ISN). For a set of n queries, the search space is r^n. The brute-force algorithm finds the set of $\{t_i\}$, which produces the smallest tail latency while satisfying all utility constraints.

THEOREM 1. *For a cluster with a single aggregator, FSL achieves an arbitrarily close-to-minimum tail latency while meeting average and tail utility constraints.*

PROOF. We show the equivalency of FSL and the optimal offline policy.

First, we perform a transformation from the optimal solution to FSL. Suppose that the offline optimal finishes query i at t_i. Without loss of generality, we assume that $t_1 \leq t_2 \leq \cdots \leq t_n$ among a total of n queries. Then, k-th tail latency of the optimal offline policy is $t' = t_{(k*n/100)}$, and we have two observations.

- For queries whose finish time is smaller than t', finishing them at t' does not change the k-th tail latency.
- For queries whose finish time is bigger than t', waiting for all ISNs does not change the k-th tail latency.

This observation indicates, in the optimal offline policy, each query can have three possible cases: finish before t' (i.e., fast query), finish exactly at t' (i.e., straggling query) or wait until receiving all ISNs' results (i.e., long query). Hence, this is the same as our policy. Algorithm 2 of FSL is a constructive proof of finding the minimum t', which is equivalent to the minimum time threshold t^* (with a deviation smaller than the step size δ). Thus, when the search time step size $\delta \to 0$, the tail latency produced by FSL can be made arbitrarily close to offline optimal. □

5. MULTILEVEL AGGREGATION

This section extends FSL for multiple levels of aggregators, which are common in practice but are much more challenging than a single level. First, the aggregators' decisions on different levels are coupled and must be coordinated to reduce tail latency subject to quality SLAs. Second, the number of communication messages between different levels of aggregators must be small, e.g., an MLA cannot simply forward the results to TLA whenever it receives an ISN response, which further magnifies the design challenge.

At runtime, an MLA may or may not know the messaging time between itself and its higher-level aggregators in advance. Here the messaging time primarily includes network latency plus some lightly-weighted processing such as reranking and removing near duplicates. We consider three different scenarios of the messaging time: (a) known messaging time, (b) unknown but varying in a small bounded range, (c) general case of unknown messaging time with potentially large variations. In this section, we first focus on two-level aggregation, and generalize our results to arbitrary levels in Section 5.4.

5.1 Known Messaging Times

Inspired by FSL and its analysis in Theorem 1, we develop an optimal online policy for two (or multiple) levels of aggregators when messaging times between aggregators are known. We call this FSL-K, where 'K' stands for Known messaging time. The key idea of FSL-K is as follows: A TLA returns the results of each query in one of the three ways: (1) return full results before/at time threshold t^*; (2) based on the query utility progress at t^*, return partial results at a time threshold t^*, or (3) wait until receiving all responses from MLAs. We show that FSL-K at TLA is optimal, performing as well as an optimal offline policy. Once the top-level aggregation policy is known, we derive the time threshold for MLA by considering both t^* and the messaging time between TLA and MLAs.

Online Processing of FSL-K. Online processing of FSL-K takes as inputs time threshold t^* and utility threshold q^*, as decided by offline analysis.

TLA policy. For each query, if TLA receives responses from all MLAs and their ISNs prior to the time threshold t^*, this query is fast and TLA returns complete results. Otherwise, TLA has two choices: (1) If the utility of the current query is no less than q^*, then return the query results at t^*; Or (2) Complete the query and collect the results from all MLAs and their ISNs. In practice, system failure timeouts are set at TLA/MLA to stop waiting in case of server/network failure.

MLA policy. Suppose the messaging time between an MLA i and its TLA is d_i. For each query, MLA i responds to TLA in one of the two cases.

- If MLA i received all ISN responses before time $t^* - d_i$, it sends back the responses to TLA. This is a common case, where an MLA only sends one message to TLA.
- Otherwise, the MLA sends its results to TLA twice, because it cannot tell if a query is straggling or long (without global information from TLA). First, at time $t^* - d_i$, it sends back all the responses it received so far to TLA, in order to catch the checkpoint time of TLA at t^*. Second, the MLA sends the additional results to TLA after it received the responses from all of its ISNs, which is to obtain full results for the long queries that do not finish prior to $t^* - d_i$.

Algorithm 3 FSL-K: Offline processing

```
Input:
  • time step size $\delta$ and log data $X = [X_{i,r}]$ for $n$ queries: $X_{i,r}$
    is the response time for $r$-th ISN and $i$- th query
  • log data $Y = [Y_{i,q}]$: $Y_{i,q}$ is the messaging time from $q$-th MLA
    to TLA for $i$-th query
  • utility constraints and latency percentile $k$ from inputs of
    Algorithm 2
Procedure:
  Step 1. $t_{max} = \max_{i,r} X_{i,r} + \max_{i,q} Y_{i,q}$; $m = \lceil \frac{t_{max}}{\delta} \rceil$
  Step 2. Construct $A = [A_{i,j}] \in \mathbb{R}^{n \times m}$: $A_{i,j}$ represents the
    utility of $i$-th query when its finish time at TLA is $j \times \delta$;
    construct $B = [B_{i,j}] \in \mathbb{R}^{n \times m}$: The $j$-th column of $B$ is
    sorted results (in descending order) of $j$-th column of $A$.
// To compute $A_{ij}$, we first compute the utility of each $q$-th MLA
    at $(j \times \delta - Y_{i,q})$, and compute the sum of them.
  Step 3. Do Steps 3 & 4 in Algorithm 2
Output: time threshold $t^* = \max\{t_T, t_A\}$ and utility threshold
    $u^* = B_{\lceil \frac{kn}{100} \rceil, t^*/\delta}$
```

Alternatively, a TLA can inform its MLAs of not sending the second message when it decides to early return. The benefits of this alternative are arguable since (1) This does not decrease the overall number of messages (i.e., reducing one MLA message for straggling queries by adding a TLA message); (2) MLAs still send two messages for the long queries; (3) Search engines usually do not abort execution of straggling queries at ISNs (Section 2).

FSL-K is efficient at runtime in terms of both computation and communication. Its computation cost at both TLA and MLAs is negligible, requiring a simple counting on the number of responses. For communication cost, each MLA sends at most two messages to TLA. The average number of messages is even smaller: in most of the cases, queries are fast, requiring one message per MLA only; even if queries are straggling or long, only the MLAs with slow ISNs send two messages. For example, MLAs send 2 messages only 5.58% of the time on our evaluation using commercial advertisement search workloads (Section 6.2).

Offline Processing of FSL-K. Algorithm 3 presents offline processing of FSL-K, which has a similar flow as FSL with two differences. (1) FSL-K requires messaging times from MLAs to TLA as additional inputs. (2) At Step 2, to compute the utility of a query i at a time threshold t at TLA, we sum over its utilities obtained at the q-th MLA at time threshold $t - d_q$. Its time complexity is still $O((rn + n\log(n))(t_{max}/\delta))$.

Optimality Proof. Theorem 2 shows the optimality of FSL-K. Its proof is analogous to Theorem 1, which we skip for interest of space.

THEOREM 2. *For two-level aggregation with known messaging time, FSL-K achieves an arbitrarily close-to-minimum tail latency while meeting utility constraints.*

5.2 Bounded Messaging Times

We extend FSL-K to FSL-B for the case when messaging times between aggregators are unknown but bounded within a small range, where 'B' of FSL-B stands for Bounded messaging time. Suppose that the messaging time from the q-th MLA to TLA for the i-th query is bounded in the range $[Y_{i,q}^{-}, Y_{i,q}^{+}]$. To meet the utility constraints, FSL-B considers the (worst-case) upper bound $Y_{i,q}^{+}$ as messaging time, and then uses the same offline and online processing as FSL-K. *Thus, FSL-B is guaranteed to satisfy all utility constraints, and the resulting tail latency is at most* $\max_{i,q}(Y_{i,q}^{+} - Y_{i,q}^{-})$ *higher than theoretically optimal in the worst case.*

5.3 Unknown Messaging Times

Now, we consider a general case where the messaging times between aggregators are unknown with potentially large variations. It models the cases such as with poor network condition or with aggregators sharing processing with other workloads. In this case, even if we find the optimal time threshold t^* at TLA, it is still difficult for MLAs to decide when to respond to TLA to meet the checkpoint t^*. While it is challenging to provide a performance guarantee, we develop an effective heuristic policy, which shows very good empirical performance (Section 6). We call this FSL-U, where 'U' stands for Unknown messaging time.

FSL-U defines three parameters: besides time and utility thresholds t^* and q^*, respectively, FSL-U includes the third parameter, i.e., MLA threshold t_m^*, which indicates the time threshold when MLAs shall return their results to TLA. Note that we choose to use the same t_m^* for all MLAs, because without knowing how much it may take each MLA to respond, there is no reason to differentiate MLAs in terms of the time thresholds for returning their responses. For search engine in practice, each MLA is often associated with the same number of ISNs and running on the same hardware.

Online Processing of FSL-U. TLA still takes t^* and u^* as inputs, working in the same way as FSL-K. MLA also works similarly as FSL-K, but it applies an additional MLA time threshold of t_m^*, obtained by offline processing. Under FSL-U, each type of queries behaves as follows.

- Fast queries: TLA receives complete results before t^*. To achieve this, every MLA must have received all its ISNs' responses before t_m^*, and all messaging times between MLAs and TLA are smaller than $t^* - t_m^*$.
- Straggling queries: The utility at TLA is not equal to 1, but bigger than or equal to u^* at t^*. This can happen if any one of the above conditions for fast queries does not satisfy.
- Long queries: The utility at TLA is smaller than u^* at t^*. In this case, TLA waits for the responses from all MLAs and their associated ISNs (within timeouts).

Offline Processing of FSL-U. Algorithm 4 uses both ISN latencies and inter-aggregator messaging times to search for the optimal values of the three parameters. Compared with FSL-K, the main difference is that FSL-U needs to search over one more dimension — the time threshold t_m^* for MLA (Step 2 of Algorithm 4). The time complexity of Algorithm 4 is $O((rn + n\log(n))(t_{max}/\delta)^2)$, which is higher than FSL-K by a multiplicative factor $O(t_{max}/\delta)$ due to the enlarged search space of t_m^*.

Note that, although offline processing of FSL-U still finds the best aggregation policy parameters (t^*, u^*, t_m^*) based on log data to minimize the tail latency, we cannot claim that FSL-U is optimal when applied online. The offline optimal solution may give different MLA time threshold values to each MLA, because it has the offline information on the inter-aggregator messaging times for each query while this information cannot be obtained online (unlike FSL-K where the messaging time is assumed to be known online). However, our empirical results in Section 6 show FSL-U performs close to offline optimal.

Algorithm 4 FSL-U: Offline processing

```
Input: take the same input as Algorithm 3
Procedure:
  Step 1. Set $t_{max}^m = \max_{i,r} X_{i,r}$ and $t_{max} = t_{max}^m + \max_{i,q} Y_{i,q}$
  Step 2. Given each candidate $t_m^*$, find $t^*$ and $u^*$ for TLA
   Set $v = 1$
// Denote the number of MLAs as p
for $v \leq \lceil t_{max}^m/\delta \rceil$ do
    Construct $D = [D_{i,q}] \in \mathbb{R}^{n \times p}$: $D_{i,q}$ is the utility of $q$-th mid-level aggregator for $i$-th query if we finish it at $v \times \delta$.
    Construct $E = [E_{i,q}] \in \mathbb{R}^{n \times p}$: $E_{i,q}$ is the finish time of the $q$-th mid-level aggregator for $i$-th query
    // $E_{i,q} = \min\{v'\delta, v\delta\}$, where $v'\delta$ is the time the q-th aggregator receives all the responses
    Construct $A = [A_{i,j}] \in \mathbb{R}^{n \times m'}$, where $m' = \lceil \frac{t_{max}}{\delta} \rceil$: $A_{i,j}$ represents the utility of $i$-th query when its finish time at the top-level aggregator is $j \times \delta$
    // For each query i, the utility at $j \times \delta$ is determined by adding up all entries of $D_{i,q}$ whose corresponding value of $E_{i,q} + Y_{i,q}$ (i.e., the receiving time of q-th mid-level aggregator's response at the top-level) is smaller than $j \times \delta$
    Construct $B = [B_{i,j}] \in \mathbb{R}^{n \times m}$: The $j$-th column of $B$ is sorted results (in descending order of the utility) of $j$-th column of $A$.
    Do Steps 3 & 4 in Algorithm 2.
    Set $t_v^* = \max\{t_T, t_A\}$ and $u_v^* = B_{\lceil \frac{kn}{100} \rceil, t_v^*/\delta}$
    $v = v + 1$
end for
  Step 3. Find $v^* = \arg\min_v \{t_v^*\}$ and set $(t^*, u^*, t_m^*) = (t_v^*, u_v^*, v^* \times \delta)$
Output: time threshold at the top-level $t^*$, utility threshold $u^*$, and time threshold at the mid-level $t_m^*$
```

5.4 Generalization to Multilevel Aggregation

We can easily extend FSL-K, FSL-B, and FSL-U to more than two levels of aggregators. Here, we still call the top-level aggregator as TLA, but there may be more than one level of MLAs. For FSL-K (known messaging time), TLA still has one time threshold t^* and utility threshold q^*. All levels of MLAs send their responses in advance such that TLA receives the responses prior to or at t^*. The optimality of FSL-K still holds, and the time complexity is $O((srn + n\log(n))(t_{max}/\delta))$ with s levels of aggregators. The extension of FSL-B to multilevel aggregation is also straightforward, following that of FSL-K. The performance bound of FSL-B becomes the summation of the upper bounds on messaging times over all levels. To extend FSL-U to multiple levels, we use a different time threshold t_m^* for each level of MLA. The resulting complexity of offline processing increases with the increased level of MLAs, due to enlarged search space, the time complexity is $O((rn+n\log(n))(t_{max}/\delta)^s)$, since each level of MLA has a search complexity of $O(t_{max}/\delta)$.

6. EVALUATION

We evaluate FSL for single-level aggregation by comparing it to alternative policies using a production trace of a 44-ISN cluster from a commercial web search engine, and we use several latency objectives and utility constraints. We also study several distributions forming synthetic workloads. For two-level aggregation, we create eight aggregation policies and compare FSL-K and FSL-U to them using a trace from a two-level commercial advertisement engine, in addition to a synthetic workload to model a large system with 1936 ISNs.

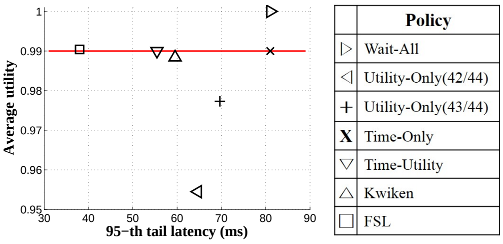


Figure 4: Tail latency comparison for single aggregator with Bing production traces.

6.1 Single Aggregator

Bing Production traces. We use a query log from a production cluster of Microsoft Bing search engine. The log records the latency of 44 ISNs over 66,922 queries. These queries missed the result cache and were sent to ISNs for processing. We use a subset of 10,000 queries for training, and the remaining 56,922 queries for evaluation. As Bing search does not abort processing of late responses at ISNs (Section 2), the recorded query execution times of ISNs remain unaffected by aggregation policy and hence we replay in our simulation to compare the alternative aggregation policies. We set the system failure timeout as 500 ms. A response that does not arrive by 500 ms is ignored by all policies.

Evaluation with production traces. Our target is to minimize the 95-th tail latency under the constraint that average utility is greater than or equal to 0.99. We implement all policies in Table 1 and compute the best parameters for each policy using the same training dataset. We show their results in Figure 4 that depicts the 95-th tail latency on the x-axis and the average utility on the y-axis. Wait-All is the baseline with $utility = 1$ and it has the highest 95-th tail latency. For Utility-Only we use two configurations because when the utility threshold is 0.99, the aggregator must wait for responses from all ISNs because $(43/44) < 0.99$. We, therefore, set utility threshold to $(43/44) = 0.977$ and to $(42/44) = 0.955$, and report both settings in the figure. FSL provides the shortest tail latency compared to all other schemes: (FSL: 38 ms, Time-Utility: 65 ms, and Kwiken: 59 ms). In particular, FSL reduces the tail latency by 53% over Wait-All and by 36% over the best alternative. All policies have similar average latencies (between 15 and 17 ms) because the average response latency is determined mainly by fast queries which constitute majority queries.

To see why FSL achieves lower tail latency, we study the utility of all queries according to their length (measured as the maximum latency of all ISN responses) quantized into bins of 20 ms as depicted in Figure 5. The first point on each curve shows the average utility for the queries with length $\in [0, 20]$, and second point for $\in [20, 40]$. Notice that the first two points represent over 92% of queries. We do not show a curve for Utility-Only because it is a horizontal line across all queries regardless of their length. Time-only terminates all queries after a threshold and we see a similar behaviour for Time-Utility: All relatively slow queries are terminated early regardless of being long or straggling. Next, Kwiken terminates queries starting from the second point which impacts many fast queries, and again, all relatively slow queries are terminated early. In contrast, FSL exploits the (5%) latency slackness to obtain high utility for long queries, which allows it to terminate straggling queries more aggressively, reducing tail latency further while meeting utility targets.

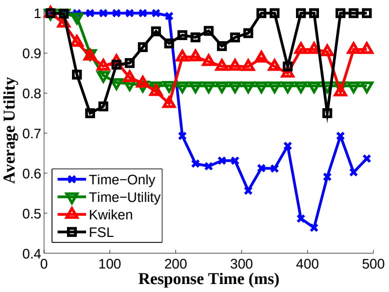


Figure 5: Average utility for quantized query length (single aggregator & Bing production traces).

Sensitivity study. Using the production traces, we study the two tail latency optimization targets (95-th and 99-th) under several utility constraints as depicted in Table 2. The values of the first row show the 95-th tail latency corresponding to data in Figure 4. The stricter the utility constraint, the fewer the chances to reduce the tail latency (compared with Wait-All) while meeting the constraint. This behaviour applies both for average and tail utility constraints as well as across the two tail latency objectives. The last row shows that we can enforce two utility constraints, one on average and another on the 95-th tail utility. Under all configurations FSL reduces the tail latencies the most: It provides the minimum tail latency while meeting the utility constraints, consistently outperforming heuristic policies.

Synthetic datasets. We evaluate the policies using several synthetic datasets with average utility constraints (AVG ≥ 0.99) assuming a single aggregator with 44 ISNs. For each dataset, we generate 66,922 queries and use 10,000 as a training set and the remaining 56,922 for evaluation. We use two methods to generate synthetic datasets: (1) In one-phase sampling, query latency at each ISN is sampled independently from a distribution. (2) In two-phase sampling, we first sample the query mean from a distribution, and then sample its latency at each ISN from a second distribution parametrized by the query mean. While one-phase sampling generates latencies from all ISNs of all queries assuming independence, two-phase sampling models the scenarios where long (short) queries tend to have overall higher (lower) latencies from many of its ISN responses. These two sampling methods are also used in [20]. We report the average of Pearson's correlation coefficient (PCC) between ISNs for all pairs of ISNs, as well as the average of coefficient of variation (σ/μ) for all queries. As a reference, for the production traces: $PCC = 0.9920$ which is a high value showing significant correlation among ISN latencies for a particular query, and $CV = 0.1777$, a small value as workload is dominated by many fast queries.

Evaluation with synthetic datasets. Table 3 reports the results. The first two rows show one-phase sampling with log-normal and exponential distributions. PCC is small indicating little correlation among ISN latencies per query:

Table 2: Percentage of tail latency reduction over Wait-All policy for different utility constraints with Bing production traces and single aggregator. A higher value is better, i.e., bigger latency reduction.

Utility constraint	95-th tail latency reduction (%)				99-th tail latency reduction (%)			
	Time-Only	Time-Utility	Kwiken	FSL	Time-Only	Time-Utility	Kwiken	FSL
AVG $\geq$ 0.99	0	31.57	26.63	53.11	15.48	15.48	15.48	18.05
AVG $\geq$ 0.999	0	0	8.91	34.60	9.90	10.09	9.90	12.90
95-th tail $\geq$ 0.95	19.79	19.79	27.93	70.38	72.11	19.45	72.11	80.26
95-th tail $\geq$ 0.99	3.75	3.75	17.31	64.21	66.53	19.45	66.53	75.54
99-th tail $\geq$ 0.99	0	0	3.42	29.66	6.04	6.04	2.29	10.76
AVG $\geq$ 0.99&95-th tail $\geq$ 0.95	0	19.79	26.63	53.11	15.48	15.48	15.48	18.05

Table 3: 95-th tail latency reduction (%) of policies over Wait-All policy for single aggregator and synthetic distributions.

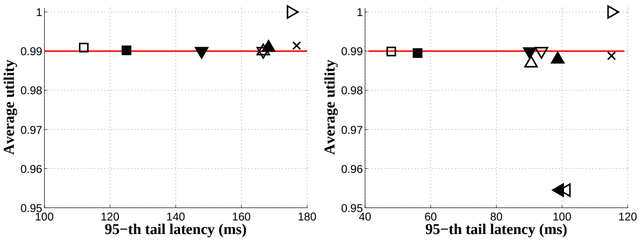
ISN latency (X) distribution	PCC	CV	Time-Only	Time-Utility	Kwiken	FSL
Log-normal distribution ($X_{i,r} \sim \ln\mathcal{N}(1,1)$)	0.0030	1.1574	50.28	50.28	50.28	53.83
Exponential distribution ($X_{i,r} \sim \exp(0.1)$)	0.0031	0.9793	30.31	31.79	30.31	34.76
$m_i \sim \exp(0.1), X_{i,r} \sim \ln\mathcal{N}(\log(m_i), \log(1+m_i)/5)$	0.4724	0.4205	39.27	49.05	39.27	60.21
$m_i \sim \exp(0.1), X_{i,r} \sim \ln\mathcal{N}(\log(m_i), \log(1+m_i)/10)$	0.8108	0.2035	12.60	29.47	22.17	41.73
$m_i \sim \exp(0.1), X_{i,r} \sim \ln\mathcal{N}(\log(m_i), \log(1+m_i)/100)$	0.9978	0.0200	0	3.20	3.92	12.57
$m_i \sim$ BoundedPareto($\alpha = 0.5, lo = 1, hi = 300$), $X_{i,r} \sim \ln\mathcal{N}(\log(m_i), \log(1+m_i)/100)$	0.9963	0.0213	4.51	6.06	4.81	25.36

each query has similar utility at any time point, which does not produce the straggling and long query behaviours. The policies Time-Only, Time-Utility, Kwiken reduce the tail latency compared to the baseline, and FSL provides bigger tail latency reduction. We observe similar behaviour under all the distributions we employed (not reported due to space).

Next, we study the two-phase sampling, which has higher correlation and is more realistic for search workloads. We use both Exponential and Bounded Pareto distributions for the first sampling phase. We use log-normal distribution for the second-phase sampling because sampling from normal distribution may result in negative values for latency. FSL clearly outperforms the other policies and the improvement of FSL over other polices increases as PCC increases.

6.2 Two-Level Aggregation

Symbol		▷	◁	◀	X	▽	▼	△	▲	□	■
Policy	MLA	Wait-All	Wait-All	Utility-Only	Time-Only	Time-Utility	Wait-All	Kwiken	Wait-All	FSL-K	FSL-U
	TLA	Wait-All	Utility-Only	Utility-Only	Time-Only	Wait-All	Time-Utility	Wait-All	Kwiken		



(a) Advertisement Engine (b) Synthetic

Figure 6: Policy comparison for cluster with two-level aggregators on (1) production traces of advertisement engine and (2) synthetic datasets.

We evaluate two-level aggregation using a production trace of Bing Advertisement Engine as well as a synthetic dataset modeling a large-scale system with a few thousand ISNs.

Advertisement engine production traces. We use a query log from a production cluster with two-level aggregation of Bing advertisement engine. The log records the latencies of 1 TLA, 16 MLAs and 4 ISNs per MLA (total ISNs is 64) over 16,311 queries. We use a subset of 10,000 queries for training and the remaining 6,311 for evaluation.

There is no policy that targets two-level aggregation in prior work. So we develop the following eight policies for the MLA&TLA policies. (1) Wait-All & Wait-All (baseline), (2) Wait-All & Utility-Only (0.95), (3) Utility-Only (0.95) & Utility-Only (0.95), (4) Time-Only & Time-Only, (5) Time-Utility & Wait-All, (6) Wait-All & Time-Utility,(7) Kwiken & Wait-All, (8) Wait-All & Kwiken. We make the MLA delay known to FSL-K (which obtains the optimal solution) and unknown to FSL-U (which is heuristic) and study how they compare to each other as well as to other polices.

Figure 6(a) compares the 95-th tail latencies for average utility target of 0.99. FSL-K offers the shortest tail latency (112 ms). FSL-U is close (125 ms), within 12% of the optimal, lower by 30% over the baseline and lower by 15% over the best alternative policy. Notice that (Wait-All & Utility-Only) and (Utility-Only & Utility-Only) produce same tail latency and average utility as baseline and therefore are not plotted in the figure.

Evaluation with synthetic datasets. We also evaluate FSL using a synthetic dataset modelling a large-scale system with two levels of aggregation containing 1 TLA, 44 MLA and 44 ISNs per MLA (total ISNs is 1,936). The synthetic dataset has the same mean and variance as Bing queries. In addition, the MLA messaging delay is sampled from an exponential distribution with parameter $\mu = 7.5$ ms, which is the measured average messaging time between aggregators. Figure 6(b) shows that FSL-K provides the shortest tail latency (49 ms), and FSL-U is close (54 ms) and is substantially better than the other polices; the tail latency of FSL-U is within 16% of the optimal, and lower by 51% over the baseline and by 38% over the best alternative policy.

Sensitivity study. Table 4 shows the impact of changing system parameters using the synthetic workload. Row 1 is the default settings as depicted in Figure 6(b). Each row reports the impact of one change: optimizing the 99-th tail latency (row 2), tightening the utility constraint $AVG \geq 0.999$ (row 3), changing the messaging time from MLAs to TLA to

Table 4: Tail Latency reduction over baseline policy with two-level aggregation using synthetic workloads.

Variable setting	Tail latency reduction of each policy (% over Wait-All & Wait-All (baseline))						
	Time-Only &Time-Only	Time-Utility &Wait-All	Wait-All & Time-Utility	Kwiken& Wait-All	Wait-All &Kwiken	FSL-K	FSL-U
Default setting	0	18.55	21.63	21.47	14.24	58.28	51.33
Minimize 99-th tail latency	15.73	9.07	12.62	9.08	12.54	60.06	19.32
AVG $\geq$ 0.999	0	10.41	0	13.71	3.68	41.77	31.34
Two-phase sampling for MLAs	0	21.38	20.73	22.44	20.12	55.33	49.20
# of MLAs = 11, ISNs/MLA = 176	0	24.50	13.47	28.28	8.46	57.38	49.39
# of MLAs = 22, ISNs/MLA = 88	0	20.65	17.41	24.69	9.83	57.88	50.87
# of MLAs = 44, ISNs/MLA = 44	0	18.55	21.63	21.47	14.24	58.28	51.33
# of MLAs = 88, ISNs/MLA = 22	0	15.41	25.73	18.42	17.93	58.67	52.64
# of MLAs = 176, ISNs/MLA = 11	0	14.97	30.81	17.22	24.31	59.50	54.43

follow a two-phase sampling instead of a one-phase sampling (row 4), changing the system layout while maintaining the same number of ISNs (row 5 to 8). For all settings, FSL-K and FSL-U consistently provide lower tail latencies. Note that FSL-K and FSL-U are less sensitive than the other policies, although the amount of their latency reduction slightly increases as the number of ISNs/MLA decreases.

7. RELATED WORK

Complementary latency reduction techniques on web search. ISNs of web search often employ dynamic pruning to early terminate the post-list processing of a query and to avoid scoring the postings for the documents that unlikely make the top-k retrieved set [3, 32]. Early termination is an example that web search trades completeness of query response for latency. Prior work [19] quantifies the relationship of quality and latency at ISNs, and trades quality for latency under heavy loads. Multicores [15, 22] and graphics processors [14] have also been used to parallelize query processing and reduce query latency. All the above techniques improve ISN latency, complementary to our study where we take ISN latency as inputs and focus on aggregation policy at aggregators.

Beyond ISN-level improvement, there are studies on reducing the latency for search queries across different system components, e.g., optimizing caching [6, 9, 10, 16, 26] and prefetching [23] to mitigate I/O costs, and improving network protocols between ISNs and aggregators [28]. These studies are also complementary to our work. Only queries that are not cached are sent to ISNs for processing. We take transmission time between ISNs and aggregators as inputs, working (orthogonally) for any transmission protocols.

Request reissue is a standard technique to reduce latency tail in distributed systems at the cost of increased resource usage [13, 20, 29, 33]. When web indices are replicated, reissuing a query at another cluster or server may improve latency by using responses that finishes first. Reissuing consumes extra resources to replicate indexes and to process a query more than once. Therefore, it is more preferable to have an effective aggregation policy to reduce latency and meet quality SLAs without reissuing many queries.

Aggregation policies. We discussed several existing aggregation policies and their performance in Sections 3 and 6, which we will not repeat. Broder and Mitzenmarcher [8] study a related problem on maximizing reward as a function of time and utility, but they assume the responses of ISNs are i.i.d (independent and identically distributed), which Section 6.1 shows that ISN responses are highly correlated in practice. Moreover, they do not consider multilevel aggregations. Finally, we note that aggregation problem also exists in other domains (e.g., aggregate data sources in wireless sensor networks for energy minimization [30]), which requires domain-specific techniques and differs from our work.

8. CONCLUSION

We develop an aggregation policy, combining data-driven offline analysis with online processing, to reduce tail latency of web search queries subject to search quality SLAs. We first focus on a single aggregator and prove the optimality of our policy. We then extend our policy for multilevel aggregation and prove its optimality when messaging times between aggregators are known. We also introduce an empirically-effective policy to address unknown messaging time. We conduct experiments using production logs from a commercial web search engine, a commercial advertisement engine and synthetic workloads. Compared with prior work, the proposed policy reduces tail latency by up to 40%.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). *SIGCOMM Comput. Commun. Rev.*, 41(4):–, Aug. 2010.

[2] I. S. Altingovde, R. Blanco, B. B. Cambazoglu, R. Ozcan, E. Sarigil, and O. Ulusoy. Characterizing web search queries that match very few or no results. In *CIKM*, 2012.

[3] V. N. Anh, O. de Kretser, and A. Moffat. Vector-space ranking with effective early termination. In *SIGIR*, 2001.

[4] I. Arapakis, X. Bai, and B. B. Cambazoglu. Impact of response latency on user behavior in web search. In *SIGIR*, 2014.

[5] C. S. Badue, R. Baeza-Yates, B. Ribeiro-Neto, A. Ziviani, and N. Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Inf. Process. Manage.*, 43(3), 2007.

[6] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR*, 2007.

[7] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[8] A. Broder and M. Mitzenmacher. Optimal plans for aggregation. In *PODC*, 2002.

[9] B. B. Cambazoglu, I. S. Altingovde, R. Ozcan, and Ö. Ulusoy. Cache-based query processing for search engines. *ACM Transactions on the Web*, 6(4):14, 2012.

[10] B. B. Cambazoglu, F. P. Junqueira, V. Plachouras, S. Banachowski, B. Cui, S. Lim, and B. Bridge. A refreshing perspective of search engine caching. In *WWW*, 2010.

[11] B. B. Cambazoglu, V. Plachouras, and R. Baeza-Yates. Quantifying performance and quality gains in distributed web search engines. In *SIGIR*, 2009.

[12] J. Dean. Challenges in building large-scale information retrieval systems (Invited talk). In *WSDM*, 2009.

[13] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[14] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance ir query processing. In *WWW*, 2009.

[15] E. Frachtenberg. Reducing query latencies in web search using fine-grained parallelism. In *WWW*, 2009.

[16] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *WWW*, 2009.

[17] J. Hamilton. The cost of latency. `http://perspectives.mvdirona.com/2009/10/31/TheCostOfLatency.aspx`, 2009.

[18] E. Hatcher and O. Gospodnetic. *Lucene in Action*. Manning Publications Co., 2004.

[19] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *SOCC*, 2012.

[20] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *SIGCOMM '13*, 2013.

[21] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner. Adaptive parallelism for web search. In *EuroSys*, 2013.

[22] M. Jeon, S. Kim, S.-W. Hwang, Y. He, A. L. Cox, and S. Rixner. Taming tail latencies in web search. In *SIGIR*, 2014.

[23] S. Jonassen, B. B. Cambazoglu, and F. Silvestri. Prefetching query results and its impact on search engines. In *SIGIR*, 2012.

[24] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *SIGMETRICS*, 2001.

[25] C. Macdonald, N. Tonellotto, and I. Ounis. Learning to predict response times for online query scheduling. In *SIGIR*, 2012.

[26] R. Ozcan, I. S. Altingovde, and Ö. Ulusoy. Cost-aware strategies for query result caching in web search engines. *ACM Transactions on the Web*, 5(2):9, 2011.

[27] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity*, 2009.

[28] G. Upadhyaya, V. S. Pai, and S. P. Midkiff. Expressing and exploiting concurrency in networked applications with aspen. In *PPoPP*, 2007.

[29] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker. More is less: Reducing latency via redundancy. In *Hot Topics in Networks*, 2012.

[30] Z. Ye, A. A. Abouzeid, and J. Ai. Optimal stochastic policies for distributed data aggregation in wireless sensor networks. *IEEE/ACM Trans. Netw.*, 17(5):1494–1507, Oct. 2009.

[31] J. Yi, F. Maghoul, and J. Pedersen. Deciphering mobile search patterns: A study of Yahoo! mobile search queries. In *WWW*, 2008.

[32] F. Zhang, S. Shi, H. Yan, and J.-R. Wen. Revisiting globally sorted indexes for efficient document retrieval. In *WSDM*, 2010.

[33] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *WWW*, 2008.

[34] S. Zhu, A. Potapova, M. Alabduljalil, X. Liu, and T. Yang. Clustering and load balancing optimization for redundant content removal. In *WWW*, 2012.