Servicing "mice"-streaming queries as a service

Bo Zong[1], Christos Gkantsidis[2], Milan Vojnovic[3]

Technical Report
MSR-TR-2013-124

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

[1]Work performed while an intern with Microsoft Research. Bo Zong is with the University of California, Santa Barbara, bzong@cs.ucsb.edu
[2]Christos Gkantsidis is with Microsoft Research, chrisgk@microsoft.com
[3]Milan Vojnovic is with Microsoft Research, milanv@microsoft.com

Abstract –

Processing streaming queries at scale has received a lot of attention recently. Systems, such as S4, Storm, Photon, MillWheel, and Kinesis, can efficiently execute ("elephant") queries that process streams with high rates, and typically scale-out the execution of each query. We envision a platform in the opposite side of the spectrum. Our queries process relatively slower streams, but the challenge is to handle very high density of queries per server. Such a system democratizes the access to a streaming platform, and enables "mice" queries, on behalf of users, to consume stream data, *e.g.*, news, weather, travel related information, and so on. A challenging problem for the design of such a system is how to place queries among available servers. On one hand, we expect balanced workload among servers. On the other hand, we expect queries are packed into servers such that the network traffic is minimized by reducing the chance of sending identical data streams to multiple servers. This trade-off makes it hard to effectively place queries into servers. When we consider queries subscribing to more than one source and query/server dynamics, this problem becomes even harder.

In this paper, we formalize the problem of placing "mice" queries into the servers of a streaming platform. We propose approximation algorithms and derive approximation bounds for the following cases (1) the offline case where queries are stable and known ahead of time, akin to an "oracle", and (2) the online case without departures and known query popularities. For the general online problem, we propose effective heuristic algorithms. An extensive set of experiments demonstrates that the proposed algorithms provide good performance in a wide-range of scenarios.

Categories and Subject Descriptors

H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software—*Distributed Systems*; H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Algorithms, Experimentation

Keywords

streaming systems, complex event processing, query placement, network stream optimizations

1 Introduction

Complex Event Processing (CEP) [20] has been important for processing streaming data in financial, monitoring, and recently data analytics applications. Platforms such as S4 [33, 23], Storm [28], Photon [4], MillWheel [1] and Amazon's Kinesis [3]. enable *scalable* data analytics over data streams, where both the stream types and the processing requirements are very flexible. However, those platforms typically target applications that ingest high data rates, and need to *scale-out* the execution of the queries; we call them big ("elephant") queries. We envision a similar type of event processing platforms, but where the intended use is to execute, on behalf of users, queries that ingest relatively slower data streams; we call those small ("mice") queries. For example, a user may want to process streams related to traffic, weather, and news updates, but, since the user typically cannot manage the required infrastructure herself, she wishes to offload the execution of the query to a cloud service (and be notified only when there are updates). A cloud platform that provides such services needs to achieve large density of queries to servers to be scalable and cost-effective.

The query execution imposes two types of costs on the platform: (a) network overheads to deliver the streams from their sources to the servers, and (b) processing overheads on the servers for executing the logic related to the query.

This paper focuses on the cost of delivering the network stream inside the platform. The platform may need to download the streams from remote network servers, but we shall assume an efficient mechanism for those downloads. As is common in current cloud services, the query execution service may run on top of a generic compute platform, such as Amazon's EC2 [2], Azure Compute [32], or Rackspace [26]. Hence, we cannot assume an efficient distribution of the network streams inside the platform, *e.g.*, by using multicasting. Therefore, when a server executes one or more queries that read from a particular source, then the platform creates an internal network stream to deliver the stream to that server, and consumes network resources proportional to the rate of the stream. However, we do assume that a stream is delivered at most once per server.

We also assume flexibility in the processing of the stream data; users may express their processing requirements using arbitrary "user defined functions" (UDFs). Hence, the queries impose a non trivial processing overhead on the servers.

Hence, to provide a scalable and cost-effective query hosting service, the platform needs to reduce network and processing overheads. Intuitively we anticipate that many queries will use the same input streams. For example, many users may be interested in traffic updates from the same city, or weather updates for the same region. Hence, it is be beneficial to co-locate those queries to the same server, and transport the relevant network streams once. However, the servers have limited processing capacity, and hence the assignment needs to also balance the server load.

Three practical requirements further complicate the query assignment problem. First, the queries should be able to subscribe to more than one stream. This is required, for example, to support joins, and it is a common feature in many CEP systems. Second, the assignment of queries to servers should be semi-permanent. That means that the platform should avoid moving queries between servers, for example to reduce overheads, as this requires moving (query) state while guaranteeing that the query does not miss any stream updates. Obviously, queries will be re-assigned when their server fails, but such events should be the exception. Hence, the platform much make a good decision about the assignment of a query to a server, when the query arrives. Third, we expect churn both in the queries (queries have limited lifetime) and in servers (due to server failures and re-cycles). The queries arrive and depart dynamically, and the assignment of queries to servers should be robust to query and server dynamics.

In this paper, we propose and study the problem of assigning streaming queries to servers, under the requirements and assumptions described above. We use both analysis and simulations to understand the complexity of the problem and to design efficient algorithms for assigning queries to servers. In summary, the contributions of this work are as follows:

- Formulate the problem of assigning streaming queries to servers (Section 2), in the context of an online platform that hosts queries on behalf of the users as a service.
- Show that the problem of reducing network load while balancing server load is NP complete (Section 3), and provide approximation bounds (Section 4).
- We propose and study offline (Section 4) and online (Section 5) heuristics for the problem. Offline heuristics assume an oracle than knows ahead of time all queries. We use the offline heuristics to reason about the performance of the assignment process, and to draw inspiration for the online heuristics.

• Using analysis and simulation (Section 6), we identify the LeastCost online heuristic that gives the best performance, even under query and server churn, and is often up to four times better than (naïve) random assignment.

2 Problem Definition

A query streaming platform contains three key components: *sources*, *queries*, and *servers*.

Source. Sources provide input data streams to queries. Each source publishes new data at some rate. Let $S = \{s_1, s_2, \ldots, s_m\}$ be the set of sources, and let $w(s)$ be the traffic rate of source $s \in S$. The total traffic rate of a set of sources $S' \subseteq S$ is $w(S') = \sum_{s \in S'} w(s)$.

Query. Queries process data streams that originate from sources. Each query processes a set of input streams from one or more sources. Define $Q = \{q_1, q_2, \ldots, q_n\}$ to be the set of queries, and let $S_q \subseteq S$ be the set of sources required by query $q \in Q$.

We also use the concept *query type* that we define as follows. Define $T \subseteq 2^S$ to be a set of source combinations such that $S_q \in T$ for any $q \in Q$. For $t \in T$, let n_t be the number of queries that require a set of sources t, and any query that requires a set of sources t is also called a type t query. Note that the total number of queries n satisfies $n = \sum_{t \in T} n_t$.

Moreover, let $N(s)$ be the set of queries that require source $s \in S$ as one of the input streams, *i.e.*, $N(s) = \{q \in Q \mid s \in S_q\}$.

Server. Servers are containers that evaluate queries. We assume there are k identical servers in a query streaming platform.

The query assignment problem for a query streaming platform considers two criteria: (1) *minimizing network traffic* and (2) *balancing server load*.

Network traffic. We are interested in minimizing network traffic between sources and servers. When two queries require data streams from the same source, if the queries are hosted by two different servers, this source has to send duplicated streams to the servers, which results in extra network traffic. Since higher traffic implies higher operational cost, a streaming query system prefers a query assignment that minimizes the network traffic. If a server is assigned a set of queries $Q' \subseteq Q$ and this set contains at least one query that requires stream $s \in S$, then this contributes a traffic cost of $w(s)$. Hence, the total network traffic cost for a server hosting Q' is given by

$$f(Q') = \sum_{s \in S} w(s) \min\{|N(s) \cap Q'|, 1\}.$$

Balanced server load. We define feasible solutions as query assignments resulting in balanced server load. A query assignment is feasible meaning that the processing workload of a server is within its capacity. In practice, it is difficult to define servers' capability; therefore, we use balanced workload across servers to remove the need to know the server capacity limits. The implicit assumption is that in a practical system the operational point is such the system runs at a query load that allows for a feasible assignment, and in this case balancing the load across different servers is a natural objective.

Moreover, we assume that each query adds a fixed amount of workload to a server. A query streaming platform usually hosts simple queries that consist of a few primitive operations on data streams and take similar processing time. Therefore, we assume that queries involve identical amount of processing cost, say of unit size. To this end, the workload of a server corresponds to the number of queries assigned to this server.

Given a real number $\nu \geq 0$ as the allowed slackness for balancing the load across servers, a query assignment is specified by k disjoint subset of queries $Q_1, Q_2, \ldots, Q_k$. A query assignment is said to satisfy the ν-load balancing constraint, if the following holds

$$|Q_j| \leq (1+\nu)\frac{n}{k}, \text{ for all } j = 1, 2, \ldots, k,$$

where ν is also referred to as the *relative relaxation ratio*.

Let $\mathscr{P}(Q)$ be the universe of all possible k-partitions $(Q_1, Q_2, \ldots, Q_k)$ with respect to Q such that $Q_1 \cup Q_2 \cup \cdots \cup Q_k = Q$ and $Q_i \cap Q_j = \emptyset$ for any $i \neq j$. We formally define the problem Query Partition (QP) as follows.

DEFINITION 1 (QP). *Given as input a set of sources S with traffic rates w, a set of queries Q, k servers, and a real number $\nu \geq 0$, the goal is to find a query partition $(Q_1, Q_2, \ldots, Q_k) \in \mathscr{P}(Q)$ such that (1) the overall traffic cost $\sum_{j=1}^{k} f(Q_j)$ is minimized, and (2) $|Q_j| \leq (1+\nu)\frac{n}{k}, \forall j \in K$.*

3 Hardness and Benchmark

In this section, we discuss query partition (QP) problem and the infeasibility of a naïve solution to QP.

First, we show that QP is NP-complete, so we have to resort to approximation algorithms. Moreover, we demonstrate a naïve approximation ratio that any feasible algorithm holds.

Second, we discuss the infeasibility of a standard random assignment algorithm. In particular, we characterize the expected network traffic cost resulting from the random algorithm, and show that it can greatly increase network traffic.

3.1 NP Hardness

It is computationally hard to solve the QP problem and the following theorem characterizes its hardness.

THEOREM 1. *The query partition (*QP*) problem is NP-complete.*

PROOF. Consider the decision problem of QP: given sources S along with their traffic rate w, queries Q, k servers, a real number $\nu \geq 0$, and another real number $\gamma \geq 0$, does there exist a query partition $(Q_1, Q_2, \ldots, Q_k) \in \mathscr{P}(Q)$ such that

1. $\sum_{j=1}^{k} f(Q_j) \leq \gamma$, and
2. $|Q_j| \leq (1+\nu)\frac{n}{k}$ for each server $j = 1, 2, \ldots, k$.

The proof consists of two steps, showing that (1) QP's decision problem is NP-hard, and (2) QP's decision problem is NP.

First, we prove the NP-hardness. Consider a special case where (1) each query type depends on exactly one source, (2) $w(s) = 1$ for any $s \in S$, and (3) $\gamma = |S|$. In other words, we need to find a feasible solution such that queries of the same type are assigned to the same server. We can reduce an arbitrary instance of bin packing problem [14] to an instance of the special case: (1) we reduce an item to a query type, where the item's size is reduced to the number of queries for the corresponding type; (2) the number of bins is reduced to the number servers; and (3) the size of each bin is reduced to the upper limit for each server's load. Since bin packing problem is NP-hard, we conclude that QP's decision problem is NP-hard.

Second, given a solution to QP's decision problem, we can check whether it is feasible in polynomial time, so the QP problem is NP.

In all, QP is NP-complete. □

Moreover, the following holds for any feasible algorithms.

PROPOSITION 1. *Given an instance of* QP*, any feasible solution cannot be more than k times larger than the optimal solution, where k is the number of servers.*

3.2 Random Query Assignment Benchmark

A naive query assignment is to assign each query to a server picked uniformly at random and independently of other query assignments. This is a standard load balancing strategy that can be efficiently approximated by hash partitioning of query identifiers. This strategy is efficient with respect to balancing the number of queries across servers. Specifically, it guarantees maximum load of

$$\frac{n}{k}+O(\sqrt{\frac{n\log k}{k}})$$

with probability $o(1)$, for $n \gg k(\log k)^3$ ([24]). However, this policy can be grossly inefficient with respect to network traffic cost, which we show here analytically and in Section 6 experimentally.

PROPOSITION 2. *Consider a streaming query system with a set S of m sources and k servers such that d_s queries use the stream from source s as input. Under random query assignment to servers, the expected network traffic cost is*

$$\left(1-\frac{1}{m}\sum_{s\in S}\left(1-\frac{1}{k}\right)^{d_s}\right)km. \quad (1)$$

PROOF. Consider an arbitrary source s and an arbitrary server j. d_s queries use as input the stream from source s. Under random query assignment, there will be at least one such query in server j with probability $1-(1-1/k)^{d_s}$. Summing over all servers j gives the expected number of servers to which the stream of source s need to be transferred, and further summing over all sources s gives the the expected number of streams that need to be transferred from sources to servers, which corresponds to total network traffic. □

Note that, assuming that the stream of each source is used by at least one query, the network traffic cost is at least the number of distinct sources m. On the other hand, the maximum possible network traffic cost is at most km, which is achieved when each server requires each input stream. From Equation 1, note that the expected network traffic cost of random query assignment is nearly equal to the worst-case network cost whenever $\frac{1}{m}\sum_{s\in S}\left(1-\frac{1}{k}\right)^{d_s} \ll 1$. In fact, the worst-case network traffic cost is achievable under the random query assignment policy: consider as input a set of sources and a set of queries that are partitioned into k balanced pieces so that there are m/k sources and n/k queries in respective pieces $S_1, S_2, \ldots, S_k$ and $Q_1, Q_2, \ldots, Q_k$, and assume that each query in Q_j requires input from each source in S_j and none from $S \setminus S_j$. The subscription of queries to sources corresponds to a collection of k disconnected complete bipartite graphs, each with m/k sources and n/k queries. In this case, from Equation 1, the expected network traffic cost is

$$(1-(1-1/k)^{n/k})km$$

which for large n tends to the worst-case network traffic cost of km. In comparison, the best strategy is to assign each piece of queries to a distinct server; this achieves minimal network traffic and perfect load balancing. The inefficiency of the random query assignment can thus be made arbitrarily large by taking k large enough.

It is worth noting from Equation 1 that the expected network cost of random query assignment is less or equal to $(1-(1-1/k)^{\bar{d}})km$ where $\bar{d}$ is the mean number of queries per source, which follows by Jensen's inequality. From the latter upper bound, it follows that the expected network cost is less or equal to $\min\{\bar{d},k\}m$. Hence, random query assignment guarantees a small expected network cost in case of small mean number of queries per source. In other cases, it can be grossly suboptimal, which we demonstrate in Section 6.

The performance inefficiency of simple random query assignment asks for designing more sophisticated query assignment algorithms that perform better with respect to the two criteria of minimizing the network traffic cost and balancing the processing load across servers. In the following sections, we propose better approximation algorithms to offline QP problem in Section 4, and discuss online algorithms that irrevocably assign queries to servers at their arrival instances in Section 5.

4 Offline Query Partition

In this section, we discuss how to approximately solve offline QP and derive theoretical approximation guarantees.

First, we study the approximation bound for a more general case, where each query might subscribe to more than one source, referred to as *multi-source* QP. When sources have identical traffic rate, we show that there exists an algorithm that is able to approximate multi-source QP within $2d_{max}(1+\log k)$, where d_{max} is the maximum number of sources to which a query subscribes, and k is the number of servers. Since d_{max} is usually small in practice [4], this is a much tighter bound compared with the naïve bound k. Moreover, we propose several approximation algorithms that might have looser approximation bounds in theory, but have competitive performance in practice.

Second, we focus on a special case of QP, where each query subscribes to only one source, referred to as *single-source* QP. We show that there exists an algorithm that is able to approximate single-source QP within a constant 2.

4.1 Multi-source Query Partition

Section 4.1.1 presents the MQP algorithm that achieves a non-trivial approximation bound, and Section 4.1.2 introduces two approximation algorithms of competitive performance.

4.1.1 MQP *algorithm*

A challenging question to multi-source QP is whether there exists a polynomial-time algorithm that can guarantee an approximation ratio that is independent of the number of sources and the number of queries, or simply tighter than the naïve approximation bound k (the number of servers). In this paper, we provide an affirmative answer to this question! Specifically, let k, S, and Q be the number of servers, the set of sources, and the set of queries considered in a multi-source QP, respectively.

THEOREM 2. *For any given S with identical traffic rates, there exists a polynomial-time algorithm for multi-source* QP *with approximation ratio*

$$2d_{max}(\log k+1).$$

where $d_{max} = max_{q\in Q}|S_q|$. Furthermore, this bound holds for general traffic rates with an extra factor of $\omega = \max_{s\in S} w(s)/\min_{s\in S} w(s)$.

This result has an important practical implication: in systems where the number of data sources required by a query is arbitrary but bounded by a small positive integer d_{max}, there exists a polynomial-time algorithm that guarantees network traffic cost that is at most

$2d_{max}$ of the optimum network traffic cost, up to a factor that is logarithmic in the number of servers k.

In the following, we prove Theorem 2 in three steps. (1) We demonstrate that given a multi-source QP with k servers, if we can optimally solve k *minimum query type packing* problems, referred to as MQP and defined shortly, we can approximate multi-source QP within $2(1+\log k)$. (2) We show that MQP is NP-complete. (3) We further show that we can approximate MQP within d_{max}. When the context is clear, we also call the algorithm achieving the bound in Theorem 2 MQP algorithm.

We start with the definition for MQP problem.

DEFINITION 2 (MQP). *Given a set of queries Q along with their query types $T \subseteq 2^S$, a corresponding set of sources S with their traffic rate w, and a positive real number $\theta > 0$,* MQP *is to find a subset of query types $T' \subseteq T$ such that (1) $\sum_{t\in T'} n_t \geq \theta$ and (2) $w(\cup_{t\in T'} S_t)$ is minimized.*

Next, we decompose QP problem into k MQP problems as follows.

1. Given k servers and n queries, select an arbitrary server as the target server for assignment.
2. Suppose we find a subset of query types $T' \subseteq T$ for the MQP problem with
$$\theta = n - (1+\nu)\frac{(k-1)n}{k}. \qquad (2)$$
Let $\hat{Q}$ be the subset of queries corresponding to T', and let its traffic cost be $f(\hat{Q})$.
3. Note that there is a capacity constraint of $(1+\nu)\frac{n}{k}$ for each server. If $\sum_{t\in T'} n_t > (1+\nu)\frac{n}{k}$, to construct a feasible $\hat{Q}$, we arbitrarily select a query type, keep a feasible number of them, and put the rest queries of the query type back into the query pool.
4. We repeat the above steps until all queries are assigned.

Note that the constraints of the QP problem imply that $|Q_j| \geq \theta$, for server $j = 1, 2, \ldots, k$. In this view, the MQP problem with θ as given in (2) is a relaxation of the QP problem.

Let $\hat{Q}_j^* \subseteq Q$ be the optimal solution for the MQP problem on the j-th server, $f(\hat{Q}_j^*)$ be the corresponding traffic cost, and OPT be the optimal solution for the original multi-source QP problem.

LEMMA 1. *Given a multi-source* QP *problem with k servers, successive solving of k* MQP *problems yields a feasible solution for the* QP *problem; moreover, if we can solve each* MQP *problem optimally with $\hat{Q}_1^*, \ldots, \hat{Q}_k^*$, we can guarantee*

$$\sum_{j=1}^{k} f(\hat{Q}_j^*) \leq 2(\log k + 1)\mathrm{OPT}.$$

PROOF. The proof follows by upper bounding the cost incurred in each round where queries are assigned to a server by solving a MQP problem. We first show the upper bound for the traffic cost of assigning queries to the first server, and then demonstrate how we bound the traffic cost for the remaining servers.

Let $\mathrm{OPT}_j(n')$ be the optimal solution for a multi-source QP problem with n' queries, j servers, and the capacity constraint $(1+\nu)\frac{n}{k}$. Note that $\mathrm{OPT}_k(n) = \mathrm{OPT}$. For the first MQP problem, an optimal subset of queries $\hat{Q}_1^*$ assigned to server 1 with traffic cost $f(\hat{Q}_1^*)$. Since MQP is a relaxation of the QP problem, it holds $f(\hat{Q}_1^*) \leq f(Q_j^*)$, where Q_j^* is the subset of queries assigned to server j in OPT for all $1 \leq j \leq k$. Since $\mathrm{OPT}_k(n) = \mathrm{OPT}$, we obtain

$$f(\hat{Q}_1^*) \leq \frac{1}{k}\mathrm{OPT}_k(n) = \frac{1}{k}\mathrm{OPT}. \qquad (3)$$

Consider the j-th server. Let $\mathrm{OPT}_{k-j+1}(n')$ be the optimal solution given n' remaining queries, $k-j+1$ servers, and the capacity constraint $(1+\nu)\frac{n}{k}$ (note that this constraint remains the same throughout the algorithm). We claim that

$$f(\hat{Q}_j^*) \leq \frac{2}{k-j+1}\mathrm{OPT}, \text{ for } 1 < j \leq k. \qquad (4)$$

Suppose Inequation (4) is true, the proof for Lemma 1 follows by summing up the upper bounds in (3) and (4) and using the fact that harmonic series H_k holds $H_k \leq \log k + 1$. We prove Inequation (4) as follows.

First, note that given n queries and the same capacity constraint per server, if there exist feasible solutions for a system of j and k servers, such that $j \leq k$, then we prove that $\mathrm{OPT}_j(n) \leq 2\mathrm{OPT}_k(n)$. For $\mathrm{OPT}_k(n)$, let $cost_i$ be the traffic cost for server i. Without loss of generality, suppose that the servers are enumerated such that $cost_1 > cost_2 > \ldots > cost_k$. Then, we have

$$\mathrm{OPT}_k(n) = \sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i.$$

Using $\mathrm{OPT}_k(n)$, we can construct a feasible solution that requires only j servers by (1) arbitrarily selecting a server $a \leq j$ with available space, and (2) sequentially assigning queries on server $b > j$ to server a. If server a is full before all queries from server b are assigned, then arbitrarily select another server $a' \leq j$ with available space for the remaining queries from server b, and we repeat the procedure until all queries from server b are assigned. If all queries from server b are assigned but server a still has available space, we find another server $b' > j$, and assign queries from server b' to server a. By the above procedure, we can construct a feasible solution using only j servers. The resulting extra cost is no more than $j * cost_{j+1}$, since in the above procedure we break the sequential assignment at most j times, and each time add in no more than the cost of $cost_{j+1}$. Therefore,

$$\begin{aligned}\mathrm{OPT}_j(n) &\leq \sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i + j * cost_{j+1} \\ &\leq 2\sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i\end{aligned}$$

and, thus, it follows that

$$\frac{\mathrm{OPT}_j(n)}{\mathrm{OPT}_k(n)} \leq \frac{2\sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i}{\sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i} \leq 2.$$

Hence,

$$\mathrm{OPT}_j(n) \leq 2\mathrm{OPT}_k(n), \text{ for all } 1 \leq j \leq k. \qquad (5)$$

Second, given k servers and the same capacity constraint, if there exist feasible solutions for assigning n_i and n_j with $n_i \leq n_j$, then

$$\mathrm{OPT}_k(n_i) \leq \mathrm{OPT}_k(n_j). \qquad (6)$$

Finally, since $f(\hat{Q}_j^*) \leq \frac{1}{k-j+1}\mathrm{OPT}_{k-j+1}(n')$, (5) and (6), it follows

$$f(\hat{Q}_j^*) \le \frac{2}{k-j+1}\mathrm{OPT}, \text{ for } 1 < j \le k.$$

□

In Lemma 1, we derived an approximation ratio for the QP problem under assumption of an oracle providing an optimal solution to MQP problem. We next show that MQP is NP-complete; therefore, it is difficult to find a polynomial-time algorithm that solves MQP optimally.

LEMMA 2. MQP *problem is NP-complete.*

PROOF. We sketch the proof as follows. (1) To prove NP-hardness, we can reduce the NP-hard minimum *k*-union problem [31] to MQP problem. (2) It is easy to verify a solution in polynomial time. □

We propose the following algorithm to approximate MQP.

1. Order query types in decreasing order with respect to the number of queries.

2. Repeatedly pick the query type of the largest number of queries until the number of assigned queries is no less than θ.

LEMMA 3. *Suppose sources have identical source traffic rates, the above algorithm approximates* MQP *problem within factor d_{max}, where $d_{max} = max_{q\in Q}|S_q|$. In the general case of arbitrary source traffic rates, where the ratio of the maximum source traffic rate to the minimum source traffic is at most a parameter ω, the above algorithm approximates* MQP *problem within factor $d_{max}\omega$.*

PROOF. In the above algorithm, we pick the query types of the largest number of queries to saturate a server. Suppose we eventually select h query types, we conclude that the number of query types considered in an optimal solution is no less than h. Let $h+\Delta$ be the number of query types obtained from the optimal solution. For each query type, the above algorithm at most takes d_{max} times more traffic rate compared with the optimal solution, when traffic rate is identical over all sources; meanwhile, it at most takes $d_{max}\omega$ times more traffic rate, when traffic rate is arbitrary. This completes the proof of the lemma. □

4.1.2 Competitive algorithms

In this section, we present several approximation algorithms, including *incremental cost based approach* (referred to as IC) and *min-max traffic cost per server* (referred to as MMS). Although these two algorithms only hold the naïve approximation bound *k*, they demonstrate competitive performance in practice (see Section 6). These two algorithms are described as follows.

Incremental cost based approach. IC assigns queries in multiple rounds. At each round, it assigns queries to a server in three steps. (1) Given a query type $t \in T$ of non-zero number of queries and a server j of space to host more queries, we can measure the *incremental traffic cost* resulting from assigning at least one query of type t to server j. (2) We select the pair (t^*, j^*) that results in the least incremental cost. (3) We assign queries of type t^* to server j^* as many as possible until server j^* is full or there is no queries of type t^* any more.

Min-max traffic cost per server. MMS aims to minimize the maximum traffic cost among servers in multiple rounds. At each round, it assigns queries to a sever in three steps. (1) Given a query type $t \in T$ of non-zero number of queries and a server j of space to host more queries, we can measure the *traffic cost* after assigning at least one query of type t to server j. (2) We select the pair (t^*, j^*) that results in the least traffic cost. (3) We assign queries of type t^* to server j^* as many as possible until server j^* is full or there is no queries of type t^* any more.

4.2 Single-source Query Partition

In this section, we study single-source QP, where each query subscribes to only one data source. Note that in single-source QP, the number of query types equals to the number of sources; therefore, we use the term source and the term query type interchangeably.

We propose an approximation for single-source QP that assigns queries to servers successively over a number of rounds as shown in Figure 1. For server j, we define d_j to be the free capacity of server j, and at the beginning of round 0, we initialize $d_j = \lfloor (1+\nu)\cdot\frac{n}{k} \rfloor$, where $\nu \ge 0$ is the relative relaxation ratio. Moreover, we define n_s to be the number of unassigned queries requiring source s (*i.e.,* all queries that use the same source s).

```
Input:  a single-source QP instance;
Output: the resulting query assignment.

1.  while True
2.      Select server $j$ with the largest free capacity
3.      Select query type (source) $s$ of the largest traffic rate $w(s)$
4.      $b \leftarrow \min(d_j, n_s)$
5.      Assign $b$ type-$s$ queries to server $j$
6.      $d_j \leftarrow d_j - b$
7.      $n_s \leftarrow n_s - b$
8.      if there is no more queries
9.          return
```

Figure 1: The 2-approximation for single-source QP.

THEOREM 3. *The approximation algorithm shown in Figure 1 has the following guarantees: (1) approximation ratio of $1+\frac{k}{m} \le 2$, where m is the number of sources and k is the number of servers, for sources of identical source traffic rates; and (2) approximation ratio of 2, for sources of arbitrary source traffic rates.*

PROOF. Note that in the proof, we focus on the cases where $0 < n_s < d_j$ for all $s \in S$ and $j = 1, 2, \ldots, k$, since the other cases can be reduced to the above cases. We assume that $k < m$; otherwise, we can find the optimal solution in polynomial time.

Identical source traffic rates. Without loss of generality, we assume $w(s) = 1$, for all $s \in S$. In the following, we show the lower bound for the optimal solution and the upper bound for the approximate solution.

The lower bound for the optimal solution is *m*, since each source will be used by at least one server in the system.

The upper bound for the above algorithm is $k+m$, and it is derived as follows. (1) At each round, we either make a query type take all the remaining space of a server, or make a server include all the remaining queries of a query type. In other words, either the number of available servers or the number of available query types decreases by 1. (2) To this end, we safely claim that the number of required rounds cannot go beyond $k+m$. (3) Since each round increases traffic rate by at most 1, the total traffic rate cannot be more than $k+m$.

Using the lower and upper bounds, we obtain the approximation ratio of $1+k/m$.

Arbitrary source traffic rates. Similarly, we demonstrate the lower bound for the optimal solution and the upper bound for the approximate solution.

The lower bound for the optimal solution is $w(S)$, since we have to transfer each source at least once.

The upper bound for the approximate solution is $2w(S)$, and it is derived as follows. Denote with r_s the number of rounds to assign queries of query type s, which in total introduces $r_s \cdot w(s)$ of traffic cost into the system. By the same argument as for the case of identical source traffic rates, the total number of rounds is at most $m+k$, *i.e.*, $\sum_{s\in S} r_s \le k+m$. From this it follows that

$$\sum_{s\in S}(r_s - 1) \le k.$$

Since $0 < n_s < d_j$, we can guarantee that the top-k query types (in terms of traffic rate) will be assigned at most once. Therefore,

$$\sum_{s\in S}(r_s - 1)w(s) \le kw_{k+1} \le w(S)$$

where w_{k+1} is the $k+1$-largest traffic rate among all query types. The total traffic rate satisfies

$$\sum_{s\in S} r_s w(s) \le w(S) + \sum_{s\in S}(r_s - 1)w(s) \le 2w(S),$$

hence the algorithm approximates the problem within factor 2. □

Remark. For single-source QP, there exists a constant factor approximation algorithm, and this guarantee holds for any number of sources m, number of queries n, and number of servers k. Moreover, if in addition the source traffic rates are identical, then an approximation ratio of $1+k/m$ can be guaranteed. Thus, this approximation ratio guarantee can be arbitrarily near to the best one by ensuring a sufficiently large number of sources relative to the number of servers. In a practical system, this suggests that in the case of single-source queries and many sources with approximately identical traffic rates, it would be computationally feasible to guarantee nearly optimal performance.

5 Online Query Partition

In this section, we consider online QP, which assign queries to servers at their arrival time. We focus on the class of online algorithms that decide which server to host an incoming query based on (1) the set of source streams required by the given incoming query and (2) the queries that were previously assigned to servers and are still in the system. The key to design such an online algorithm is to choose a metric that indicates which server to host the incoming query, and a reasonable metric should combine the two optimization criteria: load balancing and network traffic cost minimization. In the following, we present several greedy metrics for online algorithms and the intuition behind those metrics in Section 5.1. Moreover, we provide a discussion in Section 5.2 on how to make use of extra information (Section 5.2.1) or resources (Section 5.2.2) to improve the performance of online algorithms.

5.1 Greedy online algorithms

In this section, we present three greedy online algorithms, and describe the intuition behind those algorithms.

Although the three algorithms follow different metrics to decide which server to host an incoming query, they share a common pipeline as shown in Figure 2. Given an incoming query q, k servers along with the corresponding set of queries a server is hosting, the relative relaxation ratio v, and a predefined metric M, the server to host q is decided as follows. (1) From all k servers, we find the candidate servers C each of which will not violate the balance constraint if we add q into the server. (2) From C, we find the servers C^* each of which will have the highest score in terms of M if we

```
Input:  (1) an incoming query $q$ requiring a set of sources $S_q$;
        (2) $k$ servers and the queries they are hosting $Q_1,\dots,Q_k$;
        (3) relative relaxation ratio $v$;
        (4) a predefined metric $M$;
Output: the server that will host $q$.

1. Find candidate servers $C = \left\{i \mid |Q_i| + 1 < (1+v)\frac{\sum_{j=1}^{k}|Q_j|}{k}\right\}$
2. Find $C^* \subseteq C$ such that $\forall i \in C^*$,
3.     $M(Q_i \cup \{q\}) \ge M(Q_j \cup \{q\}), \forall j \in C$
4. if $|C^*| == 1$
5.     return the only server in $C^*$
6. else
7.     return the server $p = argmin_{i\in C^*}\{|Q_i|\}$
```

Figure 2: The sketch for online algorithms.

add q into the server. (3) If there is only one server in C^*, we assign q to the server; otherwise, we select the least loaded server from C^* and then assign q to the server.

In this work, we propose three metrics to support online assignment decision: (1) *least incremental cost first* (referred to as LeastCost), (2) *least source cost per server first* (referred to as LeastSource), and (3) *least number of query types first* (referred to as LeastQT). Given a query q and a set of queries Q_j hosted by server j, the three metrics behave as follows.

LeastCost. If $f(Q_j \cup \{q\}) - f(Q_j)$ results in a smaller incremental traffic cost, we will give server j a higher score as below.

$$M_{\mathsf{LeastCost}}(Q_j \cup \{q\}) = \frac{1}{f(Q_j \cup \{q\}) - f(Q_j)}.$$

LeastCost is a natural metric for QP: since the ultimate goal of QP is to minimize traffic cost in a system, LeastCost approaches this goal by locally minimizing traffic cost at each query arrival.

LeastSource. If $f(Q_j \cup \{q\})$ results in a smaller traffic cost, we will give server j a higher score as below.

$$M_{\mathsf{LeastSource}}(Q_j \cup \{q\}) = \frac{1}{f(Q_j \cup \{q\})}.$$

LeastCost has a potential issue: one server might subscribe too many sources because of local optimal decisions such that many incoming queries will be assigned to the server, the server will be full quickly, and eventually the server becomes unavailable for hosting incoming queries. If this effect propagates among servers, the overall traffic cost in the system will be very high. To mitigate this effect, we come up with LeastSource that aims to balance the traffic cost among servers such that no server will subscribe too many sources and result in too high traffic.

LeastQT. Let T_j be a set of query types such that $\forall t \in T_j$, there exists at least one query of type t hosted by server j, and t_q be the query type of q. If $|T_j \cup \{t_q\}|$ is smaller, we will give server j a higher score as below.

$$M_{\mathsf{LeastSource}}(Q_j \cup \{q\}) = \frac{1}{|T_j \cup \{t_q\}|}.$$

LeastCost and LeastSource have a common issue: a few servers might subscribe too many popular sources. If one server subscribes too many popular sources, it is able to host queries of various query types, and will be crowded quickly. If this effect propagates in the system, we have to make many servers subscribe those popular sources. One way to mitigate this effect is to limit the number of query types in a server such that no server can host too many query types and get crowded soon.

5.2 Discussion

In this section, we discuss how we develop online algorithms when we have more knowledge or more resources. The above metrics provide heuristic algorithms to solve online QP. Indeed, given the limited knowledge of only information about the incoming query and assigned queries in a system, we might not have much space to develop sophisticated algorithms. However, in practice, we might know some statistical information about queries, and might relax the load balancing constraint at specific conditions.

5.2.1 *With known query type distribution*

When we deal with online QP, we might know the information about query type statistics. In particular, such statistics consist of the rate at which specific query types will arrive in the system, and may well be available in production systems that have been in operation for some time, which allows us to collect and maintain statistics about the query workload.

Concretely, with such statistical knowledge, we can develop an online algorithm as follows. Assume that popularity of query types is known: the probability that an incoming query is of type t follows a multinomial distribution with parameters $(\lambda_t,\ t \in T)$, where $T \subseteq 2^S$ is the universe of considered query types . Knowing this distribution allows us to develop an online algorithm that makes reservations for query types in advance, and then assigns queries at their arrival times based on their query types. This allows one to emulate what an offline algorithm does. Given $(\lambda_t,\ t \in T)$, we reduce an online QP to an offline QP as follows.

1. λ_t (the probability that a type-t query arrives) is reduced to n_t (number of type-t queries);

2. δ_j, the probability that server j receives a query, is reduced to the balance constraint, the number of queries a server could host at most, and in particular, we set $\delta_j = \frac{1}{k}$ in the algorithm;

3. $\pi_{t,j}$, the probability that a type-t query is assigned to server j, is reduced to the number of type-t queries in server j.

Therefore, with the statistical information on query type distribution, we can reuse the offline algorithms discussed in Section 4 to solve online QP.

5.2.2 *Relaxed Load Balancing Constraints*

QP problem is defined as a bi-criteria optimization problem where one of the criteria is balancing the load across different servers. Specifically, the problem corresponds to finding a query partition such that the maximum load is at most $(1+\nu)$ of the mean load across different servers, for given input parameter $\nu \geq 0$. For an online QP, requiring to obey this condition at each query assignment instance may well be too restrictive and amount in sub-optimal query assignments with respect to the long-run network traffic cost.

EXAMPLE 1. *Consider a system of* 10 *servers, and a fixed relative relaxation ratio of* 0.05*. The first* 10 *queries will be distributed to* 10 *different servers, and that may result in* 10 *different copies of the same stream, if those queries read from the same source. This is because the average load times the relaxation is strictly less than* 1 *until the* 10*-th query. In general, during the initialization, the allocation of queries to servers will be grossly sub-optimal.*

To resolve the above issue, we consider a relaxation of the load balancing constraints that is defined as follows. (1) We define another balance constraint for a system in its initial phase, and use *absolute relaxation ratio* to control the balance constraint. (2) In the initial phase, a system uses a balance constraint decided by absolute relaxation ratio. When the system hosts more than n queries, we switch back to the balance constraint decided by relative relaxation ratio. Let n be the number of queries in the system. The system is said to be α-absolutely balanced, if the number of queries in any server is no more than $\frac{n}{k}+\alpha$, where $\alpha \geq 0$ is the absolute relaxation ratio. The system is said to be $(1+\nu)$-relatively balanced, if the number of queries in any server is no more than $(1+\nu)\cdot\frac{n}{k}$, where $\nu \geq 0$ is the relative relaxation ratio.

In an online system, when the number of input queries is small, we apply absolute load balancing constraint to balance the workload of servers; when the number of queries becomes sufficiently large, we switch to relative balancing constraint to balance the workload. In other words, given the values of parameters α and ν and the number of input queries n, the load balancing constraint for each server j is defined to be $d_j(n) \leq d(n)$, where $d_j(n)$ is the number of queries already assigned to server j and

$$d(n) = \max\left(\frac{n}{k}+\alpha, (1+\nu)\cdot\frac{n}{k}\right). \quad (7)$$

Note that a system switches to the relative load balancing as soon as the number of input queries n satisfies $n \geq (\alpha/\nu)k$.

One question is how to set α provided ν. In the following, we present one principled way to do so. Assume that the probability of assigning a query to a server is according to a uniform random distribution across servers. Under this assumption, $(d_1(n), d_2(n), \ldots, d_k(n))$ is a random variable according to multinomial distribution with parameters n and $(1/k, 1/k, \ldots, 1/k)$ where $\sum_{j=1}^{k} d_j(n) = n$. By the union bound, we have

$$\Pr\left(\cup_{j=1}^{k}\{d_j(n) > d(n)\}\right) \leq \sum_{j=1}^{k} \Pr(d_j(n) > d(n)). \quad (8)$$

Using Hoeffding's inequality, we obtain

$$\Pr(d_j(n) > d(n)) \leq \exp\left(-\frac{2(d(n)+1-\frac{n}{k})^2}{n}\right). \quad (9)$$

Combining with (7), we have

$$\Pr(d_j(n) > d(n)) \leq \exp\left(-\frac{2\nu^2}{k^2}n\right).$$

Therefore,

$$\Pr\left(\cup_{j=1}^{k}\{d_j(n) > d(n)\}\right) \leq k e^{-\frac{2\nu^2}{k^2}n}.$$

From this it follows that $d_j(n) \leq d(n)$ for all $j = 1, \ldots, k$ with high probability provided that the following condition holds

$$k = O\left(\nu\sqrt{\frac{n}{\log n}}\right).$$

Note that for given $\varepsilon > 0$, $d_j(n) \leq d(n)$ for all $j = 1, \ldots, k$ to hold with probability at least $1-\varepsilon$ it suffices that

$$n \geq \frac{k^2}{2\nu^2}\log\left(\frac{1}{\varepsilon}\right). \quad (10)$$

Given $\nu > 0$, we want the the algorithm to switch to relative load balancing once the probability of violation of the relative imbalance is guaranteed to be smaller or equal than given $\varepsilon > 0$. By (7) the switch over from the absolute to relative balancing constraints happens at the smallest integer n such that $n \geq \alpha k/\nu$. Combined with (10), we observe that it suffices to switch over when the number of queries n satisfies (10) and it suffices that the absolute relaxation parameter α is chosen such that

$$\alpha \leq \frac{k}{2\nu} \log\left(\frac{1}{\varepsilon}\right).$$

An important insight from this is that the absolute relaxation ratio α should not be taken too large, and it should be at most a quantity that scales linearly with the number of servers k (in practice, $\alpha \geq 10$ works well).

6 Experimental Evaluation

This section evaluates the off-line and on-line algorithms of Section 4 and Section 5 using extensive simulations. We start by describing the experimental set-up (Section 6.1), and then analyze the network traffic overheads of those algorithms. Our results support the following hypotheses:

1. Suitable assignment policies reduce network traffic substantially compared with naïve random query assignment.

2. One of our query assignment policies, LeastCost, consistently exhibits superior performance than other online, and sometimes even offline, query assignment policies, in a wide range of configurations.

3. LeastCost scales with respect to the number of queries, sources, and servers; moreover, it is robust to dynamic arrival and departure for both queries and servers.

6.1 Experimental Setup

Synthetic query subscription. We generated input queries based on the model of [16]. Queries' subscription can be represented by a bipartite graph $G = (S, Q, E)$, where S is the set of sources, Q is the set of queries, and there is an edge $(s, q) \in E$ if and only if query q receives data from source s. For convenience, we say that all queries that use the same set of sources belong to the same *query type*. Let m be the total number of sources, and n the total number of queries. We assume that G is a random bipartite graph with given degree distribution over source vertices and given distribution over query vertices. In particular, we assume a family of random bipartite graphs where (1) the degree distribution of source vertices is a Zipf distribution with exponent β (modeling heavy-tailed statistics in source popularities [16, 17]), and (2) the degree of query vertices is fixed to parameter $d > 0$. For brevity, we assume that each query increase the load on the server by one unit.

Offline algorithms. We implement the following offline algorithms: (1) incremental cost based approach IC, (2) minimum query packing based approach MQP, and (3) min-max traffic cost per server MMS. Note that those three algorithms incorporate the single-source query assignment algorithm when using single-source queries. As a baseline, we use the following random offline assignment algorithm OffRand: (1) randomly select all queries of the same query type; (2) randomly select a server of available space to host queries; (3) assign queries to the selected server until either the server is saturated or all queries (of query type) have been assigned; (4) remove server or query type from further consideration; and (5) repeat from (1) until all queries are assigned.

Online algorithms. We implemented the following online algorithms: (1) Least incremental traffic first LeastCost, (2) Least number of sources first LeastSource, and (3) Least query types first LeastQT. As a baseline, we implemented a random online algorithm OnRand as follows: given a query, we find the candidate servers that can accept the new query without violating the balance constraint, and then randomly select one of them as the target server for assignment.

Parameters. In most the experiments, we considered four parameters: (1) the exponent β that governs sources' Zipf degree (popularity) distribution ranging from 1.0 to 3.0 with default value 2.0, (2) the number of sources per query from 1 to 10 with default value 2, (3) the number of servers from 10 to 1000 with default value 100; and (4) the number of queries from 10K to 1M with default value 100K.

When considering query or server dynamics, we also consider the following parameters when studying the algorithms' performance: (a) mean query life-time (Section 6.3.2), and (b) server departure rate (Section 6.3.3).

Moreover, for all the offline algorithms, we fixed the relative relaxation ratio ν to be 0.05. For all the online algorithms, we fixed the relative relaxation ratio ν to be 0.05, and the absolute relaxation ratio α to be 10 (Section 5.2.2).

Performance metrics. We employed *replication factor* as the metric to evaluate the performance of algorithms. Let tc be the resulting traffic cost of an algorithm, S be a set of sources with traffic cost w, and $f(S)$ be $f(S) = \sum_{s \in S} w(s)$. The replication factor of an algorithm is defined by $\frac{tc}{f(S)}$. Intuitively, replication factor presents normalized traffic cost resulting from an algorithm.

We run each configuration 10 times, and present the average value.

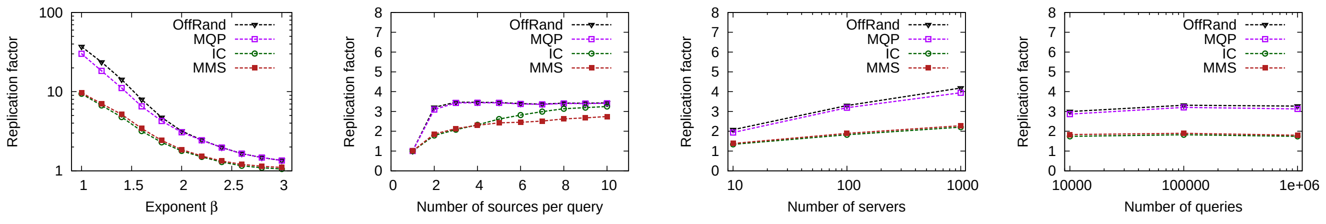
6.2 Offline Algorithms

We examine the replication factor and competitive ration of the offline algorithms in Figure 3. Note that we assume the same unit traffic rate for all sources; we examine arbitrary traffic rates in Section 6.4.

Figure 3(a) shows the network traffic replication factors while varying the exponent of the Zipf distribution for the number of queries per source (source popularity); observe that lower values are better. In particular, the number of sources per query is fixed to 2, the number of queries is set to 100K, and the number of servers is fixed to 100. We have the following observations. (1) Heavier power-law distributions (*i.e.,* smaller exponents) result to larger replication factors. This is intuitive as a heavier tail implies the existence of a few sources with many query subscription, which makes query assignment more challenging. (2) IC and MMS consistently exhibit the best performance, up to 4 times better than OffRand.

Figure 3(b) shows the network traffic replication factor while varying the number of sources per query. In particular, the exponent for the power-law source popularity distribution is fixed to 2.0, the number of queries is set to 100K, and the number of servers is 100. We have the following observations. (1) The replication factor increases with the number of sources per query. (2) For single-source queries, the replication factor is no more than 2, which confirms the theoretical guarantee in Theorem 3. In fact, it is nearly optimal. (3) Another noteworthy property is a diminishing increase of the replication factor with the number of sources per query. It is observed for all the offline algorithms considered. Specifically, OffRand and MQP reach a plateau replication factor at about two or three sources per query. (4) MMS exhibits the best performance, and this is matched by IC for sufficiently small number of sources per query.

Figure 3(c) plots the network traffic replication factor *vs.* the number of servers, when the exponent for the power-law source popularity distribution is 2.0, the number of queries is 100K, and



(a) Varying exponent β (b) Varying #sources/query (c) Varying #servers (d) Varying #queries

Figure 3: Performance of offline query assignment.

the number of sources per query is 2. We observe that the replication factor increases with the number of servers, but this increase is sub-linear in the number of servers.

Finally, Figure 3(d) shows that the network traffic replication factor is invariant to the number of input queries, (the exponent for the power-law source popularity distribution is 2.0, the number of servers is set to 100, and the number of sources per query is 2).

In summary, Figure 3 demonstrates that MMS consistently outperforms other offline algorithms, and results in up to 4 times performance gain compared with the naïve offline algorithm OffRand.

6.3 Online Algorithms

In this section, we examine the performance of online algorithms in the following settings: (1) online arrival of queries without query departure and a fixed number of servers, (2) online arrival of queries with query departure and a fixed number of servers, and (3) dynamic arrival and departure for both queries and servers.

6.3.1 Query Arrivals, No Departures

Figure 4 plots the replication factor for online algorithms, in similar settings as Figure 4. The two sets of plots are qualitatively similar, hence, we focus next on their differences.

First, LeastCost exhibits the best performance, and sometimes even outperforms the best offline algorithm MMS. Second, the performance of LeastSource is close to LeastCost. Third, LeastCost performs up to 4 times better than OnRand, and the performance gain is close to what we observed between MMS and OffRand in the offline case. The performance of LeastQT is virtually identical to that of OnRand. Thus typically LeastQT provides no benefits.

6.3.2 Query Arrivals and Departures

A streaming query service hosts queries posted by users, and many such queries would be hosted only for a limited time, *e.g.*, the user may be interested in travel updates only while on the road. Hence, it is important to examine query assignment strategies in a system with query arrivals and departures. To this end, we used the following model of query dynamics.

We consider the query arrival and departure process as a discrete time process of τ time steps: (1) at each time step, the number of arrived queries follows a Poisson distribution; and (2) each arrived query is associated with a lifetime, which is drawn from some distribution. In particular, we considered two parametric families of distributions: (1) exponential distributions that model the cases of light-tailed query lifetimes, and (2) Pareto distribution that model the cases of heavy-tailed query lifetimes. However, we found similar results for the two different families of distributions, so we only present the results for the exponential distribution.

Figure 5 plots the performance of online algorithms while considering query dynamics, where the mean query lifetime ranges from 10 to 1M time steps with default value 100K, the average number of arrived queries per time step is 1, and the total number of discrete time steps τ is set to be 1M in all cases. In general, we observe that LeastCost consistently yields the best performance. In Figure 5(d), we varied the mean query lifetime. By Little's law, the mean number of queries in the system is the product of the query arrival rate and the mean query lifetime, and thus the given range covers the mean number of queries in the system from 10 to 1M queries. Given that the number of servers is 100 (in all cases in Figure 5 except Figure 6(c)), we cover the system operating points of 0.1 to 10K queries per server, which covers the range of lightly to highly loaded servers. Figure 5(d) demonstrates that replication factor tends to increase with the load of the servers for all online algorithms; however, it is noteworthy that this increase is rather slow for LeastCost, which is sub-linear in the mean query lifetime.

6.3.3 Server Arrivals and Departures

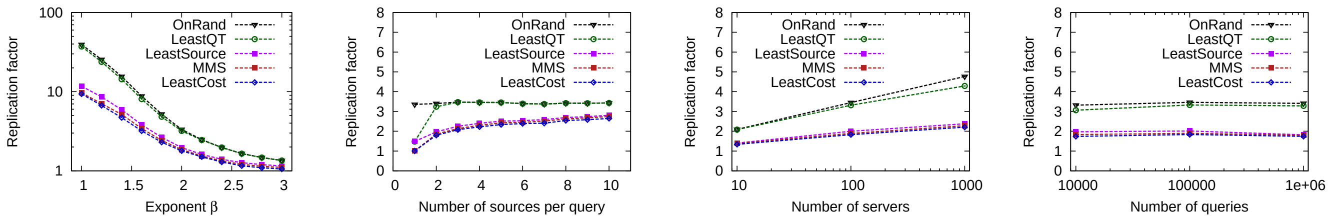
In a production environment, we also expect server churn; servers may fail, and hence queries need to be re-assigned, and new servers may be added to cope with increased demand, or after recycling failed servers. It is thus important to examine the robustness of different query assignment policies, with respect to server arrivals and departures.

We modeled server dynamics similar to query dynamics: We start with k servers, and queries arrive in τ time steps. At each time step, the number of arrived queries follows a Poisson distribution, and each query is associated with a lifetime that follows an exponential distribution. Moreover, starting at time step 1, after every γ time steps (where γ is referred to as *server departure rate*), we make a Bernoulli trial: with probability 0.5, we add a new server; otherwise, we randomly delete a server, and re-assign its queries using the online algorithm (*i.e.*, assuming that they are new queries).

Figure 6 demonstrates online algorithms' performance when we consider both query and server dynamics. By default, we consider 100 servers, the exponent β for source popularity distribution is 2.0, the number of sources per query is 2, the total number of time steps is 1M, the mean number of arrived queries per time step is 1, the mean query lifetime is 100K, and the server departure rate γ is 10K. In Figure 6(a), 6(b), 6(c), and 6(d), we varied exponent β for source popularity distribution, the number of sources per query, the number of servers, and server departure rate, respectively. Consistent with earlier results, we observed that LeastCost exhibits the best performance, and is robust with respect to the dynamics of the server arrival and departure.

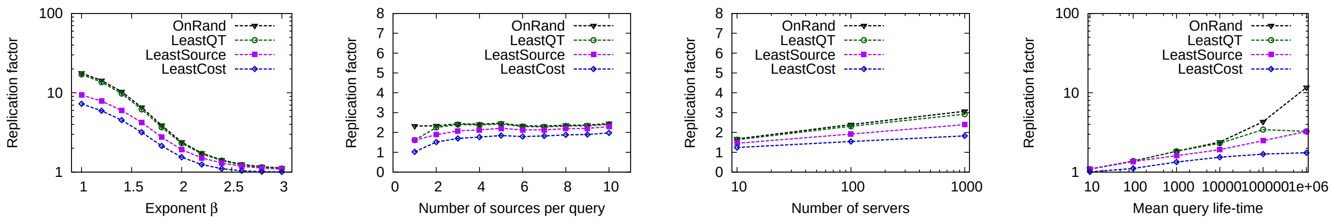
6.4 Heterogeneous Source Traffic Rates

Insofar, we have examined the performance of query assignment algorithms for the case of sources with identical traffic rates. We



(a) Varying exponent β (b) Varying #sources/query (c) Varying #servers (d) Varying #queries

Figure 4: Performance of online query assignment without query departures.



(a) Varying exponent β (b) Varying #sources/query (c) Varying #servers (d) Varying mean life-time

Figure 5: Performance of online algorithms with dynamic query arrivals and departures.

now examine a more general case where different sources have different traffic rates. Since many phenomena in nature exhibit heavy-tail distributions [17], we expect that in practice the source traffic rates to follow a heavy-tail distribution. We consider a range of values of the shape parameter that span the case of fast decaying tail (exponent value of 3) and slow decaying tail (exponent value of 1). To define the source traffic rates, we also need to decide the assignment of source traffic rates to sources, and how this assignment correlates with other factors, such as the popularity of sources (measured by the number of query subscriptions to a source). To cover different possible scenarios, we consider the following three cases: (1) *random*, where source traffic rates are assigned to sources independently of their popularity, (2) *positively correlated*, where the traffic rate of a source is proportional to the source's popularity, and (3) *negatively correlated*, where the traffic rate of a source is inversely proportional to the source's popularity.

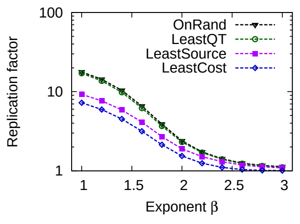
Figure 7 shows algorithms' performance on heterogeneous source traffic rates over the above three cases. In particular, we consider the best offline algorithm MMS, all three online algorithms, and the online random algorithm OnRand. We have the following observations. (1) The more positive the correlation between the source traffic rates and popularity of sources is, the larger the network traffic replication ratio will be. (2) In most cases, the best performing query assignment policy is LeastCost. Specifically, when source traffic rates are negatively correlated with source popularity, LeastCost is substantially better than MMS.

7 Related Work

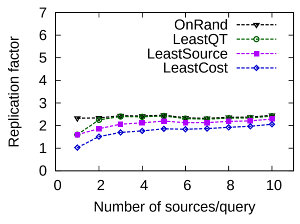
The query partition problem studied in this paper is a resource allocation problem, formulated as a combinatorial optimization problem. Specifically, it is related to load balancing over parallel machines and balanced graph partitioning problems. It can also be seen as optimizing multiple multicast information deliveries where each source connected to a set of queries (receivers) is a multicast tree and the objective is to optimize the placement of receivers. More generally, the problem can be seen to be that of optimizing the data locality for efficient processing in distributed systems.

Various different kinds of resource allocation problems were previously studied, including virtual circuit routing (e.g. [25], [5], [12]), multi-path routing ([34]), multicommodity flows (e.g. [21]), multicast virtual circuit routing (e.g. [6], [30]), assignment of tasks to parallel machines (e.g. [27]), data placement in servers (e.g. [22], [25], [15]), cache placement in the context of web servers ([19]), virtual machine placement in data centers (e.g. [9], [35]). None of these resource allocation problems correspond to the query partition problem studied in this paper. A more closely related work is that of load balancing and balanced graph partitioning which we discuss in some more detail in the following.

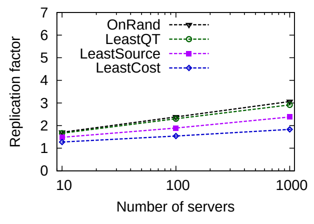
A related line of work is that of load balancing over parallel machines, and the class of load balancing problems abstracted out as *balls and bins* problems, *e.g.*, see [8, 24, 10] and the references therein). In this context, various authors studied the problem of minimizing the maximum load (or congestion), e.g. [7], and packing of requests under knapsack constraints, e.g. [27, 13]. Random query assignment benchmark considered in this paper and commonly deployed in practice, is a well-studied randomized load balancing strategy with the maximum load of $\frac{n}{k}+O(\sqrt{\frac{n\log k}{k}})$ with probability $o(1)$, for $n \gg k(\log k)^3$ ([24]). Other such randomized load balancing strategies have been studied, e.g. power of two choices, where each ball is assigned to the least loaded of two bins selected uniformly at random and independently from other ball assignments. The maximum load of such a scheme is $\frac{n}{k}+O(\log\log k)$, with high probability, for $n \gg k$ ([10]). Related work is also that on online bin packing where the bounds on the competitive ratio with respect to the offline solution were derived under arrival inputs according to random permutation or independent and identically distributed sequence, *e.g.*,, see [13] and the references therein. A main distinction with all this work is that



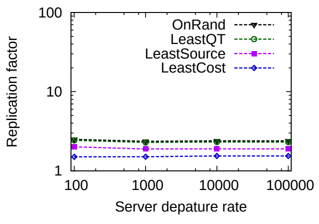
(a) Varying exponent β



(b) Varying #sources/query



(c) Varying #servers



(d) Varying departure rate

Figure 6: Performance of online algorithms with dynamic query and server arrivals/departures.

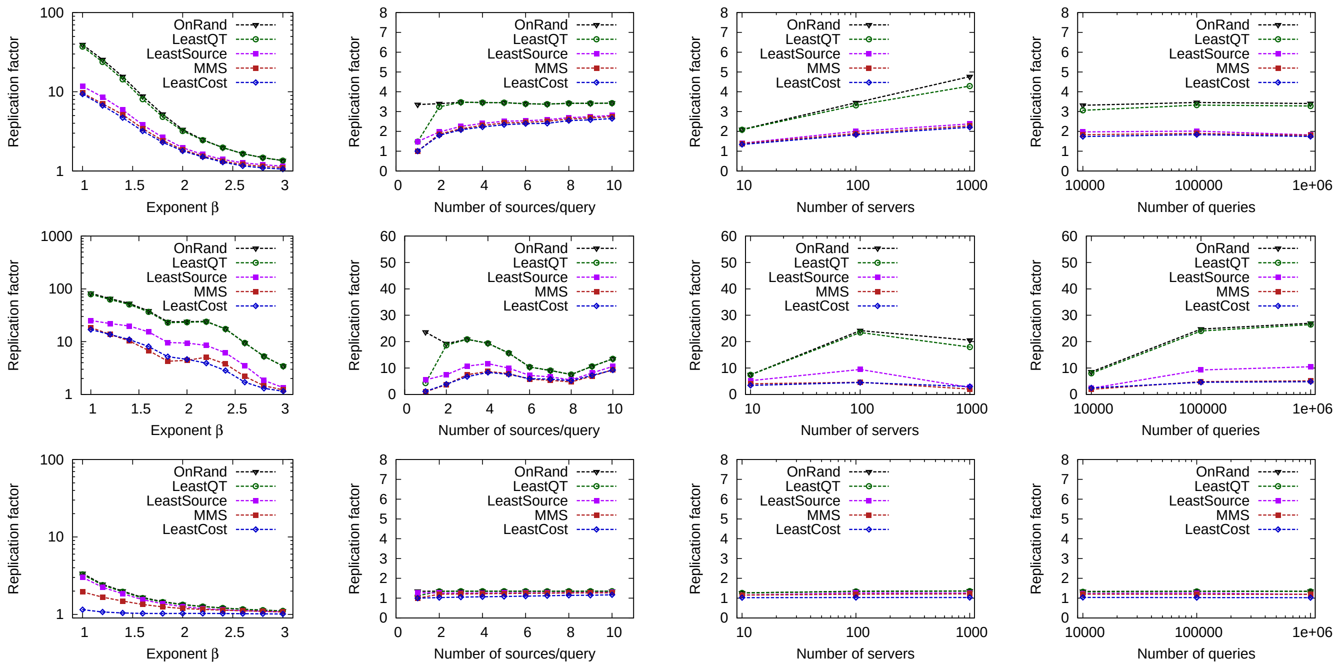


Figure 7: Performance of query assignment algorithms on heterogeneous source traffic rates: (top) random, (middle) positively correlated, and (bottom) negatively correlated.

the query partition problem is defined as a bi-criteria optimization problem, where the load balancing is only one of the two criteria.

The query partition problem is an esoteric instance of a balanced graph partitioning problem. Standard balanced graph partitioning problem is defined as follows: given a graph with n vertices, a positive integer k and a parameter $\nu \geq 0$, the goal is to partition a set of vertices into k partitions each with at most $(1+\nu)n/k$ vertices and such that the number of edges cut is minimized. The best known approximation ratio for this problem is $O(\sqrt{\log n \log k})$ which is due to [18]. We note that the query assignment problem has the same form of constraints. A notable difference is the objective function, which in case of the query assignment problem, is a submodular function of specific, different type. The unconstrained problem of minimizing a submodular function in the context of graph partitioning was considered, *e.g.*,, by [11], who derived a 2-approximation algorithm for this problem. A notable difference is that in the query assignment problem the objective function is a submodular function of specific type and the problem is constrained with cardinality constraints. [29] obtained the approximation ratio of $\Theta(\sqrt{\frac{n}{\ln n}})$ for a general class of submodular function minimization, allowing for cardinality constraints. Our result provide better approximation ratio bounds, for a more specific class of submodular functions.

8 Conclusions

In this paper, we have proposed and studied the assignment of streaming queries to servers. This is important in the design of platforms that execute small ("mice") streaming queries as a service. For such scenarios, where many streams need to be delivered to servers and the density of queries to servers is high, we need to minimize network traffic while balancing load among servers; we demonstrated that this problem is NP complete, and derived approximation guarantees. We studied, analytically and with simulations, off-line and on-line heuristics for this multi-objective problem. In particular, we proposed a heuristic that performs well under a wide range of scenarios, including query and server churn.

Our approach treats the computations done by the queries as black boxes. Even though we do balance the compute utilization among the server, we do not optimize the computations independently. Assuming that it is possible to prove that two queries that use the same input streams, also perform similar computations, then it should be possible to also optimize the query computations, *e.g.,* by executing the common computations once. Observe that in current streaming platforms, such as S4 [33], the users are encouraged to re-use existing computations; on the contrary, when executing queries as a service, such optimization need to happen automatically. In the presence of such compute optimizations, the minimization function of the assignment problem needs to capture both network and compute loads (compare to the current approach of minimizing network traffic while balancing load). We leave this problem as future work.

Acknowledgements

This work benefited from numerous useful discussions with a number of colleagues from Microsoft Bing IPE team and Microsoft Research, including (in alphabetical order) Mike Andrews, JP Ahopelto, Tony Bendis, Bart de Smet, Vlad Eminovici, Flavio Junqueira, Thomas Karagiannis, Christophe Lecas, and Savas Parastatidis.

References

[1] T. Akidau et al. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". English. *Proceedings of the VLDB Endowment* 6.11 (2013).

[2] *Amazon Elastic Compute Cloud (EC2)*. URL: `http://aws.amazon.com/ec2/` (visited on 12/10/2013).

[3] *Amazon Kinesis*. URL: `http://aws.amazon.com/kinesis/` (visited on 12/10/2013).

[4] R. Ananthanarayanan et al. "Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams". SIGMOD'13. ACM. 2013.

[5] J. Aspnes et al. "On-line routing of virtual circuits with applications to load balancing and machine scheduling". *Journal of the ACM (JACM)* 44.3 (1997).

[6] B. Awerbuch and Y. Azar. "Competitive multicast routing". *Wireless networks* 1.1 (1995).

[7] Y. Azar and L. Epstein. "On-line load balancing of temporary tasks on identical machines". *SIAM Journal on Discrete Mathematics* 18.2 (2004).

[8] Y. Azar et al. "Balanced Allocations". *SIAM Journal on Computing* 29.1 (1999).

[9] N. Bansal et al. "Minimum congestion mapping in a cloud". PODC'11. ACM. 2011.

[10] P. Berenbrink et al. "Balanced allocations: The heavily loaded case". *SIAM Journal on Computing* 35.6 (2006).

[11] C. Chekuri and A. Ene. "Approximation algorithms for submodular multiway partition". FOCS'11. IEEE. 2011.

[12] B. B. Chen and P.-B. Primet. "Scheduling deadline-constrained bulk data transfers to minimize network congestion". CCGRID'07. IEEE. 2007.

[13] N. Devanur et al. "Near optimal online algorithms and fast approximation algorithms for resource allocation problems". EC'11. ACM. 2011.

[14] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[15] D. Golovin et al. "Quorum placement in networks: Minimizing network congestion". PODC'06. ACM. 2006.

[16] J.-L. Guillaume and M. Latapy. "Bipartite graphs as models of complex networks". *Physica A: Statistical Mechanics and its Applications* 371.2 (2006).

[17] D. Gunawardena et al. "Characterizing Podcast Services: Publishing, Usage, and Dissemination". IMC'09. ACM. 2009.

[18] R. Krauthgamer, J. S. Naor, and R. Schwartz. "Partitioning Graphs into Balanced Components". SODA'09. SIAM. 2009.

[19] P. Krishnan, D. Raz, and Y. Shavitt. "The cache location problem". *IEEE/ACM Transactions on Networking (TON)* 8.5 (2000).

[20] N. Leavitt. "Complex-Event Processing Poised for Growth". *Computer* 42.4 (2009).

[21] T. Leighton and S. Rao. "Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms". *Journal of the ACM (JACM)* 46.6 (1999).

[22] B. M. Maggs et al. "Exploiting locality for data management in systems of limited bandwidth". FOCS'97. IEEE. 1997.

[23] L. Neumeyer et al. "S4: Distributed Stream Computing Platform". KDCloud'10. IEEE Computer Society, 2010.

[24] M. Raab and A. Steger. "Balls into Bins - A Simple and Tight Analysis". Vol. 1518. Springer, 1998.

[25] H. Racke. "Minimizing congestion in general networks". FOCS'02. IEEE. 2002.

[26] *Rackspace: the open cloud company*. URL: `www.rackspace.co.uk` (visited on 12/10/2013).

[27] D. B. Shmoys and Éva. Tardos. "An approximation algorithm for the generalized assignment problem". English. *Mathematical Programming* 62.1-3 (1993).

[28] *Storm: Distributed and fault-tolerant realtime computation*. URL: `http://storm-project.net/` (visited on 12/10/2013).

[29] Z. Svitkina and L. Fleischer. "Submodular approximation: Sampling-based algorithms and lower bounds". *SIAM Journal on Computing* 40.6 (2011).

[30] S. Vempala and B. Vöcking. "Approximating multicast congestion". Springer, 1999.

[31] S. A. Vinterbo. *A note on the hardness of the k-ambiguity problem*. Tech. rep. DSG 2002-006. Harvard Medical School, 2002.

[32] *Windows Azure: Microsoft's Cloud Platform*. URL: `http://www.windowsazure.com/en-us/` (visited on 12/10/2013).

[33] *S4: distributed stream computing platform*. URL: `http://incubator.apache.org/s4/` (visited on 12/10/2013).

[34] X. Zhu and B. Girod. "A distributed algorithm for congestion-minimized multi-path routing over Ad-hoc networks". ICME'05. IEEE. 2005.

[35] Y. Zhu and M. Ammar. "Algorithms for assigning substrate network resources to virtual network components". INFOCOM'06. IEEE. 2006.