Herding "Small" Streaming Queries

Bo Zong,* Christos Gkantsidis, Milan Vojnovic
Microsoft Research
Cambridge, UK
bzong@cs.ucsb.edu, {chrisgk,milanv}@microsoft.com

ABSTRACT

We study the problem of placing streaming queries into servers. Unlike previous work, we focus on queries that consume events of relative low rates, each computed in a single server (i.e. no scaling-out per query). However, we need to place a very large and dynamic number of queries in relatively few servers. Our focus is motivated by the need to support a platform for hosting end-user streaming queries that may come from a variety of applications, such as the Cortana personal assistant.

The placement strives to reduce network and computational overheads. It exploits the observation that a large number of queries consume the same sources of events, and, hence, placing them in the same server in the platform reduces network overheads. However, the placement also needs to balance the load among the servers. A further complication arises from the requirement to allow the queries to read events from multiple sources concurrently (i.e., to join multiple streams).

In this paper, we formulate the problem of placing queries into the servers of a streaming platform. We propose approximation algorithms and derive approximation bounds for the following cases (a) the offline case where queries are stable and known ahead of time, akin to an "oracle", and (b) 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

*Work performed while an intern with Microsoft Research; Bo Zong is with the University of California, Santa Barbara.

DEBS'15, June 29 - July 3, 2015, OSLO, Norway.

http://dx.doi.org/10.1145/2675743.2771825.

Keywords

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

1 Introduction

Complex Event Processing (CEP) has been popular for many applications [18]. Platforms such as S4 [30, 21], Storm [26], Photon [5], MillWheel [1] and Amazon's Kinesis [3], enable scalable data analytics over data streams. More recently, we are observing the rise of stream processing platforms that ingest slower-rate data streams, but allow more expressive operations. For example, the back-end that supports the Cortana personal assistant [19] executes, on behalf of its users, queries that monitor and process traffic, weather, news, and other events, and forwards events of interest to its users. Such queries typically: (a) have low network and compute overheads (e.g. processing weather updates for a region), (b) may join multiple streams, such as a personal event stream (e.g. calendar updates) with a global stream (e.g. traffic updates), to generate user events (e.g. reminder to leave to catch next appointment), (c) support a very expressive programming model (e.g. using UDF's expressed in LINQ [20], or JavaScript as in Amazon's Lambda [4]). Requirements (a) and (c) suggest that it is natural to execute each query in a single server (unlike systems that scale-out processing to deal with high stream data rates), but due to (c) the processing overhead is often not trivial to estimate from the query. Even though each query is rather "small", the expectation is that there will be a very large number of queries that need to be supported with low network and computational resources. Hence, the main challenge in building a platform to support a vast number of "small" queries is to allocate queries to servers such as to minimize network and compute overheads.

Figure 1 depicts a typical platform that hosts queries on behalf of users. External to the system, there are event sources that generate events of interest to the users; these can be events of general interest, such as news, weather, stocks, traffic, flight updates, and personalized events, such as calendar events, user location events, etc. Event gateways ingest events from each source, and then forward them to the internal processing nodes (*query evaluators*). It is important to observe that there is one or very few event gateways per event source, but then the gateway(s) forward the stream to all query evaluators that host user queries that depend on that event source, potentially by replicating the stream multiple times. The query evaluators execute their queries and forward the results to the end-users. In this paper, we study specifically the problem of minimizing the network overhead for forwarding the events from the event gateways to query evaluators while balancing the load among the query evaluators. We assume that the service is hosted in a

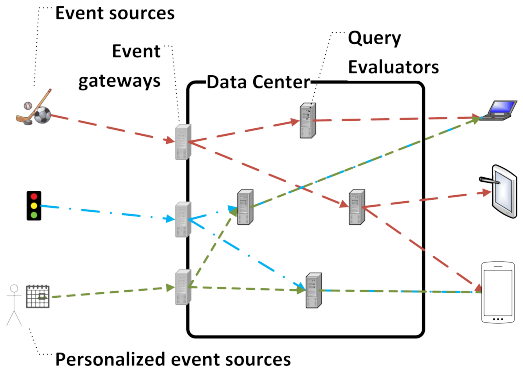

Figure 1: Example of events flowing from the event sources to the query evaluators, and finally to users. In addition to global events, there are also personalized event sources; in this example, a query combines traffic updates with calendar events to generate notifications for the user (e.g. time to leave to be on time for next appointment given current traffic).

generic compute platform, such as Azure Compute [29], Amazon's EC2 [2], or Rackspace [24]; this is a typical requirement for many modern services as it separates the management of the platform from the operation of the service. Hence, we cannot control the assignment of the query evaluators to the underlying platform, and cannot rely on an efficient transport (*e.g.,* multicast) of the events from the gateways to the evaluators. However, we can assume that each stream will be delivered at most once to each query evaluator, even when the evaluator serves multiple queries operating on that stream. Hence, there is a unique network connection per stream between the gateway and each of the query evaluators interested in the stream, and we want to minimize the network overheads incurred by those connections.

As a simple example, consider that all streams have the same popularity and that their aggregate rate is W (this is the rate from the event sources to the gateways). The aggregate rate W can be many 10's of Gbps, even though each individual stream is much smaller. The rate from the gateways to the evaluators will be at least W (best case). However, if there are k query evaluators, typically k is many 100's, a naïve allocation of queries of evaluators can result into W incoming rate per evaluator, which can violate the evaluator's capacity, and a total rate inside the platform of $k \cdot W$, which incurs significant network load.

Our approach strives to minimize total network load, and at the same time be mindful of capacity constraints and processing overheads incurred when executing the queries. These are non-trivial because the platform allows a flexible query processing programming model. In other words, a solution that places all queries using the same stream to the same query evaluator does not scale for popular queries and streams. One approach to capture this requirement would be to associate capacity limits for all critical resources of the system (*e.g.,* network bandwidth, processing and memory demands per server). However, this approach requires a priori knowledge of those limits, and in our experience may result in very unbalanced allocations that are not desirable from an operational point of view. Instead, our allocation strives to balance the load in the system *and* reduce network overheads.

The main insight that enables us to optimize the allocation of queries to evaluators is the observation 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. This is akin to the existence of many applications for *e.g.,* smart-phones that present weather, news, and other information to end-users using the same sources, and the large number of users for many of these similar applications. Hence, we anticipate large benefits by co-locating similar queries to the same query evaluator, and transporting the relevant network streams once.

Three practical requirements complicate query assignment. 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 an exception. Hence, the platform must make a good decision when assigning a query to a server, *upon* query arrival. 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 query evaluators (server), 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:

- Formulation of 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.
- We show that the problem of reducing network load and balancing server load is NP complete (Section 3), and provide approximation algorithms for the given problem (Section 4).
- We formulate and analyze offline (Section 4) and online (Section 5) heuristics for the given problem. Offline heuristics assume an oracle that knows ahead of time all queries. We use the offline heuristics to reason about the performance of the query assignment policies, and to draw inspiration for the online query assignment 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 System Model and Assumptions

We consider a stream processing platform with the following three key components (see also Figure 1): (event) *sources*, *queries*, and *servers* (query evaluators).

Sources. A source is a publisher that generates new events as a data stream at some rate. $S = \{s_1, s_2, \ldots, s_m\}$ denotes a set of sources, and $w(s)$ is the rate of events published by source $s \in S$. The total event rate for a subset of sources $S' \subseteq S$ is denoted by $w(S') = \sum_{s \in S'} w(s)$. For convenience, and without loss of generality, we shall treat sources as if located in the event gateways.

Queries. A query subscribes to one or multiple sources and processes the events originating from the sources. A set of queries is denoted by $Q = \{q_1, q_2, \ldots, q_n\}$, and the set of sources subscribed by a query $q \in Q$ is represented as S_q, which is a subset of the set of sources S.

If two queries subscribe to identical sets of sources, we say that they are of the same *query type*. Let $T \subseteq 2^S$ be the complete set of distinct query types. Each query $q \in Q$ has a query type $t \in T$,

where $t = S_q$. Given n queries, we have $n = \sum_{t \in T} n_t$, where n_t is the number of queries of query type t.

Moreover, $N(s)$ denotes the subset of queries that subscribe to source s, that is, $N(s) = \{q \in Q \mid s \in S_q\}$.

Servers. A server is a container that evaluates queries. We assume there are $k \geq 1$ servers in a stream processing platform. (The terms "server" and "query evaluator" are used interchangeably; we mostly use "server" for brevity.)

2.1 Query Assignment Problem

We consider optimizing query assignment with respect to the following two criteria: network traffic and server load.

Network Traffic. We are interested in minimizing network traffic between sources and servers. If a server hosts at least one query that subscribes to a source s, then this contributes $w(s)$ to the network traffic cost. This implies that it is desirable to co-locate queries that use the same source(s). Formally, the total network traffic cost of a server that hosts queries $Q' \subseteq Q$ is defined as

$$f(Q') = \sum_{s \in S} w(s) \cdot \mathbb{1}\{s \text{ is required by some } q \in Q'\}.$$

The total network traffic of an assignment of queries to servers according to the sets of queries $Q_1, Q_2, \ldots, Q_k$ assigned to respective servers $1, 2, \ldots, k$ is given by: $\sum_{j=1}^{k} f(Q_j)$.

Server Load. A query assignment is feasible if it approximately balances the processing load of servers up to a given slackness. In practice, it is non trivial to quantify the capacity of a server. Servers are typically hosted by virtual machines in a cloud service platform. The capacity of a server depends on several factors including the processing requirements of queries, and other factors such as the load of virtual machine to which the server is assigned. Therefore, we aim at balancing the load over different servers which does not require knowing exact capacities of individual servers. The underlying assumption is that the system operates at a load that allows for a feasible query assignment. In this case, balancing the load across different servers is a natural objective.

In our analysis, we assume that each query contributes a fixed processing load to the server it is assigned to. For simplicity of exposition, we assume that queries contribute identical processing loads, say of unit value, but our results naturally generalize to non identical query processing loads. In this way, the processing load of a server corresponds to the number of queries assigned to this server. This is used as a proxy for the load of a server to allow for a formal analysis. In an implementation in practice, one may redefine the load of a server to be some suitable metric indicating the load of the server, possibly accounting for the load of the virtual machine to which the server is placed to.

Given a *slackness parameter* $\nu \geq 0$ for balancing the load across servers, a query assignment to k servers is specified by a partitioning of the set of queries Q into k disjoint subsets $Q_1, Q_2, \ldots, Q_k$. A query assignment is said to be *ν-load balanced*, if it satisfies the following condition:

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

Throughout the paper, we interchangeably refer to the parameter ν as the slackness parameter or *relative relaxation* parameter.

Query Assignment Problem (QA) Let $\mathscr{P}(Q)$ be the set of all possible k-partitions $Q_1, Q_2, \ldots, Q_k$ of the set of queries Q such that (a) $Q_1 \cup Q_2 \cup \cdots \cup Q_k = Q$; and (b) $Q_i \cap Q_j = \emptyset$ for every $i \neq j$.

The QA problem is defined as follows: given a set of sources S, a set of queries Q, a set of k servers, and a slackness parameter $\nu \geq 0$, find $(Q_1, Q_2, \ldots, Q_k) \in \mathscr{P}(Q)$ which minimizes the network traffic

$$\sum_{j=1}^{k} f(Q_j)$$

subject to the server load balancing constraints (1).

3 Hardness and Benchmark

In this section, we first discuss the computational complexity of the query assignment problem (QA), and then characterize the inefficiency of a standard load balancing strategy that assigns each query by sampling a server uniformly at random.

3.1 NP Hardness

In general, it is computationally hard to find an optimal solution for an arbitrary QA instance in polynomial time as showed in the following theorem.

THEOREM 1. *Query assignment problem is NP-complete.*

PROOF. The proof consists of two steps: (a) we first prove QA's decision problem is NP-hard; and (b) show that it is NP.

First, we prove its NP-hardness by a reduction from the well-known bin packing problem [13].

The decision problem of QA is described as follows: Given the set of sources S, the set of queries Q, k servers, a slackness parameter $\nu \geq 0$, and a real number $\gamma \geq 0$, does there exist $(Q_1, Q_2, \ldots, Q_k) \in \mathscr{P}(Q)$ such that (a) $\sum_{j=1}^{k} f(Q_j) \leq \gamma$, and (b) every server is ν-load balanced?

Consider a special case of QA's decision problem, where (a) each query subscribes to exactly one source, (b) each source publishes events at a unit rate $w(s) = 1$ for every $s \in S$, and (c) $\gamma = |S|$. 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 to an instance of the special case under consideration: (a) we reduce an item to a query type, where the item's size is reduced to the number of queries for the corresponding type; (b) the number of bins is reduced to the number servers; and (c) 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 QA's decision problem is NP-hard.

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

The following competitive ratio holds for every feasible query assignment.

PROPOSITION 1. *For every assignment of queries to servers, the network traffic cost is at most k times the optimum network traffic cost, where k is the number of servers.*

3.2 Random Query Assignment

A naïve query assignment strategy is to assign each query to a server sampled independently, uniformly at random. This is a standard load balancing strategy that can be implemented by hash partition of query identifiers. This strategy can efficiently balance the number of queries over servers. Specifically, it is known to guarantee the maximum server load of $n/k + O\left(\sqrt{(n \log k)/k}\right)$ with probability $o(1)$ for $n \gg k \log^3 k$ [23]. However, this strategy can be grossly inefficient with respect to network traffic cost, which we show analytically below and also experimentally in Section 6.

PROPOSITION 2. *Consider the set of sources S ($|S| = m$) and k servers, with d_s denoting the number of queries subscribed to source $s \in S$. The expected network traffic cost under uniform random query assignment strategy is*

$$\left(1 - \frac{1}{m}\sum_{s\in S}\left(1-\frac{1}{k}\right)^{d_s}\right)km. \tag{2}$$

PROOF. Consider an arbitrary source $s \in S$ and an arbitrary server $j \in K$. Under random query assignment, at least one query that requires input from source s is assigned to 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. Summing further over all sources $s \in S$ gives the expected number of streams that need to be transferred from sources to servers, which corresponds to total network traffic. □

Proposition 2 implies that the naïve strategy can easily achieve the upper bound in Proposition 1 and result in a large amount of network traffic cost. From Equation 2, the expected network traffic cost of random query assignment is nearly equal to the worst-case network cost whenever $\frac{1}{m}\sum_{s\in S}(1-1/k)^{d_s} \ll 1$. In fact, the worst-case network traffic cost is achievable under the random query assignment policy: consider m sources and n queries 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 only subscribes to the sources in S_j and none in $S \setminus S_j$. The subscription between queries and sources corresponds to a collection of k disconnected complete bipartite graphs, each of which has m/k sources and n/k queries. In this case, 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. On the other hand, the best strategy in this case is to assign each piece of queries to a distinct server, which achieves minimal network traffic and perfect load balancing. Note that the inefficiency of the naïve strategy can be made arbitrarily large by taking k large enough.

The high network traffic cost caused by naïve strategies such as random query assignment asks for the design of more sophisticated query assignment algorithms. In the next section, we focus on offline QA problem, and propose approximation algorithms with better performance guarantees. In Section 5, we discuss online algorithms that irrevocably assign queries to servers at their arrival.

4 Offline Query Assignment

In this section, we present approximation algorithms for offline QA problem. We first consider approximation algorithms for the general case where each query subscribes to one or multiple sources, which we refer to as *multi-source* QA. For sources with identical event rates, we propose an approximation algorithm that guarantees the network traffic cost of at most $2d_{max}(1+\log k)$ of the optimal network traffic cost, where d_{max} is the maximum number of sources required as input to a query, and k is the number of servers. Since the value of d_{max} is usually a small constant in practice [5], this is a much tighter bound compared with the worst-case bound of k. We also develop several query assignment heuristics that are observed to exhibit comparable performance for some workloads later in Section 6. We then go on to present a 2-approximation algorithm for the special case where each query subscribes to exactly one source, which we refer to as *single-source* QA.

4.1 Multi-Source Query Assignment

We first present an approximation algorithm and then introduce two heuristics for offline multi-source QA.

4.1.1 Minimum Query Type Packing

In this section, we establish the following main theorem.

THEOREM 2. *Suppose that all sources have identical event rates. There exists a polynomial-time algorithm for multi-source* QA *with the approximation ratio*

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

where $d_{max} = \max_{q\in Q}|S_q|$ is the maximum number of sources subscribed by a query. Furthermore, the same bound holds for sources of arbitrary rates with an extra factor $\omega = \max_{s\in S} w(s)/\min_{s\in S} w(s)$.

Theorem 2 tells us that when queries subscribe to a number of sources that is bounded by a constant, we have an approximation guarantee of $O(\log(k))$ where k is the number of servers. This is of practical interest in applications where each query subscribes to a few sources. The result in Theorem 2 is established by an algorithm that runs through multiple rounds and in each requires to solve the following packing problem.

Minimum Query Type Packing (MQP): given is as input a set of queries Q, a set of sources S, and a real number $\theta > 0$, and the goal is to find a subset of query types $T' \subseteq T$ that minimizes

$$w\left(\bigcup_{q\in Q:\ S_q\in T'} S_q\right)$$

subject to the constraint

$$\sum_{t\in T'} n_t \geq \theta.$$

The multi-source QA approximation algorithm is defined by the following operations performed in each round r:

1. Select an empty server as the target for query assignment from the pool of unassigned queries $Q^{(r)}$.

2. Find an optimal subset of query types $T' \subseteq T$ for the MQP problem with the set of input queries $Q^{(r)}$ and

$$\theta = n - (1+\nu)\frac{(k-1)n}{k}. \tag{3}$$

   Let $\hat{Q} \subseteq Q^{(r)}$ be the subset of queries of query types in T'. Assign $\hat{Q}$ to the selected server.

3. If $|\hat{Q}| > (1+\nu)n/k$, we arbitrarily select a query type $t' \in T'$, remove some number of queries of type t' to make the server ν-load balanced, and put them back to the set of unassigned queries. Note that since $\theta < (1+\nu)n/k$, we only need to select one query type for query removal. If we need to select more than one query type, T' cannot be an optimal solution.

The proof of Theorem 2 follows by the following three claims: (a) if in the multi-source QA approximation algorithm presented above, we can optimally solve the MQP in each round of the algorithm, then we have a $2(1+\log k)$ approximation; (b) MQP is NP-complete; (c) MQP can be solved approximately within factor d_{max} in polynomial time, which are established next in Lemma 1, Lemma 2, and Lemma 3, respectively.

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 network traffic cost, and OPT be the optimal solution for the multi-source QA problem.

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

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

PROOF. The proof is provided in Appendix A. □

In Lemma 1, we derived an approximate algorithm for the QA problem under assumption of the existence of an oracle that provides optimal solutions to MQP problems. We next show that MQP is NP-complete; therefore, it is hard to find a polynomial-time algorithm that solves MQP optimally.

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

PROOF. We sketch the proof as follows. (a) To prove NP-hardness, we can reduce the NP-hard minimum k-union problem [28] to single-server MQP problem. (b) It is easy to verify a solution in polynomial time. □

We define the following algorithm for the MQP problem:

1. Order query types in decreasing order with respect to the number of queries;
2. Successively pick a query type with the largest number of queries until the number of assigned queries is at least θ.

LEMMA 3. *Suppose sources have identical event rates. The above algorithm approximates* MQP *within $d_{max} = \max_{q \in Q} |S_q|$.*

For arbitrary source rates with $\omega = \max_{s \in S} w(s) / \min_{s \in S} w(s)$, the above algorithm approximates MQP *within $d_{max}\omega$.*

PROOF. In the above algorithm, we pick the query types with the largest number of queries to saturate a server. Suppose we eventually select h query types, we then 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 takes at most d_{max} times more of the network traffic rate compared with the optimal solution, when the source traffic rates are identical. In the case of arbitrary source traffic rates with the ratio of the maximum source traffic rate to the minimum source traffic rate at most ω, it takes at most $d_{max}\omega$ times more network traffic rate. This completes the proof of the lemma. □

4.1.2 Heuristics

In this section we present two heuristics for the offline QA, including *incremental cost* (referred to as IC) and *min-max traffic cost per server* (referred to as MMS).

Incremental Cost Based Approach. IC assigns queries in successive rounds. At each round, it assigns queries to a server in three steps. (a) Given a query type $t \in T$ with non-zero number of unassigned queries and a server j of spare capacity to host more queries, we consider the *incremental traffic cost* resulting from assigning at least one query of type t to server j. (b) We select a pair of a query type and a server (t^*, j^*) that results in *the least incremental cost*. (c) We assign queries of type t^* to server j^* as many as possible until server j^* is full or there are no unassigned queries of type t^*.

Min-max Traffic Cost per Server. MMS aims to minimize the maximum traffic cost among servers in successive rounds. At each round, it assigns queries to servers in three steps. (a) Given a query type $t \in T$ with non-zero number of unassigned queries and a server j with spare capacity to host queries, we consider the *traffic cost* after assigning at least one query of type t to server j. (b) We select a pair of a query type and a server (t^*, j^*) that results in *the least traffic cost*. (c) We assign as many as possible queries of type t^* to server j^* until server j^* is full or there are no unassigned queries of type t^*.

4.2 Single-Source Query Assignment

In this section, we consider single-source QA, where each query subscribes to exactly one source. In this case, there is a one-to-one correspondence between query types and sources. Therefore, we use the term source and the term query type interchangeably.

We present an approximation algorithm for single-source QA that assigns queries to servers over successive rounds as shown in Figure 2. For server j, let d_j to be the spare capacity of server j. At the beginning of the first round of the algorithm, we initialize $d_j = \lfloor (1+\nu)n/k \rfloor$, where $\nu \geq 0$ is the slackness parameter. Let n_s be the number of unassigned queries that subscribe to source s.

Input: A single-source QA instance;
Output: A query assignment.

```
1. while True
2.     Select server j with the largest free capacity
3.     Select query type (source) s of the largest event rate w(s)
4.     b ← min(d_j, n_s)
5.     Assign b type-s queries to server j
6.     d_j ← d_j − b
7.     n_s ← n_s − b
8.     if there no queries left for assignment
9.         return
```

Figure 2: 2-approximation for single-source QA.

THEOREM 3. *The approximation algorithm given in Figure 2 has the following approximation guarantees:*

1. *The approximation ratio $1 + k/m \leq 2$, where m is the number of sources and k is the number of servers, for sources with identical event rates;*
2. *The approximation ratio 2, for sources with arbitrary event rates.*

PROOF. We consider only the case where $0 < n_s < d_j$, for $d_j = \lfloor (1+\nu)n/k \rfloor$, for every $s \in S$ and $j = 1, 2, \ldots, k$ as the other cases can be reduced to this case. Indeed, if there exists a source s' such that $n_{s'} \geq d_j$, then we can reserve a server for source s', reduce $n_{s'}$ by d_j, remove the server, and repeat, which is an optimal assignment. We also assume that $k < m$. Otherwise, since $0 < n_s < d_j$ for every source s and server j, an optimal assignment is to allocate each query type to a distinct server.

Identical Source Event Rates. Without loss of generality, we assume $w(s) = 1$, $\forall s \in S$. 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 every source is required by at least one server in the system.

The upper bound for the above algorithm is $k + m$. In each round, we either make a query type consume all the spare space of

a server, or make a server host 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. It follows that the number of rounds is at most $k+m$. Since each round increases network traffic rate by at most 1, the total traffic rate cannot be larger than $k+m$.

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

Arbitrary Source Event Rates. Similarly, we demonstrate a lower bound for the optimal solution and an upper bound for the approximate solution.

The lower bound for the optimal solution is $w(S)$, since we have to send the data stream of each source to a server at least once.

The upper bound for the approximate solution is $2w(S)$, which we show next. Let r_s be the number of rounds in which queries of query type s are assigned. In these rounds, we add a network traffic cost in the system of value $r_s \cdot w(s)$. By the same argument as for the case of identical source event rates, the total number of rounds is at most $m+k$, i.e. we have $\sum_{s\in S} r_s \le k+m$. From this, it follows that $\sum_{s\in S}(r_s-1) \le k$. Let w_{k+1} denote the $(k+1)$-th largest traffic rate among all query types. Since $0 < n_s < d_j$ for every query type s and server j, we can guarantee that the top-k query types with respect to the traffic rate are assigned at most once. Therefore,

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

The total source publishing event 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 provides a 2-approximation. □

Remark. For single-source QA, 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 the source event rates are identical, then the approximation ratio of $1+k/m$ can be guaranteed. Thus, this approximation ratio guarantee can be arbitrarily near to the optimal one whenever the number of sources relative to the number of servers is sufficiently large. In a practical system with single-source queries and many sources of approximately identical event rates, the proposed algorithm can guarantee nearly optimal performance.

5 Online Query Assignment

In this section, we consider online QA, where each query is irrevocably assigned to a server at its arrival time. We focus on the class of online algorithms that decide which server to host an incoming query based on (a) the set of sources required by an incoming query and (b) 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 for assigning queries to servers. Such a metric should consider both load balancing and network traffic cost. We introduce and discuss several metrics for online query assignment in Section 5.1. Moreover, in Section 5.2, we discuss how to make use of the 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 the design of these online algorithms.

Input: (a) an incoming query q requiring a set of sources S_q;
(b) k servers and the queries they are hosting $Q_1,\ldots,Q_k$;
(c) relative relaxation ratio ν;
(d) a predefined metric M;
Output: the server that will host q.

1. Find candidate servers $C = \{i \mid |Q_i|+1 < (1+\nu)\frac{\sum_{j=1}^{k}|Q_j|}{k}\}$
2. Find $C^* \subseteq C$ such that $\forall i \in C^*$,
3. $\quad M(Q_i,q) \le M(Q_j,q), \forall j \in C$
4. **if** $|C^*| == 1$
5. $\quad$ **return** the only server in C^*
6. **else**
7. $\quad$ **return** the server $p = \text{argmin}_{i\in C^*}\{|Q_i|\}$

Figure 3: Greedy algorithms for online QA

The three algorithms use different metrics to decide which server to host an incoming query; however, they share the common pipeline as shown in Figure 3. Given an incoming query q, k servers along with the corresponding set of queries a server is hosting, the relative relaxation ratio ν, and a predefined metric M, the server to host q is decided as follows. (a) 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. (b) From C, we find the servers C^* each of which with the lowest cost in terms of M if we add q into the server. (c) 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: (a) *least incremental cost first* (referred to as LeastCost), (b) *least source cost per server first* (referred to as LeastSource), and (c) *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. The least incremental cost metric is defined as

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

This is a natural metric for QA because the goal of QA is to minimize traffic cost in a system, and LeastCost attempts to achieve this goal by locally minimizing incremental traffic cost at each query arrival.

LeastSource. The least source cost per server metric is defined as

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

This metric has a potential issue: one server might subscribe to many sources because of locally optimal decisions such that many incoming queries are assigned to the server, the server gets 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 can 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 $\mathscr{T}(Q_j)$ be the set of query types such that $\forall t \in \mathscr{T}(Q_j)$ there exists at least one query of type t hosted by server j. The least query type metric is defined as

$$M_{\mathsf{LeastQT}}(Q_j,q) = |\mathscr{T}(Q_j \cup \{q\})|.$$

There is an issue common to both LeastCost and LeastSource: 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 QA. 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. In practice, we might know some statistical information about queries, and may relax the load balancing constraint at specific conditions.

5.2.1 Known Query Type Distribution

When we deal with online QA, 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 a production system that has 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 known popularity of query types: the probability that an incoming query is of type t is λ_t, for $t \in T$, where $T \subseteq 2^S$ is the universe of 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 would do. Given $(\lambda_t,\ t \in T)$, we reduce an online QA to an offline QA 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 server load balance constraint, the number of queries a server could host at most, and in particular, we set $\delta_j = 1/k$ 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 QA.

5.2.2 Relaxed Load Balancing Constraints

QA problem is defined as a bi-criteria optimization problem where one of the criteria is balancing the load of servers. Specifically, the problem corresponds to finding a query assignment 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 QA, requiring to obey this condition at each query assignment instance may be too restrictive and result in sub-optimal query assignments with respect to the long-term network traffic cost.

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

To resolve the above problem, we relax the load balancing constraints as follows. (a) We define another balance constraint for a system in its initial phase, and use *absolute slackness parameter* to control the balance constraint. (b) In the initial phase, a system uses a balance constraint decided by absolute slackness parameter. When the system hosts more than n queries, we switch back to the balance constraint decided by relative slackness parameter. 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 slackness parameter. The system is said to be $(1+\nu)$-relatively balanced, if the number of queries in any server is no more than $(1+\nu)\cdot n/k$, where $\nu \geq 0$ is the relative slackness parameter.

In an online system, when the number of input queries is small, we apply absolute slackness parameter to balance the workload of servers; when the number of queries becomes sufficiently large, we switch to relative slackness parameter. 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

$$d(n) = \max\left\{\frac{n}{k}+\alpha, (1+\nu)\cdot\frac{n}{k}\right\}. \quad (4)$$

The system switches to the relative load balancing as soon as the number of input queries satisfies $n \geq (\alpha/\nu)k$. The configuration of relaxed load balancing is discussed in Appendix B.

6 Experimental Evaluation

In this section we present performance evaluation of the offline and online algorithms in Section 4 and Section 5 by an extensive set of simulations and using data from production system. Overall, our experimental evaluations provide support to the following claims:

1. Optimizing query assignment provides significant reduction of network traffic compared to random query assignment.
2. Specific online query assignment heuristic, namely LeastCost, consistently outperforms other online (and sometimes even offline) heuristics for a wide range of configurations.
3. LeastCost scales with respect to the number of queries, sources, and servers, and it is robust to dynamic arrival and departure of queries and servers.

6.1 Synthetic Workloads

We generated subscription of queries to sources according to a random bipartite graph model [15]. The subscriptions of queries to sources are 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 input from source s. G is assumed to be a random bipartite graph with given degree distributions for the vertices that represent sources and the vertices that represent queries. Specifically, we consider (a) the degrees of source vertices according to Zipf distribution with power-law exponent $\beta > 0$, and (b) the degree of query vertices fixed to parameter $d > 0$. The popularity of sources typically follow a power-law distribution [15, 16], which is modeled by a Zipf distribution. In our experiments, we consider queries of unit processing costs.

Offline Algorithms. We evaluate performance of incremental cost based approach IC, minimum query packing based approach MQP, and min-max traffic cost per server MMS, which are defined in Section 4. As a baseline for comparison, we consider the following

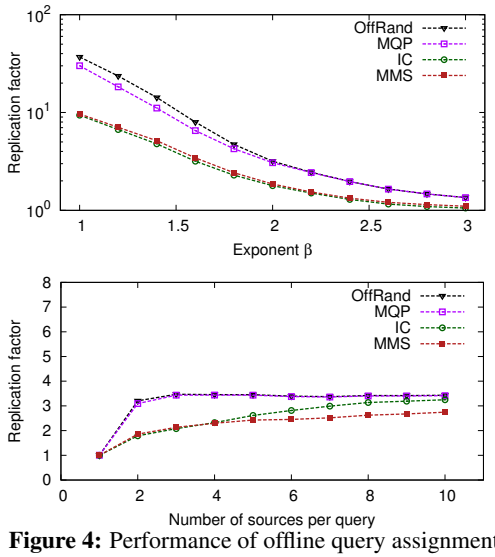

Figure 4: Performance of offline query assignment.

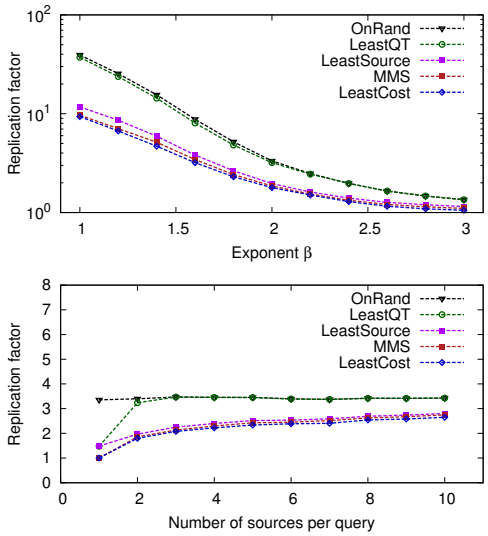

Figure 5: Performance of online query assignment without query departures.

random offline assignment heuristic OffRand: (a) randomly select all queries of the same query type; (b) randomly select a server with available space to host queries; (c) assign queries to the selected server until either the server is saturated or all queries (of query type) have been assigned; (d) remove server or query type from further consideration; and (e) repeat from step (a) until all queries are assigned.

Online Algorithms. We evaluate the following online heuristics: (a) Least incremental traffic first LeastCost, (b) Least number of sources first LeastSource, and (c) Least query types first LeastQT. As a baseline for comparison, we consider the following random online query assignment OnRand: given an input query, we find a set of candidate servers that can accept the new query without violating the load balancing constraints, and then randomly select one of servers from this for assignment.

Parameters. In our experiments, we consider four parameters: (a) the power-law exponent β for the source degree distribution in the range 1.0 to 3.0 with a default value 2.0, (b) the number of sources per query in the range from 1 to 10 with a default value 2, (c) the number of servers in the range from 10 to 1000 with a default value 100; and (d) the number of queries in the range from $10,000$ to $1,000,000$ with a default value of $100,000$.

When considering query or server dynamics, we also have the following two parameters: (a) mean query life-time (Figure 6.1.2), and (b) server departure rate (Figure 6.1.2).

For all the offline algorithms considered, we fix the relative relaxation parameter ν to value 0.05. For all the online algorithms considered, we fix the relative relaxation parameter ν to value 0.05, and the absolute relaxation parameter α to value 10 (Section 5.2.2).

Performance metrics. We consider the *replication factor* as the metric to evaluate the performance of algorithms. Let C be the resulting traffic cost of an algorithm, S be the set of sources with traffic cost w, and $f(S) = \sum_{s \in S} w(s)$. The replication factor of the algorithm is defined as $C/f(S)$. Intuitively, the replication factor represents normalized traffic cost under the given algorithm.

We run each configuration 10 times, and average the results.

6.1.1 *Offline Algorithms*

We examine the replication factor of the offline algorithms in Figure 4. These results are for sources with unit publishing rates; we examine heterogeneous source traffic rates in Figure 6.1.2.

In the top graph in Figure 4, we show the network traffic replication factor versus the power-law exponent of the Zipf distribution of the number of queries subscribed to a source (source popularity). The number of sources per query is fixed to 2, the number of queries is set to $100,000$, and the number of servers is fixed to 100. We observe that the larger the power-law exponent, the better the performance. In other words, the heavier the power-law distribution of the number of queries subscribed to a source, the larger the replication factor. This is intuitive as a heavier tail implies the existence of a few sources with many query subscribed to those sources, which would intuitively make the query assignment problem harder. The query assignment heuristics IC and MMS consistently exhibit the best performance, up to four times better than OffRand.

In the bottom graph in Figure 4, we show the network traffic replication factor versus the number of sources per query. In particular, the power-law exponent is fixed to value 2.0, the number of queries is set to $100,000$, and the number of servers is set to 100. We observe that the replication factor increases with the number of sources per query. For single-source queries, the replication factor is smaller than or equal to 2, which provides experimental confirmation of the theoretical guarantee in Theorem 3. The replication factor exhibits a diminishing returns increase with the number of sources per query. MMS exhibits the best performance, and this is matched by IC for sufficiently small number of sources per query.

We also examined the network traffic replication factor versus the number of servers, which is not presented for space reasons. The results suggest that the replication factor increases with the number of servers logarithmically. We also observed that the network traffic replication factor is largely invariant to the number of queries, which we also omit to show due to space reasons.

In summary, we observed that MMS consistently outperforms other offline algorithms, and results in a performance gain of up to factor 4 compared with random offline query assignment OffRand.

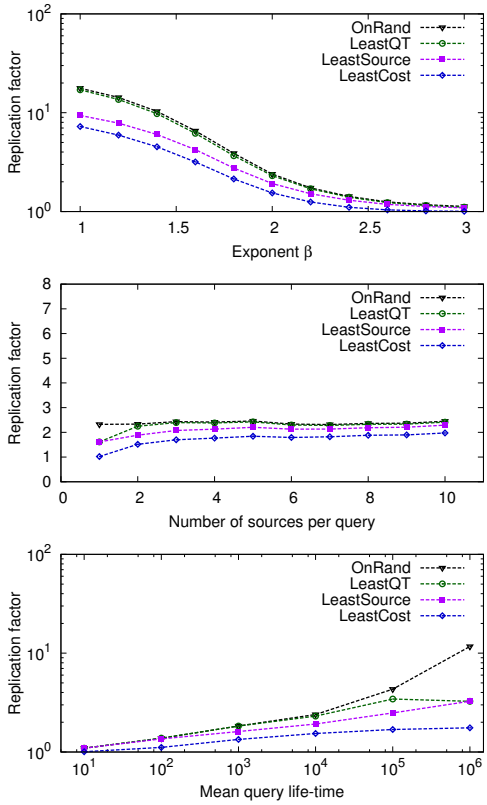

Figure 6: Performance of online algorithms with dynamic query arrivals and departures.

6.1.2 *Online Algorithms*

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

Query Arrivals, No Departures In Figure 5, show the network traffic replication factor for online algorithms, in the same settings as in Figure 5. The two sets of graphs are overall qualitatively very similar, hence, we only discuss differences.

We observe that LeastCost exhibits the best performance, and sometimes even outperforms the best offline algorithm MMS. The performance of LeastSource is close to LeastCost. LeastCost performs up to four 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. Typically, LeastQT provides no benefits.

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. We consider query dynamics according to the following model: (a) at each time step, the number of query arrivals is a random variable with a Poisson distribution; and (b) each arriving query has a lifetime, drawn independently from a given distribution. In particular, we consider two parametric families of distributions: (a) exponential distribution that models the cases of light-tailed query lifetimes, and (b) Pareto distribution that models the case of heavy-tailed query lifetimes. We found similar results for these two different families of distributions, so we only present the results for the exponential distribution.

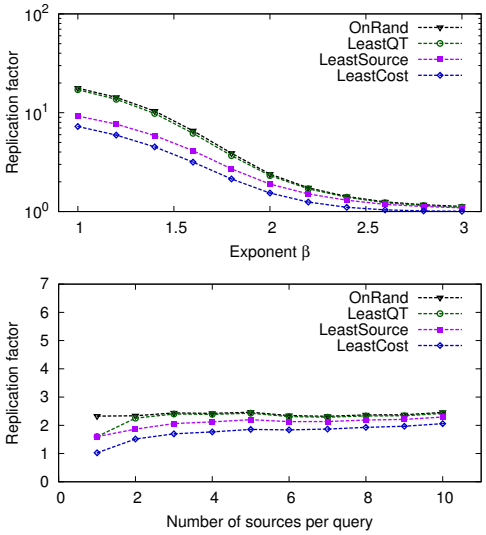

Figure 7: Performance of online algorithms with dynamic query and server arrivals/departures.

Figure 6 shows the performance of online algorithms under dynamic query arrival and departure. The mean query lifetime ranges from 10 to $1,000,000$ time steps with the default value $100,000$, the average number of query arrivals per time step is 1, and the total number of discrete time steps τ is set to be $1,000,000$ in all cases. Overall, we observe that LeastCost consistently yields the best performance. 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, thus, the given range covers the mean number of queries in the system from 10 to $1,000,000$ queries. Given that the number of servers is 100, we cover the system operating points of 0.1 to $10,000$ queries per server, which covers the range of lightly to highly loaded servers. The result indicate that the network traffic replication factor tends to increase with the load of the servers for all online algorithms. However, this increase is rather slow for LeastCost, which is sub-linear in the mean query lifetime.

Server Arrivals and Departures In practice, we also expect some level of 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 strategies with respect to arrivals and departures of servers.

We modeled server dynamics similarly to query dynamics: we start with k servers, and queries arrive in τ time steps. At each time step, the number of query arrivals is a random variable with a Poisson distribution, and each query is associated with a lifetime that is a random variable with an exponential distribution. 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).

The results in Figure 7 demonstrate the performance of online query assignment under both query and server dynamics. By default, we consider 100 servers, the power-law exponent β for

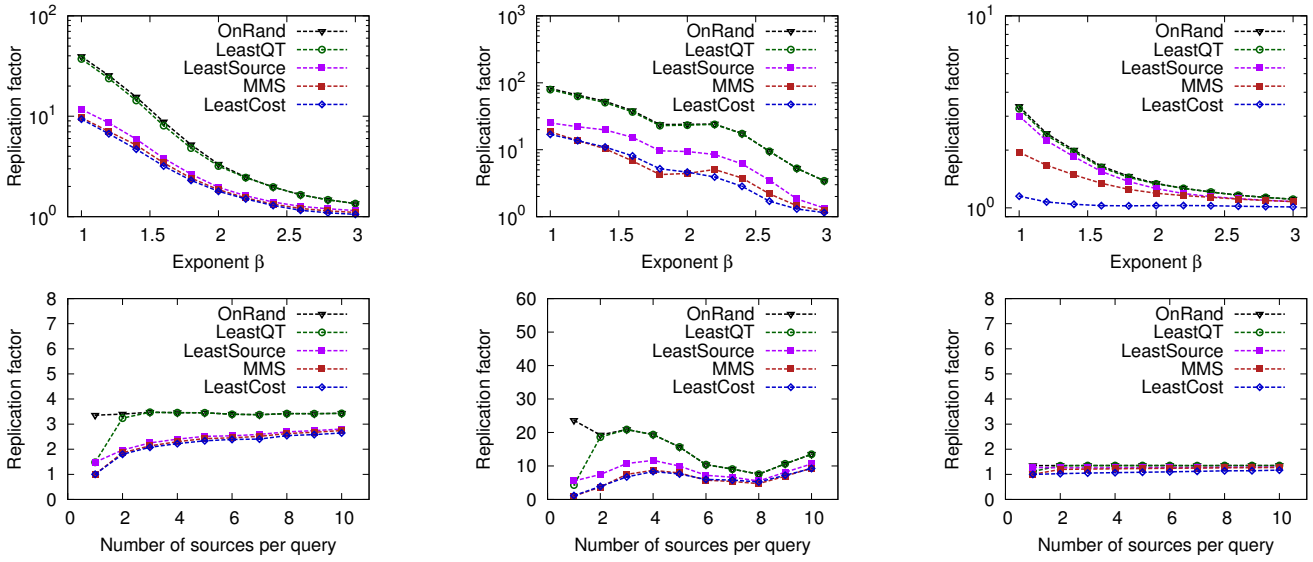

Figure 8: Performance of query assignment algorithms on heterogeneous source traffic rates: (left) random, (middle) positively correlated, and (right) negatively correlated.

source popularity distribution of value 2.0, the number of sources per query of value 2, the total number of time steps is $1,000,000$, the mean number of query arrivals per time step of value 1, the mean query lifetime of value $100,000$, and the server departure rate γ is $10,000$. Consistent with the results presented earlier, we observe that LeastCost exhibits the best performance, and is robust with respect to the dynamics of the server arrival and departure.

Heterogeneous Source Traffic Rates Thus far, we have examined the performance of query assignment algorithms for the case of sources with identical traffic rates. We now examine the case of heterogeneous source traffic rates. Since many phenomena in nature follow a power-law distribution [16], we assume that the source traffic rates follow a power-law distribution. We consider the range of values of the power-law parameter that span the case of a fast decaying tail (exponent value of 3) and a 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: (a) *random*: source traffic rates are assigned to sources independently of their popularity, (b) *positively correlated*: the traffic rate of a source is proportional to its popularity, and (c) *negatively correlated*: the traffic rate of a source is inversely proportional to its popularity.

The results are presented in Figure 8. In particular, we discuss the best offline algorithm MMS, all three online algorithms, and the online random algorithm OnRand, and note the following observations: (a) The more positive the correlation between the source traffic rates and the popularity of sources is, the larger the network traffic replication. (b) Typically, the best performing query assignment strategy is LeastCost. In the case when the source traffic rates are negatively correlated with popularity of sources, LeastCost is substantially better than MMS.

6.2 Real-World Workloads

We compare the performance of LeastCost query assignment strategy using traces from a production environment with two alternative query assignment strategies: the "worst-case" that amounts to supplying each stream of a source to every server, and round-robin that assigns each query according to the round-robin policy (whose performance is expected to be essentially that of random query assignment, which we studied in Section 3.2). The trace contains information about query arrivals over a week long interval collected in April 2014 from a production deployment of a stream processing query platform with 100's of servers. The results in Figure 9 show that round-robin query assignment strategy performs nearly as badly as the worst-case, and that a significant reduction of the network traffic cost can be achieved by LeastCost query assignment strategy. Specifically, the network traffic cost under LeastCost query assignment is observed to be approximately only 11% of that under the round-robin query assignment.

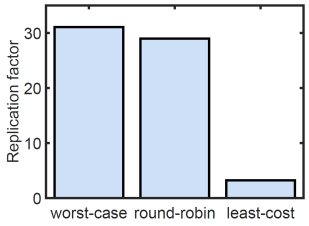

Figure 9: Performance of three different online query assignment strategies for a real-world workload.

7 Related Work

The query assignment problem studied in this paper aims at clustering similar queries together so as to minimize the network traffic and balance the load of servers. A variety of formulations of co-clustering of similar vertices of a graph was studied in previous work under various assumptions, e.g. see [12] for a survey. In particular, it was studied in the context of publish/subscribe systems, e.g. [22], [31], and [14]. However, to the best of knowledge, none of the previous work addressed the query assignment problem as formulated in this paper.

The problem of assigning tasks to machines to balance the load is a well-known problem, see [6] for a survey of online algorithms. Specifically, it is known that online greedy assignment provides a $2-1/k$ approximation. The problem of assigning *balls into bins* was also studied by various authors, *e.g.,* see [8, 23, 9] and the references therein. A standard objective here is to minimize the maximum load (minimize congestion), e.g. [7], and packing under knapsack constraints, e.g. [25, 11]. The uniform random assignment load balancing strategy is known to have the maximum load of $n/k + O(\sqrt{(n \log k)/k})$ with probability $o(1)$, for $n \gg k(\log k)^3$ [23]. Other load balancing strategies have also been studied, e.g. power of two choices, where each ball is assigned to the least loaded out of two bins picked uniformly at random for each assignment of a ball: the maximum load is known to be $n/k + O(\log \log k)$, with high probability, for $n \gg k$ [9]. A related work is online bin packing where the bounds on the competitive ratio with respect to the offline solution were derived for input arrival order according to random permutation or independent and identically distributed sequence, *e.g.,* see [11] and the references therein. Our main difference is that we consider a bi-criteria optimization, where the server load balancing is only one of the two criteria.

The query assignment problem studied in this paper is an instance of a non-standard 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 the set of vertices into k partitions such that each contains at most $(1+\nu)n/k$ vertices and the number of edges cut is minimized. The best known approximation ratio for this problem is $O(\sqrt{\log n \log k})$ [17]. The query assignment problem has the same kind of constraints. However, the objective function is a different submodular function. The unconstrained problem of minimizing a submodular function in the context of graph partitioning was considered, *e.g.,*, by [10], who derived a 2-approximation algorithm. A notable difference with our work is that the query assignment problem minimizes a specific type of a submodular objective function subject to cardinality constraints. The problem of minimizing a submodular function subject to cardinality constraints was studied by [27]: they established a $\Theta(\sqrt{n/\log n})$ approximation ratio for this problem. The approximation algorithms in this paper provide much better approximation ratios whenever the maximum number of sources to which a query is subscribed is sufficiently small, e.g. much smaller than $\Theta(\sqrt{n})$.

8 Conclusions

In this paper, we have proposed and studied the assignment of streaming queries to servers. This is important problem for the design of platforms that execute small 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.

Acknowledgments

We thank our colleagues from Microsoft Bing and Microsoft Research for numerous useful discussions, including Mike Andrews, JP Ahopelto, Tony Bendis, Bart de Smet, Vlad Eminovici, Andi Gavrilescu, 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] *AWS Lambda*. URL: `http://aws.amazon.com/lambda/` (visited on 02/23/2015).

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

[6] Y. Azar. "On-line Load Balancing". *Online Algorithms - The State of the Art, Chapter 8*. Springer, 1998.

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

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

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

[12] S. Fortunato. "Community Detection in Graphs". *Physics Reports* 486.75 (2010).

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

[14] S. Girdzijauskas et al. "Magnet: Practical Subscription Clustering for Internet-scale Publish/Subscribe". DEBS '10. ACM, 2010.

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

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

[17] R. Krauthgamer et al. "Partitioning Graphs into Balanced Components". SODA. 2009.

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

[19] Microsoft. *Scalable Information Stream Processing by Bing in Support of Cortana Scenarios*. URL: `http://channel9.msdn.com/posts/Scalable-Information-Stream-Processing-by-Bing-in-Support-of-Cortana-Scenarios` (visited on 02/20/2015).

[20] Microsoft Developer Network. *System.Linq.Expressions Namespace*. URL: `https://msdn.microsoft.com/en-us/library/system.linq.expressions(v=vs.110).aspx`.

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

[22] L. Querzoni. "Interest Clustering Techniques for Efficient Event Routing in Large-scale Settings". DEBS '08. ACM, 2008.

[23] M. Raab and A. Steger. "Balls into Bins - A Simple and Tight Analysis". *Randomization and Approximation Techniques in Computer Science*. Vol. 1518. Springer, 1998.

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

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

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

[27] Z. Svitkina and L. Fleischer. "Submodular approximation: Sampling-based algorithms and lower bounds". *SIAM Journal on Computing* 40.6 (2011).
[28] S. A. Vinterbo. *A note on the hardness of the k-ambiguity problem*. Tech. rep. DSG 2002-006. Harvard Medical School.
[29] *Windows Azure: Microsoft's Cloud Platform*. URL: `http://www.windowsazure.com/en-us/` (visited on 12/10/2013).
[30] *S4: distributed stream computing platform*. URL: `http://incubator.apache.org/s4/` (visited on 12/10/2013).
[31] Y. Zhao et al. "DYNATOPS: A Dynamic Topic-based Publish/Subscribe Architecture". DEBS '13. ACM, 2013.

APPENDIX

A Proof for Lemma 1

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

Let $\text{OPT}_j(n')$ be the optimal solution for a multi-source QA problem with n' queries, j servers, and the capacity constraint $(1+\nu)\frac{n}{k}$. Note that $\text{OPT}_k(n) = \text{OPT}$. For the first single-server MQP problem, let $\hat{Q}_1^*$ be an optimal subset of queries assigned to server 1 with traffic cost $f(\hat{Q}_1^*)$. Since single-server MQP problem is a relaxation of the QA 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 every $j = 1,2,\ldots,k$. Since $\text{OPT}_k(n) = \text{OPT}$, we obtain

$$f(\hat{Q}_1^*) \leq \frac{1}{k}\text{OPT}_k(n) = \frac{1}{k}\text{OPT}. \tag{5}$$

Consider now the j-th server. Let $\text{OPT}_{k-j+1}(n')$ 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 execution of the algorithm). We claim that

$$f(\hat{Q}_j^*) \leq \frac{2}{k-j+1}\text{OPT}, \text{ for } j = 2,3,\ldots,k. \tag{6}$$

Suppose (6) is true, then the proof follows by summing up the upper bounds in (5) and (6), and using the fact that for the harmonic series H_k it holds $H_k \leq \log k + 1$. We prove (6) as follows.

First, 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 $\text{OPT}_j(n) \leq 2\text{OPT}_k(n)$. For $\text{OPT}_k(n)$, let $cost_i$ be the traffic cost for server i. Without loss of generality, suppose that the servers are enumerated such that $cost_1 \geq cost_2 \geq \ldots \geq cost_k$. Then, we have

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

Using $\text{OPT}_k(n)$, 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,

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

and, thus, it follows that

$$\begin{aligned}\frac{\text{OPT}_j(n)}{\text{OPT}_k(n)} &\leq \frac{2\sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i}{\sum_{i=1}^{j} cost_i + \sum_{i=j+1}^{k} cost_i} \leq 2 \Rightarrow \\ \text{OPT}_j(n) &\leq 2\text{OPT}_k(n), \text{ for all } 1 \leq j \leq k.\end{aligned}$$

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

$$\text{OPT}_k(n_i) \leq \text{OPT}_k(n_j). \tag{7}$$

Since $f(\hat{Q}_j^*) \leq \frac{1}{k-j+1}\text{OPT}_{k-j+1}(n')$, (7) and (7), it follows:

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

B Configuring Relaxed Load Balancing

We provide guidelines on how to set the values of parameters α provided ν based on a probabilistic model. Assume that the probability of assigning a query to a server is according to a uniform random distribution across servers. Then, $(d_1(n), d_2(n), \ldots, d_k(n))$ is a random variable with 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

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

Using Hoeffding's inequality, we obtain

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

Combining with (4), we get: $\Pr(d_j(n) > d(n)) \leq \exp\left(-\frac{2\nu^2}{k^2}n\right)$. Therefore, $\Pr\left(\cup_{j=1}^{k}\{d_j(n) > d(n)\}\right) \leq ke^{-\frac{2\nu^2}{k^2}n}$. From this, it follows that $d_j(n) \leq d(n)$ for every $j = 1,\ldots,k$ 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 $j = 1,\ldots,k$ to hold with probability at least $1-\varepsilon$, it suffices that

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

Suppose that given $\nu > 0$, we want the algorithm to switch to relative load balancing as soon as the probability of violation of the relative imbalance is guaranteed to be smaller or equal than given $\varepsilon > 0$. By (4), the switch 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.