Seaweed: Distributed Scalable Ad Hoc Querying

Richard Mortier, Dushyanth Narayanan, Austin Donnelly and Antony Rowstron
Microsoft Research, Cambridge, UK
{mort,dnarayan,austind,antr}@microsoft.com

Abstract

Many emerging applications such as wide-area network management need to query large, structured, highly distributed datasets. Seaweed *is a distributed scalable infrastructure for querying such datasets. In this paper we describe its architecture and design features, using the Anemone network management system as a motivating example. The main contribution is a design supporting accurate query planning and efficient execution across a large number of unreliable endsystems. In contrast to prior work, Seaweed supports ad hoc querying in addition to continuous querying. The paper describes the solutions adopted by Seaweed: latency-based cost estimation, availability-based scheduling, and meta-data aggregation.*

1 Introduction

Seaweed aims to give users the abstraction of a single centralized database across a large, structured, highly distributed dataset. It provides a generic query infrastructure that can select between multiple alternate query plans, efficiently distribute the selected query to the endsystems owning relevant subsets of the dataset, and aggregate the results. Seaweed is intended to support many different applications, ranging from large data center management, to enterprise-scale endsystem and network management, up to Internet-scale distributed diagnostics. Such applications are characterized by:

- *Large, highly distributed datasets*. Data is spread over 100,000+ geographically distributed endsystems.
- *Well-provisioned but non-dedicated endsystems*. Endsystems supporting Seaweed are machines such as desktop PCs or servers, having plentiful resources which must however be shared with many foreground applications.
- *Ad hoc queries*. In addition to continuous queries on live streaming data, users often need to perform ad hoc queries on stored data, e.g. for diagnostic purposes.
- *Small query results*. Responses to ad hoc queries are typically small subsets of the data or compact aggregates such as averages.
- *Update locality*. Endsystems "own" the data they write, with very little write contention between endsystems. User queries might refer to data distributed over many endsystems, but these are typically read-only.
- *Relaxed consistency requirements*. Unlike ACID transactions, distributed queries can and must tolerate time skew and endsystem unavailability.

For example, when managing large data centers, operators may wish to query the load, the set of running services, the free disk space and so on across a large number of machines for diagnostic purposes. Enterprise-scale endsystem management requires the ability to retrieve similar data from even larger sets of machines [16]. An Internet-scale diagnostic tool such as Dr. Watson for Windows [10] collects program crash information of which only a small subset is uploaded to a centralized server for statistical analysis. A distributed endsystem-based querying approach could support much richer analysis on a wider range of data while avoiding the server-side bottlenecks involved in uploading crash dumps from millions of machines worldwide to a single server cluster.

Throughout this paper we use Anemone [12], an enterprise-scale network management system, as the principal motivating example. Consider a large enterprise network with 100,000+ endsystems distributed around the world. Operators would like to issue network-wide queries for tasks such as load monitoring ("show me the average number of flows across all Exchange servers") and fault diagnosis ("show me the applications running on all machines with more than 500 active flows"). Each endsystem can only record information about its own network usage. A distributed query infrastructure supporting queries against all endsystem data would be invaluable.

To provide the abstraction of a single, centralized database, Seaweed must efficiently distribute user queries to endsystems with relevant data and aggregate the results. It must schedule query execution taking into account the fluctuating availability of endsystems. Furthermore, when data

	src	dst	port$_{src}$	port$_{dst}$	proto	app	PID	bytes	packets	timestamp
client	160.128.6.59	160.120.7.201	2323	1005	TCP	outlook.exe	1374	1923	603	1132329858
(160.128.6.59)	160.120.7.201	160.128.6.59	1005	2323	TCP	outlook.exe	1374	1372294	1000	1132332150
server	160.128.6.59	160.120.7.201	2323	1005	TCP	exchange.exe	856	1923	603	1132331010
(160.120.7.201)	160.120.7.201	160.128.6.59	1005	2323	TCP	exchange.exe	856	1372294	1000	1132332000

Figure 1. Example flow tables demonstrating replication at both client and server.

is replicated — e.g. Anemone network flow records are kept at both ends of each flow — there may be multiple ways to execute the same query with very different costs in terms of network traffic and endsystem load. Thus, cheap but accurate *query cost estimation* is key to effective query planning and optimization.

The main contribution of this paper is a design supporting query planning and execution on a large distributed dataset. We describe in detail a latency-based cost estimation scheme that allows the comparison of alternative distributed query plans; a space-time scheduling approach that addresses the issue of endsystem unavailability; and a distributed index data structure that trades network bandwidth for latency for by pro-actively aggregating meta-data.

The rest of the paper is organized as follows. Section 2 motivates and describes Anemone. Section 3 describes the overall architecture and specific design features of Seaweed. Section 4 describes related work, and Section 5 concludes with a discussion of open research issues.

2 Anemone

Running a large-scale enterprise network requires extensive real-time monitoring. Modern routers provide features such as SNMP [2] and NetFlow [3], which enable network operators to gather simple statistics about interface byte counts, heavy-hitter flows, dominant protocols, etc. Such router based monitoring is becoming increasingly less useful. Widespread use of IPSec and of tunneling techniques such as Virtual Private Networks and protocol stacking (e.g. e-mail over HTTP) significantly reduces the information available to in-network monitoring systems. Increasingly, all a router can observe about the packets it routes is that they are IP, that they are destined for a particular address, and that they purport to originate from a particular address. This information is insufficient to optimally manage a network: to make provisioning decisions, to understand the performance of the provisioned services, etc.

The Anemone approach is to monitor network traffic at the endsystems. Each endsystem in an enterprise monitors and records all inbound and outbound network flows, as well as other information such as the application generating the traffic and the current endsystem load. This produces richer and more detailed flow information than in-network monitoring. A large enterprise such as Microsoft has approximately 300,000 endsystems. If each endsystem maintains 1 GB of local monitoring information on average, this creates a 300 TB distributed database covering about 1 month of continuous 24x7 network monitoring. Thus, the only way to avoid both excessive down-sampling and excessive network overhead is to avoid centralized collection and perform query processing at the endsystems.

In Anemone each endsystem maintains a single database table, `flow`, containing records representing the activity of flows on that endsystem. For each measurement interval (a configuration parameter currently defaulting to 5 minutes) one record is appended per active flow in that interval. Each record contains a timestamp, the flow's IP 5-tuple (source and destination IP addresses, the IP protocol, and any UDP/TCP port numbers), the number of packets and bytes, and the application name and PID of the user-level process. Figure 1 shows some example records for a flow between an email client (Outlook) and an email server (Exchange). Note that each successfully created flow will be represented by records at both the source and the destination. The IP 5-tuple identifying the flow will be the same on either side; in the absence of packet loss the byte and packet counts will also be the same.

Some examples of queries on this `flow` table are:

- `SELECT app FROM flow WHERE bytes>20e20 AND src=LOCAL_IP()`
  Return all applications that have transmitted more than 20 MB in a single interval on a single endsystem.
- `SELECT SUM(bytes) FROM flow WHERE port`$_{src}$`=80 AND dst=LOCAL_IP() AND timestamp>'2005-12-25 00:00:00' AND timestamp<='2005-12-25 23:59:59'`
  Count the total bytes downloaded from web servers on a particular day in the past.
- `SELECT COUNT(*) FROM flow GROUP BY dst HAVING SUM(bytes)>4e20 WHERE app LIKE "outlook%" AND src=LOCAL_IP()`
  Count the servers to which Outlook clients sent more than 4MB.

The second query could be rewritten as

`SELECT SUM(bytes) FROM flow WHERE port`$_{src}$`=80`
`AND src=LOCAL_IP() AND timestamp>'2005-12-25`
`00:00:00' AND timestamp<='2005-12-25 23:59:59'`

which queries the web server end of each flow. Since flow records are replicated at both endpoints, this will give approximately the same result, with small differences due to packet loss and clock skew. However the performance impact could be substantially different: querying a large number of lightly loaded web clients versus a small number of busy web servers. Similarly, the third example could query Exchange servers rather than Outlook clients:

```
SELECT ...  WHERE app LIKE "exchange%" AND
dst=LOCAL_IP()
```

An accurate estimate of the *cost* of such alternative “query plans” would allow us to pick the one that minimized some cost metric such as endsystem load or network bandwidth. To this end Seaweed provides a generic mechanism for plan cost estimation, where plans might be generated by the system, by the application, or even provided by the user.

3 Seaweed architecture

The Seaweed architecture is based on the following design principles, which follow from the application properties mentioned in Section 1.

- *Don’t push data*. Given large datasets but typically small query results, aggressively pushing or replicating data wastes network bandwidth.
- *Focus on global optimization*. We assume that well-provisioned endsystems can efficiently optimize queries on local data, perhaps using a relational database engine.
- *Best-effort consistency*. We aim to provide the “dilated-reachable snapshot” semantics defined by Huebsch et al. [5].
- *Delay-tolerant querying*. Seaweed queries might execute over minutes or even hours, due to the varying load and availability patterns of endsystems in a large network. Thus the system as well as the user must be prepared to support long-lived, delay-tolerant query executions, and to supply partial results during query execution.
- *Estimate and expose query costs*. Users need to be warned before executing queries that might impact the performance of critical services or have unacceptably high latencies.

Each endsystem runs a lightweight DBMS that manages data in one or more tables stored locally; these tables are modified only by the endsystem storing them. An application-level multicast tree [4, 6, 17] is built and maintained with the endsystems storing the data as leaves of the tree. Queries expressed in an SQL-like language are disseminated from the root of the tree to the endsystems, and results propagated back to the root with interior nodes aggregating partial results where possible. Performing a query in Seaweed involves three distinct phases: *planning*, *execution* and *aggregation*.

Planning. There are often many *query plans*, different ways to execute the same query, producing the same result but having different execution costs. A query optimizer [14] is traditionally used to pick the query plan that minimizes disk accesses. In the distributed case, the choice of query plan depends on several factors such as network bandwidth, endsystem resource consumption, and endsystem availability. In a large system the endsystems may be distributed across multiple time zones, and will have different duty cycles and downtimes. Thus query planning must take into account the availability of the endsystems required to participate in plan execution. Further, endsystems are not dedicated to processing queries, and so Seaweed runs as a low-priority background service. Hence the impact on local application performance must be factored into any measure of query plan cost. Additionally, different endsystems might have different local criteria for estimating the cost of the same query execution. For example, a heavily loaded server might assign a higher cost to executing a query than a lightly loaded desktop machine. These considerations make distributed query planning very different from the single-system case, requiring a new approach.

Execution. Once a plan is selected the query must be executed. First, Seaweed must ensure that the query is disseminated to all endsystems required to execute the query locally. This must take into account the different availability profiles of endsystems, some of which may be offline when dissemination begins. Once an endsystem receives a query it must locally schedule its execution, taking into account local criteria and the time when the response is expected. Thus, in general, Seaweed must be able to schedule query dissemination and execution over time scales of minutes or even hours.

Aggregation. For queries that request an aggregate measure such as a sum or an average, partial results are aggregated at interior nodes in the application-level multicast tree for bandwidth savings. There are a number of established techniques and proposals for such aggregation [9, 17]. For long-running queries, partial results are periodically returned to the user, allowing her to stop query execution after it has covered, for example, 90% of the data.

The remainder of this section describes the key design components in Seaweed that enable scalable, distributed, delay-tolerant querying: *latency-based cost estimation* for query planning, *space-time scheduling* based on availability, and *pro-active meta-data aggregation* using a distributed index.

3.1 Latency-based cost estimation

Query planning requires *generation* of multiple alternative plans for the same query followed by *cost estimation* for each of these plans. We currently assume plan generation is performed by the application. For example, Anemone could generate alternative plans using clients and servers for a query (Section 2). We focus here on cost estimation: *given a set of plans, how do we assign a cost to each allowing comparison?* We assume that queries and query results are usually small, and hence focus on the cost of query execution in terms of the impact on endsystem resources and local application performance.

A first approach to estimating the plan costs would be to try to capture the impact of the query on each endsystem resource. However, this has two drawbacks. First, endsystems vary in their hardware resources, and estimates of resource usage are not directly comparable across endsystems. Second, the cost perceived by each endsystem also depends on the performance requirements of the local applications and the local resource allocation policies.

In Seaweed, we avoid these difficulties by expressing endsystem costs in terms of a single metric: *latency*. Each endsystem independently estimates the time by which it would be willing to complete the execution of each plan. This estimate is based on the expected resource usage of query execution (computed by the local DBMS's cost estimator), the observed load on those resources, the local priority or resource share assigned to Seaweed, and the expected availability of the endsystem. It thus takes into account not just the actual time it will take to execute the query but also the impact of execution on foreground application performance, and any delays due to the endsystem being currently unavailable. In Section 3.2 we explore further the issues involved in predicting endsystem availability over timescales of several hours or more.

During the planning phase the plans are disseminated to the endsystems, each of which returns a latency estimate and also a *row count* estimate: this is the local DBMS's estimate of the number of records in the local tables that will be matched by the query. The latency and row count estimates from the endsystems are aggregated up the tree as a *rowcount-weighted log-latency histogram*. This histogram captures the expected number of records that will be processed after 1 seconds, 2 seconds, 4 seconds, etc. The log scale allows the histogram to accommodate a wide range of possible latencies from seconds to hours.

The histogram at the root is exposed to the user, allowing her to trade latency for data coverage as measured by row count. For example, a query might have two plans. The histogram for the first plan predicts that 90% of the records are processed after 4 seconds, but the remaining 10% will require a further 2 hours. The second plan is expected to process 90% of the records after 10 minutes and the remaining 10% after 1 hour. Based on this information, the user might decide to execute the first plan, and then to terminate it with partial results after 5 seconds (or alternatively after seeing 90% coverage of the data).

3.2 Space-time scheduling

When querying a large number of endsystems that span several locations, time zones, or even continents, the query planning and execution must be distributed both in *time* as well as in *space*. For example, it might happen that the endsystems required by a particular query have different availability patterns, and that there is no single instant at which all of them are available.

Thus a major challenge for Seaweed is coping with unavailable endsystems. Our approach is based on the idea of *space-time scheduling*, which incorporates:

- Propagating queries, queuing them for execution, and canceling them when endsystems are unavailable.
- Minimizing the impact of unavailability through judicious replication.
- Tracking and predicting the availability patterns of endsystems.

During query planning, endsystems locally generate cost and row count estimates, so unavailable endsystems will result in incomplete latency histograms. We address this problem by replicating a small amount of endsystem meta-data on other endsystems, allowing the generation of row count and latency estimates even for currently unavailable systems. In order to do this, the replicated meta-data must include:

- The data distribution histograms used by the local DBMS to estimate the row count and resource usage of query execution.
- The load (duty cycle) profile, which allows estimation of query execution latency.
- Local scheduling policy, which might delay execution of the query even after the endsystem becomes available.
- The availability (downtime) profile, which allows estimation of when the endsystem will next become available for query execution.

In addition to replicating this meta-data, Seaweed must be able to accurately predict the time at which a currently unavailable endsystem will next become available. This is required both for accurate estimation of query latency, as well as for scheduling the eventual execution of the query.

Previous studies on endsystem availability in large networks [1] have shown that endsystems have distinct patterns of up- and down-times: for example, desktop machines might be turned off on evenings and weekends, while server machines exhibit short and infrequent outages due to failures. We are working on building predictive models of endsystem availability to capture cyclic patterns in up- and down-times.

3.3 Pro-active meta-data aggregation

Depending on the query workload, there could be situations where dynamically generating latency and row count estimates is unacceptable. At high query rates the bandwidth consumed by disseminating the plans and aggregating the cost estimates could be significant. If the majority of queries require execution on only a small fraction of the endsystems, then multicasting the query to all endsystems can be expensive. Finally, multicasting the plans and aggregating the cost estimates increases the latency of query planning.

Thus under some workloads there is a need to *pro-actively* propagate and aggregate information up the tree so that plan cost estimation can be done locally at the root. This requires that the replicated meta-data described in Section 3.2 — data distribution histograms, load profiles, and availability profiles — be periodically aggregated up the tree to the root. There are many challenges in efficiently propagating and merging these data structures: we present a *distributed index* data structure that addresses the propagation of data distribution histograms, which allows row count estimation and endsystem selection to be done efficiently.

Traditionally databases maintain an index, which is a hierarchical data structure specifying which disk pages contain records with a particular key value or range of key values. Indexes are typically associated with histograms, which describe the overall distribution of key values in terms of row count. Our distributed index structure performs analogous functions, specifying the endsystems with records in some range of key values, and also the overall distribution of values for that key. The first property allows us to efficiently propagate queries (both for cost estimation and for execution) by pruning out the portions of the tree with no data of interest. The second property allows us to generate the row count estimate for any query locally at the root of the tree.

The distributed index is built by retrieving the data distribution histogram for each key of interest from the endsystem DBMS at the leaf level, and propagating it up the tree. Each interior node merges the index structures from the child node, to bound the amount of propagated data, and sends the merged histogram to its parent. The node also remembers the histograms received from each child, and uses them to route cost estimation requests as well as query execution requests. The latter are forwarded only to those children that have a non-zero row count matching the `WHERE` predicate of the query.

Thus, to ensure that we do not incorrectly prune out endsystems with relevant data, merging of index information at interior nodes must be *conservative*. At the same time, merging should be *efficient*, to bound the index size as it propagates up the tree. We have identified three types of conservative yet efficient index data structures, each suitable for a particular kind of attribute (key):

1. *Counting Bloom filters* [11] for point queries on discrete-valued attributes, such as
   `SELECT ... WHERE port`$_{dst}$`=80 OR port`$_{dst}$`=8000`
   A Bloom filter is a vector of cost values with each entry representing some set of key values. Filters are merged by summing the corresponding vectors. The vector length is a global tuning parameter.
2. *Range-based histograms* for range queries on continuous-valued attributes, such as
   `SELECT ... WHERE bytes>20e20 AND bytes<40e20`
   These resemble traditional database histograms, with slight modifications to allow conservative merging. We are currently exploring various heuristics for merging indexes with minimal information loss.
3. *Tries* (prefix trees) for string attributes and prefix queries. After merging their children's prefix trees, nodes reduce the trie size by pruning at the leaves; again, a variety of heuristics is available to choose the leaves for pruning.

4 Related work

There has been much research on index data structures for distributed databases, such as dPi-trees [8] or dB-trees [7], which attempt to provide similar transactional consistency and recoverability to centralized databases. In contrast Seaweed is explicitly designed to be simple and scalable while providing weaker consistency and requiring non-conflicting updates.

PIER [5] is a distributed SQL query engine with dilated-snapshot consistency, built over a DHT structured overlay. In PIER the tuples generated by endsystems are stored on random nodes in the structured overlay. PIER then uses the structured overlay to lookup the tuples. In Seaweed the endsystems store the tuples locally, and queries are distributed to the endsystems. This is a better approach for the case where query results are small compared to the dataset.

Many other systems designed to support distributed information management, e.g. Astrolabe [16] and SDIMS [17], focus on support for continuous queries and user-defined aggregation functions, injected by users into these systems and executed on streaming data. Hourglass [13, 15] optimizes queries on streaming data by carefully selecting the location of aggregation points. There is also a large body of work in sensor networks which use continuous queries and aggregation, such as TAG [9]. Seaweed is targeted at endsystems that store data locally for arbitrarily long periods of time but may be unavailable from time-to-time, and at providing the ability to run ad hoc queries against this historical data.

5 Open issues

This paper described our initial approach to supporting distributed scalable queries in Seaweed. There are several research challenges yet to be resolved: here we briefly describe the most important ones.

Data replication and caching. Seaweed currently replicates meta-data, but data can only be accessed on the endsystems where they are generated. Pro-actively replicating the data across multiple endsystems would increase the probability of availability, thus improving the tradeoff between data coverage and latency. This comes at the cost of additional endsystem and network resources. Similarly, caching of query results at endsystems and interior tree nodes could improve performance, if the workload contains several repeated or similar queries. We plan to investigate the tradeoffs involved in data replication and caching after gaining experience with application workloads on the current Seaweed design.

Automating plan selection. The best-effort consistency and availability of Seaweed, as well as the potential to schedule queries across time and space, results in a multi-dimensional tradeoff between consistency, availability, resource utilization, and response time. We expose these tradeoffs to the operator so they can decide between, for example, immediate but incomplete answers and slower but more complete ones. An open issue is how to automate these decisions.

Joins. For some applications, it might be valuable to support a `JOIN` primitive, in addition to simple `SELECT` queries. Distributed joins require more than filtering and aggregation for efficient execution: the system must choose good locations for intermediate processing and correctly estimate the costs of different join plans, both with respect to the amount of data returned and the location of the data. It might even be the case that efficient support for `JOIN`s requires that the results aggregation tree be constructed differently from the query dissemination tree.

Acknowledgements

The authors would like to thank Paul Barham, Miguel Castro, Evan Cooke and Rebecca Isaacs.

References

[1] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proc. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000.

[2] J. Case, M. Fedor, M. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157, IETF, May 1990.

[3] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954, IETF, Oct. 2004.

[4] Y. hua Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *Proc. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Santa Clara, CA, June 2000.

[5] R. Huebsch, J. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, Sept. 2003.

[6] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *Proc. Symposium on Operating Systems Designa and Implementation (OSDI)*, San Diego, CA, USA, Oct. 2000.

[7] T. Johnson and P. Krishna. Lazy updates for distributed search structure. In *Proc. ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1993. ACM Press.

[8] D. Lomet. Replicated indexes for distributed data. In *Proc. International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, FL, USA, 1996.

[9] S. Madden, M. Franklin, and J. Hellerstein. TAG: A Tiny AGgregation service for ad-hoc sensor networks. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[10] Microsoft. Dr. Watson for Windows. `http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/drwatson_overview.mspx`, Jan. 2006.

[11] M. Mitzenmacher. Compressed Bloom filters. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*, Newport, RI, USA, 2001. ACM Press.

[12] R. Mortier, R. Isaacs, and P. Barham. Anemone: using end-systems as a rich network management platform. In *Proc. ACM SIGCOMM Workshop on Mining Network Data (MineNet)*, New York, NY, USA, 2005. ACM Press.

[13] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. International Conference on Data Engineering (ICDE) (to appear)*, Atlanta, GA, USA, Apr. 2006.

[14] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In P. A. Bernstein, editor, *Proc. ACM SIGMOD International Conference on Management of Data*, Boston, MA, USA, May 1979.

[15] J. Shneidman, P. Pietzuch, M. Welsh, M. Seltzer, and M. Roussopoulos. A cost-space approach to distributed query optimization in stream based overlays. In *Proc. IEEE International Workshop on Networking Meets Databases (NetDB '05)*, Tokyo, Japan, Apr. 2005.

[16] R. Van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.

[17] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc. ACM SIGCOMM Symposium on Communications Architectures and Protocols*, Portland, OR, USA, Sept. 2004.