Delay Aware Querying with Seaweed

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

ABSTRACT

Large highly distributed data sets are poorly supported by current query technologies. Applications such as endsystem-based network management are characterized by data stored on large numbers of endsystems, with frequent local updates and relatively infrequent global one-shot queries. The challenges are scale (10^3 to 10^9 endsystems) and endsystem unavailability. In such large systems, a significant fraction of endsystems, and their data, will be unavailable at any given time. Existing methods to provide high data availability despite endsystem unavailability involve centralizing, redistributing or replicating the data. At large scale these methods are not scalable.

We advocate a design that trades query delay for completeness, incrementally returning results as endsystems become available. We also introduce the idea of *completeness prediction*, which provides the user with explicit feedback about this delay/completeness trade-off. Completeness prediction is based on replication of compact data summaries and availability models. This *metadata* is orders of magnitude smaller than the data.

Seaweed is a scalable query infrastructure supporting online aggregation and completeness prediction. Seaweed is built on a distributed hash table (DHT) but unlike previous DHT based approaches it does not redistribute data across the network. It exploits the DHT infrastructure for failure resilient metadata replication, query dissemination, and result aggregation. We analytically compare Seaweed's scalability against other approaches and present an evaluation of the Seaweed prototype running on a large-scale network simulator driven by real-world traces.

1. INTRODUCTION

Querying endsystem data on large networks such as data centers, enterprise networks, or the Internet requires a scalable distributed query infrastructure. Recent research has looked at building such infrastructures [1, 16, 22, 31, 32]. The challenges for these infrastructures are *availability* and *scalability*. A significant fraction of endsystems will be unavailable at any given time due to network outages, endsystem failures, and scheduled downtimes, which means the infrastructure must cope with the unavailability of some fraction of the data. Additionally, any query infrastructure must scale, i.e. the bandwidth overheads of query execution and background maintenance must not become prohibitive at large scale.

Currently proposed solutions to the problem of data unavailability involve centralization, redistribution, or replication of the data. These techniques do not scale well with data size per endsystem or data update rate per endsystem. An example of such a system is PIER [16], where every endsystem periodically reinjects all its tuples into the network, requiring network bandwidth linear in the product of network size, per-endsystem data size, and reinjection rate. We believe storing data anywhere but on the endsystem where it is produced fundamentally limits scalability.

In this paper we present Seaweed, a scalable querying infrastructure which solves the problem of data unavailability by allowing one-shot queries to persist until unavailable data becomes available. By querying data entirely *in-situ*, Seaweed scales with network size, data size, and data update rate. Results are produced incrementally with completeness improving over time. Completeness is defined as the percentage of rows in the system pertinent to the query that have been processed. Seaweed solves the problem of data unavailability by providing users an explicit trade-off between completeness and delay. It does so by providing estimates of current and predictions of future completeness.

1.1 Our Contributions

Previous approaches have addressed the problem of data unavailability using data replication which fundamentally limits scalability. We introduce the novel concept of *delay aware querying* with *completeness prediction.* Delay aware querying is scalable and solves the problem of data unavailability by explicitly trading query delay for completeness. Delay awareness is achieved by exposing to the user a prediction of the expected delay to reach any given level of completeness. We do this by predicting when currently unavailable endsystems will next become available and estimating the amount of relevant data that each endsystem has for a query. This is achieved by replicating a small amount of per-endsystem metadata consisting of an *availability model* and a compact data summary.

We describe the Seaweed architecture which uses an application level overlay or distributed hash table (DHT). Unlike other DHT-based approaches, Seaweed does not use the

DHT to replicate or redistribute the dataset but to replicate the metadata. Data is queried in-situ and Seaweed leverages the DHT's underlying overlay structure to build efficient, failure-resilient protocols for query dissemination and online result aggregation.

We show through analytic models that Seaweed scales better with network size, data size, and data update rate than approaches based on centralization, redistribution, or replication of data. We also present simulation results showing that Seaweed can efficiently disseminate queries, generate accurate completeness predictors, and aggregate query results.

1.2 Applications

There are many applications that are enabled by scalable querying infrastructures. We are particularly interested in endsystem and network management at different scales. At the small scale many Internet services, such as Google, Amazon and MSN, run multiple data centers at geographically distributed locations, each containing thousands of endsystems. Each endsystem can generate large amounts of fine-grained performance data of interest to human operators and automated support system. Effective analysis and diagnosis based on this data requires support for distributed querying.

At the next order of magnitude, we have large enterprise networks with hundreds of thousands of endsystems. The original motivation for Seaweed was to support Anemone, an endsystem based network measurement and monitoring system [26] for such enterprise networks. Endsystems in enterprise networks can capture and store data about local resource usage, network activity, running applications, etc. For example, Anemone can store network information at the per-flow and per-packet level. This data can then be queried by the network operator for aggregate statistics, diagnostics, or historical exploration.

Finally, at Internet scale, applications such as Dr. Watson [25] report crash dump data from millions of Windows machines worldwide to a single centralized site for subsequent analysis. The amount of data uploaded is limited by available bandwidth: an in-situ approach would allow queries over a richer dataset while limiting network overheads.

These applications are characterized not only by their scale, but also the need to support *one-shot queries* rather than just streaming queries. Simple streaming queries might be used to monitor aggregate statistics over time. However, when an operator observes an unexpected reading they need to perform one or more retrospective one-shot queries over the stored data to diagnose the issue. If the issue being diagnosed relates to availability (e.g. "why did I get no results from rack 10 between 08:30 and 09:00?"), then the streaming results will provide little helpful information. Hence there is a need for a scalable, efficient infrastructure supporting one-shot queries on distributed stored data.

1.3 Limitations

We restrict Seaweed queries to be either local or read-only: Seaweed does not support distributed updates. Standard techniques for distributed updates such as distributed locks and 2-phase commit do not scale well, and our design philosophy was to eschew any functionality that would limit scalability. Our current prototype also does not support distributed joins as they are difficult to make scalable. For example, joins in PIER [16] can require cross-network bandwidth linear in the size of the joined data tables. By restricting read-only queries to be single-table and updates to be single-endsystem, we gain scalability at the cost of query functionality. This seems an acceptable trade-off for the management applications we have examined. Functionality such as distributed updates or joins over small numbers of endsystems could be provided in a layer above Seaweed for applications that require it.

Seaweed's query dissemination is scalable with respect to network size, and resilient to faults in the network. However, it disseminates queries to all endsystems, which must perform at least the minimal processing to determine if they have data matching the query. This could cause significant overheads at high query rates, where approaches such as distributed indexes [22, 27] might prove useful. Currently, our target scenarios contain a small number of human users such as network administrators who issue one-shot queries. We evaluated the benefit of distributed indexes and concluded that the query rates in our target applications did not justify the overheads and complexity of maintaining a distributed index.

1.4 Map

The remainder of the paper is organized as follows. Section 2 describes the design philosophy, high level design decisions, and novel features of Seaweed. Section 3 describes the detailed design of our prototype, including the protocols used for query dissemination and result aggregation. Section 4 compares analytic models of Seaweed with three alternative architectures, demonstrating the superior scalability of Seaweed. It also provides simulation results quantifying the network overheads of various components of Seaweed and the accuracy of completeness prediction. Section 5 briefly summarizes related work, and Section 6 concludes the paper.

2. DESIGN PRINCIPLES AND INSIGHTS

For simplicity, we use standard data models and querying languages for our implementation. We assume that data is relational and that for any given application there is a standard schema across endsystems. The data thus consists of a set of tables, each of which is horizontally partitioned across a large number of endsystems. Each endsystem is capable of executing relational queries and updates on its local data. For many applications, there may be data integration issues which render such a model over-simplistic [14]; these are outside the scope of this paper.

Our query language is a subset of SQL. Read-only queries may be distributed across endsystems but must not perform distributed joins. Updates are constrained to a single endsystem at a time.

A Seaweed query is inserted into the system by the application layer on any endsystem. Seaweed dynamically builds an application-level tree that disseminates the query to all available endsystems. The endsystems return completeness predictors that are aggregated up the tree. The predictor at the root allows the user to estimate the completeness of the incremental result at any time and also its expected future progress.

Meanwhile, endsystems also execute the query locally and generate results. These results are propagated to the root

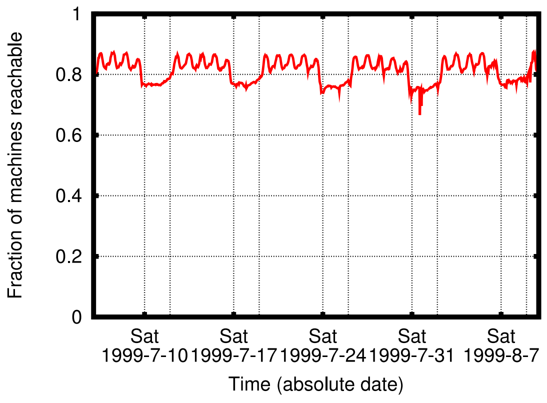

Figure 1: Availability of 51,663 endsystems on the Microsoft corporate network in July/August 1999.

using another tree which is built dynamically from the leaf level upward. If the query uses standard aggregation operators, results are aggregated in the tree to reduce bandwidth usage. Any new or previously unavailable endsystem that joins Seaweed receives a list of currently active queries for which it generates results and returns them to the root. Incremental results will thus continue to arrive for any query until it times out or is explicitly cancelled.

2.1 Availability

Endsystem availability is a major challenge for any distributed querying infrastructure. Studies of endsystem availability in widely deployed peer-to-peer applications such as Gnutella [29] and Overnet [5] show that there is significant churn in the set of available endsystems. Even studies of more benign enterprise network environments show that a significant fraction of endsystems are unavailable at any time. Figure 1, reproduced from one such study [8], shows the availability of 51,663 endsystems on the Microsoft Corporate network in July–August 1999. Each endsystem was probed once per hour to test its availability. On average, 81% of the endsystems were available at any time. Further, there is a clear periodic pattern, suggesting that endsystem availability is predictable.

Therefore a guiding principle for all distributed scalable querying systems is to design for unavailability. Solutions involving replication of *all* data in a large system place a prohibitive load on the network, even if the amount of data per endsystem is relatively small. This observation is validated by our analysis in Section 4 as well as other studies on wide-area distributed applications [7]. This motivated our design decision not to replicate the raw data but to address the availability problem through delay aware querying.

A key component of delay aware querying is completeness prediction. Completeness predictors are computed at endsystems from the replicated metadata and aggregated up the query distribution tree. A completeness predictor is a cumulative histogram of expected row count over time. For example, a user could use it to estimate that 80% of the rows are immediately available, 99% within 1 hour, but 100% only after several days. The user might then decide to accept the results after 1 hour and then cancel the query rather than waiting for perfect completeness.

Seaweed provides metadata replication as an application-independent service, where the metadata consists of histograms on indexed attributes and availability models. The replication frequency and the set of indexed attributes are application-specific parameters.

Replication of indexed attribute histograms can be viewed as a special case of *selective replication*. In general, one could imagine the application designer specifying any subset of the data (e.g. projection) or derived values (e.g. materialized views) for replication. Queries on the replicated portion alone would be answered with comparatively low latency, albeit with some staleness dependent on the replication frequency. Obviously careless selection of data for replication could result in an unscalable application.

2.2 Scalability

Seaweed's design achieves scalability through a combination of two factors. First, by not replicating the data the network overheads of dealing with unavailability are vastly reduced: the combined size of the availability models and histogram metadata is several orders of magnitude smaller than the raw data. Second, Seaweed's multicast tree protocols are designed to be both scalable and fault-tolerant, with each endsystem only sending or receiving a small number of messages per query.

2.3 Consistency

A highly distributed system precludes certain kinds of consistency such as ACID. Even snapshot validity, which guarantees that a read-only query sees a snapshot at a single time across the entire system, cannot be guaranteed under a relaxed asynchronous model of distributed systems [4]. Systems such as PIER [16] provide relaxed consistency in the form of a 'dilated reachable snapshot' where only available endsystems will respond to a query, and the 'snapshot' across these endsystems will be dilated by clock skew.

Seaweed provides more precise guarantees than 'dilated reachable snapshot' on the set of endsystems that will respond to a query. We define our consistency in terms of *single-site validity* [4]. We define the set $H_C(t_1, t_2)$ as the set of hosts that were available at *all* instants in $[t_1, t_2]$. Note that Seaweed does not distinguish between unreachable and unavailable endsystems: an available endsystem is reachable within Seaweed by definition. We define $H_U(t_1, t_2)$ as the set of hosts that were available at some instant in $[t_1, t_2]$ for sufficient time to execute the query. Consider a user who injects a query into Seaweed at time 0 and observes the partial result at time T. We guarantee that the set of endsystems H contributing to the result equals $H_U(0, T)$. This is a more precise form of single-site validity, which guarantees only that $H_C(0, T) \subseteq H \subseteq H_U(0, T)$.

For completeness prediction we can offer a stronger guarantee yet. The metadata replicas for any endsystem that was ever available in the past remain available with high probability. Thus, if Seaweed provides the aggregated predictor for the query at time $T_e \leq T$, then the set of endsystems H contributing to this predictor will with high probability satisfy $H_U(-\infty, 0) \subseteq H \subseteq H_U(-\infty, T_e)$. In practice, T_e is on the order of seconds, and the difference between the upper and lower bounds is small.

The key feature of Seaweed allowing us to provide stronger single-site validity is that we guarantee to count each endsystem's contribution to the result *exactly-once* provided it becomes available during the lifetime of the query, $[0, T]$. Guaranteed counting is provided since the underlying Pastry overlay automatically keeps track of the membership of Seaweed, and guarantees that any node in Seaweed that is available *will* have a path to the query's root node. Exactly-once counting is ensured by the way that Seaweed constructs its broadcast and convergecast trees as described in the following section.

3. SEAWEED DESIGN

Seaweed is implemented on top of Pastry [28], a scalable, self-organizing, structured overlay network. We provide a brief overview of Pastry before describing the three main components of Seaweed: replication of availability models and data summaries; query dissemination and completeness prediction; and result aggregation.

3.1 Background: Pastry

Endsystems and objects in Pastry are assigned random identifiers, known as *endsystemIds* or object *keys* respectively, from a large sparse wrapped namespace. Keys and endsystemIds are 128 bits in length and can be thought of as a sequence of digits in base 2^b, where b is a configuration parameter with a typical value of 4. Given a message and a key, Pastry routes the message to the key's *root*: the endsystem with the endsystemId numerically closest to the key. When a message is delivered successfully it is then passed to the application running on that endsystem.

Messages can be routed from any endsystem to any other. Each endsystem maintains a *routing table* of size $O(\log_{2^b} N)$, where N is the total number of endsystems in the system, and a *leafset* of the $l/2$ neighboring endsystems clockwise and counter-clockwise in the namespace. The leafset size l is a configuration parameter typically set to 8. Using these data structures, Pastry can deliver any message in $O(\log_{2^b} N)$ hops.

Our Seaweed implementation is built on the MSPastry [9] implementation of Pastry. MSPastry provides a distributed hash table (DHT) API [12], which is used by Seaweed for metadata replication. MSPastry also provides a lower-level key-based routing (KBR) API which is used by Seaweed to build and maintain trees. MSPastry has low overhead and provides reliable message delivery under adverse network conditions: even with network message loss rates as high as 5% together with high overlay membership churn, the incorrect delivery rate is only 1.6×10^{-5} [9].

3.2 Metadata replication

In order to be able to generate completeness predictors, metadata consisting of the data summaries and availability model of each endsystem is actively replicated. Pastry's DHT API allows insertion and lookup of key/value pairs as in a traditional hash table. At insertion a replication factor r is specified. Pastry will replicate the inserted value on the r numerically closest endsystems to the key.

In Seaweed, each endystem inserts into the DHT its metadata using its own endsystemId as the key, and a replication factor of $k + 1$ where k is the desired number of additional replicas. The effect of this is that this metadata is replicated on k members of the endsystem's leafset: we refer to these members as the replica set. The replica messages are routed in a single hop, and thus both the network latency and the bandwidth usage are small. Any member of an endsystem x's replica set can thus generate a completeness predictor for any query on behalf of x when x is unavailable.

We now describe the two components of the metadata: the availability model and the data summaries.

3.2.1 Availability model

For each unavailable endsystem the availability model is used to determine when it is likely to become available again. In particular, if an endsystem has currently been unavailable for time t, what is the likely duration before it becomes available once more?

Two distributions are maintained per-endsystem: *down duration* and *up-event by hour of day*. The down duration captures the length of time for which an endsystem stays unavailable, and the up-event distribution captures the hour of day (0–23) at which it comes back up.

Many endsystems follow a periodic cycle, e.g. people turning their desktop machine on when they arrive at work. If the up event distribution for an endsystem is heavily concentrated in a certain hour (if the peak-to-mean ratio exceeds 2), we classify it as periodic and use the up event distribution for availability prediction. Otherwise, we use the down duration distribution for prediction: in this case, the prediction also takes into account t, the time for which the endsystem has currently been unavailable.

The two distributions are persisted at each endsystem and dynamically updated over time. Whenever an endsystem becomes available, it updates the distributions and locally classifies itself as periodic or non-periodic. It then pushes out the relevant distribution to its replica set.

When a member y of the replica set notices that an endsystem x is unavailable, it remembers the time at which this occurred. At any subsequent point, it can predict when x will next become available based on its copy of x's availability model.

3.2.2 Data summaries

Each endsystem x pushes its data summary to its replica set when it (re)joins the network; the summary is also pushed to new replica set members when the replica set changes due to failure. Additionally, endsystems periodically push their summary to the replica set if the data has changed.

In Seaweed the summary currently consists of all key value distribution histograms on indexed attributes of the local database. When an available endsystem generates a row count estimate for a query on its own behalf, it generates the estimate directly from the local DBMS. When row count estimation is done on behalf of an unavailable endsystem, we use standard row count estimation techniques on the replicated histogram information.

Currently we take the conservative approach of pushing the histogram periodically if there is any change at all in the data. We are looking at methods to dynamically vary the push rate based on the data change rate, as well as sending delta-encoded histograms which could reduce network overhead compared to pushing the entire histogram.

3.3 Query dissemination and completeness prediction

When a query is submitted to Seaweed the first stage is to generate the completeness predictor. The query is assigned a key, its SHA-1 hash, referred to as the *queryId*. The query must be reliably disseminated to the available endsystems and estimates must be generated on behalf of the unavailable endsystems. The dissemination algorithm must ensure that the exactly-once semantics are maintained, even as endsystems concurrently join and fail during the process.

To provide robust query dissemination and to generate the completeness predictor Seaweed dynamically builds a distribution tree. For ease of explanation we describe the tree here as a binary tree; our implementation uses a 2^b-ary tree.

The root of the tree is the endsystem with the endsystemId numerically closest to the queryId. The root initiates an efficient broadcast using a divide-and-conquer approach. Each broadcast message contains an explicit namespace range for which predictions are required; at the root level, this corresponds to the entire namespace range of Pastry. When an endsystem receives a broadcast, it subdivides the range into two equal ranges, and sends one message for each of the subranges. One of the messages will be sent to itself, and the other will be routed towards the midpoint of the other subrange. This will eventually reach an endsystem within that subrange, typically within one hop.

When an endsystem detects that it is the only live endsystem in a range or that it is the numerically closest live endsystem to a range containing no live endsystems, it takes responsibility for all unavailable endsystems in that range, and generates completeness predictors for them from the replicated metadata. If it lies within the range it also generates its own completeness predictor based on row count estimates from its local DBMS. This recursive process creates a tree with depth $O(\log_{2^b} N)$, which determines the latency of query dissemination.

The endsystem row count estimates are aggregated to a cumulative histogram of row counts against predicted time of availability, where time is on a log scale to accommodate wide variations in availability ranging from seconds to days. These histograms are the per-endsystem completeness predictors. They are propagated and aggregated up the tree, with each endsystem transmitting the predictor to its parent: the endsystem that originally sent the query to it. The predictors are aggregated at each step and are thus maintained at constant size.

In order to make this process robust endsystems send heartbeats to their parents. If an endsystem does not receive a heartbeat or predictor within a specified period then it reissues the request for that sub-range. Since predictor generation takes place on the order of seconds, there will typically be very little churn during this window, and thus the retransmission costs will be low.

The protocol also exploits the format of Pastry routing tables to achieve a message overhead of $O(N)$. It relies on the property that when a broadcast is forwarded by node x to a subrange, with high probability there is a live endsystem y in that subrange, in x's routing table. Thus each step of the divide-and-conquer dissemination is $O(1)$.

3.4 Result aggregation

Once the completeness predictor is generated, each available endsystem generates the result for the query. While predictor generation takes place in seconds, incremental result generation can span hours as more endsystems become available. This means that a different tree must be built from the leaves up for result aggregation: since churn now becomes a significant factor, this tree cannot rely on aggressive retransmission for failure-resilience.

The result aggregation tree must also ensure that once an endsystem becomes available and submits its result, it must be counted exactly once in the result at the root. Maintaining a list of all endsystems that have contributed results is not feasible, as this will result in messages of size $O(N)$. Instead, we maintain $O(1)$ information in each node of the aggregation tree: the current results received from each child. When new results are received from any child (due to an endsystem in that subtree becoming available), this list of child results is updated, and a new aggregate is computed and forwarded up the tree.

This protocol requires that each vertex in the tree deterministically computes the vertexId of its parent: a vertexId is a key in the DHT namespace. We accomplish this through a deterministic function $v(queryId, vertexId) \mapsto vertexId$. This function defines a tree of depth $O(\log N)$ rooted at the query originator.

The aggregation protocol guarantees that results are generated exactly once for each endsystem when it becomes available, assuming that there are no failures in the interior nodes of the tree. To provide this property, we implement each interior vertex as a failure-resilient replica group.

Each group is represented by a primary with k backups. The primary is always the endsystem whose endsystemId is numerically closest to the vertexId, thus guaranteeing that messages sent to the vertexId are always routed to the primary. The primary replicates its state to the backups before acknowledging any message or transmitting any message to its parents. If any member of the group fails, then a new endsystem joins the group, and a new primary is selected automatically if necessary , always maintaining the property that the primary has the endsystemId closest to the vertexId.

This protocol provides exactly-once semantics with very high probability: for an entire vertex to fail, the primary and all backups would have to fail within a short period of time determined by the Pastry leafset heartbeat interval, currently 30 s.

This protocol also makes it possible to support continuous queries in a failure-resilient manner; however this is outside the scope of this paper.

4. EVALUATING SEAWEED

In this section we present simplified analytical models of Seaweed's scalability, and evaluate them against three alternative architectures. Our aim is to understand the inherent trade-offs and limitations in the models with respect to network overheads, network size, data size, and data update rate.

These analytical models simplify many of the engineering issues involved in building real distributed systems. To better understand the performance of Seaweed in a real application scenario, we also present an evaluation of Seaweed running in a network simulator driven by real-world traces.

Seaweed can be compiled to run in the simulator or standalone. We do not present results from the stand-alone version, as our focus is in this paper is scalability, and we do not have a large-scale deployment.

Before describing the analytical models we briefly describe the application we use to drive this evaluation.

4.1 Application

In this paper, we use Anemone [26], an endsystem-based network management application, as our driving application for Seaweed. In Anemone, each local machine has two tables, `Packet` and `Flow`, which capture the network activity of that machine. Each record in the packet table contains a timestamp, the source and destination IP addresses and ports, the protocol, the direction of the packet (Rx or Tx), and the size in bytes. The flow table is a per-flow summary of the packet data, which periodically records for each active flow the timestamp, the interval of measurement, the IP addresses and ports, and the number of bytes and packets sent and received. The flow measurement interval is currently set to 5 min.

A typical query by a network operator on this data might be:

```
SELECT SUM(Bytes) FROM Flow WHERE SrcPort=80 WHERE ts
<= NOW() AND ts >= NOW() - 86400
```

which gives an idea of the total amount of web activity in the network in the last 24 hrs. Note that `NOW()` will be generated using the querying machine's timestamp, assuming the loose clock synchronization described in Section 2.3.

Seaweed will disseminate the query to all nodes; generate an estimate of recall over time; and propagate incremental results as they become available. In this case, since the query uses a standard aggregation operator, these incremental results will be aggregated in-network to minimize network overheads.

4.2 Modeling

In this section we present an analytical model of Seaweed, and of three alternative designs: *centralized*, *DHT-replicated*, and *PIER*. For each design we derive formulas for the background maintenance overhead in terms of network bandwidth, measured as bytes/second transferred system-wide.

All the models are driven by system parameters that characterize the network size, availability characteristics, data size, and data update rate. We denote the network size — the total number of endsystems — by N. Of these, we expect some fraction f_{on} to be available on average at any given time. The churn rate c is the average rate at which any single endsystem switches between available and unavailable. It measures the dynamics of availability, i.e. the rate at which endsystems change between available/unavailable. Since we assume f_{on} remains stable, we assume that the system-wide rates of joining and leaving are equal, and we add them to get the total churn Nc. The data update rate u measures the average amount of new data generated by each endsystem per second; for simplicity we assume that only available systems generate data. The database size d measures the average amount of data stored by each endsystem.

For each of these parameters, we choose values from real-world scenarios. The availability parameters are derived from the Farsite availability traces [8], a 4-week long measurement of availability characteristics across an enterprise network. The data update rate and data size are based on our measurements of Anemone packet data, with each endsystem storing its local packet data for 1 month. Table 1 summarizes these parameters as well as additional model-specific parameters used in some of the models.

4.2.1 *Centralized*

This is the simple "data warehousing" model where all available endsystem data is backhauled onto a single central repository before being queried. The maintenance costs thus lie in backhauling all the generated data, and are given by:

$$f_{on}Nb \tag{1}$$

4.2.2 *Seaweed*

The maintenance costs of Seaweed are driven by the replication of metadata. They also depend on the replication factor k. When an endsystem fails, the metadata stored by it must be replicated on some other endsystem to maintain k replicas. If all k replicas fail during the window of vulnerability between failure detection and replication, the data will become available. Thus the choice of k is a trade-off between overhead and availability, and depends on the environment. Typical values of k are between 3 and 8; here we choose a value of 4.

Seaweed replicates both the availability models and the data summaries, which have average sizes a and h respectively. Here h is the total compressed size per endsystem of all metadata, i.e. the histograms on all indexed attributes; in the Anemone case there are 5 such histograms per endsystem. Each available endsystem proactively pushes its metadata to its replicas, p times per second, at a bandwidth cost of $f_{on}Nkph$. Additionally, Seaweed incurs the cost of replicating both availability models and metadata whenever an endsystem joins or leaves the system. In the first case, the joiner must acquire the metadata that it will be responsible for. In the second case, the metadata held by the leaving endsystem must be re-replicated on some other endsystem. Since each endsystem has $h + a$ bytes of metadata on average which must be replicated k times, the total amount of replicated data is $Nk(h + a)$. This metadata must be replicated on the available nodes, thus each available node will store on average $\frac{1}{f_{on}}k(h + a)$ bytes. These bytes must be transferred on each churn event, consuming a bandwidth of $\frac{1}{f_{on}}Nck(h + a)$. Thus the total maintenance overhead for Seaweed is

$$f_{on}Nkph + \frac{1}{f_{on}}Nck(h + a) \tag{2}$$

4.2.3 *DHT-replicated*

Here we consider using a typical DHT approach to store the data: each tuple is mapped onto a key in the DHT based on its primary key, regardless of where it was generated. The tuple is k-way replicated on a replica set determined by the DHT key. This incurs the cost of transferring each new tuple from the generating endsystem to the k replicas, which is $f_{on}Nkb$.

Additionally, the DHT must pay the cost of re-replicating data when endsystems join or leave. The average amount of replicated data stored per endsystem is $\frac{1}{f_{on}}kd$, thus the bandwidth consumption of re-replication is $\frac{1}{f_{on}}Nckd$.

Thus the total maintenance bandwidth for this architecture is

$$f_{on}Nkb + \frac{1}{f_{on}}Nckd \tag{3}$$

This ignores the overhead of discovering the root of each tuple, each of which would typically require an $O(\log N)$ lookup over the network. The actual costs of this would depend on the distribution of primary keys, and we simplify the model by assuming the cost to be negligible.

4.2.4 *PIER*

Variable	Description	Value	Source
N	Number of endsystems	300,000	CorpNet
f_{on}	Fraction of available endsystems	0.81	Farsite
c	Churn rate	5.5 x 10^{-6} /s	Farsite
u	Data update rate per endsystem	970 bytes/s	Anemone
d	Database size per endsystem	2.6 GB	Anemone
k	Number of replicas stored	4	Farsite
h	Size of data summary	6,473 bytes	Seaweed/Anemone
a	Size of availability model	48 bytes	Seaweed
p	Summary push rate	0.033 /s	Seaweed (30 s period)
r	PIER data renewal rate	0.0033	PIER (300 s period)

Table 1: Model parameters

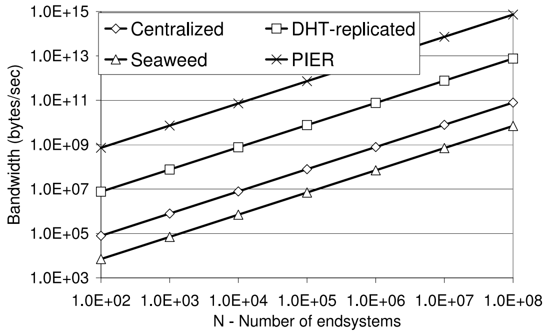

Figure 2: Overhead versus N.

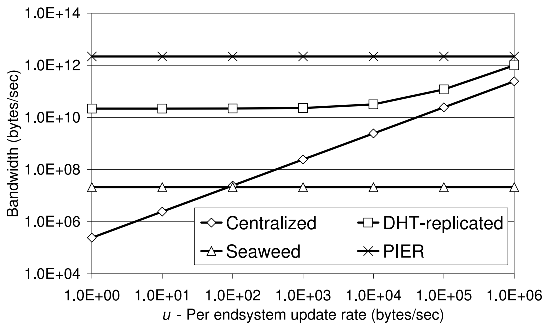

Figure 3: Overhead versus u.

PIER [16] uses a DHT but does not replicate data as described above. Instead, each available endsystem periodically re-inserts its data into the DHT, with tuples mapped to DHT keys according to their primary keys. However, the *renewal process* serves to maintain the freshness of the data as well as to provide additional availability when endsystems fail. Thus the maintenance overheads in PIER are independent of endsystem churn, and only depend on the data size d and the renewal rate r. The overhead is

$$f_{on}Ndr \qquad (4)$$

Note that avoiding churn-related overheads comes at a price: PIER cannot provide the same availability as k-way replication. Specifically, if both the source and the root of some tuple fail, then the tuple will become unavailable.

4.2.5 Comparison

We use the four models to compare the scalability of the different solutions. Specifically, we compare the scalability of maintenance overheads with increasing network size (N), database size per endsystem (d), and data update rate per endsystem (u), in each case keeping all the other parameters constant with the values in Table 1.

Figure 2 shows how these different systems scale with network size N. Note that both axes are on a log scale, to illustrate the order-of-magnitude effects involved. The total system bandwidth for all the designs increases linearly with N, however there are order-of-magnitude differences in the constant factors involved. Essentially, all PIER endsystems must periodically refresh all their data at a rate r, causing a very high overhead. The DHT-replication scheme must replicate each endsystem's data at a rate proportional to the churn rate c. The factor for the centralized system is the data update rate u. Finally, Seaweed's overhead depends on the churn rate c as well as the metadata size. Since the metadata is orders of magnitude smaller than the data, Seaweed has correspondingly lower overhead: 10 times lower than the centralized solution, and 1000 or more times lower than the other distributed solutions.

Figure 3 shows the system-wide bandwidth in bytes per second for each of the models as the number of bytes generated per second per online endsystem u is varied. u is shown on a linear scale, but bandwidth is on a log-scale, since the different systems have order-of-magnitude differences in their overheads.

We see that PIER's overhead is independent of u but extremely large, due to the periodic reinsertion of the entire database into the network. DHT-replication incurs both the overhead of replicating fresh data, which depends on u, and of replicating on endsystem churn, which is independent of u. The latter dominates and thus DHT-replication also has a near-constant and high overhead, though two orders of magnitude less than that of PIER. The centralized system has no churn-related overheads, and its overhead scales linearly with the data update rate. Finally, Seaweed overheads are independent of data update rate, and also several orders of magnitude lower than either DHT-replicated or PIER.

When the update rate u is low, the centralized approach will require lower overhead than Seaweed. As the data rate increases, the overhead of metadata replication becomes small compared to that of sending the data to the centralized database. At the Anemone update rate of 970 bytes/s per endsystem, a relatively modest rate for today's endsystems, Seaweed already outperforms the centralized solution by a factor of 10. Thus Seaweed demonstrates better scalability

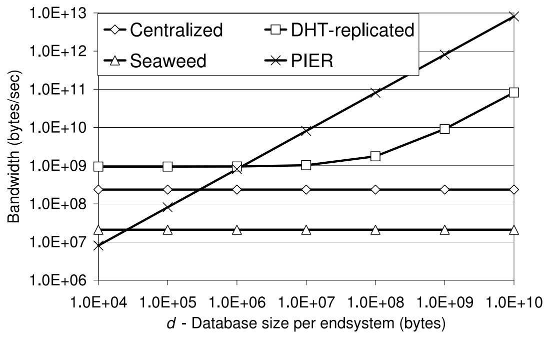

Figure 4: Overhead versus d.

than the centralized approach, as well as orders of magnitude lower overhead than the other distributed approaches.

Figure 4 compares the scalability of the four designs with increasing database size per endsystem d. PIER's overhead is dominated by the cost of periodic reinsertion, which is linear in d. The cost of DHT-replication is dominated by the need to replicate some of the data on each churn event, also linear in d but with a much smaller factor than PIER. The cost of the centralized solution is independent of database size, depending only on the rate u at which new data is generate. Finally, Seaweed is also independent of d, and has orders of magnitude lower overhead than all the other designs.

The other component of the network overhead of distributed querying is that of performing a query. The centralized solution does not incur any networking costs for querying, whereas all three decentralized architectures have a cost that is linear in the network size. In practice, the query cost is dominated by the size of the result; for aggregation queries with in-network aggregation, it is negligible compared to the maintenance overheads, as we show in our simulation results.

4.2.6 Summary

Our simplified analytic models do not capture many real-world engineering optimizations that each implementation could employ. However, we believe that they capture the general scalability issues of each approach. Our analysis demonstrates that Seaweed's design is much more scalable in terms of maintenance overhead than the other approaches. Although this comes at the cost of increased query latency when compared to the centralized and DHT-replicated solutions, we believe that for truly scalable, highly distributed querying, this price must be paid to avoid prohibitive network costs.

4.3 Simulation

Here we present results from a discrete event simulator that allows us to evaluate the scaling properties of Seaweed. The simulations are driven by real-world application data, traces of endsystem availability, and network topologies.

The difficulties of running a discrete event simulator at this scale should not be underestimated: we have thousands of endsystems, the events to be simulated occur at the granularity of milliseconds and we simulate them over a period of 4 weeks. We made some optimizations that would not affect our evaluation metrics. We pre-computed the results of each query as well as the histograms on all endsystem data, by loading each endsystem's data into SQL Server 2005, running the queries on them and also extracting all histograms on indexed attributes. This enabled the simulation to run much faster by not executing a large number of database queries during the simulation.

These optimizations did prevent us from supporting dataset update during simulation. In our experiments we pessimistically assume the total data size as of the end of the trace, i.e. containing all the packet and flow data irrespective of the query time. Further, since we could no longer tell if the histogram data would change in any given push interval, we push the histograms with an average period of 17.5 min, with each endsystem choosing its push time randomly to avoid spikes in network bandwidth.

Unfortunately, these optimizations still did not allow simulation of more than 6,000 endsystems in a reasonable amount of time. Thus our overhead results are based on simulations of up to 6000 endsystems. Our evaluation of the completeness prediction uses a simplified simulator that avoids packet-level network simulation, and thus runs at the full scale of 51,663 endsystems, which is the size of our availability data set.

4.3.1 Experimental setup

We generated an Anemone application data set for the endsystems by instrumenting the network routers in our building. This enabled us to capture a complete packet trace of all inter-LAN traffic from the 30th August 2005 to the 20th September 2005 for 456 workstations and servers. This is representative of though not identical with the data from a full endsystem-based deployment of Anemone. The raw packet data was processed to generate per-endsystem `Flow` and `Packet` tables.

Simulated endsystem availability is based on the *Farsite* trace of endsystem availability gathered over approximately 4 weeks in July/August 1999 in the Microsoft Corporate network [8]. The trace was generated using hourly pings to detect whether each of 51,663 endsystems was connected to the corporate network.

The network simulation results presented here use the *Corpnet topology*, which has 298 routers generated from measurements of the world-wide Microsoft corporate network. The topology includes the minimum round trip time (RTT) per-link and this in used as the proximity metric in the simulations. Each endsystem was directly attached by a LAN link with delay of 1ms to a randomly chosen router.

Our simulations were run at a number of different network sizes. In each case, each simulated endsystem was given an availability profile randomly selected from the availability trace and an endsystem data set randomly selected from the Anemone data.

MSPastry was configured to use $b = 4$ and a leafset size of $l = 8$, with the leafset heartbeat rate set to 30 seconds. Seaweed was configured with a replication factor of $k = 3$ for the result tree vertexes and $k = 8$ for the metadata.

4.3.2 Completeness prediction

The first set of experiments evaluates the ability of Seaweed to generate accurate completeness predictions. The experiments were run using the full Farsite set of 51,633 endsystems. We simulated from the 6th July 1999 onward and injected queries into the system at various points during the work week starting Monday 19th July 1999. The

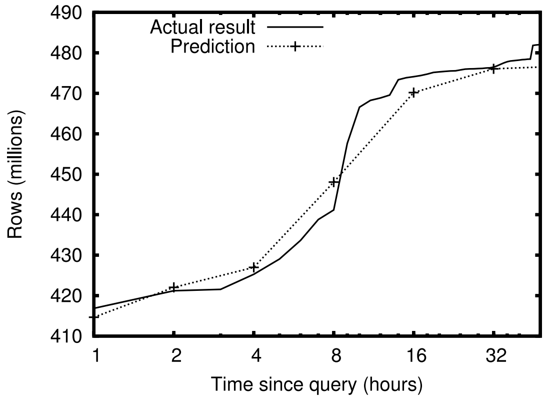

Figure 5: Predicted completeness versus actual completeness overtime for the query `SELECT SUM(CAST(Bytes As Float)) FROM Flow WHERE SrcPort=80` **injected on Tuesday 20th July 1999 at 00:00.**

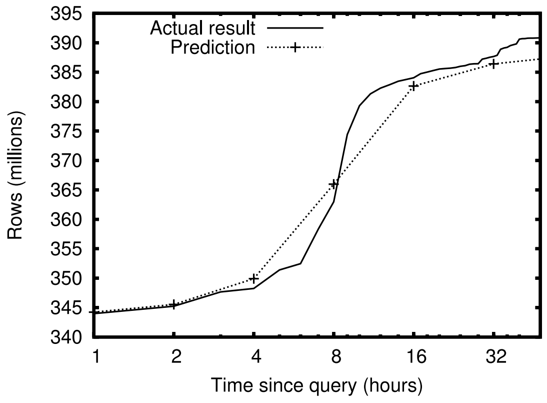

Figure 6: Predicted completeness versus actual completeness overtime for the query `SELECT COUNT(*) FROM Flow WHERE Bytes > 20000` **injected on Thursday 22th July 1999 at 00:00.**

warmup period allowed each endsystem to learn an availability model. For each injected query we generated the completeness predictor and then monitored the actual results returned over the 48 hours after injection, after which the query was terminated.

Figure 5 compares the completeness predictions generated when the query is injected with the actual completeness observed over time. The query was injected on Tuesday 20th July 1999 at 00:00, and actual query was:

```
SELECT SUM(CAST(Bytes As Float)) FROM Flow
WHERE SrcPort=80
```

which captures the amount of `http` traffic in the network. The completeness prediction is shown as a cumulative function of rows queried against time: Figure 5 shows that it matches the observed result well. Note that when the query is first injected only 85% of the matching rows are available. After approximately 8 hours, when the employees arrive at work there is a significant increase in the number of rows queried, which is accurately predicted.

Figure 6 shows the same results for a second query:
`SELECT COUNT(*) FROM Flow WHERE Bytes > 20000`,
which is injected on Thursday 22th July 1999 at 00:00. The query examines the number of flows with significant number of bytes. As with the previous results the completeness prediction closely matches the observed completeness over time. When the query is first injected only 87% of the rows expected to be queried are available.

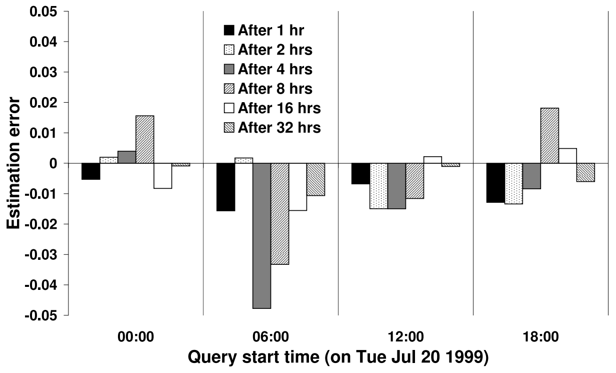

Figure 7: Relative error `SELECT SUM(CAST(Bytes As Float)) FROM Flow WHERE SrcPort=80.`

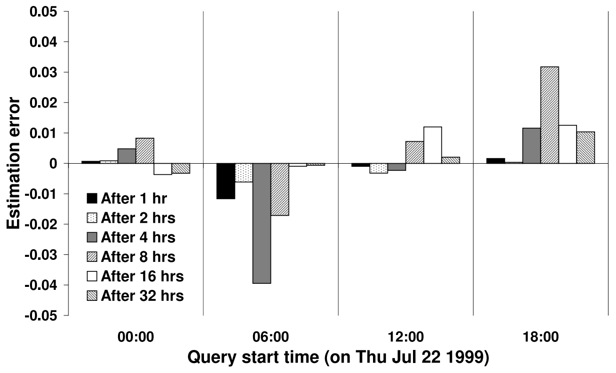

Figure 8: Relative error `SELECT COUNT(*) FROM Flow WHERE Bytes > 20000.`

Figures 7 and 8 show the relative error for the previous two queries injected every 6 hours during the selected day (00:00, 06:00, 12:00 and 18:00). The completeness predictions generated at query time provide the predicted completeness after 1, 2, 4, 8, 16, and 32 hours. We show the relative error of predicted completeness compared to the observed result for all these time periods. Prediction error is low, less than 5% in all cases.

We have determined that the primary source of error for these queries is in the availability prediction. Row count estimation is extremely accurate for queries such as these with range predicates involving a single, indexed attribute. We are currently exploring summarization techniques that will enable accurate estimation for more sophisticated queries.

4.3.3 *Overhead performance*

The second set of experiments measures the overheads of running Seaweed using the packet-level simulator. We ran experiments to measure the bandwidth overhead with different network sizes. For each run we simulated from the 6th July 1999 to the 9th August 1999. We injected the query

```
SELECT SUM(CAST(Bytes As Float)) FROM Flow
WHERE SrcPort=80
```

on Tuesday 20th July 1999 at 00:00. In this case we allowed

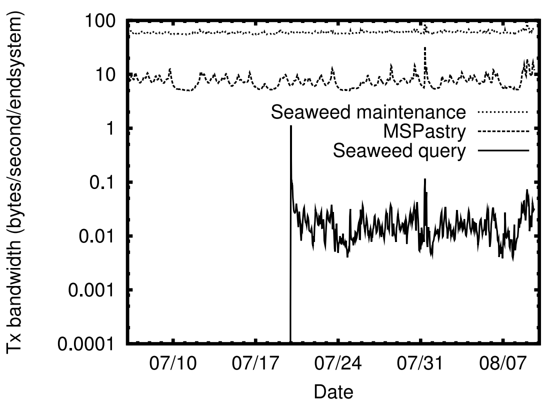

Figure 9: Overhead over time for a query injected on 07/20 at 00:00

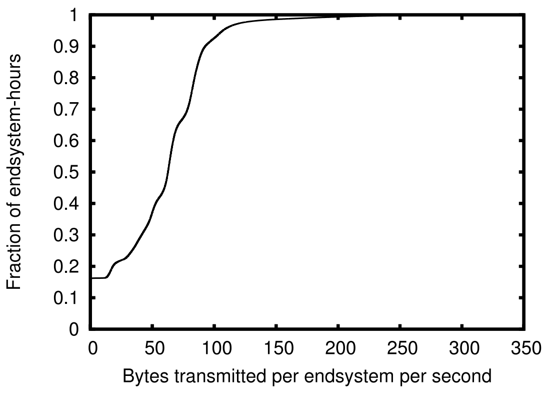

Figure 10: CDF of endsystem versus number of bytes transmitted.

the query to run until the end of the simulation to test the long-term performance and stability of the prototype.

Figure 9 shows the overhead in bytes per second per endsystem when running with 12,000 endsystems. On average there are 9662 endsystems online and the graph shows the bytes per second per online endsystem. The overhead is sub-divided into the MSPastry overhead, the Seaweed maintenance overhead and the query overhead. In general this shows that the overhead is low, in total less than 100 bytes per second per endsystem. The Seaweed maintenance traffic is the highest overhead and is dominated by the cost of periodically replicating the indexed attribute histograms, and could be substantially reduced by using some form of delta encoding between successive histogram versions. However, even without this optimization the overhead is so low as to be insignificant.

Figure 10 shows the distribution of network load from each endsystem over time, as a cumulative distribution of the average bytes transmitted per second. The maximum is 3540 bytes per second, while the 99th percentile is only 175 bytes per second. The distribution of bytes received per second is similar. This shows that the overhead is not only low overall but also evenly distributed across all the endsystems and across time.

The next results examine the overhead as the number of endsystems in the network (N) is varied between 2,000 and 12,000. Figure 11 shows the overhead in bytes per second per

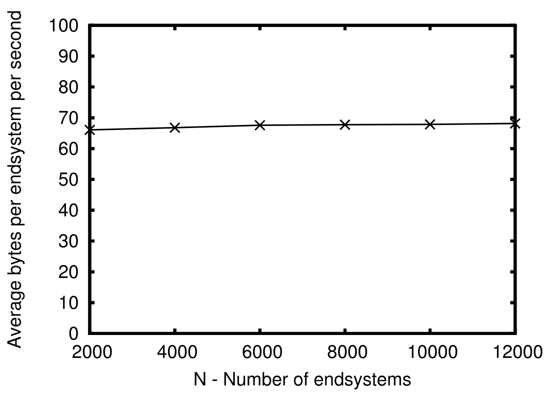

Figure 11: Overhead versus number of endsystems.

endsystem versus total number of endsystems. The Seaweed maintenance overhead per endsystem, which dominates, is $O(1)$. Both the Seaweed query overhead and MSPastry overhead grow with $O(\log N)$. However, the MSPastry overhead is an order of magnitude less than the Seaweed maintenance overhead, and the query overhead is three orders of magnitude less. This leads us to believe that the design will scale to 1,000,000 endsystems or more.

Finally, we evaluated the latency between query injection and returning the completeness predictions to the user. In the worst case (with 12,000 endsystems) this latency was 10.1 s. We feel that this is an acceptable latency for queries whose actual execution could take minutes or hours due to endsystem unavailability.

5. RELATED WORK

We have already discussed PIER [16]; here we mention a selection of other related work.

Distributed information management. Systems that support distributed information management such as Astrolabe [31] and SDIMS [32] build aggregation trees supporting continuous queries using user-defined aggregation functions. Queries are injected into the system and continuously compute summaries of data. In contrast, Seaweed aims to support one-shot queries across stored data and so is principally concerned with problems due to data unavailability.

Distributed indexes. Earlier work in the field of distributed databases provided index structures [19, 20, 21] to enable efficient search for distributed data and distributed updates with strong consistency semantics. More recently, distributed indexes using various peer-to-peer structures have been designed [1, 6, 18, 27]. These provide efficient access to and range queries over data distributed over many endsystems.

In contrast, Seaweed replicates neither indexes nor data, aiming for far greater scalability by only replicating compact data summaries. This permits queries to be failure-tolerant, remaining in the system to access data on currently unavailable endsystems. For applications with sufficiently high query rates, distributed index structures may prove useful but the requirement that data not be moved away from the producing endsystem still holds.

Data stream management. Due partly to the recent popularity of sensor networks, executing long-running queries over multiple data streams is an extremely active research area. A wide range of large-scale systems have been built which route tuples through long-standing pre-installed queries [2, 3, 10, 13, 23, 30]. Borealis [3] deals with data unavailability on much smaller time scale than Seaweed, buffering stream data to tolerate transient network failures on the order of a minute.

In contrast, Seaweed leaves data where it is generated and supports efficient one-shot select-project-aggregate queries on stored data, which is sufficient for a wide variety of useful and interesting applications. This requires that we deal with endsystem unavailability on the scale of hours to days.

Availability models and data summarization. A key feature of Seaweed is the prediction of endsystem availability and the ability to estimate row count from data summaries. Seaweed uses a very simple availability predictor. Concurrently with this work, others have developed alternative predictors [24], which could potentially improve Seaweed's performance. Similarly, the data summaries currently distributed in Seaweed are relatively simple: just the histograms computed by the local DBMS across manually selected attributes. PTQs [11], or histogram-based approximation approaches [17] could both be promising ways to provide data summaries that permit accurate estimation of the data stored on an unavailable endsystem.

Online aggregation. Online aggregation was first proposed by Hellerstein et al. [15] in the context of single-site databases, along with statistical estimators of result accuracy. Seaweed uses row-count based estimates of completeness rather than estimators of result accuracy as there is no guarantee that incrementally processed tuples will be in random order: the data being queried may well be correlated with endsystem availability.

6. CONCLUSION

In this paper we describe Seaweed a query infrastructure for highly distributed data sets. The major challenge for such systems is managing the unavailability of endsystems in a scalable manner. Prior systems use replication, which fundamentally limits their scalability.

Seaweed adopts a different approach, *delay aware querying*. Rather than replicating the data, Seaweed replicates only metadata and uses this to provide the user with a completeness predictor. The predictor allows the user to estimate the completeness of the result so far and also the expected future progress. The Seaweed approach is scalable but trades off query latency for scalability.

The analysis and simulation results show that Seaweed scales well. The simulation results show that replicating the metadata allows the generation of accurate completeness predictors. To conclude, it seems that Seaweed represents a novel and interesting point in the design space for query infrastructures for highly distributed data sets.

7. REFERENCES

[1] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt. Indexing data-oriented overlay networks. In *VLDB*, pages 685–696, Trondheim, Norway, Aug. 2005.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, Dallas, TX, May 2000.

[3] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD*, pages 13–24, Baltimore, MD, June 2005.

[4] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *SIGMOD*, pages 515–526, 2004.

[5] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 256–267, Feb. 2003.

[6] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, pages 353–366, Portland, OR, Aug. 2004.

[7] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *HotOS-IX*, pages 1–6, Kauai, HA, May 2003.

[8] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *SIGMETRICS*, pages 34–43, Santa Clara, CA, June 2000.

[9] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *DSN*, pages 9–18, Florence, Italy, June 2004.

[10] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *SIGMOD*, pages 379–390, Dallas, TX, May 2000.

[11] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *VLDB*, pages 876–887, Toronto, CN, Aug. 2004.

[12] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 33–44, 2003.

[13] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, Toronto, CN, Aug. 2004.

[14] A. Y. Halevy, N. Ashish, D. Bitton, M. J. Carey, D. Draper, J. Pollock, A. Rosenthal, and V. Sikka. Enterprise information integration: successes, challenges and controversies. In *SIGMOD*, pages 778–787, Baltimore, MD, June 2005.

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, pages 171–182, Tucson, AZ, May 1997.

[16] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, pages 321–332, Berlin, Germany, Sept. 2003.

[17] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, pages 174–185, Edinburgh, UK, Sept. 1999.

[18] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, Trondheim, Norway, 2005.

[19] T. Johnson and P. Krishna. Lazy updates for distributed search structure. In *SIGMOD*, Washington DC, USA, May 1993.

[20] W. Litwin, M.-A. Neimat, and D. A. Schneider. RP*: A family of order preserving scalable distributed data structures. In *VLDB*, pages 342–353, Santiago de

Chile, Chile, Sept. 1994.

[21] D. B. Lomet. Replicated indexes for distributed data. In *PDIS*, pages 108–119, Miami Beach, FL, Dec. 1996.

[22] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P file-sharing with an internet-scale query processor. In *VLDB*, pages 432–443, Toronto, CN, Aug. 2004.

[23] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60, Madison, WI, June 2002. ACM.

[24] J. W. Mickens and B. D. Noble. Exploiting availability prediction in distributed systems. In *NSDI (to appear)*, San Jose, CA, May 2006.

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

[26] R. Mortier, R. Isaacs, and P. Barham. Anemone: using end-systems as a rich network management platform. In *SIGCOMM MineNet*, Philadelphia, PA, Aug. 2005.

[27] R. Mortier, D. Narayanan, A. Donnelly, and A. Rowstron. Seaweed: Distributed scalable ad-hoc quering. In *NetDB Workshop*, Atlanta, GA, Apr. 2006.

[28] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, Nov. 2001.

[29] S. Saroiu, K. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *MMCN*, San Jose, CA, Jan. 2002.

[30] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, Berlin, Germany, Sept. 2003.

[31] R. Van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2):164–206, 2003.

[32] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM*, Portland, OR, Sept. 2004.