Dushyanth Narayanan · Austin Donnelly · Richard Mortier · Antony
Rowstron

# Delay Aware Querying with Seaweed

**Abstract** Large highly distributed data sets are poorly
supported by current query technologies. Applications
such as endsystem-based network management are char-
acterized by data stored on large numbers of endsys-
tems, with frequent local updates and relatively infre-
quent 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 endsys-
tems and their data will be unavailable at any given
time. Existing methods to provide high data availabil-
ity 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 endsys-
tems 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
incremental results, online in-network aggregation and
completeness prediction. It is built on a distributed hash
table (DHT) but unlike previous DHT based approaches
it does not redistribute data across the network. It ex-
ploits the DHT infrastructure for failure-resilient meta-
data replication, query dissemination, and result aggre-
gation. We analytically compare Seaweed's scalability
against other approaches and also evaluate 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

Microsoft Research, 7 JJ Thomson Avenue, Cambridge CB3
0FB, United Kingdom
E-mail: {dnarayan,austind,mort,antr}@microsoft.com

scalable distributed query infrastructure. Recent research
has looked at building such infrastructures [1, 16, 22, 31,
32]. The challenges are *availability* and *scalability*. A sig-
nificant fraction of endsystems will be unavailable at any
given time due to network outages, endsystem failures,
and scheduled downtimes, which means the system must
tolerate 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 require centralization, redistribution, or repli-
cation 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 re-injects all its tu-
ples into the network, requiring network bandwidth lin-
ear in the product of network size, per-endsystem data
size, and refresh rate. We believe that storing data any-
where but on the endsystem where it is produced funda-
mentally limits scalability.

In this paper we present Seaweed, a scalable query
infrastructure which solves the problem of data unavail-
ability by allowing queries to persist until unavailable
data becomes available. By querying data entirely *in-
situ*, Seaweed scales well with network size, data size,
and data update rate. Query results are updated incre-
mentally with completeness improving over time, where
completeness is defined as the ratio of tuples processed
to the total number of tuples relevant to the query. Sea-
weed addresses data unavailability through an explicit
trade-off between completeness and delay, by providing
the user with estimates of current completeness and pre-
dictions of future completeness.

### 1.1 Our contributions

Previous approaches have addressed the problem of data
unavailability using data replication which fundamen-
tally 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, exposing to the user a prediction of the expected delay to reach any given level of completeness. This is achieved by estimating the amount of data relevant to a query held by each currently unavailable endsystem, and also predicting when it will next become available. This in turn requires replication of a small amount of per-endsystem metadata consisting of a compact data summary and an *availability model*.

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 overlay structure to build efficient, failure-resilient protocols for query dissemination and result aggregation.

We show through analytic models that Seaweed scales better with network size, data size, and data update rate than approaches based on data replication, centralization, or redistribution. We also show through simulation results that Seaweed efficiently disseminates queries, generates accurate completeness predictors, and aggregates query results.

## 1.2 Applications

Many applications are enabled by scalable query 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 systems. Effective analysis and diagnosis based on this data requires distributed querying support.

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 with lower network overheads.

These applications are characterized by their scale, as well as the need to support *one-shot queries* and not 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 8:30 and 9:00?"), 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 base tables. By restricting read-only queries to be single-table and updates to be single-endsystem, we gain scalability at the cost of restricted query functionality. This seems an acceptable trade-off for the 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 we target applications with a small number of human users such as network administrators who issue one-shot queries. We evaluated the benefits of maintaining distributed indexes for these applications and concluded that they do not justify the resulting overheads and complexity.

## 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 summarizes related work, and Section 6 concludes the paper.

## 2 Design principles and insights

For simplicity, we use standard data models and query 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 query distribution tree that disseminates the query to all available endsystems, which then return completeness predictors that are aggregated back up to the root. The predictor at the root can 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 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 which are propagated to the root using the tree. Incremental results will thus continue to arrive for any query until it times out or is explicitly canceled.

### 2.1 Availability and delay-aware querying

Endsystem availability is a major challenge for any distributed query 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 is unavailable at any time. Figure 1, reproduced from one such
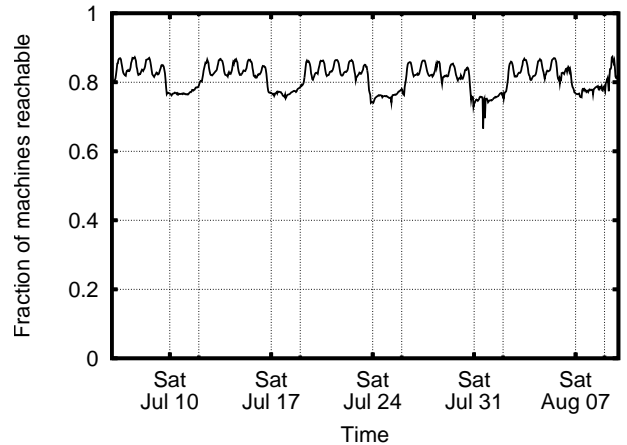


Fig. 1: Availability of 51,663 endsystems on the Microsoft corporate network in July/August 1999.

study [8], shows the availability of 51,663 endsystems on the Microsoft corporate network, 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, a clear periodic pattern suggests endsystem availability is predictable.

Therefore a guiding principle for all scalable distributed query 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. She might then decide to accept the results after 1 hour and then cancel the query rather than waiting for perfect completeness. Completeness predictors are *query-specific*: they depend on the data rows that are relevant to the query and also on the distribution of these rows across available and unavailable endsystems. Figure 2 shows an example completeness predictor.

### 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 replicated metadata is several orders
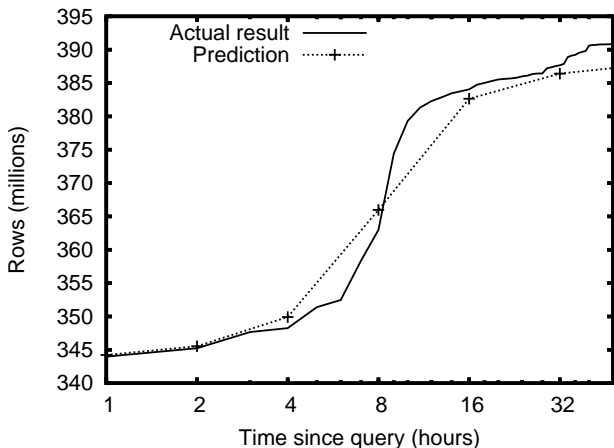
Fig. 2: Example of a completeness predictor.

of magnitude smaller than the raw data. Second, Seaweed's 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 the time range $[t_1, t_2]$. 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 a query. Consider a user who injects a query into Seaweed at time 0 and observes the partial result at time $T$ generated by some set of endsystems $H$. Single-site validity would guarantee only that $H_C(0, T) \subseteq H \subseteq H_U(0, T)$ whereas Seaweed guarantees $H = H_U(0, T)$, a strictly stronger semantics.

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.

Seaweed is able to provide these semantics because with high probability each endsystem's contribution to the result is counted *exactly once* provided it becomes available during the lifetime of the query, $[0, T]$, and remains available for long enough to receive and process the query. This property is provided by the distributed data structures described in the next section. At-least-once counting results from the property that any endsystem in Seaweed available for sufficient time will have a path to the root with high probability. At-most-once counting is achieved through persistent, replicated versioning of messages in the aggregation tree.

## 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 circular 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 is expected to deliver messages in $O(\log_{2^b} N)$ hops.

Our Seaweed implementation is built on the MSPastry [9] implementation of Pastry. MSPastry provides a low-level key-based routing (KBR) API [12] which is used by Seaweed. 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 $1.6 \times 10^{-5}$ [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. Seaweed provides an application-independent metadata replication service, where the metadata consists of column histograms and availability models. The replication frequency and the set of histograms are application-specific parameters. This proactively replicated metadata is query-independent and allows us to compute query-specific completeness predictors when a query is injected.

For any endsystem with endsystemId $x$, Seaweed replicates the metadata on the $k$ numerically closest endsystems to $x$. As endsystems join and fail new replicas are generated as necessary to ensure that this holds. The $k$ endsystems storing the metadata for $x$ form its *replica set*. The replication messages are routed in a single Pastry hop, and thus both the network latency and the bandwidth usage are small. When $x$ becomes unavailable any of the $k$ members of $x$'s replica set can generate a completeness predictor for any query on behalf of $x$.

### 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? Availability prediction must be done on a per-endsystem basis since the effect of an endsystem's availability on completeness depends on the number of query-relevant rows on that endsystem.

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 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 records the time at which this occurred. Subsequently 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 histograms on indexed columns of the local database. When an available endsystem generates a row count estimate for a query on its own behalf, it queries the local DBMS for the estimate. When row count estimation is done on behalf of an unavailable endsystem, it uses 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 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.

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

## 3.3 Query dissemination and completeness prediction

When a query is submitted to Seaweed the first stage is to disseminate it to all available endsystems and 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 exactly-once semantics are maintained even as endsystems concurrently join and fail during the process.

For robust query dissemination and completeness predictor generation, 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 (where $b$ is typically 4). 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, within one Pastry hop in the common case where the sender has a Pastry routing table entry for a live endsystem within that subrange.

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 distribution 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 row-count distributions 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 it the query. 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 structure of Pastry routing tables to achieve a message overhead of $O(N)$. It relies on the property that when a broadcast is forwarded by endsystem $x$ to a subrange, with high probability $x$'s routing table contains a live endsystem $y$ in that subrange. 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 endsystems become available their results need to be included in the result, and results submitted by endsystems need to persist

even if the endsystem fails. This means that a different tree must be built since churn now becomes a significant factor. Simply relying on aggressive retransmission for failure-resilience is infeasible.

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 at the root of all endsystems that have contributed results is not feasible, as this will result in messages of size $O(N)$.

Instead, we dynamically build an aggregation tree from the leaves upwards, maintaining $O(1)$ information in each endsystem in the aggregation tree: the current results received from each child. When new results are received from any child, this list of child results is updated, and a new aggregate is computed and forwarded up the tree.

The aggregation tree is embedded in the Pastry namespace and is unique for each *queryId*. Each tree vertex is a key in the Pastry namespace, referred to as a *vertexId*. Given a vertexId each endsystem is able to determine its parent vertexId in the tree, using a deterministic function: $V(queryId, vertexId) \mapsto vertexId$. Many different functions could be used, which provide different properties. We use a simple function that provides good load distribution:

V(queryId,vertexId)
    *int* len := PREFIXLENGTH(queryId,vertexId)
    *return* PREFIX(vertexId, $\frac{128}{b}$-(len+1)) +
           SUFFIX(queryId, len+1)

where PREFIXLENGTH(idA,idB) returns the length of the common prefix match between the keys idA and idB (expressed in base $2^b$). PREFIX(id, count) and SUFFIX(id, count) return the prefix and suffix of key id of length count, respectively. The operator + concatenates a suffix and prefix to generate a vertexId key. When endsystem $x$ submits a result to the aggregation tree it determines its parent in the tree using V(queryId,x). In general, given a vertexId any endsystem can determine the parent vertexId. The function ensures that the vertexId of the root is queryId. Due to the Pastry namespace being sparsely populated with endsystems, we optimize the function to create a tree of depth $O(\log N)$ rooted at the query originator, where $N$ is the number of endsystems. When an endsystem $x$ submits its result, it repeatedly applies V (starting from its own endsystemId $x$) until it generates a vertexId to which $x$ is no longer the numerically closest endsystemId. It then persists that vertexId with the query and submits its result to that vertexId. At higher levels in the tree, the function V is applied exactly once to compute the parent's vertexId. This optimization ensures that the tree has $N$ leaves, and hence a depth of $O(\log N)$.

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 $m$ 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 parent. If any member of the group fails then a new endsystem joins the group and a new primary is selected automatically if necessary, always with 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 seconds.

The same protocol can be extended easily 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 with respect to network overheads, network size, data size, data update rate, and endsystem churn.

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 stand-alone. 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 endsystem captures its network activity into two tables, `Packet` and `Flow`. Each record in `Packet` 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. `Flow` 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, ports, and protocol, and the number of bytes and packets sent and received. The flow measurement interval is currently set to 5 min.

A typical query on `Flow` by a network operator might be:
```
SELECT SUM(Bytes) FROM Flow WHERE SrcPort=80
AND 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 endsystem's timestamp and compared locally against each endsystem's local timestamp, assuming loose clock synchronization across endsystems.

Seaweed will disseminate the query to all endsystems; generate a predictor of completeness 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 analytical models of Seaweed and 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 in bytes per 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; we assume here 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 based on real-world enterprise networks. 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.

| Variable | Description | Value | Source |
|---|---|---|---|
| $N$ | Number of endsystems | 300,000 | Microsoft CorpNet |
| $f_{on}$ | Fraction of available endsystems | 0.81 | Farsite |
| $c$ | Churn rate | $6.9 \times 10^{-6}\,s^{-1}$ | Farsite |
| $u$ | Data update rate per endsystem | $970\,\text{bytes/s}$ | Anemone |
| $d$ | Database size per endsystem | $2.6\,\text{GB}$ | 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 | $0.033\,s^{-1}$ | Seaweed (30 s period) |
| $r$ | PIER data refresh rate | $0.0033\,s^{-1}$ or $0.00028\,s^{-1}$ | PIER (5 mins or 1 hr period) |

Table 1: Model parameters

### 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:

$$f_{on}Nu \qquad (1)$$

### 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 and replication, the data will become unavailable. 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 availability models and 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 columns; in the Anemone case there are 5 such histograms per endsystem. Each available endsystem proactively pushes its metadata to its replicas at rate $p$, at a bandwidth cost of $f_{on}Nkph$. Additionally, Seaweed incurs the cost of replicating 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 endsystems, each of which will store on average $\frac{1}{f_{on}}k(h + a)$ bytes. These bytes must be transferred on each churn event at a bandwidth cost of $\frac{1}{f_{on}}Nck(h + a)$. Thus Seaweed's total maintenance overhead is:

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

### 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 $f_{on}Nku$.

Additionally, the DHT must re-replicate data whenever 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 DHT-replication requires a total maintenance bandwidth of:

$$f_{on}Nku + \frac{1}{f_{on}}Nckd \qquad (3)$$

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. We simplify the model by assuming this overhead to be negligible.

### 4.2.4 PIER

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. This refresh 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 refresh 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 in the face of churn as $k$-way replication. The root corresponding to any key will change whenever the current root becomes unavailable or a new endsystem whose ID is closer to the key becomes available. Tuples with that key will then be unavailable for querying until the next refresh.

| Time since last refresh | 5 min | 1 hour | 12 hours |
|---|---|---|---|
| Availability (Farsite) | 99.8% | 98.0% | 78.9% |
| Availability (Gnutella) | 97.3% | 71.6% | 1.8% |

Table 2: Expected availability in PIER

For each source, the expected fraction of available tuples decays exponentially as $e^{-ct}$ where $c$ is the churn rate and $t$ is the time since the last refresh by that source.

Since PIER is targeted at Internet environments, the recommended policy is aggressive refresh, with a period of 5 min. However, this can have substantial network overheads: in this paper we evaluate PIER both with a 5 min period and a less aggressive 1 hour period.

This analysis assumes that source endsystems are always able to refresh periodically. In reality, even enterprise networks have significant numbers of endsystems that become unavailable for much longer than the refresh period: e.g. desktop machines might be turned off overnight.

Table 2 shows the expected availability of a source endsystem's tuples 5 min, 1 hour, and 12 hours after the last refresh. These values represent a short refresh period, a long refresh period, and a long downtime respectively. We show this for two values of churn: a low value from the Farsite enterprise network study [8], and a higher value from Gnutella traces [29].

### 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 ($d$), data update rate per endsystem ($u$), and endsystem churn rate ($c$), in each case keeping all the other parameters constant with the values in Table 1. We show PIER configured with two different refresh periods: 5 min and 1 hour. The former setting causes a higher network overhead, but the latter results in increased staleness of queryable data and reduces availability when endsystems fail.

Figure 3(a) shows how these different systems scale with network size $N$. 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 $N$, but there are order-of-magnitude differences in the constant factors involved. 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 $c$ and 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(b) shows the system-wide bandwidth in bytes per second for each of the models as $u$, the number of bytes generated per second per online endsystem, is varied. PIER's overhead is independent of $u$ but extremely large, due to the periodic reinsertion of the entire database into the network. Even if we increase the reinsertion period from the default 5 min to 1 hour, PIER's overheads are still typically orders of magnitudes larger than the other designs. DHT-replication incurs both the overhead of replicating fresh data, which depends on $u$, and of replicating on endsystem churn, which is independent of $u$. Thus DHT-replication outperforms PIER by two orders of magnitude at low update rates but approaches and then exceeds the overhead of PIER at high update rates. 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 per second per endsystem, a relatively modest rate for today's endsystems, Seaweed already outperforms the centralized solution by a factor of 10. Thus Seaweed scales better than the centralized approach and has orders of magnitude lower overhead than the other distributed approaches.

Figure 3(c) shows 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 $d$. DHT-replication's overhead is due to re-replication of data on churn, also linear in $d$ but with a much smaller constant factor than PIER. The cost of the centralized solution is independent of $d$, depending only on the data update rate $u$. Finally, Seaweed's overhead is also independent of $d$, and is orders of magnitude lower than that of the other designs except for very small values of $d$.

Figure 3(d) shows the overhead as a function of the churn rate $c$: the number of endsystem departures and arrivals per second, as a fraction of the total number of endsystems. PIER's overhead is independent of churn but very high, since it periodically refreshes data regardless of churn. DHT-replication has an overhead that is linear in the churn rate, since data must be re-replicated on each churn event. The centralized case has a fixed, churn-independent overhead. Finally, Seaweed's overhead is low, and mostly due to periodic metadata replication; the re-replication induced by churn is significant only at very high churn rates (a churn rate of $1 \times 10^{-2}$ corresponds to a mean endsystem lifetime of only 200 s).

Figure 4 shows the same analysis but with a smaller database size per endsystem (100 MB rather than 2.6 GB) and a smaller update rate (10 rather than 970 bytes/s).
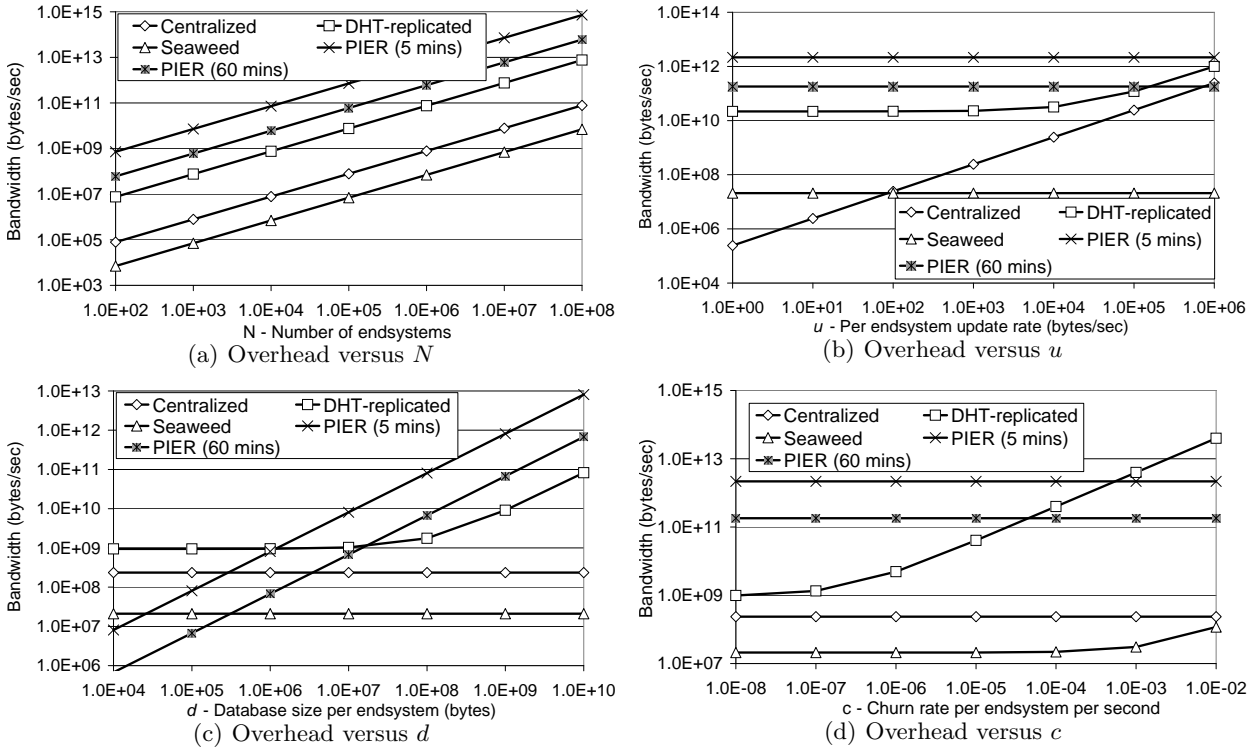
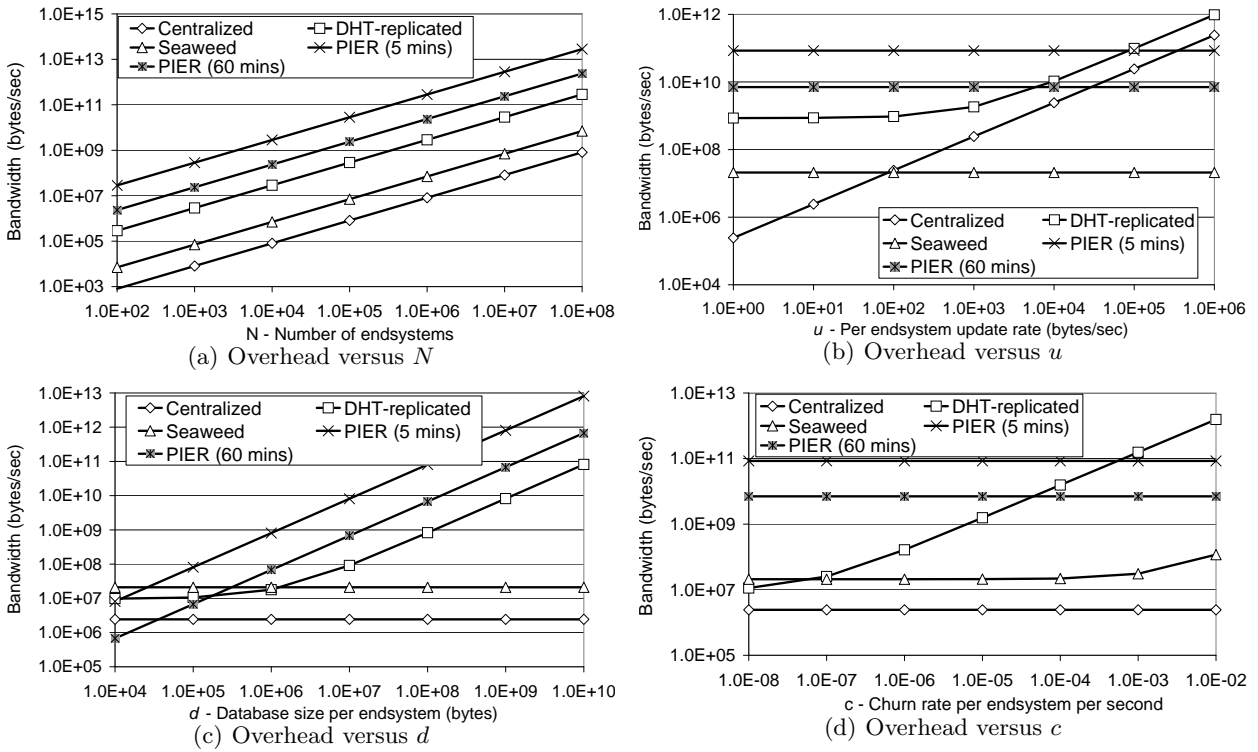Fig. 3: Scalability of network overheads



Fig. 4: Scalability of network overheads with a small database and low update rate

In general, a small database favors PIER over the other systems, and a small update rate favors the centralized approach. We see that the centralized approach is the best at these low update rates, since it has high availability and the lowest overheads. As before, PIER has competitive overheads only at small database sizes; further, it provides lower availability than centralized or DHT-replicated and unlike Seaweed provides no feedback on availability.

The other component of overhead is the foreground or per-query overhead. The centralized solution does not incur any networking costs for querying, whereas all three decentralized architectures have a cost that increases with network size. Simple analytical models of Seaweed's per-query overhead are difficult to derive: the overhead depends on the number of retransmissions and hence on the endsystem failure patterns. However, our simulation results show that in practice, Seaweed's per-query overheads are three orders of magnitude smaller than its background maintenance overhead, and hence will not be significant until there are thousands of active queries in the system at the same time.

### 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 shows that Seaweed's design is much more scalable in terms of maintenance overhead than the other approaches. Although this increases query latency 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 data updates 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.

### 4.3.1 Experimental setup

We generated an Anemone application data set for the endsystems by instrumenting the network routers in our building. We captured a complete packet trace of all inter-LAN traffic for the period 30 Aug 2005–20 Sep 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 test the availability of each of 51,663 endsystems on the 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 is used as the proximity metric in the simulations. Each endsystem was directly attached by a LAN link with delay of 1 ms to a randomly chosen router.

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

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

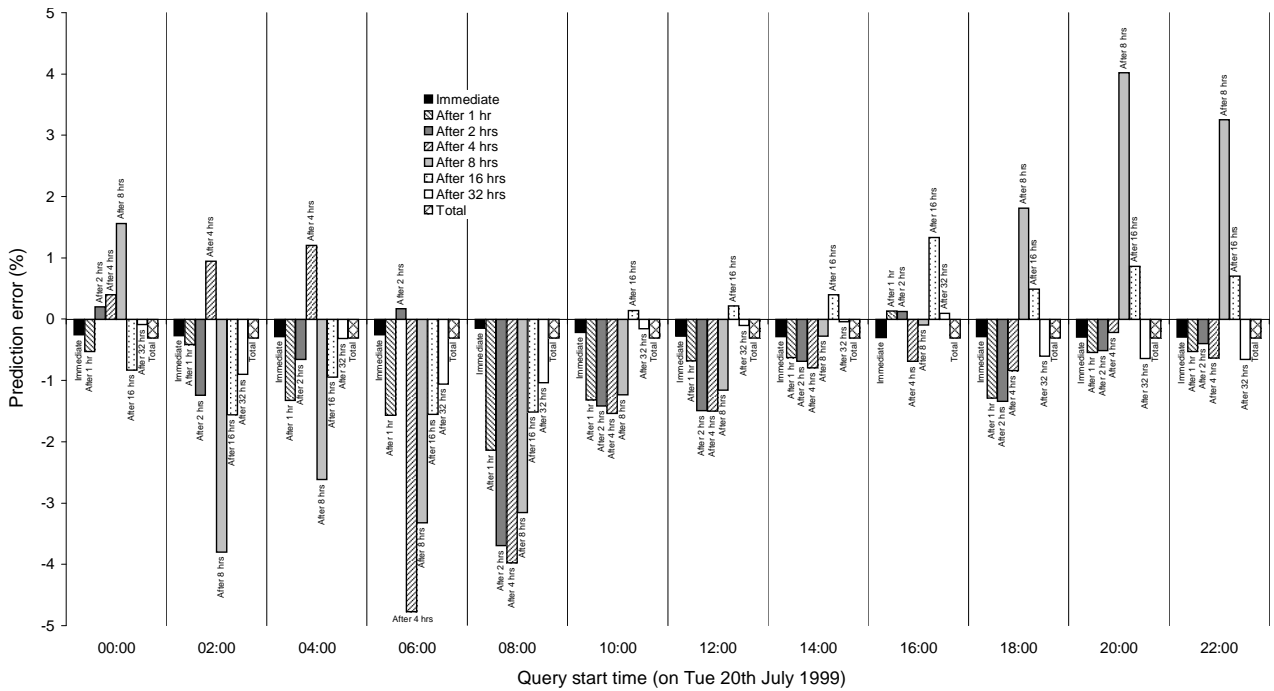### 4.3.2 Completeness prediction

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. Since packet-level network simulation is expensive to run on a large data set, and we wished to experiment with multiple queries as well as multiple query
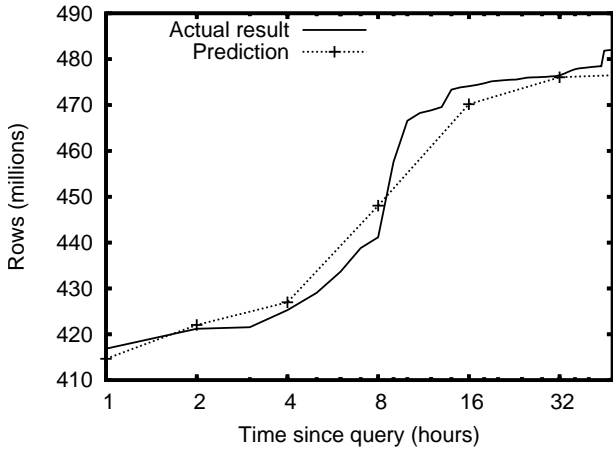
(a) Predicted versus actual completeness
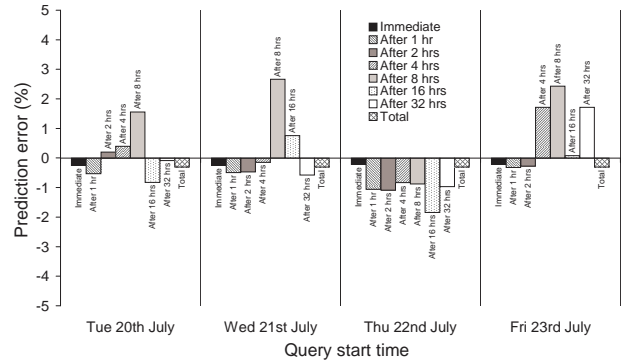


(b) Prediction error on different days of the week



(c) Prediction error for different injection times

Fig. 5: `SELECT SUM(Bytes) FROM Flow WHERE SrcPort=80`

start times, these experiments used a simplified simulator that correctly captures the effect of availability on completeness but does not do packet-level simulation.

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 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(a) 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 the query was:
`SELECT SUM(Bytes) 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: we see that it matches the observed result well. Note that when the query is first injected only 81% 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 5(b) shows the relative error of prediction for this query at different points during the query lifetime:
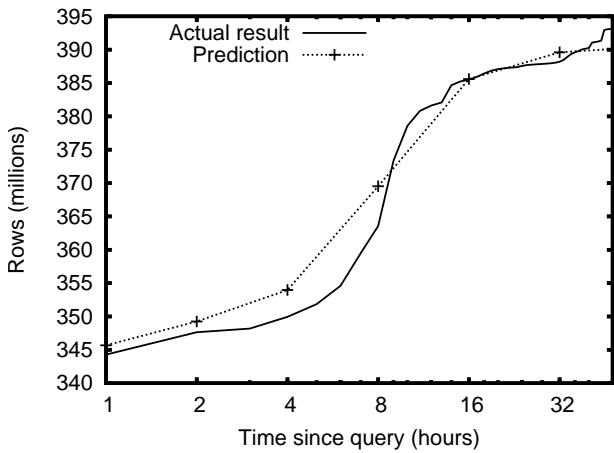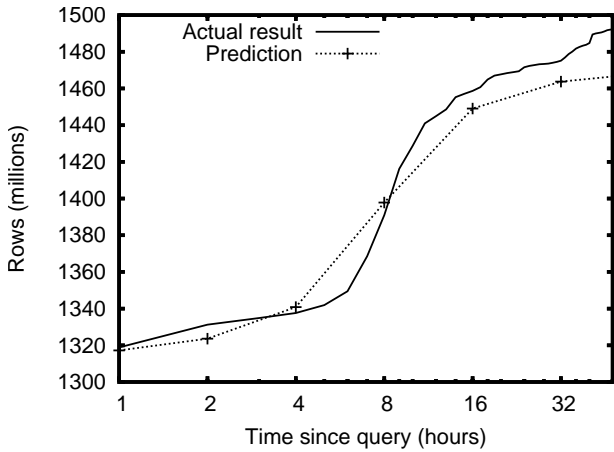
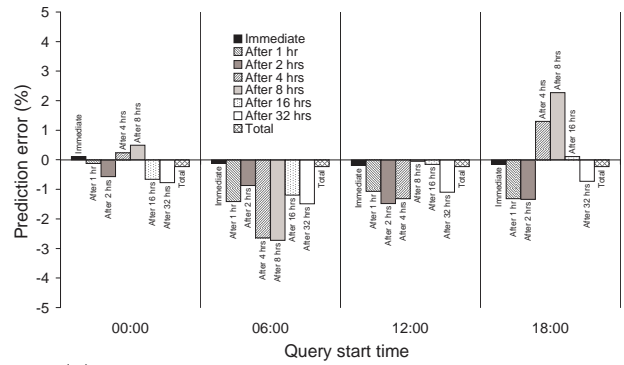(a) Predicted versus actual completeness
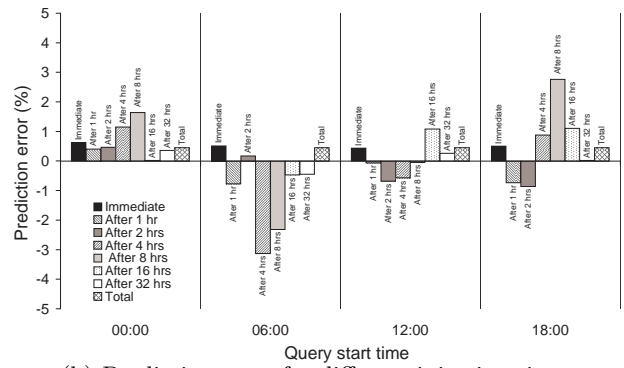
(b) Prediction error for different injection times

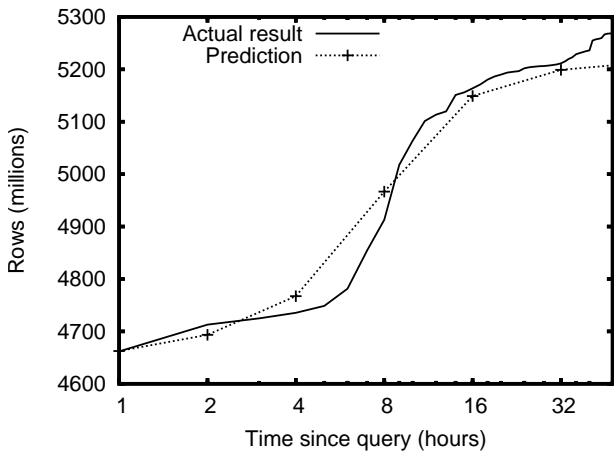Fig. 6: `SELECT COUNT(*) FROM Flow WHERE Bytes > 20000`



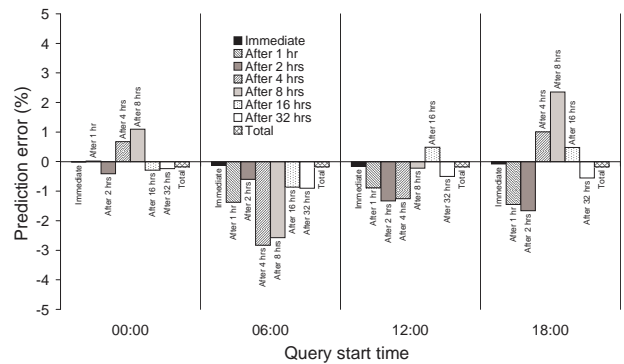(a) Predicted versus actual completeness

(b) Prediction error for different injection times

Fig. 7: `SELECT AVG(Bytes) FROM Flow WHERE App='SMB'`



(a) Predicted versus actual completeness

(b) Prediction error for different injection times

Fig. 8: `SELECT SUM(Packets) FROM Flow WHERE LocalPort < 1024`

immediately after the query is injected, after 1 hour, after 2 hours etc., as well as the prediction error on the total row count for this query. To test the effects of any weekly patterns, we show the errors measured on four consecutive weekdays. We also tested the effect of diurnal patterns, by varying the injection time by 2-hour intervals from midnight on Tuesday 20th July to the following midnight: the resulting errors are shown in Figure 5(c). The prediction error is consistently low, less than 5% in all cases.

Figures 6–8 show similar results for three other queries which compute respectively the number of flows with significant amounts of traffic, the average per-host SMB traffic, and the number of packets with privileged port numbers. The left-hand graphs show predicted versus actual rows for the query injected on Tuesday 20th July 1999 at 00:00, and the right-hand graphs show the prediction error for query injection times of 00:00, 06:00, 12:00 and 18:00. Again, the prediction error is under 5% in all cases.

The main source of error for these queries is in the availability prediction. In other words, we accurately predict the number of relevant rows on each endsystem: the prediction error for total row count is under 0.5% in all cases. However, we cannot exactly predict the time at which endsystems will next become available, due to variation in diurnal patterns, unpredictable failures, etc.

Row count estimation based on histograms is extremely accurate for queries such as these with range predicates on a single indexed column. We are currently exploring summarization techniques that will enable accurate estimation for more sophisticated queries.

### 4.3.3 Performance overheads

The second set of experiments measures the bandwidth overhead of running Seaweed at different network sizes, using the packet-level simulator. For each run we simulated from the 6th July 1999 to the 9th August 1999. We injected the query
`SELECT SUM(Bytes) FROM Flow WHERE SrcPort=80`
on Tuesday 20th July 1999 at 00:00. We allowed the query to run until the end of the simulation.

Figure 9(a) shows the overhead in bytes per second per online endsystem when running with 20,000 endsystems. On average there are 16,080 endsystems online. The overhead is sub-divided into the MSPastry overhead, the Seaweed maintenance overhead and the query overhead. The mean overhead of all three components put together is 69 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 low.

Figure 9(b) shows the cumulative distribution of load across endsystems and time, aggregated by 1-hour intervals for 20,000 endsystems. Each sample in this distribution is the average bandwidth used by a single endsystem in a single hour of the trace period. A transmission bandwidth of zero bytes in some hour indicates that the endsystem was unavailable in that hour: hence the $y$-intercept of this line is the mean unavailability. The 99th percentile of this distribution is only 178 bytes per endsystem per second, and the mean is 69 bytes per online endsystem per second. The distribution of receive bandwidth usage is similar, with a 99th percentile of 195 bytes per endsystem per second and a mean of 69 bytes per online endsystem per second. This shows that the overhead is not only low overall but also evenly distributed across endsystems and across time.

Since our query protocols are based on the endsystemIds in the Pastry overlay, we verified that the results were not sensitive to different assignments of these. We repeated the experiment with 8,000 endsystems and five different random assignments of endsystemIds. Figure 9(c) shows the resulting cumulative distribution of transmission bandwidth. The five curves are visually indistinguishable, with the maximum horizontal difference between any two curves at any point being $1 \times 10^{-6}$ bytes per endsystem per second.

Figure 9(d) shows the overhead in transmitted bytes per second per endsystem as the number of endsystems in the network ($N$) is varied between 2,000 and 51,663. The Seaweed maintenance overhead per endsystem, which dominates, is $O(1)$. The Seaweed query and MSPastry overhead both grow as $O(\log N)$. However, the MSPastry overhead is an order of magnitude lower than the Seaweed maintenance overhead, and the query overhead is three orders of magnitude lower. This leads us to believe that the design will scale to 1,000,000+ endsystems.

We also evaluated the latency between query injection and returning the completeness predictions to the user. With 2,000 endsystems the latency was 3.1 seconds, rising to 12.0 seconds for 51,663 endsystems. We feel that this is an acceptable latency for queries whose actual execution could take minutes or hours due to endsystem unavailability. The network bandwidth consumed for query dissemination with 20,000 endsystems was 1,043 bytes per query per endsystem and that for completeness predictor aggregation was 776 bytes per query per endsystem; these could be reduced further through packet format and protocol optimizations. For the usage scenarios we are targeting, with new queries submitted infrequently by a small number of human users, we feel that the cost of disseminating queries to all endsystems is justified by the resulting simplicity and high coverage.

Most Seaweed applications such as endsystem-based network management are targeted at fairly stable enterprise networks with low churn rates. This is captured by the Farsite availability traces with a mean endsystem departure rate of $4.06 \times 10^{-6}$ departures per online
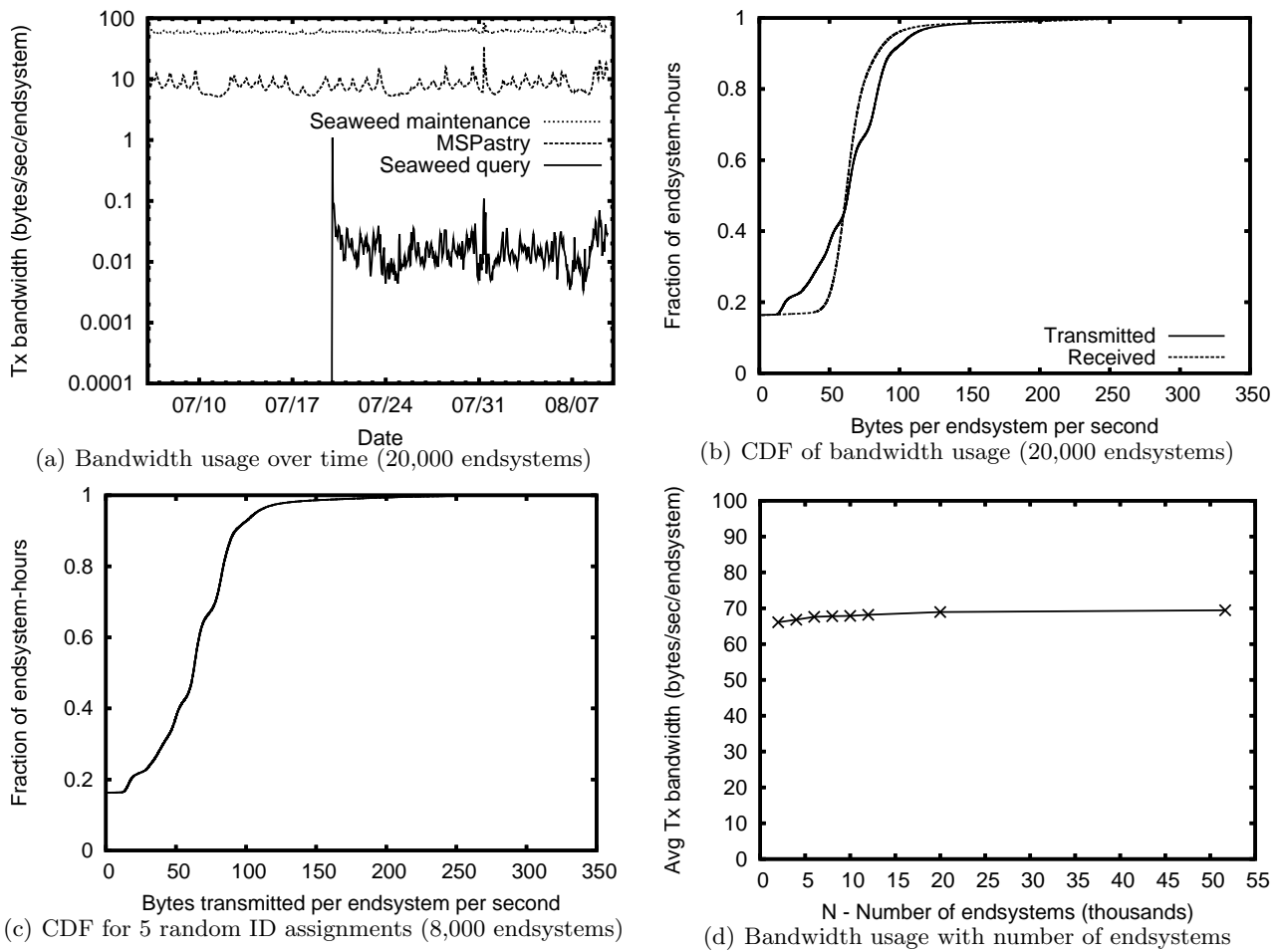
(a) Bandwidth usage over time (20,000 endsystems)

(b) CDF of bandwidth usage (20,000 endsystems)

(c) CDF for 5 random ID assignments (8,000 endsystems)

(d) Bandwidth usage with number of endsystems

Fig. 9: Seaweed overhead

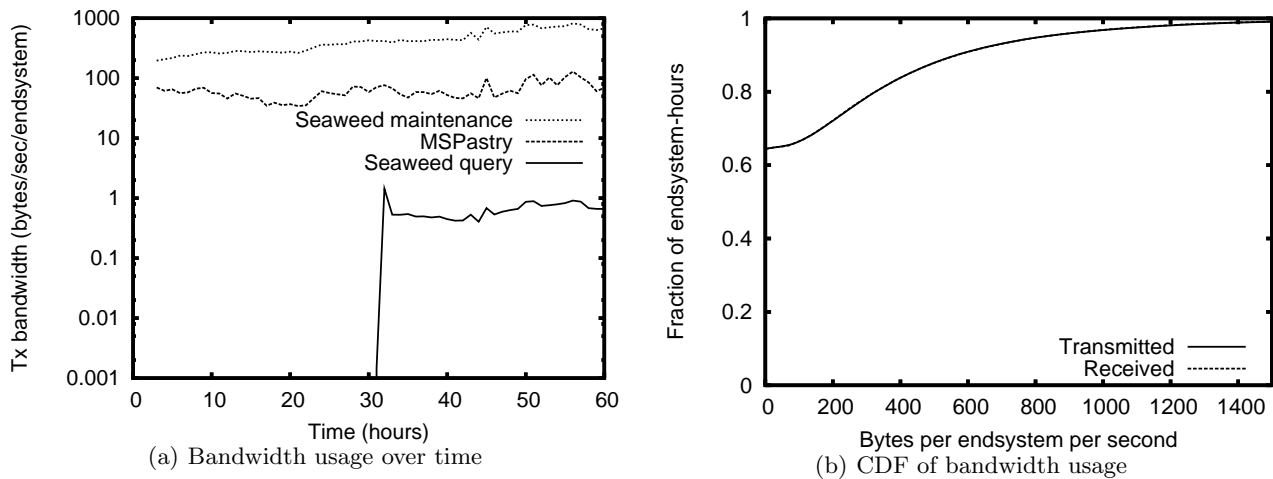(a) Bandwidth usage over time

(b) CDF of bandwidth usage

Fig. 10: Seaweed overhead in a high churn network

endsystem per second. To measure Seaweed's overhead under high churn, we repeated the experiment using the Gnutella activity traces [29]. We used a 60-hour trace with 7,602 endsystems and an average departure rate of $9.46 \times 10^{-5}$ departures per online endsystem per second. Figure 10(a) shows the total overhead over time in this case, and Figure 10(b) shows the cumulative distribution across endsystems and time. The mean transmission bandwidth used was 472 bytes per second per online endsystem and the 99th percentile was 1,515 bytes per second, and the distribution of receive bandwidth was almost identical. Thus the mean overhead increased only by a factor of 7 even though the departure rate per online endsystem increased by a factor of 23.

## 4.4 Deployment

We currently have Seaweed and an Anemone client deployed on approximately 30 machines in our office building. We are able to successfully inject queries and generate both completeness predictors and results for them. The stand-alone version uses the same codebase as the simulation results, but unfortunately, the scale of the current deployment means that it exists as a proof-of-concept rather than as a platform for running detailed experiments. We hope to deploy Seaweed on a significantly larger set of machines distributed across multiple sites in the near future.

## 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 [19–21] 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.

Seaweed replicates neither indexes nor data, aiming for far greater scalability by only replicating compact data summaries. Instead, it disseminates queries to all endsystems. For applications with sufficiently high query rates, distributed index structures may prove useful. However, a scalable design will still require that the data remain on the producing endsystems.

**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. Many large-scale systems route tuples through long-standing pre-installed queries [2,3,10, 13,23,30]. Borealis [3] deals with data unavailability on much smaller time scales 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] and histogram-based approximation [17] are examples of more sophisticated techniques that might support summary-based estimation for a wider range of queries.

**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 query latency for scalability.

Analysis and simulation show that Seaweed scales well and that metadata replication enables the gener-

ation 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.

## References

1. Aberer, K., Datta, A., Hauswirth, M., Schmidt, R.: Indexing data-oriented overlay networks. In: VLDB, pp. 685–696. Trondheim, Norway (2005)
2. Avnur, R., Hellerstein, J.M.: Eddies: Continuously adaptive query processing. In: SIGMOD, pp. 261–272. Dallas, TX (2000)
3. Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-tolerance in the Borealis distributed stream processing system. In: SIGMOD, pp. 13–24. Baltimore, MD (2005)
4. Bawa, M., Gionis, A., Garcia-Molina, H., Motwani, R.: The price of validity in dynamic networks. In: SIGMOD, pp. 515–526. Paris, France (2004)
5. Bhagwan, R., Savage, S., Voelker, G.M.: Understanding availability. In: IPTPS, pp. 256–267 (2003)
6. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: SIGCOMM, pp. 353–366. Portland, OR (2004)
7. Blake, C., Rodrigues, R.: High availability, scalable storage, dynamic peer networks: Pick two. In: HotOS-IX, pp. 1–6. Kauai, HA (2003)
8. Bolosky, W., Douceur, J., Ely, D., Theimer, M.: Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In: SIGMETRICS, pp. 34–43. Santa Clara, CA (2000)
9. Castro, M., Costa, M., Rowstron, A.: Performance and dependability of structured peer-to-peer overlays. In: DSN, pp. 9–18. Florence, Italy (2004)
10. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A scalable continuous query system for Internet databases. In: SIGMOD, pp. 379–390. Dallas, TX (2000)
11. Cheng, R., Xia, Y., Prabhakar, S., Shah, R., Vitter, J.S.: Efficient indexing methods for probabilistic threshold queries over uncertain data. In: VLDB, pp. 876–887. Toronto, CN (2004)
12. Dabek, F., Zhao, B.Y., Druschel, P., Kubiatowicz, J., Stoica, I.: Towards a common API for structured peer-to-peer overlays. In: IPTPS, pp. 33–44 (2003)
13. Deshpande, A., Hellerstein, J.M.: Lifting the burden of history from adaptive query processing. In: VLDB, pp. 948–959. Toronto, CN (2004)
14. Halevy, A.Y., Ashish, N., Bitton, D., Carey, M.J., Draper, D., Pollock, J., Rosenthal, A., Sikka, V.: Enterprise information integration: successes, challenges and controversies. In: SIGMOD, pp. 778–787. Baltimore, MD (2005)
15. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. In: SIGMOD, pp. 171–182. Tucson, AZ (1997). DOI http://doi.acm.org/10.1145/253260.253291
16. Huebsch, R., Hellerstein, J.M., Lanham, N., Loo, B.T., Shenker, S., Stoica, I.: Querying the Internet with PIER. In: VLDB, pp. 321–332. Berlin, Germany (2003)
17. Ioannidis, Y.E., Poosala, V.: Histogram-based approximation of set-valued query-answers. In: VLDB, pp. 174–185. Edinburgh, UK (1999)
18. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: BATON: A balanced tree structure for peer-to-peer networks. In: VLDB, pp. 661–672. Trondheim, Norway (2005)
19. Johnson, T., Krishna, P.: Lazy updates for distributed search structure. In: SIGMOD, pp. 337–346. Washington DC, USA (1993). DOI http://doi.acm.org/10.1145/170035.170085
20. Litwin, W., Neimat, M.A., Schneider, D.A.: RP*: A family of order preserving scalable distributed data structures. In: VLDB, pp. 342–353. Santiago de Chile, Chile (1994)
21. Lomet, D.B.: Replicated indexes for distributed data. In: PDIS, pp. 108–119. Miami Beach, FL (1996)
22. Loo, B.T., Hellerstein, J.M., Huebsch, R., Shenker, S., Stoica, I.: Enhancing P2P file-sharing with an Internet-scale query processor. In: VLDB, pp. 432–443. Toronto, CN (2004). URL http://www.vldb.org/conf/2004/RS11P2.PDF
23. Madden, S., Shah, M.A., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: SIGMOD, pp. 49–60. ACM, Madison, WI (2002)
24. Mickens, J.W., Noble, B.D.: Exploiting availability prediction in distributed systems. In: NSDI, pp. 73–86. San Jose, CA (2006)
25. Microsoft: Dr. Watson for Windows. http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/drwatson_overview.mspx (2006)
26. Mortier, R., Isaacs, R., Barham, P.: Anemone: using end-systems as a rich network management platform. In: SIGCOMM MineNet, pp. 203–204. Philadelphia, PA (2005). DOI http://doi.acm.org/10.1145/1080173.1080184
27. Mortier, R., Narayanan, D., Donnelly, A., Rowstron, A.: Seaweed: Distributed scalable ad-hoc querying. In: NetDB Workshop. Atlanta, GA (2006)
28. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Middleware, pp. 329–350 (2001)
29. Saroiu, S., Gummadi, K., Gribble, S.: A measurement study of peer-to-peer file sharing systems. In: MMCN. San Jose, CA (2002)
30. Tian, F., DeWitt, D.J.: Tuple routing strategies for distributed eddies. In: VLDB, pp. 333–344. Berlin, Germany (2003)
31. Van Renesse, R., Birman, K., Vogels, W.: 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). DOI http://doi.acm.org/10.1145/762483.762485
32. Yalagandula, P., Dahlin, M.: A scalable distributed information management system. In: SIGCOMM, pp. 379 – 390. Portland, OR (2004)