Moara: Flexible and Scalable Group-Based Querying System

Steven Y. Ko[1], Praveen Yalagandula[2], Indranil Gupta[1],
Vanish Talwar[2], Dejan Milojicic[2], Subu Iyer[2]

[1]University of Illinois at Urbana-Champaign [2]HP Labs, Palo Alto

Abstract. Users and administrators of large-scale infrastructures (e.g., datacenters and PlanetLab) are frequently in need of monitoring *groups* of machines in the infrastructure. Though there exist several distributed querying systems for this monitoring purpose, they are not group-based; they mostly focus on querying the entire system. In this paper, we present *Moara*, a new querying system that makes two novel contributions. First, Moara builds aggregation trees for different groups and adaptively maintains the trees to optimize the total message cost. Second, Moara supports a query language allowing groups to be specified implicitly via predicates consisting of arbitrarily nested unions and intersections. Our evaluations on Emulab, on PlanetLab, and with large-scale simulations, demonstrate Moara's ability to answer complex queries within a fraction of a second, to deal with high levels of dynamism in groups, and to incur a low bandwidth overhead per host per query in comparison to existing centralized and distributed aggregation systems.

1 Introduction

Large-scale distributed infrastructures have become increasingly common in various domains. Today's enterprise data centers [1] are equipped with thousands of machines and run thousands of different applications and services. Federated computing infrastructures such as PlanetLab [2], proposed GENI infrastructure [3], and computational grids [4] consist of thousands of hosts providing resources for a number of projects.

A frequent need of the users and the administrators of such infrastructures is monitoring and querying the status of *groups* of machines in the infrastructure, as well as the infrastructure as a whole. These groups may be static or dynamic, e.g., the PlanetLab slices, the machines running a particular service in a datacenter, or the machines with CPU utilization above 50%. Further, users typically desire to express complex criteria for the selection of the host groups to be queried. For example, "find top-3 loaded hosts where (ServiceX = true) and (Apache = true)" is a query that targets two groups - hosts that run service X and hosts that run Apache. Dynamic groups mean that the size and composition of groups vary across different queries as well as time.

In general, users and administrators desire to monitor the performance of these groups, to troubleshoot any failures or performance degradations, and to track usage of allocated resources. These requirements point to the need for a *group-based querying system* that can provide instantaneous answers to queries over in-situ data targeting one or more groups. In fact, several existing distributed aggregation systems [5–7] can be considered as a special case of group-based querying systems, as they target querying of only a single group, *i.e.,* the entire system.

Any group-based querying system should satisfy three requirements: *flexibility*, *efficiency*, and *scalability*. First, the system should be flexible to support expressive queries

that deal with multiple groups, such as unions and intersections of different groups. Second, the system should be efficient in query resolution—it should minimize the message overhead while responding quickly with an answer. Third, the system should scale with the number of machines, the number of groups, and the rate of queries.

In this paper, we propose Moara, a new group-based distributed aggregation system that targets all three requirements. A query in Moara has three parts: (*query-attribute*, *aggregation function*, *group-predicate*), e.g., (Mem-Util, Average, Apache $=$ true). Moara returns the resulting value from applying the *aggregation function* over the values of *query-attribute* at the machines that satisfy the *group-predicate*.

Moara makes two novel design contributions over existing systems [5–7]. First, Moara maintains aggregation trees for different groups adaptively based on the underlying environment and the injected queries to minimize the overall message cost and query response time. Basically, the aggregation tree for a group in Moara is an optimized sub-graph of a global spanning tree, which spans all nodes in the group. By aggregating data over these group-based aggregation trees, Moara achieves lower message cost and response latency for queries compared to other aggregation systems that contact all nodes. Further, we adapt each aggregation tree to deal with dynamism.

Second, Moara's query processor supports *composite* queries that target multiple groups simultaneously. Composite queries supported by Moara are arbitrary nested set expressions built by using logical operators `or` and `and`, (respectively set operations $\cup$ and $\cap$) over simple group-predicates. Simple group-predicates are of the form (*attribute op value*), where *op* $\in \{<, >, \leq, \geq, =, \neq\}$. Consider our previous example "find top-3 loaded hosts where (ServiceX $=$ true) and (Apache $=$ true)", which is a composite query that targets the intersection of two groups - hosts that run service X and hosts that run Apache. Instead of blindly querying all the groups present in a query, Moara's query processor analyzes composite queries and intelligently decides on contacting a set of groups that minimizes the communication overhead.

We implemented a prototype of Moara by leveraging the FreePastry DHT (Distributed Hash Table) [8] and SDIMS [7] systems. Our evaluation consists of experiments on Emulab [9] and PlanetLab, as well as large-scale simulations. Our experimental results indicate that, compared to previous global hierarchical aggregation systems, Moara reduces response latency by up to a factor of 4 and achieves an order of magnitude bandwidth savings. Our scalability experiments confirm that Moara's overhead for answering a query is independent of the total number of nodes in the system, and only grows linearly with the group size. Finally, we show that Moara can answer complex queries within hundreds of milliseconds in systems with hundreds of nodes under high group churn.

In this work, we focus on efficiently supporting one-shot queries (as opposed to repeated continuous queries) over a common set of groups, since we expect this type of queries to be more common in the kind of infrastructures we are targeting at — datacenters and federated computing systems. We expect most users will be performing one-shot queries over common groups (e.g., the same PlanetLab slice, machines in a datacenter, etc) during the time when their service or experiment is running. Further, a user interested in monitoring groups continually can invoke one-shot queries periodically. Our use cases in Section 2 motivates this design decision further.

Any distributed system subjected to dynamism in the environment, suffers from the CAP dilemma [10], which states that it is difficult to provide both strong consistency guarantees and high availability in failure-prone distributed settings. Moara treads this dilemma by preferring to provide high availability and scalability, while providing eventual consistency guarantees on aggregation results. This philosophy is in line with that of existing aggregation systems such as Astrolabe [6] and SDIMS [7]. Moara could also allow the use of metrics proposed by Jain et al. [11, 12] in order to track the imprecision of the query results; however, studying these is beyond the scope of the current paper.

2 Motivation and Use Cases

We highlight the need for on-demand flexible querying and for dealing with dynamism by presenting two motivating scenarios - data centers and federated infrastructures.

Consolidated Data Centers: In the last few years, medium and large-scale enterprises have moved away from maintaining their own clusters, towards subscribing to services offered by consolidated data centers. Such consolidated data centers consist of multiple locations, with each location containing several thousands of servers [1]. Each server runs heterogeneous operating systems including virtual machine hosts. While such consolidation enables running unified management tasks, it also introduces the need to deal with scale.

Workloads on these data centers typically include Terminal Services, SOA-based transaction workloads (e.g., SAP), and Web 2.0 workloads, e.g., searching and collaboration. Figure 1 presents some on-demand one-shot queries that data center managers and service owners typically desire to run on such a virtualized enterprise. Several of these one-shot queries are for aggregating information from a common group of nodes including cases where groups are expressed as unions of groups (e.g., the third query in table), or intersections (e.g., the last query). We would like to generalize this to provide managers with a powerful tool supporting flexible queries using arbitrarily nested unions and intersections of groups. In addition, these workloads vary in intensity over time, causing considerable dynamism in the system, e.g., terminal services facing high user turnaround rates.

Federated Computing Infrastructures: In today's federated computing infrastructures such as PlanetLab [2] and global Grids [4], as well as in proposed infrastructures, e.g., GENI [3], users wish to query current statistics for their distributed applications or experiments. For instance, PlanetLab creates virtual subgroups of nodes called "slices" in order to run individual distributed applications. Monitoring is currently supported by tools such as CoMon [13] and Ganglia [14], which periodically collect CPU, memory, and network data per slice on PlanetLab [2]. Due to their periodic nature, they are not open to on-demand queries that require up-to-date answers. Further, increasing the frequency of data collection is untenable due to storage and communication costs.

In contrast to the above systems, we need a system to answer one-shot queries that seek to obtain up-to-date information over a common group of machines, that can be run on-demand or periodically by an end-host, and are flexibly specified. Some examples of our target queries include: number of slices containing at least one machine with CPU utilization $> 90\%$ (basic query), CPU utilization of nodes common to two given slices (intersection query), or free disk space across all slices in a given organization (union query).

Tasks	Queries
Resource Allocation	Average utilization for servers belonging to (i) floor F, (ii) cluster C, (iii) rack R
	Number of machines/VMs in a given cluster C
VM Migration	Average utilization of VMs running application X version 1 or version 2
	List of all VMs running application X and are VMWare based
Auditing/Security	Count of all VMs/machines running firewall
	Count of all VMs running ESX server and Sygate firewall
Dashboard	Max response time for Service X
	Count of all machines that are up and running Service X
Patch management	List of version numbers being used for service X
	Count of all machines that are in cluster C and running service X.version Y

Fig. 1: Illustrative Queries for Managing the Virtualized Enterprise

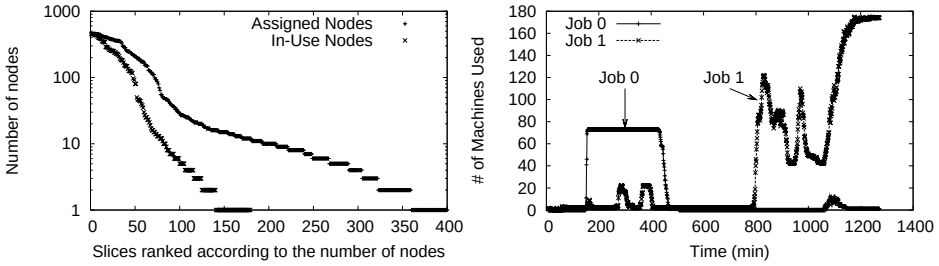

Fig. 2: (a) Usage of PlanetLab nodes by different slices. We show both node assignment to slices and active usage of nodes. Data collected from a CoTop snapshot [15]. (b) Usage of HP's utility computing environment by different animation rendering jobs. We show the number of machines each job uses.

Need for Group-based Aggregation: As illustrated by above two target scenarios, we expect that most of the queries are one-shot queries over common groups of machines. Moreover, the predicate in a query specified as a logical expression involves several groups, e.g., some groups in the above examples include the set of nodes in a PlanetLab slice, the set of nodes running a given Grid task, the set of nodes with CPU utilization $> 90\%$, etc. In the worst case, such a group may span the entire system.

In practice though, we expect the group sizes to vary across different queries and with time. In Figure 2(a), we plot the distribution of PlanetLab slice sizes, analyzed from an instance of CoMon [13] data. Notice that there is a considerable spread in the sizes. As many as 50% of the 400 slices have fewer than 10 assigned nodes, thus a monitoring system that contacts all nodes to answer a query for a slice is very inefficient. If we consider only nodes that were actually in use (where a slice has more than one process running on a node), as many as 100 out of 170 slices have fewer than 10 active nodes. In another example case, Figure 2(b) presents the behavior of two jobs over a 20-hour period from a real 6-month trace of a utility computing environment at HP with 500 machines receiving animation rendering batch jobs. This plot shows the dynamism in each group over time.

These trace studies indicate that group sizes can be expected to be varying across time in both consolidated centers as well as in federated computing infrastructures. Thus, an efficient querying system has to avoid treating the entire system as a single group and globally broadcasting queries to all nodes.

3 The Basics of Moara

In this section, we first discuss how Moara end-nodes maintain data and how queries are structured. Then we discuss how Moara builds trees for individual groups.

3.1 Data and Query Model

Information at each node is represented and stored as (*attribute, value*) tuples. For example, a machine with CPU capacity of 3Ghz can have an attribute (CPU-Mhz, 3000). Moara has an agent running at each node that monitors the node and populates (*attribute, value*) pairs.

A query in Moara comprises of three parts: (*query-attribute*, *aggregation function*, *group-predicate*). The first field specifies the attribute of interest to be aggregated, while the second field specifies the aggregation function to be used on this data. We require this aggregation function to be partially aggregatable. In other words, given two partial aggregates for multiple disjoint sets of nodes, the aggregation function must produce an aggregate that corresponds to the union of these node sets [6, 7]. This admits aggregation functions such as enumeration, max, min,sum, count, or top-k. Average can be implemented by aggregating both sum and count.

The third field of the query specifies the group of machines on which the above aggregation is performed. If no group is specified, the default is to aggregate values from all nodes in the system. A *group-predicate* (henceforth called a "predicate") is specified as a boolean expression with `and` and `or` operators, over *simple* predicates of the following form: (*group-attribute op value*), where *op* $\in \{<, >, =, \leq, \geq, \neq\}$. Note that this set of operators allows us to implicitly support `not` in a group predicate. Any *attribute* that a Moara agent populates can be used as either *query-attribute* or *group-attribute*.

A simple query contains a simple predicate. For example, the simple predicate (ServiceX = true) defines all machines running ServiceX. Thus, a user wishing to compute the maximum CPU usage across machines where ServiceX is running will issue the following query: (CPU-Usage, MAX, (ServiceX = true)). Alternately, the user could use a *composite* predicate, e.g., (ServiceX = true `and` Apache = true). This composite query is defined with set operators $\cup$ and $\cap$.

Note that the query model can be easily extended so that instead of a *query-attribute*, a querier can specify any arbitrary program that operates upon simple *(attribute, value)* pairs. For example, a querier can specify a program that evaluates (CPU-Available $>$ CPU-Needed-For-App-A) as *query-attribute*, to see how many nodes are available for the application A. Similarly, *group-predicate* can be extended to contain multiple attributes by defining new attributes. For example, we can define a new attribute *att* as (CPU-Available $>$ CPU-Needed-For-App-A), which takes a boolean value of true/false. Then *att* can be used to specify a group. However, for this paper, we mainly focus on the techniques for efficiently answering the queries for given *group-predicates* and hence restrict query model to contain only simple attributes.

3.2 Scalable Aggregation

We describe here how Moara aggregates data for each group.

DHT trees: For scalability with large number of nodes, groups, and queries, Moara employs a peer-to-peer in-network aggregation approach that leverages the computing

and network resources of the distributed infrastructure itself to compute results. These trees are used for spreading queries, and aggregating answers back towards the source node. In our architecture, a lightweight Moara agent runs at each server from which data needs to be aggregated. These agents participate in a structured overlay routing algorithm such as Pastry [8], Tapestry [16], or Chord [17]. These systems allow routing within the overlay, from any node to any other node, based on the IDs of these nodes in the system. Moara uses this mechanism for building aggregation trees called DHT trees, akin to existing systems [7, 18, 19]. A DHT tree contains all the nodes in the system, and is rooted at a node that maps to the ID of the group. For instance, Figure 3 shows the tree for an ID with prefix 000 using Pastry's algorithm with one-bit prefix correction. We choose to leverage a DHT, since it handles physical membership churn (such as failures and join/leave) very modularly and efficiently. Also, we can construct aggregation trees clearly, given a group predicate.

Basics of Resolving Queries: Given a simple query with predicate p, Moara uses MD-5 to hash the *group-attribute* field in p and derives a bit-string that stands for the group ID. The DHT tree for this ID is then used to perform aggregation for this query, e.g., Figure 3 shows the DHT tree for an attribute "ServiceX" that hashes to 000.

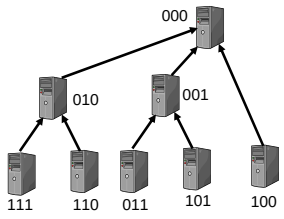

Fig. 3: DHT tree for an ID with prefix 000

When a simple query is generated at any node in Moara, it is first forwarded to the root node of the corresponding DHT tree via the underlying DHT routing mechanism. The root then propagates it downwards along the DHT tree to the leaves. When a leaf receives a query, it evaluates the predicate p in the query (e.g., ServiceX=true). If the result is true, it replies to its parent the local value for the query attribute (e.g., CPU-Usage). Otherwise, it sends a null reply to its parent. An internal node waits to reply to its parent until *all* its children have replied or until a timeout occurs (using values in Section 7). Then, it aggregates the values reported by its children, including its own contribution if the predicate is satisfied locally, and forwards the aggregate to its parent. Finally, the root node replies to the original querying node with the aggregated value.

Moara Mechanisms: The above "global aggregation" approach has every node in the system receive every query. Hence, it is inefficient in resolving queries targeting specific groups. Moara addresses this via three mechanisms.

First, Moara attempts to prune out branches of the tree that do not contain any node satisfying the predicate p. We call this tree a pruned tree or a *group tree* for p. For example, in Figure 3, if nodes 111, 110, and 010 do not satisfy the predicate, then the root does not forward the query to 010. However, this raises a challenge – how do internal nodes know whether any of their descendants satisfy the predicate. For instance, if node 110 decides to install ServiceX and thus satisfies the predicate, the path from the root to this node will need to be added to the tree. Further, if the composition of a group changes rapidly, then the cost for maintaining the group tree can become higher than query resolution costs. Section 4 presents Moara's dynamic adaptation mechanism that addresses this dilemma.

Second, Moara reduces network cost and response latency by short-circuiting the group trees, thus reducing the number of internal tree nodes that do not satisfy the predicate. For instance, in Figure 3, if node 010 does not satisfy the predicate but node 110 does, then the former can be eliminated from the tree by having 110 receive queries directly from the root. Section 5 describes how this reduces the bandwidth cost of aggregating a group with m nodes in a system of N nodes, from $O(m \log N)$ to $O(m)$.

Third, Moara efficiently resolves composite queries involving multiple groups by rewriting the predicate into a more manageable form, and then selecting a minimal set of groups to resolve the query. For example, an intersection query (CPU-Util, avg, (floor=F1 `and` cluster=C12)) is best resolved by sending the query to only one of the two groups - either (floor=F1) or (cluster=C12) - whichever is cheaper. This design decision of Moara is detailed in Section 6.

4 Dynamic Maintenance

Given a tree for a specific group, Moara reduces bandwidth cost by adaptively pruning out parts of the tree, while still guaranteeing correctness via *eventual completeness*. Eventual completeness is defined as follows - when the set of predicate-satisfying nodes as well as the underlying DHT overlay do not change for a sufficiently long time after a query injection, a query to the group will eventually return answers from all such nodes. For now, we assume that the dynamism in the system is only due to changes in the composition of the groups ("group churn"); we will describe how our system handles node and network reconfigurations (churn in system) later in Section 7.

To resolve queries efficiently, Moara could prune out the branches of the corresponding DHT tree that do not contain any nodes belonging to the group. However, to maintain completeness of the query resolution, Moara can perform such aggressive pruning only if it maintains up-to-date information at each node about the status of branches at that node. For groups with high churn in membership relative to the number of queries (e.g., CPU-Util < 50), maintaining group status at each node for all its branches can consume high bandwidth - broadcasting queries system-wide may be cheaper. For relatively stable groups however (e.g., (sliceX $=$ true) on PlanetLab), proactively maintaining the group trees can reduce bandwidth and response times. Instead of implementing either of these two extreme solution points, Moara uses a distributed adaptation mechanism that, at each node, tracks the queries in the system and group churn events from children for a group predicate and decides whether or not to spend any bandwidth to inform its parent about its status.

Basic Pruning Mechanism: Each Moara node maintains a binary local state variable $prune$ for each group predicate. If $prune$ for a predicate is true (PRUNE state), then the branch rooted at this node can be pruned from the DHT tree while querying for that predicate. Whenever a node goes from PRUNE to NO-PRUNE state, it sends a NO-PRUNE message to its parent; the reverse transition causes a PRUNE message to be sent. When the root or an internal node receives a query for this predicate, it will forward the query to only those of its children that are in NO-PRUNE state.

Note that it is incorrect for an internal node to set its state for a predicate to PRUNE based merely on whether it satisfies the predicate or not. One or more its descendants may satisfy the predicate, and hence the branch rooted at the node should continue to

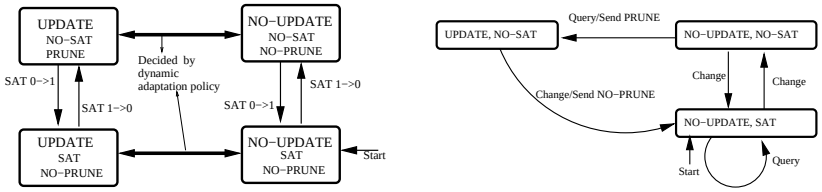

Fig. 4: (a) State machine for dynamic adaptation mechanism (b) State changes at a Moara node for $k_{UPDATE} = 1$ and $k_{NO-UPDATE} = 1$. With these values, (UPDATE, SAT) is not reachable, thus is not shown here.

receive any queries for this predicate. Further, an internal or a leaf node should also consider the churn in the predicate satisfiability before setting the $prune$ variable. For example, suppose the predicate is (CPU-Util < 50) and a leaf node's utilization is fluctuating around 50% at a high rate. In this case, the leaf node will be setting and unsetting $prune$ variable, leading to a large number of PRUNE/NO-PRUNE messages.

Due to the above reasons, we define the $prune$ variable as a variable depending on two additional local state variables—sat and $update$. sat is a binary variable to track if the subtree rooted at this node should continue receiving queries for the predicate. Thus sat is set to 1 (SAT) if either the local node satisfies the predicate or any child node is in NO-PRUNE state.

$update$ is a binary state variable that denotes whether the node will update its $prune$ variable or not. So, when $update = 1$ (UPDATE state), the node will update the $prune$ variable; but, when $update = 0$ (NO-UPDATE state), the node will cease to perform any updates to the $prune$ variable irrespective of any changes in the local satisfiability, or any messages from its children. In other words, a node does not send any PRUNE or NO-PRUNE messages to its parent when it is in NO-UPDATE state. So, to ensure correct operation, a node can move into NO-UPDATE state only after setting $prune = 0$. This guarantees that its parent will always send the queries for the predicate to this node. Formally, we maintain the following invariants:

$$update = 1 \text{ AND } sat = 1 \implies prune = 0$$
$$update = 1 \text{ AND } sat = 0 \implies prune = 1$$
$$update = 0 \implies prune = 0$$

The transition rules for the state machine at each node is illustrated in Figure 4(a). Note that a node sends a status update message to its parent whenever it moves from PRUNE to NO-PRUNE state or vice-versa. This state machine ensures the following invariant – *each node in the system performs at least one of the following: (a) sends status updates upwards to its parent, or (b) receives all queries from its parent*. This invariant suffices to guarantee eventual completeness because after the group stops changing, any node that satisfies the predicate will be in SAT state. Therefore, the node and its ancestors will all be in NO-PRUNE state, and thus the node will receive the next query. Procedure 1, 2, and 3 show pseudo-code on how Moara evaluates each variable.

Adaptation Policy: To decide the transition rules for the *update* state variable, Moara employs an adaptation mechanism that allows different policies. Our goal is to use a policy that minimizes the overall message cost, *i.e.*, sum of both update and query costs. In Moara, each node tracks the total number of *recent* queries and local changes it has seen (in the tree) - we will explain recentness soon. Each node keeps two

Procedure 1 Updating *sat* variable for each predicate

Initial Value: *sat* $\leftarrow 0$
Procesure Call: whenever there is a local attribute change or an update from any child
  cnt $\leftarrow 0$
  for each child **do**
    if there is no state associated with this child regarding the given predicate **then**
      // by default, a parent does not maintain any state on its children
      // Also, states can be garbage-collected after a period of inactivity
      cnt++
    else if child is in NO-PRUNE state **then**
      cnt++
    end if
  end for
  if the predicate is locally satisfied **then**
    cnt++
  end if
  if *cnt* > 0 **then**
    sat $= 1$
  else
    sat $= 0$
  end if

Procedure 2 Updating *update* variable for each predicate

Initial Value: *update* $\leftarrow 0$ // in the beginning, a node receives every query
Procesure Call: whenever there is a new query received or a *sat* variable change
  if $2 \times q_n < c$ **then**
    update $\leftarrow 0$
  else if $2 \times q_n > c$ **then**
    update $\leftarrow 1$
  end if

query counts - q_n, the number of queries recently received by the system while the node is in NO-SAT state, and q_s, the number of recent queries received by the system while it was in SAT state. The node also keeps track of the number of times the sat variable toggled between 0 and 1, denoted as c.

A node in NO-UPDATE state would exchange a total of $B_{NU} = 2 \times (q_n + q_s)$ messages with its parent (two per query), while a node in UPDATE state would exchange $B_{UP} = c + 2 \times q_s$ messages (one per change, and two per query). Thus, to minimize bandwidth, the transition rules are as follows: (1) a node in UPDATE state moves to NO-UPDATE if $B_{NU} < B_{UP}$, *i.e.*, $2 \times q_n < c$; (2) a node in NO-UPDATE state moves to UPDATE if $B_{NU} > B_{UP}$, *i.e.*, $2 \times q_n > c$. In order to avoid flip-flopping around the threshold, we could add in hysteresis, but our current design performs well without it.

Remembering Recent Events: Each node in Moara maintains a recent "window" of events for the counters mentioned above (q_n, q_s, and c). We use a window of k_{UPDATE}

Procedure 3 Updating *prune* variable for each predicate

Procesure Call: whenever there is a change in either *update* or *sat*
  if *update* $==1$&& *sat* $==1$ **then**
    prune $\leftarrow 0$
  else if *update* $==1$&& *sat* $==0$ **then**
    prune $\leftarrow 1$
  else if *update* $==0$ **then**
    prune $\leftarrow 0$
  end if

events if the node is in UPDATE state, and a window of $k_{NO-UPDATE}$ events if it is NO-UPDATE. In practice, we found that $k_{UPDATE} = 1$, $k_{NO-UPDATE} = 3$ works well, and we use these values in our implementation. For illustration purposes though, Figure 4(b) depicts the state machine for $k_{UPDATE} = k_{NO-UPDATE} = 1$. In this case, notice that whenever a node: (i) is in the PRUNE state and observes a change in sat ($c = 1, q_n = 0$), it switches to (NO-UPDATE, SAT); (ii) is in (NO-UPDATE, NO-SAT) and receives a query ($q_n = 1, c = 0$), it switches to UPDATE.

One corner issue with the above approach is that when a node is in the PRUNE state, it does not receive any more queries and thus cannot accurately track q_n. Note that this does not affect the correctness (*i.e.*, eventual completeness) of our protocol but may cause unnecessary status update messages. To address this, the root node of an aggregation tree in Moara assigns a sequence number for each query and sends that number piggybacked along with the queries. Thus, any node that receives a query with sequence number s is able to track q_n using the difference between s and its own last-seen query sequence number. In addition, our implementation suffers only minimally since we use small k_{UPDATE} values. For instance, for $k_{UPDATE} = 1$, when a node in (UPDATE, SAT) undergoes a local change, it immediately switches to NO-UPDATE, and sends no more messages to its parent.

State Maintenance: By default, each node does not maintain any state, which is considered as being in NO-UPDATE state. A node starts maintaining states only when a query arrives at the node. Without dynamic maintenance, merely maintaining pruned trees for a large number of predicates (e.g., a tree for each slice in the PlanetLab case or a tree for each job in the data center) could consume very high bandwidth in an aggregation system. With dynamic maintenance, pruning is proactively performed for only those predicates that are of interest at that time. Once queries stop, nodes in the aggregation tree start moving into NO-UPDATE state with any new updates from their children and hence stop sending any further updates to their parents.

We note that a node in NO-UPDATE state for a predicate can safely garbage-collect state information (e.g., predicate itself, recent events information, etc) for that predicate without causing any incorrectness in the query resolution. So, once a predicate goes out of interest, eventually no state is maintained at any node and no messages are exchanged between nodes for that predicate. Several policies for deciding when to garbage-collect state information are possible: we could 1) garbage-collect each predicate after a time-out expires, 2) keep only the last k predicates queried, 3) garbage-collect the least fre-

quently queried predicate every time a new query arrives, etc. However, studying these policies is beyond the scope of this paper. We also note that we do not consider DHT maintenance overhead. In addition, note that global aggregation trees are implicit from the DHT routing and hence require no separate maintenance overhead.

Finally, since Moara maintains state information for each predicate, it could be more efficient if we aggregated different predicates. For example, predicates such as CPU-Util > 50, CPU-Util > 60, and CPU-Util > 70 could be aggregated as one predicate, CPU-Util > 50, so that Moara could maintain only one tree. This design choice requires careful study on the tradeoff between the state maintenance overhead and the bandwidth overhead incurred by combining different trees with the same attribute. This is outside of the scope of this paper, since we focus on the tradeoff of the bandwidth overhead based on the query rate and the group churn rate.

5 Separate Query Plane

Given a tree that contains m predicate-satisfying nodes, using the pruned DHT trees of the previous section may lead to $O(m \log N)$ additional nodes being involved in the tree. These extra nodes would typically be internal tree nodes that are forwarding queries down or responses up the tree, but which do not satisfy the predicate themselves. This section proposes modifications to the protocol described in Section 4 in order to reduce the traffic through these internal nodes.

Our idea is to bypass the internal nodes, thus creating a *separate query plane* which involves mostly nodes satisfying the predicate. This optimizes the tree that we built (Section 4) further by eliminating unnecessary internal nodes. This reduces the tree to contain only $O(m)$ nodes, and thus resolves queries with message costs independent of the number of nodes in the system. Note that this technique has similarities to adaptations of multicast trees (e.g., Scribe [18]), but Moara needs to address the challenging interplay between dynamic adaptation and this short-circuiting.

To realize a separate query plane, each node uses the states, constraints and transitions as described in Section 4. In addition, each node runs operations using two locally maintained sets: (i) *updateSet* is a list of nodes that it forwards to its parent; (ii) *qSet* is a list of children or descendant nodes, to which it forwards any received queries. We consider first, for ease of exposition, modified operations only for nodes in the UPDATE state. When a leaf node in UPDATE state begins to satisfy the tree predicate, it changes to SAT state as described in Section 4 and sets its *UpdateSet* to contain its ID. In addition, when sending a NO-PRUNE message to its parent, it also sends the *updateSet*. Each internal node in turn maintains its *qSet* as the union of the latest received *updateSet*s from all its children, adding its own ID (IP and port) if the tree predicate is satisfied locally. The leaf nodes do not need to maintain *qSet*s since they do not forward queries.

Finally, each internal node maintains its *updateSet* by continually monitoring if $|qSet| < threshold$, where *threshold* is a system parameter. If so, then *updateSet* is the same as *qSet*, otherwise *updateSet* contains a single element that is the node's own ID regardless of whether the predicate is satisfied locally or not. Whenever the *updateSet* changes at a node and is non-empty, it sends a NO-PRUNE message to its parent along with the new *updateSet* informing the change. Otherwise, it sends a PRUNE message.

The above operations are described assuming that all nodes are in UPDATE state. When a node is NO-UPDATE state, it maintains *qSet* and *updateSet* as described above,

but does not send any updates to its parent. For correctness, a node moving from UPDATE to NO-UPDATE state sends its own ID along with the NO-PRUNE message to its parent so that it receives future queries.

If parameter *threshold*=1, the above mechanisms produce the pruned DHT tree described in Section 4, while *threshold* > 1 gives trees based on a separate query plane. This is because with *threshold*=1, an internal node that receives an *updateSet* from any of its children will pass along to its parent an *updateSet* containing its own ID, even if the predicate is not satisfied locally. However, with *threshold* > 1, the only internal nodes that do not satisfy the predicate locally but receive queries, are ones that are maintaining a *qSet* of size $\geq$ *threshold*. Such nodes are required to receive queries so that they can be forwarded to its descendants. However, the tree bypasses several other nodes that do not satisfy the predicate, thus obtaining bandwidth savings. Specifically, an internal node that has $|qSet| <$ *threshold* and does not satisfy the predicate, does not include its own ID in the *updateSet*, and thus does not receive queries.

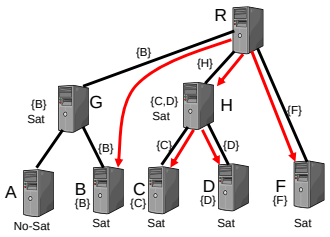

Fig. 5: Separate Query Plane for *threshold*=1. We assume all nodes are in UPDATE mode. Each node's *qSet* is shown next to it, and *updateSet* on the link to its parent.

Having a high value of *threshold* in the system bypasses several internal nodes in the tree. However, this comes at the expense of a higher update traffic since any *updateSet* changes need to be communicated to the parent. Figure 5 shows an example with *threshold*=1.

Adaptation and SQP: Our SQP design with *updateSet* and *qSet* variables at nodes, as described above, allow us to easily use the adaptation policy rules described in Section 4. In this case, q_n at a node is the number of queries received by the system when that node's *updateSet* does not contain its ID (similar to NO-SAT state) and q_s is the number of queries received at other times. The number of changes c is the number of changes to the *updateSet* variable. With these definitions, a node can use same adaptation policies as described in Section 4. One exception is the use of the query sequence number: for correct calculation of q_n at a bypassed node, each node piggybacks its last seen sequence number alongside all its status update messages to its parent.

Overhead analysis: For a group with m nodes, we analyze the overhead for forwarding a query in the separate query plane, assuming all nodes are in UPDATE state. First, notice that all leaf nodes in this tree satisfy the predicate - if some leaf did not, then it would be pruned out by the above rules. Second, the tree has the maximum links when all m predicate-satisfying nodes are at the leaves of this tree. This means that since *threshold* > 1, no internal node (other than the root node) in the tree has fewer than 2 children - if it did, it would be bypassed by the above rules. However, no tree with m leaves and internal node degree > 2 has more than m internal nodes. Thus, the total number of nodes, other than the root, receiving the query is $\leq 2 \cdot m = O(m)$, independent of system size.

6 Composite Queries

So far, we have described how to build and maintain a *single* tree corresponding to one simple predicate. We now describe how a query with a composite predicate is satisfied. Specifically, we first expand on the multiple possible trees, one tree per simple predicate in the composite query, that such a query entails (Section 6.1). Then, we explain how Moara plans a given query (Section 6.2), and how it selects a low-cost groups of nodes to execute a given composite query (Section 6.3).

6.1 Maintaining Multiple Trees

Section 4 explains the maintenance of trees for simple predicates, starting from the time a predicate is first encountered. If this predicate does not reappear again in subsequent queries in the system, then all nodes in the tree will eventually move to NO-UPDATE state (due to group churn events), and thus there will be no load, either query or update, along the tree. Thus, Moara trees become silent and incurs zero bandwidth cost if not used, obviating the need to explicitly delete trees for simple predicates. Furthermore, Moara does not maintain trees for composite queries, since these might be exponentially large in number - instead, it decides which simple predicate trees (existing or not) will be selected to execute a given composite query. This decision process is described next.

6.2 Composite Query Planning

Consider the following composite query: "find the average free memory across machines where service X and Apache are running". Suppose we have one group tree for (ServiceX=true) and another tree for (Apache=true). A naïve way to resolve the query would be to query both trees in parallel. However, we observe that bandwidth can be saved, without compromising completeness of answers, by (1) sending the query to any *one* of the trees (because it is an intersection query), and (2) choosing the tree that incurs a lower query cost.

Based on this observation, Moara answers arbitrary nested queries involving `and` and `or` boolean expressions across simple predicates by selecting a small *cover*. A cover for a given composite query Q is defined as a set of groups (selected from among simple predicates inside Q) which together contain all nodes that satisfy the composite predicate in Q. Thus, we only need to send Q to a cover to obtain a complete answer.

We can compute a cover for a query Q by exploring the boolean expression structure recursively as follows:

- cover(Q="A") = $\{$A$\}$ if A is a predefined group.
- cover(Q="A `or` B") = cover(A) $\cup$ cover(B).
- cover(Q="A `and` B") = cover(A), cover(B), or (cover(A) $\cup$ cover(B)).

For example, for a query with expression ((A `and` B) `or` C), the above rules derive $\{$A,C$\}$, $\{$B,C$\}$, and $\{$A,B,C$\}$ as possible covers. We call such covers as *structural* covers since we infer them from the structure of the boolean expression.

Once the query originating node calculates the cover for a given query Q, the composite query is forwarded to the roots of trees corresponding to each group in the cover, the answers from these trees are aggregated, and finally returned to the querying node. Notice that it is possible for some node(s) to receive multiple copies of the query, if they are present in multiple trees which appear in the cover for Q. Such nodes reply with the

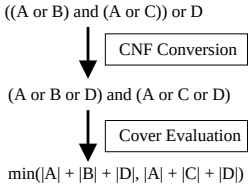

Fig. 6: Example query processing

Semantic information	(A or B)	(A and B)
$A \cap B = \phi$	{A,B}	{}
$A \cap B \neq \phi$ and $A \supseteq B$ and $A \subseteq B$ $\Rightarrow$ (A=B)	{A}	{A}
$A \cap B \neq \phi$ and $A \supseteq B$ and $A \nsubseteq B$	{A}	{B}
$A \cap B \neq \phi$ and $A \not\supseteq B$ and $A \subseteq B$	{B}	{A}
$A \cap B \neq \phi$ and $A \not\supseteq B$ and $A \nsubseteq B$	{A,B}	{A},{B}

Fig. 7: Semantic info to reduce cover sizes

attribute value to only *one* of the trees they are present in, eliminating duplicate answers. This requires nodes to remember the query ids (based on sender IP and sequence number). Such information is cached for 5 minutes in our Moara implementation.

To further save on bandwidth, we would like to select a low-cost cover. This is done by minimizing both the number of groups in the selected cover, as well as the total cost of querying this cover. We explore below three ways of deriving a low-cost cover: (1) *structural optimizations*, which rewrite the nested query to select a low-cost structural cover consisting of simple predicates that already appear within the query, (2) *estimates of query costs* for individual trees, and (3) *semantic optimizations*, which take into account semantic information obtained from users or query attributes.

6.3 Query Optimization: Finding Low-Cost Covers

Given a composite query, Moara first transforms it into a Conjunctive Normal Form (CNF) expression using distributive laws of `and` and `or` operators. A CNF form is a two level expression of `and`'s across a series of `or` terms.

It is important to notice that in the CNF form of a composite predicate for query Q, each series of `or` terms is a possible cover - this is due to the same reason as our intersection optimization explained earlier. Thus, if Moara can evaluate the query cost of each of these structural covers (as a sum of the query costs for all sets in the cover), then it can select the minimal cost cover for executing the query Q. We will describe query cost calculation soon, but before that we give an example of the query rewriting as well as proof sketch of why the CNF form gives the *minimal-cost* cover for a composite predicate.

Figure 6 shows an example transformation. Consider a query targeting ((A `or` B) `and` (A `or` C)) `or` D. Moara first transforms the expression to the equivalent CNF: (A `or` B `or` D) `and` (A `or` C `or` D). Moara chooses one cover between the two structural covers - either $\{A, B, D\}$ or $\{A, C, D\}$, whichever has a lower cost.

If query cost estimates for individual groups are up-to-date and available at the tree roots, we can prove by contradiction that our structural optimizations produce a cover that is minimum in cost. Suppose that the given CNF expression is $E = A_1$ `and` A_2 `and` $\ldots$ `and` A_n, where each term A_i is an `or` of positive literals and hence a structural cover for E. Assume the contrary, *i.e.*, suppose there exists a structural cover C with a lower cost than our covers. In each term A_i of expression E, if we substitute the literals from set C with 0, the expression should evaluate to 0 (since C is a structural cover). However, since A_i's are `and`-ed, there has to be *some* A_j that evaluates to 0 in this substitution. Thus, all groups in this A_j have to be a part of C. However, this is a contradiction since A_j is a cover with cost no more than C.

Relation between pair of groups	Description	Example pair of groups
Intersection (without inclusion)	Two groups intersect properly	(CPU-Util $<$ 50), (CPU-Util $>$ 20)
Discontinuous Intersection	The intersection is not continuous	(CPU-Util $<$ 50), (CPU-Util $\neq$ 20)
Equivalence	Two groups are identical	(CPU-Util $<$ 50), (CPU-Util $<$ 50)
Inclusion	One group is a subset of another	(CPU-Util $<$ 50), (CPU-Util $<$ 20)
Disjointedness	Two groups do not intersect	(CPU-Util $<$ 50), (CPU-Util $>$ 80)

Fig. 8: Defining operators of groups that allow relation inference between two groups

Estimating Query Costs for Trees: In order to enable low-cost cover calculation, the root node of each tree for a simple predicate continually maintains the query cost for that tree. The query cost is fetched by the querying node and used in the low-cost cover calculation described above. Within the tree, the cost for each query is simply $2 \times np$, where np is the number of nodes in NO-PRUNE state. The values of np are aggregated continually up the tree. Each internal node stores this count for its own subtree, modifies the count according to its own state, and piggybacks this information atop all updates and query responses to its parents. Although this lazy updating of the counts means the query costs may be stale at times, this only affects communication overhead, but not the correctness of the response.

Using Semantic Optimizations: If semantic information is available about the groups, then Moara can further optimize the communication costs by choosing a better cover. We explore two kinds of semantic information in our system: (i) information from description of the group, and (ii) user supplied semantic information. For example, consider two groups A and B defined as follows: $A = \{\text{nodes with memory} < 2G\}$ and $B = \{\text{nodes with memory} < 1G\}$. Then, we can infer from these definitions that $B \subseteq A$. In Figure 8, we list relations between two groups that Moara infers by analyzing the operations that define those pair of groups. Once we have semantic information either inferred from the description of the groups or supplied by a user, Moara applies the optimizations detailed in Figure 7 to obtain a low-cost cover. As another instance, Moara implicitly supports `not` operator by observing complement relations in the specified groups (the last row in Figure 8), and the following optimizations:

- (A `or` B) `and` (A `or` C) = A, if C = `not` (B)
- (A `or` C) `and` B = A `and` B, if C = `not` (B)
- (A `or` B) `and` C = A `and not` (B), if C = `not` (B)

7 Implementation and Evaluation

We have built a prototype of Moara using SDIMS [7] and FreePastry [8]. All other Moara protocols, described in Section 3 through Section 6, are built atop these systems. Here, we discuss our implementation details and evaluation methodology.

Moara Front-End: The Moara front-end is a client-side interface of Moara. It includes an interactive shell, a query parser, and a query optimizer. Through the interactive shell, a user can submit SQL-like aggregation queries to Moara. The query parser parses the queries, and the query optimizer determines the groups that need to be queried through the algorithm described in Section 6. Once the front-end determines the groups to be queried, it generates a *sub-query* for each group. Each sub-query is resolved exactly the same way as a normal query, except that the front-end waits until it receives all the results from sub-queries, aggregates the results returned by the sub-queries, and returns the final aggregate to the user.

Reconfigurations: To handle reconfigurations, we leverage the underlying FreePastry mechanism for failure detection and neighbor set repair. When a node gets a new parent for a predicate, it sends its current state information (e.g., *updateSet*) for that predicate to the new parent. Also, when a node is waiting for a response from a child and if the underlying DHT notifies that the child has failed or is not reachable, then the node will proceed assuming a NULL response from the child. In addition to this, Moara also implements a time-out mechanism (in waiting for a child's reply to a query) to ensure that all queries are responded to independent of FreePastry's timeout values for failure detection.

Evaluation Environments: We use simulation, Emulab, and PlanetLab, and choose a suitable environment to evaluate each of our design choices. We use simulation exclusively for measuring bandwidth consumption in a large-scale environment. We use Emulab and PlanetLab to mainly measure the latency in realistic environments, namely, a medium-scale datacenter (Emulab) and a wide-area infrastructure (PlanetLab).

For each design choice (group-based aggregation, dynamic maintenance, separate query plane, and composite query processor), we choose the evaluation environments that are most suitable. First, we evaluate group-based aggregation on Emulab and PlanetLab, since group-based aggregation is designed to reduce both latency and bandwidth consumption. Second, we evaluate dynamic maintenance and separate query plane using simulation, since both mechanisms are designed for bandwidth optimization and have wide choices of parameters. However, we evaluate the separate query plane on Emulab as well to measure the latency. Lastly, we evaluate our composite query processor on Emulab, since it only affects latency.

Workload: The workload is characterized by two factors - group churn rate and query rate. First, since a group is defined over a particular attribute, the group churn rate depends on how dynamic the attribute is (e.g., a group of (OS = Linux) is likely to be static, while a group of (CPU-util $< 60\%$) is likely to be dynamic). Second, the query rate depends on the usage of Moara and is expected to vary widely. For example, a datacenter operator might typically query a group once an hour on a day, but several times a minute on days with high workloads or unscheduled downtimes. Thus, we parameterize these factors and present the performance of Moara over the parameter range.

7.1 Simulation Results

We perform simulation experiments to measure the bandwith overhead of Moara's dynamic tree maintenance and separate query plane. Our simulations are performed with the FreePastry simulator environment, simulating up to 16,384 nodes. Each node maintains an attribute A with value $\in \{0, 1\}$. All queries are simple queries for (A, SUM, $A = 1$), which counts the number of nodes where A is set to 1.

Dynamic Maintenance: To study the dynamic maintenance mechanism under different workload types, we stress the system by injecting two types of events - query events and group churn events - at different ratios. For example, a query:churn ratio of 0:500 represents an extreme type of workload where there is high group churn, but no queries at all. On the other hand, the query:churn ratio of 500:0 represents the other extreme where there is high query rate, but no group churn. Each group churn event selects m nodes at random, and toggles the value of their attribute A. The value of m

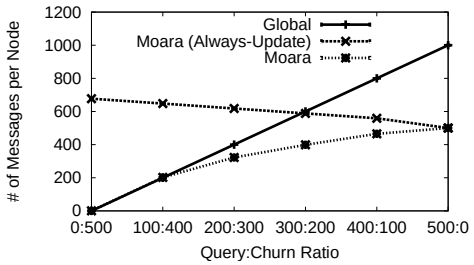

Fig. 9: Bandwidth usage with various query-to-churn ratios

determines the "burst size" of attribute churn. We fix the total number of events to 500, and randomly inject query or group churn events at the chosen ratio. All data points are averaged over 3 runs.

Figure 9 shows the average number of messages per node in Moara under various query:churn ratios, in a system of 10,000 nodes with $m = 2000$-sized group churn events. In addition to Moara, we also plot the number of messages generated by two other static approaches that lie at the opposing extremes. These are: 1) the *Global* approach, where no group trees are maintained and queries are sent to all the nodes on the DHT trees, and 2) *Moara (Always-Update)* approach, where a tree is aggressively maintained by having each child send an update to its parent on each attribute churn event.

The Global approach is inexpensive when there are fewer queries in the system, since it avoids the overhead of tree maintenance. On the other hand, with a high-query:low-churn ratio, Moara (Always-Update) performs well because it always maintains group trees and hence incurs lower traffic than Global approach. The plots show that Moara meets or lowers the message overhead in comparison to either of these extreme design choices, at all values of query:churn ratios. When group churn is high, Moara suppresses attribute churn events from propagating to other nodes. With more queries than group churn events, Moara reduces query cost by maintaining trees aggressively. Thus, Moara is able to adapt to various workload patterns. In the above experiment, we use k_{UPDATE}=1 and $k_{NO-UPDATE}$=3. Here, we study the sensitivity of the performance for different values for these knobs. Figure 10 plots the average number of messages per node in a system of 500 Moara nodes under a range of query:churn ratios for different threshold values. Although we have tried a wide range of values (each up to 10), we only show a few representative pairs that are sufficient to show the conclusion, in order to prevent the plot from being too crowded. When there are very few query events in the system (compared to churn events), different threshold values perform similarly. However, when the number of queries is high (e.g., # Queries=400), large k_{UPDATE} values (e.g., $(3, 1)$ and $(2, 1)$) coupled with small $k_{NO-UPDATE}$ values lead to slightly more overall messages. This is because with larger k_{UPDATE} and smaller $k_{NO-UPDATE}$, more Moara nodes stay in UPDATE state - thus, each child updates its parent more often, even with small churn rates. Overall, we observe that the sensitivity of the performance to different thresholds is very small. For all other experiments, we use the default values of k_{UPDATE}=1 and $k_{NO-UPDATE}$=3.

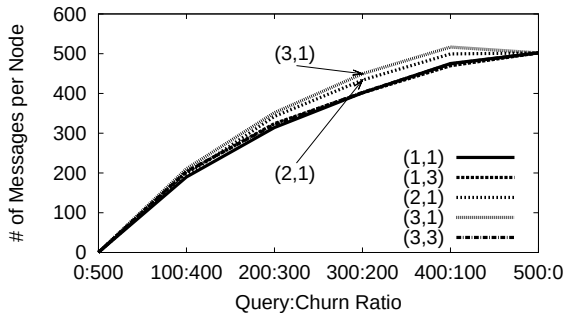

Fig. 10: Bandwidth usage with various (k_{UPDATE}, $k_{NO-UPDATE}$). Although we have tried a wide range of values in our simulation, we only show a few representative pairs for clarity of presentation.

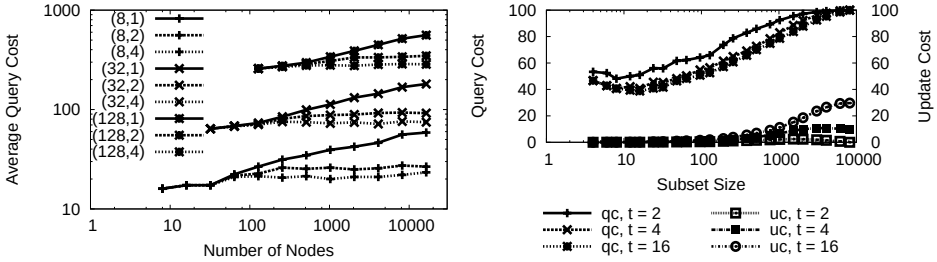

Fig. 11: (a) Bandwidth usage with (threshold > 1) and without the separate query plane (threshold=1) for different group sizes. Each line represents a (group size, threshold) pair (b) Query costs (qc) and update costs (uc) of the separate query plane in a 8192-node system

Separate Query Plane: In Figure 11(a), we plot the query cost against the number of nodes in the system for different threshold values and different group sizes. Note that the *threshold* value of 1 implies the absence of a separate query plane, while higher threshold values create a separate query plane (refer to Section 5). For this experiment, we do not introduce any group churn during the experiment. We perform 1,000 queries and compute the average of the query cost. Even though there is no group churn, there are updates sent by nodes to their parents as they move into UPDATE state with the first query message. We count those messages as the update cost.

Figure 11(a) shows that without the separate query plane (*threshold*=1), the query cost increases logarithmically as the total system size is raised. However, while maintaining a separate query plane (*threshold*>1), the query cost reaches a constant value and stays flat, independent of the number of nodes in the system. While increasing the value of *threshold* decreases query cost, it can lead to more update messages as discussed in Section 5. In Figure 11(b), we plot the query costs for different *threshold* values as a percentage of the query cost for *threshold*=1 and also plot the percentage increase in the update costs in comparison to *threshold*=1. From these two plots, we observe that (1) with small groups and large total nodes (e.g., 8192 total nodes with group size=8 or 32), using a query plane saves more than 50% bandwidth in query costs, and (2) while using a higher value of threshold does reduce bandwidth, the savings are marginal beyond a threshold of 2 and can incur higher update costs at large group sizes.

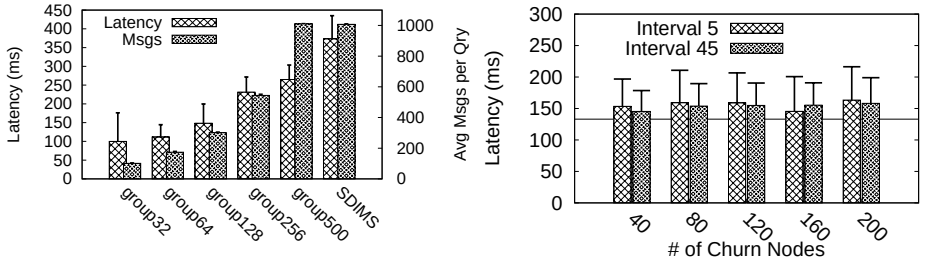

Fig. 12: (a) Latency and bandwidth usage with static groups (b) Average latency of dynamically changing groups. The horizontal line shows the average latency with a static group of the same size.

7.2 Emulab Experiments

In this section, we study both the latency and communication overhead of Moara under a real deployment scenario in Emulab, that emulates a medium-scale datacenter. Specifically, we evaluate three different workloads. First, we study performance of Moara when querying groups of static attributes (e.g., OS = Linux). We vary the size of groups and show the benefits of using Moara. Second, we study Moara with groups defined over dynamic attributes (e.g., CPU-util $< 60\%$). We stress Moara by varying the frequency of changes. Third, we study composite queries with varying numbers of groups per query.

Methodology: We create a network of 50 machines on a 100 Mbps LAN and instantiate 10 instances of Moara on each machine, thus emulating a 500 node Moara system. Each experimental run is started with one bootstrap node, followed by a batch of 100 new instances joining after intervals of 10 seconds each. After the last join, we wait an additional 5 minutes to warm up before initiating queries and group churn from a Moara node. Since we are mainly interested in per-query latency and bandwidth consumption, we fix the query rate and repeat the same query multiple times. As previously, each node maintains one binary attribute A. Our default query is a count, providing the number of nodes with A=1. All data points are the average of 3 runs.

Static Groups: Figure 12(a) compares the performance of Moara (with separate query plane) w.r.t. both latency and bandwidth. We vary the group sizes and query 100 times for each experiment. In addition, we compare this performance against an approach where a single global tree is used system-wide - this is labelled as the *SDIMS* approach in the plot. As we can see from the figure, Moara's latency and bandwidth scale with the size of the group. The savings are the most significant for small groups (e.g., set32 which has 32 nodes), where the savings compared to the SDIMS approach are up to 4X in latency and 10X in bandwidth. The latency is reduced due to the use of separate query plane because of short-circuiting long chains of intermediate nodes.

Dynamic Groups: We study the effect of group churn due to attribute-value changes at individual nodes. We considered a group of 100 nodes, with group churn controlled by two parameters *churn* and *interval*. Every *interval* seconds, we randomly select *churn* nodes in the group to leave, and *churn* nodes outside the group to join.

Figure 12(b) shows the effect on query latency, of different *churn* values (x-axis) for two different *interval* values. Queries are inserted at the rate of one query per

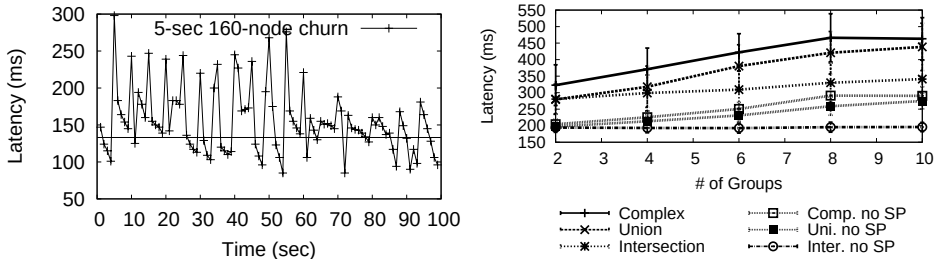

Fig. 13: (a) Latency over time with a dynamically changing group. The horizontal line shows the average latency with a static group of the same size. (b) Latency with composite queries

second, and the data points are averages of 100 queries per run. The plot shows that Moara's query latency is not affected significantly by group churn - (1) even when we increase the group churn rate by a 9-fold factor from *Interval*=45 to *interval*=5, Moara experiences only a small increase in latency, and (2) the latency stays low, and around 150 ms even when the entire group membership changes every 5 seconds (*interval*=5, *churn*=200).

Figure 13(a) provides an insight into the workings of Moara under the above workload, for *interval*=5, *churn*=160. Notice that the spikes in query latency occur once every 5 seconds, around the time that the group churn batch occurs. However, notice that (1) the peak latency stays within 300 ms, and (2) Moara query latency stabilizes very quickly after each group churn batch, typically within 1-2 seconds. These plots thus show that Moara is highly resilient to dynamism due to rapidly occurring attribute-value changes.

Composite Queries: The experiments so far have focused on single groups in Moara. Here, we microbenchmark the performance of Moara on composite queries. Assuming $S_1, S_2, \ldots, S_n$ are simple single predicate groups, we study three types of composite queries: (1) Intersection queries of the form $S_1 \cap S_2 \cap \ldots \cap S_n$, for different values of n; (2) Union queries of the form $S_1 \cup S_2 \cup \ldots \cup S_n$, for different values of n; and (3) Complex queries, which are structured as $T_1 \cap T_2 \cap \ldots \cap T_m$, where each T_i is a union of multiple groups. These experiments suffice to characterize Moara's performance since the query optimization reduces all query expressions to one of the three. Each basic group S_i consists of 50 nodes selected at random. The complex expression we use[1] is $T_1 \cap T_2 \cap T_3$, and each T_i is a union of n basic groups for different values of n. Figure 13(b) plots the latency for above three types of queries with different values of n. For composite queries, recall that Moara first sends size probes to root nodes of group trees, in order to make a query optimization decision. Thus, we plot not only the total latency of a Moara query, but also the latency excluding the time to finish the size probes. Each data point is averaged over 300 queries.

First, notice that the average completion times of all queries, including queries with up to 10 groups, is less than 500 *ms*. For intersection queries, the completion times excluding time for size probes (plot line "Inter. no SP") do not depend on the size of the expression. This is because Moara selects only one of these groups to propagate the

[1] We found that the number of T_i's has little effect on latency because Moara queries only one of all T_i's.

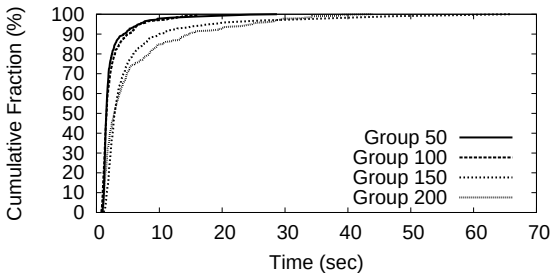

Fig. 14: PlanetLab Latency

query. Although size probes are sent in parallel, the latency for size probes increases slightly since Moara waits until the slowest probe response arrives. For union queries, the total completion time of a query rises gradually with the size of the expression, as Moara needs to contact all groups (two "Union" plots). Finally, the completion time for complex queries is only slightly more than that of union queries, since Moara's query optimization selects only one of T_i's. The additional latency is caused by two factors: (a) the time taken for size probes is higher as we have to query the sizes for larger number of groups, and (b) a complex set expression adds more overhead at each node, because each node evaluates the entire complex expression to determine if it satisfies it or not (this step could be further optimized).

7.3 PlanetLab Experiments

Methodology: We deploy Moara atop 200 PlanetLab nodes, which span several continents. Each PlanetLab node runs one instance of Moara. The instances are started sequentially, the system is given 5 minutes to warm up, and then a series of queries is injected from a Moara front-end running on a local machine. In order to study the behavior of Moara's query latency in-depth, we perform experiments on only one group at a time, but for different sizes of this group. Each experiment involves a total of 500 queries injected 5 seconds apart. All plotted data points are the average of 3 runs. We do not timeout on queries, in order to obtain complete answers.

Query Response Latency: Figure 14 plots the cumulative fraction of replies received as a function of time since query injection, on four different-sized groups. The plot shows the responsiveness of Moara in a wide-area setting - even with as many as 100 nodes in the group, the median answer is received back within 1-2 seconds, while 90% of the answers are received within 5 seconds.

Moara versus Centralized Aggregation: Figure 15 compares Moara against a centralized approach which maintains no trees but has the Moara front-end directly query all nodes in parallel regardless of whether they satisfy the given predicate or not (labelled "Central"). The response for a query from this centralized aggregator is considered complete when the centralized aggregator has received a response from every node regarding the query. The figure plots the cumulative fraction of replies received as a function of time since query injection. This plot illustrates that the comparison between the centralized aggregator and Moara is intuitively akin to the comparison of "the tortoise and the hare". In other words, for both groups of size 100 and 150, we notice

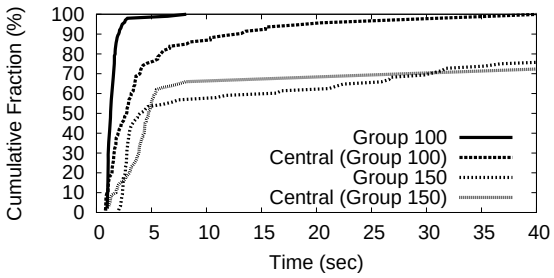

Fig. 15: Moara vs. Centralized Aggregator

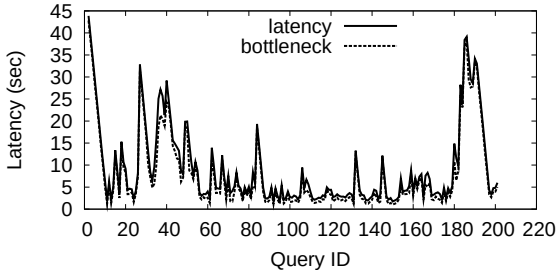

Fig. 16: PlanetLab Bottleneck Latency

that the centralized aggregator obtains initial replies faster than Moara, but then it slows down waiting for the remainder of the query answers from nodes.

Figure 16 further explains why Moara's overall completion time is shorter than the centralized aggregator with smaller groups. We plot the total completion latency for a Moara query in a 200-node group, along with the latency on a single bottleneck link in the Moara tree. This bottleneck link is obtained via offline analysis, and by picking the largest round-trip-time among all parent-child pairs in the tree. This plot shows that it is a *single* bottleneck link that contributes to the latency of Moara. Moara is faster overall in obtaining a large fraction of replies, because it avoids bottlenecks that are not part of the queried group. In comparison, the centralized aggregator is subject to being slowed down by *all* nodes that suffer from bottlenecks, mainly the slowest bottleneck.

8 Related Work

PlanetLab has several management tools in use, such as CoTop, CoMon, etc [15]. However, none of the tools addresses scalability and expressive queries simultaneously. Several distributed systems have been proposed for aggregating data. Astrolabe [6] provides a generic aggregation abstraction, but uses a single static tree and hence has limited scalability with the number of metrics. SDIMS [7] constructs multiple trees for scalability with the number of metrics, but assumes a single group of the entire system. PIER [20] supports recursive SQL-style queries, but does not leverage in-network aggregation. Huebsch *et al.* [21] present a way to optimize global aggregation queries, while Moara optimizes multiple group-based aggregation trees. Seaweed [5] focuses on dealing with data unavailability. MON [22] supports one-shot queries and constructs query trees on-demand, but does not support expressive queries. Finally, Ganglia [14] uses a single hierarchical tree, but collects all data without in-network aggregation.

Vanilla DHTs by themselves, e.g., Pastry [8], Chord [17], Tapestry [16], Kelips [23], etc., do not support complex queries. In sensor-networks, there are several aggregation systems whose goals bear similarities to ours (e.g., TinyDB [24] and Synopsis diffusion [25]). However, their applicability is limited to wireless sensor networks with multi-hop connectivity; utilizing these techniques in our setting would contact a large number of nodes for any complex query.

Structured overlay based multicast systems such as Scribe [18], SAAR [26], and SelectCast [27] bear some similarities with Moara, e.g., path collapsing of Scribe [18], the shared control plane idea of SAAR [26], and predicate-based multicast of SelectCast. However, all these system focus on building efficient trees for multicast where maintenance overhead is assumed to be much smaller than the data plane costs. CUP [28] and Shruti [29], while proposing adaptation techniques to reduce query cost, addresses a different optimization problem than us. In these systems, queries are only spread down to the nodes where updates are also propagated to (rendezvous points). Moara uses updates for pruning the group trees and queries are sent to all predicate-satisfying nodes.

9 Conclusion

In this paper, we have presented the design and evaluation of Moara, a group-based aggregation system. Moara achieves scalability with increasing numbers of machines, injected queries, and groups, by: (1) intelligently resolving composite query expressions, (2) constructing single-attribute aggregation trees that perform in-network aggregation, and (3) dynamically maintaining group trees based on query rates and group churn rates, thus reducing bandwidth consumption. Our experimental evaluations using simulations and deployments atop Emulab and PlanetLab demonstrate the effectiveness of Moara in answering queries accurately within hundreds of milliseconds across hundreds of nodes, and with low per-node bandwidth consumption.

References

[1] HP: HP Data Centre Consolidation. `http://h20331.www2.hp.com/enterprise/cache/141741-0-0-225-121.html`

[2] PlanetLab. `http://www.planet-lab.org/`

[3] NSF: The NSF GENI Initiative. `http://www.nsf.gov/cise/geni/`

[4] Foster, I.T.: The Grid2003 Production Grid: Principles and Practice. In: Proc. HPDC-13. (2004)

[5] Narayanan, D., Donnelly, A., Mortier, R., Rowstron, A.: Delay Aware Querying with Seaweed. In: Proc. VLDB. (2006)

[6] Renesse, R.V., Birman, K.P., Vogels, W.: Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. ACM Trans. on Comp. Syst. **21**(2) (May 2003) 164 – 206

[7] Yalagandula, P., Dahlin, M.: A Scalable Distributed Information Management System. In: Proc. SIGCOMM. (2004)

[8] Rowstron, A., Druschel, P.: Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In: Proc. Middleware. (2001)

[9] Emulab. `http://www.emulab.net`

[10] Brewer, E.: Towards Robust Distributed Systems (Invited Talk). In: Proc. PODC. (2000)

[11] Jain, N., Kit, D., Mahajan, P., Yalagandula, P., Dahlin, M., Zhang, Y.: STAR: Self Tuning Aggregation for Scalable Monitoring. In: 33rd International Conference on Very Large Databases (VLDB). (2007)
[12] Jain, N., Kit, D., Mahajan, P., Yalagandula, P., Dahlin, M., Zhang, Y.: PRISM: Precision-Integrated Scalable Monitoring (extended). In: 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI). (December 2008)
[13] Park, K., Pai, V.S.: CoMon: a Mostly-scalable Monitoring System for PlanetLab. SIGOPS OSR **40**(1) (2006) 65–74
[14] Massie, M.L., Chun, B.N., Culler, D.E.: The Ganglia Distributed Monitoring System: Design, Implementation and Experience. Parallel Computing **30**(7) (July 2004)
[15] PlanetLab: Contributed Software. `https://wiki.planet-lab.org/twiki/bin/view/Planetlab/ContributedSoftware`
[16] Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.: Tapestry: A Resilient Global-scale Overlay for Service Deployment. IEEE JSAC **22**(1) (January 2004)
[17] Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proc. SIGCOMM. (2001)
[18] Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.: SCRIBE: A Large-scale and Decentralised Application-level Multicast Infrastructure. IEEE JSAC (2002)
[19] Castro, M., Druschel, P., Kermarrec, A.M., Nandi, A., Rowstron, A., Singh, A.: SplitStream: High-bandwidth Multicast in a Cooperative Environment. In: Proc. SOSP. (2003)
[20] Huebsch, R., Chun, B., Hellerstein, J.M., Loo, B.T., Maniatis, P., Roscoe, T., Shenker, S., Stoica, I., Yumerefendi, A.R.: The Architecture of PIER: an Internet-Scale Query Processor. In: Proc. CIDR. (2005)
[21] Huebsch, R., Garofalakis, M., Hellerstein, J.M., Stoica, I.: Sharing Aggregate Computation for Distributed Queries. In: UC Berkeley Tech Report UCB/EECS-2006-98. (2006)
[22] Liang, J., Ko, S.Y., Gupta, I., Nahrstedt, K.: MON: On-demand Overlays for Distributed System Management. In: Proc. USENIX WORLDS. (2005)
[23] Gupta, I., Birman, K.P., Linga, P., Demers, A.J., van Renesse, R.: Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead. In: Proc. IPTPS. (2003)
[24] Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: An Acqusitional Query Processing System for Sensor Networks. ACM Trans. on Database Syst. **30**(1) (March 2005) 122–173
[25] Nath, S., Gibbons, P.B., Seshan, S., Anderson, Z.R.: Synopsis Diffusion for Robust Aggregation in Sensor Networks. In: Proc. SenSys. (2004)
[26] Nandi, A., Ganjam, A., Druschel, P., Ng, T.S.E., Stoica, I., Zhang, H., Bhattachargee, B.: SAAR: A Shared Control Plane for Overlay Multicast. In: Proc. NSDI. (2007)
[27] Bozdog, A., van Renesse, R., Dumitriu, D.: SelectCast: A scalable and self-repairing multicast overlay routing facility. In: SSRS '03: Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems, New York, NY, USA (2003) 33–42
[28] Roussopoulos, M., Baker, M.: CUP: Controlled Update Propagation in Peer-to-Peer Networks. In: USENIX. (2003)
[29] Yalagandula, P., Dahlin, M.: Shruti: A Self-Tuning Hierarchical Aggregation System. In: SASO 2007