Wide-Scale Data Stream Management

Dionysios Logothetis and Kenneth Yocum
UCSD Department of Computer Science & Center for Networked Systems

Abstract

This paper describes Mortar, a distributed stream processing platform for building very large queries across federated systems (enterprises, grids, datacenters, testbeds). Nodes in such systems can be queried for distributed debugging, application control and provisioning, anomaly detection, and measurement. We address the primary challenges of managing continuous queries that have thousands of wide-area sources that may periodically be down, disconnected, or overloaded, e.g., multiple data centers filled with cheap PCs, Internet testbeds such as Planetlab, or country-wide sensor installations.

Mortar presents a clean-slate design for best-effort in-network processing. For each query, it builds multiple, static overlays and leverages the union of overlay paths to provide resilient query installation and data routing. Further, a unique data management scheme mitigates the impact of clock skew on distributed stream processing, reducing result latency by a factor of 8, and allows users to specify custom in-network operators that transparently benefit from multipath routing. When compared to a contemporary distributed snapshot querying substrate, Mortar uses a fifth of the bandwidth while providing increased query resolution, responsiveness, and accuracy during failures.

1 Introduction

There is a growing need to monitor, diagnose, and react to data and event streams emitted by wide-scale networked systems. Examples include big-box retailers analyzing retail streams across thousands of locations, real-time weather predictors sourcing hundreds of doppler radars [13], studying network attacks with distributed Internet telescopes [3] or end systems [11, 15], and anomaly detection across wired [20] or wireless network infrastructures [9, 2].

These systems represent a global pool of nodes continuously emitting system and application-specific data streams. In these scenarios, in-network data processing is often necessary as the data streams are too large, too numerous, or the important events within the streams too sparse to pay the cost of bringing the data to a central location. While distributed data processing is important for monitoring large backbone networks [36, 16], where ISPs collect summary statistics for thousands of network elements, other important applications are emerging. For instance, end-system monitoring specifically leverages the host vantage point as a method for increasing the transparency of network activity in enterprise networks [11] or observing the health of the Internet itself [15]. These environments pose challenges that strongly affect stream processing fidelity, including frequent node and network failures and mis-configured or ill-behaved clock synchronization protocols [24]. This has recently been referred to as the “Internet-Scale Sensing” problem [26].

Mortar provides a platform for instrumenting end hosts, laptop-class devices, and network elements with data stream processing operators. The platform manages the creation and removal of operators, and orchestrates the flow of data between them. Our design goal is to support best-effort data stream processing across these federated systems, specifically providing the ability to manage in-network queries that source tens of thousands of streams. While other data management systems exist, their accuracy is often encumbered by processing all queries over a single dynamic overlay, such as a distributed hash table (DHT) [18, 41, 27]. Our own experience (and that of others [33]) indicates an impedance mismatch between DHT design objectives and in-network stream processing. Even without failures, periodic recovery mechanisms may disrupt the data management layer during route table maintenance, inconsistencies, and route flaps [17].

Mortar incorporates a suite of complementary techniques that provide accurate and timely results during failures. Such an ability facilitates stream processing across federated environments where the set of all nodes in the system is well known, but many nodes may periodically be down, disconnected, or overloaded, e.g., multiple data centers filled with cheap PCs, Internet testbeds such as Planetlab, or city-wide sensor installations [25]. This work complements prior research that has primarily focused on querying distributed structured data sources [18, 14, 27], processing high speed streams [19], managing large numbers of queries [8], or maintaining consistency guarantees during failures [4].

This paper makes the following contributions:

- **Failure-resilient aggregation and query manage-**

ment: Mortar uses the combined connectivity of a static set of overlay trees to achieve resilience to node and network failures. By intelligently building the tree set (Section 3), the overlays are both network aware and exhibit sufficient path diversity to connect most live nodes during failures. Our data routing (Section 3.3) and query recovery protocols (Section 6) ensure that even when 40% of the nodes in a given set are unavailable, the system can successfully query 94% of the remaining nodes.

- **Accurate stream processing in the presence of clock offset:** The lack of clock synchronization, such as the presence of different clock skews (frequencies), can harm result fidelity by changing the relative time reported between nodes (relative clock offset)[1]. For distributed stream processing this can increase latency and pollute the final result with values produced at the wrong time. Mortar's *syncless* mechanism (Section 5) replaces traditional timestamps with *ages*, eliminating the effect of clock offset on results and improving result latency by a factor of 8.
- **Multipath routing with duplicate-sensitive operators:** Mortar's *time-division* data management model isolates data processing from data routing, allowing duplicate-free processing in the presence of multipath routing policies. This enables our dynamic tuple striping protocol (Section 3.3), and allows user-defined aggregate operators, without requiring duplicate-insensitive operators or synopses [28].

Extensive experimentation with our Mortar prototype using a wide-area network emulator and Internet-like topologies indicate that it enables accurate best-effort stream processing in wide-scale environments. We compare its performance to a release of SDIMS [41], an aggregating snapshot query system, built on the latest version of FreePastry(2.0_03). Mortar uses 81% less bandwidth with higher monitoring frequency, and is more accurate and responsive during (and after) failures. Additionally, Mortar can operate accurately in environments with high degrees of clock offset, correctly assigning 91% of the values in the system to the right 5-second window, outperforming a commercial, centralized stream processor. Finally, to validate the design of the operator platform, we design a Wi-Fi location sensing query that locates a MAC using three lines of the Mortar Stream Language, leveraging data sourced from 188 sensors throughout a large office building.

2 Design

As a building block for data processing applications, Mortar allows users to deploy continuous queries in federated environments. It is designed to support hundreds of in-network aggregate queries that source up to tens of thousands of nodes producing data streams at low to medium rates, issuing one to 1000's of records a second. Given the size of such queries, we expect machine failures and disconnections to be common. Thus a key design goal is to provide failure-resilient data stream routing and processing, maximizing result accuracy without sacrificing responsiveness.

We begin by motivating a clean-slate approach for connecting continuous aggregate operators based on static overlays. A goal of this data routing substrate is to capture all constituent data that were reachable during the query's processing window [18, 41, 40], and this work uses result *completeness*, the percentage of peers or nodes whose data are included in the final result, as the primary metric for accuracy. We end this section with how users specify stream-based queries and user-defined operators in Mortar.

2.1 Motivating static overlays

While it is natural to consider a dynamic overlay, such as a distributed hash table (DHT), as the underlying routing substrate, we pursue a clean-slate design for a number of reasons. First, we desire *scoped* queries; only the nodes that provide data should participate in query processing. It is difficult to limit or to specify nodes in the aggregation trees formed by a DHT's routing policy. Second, we can reduce system complexity and overhead by taking advantage of our operating environment, where the addition or removal of nodes is rare. While Mortar peers may become unavailable, they never explicitly "join" or "leave" the system. Further, DHTs are not optimized for tasks such as operator placement [33], and, more importantly, complicated routing table maintenance protocols may produce routing inconsistencies [17].

In contrast, Mortar connects query operators across multiple *static* trees, allowing query writers to explicitly specify the participants or *node set*. Here we take advantage of the relatively stable membership seen in federated systems, which usually have dedicated personnel to address faults. Machines in these environments may temporarily fail, be shutdown for maintenance, or briefly disconnected, but new machines rarely enter or leave the system. The combined connectivity of this tree set not only allows data to flow around failed links and nodes, but also query install and remove commands. This allows users to build queries across the live nodes in their system simply with lists of allocated IP addresses.

This idea builds upon two existing, basic approaches to improving result completeness. Data mirroring, explored by Borealis [4] and Flux [37], runs a copy of the logical query plan across different nodes. Static striping, found in TAG [21], sends $1/n$ of the data up each of n different spanning trees. We compare these ap-

Technique	**Benefits**
Tree set planning (Section 3)	A *primary* static overlay tree places the majority of data close to the root operator by clustering network coordinates. Static *sibling* trees preserve the network awareness of the primary, while exhibiting the path diversity of random trees.
Dyamic tuple striping (Section 3.3)	Route tuples toward root operator while leveraging available paths. Ensures low path length and avoids cycles. Even when 40% of the nodes are unreachable, data from 94% of the remaining nodes is available.
Time-division data partitioning (Section 4)	Isolates tuple processing from tuple routing, allowing multipath tuple routing, and avoiding duplicate data processing.
Syncless operation (Section 5)	Allows accurate stream processing in the presense of relative clock offset, and reduces result latency by a factor of 8.
Pair-wise reconciliation (Section 6)	Leverages combined connectivity of D overlay trees for eventually consistent query installation and removal.

Table 1: A roadmap to the techniques Mortar incorporates.

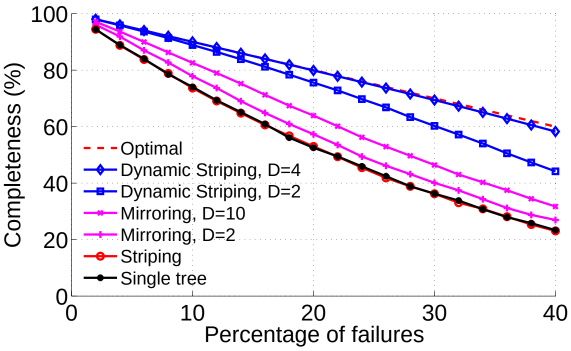

Figure 1: Result completeness under uniformly random failures for mirroring, striping, and dynamic striping. D is the tree set size.

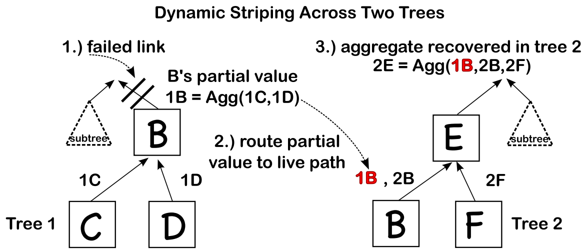

Figure 2: This figure illustrates how dynamic striping avoids failed links (or nodes). Here, after a failed link, node B routes successive partial values to node C on the second tree.

proaches for an aggregate sum operator using a simple simulation. We build random trees (with 10k nodes) of various branching factors, uniformly "fail" random links, and then simply walk the in-memory graph and count the number of nodes that remain connected to the root. Each trial subjects the same tree set to N uniformly random link failures, and we plot the average performance across 400 trials.

Figure 1 shows the ability of mirroring and striping to maintain connections to nodes under different levels of failure, number of trees (D), and a branching factor of 32. Both static options perform poorly. Striping performs no better than a single random tree; many slices of a tree behave, in expectation, as a single random tree. Data mirroring improves resiliency to failure, but at significant cost. When 20% of the links fail, mirroring across 10 trees ($D = 10$) improves consistency by 10% while increasing the bandwidth footprint by an order of magnitude. Obviously, this approach is not scalable.

Instead, we propose dynamic striping, a multipath routing scheme that combines the low overhead of static striping with adaptive overlay routing. Without failures, each operator stripes outbound data across each downstream parent in the tree set. When a failure occurs, disconnecting a parent, the operator migrates the stripe to a remaining, live parent. Note that because Mortar is a best-effort system, it does not retransmit data lost due to the failure. Figure 2 illustrates this three step process. This allows nodes to continue to push values towards the root as long as there remains a single live path across the union of upward paths in the tree set. Figure 1, shows that this technique is effective, even with a low number of stripes.

Routing data across a set of static spanning trees for each query sidesteps many of the issues raised by dynamic overlays, but poses new questions. How should one design the tree set so that it is both network aware, but provides a diverse set of overlay paths? How does one route data towards the root while ensuring low path length and avoiding cycles? How should the system prevent duplicate data processing to allow duplicate-sensitive aggregate operators? Meeting these design challenges required us to innovate in a number of areas and led to the development of a suite of complementary techniques (Table 1).

2.2 Queries and in-network operators

Mortar consists of a set of peering processes, any of which may accept, compile, and inject new queries. Each query is defined by its in-network operator type and produces a single, continuous output data stream. It may take as input one or more raw sensor data streams or subscribe to existing data streams to compose complex data processing operations. Users write queries in the Mortar Stream Language[2].

We require that all operators are non blocking; they may emit results without waiting for input from all sources. An operator's unit of computation is a tuple, an ordered set of data elements. Operators use sliding windows to compute their result, issuing answers that summarize the last x seconds (a time window) or the last x tuples (a tuple window) of a source stream. This is the window *range*; the window *slide* (again in time or tuple count) defines the update frequency (e.g., report the average of the last 20 tuples every 10 tuples).

Mortar provides a simple API to facilitate programming sophisticated in-network operators. Many application scenarios may involve user-defined aggregate functions, like an entropy function to detect anomalous traffic features or a bloom filter for maintaining an index. However, multipath routing schemes often require special duplicate and order-insensitive synopses to implement common aggregate functions [28]. When combined with a duplicate-suppressing network transport protocol, Mortar's data model (Section 4) ensures duplicate-free operation. Thus each in-network operator only needs to provide a `merge` function, that the runtime calls to inject a new tuple into the window, and a `remove` function, that the runtime calls as tuples exit the window. Each function has access to all tuples in the window. This API supports a range of streaming operators, including maps, unions, joins, and a variety of aggregating functions, which are the focus of this work.

3 Planning and using static overlay trees

Mortar's robustness relies on the inherent path diversity in the union of multiple query trees. Our physical dataflow planner arranges aggregate operators into a suitable set of aggregation trees. This means that the system deploys an operator at each source, whether it is a raw sensor stream or the output stream of an existing query. This allows operators to label the tuples according to our data model and reduce the data before crossing the network. The first planning step is to build a network-aware "primary" tree, and then to perform permutations on that tree to derive its siblings. Finally, a routing policy explores available paths while preventing routing cycles and ensuring low-length paths.

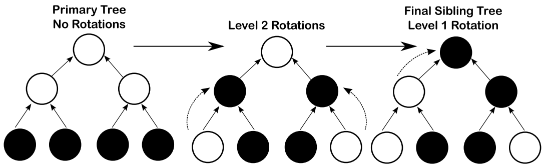

Figure 3: We derive sibling trees from the primary tree through successive random rotations of internal subtrees. This introduces path diversity while retaining some clustering.

3.1 Building the primary tree

Our primary objective is to plan an aggregation tree that places the majority of nodes "close" to the root operator. This allows the root to return answers that reflect the majority of the data quickly. The idea is to minimize the latency between stream sources and their parent operator through recursive data clustering on network coordinates [12]. In network coordinate systems, each peer produces a coordinate whose Euclidean distance from other peers predicts inter-peer latency. Our planning algorithm places operators at the centroids of clusters, avoiding high-latency paths in the top of the query tree.

Mortar treats network coordinates as a data stream, and first establishes a union query to bring a set of coordinates to the node compiling the query. Once at the compiling peer, Mortar invokes a clustering algorithm that builds full trees with a particular branching factor (bf). The recursive procedure takes a root node and the node set. It proceeds by first finding bf clusters, finding the centroids of each cluster, and making each a child of the root. The procedure is then called with each child as the root, and that child's cluster as the node set. The recursion ends when the input node set size is less than or equal to the branching factor.

This design distributes tree building across a small subset of nodes actively used to inject queries. Though the total amount of data brought to the injecting node is relatively small, 10,000 nodes issuing 5-dimensional coordinates results in 0.5MB, the cost is amortized across the compilation of multiple queries. The union query may have a slide on the order of tens of minutes, as latency measurements are relatively stable for those time periods [29].

3.2 Building sibling trees

The key challenge for building sibling trees is retaining the majority of the primary's clustering while providing path diversity. These are competing demands, large changes to the primary will create a less efficient tree.

Our algorithm works in a bottom-up fashion, pushing leaves into the tree to create path diversity. This is important because interior-node disjoint (IND) trees ensure that failures in the interior of one tree only remove a single node's data in any other. However, complete IND trees would fail to retain the primary trees clustering.

We derive each sibling from the primary tree. The process walks the tree according to a post-order traversal and performs random rotations on each internal node. Figure 3 illustrates the process for a binary tree. Starting at the bottom of the tree, the algorithm ascends to the first internal node and rotates that subtree. The rotation chooses a random child and exchanges it with the current parent. Rotations continue percolating leaves up into the tree until it rotates the root subtree.

While this pushes $\frac{numLeaves}{bf}$ leaves into the interior of the tree, it doesn't replace all interior nodes. At the same time, it is unlikely that a given leaf node will be rotated into a high position in the tree, upsetting the clustering. Our experimental results (Section 7) confirm this. Note that sibling tree construction makes no explicit effort to increase the underlying path diversity. Doing so is the subject of future work. Finally, an obvious concern is a change in the network coordinates used to plan the primary tree. While we have yet to investigate this in detail, large changes in network coordinates would require query re-deployment.

3.3 Dynamic tuple striping

As operators send tuples towards the root, they must choose a neighboring operator in one of the n trees. For dynamic tuple striping the default policy is to stripe newly created tuples in a round-robin fashion across the trees. However, when a parent becomes disconnected, the operator must choose a new destination. The challenge is to balance the competing goals of exploring the path diversity in the tree set while ensuring progress towards the query root. We explore a staged policy that leverages a simple heartbeat protocol to detect unreachable parents.

Each peer node maintains a list of live parents for all locally installed queries. Each node also maintains a set of nodes from which it expects heartbeats and a set of nodes to which it delivers heartbeats. When the node installs a query, it updates these sets based on the parents and children contained in the query operator. Heartbeats are the primary source of bandwidth overhead in Mortar.

Figure 4 illustrates the intuition behind our scheme. In general the routing policy allows tuples to choose a parent in a given tree only if it moves the tuple closer to the root. To do so, each tuple maintains a list of {tree,level} pairs that indicates the last level of each tree the tuple visited. Operators consult this list to implement the routing policy. To explain the policy we define four functions. The function $\mathrm{OL}(t)$ determines the level occupied by the local operator on tree t. The function $\mathrm{TL}(t)$ specifies the last level at which the current tuple visited tree t. Functions P and C of t indicate the parent/child of the current node in tree t.

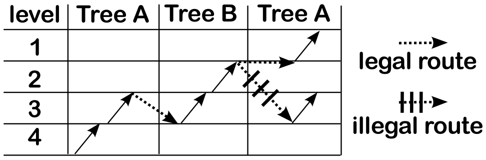

Figure 4: Multipath tuple routing up two trees. To ensure forward progress, tuples route to trees at levels less than the last level they occupied on the tree.

```
TUPLE ARRIVES ON TREE $t$
1  **Same tree:** Route to $\mathrm{P}(t)$
2  **Up*:** Route to $\mathrm{P}(x)$ such that $\mathrm{OL}(x) \leq \mathrm{TL}(t)$
3  **Flex:** Route to $\mathrm{P}(x)$ such that $\mathrm{OL}(x) \leq \mathrm{TL}(x))$
4  **Flex down:** Route to $\mathrm{C}(x)$ such that $\mathrm{OL}(x) \leq \mathrm{TL}(x)$
5  **Drop**
```

Figure 5: A staged multipath routing policy. Note that we may choose the tree with the *minimum* level given the constraint.

Figure 5 shows the decision process operators use to choose a destination node. Each successive stage allows for more routing freedom, but may also lengthen the path. The first policy attempts to use the same tree on which the tuple arrived. If this parent, $\mathrm{P}(t)$, is down, we try "up*", which tries a parent on a different tree, x, that is at least as close to the root as the current tree t. If no such tree can be found, we allow the tuples overlay path to lengthen. The "flex" policy tries to make forward progress on any tree. These first three stages prevent cycles by ensuring that tuples do not re-enter any tree at a level already visited. However, initial experiments showed that they overly constrain the available paths.

Thus, we allow a tuple to descend to the child of a tree chosen by the "flex" policy. This however, does not ensure cycle-free operation, and, when using "flex down", we increment a TTL-down field to limit the possible number of backward steps a tuple can make. When this field is greater than three, stage 4 is no longer available, and the operator drops the tuple. While not shown in Figure 5, we may choose the tree with the minimum level at each stage.

4 Time-division data partitioning

Dynamic tuple striping requires a data model that allows multipath routing. At any moment, a single query may have thousands of tuples in flight across multiple physical dataflows. For example, an aggregate operator participates in each tree (dataflow) simultaneously, and could receive a tuple from any of its children on any tree.

The insight is to allow operators to label tuples with an index that describes the particular processing window to which it belongs. Both time and tuple windows can be uniquely identified by a time range, thus the name *time-division*. With these time-division indices, operators need only inspect the index to know which tuples may be processed in the same window. The scheme is independent of operator type; the data model supports standard operators such as joins, maps, unions, filters, and aggregates. Though this work focuses on in-network aggregates because of their utility, the model also supports content-sensitive operators, those to whom specific tuples must be routed (e.g,. a join must see all of the data), by using a deterministic function that maps tuple indices to particular operator replicas.

In many respects, the time-division data model builds upon Borealis' SUnion operator, which uses tuple timestamps to maintain deterministic processing order across operator replicas [4]. However, instead of a single timestamp, it indexes tuples with validity intervals, and defines how to transform those indices as operators process input. Unlike SUnion, the data model underlies all Mortar operators. Its purpose is to allow replicas to process different parts of a stream, not to support a set of consistent, mirrored operator replicas. The data model also differs from previous approaches that parallelize operators by partitioning input data based on its content [6, 37].

The model impacts operator design in two ways. First, an operator computes across a window of raw tuples streamed from the local source, upcalling the `merge` function for each tuple. This first merge transforms raw tuples into *summary* tuples and attaches an index; we call this merging across time. Note that, if the operator is an aggregation function, then the summary tuple is a partial value. All tuples sent on the network (sent between operators) are summary tuples. An operator's second duty is to merge summary tuples from all its upstream (children) operators. We call this merging across space. The runtime, using the index attached to the summary tuple, calls the same `merge` function with summary tuples that all belong to the same processing window.

4.1 Indexing summary tuples

Operators create summary tuple indices using two timestamps [t_b,t_e] that indicates a range of time for which the summary is valid. If the window is defined in time, t_b indicates the beginning of the time window and t_e represents the end. If the window is defined across tuples, t_b indicates the arrival time of the first tuple and t_e the arrival time of the last. Thus each summary tuple represents a particular slide of the window across the raw input tuples.

Figure 6 illustrates two nodes creating summary tuples and transmitting them to the root operator. This is a time window with the range equal to the slide; operators at nodes A and B create indices for each produced summary tuple. This figure illustrates that, for time windows, we can actually use a logical index instead of a time range. The root groups arriving summary tuples with identical indices, upcalls the operator's merge function, and reports a final result R. Here, the root only receives summary tuples.

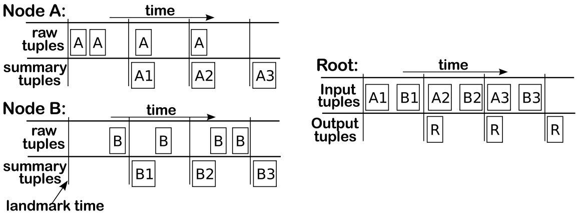

Figure 6: Two nodes creating summary tuples transmit them to the root. Each node (A and B) receives tuples only from its sensor, and labels the summary tuples with a *window index* that uniquely identifies which set of summaries can be merged.

For time windows, this scheme provides semantics identical to that of a centralized interpretation, assuming synchronized clocks. In our example the root would return identical results had it sourced the data streams directly. This scheme also provides useful semantics for tuple window processing. Instead of calculating over the globally last n received tuples (no matter the source), Mortar's query operators process the last n tuples from each source.

Summaries contain disjoint data for a given time span, and as long as the routing policy and underlying transport avoid duplicates, time-division data partitioning ensures duplicate-free operation. Nodes are now free to route tuples along any available physical query tree, even if it means the summary re-visits a physical node. Note that if a Mortar query consists of content-sensitive operators, upstream operators are constrained in their tuple routing options. In that case, source operators must agree to send the same indices to the same replica.

4.2 The time-space list

An operator may receive summary tuples in any order from upstream operators, and it must merge summary tuples with matching indices. A per-operator time-space (TS) list tracks the current set of *active* indices, indices for which the operator is actively merging arriving summary tuples. The TS list either inserts or removes (evicts) summaries. A TS list is a sorted linked list of summary tuples representing potential final values to be emitted by this operator. Here we assume that each summary tuple is *valid* for its index: [t_b,t_e].

Upon arrival the operator inserts the tuple into the TS list and merges it with existing summary tuples with overlapping indices. If indices do not overlap, we insert the tuple in order into the list. An exact index match results in the two tuples being merged (calling `merge`). The index of the resulting summary tuple is unchanged. However, when two tuples T_1 and T_2 have partially overlapping indices, the system creates a new tuple, T_3. T_3 represents the overlapping region, and its value is the result of merging T_1 and T_2. T_3's index begins at $Max(T1_{begin}, T2_{begin})$ and ends at $Min(T1_{end}, T2_{end})$. The non-overlapping regions retain their initial values and shrink their intervals to accommodate T_3. Thus, values are counted only once for any given interval of time.

4.3 Dealing with loss and delay

A common problem in distributed stream processing is telling the difference between sources that have stalled, experienced network delay, or failed. This ambiguity makes it hard for an operator to choose when to output an entry (window) in the TS list. Mortar uses dynamic timeouts to balance the competing demands of result latency and query completeness. The runtime expires entries after a timeout based on the longest delay a tuple experiences on a path to this operator. Each tuple carries an estimate of the time it has taken to reach the current operator (T.*age*), which includes the tuple's residence time at each previous operator. Operators maintain a latency estimate, called *netDist*, using an EWMA of the maximum received sample[3]. When the first tuple for a particular index arrives, the TS list sets the timeout in proportion to *netDist*$-T$.*age*. This is because, by the time tuple T arrives, T.*age* time has already passed; the most delayed tuple should already be in flight to the operator.

Stalled streams also impact our ability to ascertain summary tuple completeness, and determine how long a tuple-window summary remains valid. To remedy this, operators periodically inject "boundary" tuples when a raw input stream stalls. They are similar in spirit to the boundary tuples used in Borealis [4]. For time windows, boundary tuples are only used to update the tuple's completeness metric (a count of the number of participants); they never carry values. However, boundary tuples play an additional role when maintaining tuple windows. A tuple window only ends when the first non-boundary tuple of the next slide arrives. When a stream stalls, boundary tuples tell downstream operators to extend the previous summary tuple's index, extending the validity interval of the summary.

Finally, Mortar requires that the underlying transport protocol suppress duplicate messages, but otherwise makes few demands of it.

```
ARRIVAL OF TUPLE $T$
    $O$.t_ref ← $O$.t_ref + elapsed_time
    index ← ($O$.t_ref - $T$.age) / $O$.slide
EVICTION OF TUPLE $S$
    $O$.t_ref ← $O$.t_ref + elapsed_time
    $S$.age ← AVG($T_1$.age,...,$T_n$.age)
```

Figure 7: Syncless indexing pseudocode.

5 Reducing the impact of clock skew

The performance of distributed stream processing ultimately depends on accurate timekeeping. But assuming synchronized clocks is a well-known problem across large, distributed systems. Even with its wide-spread adoption, NTP may be mis-configured, its ports may be blocked, or it may have limited resolution on heavily loaded nodes [24]. In such cases, differences in clock skew or large clock adjustments can cause substantial differences in reported time between nodes, the relative clock offset. This offset impacts traditional completeness, the percentage of participants included in a window, but also whether the correct tuples are assigned to the window. Here we assess the impact of relative clock offset on *true* completeness, the percentage of correct tuples assigned to a window, tuple dispersion, the distribution of tuples from their true window, and result latency. Our results, presented at the end of this section, show that even mild amounts of offset impact completeness and can increase result latency by a factor of 8.

5.1 Going syncless

This section describes a simple mechanism that improves true completeness, bounds temporal dispersion, and reduces result latency. The *syncless* mechanism requires no explicit synchronization between peers. The intuition is that correct tuple processing depends on the relative passage of time experienced for each tuple. Instead of assigning each tuple a timestamp, we can leverage the *age* of each tuple, T.*age*, a field that represents the number of milliseconds since its inception. Recall from Section 4.3 that this includes operator residence time and network latency. Operators then merge tuples that are alive for similar periods of time at the same index within the time-space list, in the same summary tuple (Section 4.2).

Figure 7 shows the pseudocode used to assign incoming tuples to the correct local index. As Figure 8 illustrates, O.*t_ref* maintains a relative position in time for each operator, and begins to accumulate time on operator installation. Thus indices are purely local, indicating the set of tuples that should be merged, and may even be negative for some tuples. The evicted summary tuple, S, represents the aggregate of those tuples, and we

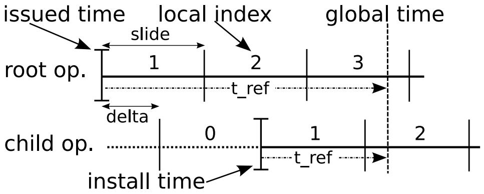

Figure 8: With the syncless mechanism, operators have different install *deltas* relative to the root node.

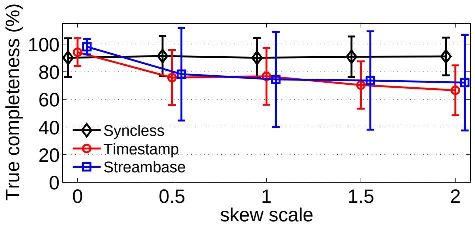

Figure 9: Total completeness for a 5-second window.

set the age of S to the average age of its constituents. This weights the tuple age towards the majority of its constituent data.

One benefit of syncless operation is that it limits tuple dispersion to a tight boundary around the correct window. To see why, first note that operators are not guaranteed to install at the same instant. This results in an install *delta*, $(t_{install} - t_{issued})\%slide$, of the query's install time ($t_{install}$) (seen in Figure 8) relative to the root's install time (t_{issued}). This shifts the local indices for an operator, changing the set of summary tuples merged. Thus, between any two operators, the interpretation of a tuple's age can differ by at most one window. That is, once merged, the tuple may be included in a summary tuple with an average age that places it outside of the true window. We correct for this effect by tracking the *age* of the query installation message, and subtracting *age* from $t_{install}$ on installation. While here the upper bound on tuple dispersion is directly proportional to tree height, dispersion with timestamps is virtually unbounded.

To determine the efficacy of the syncless mechanism we deployed the Mortar prototype over our network emulation testbed, both described in Section 7. Here 439 peers, connected over an Inet-generated network topology, have their clocks set according to a distribution of clock offset observed across Planetlab. 20% of the nodes had an offset greater than half a second, a handful in excess of 3000 seconds. We measure true completeness for an in-network sum, with a five-second window, as we scale the distribution linearly along the x-axis. Each data point is the average of 5 runs. For comparison we plot results from a commercially available centralized stream processor, StreamBase, whose tuple re-order buffer (BSort operator) we configured to hold 5k tuples.

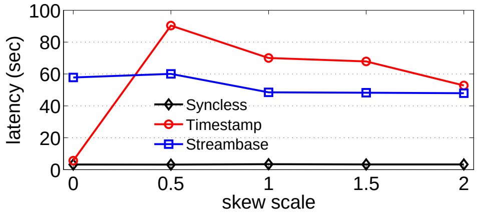

Figure 10: Result latency for a 5-second window.

Figure 9 illustrates true completeness (with std. dev.) while Figure 10 shows result latency for the same experiments. As expected, the timestamp mechanism results in a high-degree of accuracy and low result latency when there is little clock offset. However, at 50% of Planetlab skew, true completeness drops to 75% and result latency for timestamps increases by an order of magnitude. Offset also affects the results from the centralized stream processor, though latency is nearly constant because of the static buffering limit.

In contrast, syncless performance is independent of clock offset[4], and provides better completeness (averages 91%) than timestamped Mortar or the centralized stream processor at low-levels of skew. Equally important, is that result latency is constant and small (6 seconds).

Large relative clock skew and drift remain potentially problematic. The longer a tuple remains at any node, the more influence a badly skewed clock has on query accuracy. The tuple's residence time is primarily a function of the furthest leaf node in the tree set, and, from our experiments across Inet-generated topologies, is on the order of only a few seconds. Determining this penalty remains future work. However, techniques already exist for predicting the impact of clock skew on one-way network latency measurements [22], and could likely be applied here. Even with these limitations, syncless operation provides substantial benefits in the event NTP is impaired or unavailable.

6 Query persistence

This section discusses how Mortar reliably installs and removes queries across the system. As a best-effort system, Mortar makes no attempt to salvage data that was contained in an operator at the time of node failure. Instead, Mortar uses a pair-wise reconciliation protocol to re-install the same kind of operator, including its type, query-specific arguments, and position in the static primary and sibling aggregation trees, on a recovering node.

Initially, a peer installs (and removes) a query using the primary tree as the basis for an un-reliable multicast. However, because the trees are static, the message must contain the primary and sibling tree topologies. To reduce message size and lessen the impact of failed nodes, the peer breaks the tree into n components, and multicasts the query down each component in parallel. While fast, it is unreliable, and the reconciliation mechanism guarantees eventual query installation and removal.

Our protocol draws inspiration from systems such as Bayou [31], but has been streamlined for this domain. In particular, the storage layer guarantees single-writer semantics, avoiding write conflicts, and communication is structured, not random. Like those prior pair-wise reconciliation protocols, the process is eventually consistent.

6.1 Pair-wise reconciliation

Mortar manages queries in a top-down fashion, allowing children who miss install or remove commands to reconcile with parents, and vice versa. The reconciliation protocol leverages the flow of parent-to-child heartbeats in the physical query plan. Periodically, parent-child node pairs exchange summaries describing shared queries. The reconciliation protocol begins when a node receives a summary, a hash (MD5) of relevant queries ordered by name, that disagrees with its local summary. The process is identical for removal operations, but because removals cancel parent-child heartbeats, Mortar overloads tuple arrivals (child-to-parent data flow) for summary comparisons.

First, the two nodes, A and B, exchange their current set of installed queries, $I_{<node>}$, and their current set of cached query removals, $R_{<node>}$. Each node then performs the same reconciliation process. Each node computes a set of *install candidates* $IC_{<node>}$ and *removal candidates* $RC_{<node>}$.

$$IC_A = I_B - (I_B \cap I_A) - (I_B \cap R_A)$$
$$RC_A = I_A \cap R_B$$

IC_A represents the set of queries for which A has missed the installation. Additionally, node A removes all queries for which there is a matching remove in R_B. Peers use sequence numbers, issued by the object store, to determine the latest management command for a particular query name. Node B computes IC_B and RC_B similarly. At this point, $I_A == I_B$; reconciliation is complete.

The last step in this process is for the installing peer to re-connect the operator, discovering the parent/child set for each tree in the physical query plan. In this case, the peer contacts the query root, who, acting as a topology server, returns the n parent/child sets. Thus, like planning, Mortar distributes the topology service across all query roots in the system.

7 Evaluation

Mortar has taken a different approach to adaptivity than traditional DHT-based systems that create a single, dynamic overlay. The ultimate purpose of our techniques is to ensure accurate wide-scale stream processing when node sets contain failed nodes.

Our Java-based Mortar prototype implements the Mortar Stream Language and the data management, syncless, and recovery mechanisms. Each Mortar peer is event driven, leveraging Bamboo's [34] `ASyncCore` class that implements a single-threaded form (based on `SFS/libasync`) of the staged event-driven architecture(SEDA). Other advantages of this low-level integration include UdpCC, a congestion-controlled version of UDP, and their implementation of Vivaldi [12] as the source of network coordinates[5]. We use the X-Means data clustering algorithm to perform planning [30]. Beyond the usual in-network operators, the prototype supports a custom trilateration operator for our Wi-Fi location service. Last, aggregate operator results include a completeness field.

We evaluate Mortar primarily on a local-area emulation testbed using ModelNet [39]. A ModelNet emulation provides numerous benefits. First, it tests real, deployable prototypes over unmodified, commodity operating systems and network stacks. A Mortar configuration running over our local cluster requires no code changes to use ModelNet; the primary difference is that, in ModelNet, network traffic is subjected to the bandwidth, delay, and loss constraints of an arbitrary network topology. Running our experiments in this controlled environment allows direct comparison across experiments. 34 physical machines, running Linux 2.6.9 and connected with gigabit Ethernet, multiplex the Mortar peers.

Unless stated otherwise, ModelNet experiments run across an Inet-generated network topology with 34 stub nodes. We uniformly distribute 680 end nodes across those stubs, emulating small node federations. All link capacities are 100 Mbps, the stub-node latency is 1 ms, the stub-stub latency is 2 ms, the stub-transit latency is 10 ms, and the transit-transit latency is 20 ms. The longest delay between any two peers is 104 ms. Each mortar query uses four trees and a branching factor of 16.

7.1 Query installation

Ultimately, query results are only as complete as the operator installation coverage. Reconciliation should install queries across all live, reachable nodes within a node set, even when a significant fraction of the set is down. Here we use a query that sources 680 nodes, but disconnect a random node subset before installation.

Figure 11 shows both the rate and coverage of query installation. Recall that while installation is a multicast operation, it is "chunked", i.e., the installer splits the tree into separate pieces and installs them in parallel. Our experiments use 16 chunks, and with no failures, it takes less than ten seconds to install 680 nodes. We recon-

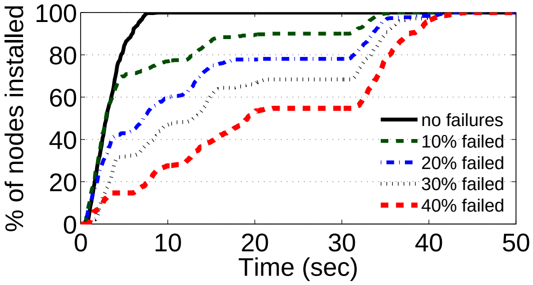

Figure 11: Query installation behavior across 680 nodes with inconsistent node sets.

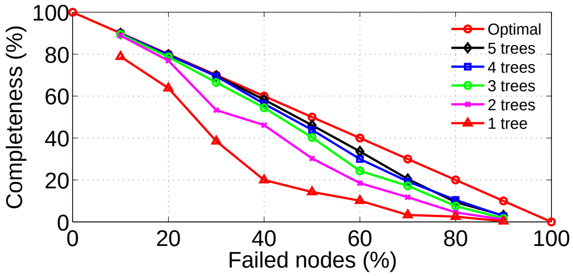

Figure 12: Coverage as a function of the number of trees.

nect all nodes after 30 seconds. Note that reconciliation runs every third heartbeat, i.e., every six seconds, and this results in the slower rate of installation when using reconciliation. However, as predicted by the simulations in Section 2.1, reconciliation installs operators on 54.5% of all nodes even when 40% of nodes are unreachable, resilience similar to that achieved by multipath routing.

7.2 Failure resilience

With the operators successfully installed, the system must now route data from source to query root, avoiding network and node failures. Here our goal is to study how Mortar responds to failures of "last mile" links. Unless mentioned otherwise, these microbenchmarks deploy a `sum` query that subscribes to a stream at each peer in the system, counting the number of peers. Mortar uses a time window with range and slide equal to one second. A sensor at each system node produces the integer value "1" every second.

7.2.1 The impact of tree set size

Increasing the tree set size improves failure resilience as additional trees add more overlay paths. Here we measure the resiliency additional sibling trees provide and discuss the overhead that comes with it.

Figure 12 plots query completeness as a function of the percentage of disconnected nodes in the system. Here each data point is the average of five runs, each run lasting three minutes. The first thing to note is that with four trees, query results reflect most live nodes, achieving perfect completeness for 10 and 20% failures. For 30% and 40% failures, Mortar's results include 98% and 94% of the remaining live nodes respectively. This is nearly identical to the results from our simulation and attest to the ability of our sibling tree construction algorithm to create overlay path diversity that approaches that of random trees. Secondly, this appears to be the point of diminishing returns, as five trees provides small additional improvements in connectivity.

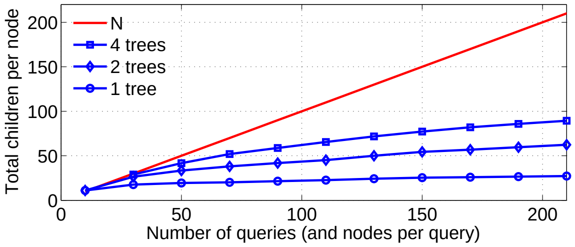

Figure 13: System scaling is directly proportional to the number of unique children at each peer. This plot illustrates sharing across sibling and primary trees as the number of queries increase.

Note that each additional tree increases background heartbeat traffic by adding to the number of unique parent-child pairs in the tree set[6]. However, the same heartbeats may be used across the trees in different queries. Figure 13 shows the number of unique children that a given node must heartbeat as a function of the number of queries in the system. Here there is a query for every peer, and that query aggregates over all other nodes. Empirically, overhead scales sub-linearly with both additional queries, and additional siblings per query. In the first case, repeated clusterings on the same coordinate set result in similar primary trees across queries. In the second case, adding a single sibling (2 trees total) roughly doubles the overhead of using a single, primary tree. However, three additional siblings (4 trees total) does not double the overhead of using two trees, but results in a 50% relative increase. This is due to our sibling construction algorithm constraining the possible children a node can have.

7.2.2 Responsiveness

A best-effort system should provide accurate answers in a timely fashion. We first explore the impact of transient "rolling" failures. These time-series experiments disconnect a percentage (10, 20, 30, and 40%) of random nodes for 60 seconds, and then reconnect them. Note that result completeness is identical to that seen in Figure 12 for four trees; here the point is to assess the impact of failure on result latency, completeness, tuple path length, and total network load, the sum of traffic across all links.

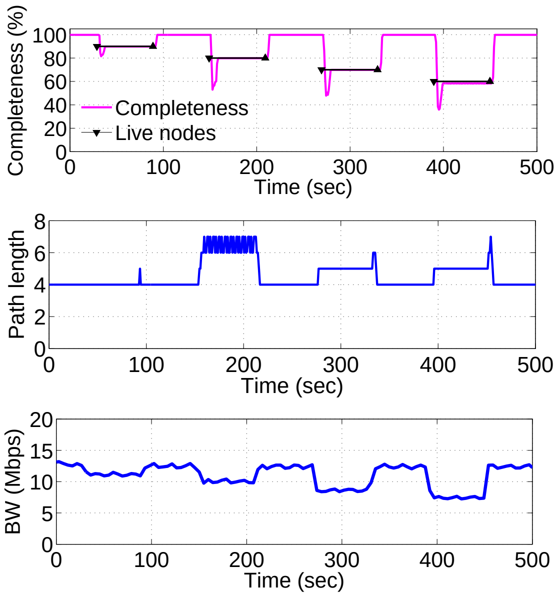

Figure 14: Accuracy and total network load for Mortar during rolling failures of 10,20,30, and 40% of 680 nodes.

Here, result latency is the time between when the result was due and when the root operator reported the value.

Figure 14 shows that Mortar responds quickly to failures, returning stable results on average 7 seconds after each failure. This is a function of our heartbeat period (2 seconds), and appears independent of the number of failures. The system captures the majority of the data with an average result latency of 4.5 seconds. Our branching factor of 16 results in a tree of height 4, which is the path length when there are no failures. Even during 40% failures, the majority of tuples can route around failures with three extra overlay hops. The steady-state network load is 12.5 Mbps (3.4 Mbps of which is heartbeat overhead). As a point of comparison, the same experiment without aggregation incurred twice the network load (26 Mbps).

Finally, while not precisely churn, a query should still be resilient to nodes cycling between reachable and unreachable states. Figure 15 shows results where we randomly disconnect 10% of the nodes. Then, every 10 seconds, we reconnect 5% of the failed nodes and fail an additional, random 5%. Mortar always reconnects all live nodes before the 10 seconds are up. Result latency, network load, and tuple path length are similar to that seen in the rolling failures experiment.

7.2.3 Comparing to a DHT-based system

We compare Mortar to SDIMS [41], an information management system built over the Pastry DHT [35]. We chose SDIMS because of considerable support from its authors, including providing us with a version that uses the latest FreePastry release (2.0_03). This was critical,

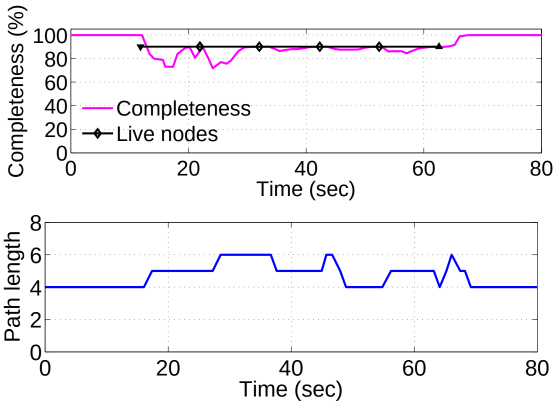

Figure 15: Query accuracy across 680 nodes on an Inet topology during 10% churn.

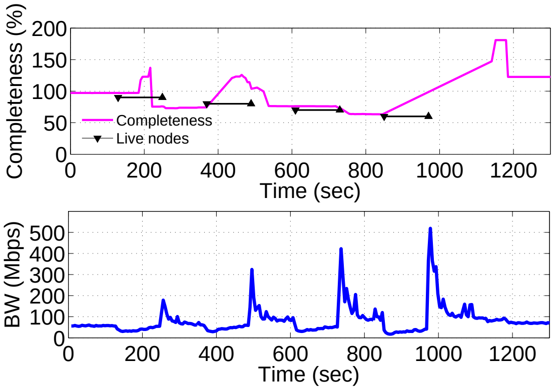

Figure 16: Query accuracy and total network load for SDIMS for 680 nodes. Though we probe five times less often, the steady-state bandwidth is five times greater for the same query.

as it provides routing consistency and explicitly tests for network disconnections[7]. Since SDIMS is a "snapshot" in-network aggregation system, we continuously issue probes to emulate a streamed result.

Figure 16 shows query results and total network load for an SDIMS experiment using 680 peers across the same topology. We fail nodes in an identical fashion, but the down time is 120 instead of 60 seconds. The SDIMS update policy ensures that only the root receives the aggregate value, the ping neighbor period is 20 seconds, the lease period is 30 seconds, leaf maintenance is 10 seconds and route maintenance is 60 seconds. SDIMS nodes publish a value every five seconds and we probe for the result every 5 seconds.

Accurate results at the beginning of the experiment soon give way to highly variable results during even low failure levels. Failures appear to generate over counting as completeness exceeds 100%, hitting almost 180% by the end of the experiment. Probe results remain in-

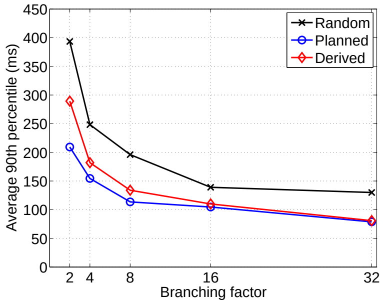

Figure 17: Interconnecting operators across Internet-like topologies. This shows the average latency to the root for the 90th percentile of nodes in the query.

accurate through the end of the experiment, even after all nodes have re-connected. Increasingly large disconnections cause bandwidth spikes as the reactive recovery mechanism engages larger numbers of peers. The steady-state bandwidth usage is 67 Mbps (9 Mbps of which is Pastry overhead); this is 5.3 times as much as Mortar, which produces results with five times the frequency. We hypothesize that the large difference in bandwidth usage is due to a lack of in-network aggregation, as nodes fail to wait before sending tuples to their parents.

As mentioned previously, SDIMS poor accuracy is likely due to its dependence on the underlying DHT for adaptation. Aggressive leaf set and route table maintenance frequencies had little effect.

7.3 Network-aware queries

This section evaluates the ability of our primary and sibling tree building algorithms (physical dataflow planner) to place data within a low-latency horizon around the root operator. Experiments use an aggregate query across 179 randomly chosen nodes over the Inet topology. Vivaldi runs for at least ten rounds before interconnecting operators. We then generate 30 random, primary(planned), and derived (sibling) trees for branching factors (bf) of 2, 4, 8, 16, and 32. For each tree we calculate the latency, across the overlay, from each operator hosting peer to the root operator. This represents the minimum amount of time for a summary tuple from that peer to reach the query root.

Figure 7.3 distills our data to make a quantitative assessment of our planning algorithm. Across each set of 30 graphs, we calculate the average 90th percentile peer-to-root latency. The amount of time the root must wait before it can have a 90% complete value is proportional to this average. First, our recursive cluster planner improves upon random by 30 to 50%. Second, our sibling tree algorithm preserves the majority of this benefit for a range of branching factors.

7.4 The Wi-Fi location service

As a proof of concept we have designed a Wi-Fi device tracking service using the local Jigsaw wireless monitoring system [9] as the source for authentic workloads. Here Wi-Fi “sniffers” create tuples for each captured 802.11a/g frame, containing the relative signal strength indicator (RSSI) measured by the receiver. At each sniffer a `select` operator filters frames for the target source MAC address. A `topk` query finds the three “loudest” frames (largest RSSI) received by the sniffers. Finally, a custom `trilat` operator takes the resulting `topK` stream and computes a coordinate position based on simple trilateration, given the coordinates of each sniffer[8].

Unfortunately the Jigsaw sniffers have limited RAM, and cannot accommodate the footprint of our JVM. Instead, we emulate the 188 Wi-Fi network sniffers across the ModelNet testbed; each Mortar peer hosts a “Wi-Fi” sensor that replays the captured frames in real time. The topology is a star with 1 ms links (2 ms one-way delay between each sniffer). Here the primary benefit of physical planning is path diversity, not result latency.

In our experiment, a user circled the four building floor, from the fourth to the first, while downloading a file to their laptop. Figure 18 plots the coordinate stream ($\times$’s); our naive scheme had trouble distinguishing floors, and we plot the points on a single plane. However, this simple query returns the L-shaped path of the user, even distinguishing hallways. Relative to a query that did not allow the TopK to aggregate (bf=188) (but still performing the distributed select), the Mortar query resulted in a 14% decrease in total network load. Without such a selective filter, traditional summary traffic statistics would yield savings similar to those seen in our microbenchmarks.

8 Related work

Mortar’s data model is related to prior work on parallelizing operators, as it allows replicas to process different portions of the same stream. For instance, Flux [37] may partition the input for a hash-join operator using the hash of the join key. Other systems may try to automatically partition the data based on observed statistical properties [6]. However, time-division data partitioning is independent of data content and operator type.

A number of wireless sensor systems employ forms of multipath tuple routing for in-network aggregates. While TAG proposed statically striping data up a DAG [21], two other wireless in-network aggregation protocols, synopsis diffusion [28] and Wildfire [5] allow dynamic multipath routes. Like Mortar, synopsis diffusion de-couples aggregation (for Mortar, merging) and tuple routing, allowing tuples to take different paths towards the root operator. While diffusion allows tuples to be multicast

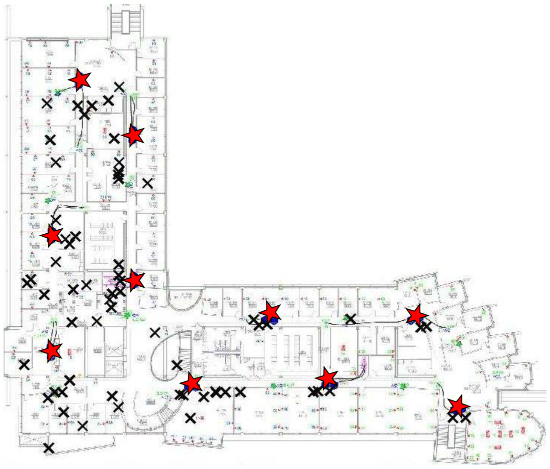

Figure 18: The position of a Wi-Fi user circling the hallways of the UCSD Computer Science department. ×'s represent query results, stars Wi-Fi sensors.

along separate paths, Mortar's data model requires the absence of duplicate summaries. What Mortar offers in its place is a straight-forward operator programming interface. This is in contrast to the special duplicate and order-insensitive operators required of both wireless routing schemes.

The role of Mortar's physical operator mapping is to interconnect operators to create a set of efficient, yet diverse routing paths. Recent work in "network-aware" operator placement tackles a similar problem: placing *unpinned* operators, those that can be mapped to any node in the network, to reduce network load [32, 1]. For example, SBONs [32] use distributed spring relaxation across a cost space combining both network latency and operator bandwidth usage. Our scheme would benefit from their insights in adapting to operator bandwidth usage.

The time management framework proposed in [38] is close in spirit to our syncless mechanism. In that model, a centralized stream processor sources external streams that may have unsynchronized clocks and experience network delays. The system continuously generates per-stream heartbeats that guarantee that no tuple arrives with a timestamp less than τ. However, determining each τ requires the construction of an nxn (n is the number of streams) matrix whose entries bound the relative clock offset between any two sources. Filling the matrix requires potentially complicated estimations of offset bounds. In contrast, Mortar's syncless mechanism ignores offset altogether, using *ages* to both order tuples and calculate the operator's timeout.

Using multiple trees for increased failure-resiliency has been explored in both overlay and network-layer routing. For example, SplitStream builds a set of interior-node disjoint (IND) trees for the multicast group to balance load and improve failure resilience. They ensure the trees are IND by leveraging how the Pastry DHT performs routing [7]. Like Mortar, SplitStream sends a separate data stripe down each tree, but the system drops stripe data when encountering failed nodes. An area of future investigation is determining dynamic tuple striping rules for multicasting across a static tree set.

Finally, Motiwala et al. recently proposed a technique, Path Splicing, to improve end-to-end connectivity at the network level [23]. In this scheme, nodes run multiple routing protocol instances to build a set of routing trees; the trees are made distinct by randomly permuting input edge weights. Like Mortar, nodes are free to send packets onto a different tree when a link fails. Their preliminary results show that five trees extracts the majority of the available path diversity, agreeing with ours. While they hypothesize whether such a scheme eliminates the need for dynamic routing in the general case, our experiments indicate that it does for the many-to-one communication patterns in our stream processing scenarios.

9 Conclusion

Mortar presents a clean-slate design for wide-scale stream processing. We find that dynamic striping across multiple static physical dataflows to be a powerful technique, allowing up to 40% of the nodes to fail before severely impacting result streams. Because time-division data partitioning logically separates stream processing and tuple routing, Mortar sidesteps the failure resilience issues that affect current data management systems built over DHT-based overlays. Finally, by reducing the dependence on clock synchronization, Mortar can accurately operate in environments where such mechanisms are mis-configured or do not exist. While it is certain that new issues will arise when deploying a query across a million nodes, Mortar is a significant step towards building a usable Internet-scale sensing system.

References

[1] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *Proc. of 30th VLDB*, September 2004.

[2] P. Bahl, R. Chandra, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. Enhancing security of corporate Wi-Fi networks using DAIR. In *Proc. of the Fourth MobiSys*, June 2006.

[3] M. Bailey, E. Cooke, F. Jahanian, N. Provos, K. Rosaen, and D. Watson. Data reduction for the scalable automated analysis of distributed darknet traffic. In *Proc. of IMC*, October 2005.

[4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *Proc. of ACM SIGMOD*, June 2005.

[5] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *Proc. of ACM SIGMOD*, June 2004.

[6] P. Bizarro, S. Babu, D. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *Proc. of 31st VLDB*, September 2005.

[7] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proc. of the 19th SOSP*, October 2003.

[8] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *Proc. of ACM SIGMOD*, 2000.

[9] Y.-C. Cheng, J. Bellardo, P. Benko, A. C. Snoeren, G. M. Voelker, and S. Savage. Jigsaw: Solving the puzzle of enterprise 802.11 analysis. In *Proc. of SIGCOMM*, September 2006.

[10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of CIDR*, January 2003.

[11] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming network-wide visibility using ubiquitous endsysem monitors. In *Proc. of USENIX Technical Conf.*, May 2006.

[12] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proc. of SIGCOMM*, August 2004.

[13] K. K. Droegemeier and Co-Authors. Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education. In *20th Conf. on Interactive Info. Processing Systems for Meteorology, Oceanography, and Hydrology*, 2004.

[14] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. Iris-Net: An architecture for a world-wide sensor web. 2(4), October/December 2003.

[15] J. M. Hellerstein, T. Condie, M. Garofalakis, B. T. Loo, P. Maniatis, T. Roscoe, and N. A. Taft. Public health for the Internet (φ). In *Proc. of CIDR*, January 2007.

[16] L. Huang, X. Nguyen, M. Garofalakis, J. Hellerstein, M. I. Joran, A. D. Josepth, and N. Taft. Communication-efficient online detection of network-wide anomalies. In *Proc. of IEEE Infocom*, May 2007.

[17] R. Huebsch, B. Chun, and J. M. Hellerstein. PIER on PlanetLab: Initial experience and open problems. Technical Report IRB-TR-03-043, Intel Corporation, November 2003.

[18] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER, September 2003.

[19] T. Johnson, S. Muthukrishnan, V. Shkapenyuk, and O. Spatscheck. A heartbeat mechanism and its application in Gigascope. In *Proc. of 31st VLDB*, September 2005.

[20] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proc. of SIGCOMM*, August 2004.

[21] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proc. of the fifth OSDI*, December 2002.

[22] S. B. Moon, P. Skelly, and D. Towsley. Estimation and removal of clock skew from network delay. In *Proc. of IEEE Infocom*, March 1999.

[23] M. Motiwala, N. Feamster, and S. Vempala. Path Splicing: Reliable connectivity with rapid recovery. In *Proc. of ACM HotNets-VI*, November 2007.

[24] S. Muir. The seven deadly sins of distributed systems. In *Proc. of WORLDS*, December 2004.

[25] R. Murty, A. Gosain, M. Tierney, A. Brody, A. Fahad, J. Bers, and M. Welsh. CitySense: A vision for an urban-scale wireless networking testbed. Technical Report TR-13-07, Harvard University, September 2007.

[26] R. N. Murty and M. Welsh. Towards a dependable architecture for Internet-scale sensing. In *Second HotDep06*, November 2006.

[27] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with seaweed. In *Proc. of VLDB*, September 2006.

[28] S. Nath, P. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. of the 2nd SenSys*, November 2004.

[29] D. Oppenheimer, B. Chun, D. A. Patterson, A. Snoeren, and A. Vahdat. Service placement in a shared wide-area platform. In *Proc. of USENIX Technical Conf.*, June 2006.

[30] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proc. 17th ICML*, 2000.

[31] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of 16th SOSP*, October 1997.

[32] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. of ICDE*, April 2006.

[33] P. Pietzuch, J. Shneidman, J. Ledlie, M. Welsh, M. Seltzer, and M. Roussopoulos. Evaluating DHT service placement in stream-based overlays. In *Proc. of IPTPS*, February 2005.

[34] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. of USENIX Technical Conf.*, June 2004.

[35] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, pages 329–350, November 2001.

[36] V. Sekar, N. Duffield, O. Spatscheck, K. van der Merwe, and H. Zhang. LADS: Large-scale automated DDoS detection system. In *Proc. of USENIX Technical Conf.*, May 2006.

[37] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly available, fault tolerant, parallel dataflows. In *Proc. of ACM SIGMOD*, June 2004.

[38] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of PODS*, June 2004.

[39] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proc. of OSDI*, December 2002.

[40] R. VanRenesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM TOCS*, 2003.

[41] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc. of SIGCOMM*, September 2004.

Notes

[1] We borrow definitions of skew (differences in clock frequency) and offset (difference in reported time) from the network measurement community [22].

[2] It is a text-based version of the “boxes and arrows” query specification approach [10, 4].

[3] $\alpha = 10\%$ worked well in practice.

[4] What variations exist are due to the random placement of offset across the nodes for each test.

[5] Experiments use 3-dimensional coordinates.

[6] In the worst case each tree adds $O(N)$ pairs globally and $O(bf)$ at involved peers. N is the node set size and bf the branching factor.

[7] Experiments with PIER [18] showed that it was badly affected by the dynamism in the Bamboo DHT’s [34] periodic recovery protocols. The PIER authors have made similar observations [17].

[8] We are not innovating here; more advanced methods exist but could use similar queries.