Reliable Storage and Querying for Collaborative Data Sharing Systems

Nicholas E. Taylor and Zachary G. Ives

Computer and Information Science Department, University of Pennsylvania
Philadelphia, PA, U.S.A.
{netaylor,zives}@cis.upenn.edu

***Abstract*— The sciences, business confederations, and medicine urgently need infrastructure for sharing data and updates among collaborators' constantly changing, heterogeneous databases. The ORCHESTRA system addresses these needs by providing data transformation and exchange capabilities across DBMSs, combined with archived storage of all database versions. ORCHESTRA adopts a peer-to-peer architecture in which individual collaborators contribute data and compute resources, but where there may be no dedicated server or compute cluster.**

We study how to take the combined resources of ORCHESTRA's autonomous nodes, as well as PCs from "cloud" services such as Amazon EC2, and provide reliable, *cooperative* storage and query processing capabilities. We guarantee reliability and correctness as in distributed or cloud DBMSs, while also supporting cross-domain deployments, replication, and transparent failover, as provided by peer-to-peer systems. Our storage and query subsystem supports dozens to hundreds of nodes across different domains, possibly including nodes on cloud services.

Our contributions include (1) a modified data partitioning substrate that combines cluster and peer-to-peer techniques, (2) an efficient implementation of replicated, reliable, versioned storage of relational data, (3) new query processing and indexing techniques over this storage layer, and (4) a mechanism for incrementally recomputing query results that ensures correct, complete, and duplicate-free results in the event of node failure during query execution. We experimentally validate query processing performance, failure detection methods, and the performance benefits of incremental recovery in a prototype implementation.

I. INTRODUCTION

There is a pressing need today in the sciences, medicine, and even business for tools that enable autonomous parties to *collaboratively* share and edit data, such as information on the genome and its functions, patient records, or component designs shared across multiple teams. Such collaborations are often characterized by diversity across groups, resulting in different data representations and even different beliefs about some data (such as competing hypotheses or diagnoses from the same observations). Data is added and annotated by different participants, and occasionally existing items are revised or corrected; all such changes may need to be propagated to others. To maintain a record across changes, different versions of the data may need to be archived. In these collaborative settings, there is often no single authority, nor global IT group, to manage the infrastructure. Hence, it may be economically or politically infeasible to create centralized services in support of data transformation, change propagation, and archival.

To address these needs, we have been developing the ORCHESTRA *collaborative data sharing system* (CDSS) [1]. Briefly, ORCHESTRA adopts a peer-to-peer architecture for data sharing, where each individual participant owns a local DBMS with its own preferred schema, makes updates over this DBMS, and periodically *publishes* updates to others. Then the participant translates others' published updates to its own schema via *schema mappings* and *imports* them. ORCHESTRA especially targets scientific data sharing applications such as those in the life sciences, where data sets are typically in the GB to 10s of GB, and changes are published periodically and primarily consist of new data insertions.

Previous work on ORCHESTRA has developed the upper layers of our system architecture: strategies and algorithms for resolving conflicts [2], and for generating the necessary queries to propagate data and updates across sites or peers [3]. Such work temporarily used a centralized DBMS to handle storage and query processing. In this paper, we complete the picture, with a highly scalable and reliable versioned storage and query processing system for ORCHESTRA, which does not require dedicated server machines. Rather, we employ the existing CDSS nodes, possibly in combination with machines leased as-needed from cloud services such as Amazon EC2.

Our goal is to provide the benefits of peer-to-peer architectures [4], [5], [6], [7], [8] (such as support for autonomous domains with no common filesystem, transparent handling of membership changes, and plug-and-play operation), *hybridized* with the benefits commonly associated with traditional parallel DBMSs and with emerging cloud data management platforms [9], [10], [11], [12] (such as efficient data partitioning, automatic failover and partial recomputation, and guarantees of complete answers). We avoid what we perceive to be the negative aspects of each architecture: the lack of completeness or consistency guarantees in peer-to-peer query systems, and requirements for shared filesystems and centralized administration in the existing cloud data management services (e.g. Google's GFS [9], Amazon's S3 [12]).

To accomplish this, we exploit the fact that our system does not need *all* of the properties provided by existing distributed substrates. Our problem space is less prone to "churn" than a traditional peer-to-peer system like a distributed hash table: we assume that membership in a CDSS, while not completely stable, consists of perhaps **dozens to hundreds** of participants at academic institutions or corporations, with **good bandwidth** and relatively **stable machines**. We support archived storage of data under a **batch-oriented** update load:

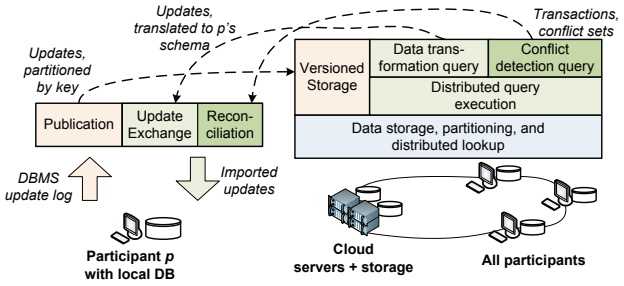


Fig. 1. Basic architectural components in the ORCHESTRA system, as a participant (peer) publishes its update logs and imports data from elsewhere. Components on the left were the focus of [2], [3], and this paper focuses on the components shown on the right.

in a CDSS, users first make updates only to their local storage, and they occasionally *publish* a log of these updates (which are primarily insertions of new data items) to the CDSS. Then they perform an *import* (transforming and importing others' newly published data to their local replica). Only in this step is information actually shared across users, and it is then that conflict resolution is performed. Hence, we **do not need special support for global consistency**, such as distributed locking or version vectors, at the distributed storage level.

We address these needs through a custom data partitioning and storage layer, as well as a new distributed query processor. We develop novel techniques for ensuring versioning, consistency, and failure recovery in order to guarantee complete answers. Our specific contributions are as follows:

- Modifications to the standard data partitioning techniques used in distributed hash tables [4], customizing them to a more stable environment, and providing greater transparency of operation to the layers above.
- A distributed, replicated, *versioned* relational storage scheme that ensures that queries see a consistent, complete snapshot of the data.
- Mechanisms for detecting node failures and either completely restarting or *incrementally* recomputing the query, while ensuring the correct answer set is returned.
- Experiments, using standard benchmarks for OLAP and schema mapping tasks, across local and cloud computing nodes, validating our methods under different network settings and in the presence of failures.

We implement and evaluate our techniques within the ORCHESTRA collaborative data sharing system. However, the techniques are broadly applicable across a variety of emerging data management applications, such as distributed version control, data exchange, and data warehousing.

Section II presents the ORCHESTRA architecture, and Section III details our modified data distribution substrate. Section IV describes our storage and indexing layer, upon which we build the fault-tolerant distributed query engine presented in Section V. Section VI validates our techniques through experimental analysis. We describe related work in Section VII, and conclude and discuss future work in Section VIII.

II. SYSTEM ARCHITECTURE AND REQUIREMENTS

Figure 1 shows ORCHESTRA's architecture, and sketches the dataflow involved in its main operations. Each participant (illustrated on the left) operates a local DBMS with a possibly unique schema, and uses this DBMS to pose queries and make updates. ORCHESTRA is invoked when the participant has a stable data instance it wishes to "synchronize" with the world: this involves *publishing* updates from the local DBMS log to versioned storage, and *importing* updates from elsewhere. The import operation consists of *update exchange* [3] and *reconciliation* [2]. Update exchange finds updates satisfying a local participant's filtering criteria and, based on the schema mappings, executes SQL queries that convert data into the participant's local schema. Reconciliation finds sets of conflicts, among both updates and the transactions they comprise, by executing SQL queries over the versioned storage system.

To this point, our work has focused on the left half of the figure: the logic needed to create and use the SQL queries supporting update exchange and reconciliation, and the modules to "hook" into the DBMS to obtain update logs. In this paper, we focus on the right half of the diagram: how to implement distributed, versioned storage and distributed query execution. We are particularly concerned with performance in support of update exchange (data transformation) queries, which are more complex than the conflict detection queries, and by far the main bottleneck in performance [2], [3]. We also develop capabilities in the query execution layer to support mapping and OLAP-style queries directly over the distributed, versioned data. Data is primarily stored and replicated among the various participants' nodes. However, as greater resources, particularly in terms of CPU, are required, participants may purchase cycles on a cloud computing service capable of running arbitrary code, such as Amazon's EC2 (considered in this paper) or Microsoft's Azure.

In the remainder of this section, we explain the unique requirements of ORCHESTRA and why they require new solutions beyond the existing state of the art. In subsequent sections, we describe our actual solutions.

A. Data Storage, Partitioning, and Distributed Lookup

As discussed previously, we assume that the participants number in the dozens to hundreds, are usually connected, and have enough storage capacity to maintain a log of all data versions. Our target domain differs from conventional P2P systems where connectivity is highly unstable. We only expect low "churn" (nodes joining and leaving the system) rates, perhaps as participants go down for maintenance or are replaced with new machines. We expect failures to be infrequent enough that keeping a few replicas of every data item is sufficient. We avoid single points of failure, as we want the service to remain available at all times, even if some nodes go down for maintenance.

In a distributed implementation of a CDSS, we need a means of (1) partitioning the stored data (such that it is distributed reasonably uniformly across the nodes), (2) ensuring efficient re-partitioning when nodes join and leave, (3) supporting distributed query computation, and (4) supporting background replication. There are two main schemes for doing this in a distributed system: distributing data *page-by-page* in a *distributed filesystem*, and then using a sort- or hash-based

scheme to combine and process the data; and distributing data *tuple-by-tuple* according to a key, and using a distributed hash scheme to route messages to nodes in a network. Google's MapReduce and GFS, as well as Hadoop and HDFS, use the former model. Distributed hash tables (DHTs) [4], [7], [8] and directory-based schemes use the latter.

Distributed filesystems suffer from several drawbacks as the basis of a query engine. First, they require a single administrative domain and (at least in current implementations like HDFS) a single coordinator node (the NameNode), which introduces a single point of failure. Moreover, they actually use two different distribution models: base data is partitioned on a per-page basis, then all multi-pass query operations (joins, aggregation, nesting) must be executed through a MapReduce scheme that partitions the data on keys (via sorting or hashing).

We instead adopt a tuple-by-tuple hash-based distribution scheme for routing messages: this is commonly referred to as a *content addressable overlay network* and is exemplified by the DHT. Our goal is to provide good performance and to tolerate nodes joining or failing, but we do not require scalability to millions of nodes as with the DHT. In Section III we adapt some of the key ideas of the DHT in order to accomplish this.

B. Versioned Storage

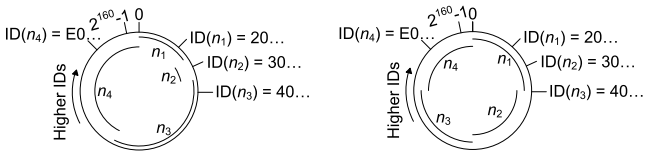
Each time a participant in ORCHESTRA publishes its updates, we create a new *version* of that participant's update log (stored as tables). This also results in a new version of the global state published to the CDSS. Now, when a participant in ORCHESTRA imports data via update exchange and reconciliation, it expects to receive a *consistent*, *complete* set of answers according to some version of that global state. We support this with a storage scheme (described in Section IV) that tracks state across versions, and manages replication and failover when node membership changes, such that queries receive a "snapshot" of the data according to a version. We optimize for the fact that most published updates will be new data rather than revisions to current data.

When data is stored in a traditional content-addressable network, background replication methods ensure that all data *eventually* is replicated, and gets placed where it belongs when a node fails — but if the set of participants is changing then data may temporarily be missed during query processing. Furthermore, such systems also require the data assigned to each key to be immutable. Similarly, existing distributed filesystems like GFS and HDFS assume data is within immutable files, and they are additionally restricted to a single administrative domain.

Hence our versioned storage scheme must provide bookkeeping than a traditional distributed hash table, but offers more autonomy and flexibility than a distributed filesystem. In Section III we describe our customized data storage, partitioning, and distributed lookup layer.

C. Query Processing Layer

As is further discussed in Section VII, a number of existing query processing systems, including PIER [5] and Seaweed [6], have employed DHTs to perform large-scale, "best-effort" query processing of streaming data. In essence, the DHT is treated like a very large parallel DBMS, where hashing is used as the basis of intra-operator parallelism. Immutable data can be stored at every peer, accessed by hashing its index key. Operations like joins can be performed by hashing both relations according to their join key, co-locating the relations to be joined at the same node. Such work has two shortcomings for our context: multiple data versions are not supported, and their "best-effort" consistency model in the presence of failures or node membership changes is insufficient.



(a) Pastry-style range allocation (b) Balanced range allocation

Fig. 2. Range allocation schemes

Our goal is not only to support efficient distributed computation of query answers, but also to detect and recover from node failure. We emphasize that this is different from recovery in a transactional sense: here our goal is to **compensate for missing answers in a query**, ideally without redoing the entire query from scratch (whereas transactional recovery typically does involve aborting and recomputing from the beginning). Failure recovery in query answering requires us (in Section V) to develop techniques to track the processing of query state, all the way from the initial versioned index and storage layer, through the various query operators, to the final output.

Furthermore, we develop techniques for *incrementally* recomputing only those results that a failed node was responsible for producing. Given that every operator in the query plan may be executed in parallel across all nodes, the failure of a single node affects intermediate state at all levels of the plan. Our goal is to restart the query only over the affected portions of the data, and yet to ensure that the query does not produce duplicate or incorrect answers.

III. HASHING-BASED SUBSTRATE

Any scalable substrate for data storage in a peer-to-peer setting needs to adopt techniques for (1) data partitioning, (2) data retrieval, and (3) handling node membership changes, including failures. We describe how our custom hashing-based storage layer addresses these issues, in a way that is fully decentralized and supports multiple administrative domains.

A. Data Partitioning

Like most content-addressable overlay networks, we adopt a hash-based system for data placement. Similar to previous well-known *distributed hash tables* (DHTs) such as Pastry [4], we use as our key space 160-bit unsigned integers, matching the output of the SHA-1 cryptographic hash function.

It is convenient to visualize the key space as a ring of values, starting at 0 and increase clockwise until they get to $(2^{160}-1)$ and then overflow back to 0. Figure 2 shows two examples of this ring that we will discuss in more detail.

Most overlay networks assign a position in the ring to each node according to a SHA-1 hash of the node's IP address (forming a DHT ID). Values are placed at nodes according to the relationship with their hash keys. In Chord, keys are placed

at the node whose hashed IP address lies *ahead* of them on the ring; in Pastry the keys are placed at the node with *nearest* hash value. The Pastry scheme is visualized in Figure 2(a). Both of these approaches can determine the range a node "owns," given its ID and the IDs of its neighbors. These schemes are optimized for settings with large numbers of nodes, and assume the nodes will be more or less uniformly distributed across the ring. Each node maintains information about the position of a limited number of its neighbors, as it has a routing table with a number of entries logarithmic in the membership of the DHT. When there are only dozens or hundreds of nodes, we often see highly nonuniform distributions of values among the peers. Indeed, in the figure, nodes n_3 and n_4 are together responsible for more than $\frac{3}{4}$ of the key space, while node n_2 is only responsible for $\frac{1}{16}$ of it.

Our substrate adopts Pastry's routing approach for large numbers of peers (with an expanded routing table, as discussed later in this section). However, for smaller numbers of peers, we support an alternative solution that provides more uniform data distribution (which we use for the experiments in this paper). We divide the key space into evenly sized sequential ranges, one for each node, and assign the ranges in order to the nodes, sorted by their hash ID. Such an assignment for the same network we examined for Pastry-style routing is shown in Figure 2(b); it distributes the key space, and therefore the data, uniformly among the nodes. In principle, we could also use many virtual nodes at each physical node to better distribute the key space. However, it is advantageous to assign a single contiguous key range to each node; in addition to reducing the size of the routing table, this improves data retrieval performance, as discussed in Section IV. In response to node arrival or failure, we redistribute the ranges over the new node set. We consider the implications of this when we describe node arrival and departure later in this section.

B. Data Retrieval

As mentioned above, a traditional DHT node maintains a routing table with only a limited number of entries (typically logarithmic in the number of nodes). This reduces the amount of state required, enabling greater scale-up, but requires multiple hops to route data. Recent peer-to-peer research has shown [13] that storing a complete routing table (describing all other nodes) at each node provides superior performance for up to thousands of nodes, since it provides single-hop communication in exchange for a small amount of state; we therefore adopt this approach. Our system requires a reliable, message-based networking layer connection with flow control. We found experimentally that, for scaling at least to one hundred nodes, maintaining a direct TCP connection to each node was feasible. With the use of modern non-blocking I/O, a single thread easily supports hundreds or thousands of open connections. For larger networks, a UDP-based approach could be developed to avoid the overhead of maintaining TCP's in-order delivery guarantees, as all of the techniques in this paper are independent of message ordering.

C. Node Arrival and Departure

Traditional DHTs deal with node arrival and departure through background replication. Each data item is replicated at some number of nodes (known as the *replication factor*). In Pastry, for example, for a replication factor r, each item is replicated at $\lfloor \frac{r}{2} \rfloor$ nodes clockwise from the node that owns it, and the same number counterclockwise from it, leading to r total copies. In the ring of Figure 2(a), if $r = 3$, each data item that is owned by node n_1 will be replicated to n_4 and n_2 as well. When a node joins, background replication slowly brings all data items that a node owns to it, as they must be stored at one of its neighbors. If a node leaves, each of its neighbors already has a copy of the data that it owned, so they are ready to respond to queries for data stored at the departed node.

This approach makes an implicit assumption that all of the state at the nodes is stored in the DHT, and therefore that any node that has a copy of a particular data item can handle requests for it. If a node joins or fails, certain requests will suddenly be re-routed to different nodes, which are assumed to provide identical behavior (and hence do not get notified of this change). This does not work in the case of a distributed query processor, where in addition to persistent stored data there may be distributed "soft state" that is local to a query and is not replicated; this includes operator state, such as the intermediate results stored in a join operator or an aggregator. If data for a particular range is suddenly rerouted from one node to another, tuples might never "meet up" with other tuples they should join with, or data for a single aggregate group may be split across multiple nodes, causing incorrect results.

To solve this problem, our system works on *snapshots* of the routing table. When a participant initiates a distributed computation, it sends out a snapshot of its current routing table, which all nodes will use in processing this request. Therefore, if a new node joins in mid-execution, it does not participate in the current computation (otherwise it may be missing important state from messages prior to its arrival). If a node fails, the query processor can detect what data was owned by the failed node, and thus can reprocess this state (this is discussed in Section V-D).

Our system must still handle replication of base data, which is done in a manner very similar to that of Pastry; each data point is replicated at $\lfloor \frac{r}{2} \rfloor$ nodes clockwise and counterclockwise from the node that owns it. This ensures that data can survive multiple node failures, and that in the event of a node failure, the nodes that take over for a failed node have copies of the base data for the sections of the ring they are newly responsible for. Unlike in Pastry, a single node arrival or departure will cause all the ranges in the range to change slightly; this causes a membership change to be more expensive, but we are assuming reasonable bandwidth and less frequent failures. With smaller numbers of fairly reliable nodes, the performance benefits of uniform distribution likely outweigh the costs of extra shipping.

Currently we only replicate data as it is inserted into the DHT. This has been sufficient for the development and experimental analysis of our system, since we inserted data

before any node failures, and failed few enough nodes that data was never lost. For completeness we plan to implement the Bloom filter-based background replication approach of the Pastry-based PAST storage system [14], which can be directly applied to our context.

IV. Versioned Data Storage

Recall from our earlier discussion that ORCHESTRA supports a batched publish/import cycle, where each participant stores its own updates in the CDSS, disjoint from all others. There is no need for traditional concurrency control mechanisms, as conflicts among concurrent updates are resolved during the import stage (via reconciliation) by the participant.

However, there is indeed a notion of global consistency. We assign a logical timestamp (*epoch*) that advances after each batch of updates is published by a peer. When a participant performs an import or poses a distributed query, it is with respect to the data available at the specific epoch in which the import starts. The participant should receive the effects of *all* state published up to that epoch, and no state published thereafter (until its next import). The current epoch can be determined through a simple "gossip" protocol and does not require a single point of failure.

Of course, in order to support queries over versioned data, we must develop a storage and access layer capable of managing such data. There are several key challenges here:

- Between database versions, we want to efficiently reuse storage for data values that have not changed.
- We must track which tuples belong to the desired version of a database. Such metadata should be co-located with the data in a way that minimizes the need for communication during query operation.
- Each tuple must be uniquely identifiable using a *tuple identifier* that includes its version. Yet, for efficiency of computation, we must partition data along a set of key attributes (as with a clustered index). It must be possible to convert from the tuple ID to the tuple key, so that a tuple can be retrieved by its ID; therefore a tuple's hash key must be derived from (possibly a subset of) the attributes in its ID.

We maintain all versions of the database in a log-like structure across the participants: instead of replacing a tuple, we simply update our records to include the new version rather than the old version, which remains in storage. Disk space is rarely a constraint today, and the benefits of full versioning, such as support for historical queries, typically outweigh the drawbacks.

Each node, therefore, may contain many versions of each tuple. If the set of nodes is in flux, nodes may come and go between when a tuple is inserted or updated and when it is used in a query; therefore, a node may not have the correct version of a particular tuple. We assume that background replication is sufficient to ensure that each tuple exists somewhere in the system, but that it may not exist where the standard content-addressable networking scheme can find it. The key to our approach is a hierarchical structure that maps from a point in time to the collection of tuple IDs present in a relation at that time. This collection is used during processing to detect which tuples are missing or stale, and must therefore be retrieved from another node in the system.

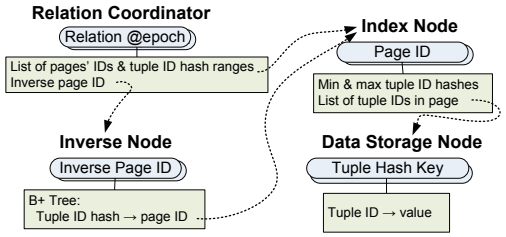


Fig. 3. Storage scheme to ensure version consistency and efficient retrieval. Rounded rectangles indicate the key used to contact each node (whose state is indicated with squared rectangles).

Figure 3 shows the main data structures used to ensure consistency. All data structures are replicated using the underlying network substrate, so failure of any node will cause all of its functionality to be assumed transparently by one or more neighboring nodes. We distribute all tuples are according to a hashing scheme. Relations are divided into versioned *pages*, each of which represents a partition over the space of possible tuple keys' hash values. Tuples assigned to the same page will be likely be co-located on a single node, or span two nodes in the worst case. As an optimization, we place the index node entry at the same node as the tuples it references, by storing the index page at the middle of the range of tuple keys it encompasses. This is why the network substrate, as discussed in Section III-A, assigns a large, contiguous region in the key space to each node; it means that the vast majority of tuple keys are never sent over the network. If each node is responsible for many smaller ranges, this is no longer the case, and performance suffers.

When requesting a given relation at a given epoch, the storage system hashes these values to get the address of a *relation coordinator*, who has a list of the pages in the relation at that epoch. The system uses this list, which contains the hash ID associated with each page, to find the *index nodes* that contain these pages. From the index nodes, the system retrieves the tuple IDs belonging to the relation at the epoch, which are used to retrieve the full versions of all the tuples in the relation from the *data storage node*. Recall that as the pages are colocated with most of the tuples they reference, typically a single node serves as both the index node and the data storage node for an entire page, reducing network traffic and improving performance.

Our scheme is designed to efficiently support small changes to tables. Modifying a tuple in a relation requires us to look up the page holding the old version of the tuple using an inverse node, modify that page to include the ID of the new tuple, and write out that modified page as the new index page for the region of the table surrounding the updated tuple. The entire contents of the new tuple must also be written out to the network. The system then creates a new version record linking to the updated index page, and all of the unaffected pages from the previous version.

We were initially inspired by filesystem i-nodes, the CFS filesystem [15], and log-structured filesystems, where for ap-

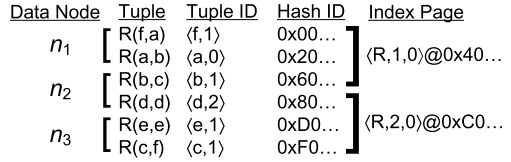


Fig. 4. Final state for example. Data is partitioned across nodes by the key (the first attribute), which is a subset of the Tuple ID. Redundant copies of replicated data are not shown. The left brackets indicate which nodes a tuple is stored on, while the right brackets indicate which index page a tuple's ID is on.

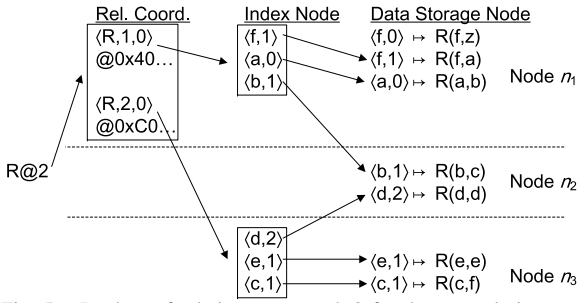


Fig. 5. Lookup of relation R at epoch 2 for the example instance.

pend operations and small changes, the page-level data in a large file mostly remains unchanged. Such schemes all make use of a versioned system for tracking the contents of a file, which greatly resembles our index nodes. A direct translation of the CFS approach would create a small number of index node entries with tuple IDs arranged in the order they appear in the table; retrieval of the referenced tuples would require communication with many data storage nodes. We instead use a slightly higher number of entries representing partitions of the tuple space; each such page can be retrieved from one or at most a few data storage nodes.

A key property we adopt from CFS is that, once there is enough information to begin a request (i.e. the current epoch has been determined), it is always clear what data *should* be present in the distributed storage layer. Therefore stale data will never be retrieved. If expected data is not found at the node that should own it, this is likely due to network churn. The request can either be retried after background replication has moved state around, or the system can proactively try to retrieve the missing state from other nearby nodes.

Example 4.1: Suppose we have three participants, each storing a partition of a simple, one-table database, $R(\underline{x}, y)$, where x is the key and y is a non-key attribute. Node n_1 is responsible for the range [0x00…,0x55…], n_2 for [0x55…,0xAA…], and n_3 [0xAA…,0x00…]. In the first epoch (epoch 0), a participant inserts the tuples $R(a, b)$ and $R(f, z)$. In epoch 1, someone inserts $R(b, c)$, $R(e, e)$, and $R(c, f)$ while also changing $R(f, z)$ to $R(f, a)$. In epoch 2, someone inserts $R(d, d)$. The final state of the system is shown in Figure 4. The Tuple ID is the key attribute of a tuple and the epoch in which it was last modified, e.g., $\langle f, 1\rangle$ for $R(f, a)$. The index page ID consists of the relation name, the epoch in which it was last modified, and a unique identifier for that relation and epoch, which is 0 for both example pages here. It also includes the hash ID where the index page is stored.

Pseudocode for performing a lookup appears as Algorithm 1. Retrieval starts at the relation coordinator for the requested epoch, from which a list of index nodes can be obtained. It sends a scan request to each index node, along with the sargable predicate. The index nodes apply the sargable predicates to the list of tuples for each index page, and requests that the matching tuples be retrieved. This operation is highly parallelizable; the only operation done at a single node is the sending of the scan requests, which is very fast.

Example 4.2: Figure 5 shows how the lookup procedure works for our example instance. First, the lookup request from n_2 for relation R at epoch 2 is hashed to find the node (in this case, n_1) that is the relation coordinator for the relation at the desired epoch. The data stored there contains the list of index pages that contain the tuple IDs for that version of the relation. The request to scan those pages is sent to the index nodes that contain the contents of the pages, in this case n_1 and n_3. Those index nodes then send requests on to the data storage nodes that contain the full tuples (stored as a mapping from Tuple ID to full tuple) to scan the desired tuples given their IDs. The data storage nodes then retrieve the desired tuples and return them to the requester (not shown). Note that only two of the six Tuple IDs were actually sent over the network, due to the colocation of index pages and tuple data.

As mentioned before, this approach avoids any possibility of seeing stale data due to replication lag. Suppose that, for some reason, n_1 had not yet received a copy of the record for R at epoch 2. It would search other nodes nearby in the system until it found a copy before proceeding. Similarly, if n_1 had not yet received the data $\langle f, 1\rangle$, it would never simply return the data for $\langle f, 0\rangle$; it knows that data is stale because it does not appear in the index page. It would instead try to retrieve the full tuple for $\langle f, 1\rangle$ from the network before proceeding.

Algorithm 1 Retrieve$(R, e, f(\bar{k}))$. **Input**: R relation, e epoch, $f(\bar{k})$ filter function over key $\bar{k}$. **Output**: Matching tuples $t \in R$ satisfying $f(\bar{k})$.

```
1: relCoord ← h(⟨R, e⟩)
2: Contact Relation Coordinator at relCoord, retrieve pageIDList
3: for page ∈ pageIDList do
4:     Ask Index node at h(⟨e, (page.max+page.min)/2⟩) to scan page page
5:     Index node retrieves page contents Tuples
6:     Index node filters Tuples with f(k̄) → fTuples
7:     for t ∈ fTuples do
8:         Index node requests that Data Storage node at h(t.key) scan the tuple t
9:         Data Storage node sends t to node that requested scan, bypassing the Index node and Relation Coordinator
10:    end for
11: end for
```

V. Reliable Query Execution

As in prior peer-to-peer query engines [5], [6], we adopt a dataflow ("push") style of distributed query processing. The operators at each node either receive data directly from a local scan of persistent storage, or receive tuples as they arrive from other nodes in system. All data is ultimately collected at the *query initiator* node, which may do final processing, such as the last stage of aggregation, or a final sort.

The main difference in ORCHESTRA is that we are concerned with computing the exact (i.e., correct and complete) answer set over finite relations, rather than doing best-effort computations over unbounded streams. Moreover, we support scientific data sharing confederations with hundreds of peers, not hundreds of thousands or millions of peers.

A. Architecture for Performance and Failure Detection

Several aspects of our query processing architecture are enabled by our custom hash-based substrate, versus an existing DHT like those used in PIER (Chord [8] or a hybrid between Gnutella and Chord [16]) or Seaweed (Pastry). However, we additionally develop several techniques at the query execution level that are vital for performance and correctness.

First, for **failure detection** and **efficiency**, the query processor benefits from the fact our substrate uses TCP to manage connections between machines. A "downstream" node almost immediately detects when an "upstream" node has failed, because the TCP connection drops. It also automatically provides flow control in the event of a congested network. In contrast, the DHTs in prior work tend to do little or no flow control, and they rely on occasional pings to eventually detect failures. (Of course, we could have alternatively used UDP, and implemented fast failure detection and flow control via periodic handshaking.)

Second, for **failure recovery**, the query processor is given direct information about the state of the routing tables. A *snapshot* of the routing tables is taken by the query initiator as it invokes the query. This snapshot is disseminated along with the query plan to all nodes, in order to ensure absolute consistency of the routing tables. If one or more nodes fail in the middle of execution, the difference in the routing tables is reported back to the query initiator, such that it can incrementally recompute *only* the lost portion of the query state (Section V-D).

Third, for **performance**, the query processor *batches* tuples into blocks by destination, compressing them (using lightweight Zip-based compression) and marshalling them in a format that exploits their commonalities. This makes query processing much more efficient than if it were built over a DHT with many smaller messages, and reduces CPU and bandwidth use.

Finally, for **correctness**, each tuple is annotated with information about which source nodes supplied the data from which the tuple was derived. This is used to prevent duplicate answers when recovering from a failed node.

B. Query Execution

A query plan consists of the operators listed in Table I. The query is "driven" by some combination of the leaf-level scan operators described in the table — each is novel to our system, as it exploits the specific versioned indexing scheme used in our storage system. Such operators typically are run concurrently across all of the nodes in the system — each operating on a data partition stored at those nodes.

From there, the retrieved data may be passed locally through a series of pipelined operators, such as joins or function evaluation. This continues until the next operator is either a **ship** operator or a **rehash** operator. The ship operator sends the data it receives to the query initiator. The rehash operator partitions its input tuples by their hash IDs in the networking substrate and sends them to other nodes in the system. Rehashing is commonly used to enable joins or aggregation, when a relation needs to be re-partitioned on a join or grouping key. The rehash operator routes tuples to a destination node by first hashing the key using the SHA-1 hash function, then consulting the snapshot of the *query routing table* described previously.

Covering index scan retrieves data directly from the index nodes, if only key attributes are required, bypassing the data storage nodes.
Distributed scan executes at both index nodes and data storage nodes, similar to in Algorithm 1, with the data storage nodes received filtered collections of tuple IDs that pass a sargable predicate. The tuples from each index page are stored nearby on disk, and are retrieved in a single pass through the hash ID range for that page. Instead of being sent back to the query initiator, the resulting tuples are pushed through the query plan.
Select implements selection on intermediate results.
Project is the standard projection operator.
Join is a *pipelined hash join* [17].
Aggregate is a a blocking, hash-based grouping operator, which supports re-aggregation of partially aggregated intermediate results.
Ship sends the tuples it receives to the query initiator.
Rehash partitions its input among the system nodes by hashing on some subset of the tuples' attributes.
Compute-function performs scalar function evaluation, such as arithmetic or string concatenation.

TABLE I
OPERATORS IN THE ORCHESTRA QUERY ENGINE

Each operator sends an *end-of-stream notification* to its parent operator when it finishes executing. Scans can easily detect when they are done, and most other operators simply propagate an end-of-stream notification downstream after they receive it (perhaps first performing some final computation to produce results, as in a aggregate operator). However, detecting end-of-stream with the rehash operator is slightly tricky: it cannot complete until it has acknowledgment from all downstream nodes that they have received all of the data it sent. Once all operators have encountered the end of the stream, the query is complete.

Example 5.1: Continuing our example, a given node may initiate the following query:

```
SELECT x, MIN(z)  FROM R, S
WHERE R.y = S.y   GROUP BY x
```

ORCHESTRA can use the execution plan in Figure 6 (annotated with the tuples at each stage in the plan, in italics). Each node joins tuples from relations R and S on attribute y, and then groups the results by x. Before this computation can begin, tuples must be redistributed so items that join are on the same node. We *rehash* tuples from R on the y attribute: now both R and S are partitioned according to their join key. Next we execute the join, producing intermediate relation RS, partitioned on the y attribute. The group-by operation requires one additional rehash, this time on its grouping attribute x. Each node aggregates its values, then all nodes ship their results to Node 1, the query initiator. When the scans complete, they propagate end-of-stream to the rehash operators. The rehash operators confirm that all of their sent

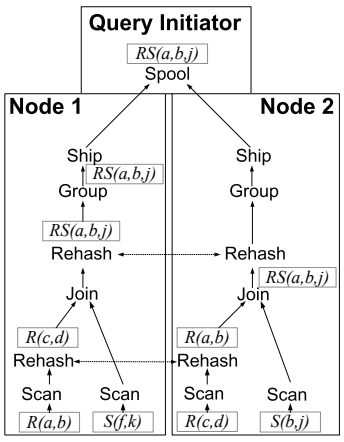


Fig. 6. Distributed query plan for running example.

data has been received, then propagate the end-of-stream. The process continues until the Group operators are encountered. When these operators receive end-of-stream, they first output their final aggregate values, before propagating the end-of stream, which gets forwarded to the query initiator.

C. Handling Node Membership Changes

The major challenge of reliable query processing is how to handle changes to the node set. Recall from Section III that the query initiator takes a *snapshot* of the routing tables in the system during query initiation. It disseminates this snapshot along with the query plan so all machines will use a consistent assignment between hash values and nodes.

Node arrival. Suppose a node joins the system in the midst of execution. In a DHT, such a change immediately affects the routing of the system — and begins forwarding messages to the new node, which may not have participated in any prior computation. In principle, one might develop special protocols by which the new node would be "brought up to speed" by its neighbors. However, this becomes quite complex when multiple nodes join at different times. Instead, we let the query complete on its initial set of nodes, and only make use of the new node when a fresh query (with a new routing table snapshot) is invoked. This approach provides simplicity and avoids expensive synchronization.

Node departure/failure. Our use of TCP connections between nodes is generally adequate to detect a total node failure (we assume complete failure rather than incorrect operation) or network partition. If a sending node (and query operator) drops its connection before sending an end-of-stream message, or a receiving node drops its connection before query completion, then this represents a failure. Additionally, the system performs periodic ping operations in the background to detect a "hung" machine. Clearly in this case, continuing query computation will result in missing or possibly incorrect answers. This leads us to the problem of recomputation, described in the next subsection.

D. Recovery from Failure

Our system supports two forms of recovery from failure. One option, upon detecting a node failure, is to terminate and and *restart* any in-process queries. Assuming low failure rates, we will ultimately get the answers this way. This approach is straightforward to implement in ORCHESTRA, since we can detect which queries are still in-flight — in contrast to systems like PIER or Seaweed.

When failures are more common, as in longer-lived queries running on large numbers of nodes, better performance might be obtained by performing *incremental* recomputation, where we only repeat the computations affected by the failed node, using a different node that has data replicated from the failed one. The key challenge here is that simply recomputing will likely result in the creation of some number of duplicate tuples — which in turn will either lead to duplicate answers or (in many cases) to incorrect aggregate results.

After a failure, any derived state in the system that originated from the failed nodes is likely to be inconsistent, due to propagation and computation delays. We can re-invoke the computation from the failed nodes and then remove duplicate answers, or instead we can remove all state derived from the failed nodes' data before performing the recomputation. We adopt the latter approach due to the difficulty of detecting which tuples are duplicates. As was hinted at previously, this means we must track which intermediate and final results are derived from data processed at one of the failed nodes. We tag each tuple in the system with the set of nodes that have processed it (or any tuple used to create it), and maintain these sets of nodes as the tuples propagate their way through the operator graph. As we validate experimentally, this can be done with minimal overhead.

We divide incremental recomputation into four stages.

Determine change in assignment of ranges to nodes. When a node or set of nodes fail, other nodes "inherit" a portion of the hash key space from failed nodes. The query initiator computes a new routing table from the original one, assigning the ranges owned by the failed nodes to remaining ones. If the failed nodes' data is available on more than one replica, the initiator will evenly divide among them the task of recomputing the missing answers.

Drop all intermediate results dependent on data from the failed nodes. To prevent duplicate answers, we scan the internal state of all operators and discard any tuples that are tagged as having passed through a failed node (we term these *tainted* tuples). It is critical that any state *not* dependent on the failed nodes remains available. This is easy to accomplish with join operator state. For aggregate operators, we partition each group into sub-groups that summarize the effects of all of the tuples for each possible set of contributing nodes, and drop the sub-groups for failed nodes. While the number of subgroups is exponential in the number of rehashes (for n nodes and m rehashes, $\sum_{k=1}^{m+1} \binom{n}{k}$), this number is typically small; critically, it does not depend on the number of input tuples. Tuples that are in flight between operators (or crossing the network) must also be filtered in this way.

Restart leaf-level operations for the failed nodes' hash key space ranges. We restart leaf-level operations such as tablescans, re-producing any data that would have originated at the failed nodes. As the data propagate through the system, they will be re-processed against the data from other nodes,

generating all join and grouping results dependent on them.

Re-create data that was sent to the failed nodes' hash key space ranges. Additionally, any data that was sent *to* a failed node was either lost when the node failed or has become tainted by passing through the node and will therefore be discarded. Now all data that was to have been sent to the failed nodes must be retransmitted. If an operator maintains an in-memory snapshot of all data necessary to re-produce its answers (as with a pipelined hash join) this is relatively efficient. For more costly operations such as tablescans, we add a *cache* of their output data, at the downstream *rehash* or *ship* operator. It is easy to detect which of the reproduced tuples would have been sent to a failed node by consulting the query's original routing table.

Perhaps the most difficult task in recovery is avoiding race conditions that lead to subtly incorrect query results. We have chosen to divide computation into *phases* corresponding to the initial execution, followed by successive incremental recovery invocations. Each tuple gets tagged with a phase. As each stateful operator processes a recovery message, it purges tainted data and increments its phase counter. All tuples it (re)produces are in this new phase. This allows the system to differentiate between old, in-flight data from a failed node and new, recomputed results from recovery.

VI. Experimental Evaluation

We briefly describe our implementation, which has been under development for more than two years.

Query Engine. Our execution engine is implemented in approximately 50,000 lines of Java. It uses BerkeleyDB Java Edition 3.3.69 for persistent storage of data. We conducted most experiments on a 16-node cluster of dual-core 2.4GHz Xeon machines with 4GB RAM running Fedora 10, connected by Gigabit Ethernet. To study performance at scale, we used up to 100 2GHz dual core nodes from Amazon's EC2 cloud computing service.

Query Optimizer. The focus of this paper is on the distributed execution engine of ORCHESTRA, but we briefly describe its optimizer. It currently handles single-block SQL queries, including function evaluation and grouping. It adopts the Volcano [18] transformational model, using top-down enumeration of plans with memoization, and employing branch-and-bound pruning to discard alternative query plans when their cost exceeds the cost of a known query plan. Our optimizer considers bushy as well as linear query plans. It relies on information (previously computed and stored) about machine CPU and disk performance, as well as pairwise bandwidth. The optimizer estimates costs by assuming that each horizontally partitioned relation will be evenly distributed by the storage layer across all nodes. It then estimates the cost of a subplan by considering the cost at the slowest node or link that must be used at each stage — in a sense estimating the worst-case expected completion time of each operation.

A. Workload

Queries that are generated from schema mappings, as in data exchange and collaborative data sharing systems, are primarily select-project-join queries that vary from domain to domain, and are seldom publicly available. A recent benchmark suite, STBenchmark [19], has been proposed to create synthetic data exchange schema mappings along a variety of dimensions. We ran the STBenchmark instance and mapping generator with the default parameters, but with the nesting depth set to zero to produce relational data. We varied the size of each generated relation from 100K to 1.6M tuples (the maximum the ToXGene generator would produce due to memory constraints). Except for one field, all STBenchmark tables are wide relations containing many 25-character variable length strings (which are not necessarily representative of typical data exchange settings). Nonetheless, we selected a representative subset of the STBenchmark mapping scenarios to study: (1) **Copy**, which retrieves an entire 7-attribute relation, (2) **Select**, which retrieves the tuples from a 6-attribute relation that satisfy a simple integer inequality predicate, (3) **Join**, which combines a 7-, a 5-, and a 9-attribute relation by joining them on two attributes, (4) **Concatenate**, which retrieves a 6-attribute relation, concatenates three of those attributes together, and returns the result along with the remaining three attributes, and (5) **Correspondence**, which retrieves a 7-attribute relation and uses a correspondence table to add an integer-valued ID based on two of the input attributes to the result. The last query used a Skolem function (ID generator) in the output, which we replaced with a value correspondence table, as would likely be used in practice.

To add diversity and scale to our data and queries, we also experimented with the standard TPC-H OLAP benchmark: (1) it scales to a variety of sizes, enabling us to consider dataset scalability, (2) it contains a diverse set of queries, enabling us to identify different performance factors, (3) it is a well-understood and standard benchmark for comparison. We used the standard TPC-H data generator to create source data at several scale factors, and we selected the TPC-H queries meeting the single-SQL-block requirement of our optimizer. We distributed the 8 TPC-H tables by partitioning on their key attribute (first key attribute, if more than one attribute was present). Two of the tables, *Nation* and *Region*, were small enough that we replicated them at each node; together they take up less than 3KB on disk. We use TPC-H queries 1, 3, 5, 6, and 10, and measure running time to completion of the full query. Queries 1 and 6 are aggregation queries over the *Lineitem* table; Q1 performs a distributed aggregation followed by re-aggregation at the query coordinator, while Q6 only performs an aggregation at the coordinator. Queries 3, 5, and 10 are 3-way, 6-way, and 4-way joins, respectively, followed by aggregation.

All measurements were taken after results converged to a stable range of values; this is done to ensure warm caches and to avoid invoking the Java JIT compiler, which otherwise adds a large amount of noise. We present the average of five runs, and show 95% confidence intervals for all data points.

B. Performance in the Local Area

We first study the performance of our engine over our cluster's local network (running at the full Gigabit speed), to see how the architecture scales.

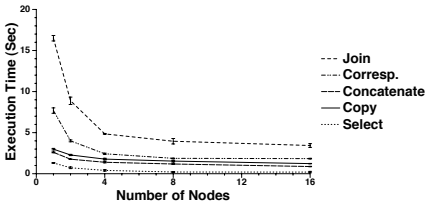


Fig. 7. Running time: STBenchmark, 800K tuples/relation, 1-16 nodes.

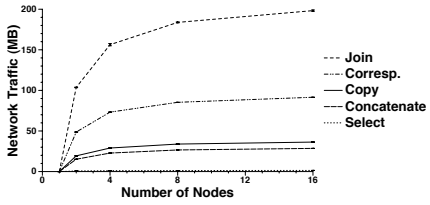


Fig. 8. Network traffic: STBenchmark, 800K tuples/relation, 1-16 nodes.

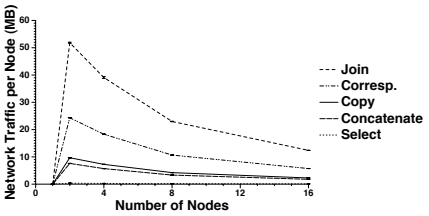


Fig. 9. Per-node network traffic: STBenchmark, 800K tuples/relation, 1-16 nodes.

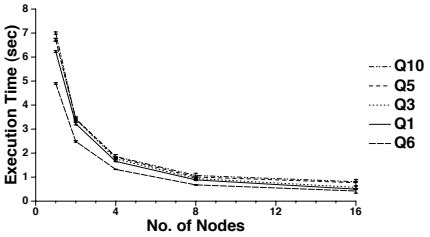


Fig. 10. Running time: TPC-H Scale Factor 0.5, 1-16 nodes.

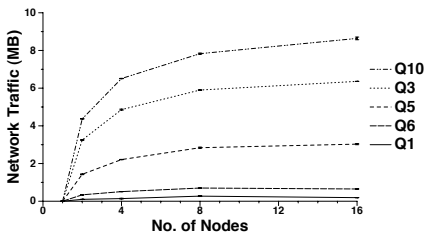


Fig. 11. Network traffic: TPC-H Scale Factor 0.5, 1-16 nodes.

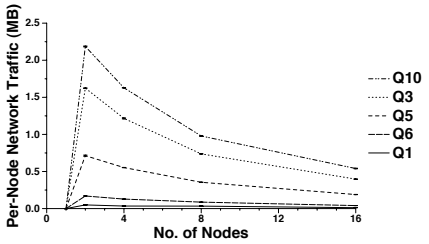


Fig. 12. Per-node network traffic: TPC-H scale factor 0.5, 1-16 nodes.

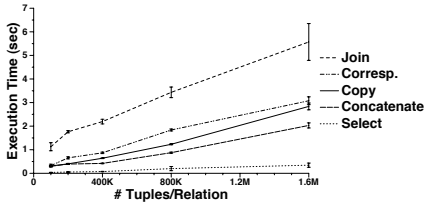


Fig. 13. Running time vs. data size, STBenchmark, 8 nodes.

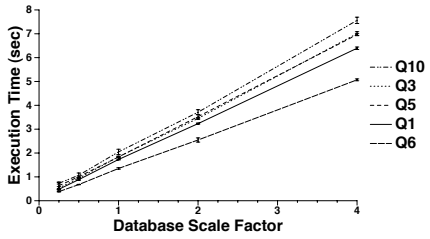


Fig. 14. Running time vs. data size, TPC-H, 8 nodes.

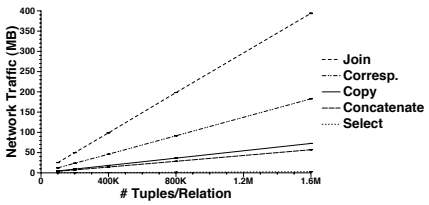


Fig. 15. Network traffic vs. data size, STBenchmark, 8 nodes.

Scaling Nodes. Figure 7 shows execution times for STBenchmark (at 800,000 tuples/relation) for 1 to 16 physical nodes, while Figure 10 shows times for TPC-H queries over the 500MB data set (scale factor 0.5). Note that results for STBenchmark are directly above the corresponding results for TPC-H to emphasize that the trends are very similar. Ideally, the running times would be halved each time we double the number of nodes. Our results come very close to matching this expectation for all of the TPC-H queries and about half of the STBenchmark queries. In the other STBenchmark queries (in particular Copy), so much data is returned (because the tuples consist of many long strings), that collecting the results at the query initiator becomes a bottleneck. With 16 nodes, all but 0.1 sec of the Copy query is spent transmitting and receiving the results. We conducted separate experiments to verify that performance is mostly limited by network bandwidth, with some additional performance degradation due to the unmarshaling and storage at the query initiator. All queries continue to show some performance improvement as the number of processing nodes increases.

Figures 8 and 11 show the total network traffic while executing these queries, and Figures 9 and 12 show the per-node traffic. As expected, the network traffic increases as we scale up the number of nodes, but not dramatically so, and the per-node traffic (after rising significantly when we move from single-node computation to distributed operation) continues to decrease as nodes are added to the system.

Scaling Data Set Size. We next consider the effects of scaling the data. Figure 13 shows execution times for STBenchmark on the 16-node cluster for 100K to 1.6M tuples/relation, and Figure 14 shows the same for the TPC-H queries over the 8-node cluster while varying the data size from 250MB to 4GB (scale factors 0.25 to 4). Figures 15 and Figure 16 show total network traffic for the same scenarios. Execution times and network traffic for all queries scale approximately linearly in the size of the data, as one would expect since there are only foreign-key joins and the data is fairly evenly distributed. We conclude that our system scales well on a LAN, and move on to consider other network settings.

C. Performance over a Simulated Wide Area Network

We next consider possible variations on Internet connectivity among compute nodes. We made use of the traffic shaping and network emulation features built into recent versions of Linux to simulate various parameter changes. Specifically, we used NetEm to delay outgoing packets, simulating a higher latency network, and we used the HTB queue discipline to simulate a lower bandwidth network. Here we focus on the TPC-H benchmark, since STBenchmark, due to its large strings, becomes increasingly bandwidth-constrained at the query initiator, and since we feel its data is actually less representative than TPC-H's.

Limited Bandwidth Settings. Our experimental results, shown in Figure 17, demonstrate that while performance suffers in very low-bandwidth connections, execution times

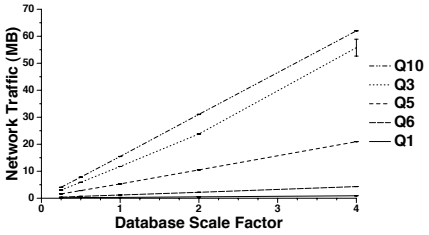


Fig. 16. Network traffic vs. data size, TPC-H, 8 nodes.

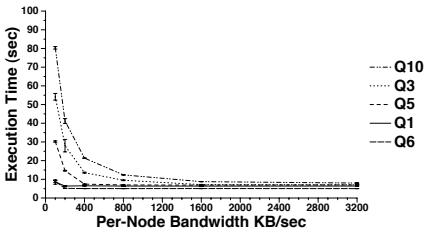


Fig. 17. Running time vs. per-node bandwidth, 8 nodes, TPC-H scale factor 4.

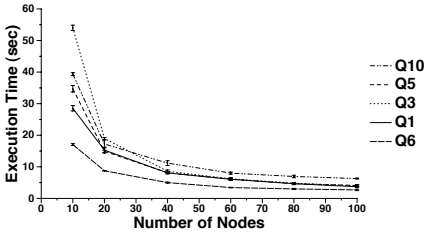


Fig. 18. Larger-scale performance on EC2, TPC-H scale factor 10.

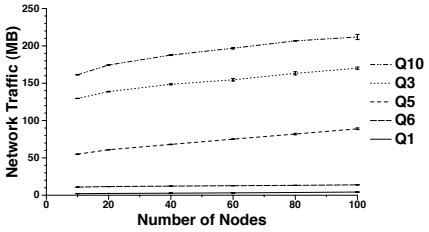


Fig. 19. Total traffic on EC2, TPC-H scale factor 10.

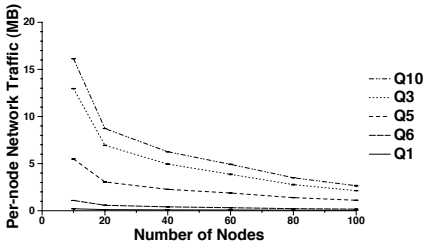


Fig. 20. Per-node traffic on EC2, TPC-H scale factor 10.

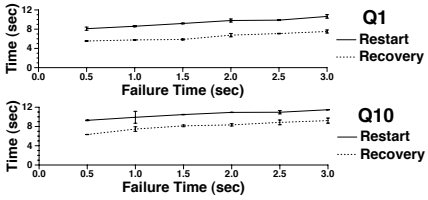


Fig. 21. Running times for Q1 and Q10 with a failure with and without incremental recovery, 8 nodes, TPC-H scale factor 2.

are degraded but reasonable for the bandwidths likely to be available between academic, institutional, or corporate users ($>$ 400 kB/sec). Queries 1 and 6, which perform no rehash operations and therefore send much less data over the network, are less impacted than queries 3, 5, and 10, which join multiple relations and rehash data while doing so.

Higher Latency Settings. We omit a full presentation of our latency experiments due to space constraints. Realistic latencies (up to 200ms) had little impact on query performance.

D. Scalability to Larger Numbers of Nodes

Since we have a limited number of local machines in our cluster, we next tried several alternatives to scale to higher numbers. Our initial efforts were with the PlanetLab network testbed — but disappointingly, we found that most nodes here were severely underpowered and overloaded, and disk- and memory-intensive tasks like ours were constantly thrashing, resulting in inconsistent and uninformative results.

Instead, we leased virtual nodes from Amazon's EC2 service — something we envision ORCHESTRA's user base doing as needed. Amazon has data centers geographically distributed across the world, so round-trip times are short and bandwidth is high. We used EC2's "large" instances with 7.5GB RAM, and a virtualized dual-core 2GHz Opteron CPU. We show settings with only EC2 nodes to make the execution time results simpler to understand, although we performed additional experiments showing similar results using a mixture of local and EC2 nodes. We experimented with the TPC-H scenario, as performance on STBenchmark at the data sizes we could generate was either too fast to be measured reliably or dominated by the cost of collecting the results.

We varied the number of total participants in the setting from 10 to 100, using TPC-H scale factor 10 (10GB data). Network traffic results, shown in Figures 19 and 20, are similar to the results shown in Figures 11 and 12 for smaller numbers of nodes. Execution times are shown in Figure 18. As before, increasing the number of nodes leads to a dramatic decrease in execution time. This experiment validates the scalability of our system to large numbers of nodes.

E. Failure and Recomputation

Finally, we study recovery when a node fails or becomes unreachable. One option is to abort the query and restart it over the remaining nodes. The other is to use the remaining nodes to recompute the "lost" results. Our experiments used 8 nodes and TPC-H scale factor 2.

Incremental Recomputation vs. Total Restart. To explore the trade-offs between incremental recomputation versus full restart, we first ran a series of experiments using Q1 (a selection and aggregation query) and Q10 (which performs three joins followed by an aggregation), chosen to represent the two classes of TPC queries we studied. We started each query and at varying points after the start of the query (before it finished) we caused one of the nodes to fail. To avoid giving incremental recomputation an unfair advantage, we recompute using the same routing tables (which spreads the range of the failed node evenly over the nodes holding its replicated data). Figure 21 shows performance results for Q1 and Q10. In both cases, incremental recovery outperforms aborting and restarting by approximately 20%, validating the approach. Execution is slow for both techniques (compared to no failure) due to the cache misses inherent when a new node takes over a portion of the substrate key space.

Overhead of Incremental Recomputation. Incremental recomputation requires more data to be stored and sent over the network (to track the provenance of intermediate results), and requires that all intermediate results be kept around until the end of the query. Clearly, if this adds significant overhead to an average query, it may actually be preferable to restart after nodes fail. We measured the overhead of incremental recovery support on the TPC-H queries, which we briefly summarize due to space constraints. As expected, recovery support slightly increased execution time: queries ran from 2%-7% slower. Network traffic increased by negligible amounts, at

most 2% (for Q10). In our view, this overhead is low enough to make it worthwhile if there is a reasonable expectation of node failure — particularly for long-running queries where the cost of restart may be high. Such an expectation goes up as more nodes join (and query running times go down, reducing the overall amount of overhead). Also, if query performance is limited by available network bandwidth, incremental recovery becomes almost free due to the low network overhead, and restarting becomes more expensive.

VII. Related Work

Distributed hash table-based query processors have largely targeted the domain of Internet-scale network monitoring, where nodes located throughout the Internet each process large amounts of typically streaming data. The PIER system [5] developed implementations of the pipelined hash join and Bloomjoin over a DHT, as well as schemes for computing aggregation over a tree-like structure of nodes. Seaweed [6] focuses on distributed aggregation, including proactive computation of aggregates, and latency-based cost estimation. In both of these systems, the focus is on throughput and best-effort query processing using many peers operating on large amounts of data; completeness and consistency are not essential. Our target domain is more controlled and smaller — with certain parameters closer to distributed DBMSs — but also has storage, consistency, and completeness requirements.

Reliable query processing is a topic of study dating back at least to IBM's R* [20] and perhaps best known commercially as Tandem NonStop SQL [21]. However, their consistency model and definition of reliability differ from ours. In existing work, the problem is detecting a failed machine in a local cluster and possibly aborting and restarting a query. Our goal is to incrementally recompute "missing" answers where possible, in order to complete query computation. Also, our consistency model is somewhat simpler because we do not consider transactions, and relations are only updated by their "owners." Recent work on cloud data services, such as [10], [11] seeks to develop reliable, batch-oriented, DBMS-like capabilities over Hadoop and immutable files stored in HDFS. Sinfonia [22] seeks to develop failure-tolerant "mini-transactions" to support distributed state management in a cluster.

VIII. Conclusions and Future Work

This paper has shown how to provide a reliable peer-to-peer storage and query execution engine for a CDSS. This involves a richer networking substrate, novel differential indexing schemes to guarantee the correct versions of all tuples are used during processing, and a query evaluator that is carefully matched to this substrate. We developed techniques for handling failures through incremental or full recomputation, and showed the trade-offs between these approaches.

There are a number of directions in which we would like to extend this work. One is to make use of materialized views, perhaps arising from the cached results of previous queries, to improve execution performance, though as in a centralized database the cost of freshening and using a view may outweigh its benefit. Another promising avenue is to implement automatic load-balancing by adjusting the routing table, to compensate for unequal network bandwidth or available machine resources. Finally and most importantly, now that the system is stable and fully functional, we plan to integrate it as a component of the ORCHESTRA system, realizing the truly peer-to-peer nature of a CDSS.

Acknowledgments

This work was funded in part by NSF grants IIS-0477972, IIS-0713267, IIS-0513778, and CNS-0721541. We thank the Penn Database Group, especially TJ Green, Greg Karvounarakis, and Svilen Mihaylov, and our anonymous reviewers for their feedback. We are also grateful to the authors of STBenchmark for their technical assistance.

References

[1] Z. G. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira, "The ORCHESTRA collaborative data sharing system," *SIGMOD Rec.*, 2008.

[2] N. E. Taylor and Z. G. Ives, "Reconciling while tolerating disagreement in collaborative data sharing," in *SIGMOD*, 2006.

[3] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen, "Update exchange with mappings and provenance," in *VLDB*, 2007, amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

[4] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Middleware*, 2001.

[5] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi, "The architecture of PIER: an Internet-scale query processor." in *CIDR*, 2005.

[6] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron, "Delay aware querying with Seaweed," in *VLDB*, 2006.

[7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *SIGCOMM*, 2001.

[8] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *SIGCOMM*, 2001.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *SOSP*, 2003.

[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008.

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, 2008.

[12] "Amazon Simple Storage Service (Amazon S3)," 2008, aws.amazon.com/s3.

[13] A. Gupta, B. Liskov, and R. Rodrigues, "Efficient routing for peer-to-peer overlays," in *NSDI*, 2004.

[14] P. Druschel and A. I. T. Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *HotOS*, 2001, pp. 75–80.

[15] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *SOSP*, 2001.

[16] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica, "Enhancing p2p file-sharing with an internet-scale query processor," in *VLDB*, 2004.

[17] L. Raschid and S. Y. W. Su, "A parallel processing strategy for evaluating recursive queries," in *VLDB*, 1986.

[18] G. Graefe and W. J. McKenna, "The Volcano optimizer generator: Extensibility and efficient search," in *ICDE*, 1993.

[19] B. Alexe, W. C. Tan, and Y. Velegrakis, "STBenchmark: towards a benchmark for mapping systems," *PVLDB*, vol. 1, no. 1, pp. 230–244, 2008.

[20] B. G. Lindsay, L. M. Haas, C. Mohan, P. F. Wilms, and R. A. Yost, "Computation and communication in R*: a distributed database manager," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, 1984.

[21] Tandem Database Group, "NonStop SQL, a distributed, high-performance, high-availability implementation of SQL," HP Labs, Tech. Rep., April 1987, report TR-87.4.

[22] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, and C. T. Karamanolis, "Sinfonia: a new paradigm for building scalable distributed systems," in *SOSP*, 2007.