Cache-and-Query for Wide Area Sensor Databases

Amol Deshpande[†,*] Suman Nath[‡,*] Phillip B. Gibbons[*] Srinivasan Seshan[‡,*]

[*]Intel Research Pittsburgh [†]U.C. Berkeley [‡]Carnegie Mellon University

ABSTRACT

Webcams, microphones, pressure gauges and other sensors provide exciting new opportunities for querying and monitoring the physical world. In this paper we focus on querying *wide area sensor databases*, containing (XML) data derived from sensors spread over tens to thousands of miles. We present the first scalable system for executing XPATH queries on such databases. The system maintains the logical view of the data as a single XML document, while physically the data is fragmented across any number of host nodes. For scalability, sensor data is stored close to the sensors, but can be cached elsewhere as dictated by the queries (auto-tuning). Our design enables *self-starting distributed queries* that jump directly to the lowest common ancestor of the query result, dramatically reducing query response times. We present a novel *query-evaluate-gather* technique (using XSLT) for detecting (1) which data in a local database fragment is part of the query result, and (2) how to gather the missing parts. We define partitioning and cache invariants that ensure that even *partial matches* on cached data are exploited and that correct answers are returned, despite our dynamic query-driven caching. Experimental results demonstrate that our techniques dramatically increase query throughputs and decrease query response times in wide area sensor databases.

1. INTRODUCTION

From webcams to smart dust, pervasive sensors are becoming a reality, providing opportunities for new sensor-based services. Consider for example a Parking Space Finder service for locating available parking spaces near a user's destination. The driver enters into a PDA (or a car navigation system) her destination and her criteria for desirable parking spaces — e.g., within two blocks of her destination, at least a four hour meter, minimum price, etc. She gets back directions to an available parking space satisfying her criteria. If the space is taken before she arrives, the directions are automatically updated to head to a new parking space.

What is required to create such a Parking Space Finder service for a large metropolitan area? First, sensors are needed that can determine whether or not a parking space is available. We envision a large collection of webcams overlooking parking spaces, pressure sensors on the spots themselves, or some combination of both. Second, a sensor database is needed, which stores the current availability information along with static attributes of the parking spaces such as the price and hours of their meters. Finally, a web-accessible query processing system is needed, to provide answers to high-level user queries.

Given the large number of sensors spread over a wide area, and the desire to support high query volumes, what is a good sensor database system architecture? Our group has developed a wide area sensor database system as part of the Internet-scale Resource-Intensive Sensor Network services (IrisNet) project, with the following architectural design.

- For high update and query throughputs, the sensor database is partitioned across multiple sites operating in parallel, and the queries are directed to the sites containing the answers. We assume that sites are Internet-connected (powered) PCs. A *site manager* runs on each site.
- Communication with the sensors is via *sensor proxies* [26], which collect nearby sensor data, process/filter it to extract the desired data (e.g., parking space availability information), and send update queries to the site that owns the data (and possibly other sites). As in [26], we assume that sensor proxies run on Internet-connected (powered) PCs.[1] A large collection of sensor proxies operating in parallel ensure that the system scales to a larger number of sensors. Moreover, a potentially high volume of data (webcam feeds) is reduced at the sensor proxy to a much smaller volume of data (e.g., availability information) sent to site managers.
- To make posing queries easy for users, IrisNet supports a standard data representation (XML), a standard high-level query language (XPATH), and a logical view of its distributed database as a single centralized database (*data transparency*)[2]. XML is used to accommodate a heterogeneous and evolving set of data types, aggregate fields, etc., best captured by self-describing tags. Moreover, each sensor takes readings from a geographic location, so it is natural to organize sensor data into a geographic/political-boundary hierarchy (as depicted in Figure 1) — again, a good match for XML. We use an off-the-shelf XML database system, and interact with the database using standard APIs (XPATH or XSLT for querying, and SixDML or XUpdate for updates), in order to take advantage of future improvements in XML database technology.

[1]For example, webcams are typically attached to PCs that can host a sensor proxy. For battery-powered motes, the sensor proxy is the powered base station communicating with the motes [26].

[2]There is one exception to our data transparency: user queries can specify a degree of tolerance for answers based on stale (cached) data, as discussed in Section 4.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Submitted and accepted to SIGMOD '03 San Diego, California USA
Copyright 2003 ACM ...$5.00.

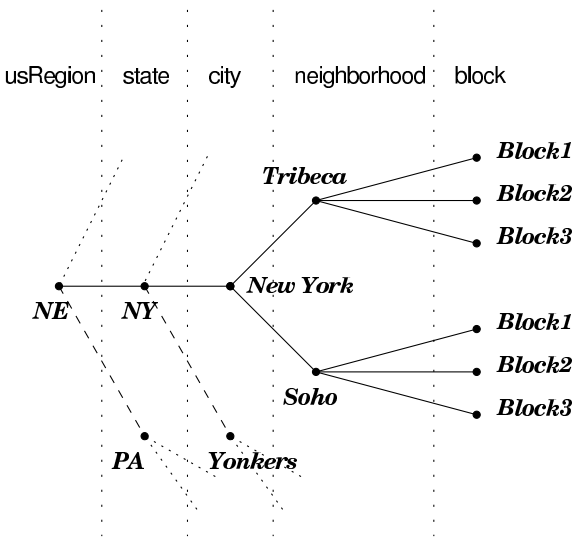

Figure 1: Logical Site Hierarchy

Although we have used Parking Space Finder as a motivating application, we envision IrisNet as a general platform for wide area sensor services. For example, we are currently working with a team of oceanographers to deploy IrisNet along the Oregon coastline. Moreover, we believe that future wide area sensor databases (not just IrisNet) will build upon a similar design, in order to meet the desired performance requirements.[3] Thus providing techniques for correctly and efficiently processing queries on such databases is crucial to the widespread use of wide area sensor systems. However, as outlined below, there are a number of challenges to providing fast execution of XPATH queries on such a system; this paper is the first to address these challenges.

In this paper, we present techniques for query processing and caching in wide area sensor databases. The contributions of this paper are:

- We present the first scalable system for executing XPATH queries on wide area sensor databases. The system uses a logical hierarchy of sites dictated by the XML document; these logical sites are mapped to a smaller collection of physical sites, as dictated by the system administrator or the queries themselves.
- We give a technique for *self-starting distributed queries*, which jump directly to the lowest common ancestor (LCA) of the query result, by using DNS-style site names extracted from the query itself. A key feature is that although a query can be posed anywhere in the Internet, no global information is needed to produce the LCA site name for the query.
- We show how general XPATH queries can be evaluated on a single XML document when the document itself is fragmented across machines, and the fragmentation is constantly changing. We propose a novel *query-evaluate-gather (QEG)* technique for detecting (1) which data in the database fragment at a site is part of the query result, and (2) how to gather the missing parts. To our knowledge, no effective solution to this problem was known prior to our work.
- Our QEG technique uses XSLT programs, and we show how these programs can be generated directly from the XPATH query, without accessing the database. Moreover, we show how to avoid the overheads of query-time compilation of XSLT programs by generating them (almost) already compiled.
- We present a scheme for caching query results at sites as dictated by the queries (auto-tuning). Owned data and cached data are stored in the same site database, with different tags, unifying the query processing at a site. Moreover, we support *query-based consistency*, in which each user query may specify its tolerance for using stale (cached) data to quickly answer the query.
- We define partitioning and cache invariants supporting *partial match caching*, which ensures that even partial matches on cached data can be exploited and that correct answers are returned, despite our dynamic query-driven caching. Our approach is based on leveraging the data residing in the site database, which differs from previous approaches based on leveraging a collection of views.
- We present experimental results demonstrating the effectiveness of our techniques in dramatically increasing update and query throughputs and decreasing query response times in wide area sensor databases.

[3]For example, an object-relational wide area sensor database would face many of the same issues and would greatly benefit from the techniques presented in this paper (see Section 6).

```
/usRegion[@id='NE']/state[@id='NY']/city[@id='New York']
       /neighborhood[@id='Soho' OR @id='Tribeca']
       /block[@id='1']/parkingSpace[available='yes']
```

Figure 2: An XPATH query requesting all available parking spaces in *Soho block 1* or *Tribeca block 1*.

```
<usRegion @id='NE'>
  <state @id='NY'>
    <city @id='New York'>
      <neighborhood @id='Soho'>
        <block @id='1'>
          <parkingSpace @id='1'>
            <available>yes</available>
          </parkingSpace>
          <parkingSpace @id='2'>
            <available>no</available>
          </parkingSpace>
        </block>
      </neighborhood>
    </city>
  </state>
</usRegion>
```

Figure 3: An XML fragment at the New York site.

The remainder of this paper is organized as follows. Section 2 describes the query processing challenges. Section 3 presents our basic query processing and caching techniques. Then in Section 4, we describe extensions to more general XPATH queries, cache consistency, and ownership migration. Experimental results are in Section 5. Section 6 describes related work, and conclusions are in Section 7.

2. QUERY PROCESSING CHALLENGES

In this section, we consider a core problem in query processing in wide area sensor databases, and demonstrate the challenges in solving this problem.

We would like to evaluate an XPATH query on a single XML document that has been fragmented across multiple sites. Consider

the example query in Figure 2 and the document fragment (at the New York site) in Figure 3. The query asks for all available parking spaces in two adjacent blocks of Soho and Tribeca. If this query is posed to the New York site, parking space 1 in Soho block 1 will be returned. The challenge is in determining whether this is the entire answer. In particular, are there other parking spaces in block 1 of Soho? Moreover, no parking spaces were returned from Tribeca: was that because they are all taken or the site database was missing Tribeca? This information cannot be determined solely from the query answer.

How might we try to solve this problem? First, we might try splitting the XPATH query into two queries, one for Soho and one for Tribeca, but we would not learn anything more. Second, we might try augmenting the database with place holders (e.g., for Tribeca) that tag missing data. However, unless the site database contains placeholders for *all* data in New York, which is not a scalable solution, the XPATH query would not return all the needed placeholders. E.g., adding

```
<neighborhood @id='Tribeca' @tag='placeholder'>
```

to the New York site would not change the answer, because the query calls for specific data within the Tribeca neighborhood. Finally, we might try maintaining meta information about what fragment of the XML document is in the site database. There is a tradeoff between how much meta data we would need and how flexible the partitioning is — this approach may require significant restrictions on the fragmentation, otherwise the meta data may be as large as the database itself! Moreover, given an XPATH query, it is not clear how to combine this meta data outside the database with what is inside the database. For example, suppose that neighborhoods had a `numberOfFreeSpots` attribute and parking spaces had a `price` attribute, and the query asks for available no cost spots in block 1 of Soho and Tribeca:

```
/usRegion[@id='NE']/state[@id='NY']/city[@id='New York']
 /neighborhood[@id='Soho' OR @id='Tribeca']
              [@numberOfFreeSpots > 0]
 /block[@id='1']/parkingSpace[available='yes'][@price='0']
```

If the site database contained this attribute for both neighborhoods, and the meta data reflected this, then whether or not we need to visit the site(s) containing Tribeca block data depends on the *value* of this attribute. In order to make this decision, we would need to determine that a specific subquery requesting just this attribute should be posed to the site database, and then combine its answer with the meta data – clearly this would not be easy. Given the generality of XPATH queries, this type of scenario may arise frequently and in more complicated contexts.

A better approach is to include *inside the database* any meta information on what fragment of the XML document is present at the site, as tag attributes associated with the relevant portions of the database. These attributes must indicate not only what data is present or missing, but *which sites to visit to find the missing data.* As illustrated above, XPATH is insufficiently powerful to effectively use these tags. In the next section, we present our novel solution to these challenges, based on the more powerful XSLT language. Our technique solves these fragmentation challenges, and moreover, it enables very flexible query result caching at the sites.

3. OUR SOLUTION

In this section, we describe our solution to the challenges outlined in the previous section. We begin with an overview of the system before describing the various components in detail.

3.1 Overview

Our query processing system starts with the XML document defined by the service. For simplicity, assume that the document is fixed: only the values of attributes and fields are changing.[4] The document defines a logical hierarchy on the data (see Figure 1). At any point in time, we have partitioned *ownership* for fragments of the document to a collection of sites (discussed in Section 3.2). A site may also cache data owned by other sites (Section 3.3).

Users pose XPATH queries on the view of the data as a single XML document. The query selects data from a set of nodes in the hierarchy. The query is sent directly to the lowest common ancestor of these nodes, using our technique for *self-starting distributed queries* (Section 3.4). There it begins a potentially recursive process, which we denote *query-evalute-gather* (Section 3.5). Upon receiving a query, the site manager queries its local database and cache, and evaluates the result. If necessary, it gathers missing data by sending subqueries to other sites, who may recursively query additional sites, and so on. Finally the subquery answers return from the other sites, and the combined results are sent back to the user.

XPATH queries supported. Throughout this paper, we take the common approach of viewing an XML document as *unordered*, in that we ignore any ordering based solely on the linearization of the hierarchy into a sequential document. For example, although siblings may appear in the document in a particular order, we assume that siblings are unordered, as this matches our data model. Thus we focus on the *unordered* fragment of XPATH, ignoring the few operators such as *position*() or axes like *following-siblings* that are inappropriate for unordered data. We support (and have implemented) the entire unordered fragment of XPATH 1.0.

3.2 Data partitioning

There are two distinct aspects to data partitioning: *data ownership*, which dictates who owns what data, and *data storage*, which describes the actual data stored at each site. Our partitioning scheme is based on a series of partitioning rules, tags (i.e., special attributes), and invariants that must be maintained to ensure correct answers. Our scheme uses the following definitions.

IDable nodes. We associate with certain (element) nodes in a document an "id" attribute (see Figure 3). This *id* attribute is in the spirit of a DTD's *ID type*.[5] However, we require only that the id be unique among its siblings with the same element name (e.g., the `parkingSpace` siblings within a block), whereas *ID type* attributes must have globally unique values. This distinction is quite helpful for us, because our document is fragmented across many sites, so it would be difficult to ensure globally unique names. Moreover, the values for our id attributes are short names that make sense to the user query (e.g., New York).

DEFINITION 3.1. *A node in a document is called an IDable node only if (1) it has a unique id among its siblings with the same element name, and (2) its parent is an IDable node. The root node of a document is also an IDable node. The ID of an IDable node is defined to be its* $\langle elementname, id \rangle$ *pair.*

[4] The more general scenarios are addressed in Section 4.

[5] A *DTD* defines the schema for XML documents, including restrictions on attribute values.

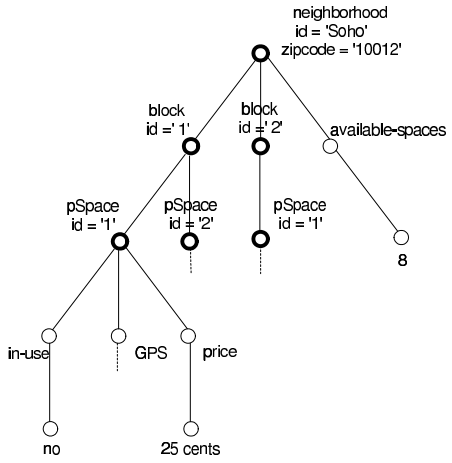

Figure 4: IDable Nodes in a Document

Figure 4 shows an example document, and its IDable nodes (denoted in bold). Note that each IDable node can be uniquely identified by the sequence of IDs on the path from the root to the node. We will frequently exploit this feature in our system. For instance, IDable nodes can be placed on different sites than their parents or siblings, without any ambiguity.

DEFINITION 3.2. *The local information of an IDable node n is defined to be the document fragment containing: (1) all the attributes of n, (2) all its non-IDable children and their descendants, and (3) the IDs of its IDable children. The local ID information of an IDable node n is defined to be the document fragment that contains (1) the ID of the node, and (2) the IDs of all its IDable children.*

Thus the local ID information of an IDable node is a subset of the local information of that node. For example, the local information of the Soho node in Figure 4 is:

```
<neighborhood @id='Soho' @zipcode='10012'>
  <block @id='1'></block>
  <block @id='2'></block>
  <available-spaces>8</available-spaces>
</neighborhood>
```

Its local ID information is:

```
<neighborhood @id='Soho'>
  <block @id='1'></block>
  <block @id='2'></block>
</neighborhood>
```

Note that the document fragments corresponding to the local informations of the IDable nodes form a nearly-disjoint partitioning of the original document, with the only overlap being the IDs of the IDable nodes. Also, the notion of an IDable node could be extended to include any node with a uniquely named root-to-node path (such as `available-spaces` in Figure 4); however, this would complicate our technique below for self-starting queries.

Data ownership. We permit each site to own an arbitrary set of nodes from the document, under the following constraints:

- Each node in the document must be owned by exactly one site.
- A node may have a different owner than its parent only if it is an IDable node.

This enables considerable flexibility for partitioning. For example, a site may own a node, a few of its grandchildren and cousins, but not the intervening nodes in the hierarchy. Our query processing algorithms must ensure correct answers in the presence of any such partitionings.

The owner of a node is ultimately responsible for any queries on that node. By requiring that only an IDable node may be on a different site than its parent, we ensure that any such node is globally addressable. (We describe in Section 3.4 how the system locates the site corresponding to the node.)

Data stored at each node. The data that is stored at each site is essentially the union of the local informations and the local ID informations of a set of nodes. There are two invariants we maintain on the stored data: (I1) each site must store the local information for the nodes it owns, and (I2) if (at least) the ID of a node is stored, then the local ID information of its parent must also be stored. Note that this implies that the local ID information of all ancestors of such a node is also stored.

A special attribute called *status* (meaningful only for IDable nodes) is used to summarize the stored data for a node, in order to ensure correct and efficient query processing. It has four possible values:

- *owned:* The site owns this node. By invariants (I1) and (I2), it has the local information of the node and at least the local ID information of all its ancestors.
- *complete:* The site has the same information stored as *owned*, except that it does not own the node.
- *ID-complete:* The site has the local ID information for this node and the local ID information of all its ancestors, but it does not have all the local information for the node.
- *incomplete:* For this node, the site has only its ID.

Our system maintains the invariant that each node at a site falls into one of these four categories (a non-IDable node is implicitly assumed to have the same *status* as its lowest IDable ancestor).

Intuition behind this structure. What have we accomplished by these partitioning rules, special attributes, and invariants? *If a site has information about a node (beyond just its ID), it knows at least the IDs of all its IDable children, and also the IDs of all its ancestors and their siblings*. Thus when a query is routed to this site, it can either answer it using the document fragment it has, or it knows which parts are missing (the missing parts will always be IDable nodes and the information in the subtrees below them). It can then contact the appropriate sites that own the missing parts (because it has their unique names) and get the information from them.

As such, any given site has in its document fragment the information needed to gather an answer to a query, even if it does not have all the data itself.

3.3 Caching

Our goals are (1) to have considerable flexibility for replicating data at multiple sites on the fly, and (2) to enable efficient and correct query processing despite this dynamic caching. The key observation is that the scheme outlined in Section 3.2 is well-suited to accomplishing these goals.

Storing additional data. A site can add to its current document any document fragment that satisfies: (C1) the document fragment is a union of local informations or local ID informations for a certain set of nodes, and (C2) if the document fragment contains local information or local ID information for a node, it also contains the

local ID information for its parent (hence all its ancestors). Then by merging this new document fragment with the existing document, we are guaranteed to maintain invariants (I1) and (I2) above. Moreover, updating the status attributes is straightforward.

Caching query results. An important special case of the above is the caching of query results. Recall that we route a query to its LCA and then recursively progress down the hierarchy to pull together the answer. In our current prototype, whenever a (partial) answer is returned to a site, we cache the answer at the site. Thus a site manager aggressively caches any data that it sees. This has two benefits. First, subsequent queries on the same data can be answered by the site, thereby avoiding the expense of gathering up the answer again.[6] Second, it automatically tunes the creation of additional copies to the query workload; such copies can distribute the workload over multiple sites. To make this caching possible, we generalize the subqueries sent to sites, making them fetch the smallest superset of the answer that satisfies (C1) and (C2) above. (This does not alter the answer returned to the user. Details are in Section 3.5.)

One could choose to use our generalized subqueries and caching techniques only when the workload seems to dictate it. However, we found that the cost of initially transferring additional data was minimal compared to the significant gains achieved by caching (see Section 5).

Partial match caching. A key feature of our caching scheme is its ability to support *partial match caching*, which ensures that even partial matches on cached data can be exploited and that correct answers are returned. Because our invariants ensure that we can always use whatever data we have in a site database, and fetch any missing parts of the answer, we can provide very flexible partial match caching. For example, the query in Figure 2 may use data for *Soho* cached at the *New York* site, even though this data is only a partial match for the new query. Similarly, if distinct *Soho* and *Tribeca* queries result in the data being cached at *New York*, the query may use the merged cached data to immediately return an answer. Even if the earlier queries have different predicates, our generalization of subqueries may enable the later queries to use the cached data. Note also that a site can determine when a collection of queries has resulted in all the IDable siblings being cached, and hence respond to queries over *all* such siblings (*subsumption*). For example, suppose there are three neighborhoods *downtown*, *midtown*, and *uptown* in a city *Brooktown*, all on different sites than the city node. Then independent queries that cause the three neighborhoods to be cached at the Brooktown node, would lead to the query

```
... /city[@id='Brooktown']/neighborhood/block
        /parkingSpace[@available = 'yes']
```

being correctly answered from just the Brooktown site. (This query requests all available parking spaces in Brooktown.) This is because invariant (I1) above ensures that Brooktown always maintains the IDs of all its neighborhoods, so it can detect when it has all of its neighborhoods cached.

Evicting data. Any data can be removed as long as it is always removed in units of local informations or local ID informations of IDable nodes and the conditions outlined above are still valid after the removal.

In summary, our scheme makes it easy to provide flexible caching. The main challenge is to do the above operations efficiently, without having to fetch the entire document into memory and without touching any more of the document than necessary. As it turns out, this task is accomplished using the mechanism for query processing in general, which we discuss in Section 3.5.

[6] Issues of staleness of cached data are discussed in Section 4.

3.4 Finding sites

In this subsection, we discuss how the system can determine the IP address for any site needed during query processing. Recall that the mapping of IDable nodes to sites is arbitrary and dynamically changing. However, there are only two situations in which IP addresses are needed during query processing: (1) when the query initially is posed by a user somewhere in the Internet, and (2) when a site manager determines that it needs to pose a subquery to a specific IDable node. We consider each situation in turn.

Self-starting distributed queries. Users anywhere on the Internet can pose queries. For scalability, we clearly do not want to send all queries to the site(s) hosting the root node of the hierarchy. Instead, our goal is to send the query directly to the lowest common ancestor (LCA) of the query result. But how do we find the site that owns the LCA node, given the large number of nodes and the dynamic mapping of nodes to sites? Our solution is (1) to have DNS-style names for nodes that can be constructed from the queries themselves, (2) to create a DNS server hierarchy identical to the IDable node hierarchy, and (3) to use DNS lookups to determine the IP addresses of the desired sites. Recall that each IDable node is uniquely identified by the sequence of IDs on the path from the root to the node. Thus our DNS-style names are simply the concatenation of these IDs. For example, for the query in Figure 2, its LCA node is New York. We construct the DNS-style name

```
city-new_york.state-ny.usregion-ne.parking.ourdomain.net
```

perform a DNS lookup to get the IP address of the New York site, and route the query there.

A key feature is that no global information is needed to produce this DNS-style name: it is extracted directly from the query! We have a simple parser that processes the query string from its beginning, and as long as the parser finds a repeated sequence of `/elementname[@id=x]`, it prepends to the DNS name. The DNS lookup may need several hops to find the appropriate DNS entry, but then this entry is cached in a DNS server near to the query, so subsequent lookups will find the IP address in the nearby DNS server. Note that no information about the XML document (or its schema) is needed by the parser.

Sending a subquery. When a site manager determines that a query requires data not in its site database, then by our invariants, it has the root-to-node ID path for the IDable node it needs to contact. To see this, observe that each piece of missing data is in the local information of some IDable node. Consider one such IDable node n. By invariant (I1), this node is owned by a different site. By invariant (I2), regardless of n's status value, we have its ID, and the IDs of all its ancestors. Thus we can produce the DNS-style name for any needed IDable node solely from the information in the site database, and then perform the lookup to get the IP address. A key feature of this design is that the mapping of IDable nodes to IP addresses is encapsulated entirely in the DNS entries, and not in any site databases. This makes it relatively easy to change the mapping as desired for load balancing and other purposes.

3.5 Query-Evaluate-Gather

We now describe our *query-evaluate-gather* technique for detecting (1) which data in a local database fragment is part of the query result, and (2) how to gather the missing parts. As noted above, our invariants guarantee that the site has all the information required to answer a query (including whether it is required to contact other sites). Handling arbitrary XPATH queries turns out to be

quite challenging though, because of the richness of the language.

As an example, consider the following (only moderately complex) query:

```
/usRegion[@id='NE']/state[@id='NY']/city[@id='New
York']
/neighborhood[@id='Soho']/block[@id='1']
/parkingSpace[not(price > ../parkingSpace/price)]
```

This query requests the least pricey parking spot in a particular block in Soho (XPATH 1.0 does not have a **min** operator). Consider a scenario where the individual parkingSpaces are owned by different sites and moreover, each site only stores the local information for the parkingSpace it owns (this is a permissible configuration). Such a configuration is problematic for this query, because none of the sites have sufficient information to evaluate the predicate. This motivates the following definition.

DEFINITION 3.3. *The nesting depth of an XPATH query is defined to be the maximum depth at which a location path that traverses over IDable nodes occurs in the query.*

We will illustrate this definition through a few examples:

`/a[@id=x]/b[@id=y]/c` → nesting depth = 0
`/a[@id=x]//c` → nesting depth = 0
`/a[./b/c]/b` → nesting depth = 1 (if *b* is IDable) or 0 (otherwise)
`/a[count(./b/c) = 5]/b` → nesting depth = 1 (if *b* is IDable) or 0 (otherwise)
`/a[count(./b[./c[@id=1]])]` → nesting depth = 2 (if *c* is IDable) or 1 (if *c* is not IDable, but *b* is) or 0 (otherwise)

The complexity of evaluating a query increases with the nesting depth of the query. Queries with nesting depth 0 are the easiest to solve, because the predicates can always be evaluated using the local information (which is always present at the site that owns the node). However, as the examples in Section 2 showed, even this case is challenging, and there were no good previously known solutions.

The basic QEG scheme. In the remainder of this section, we describe our approach, assuming nesting depth 0 (Extensions to larger nesting depths are discussed in Section 4.) Because XPATH is insufficiently powerful, we use XSLT to *query* the database, *evaluate* what is there, and send subqueries to *gather* the missing parts of the answer. We show how the XSLT programs used by our scheme can be generated directly from the XPATH query.

Given an XPATH query Q, let LOCAL-INFO-REQUIRED be the set of element names (tags) such that the final answer should include the entire local information for any IDable node with one of these tags, if the node satisfies Q. As an example, for the following query on the database shown in Figure 4,

`... /neighborhood[@id='Soho']/block`

LOCAL-INFO-REQUIRED = {`block`, `parkingSpace`} The query

`... /neighborhood[@id='Soho']/block/parkingSpace`

on the other hand, only requires local information about `parkingSpace`. Note that, this is consistent with the semantics of XPATH, because XPATH returns entire subtrees in the document rooted at the nodes selected by the query.

When a site manager receives a query Q, it generates an XSLT program from that query that executes the following algorithm:

1. Let *cur* be the node in the document under consideration at any time, tag_{cur} be the element name (tag) of *cur*, and let P be the set of predicates on tag_{cur} in the query.
2. Set *cur* to be the root of the document at the site.
3. Depending on the *status* of *cur* in the document, there are four possibilities :
   (a) *status = incomplete:* In this case, see if P can be divided into two predicates P_{id} and P_{rest}, such that P_{id} contains only predicates on the id attribute, and $P = P_{id}\&\&P_{rest}$. If this is possible, evaluate P_{id} against the current node. If it evaluates to true, then form a subquery for evaluating the rest of the query and note this by adding an *asksubquery* tag to the answer being formed. A post-processing step will then send this subquery to its LCA, in order to gather missing parts of the answer. If P_{id} evaluates to false, it is also noted in the answer being formed, so that the post-processor knows that a subquery does not need to be asked.
   If the division of P into two such predicates is not straightforward, we assume that the node may be part of the answer and form a subquery for evaluating the rest of the query as above (i.e., we were unable to avoid this subquery).
   (b) *status = id-complete:* The actions performed in this case are similar to the above case, except that if tag_{cur} is not in LOCAL-INFO-REQUIRED and $P = P_{id}$, then we can recurse on the children nodes without having to form any subquery. On the other hand, if tag_{cur} is in LOCAL-INFO-REQUIRED, then the local information for this node is required in the answer, and as such, we must ask a subquery to gather that information.
   (c) *status = owned:* In this case, we have complete information to evaluate the predicate P. If P is satisfied, then recurse on the IDable children of *cur*, and also copy the local information into the answer being formed depending on whether tag_{cur} is in LOCAL-INFO-REQUIRED. Only local ID information needs to be copied if tag_{cur} is not in LOCAL-INFO-REQUIRED.
   (d) *status = complete:* The actions performed in this case are similar to that for the above case, except for any predicates that specify consistency requirements. We will discuss this case in the next section.

This XSLT program is compiled and then executed on the site document, with the result being an annotated document that contains a subset of the answer plus placeholders for where subqueries need to be asked (if any). When the subqueries return, the returned answers are spliced in, replacing the placeholders. When all the subqueries have returned, the resulting answer is returned to the site that sent Q.

4. EXTENSIONS

In this section, we discuss extensions to our scheme, including cache consistency issues, handling larger nesting depths, ownership changes, schema changes, and speeding up XSLT processing.

Query-based consistency. Due to delays in the network and the use of cached data, answers returned to users will not reflect the absolutely most recent data. Instead, we provide a very flexible mechanism in which *each query* may specify its tolerance for stale data. We store timestamps along with the data, indicating when the data was created. An XPATH query specifying a tolerance is automatically routed to the data of appropriate freshness. In particular, each query will take advantage of cached data only if the data

is sufficiently fresh. For example, a consistency predicate such as `[timestamp > now - 30]` in a query Q means that Q can be answered using any data whose timestamp is within 30 seconds of the time Q was posed.

Although allowing users to specify a degree of staleness violates strict data transparency, we believe it provides an easily understandable means for queries that trade off freshness for possibly large performance improvements. For example, when a user is several miles from her destination, the Parking Space Finder service may fire off queries that tolerate minutes-old availability information. As the user approaches her destination, the service fires off queries that insist upon the most recent data.

The following changes to the XSLT program are needed to handle consistency predicates in queries. If *status* = *owned*, then we ignore consistency predicates, because the owner of the data has the freshest copy. Thus the semantics of the above consistency predicate allows for returning the freshest data even if that data is more than 30 seconds old. This ensures that users get an answer. Alternatively, the system could return an error message. If *status* = *complete*, and we can separate out the consistency predicates $P_{consistency}$ from P, then we first check P_{rest} — the rest of the predicates in P. If that evaluates to false, then there is no need to check for the consistency predicates. If that evaluates to true and $P_{consistency}$ evaluates to false, then we add a *asksubquery* tag at this point to signal the post-processor. As before, if $P_{consistency}$ is not readily divided out, we fall back to adding a *asksubquery* tag, in order to query the owner.

Larger nesting depths. The main challenge with XPATH queries with nesting depths greater that 0 is that the query may specify predicates that can not be evaluated locally (recall the example query in Section 3.5). Many such queries are quite natural in the kinds of applications we are interested in.

There are two approaches to solving such queries. The first approach is to collect all the data needed to evaluate the predicate at one site. The main question here is: what data needs to fetched and at which site should the data be collected? Referring to our example query from Section 3.5,

```
.../block[@id='1']
      /parkingSpace[not(price > ../parkingSpace/price)]
```

even though the predicate with nesting depth 1 is associated with the *parkingSpace* tag, because of the *upward* reference in the predicate (".."), the data needed to check this predicate is the entire subtree under the *block* tag.

Currently, we solve such a query by analyzing the query to find out the earliest tag that is referred to in such a nested predicate in the query. In this example query, this tag would be the *block* tag. During query execution, when the XSLT program encounters a node with this tag, it stops the execution at that point, issues a subquery to fetch all the data under that block (in this case, using the query `.../block[@id='1']/`), and proceeds with the execution when the answer to the subquery returns. This approach guarantees that whenever the predicate is evaluated, all the data that it requires is always present locally.

This approach may not be optimal for certain queries. For example, consider a (frivolous) query requesting all available spaces in all cities that have Soho as a neighborhood:

```
...  /city[./neighborhood[@id='Soho']]/ ...
```

Fetching all the data below the city nodes at the corresponding sites (as will be done by the above approach) may be an overkill for this query. It would be preferable to just evaluate the predicates directly, which can be done by firing off subqueries of the form

```
boolean(...  /city/neighborhood[@id='Soho'])
```

We are planning to implement and experiment with this approach in the future.

Ownership changes. For the purposes of load balancing or accommodating arriving or departing sites, it is useful to be able to dynamically change the ownership of a set of IDable nodes. The transition must appear simultaneous to the rest of the system. The steps to be done to transfer an IDable node are (1) the site taking ownership of an IDable node fetches a copy of the local information from the owner, (2) any sensor proxy reporting to the previous owner is asked to report to the new owner, and (3) the new owner sets the *status* for that node to *owned* while the old owner sets the *status* to *complete*. In addition, the DNS entry needs to be updated to the IP address of the new owner. Various relaxations in simultaneity are possible, due to the fact that fresh data obsoletes the old data. The DNS update can be done lazily because the old site can reject (or possibly forward) messages targetting the transferred IDable node.

Schema changes. Schema changes that do not affect the hierarchy of IDable nodes can be done locally by the site manager that owns the relevant fragment of the data. Such schema changes include adding attributes to, or deleting attributes from the data, and adding or removing non-IDable nodes. This might lead to transient inconsistencies as the site manager has no way of knowing who else might have cached that part of the data. But in a continously changing environment such as ours, we expect this inconsistency to be corrected quickly.

Some of the schema changes that affect the hierarchy of the IDable nodes can be handled similarly. For example, addition or deletion of IDable nodes is performed by the site manager that owns the parent of the affected IDable node. More drastic schema changes, such as a restructuring of the hierarchy, may have to be performed by collecting the entire document at one site, making the changes, and redistributing the document back amongst the nodes. Once again, such a change might lead to transient inconsistencies in the data that, although undesirable, are permissible for the kinds of applications we are looking at. More stringent consistency guarantees can also be implemented by maintaining auxiliary information about the copies of the data as we will discuss in Section 6.

Speeding up XSLT processing. Recall that the QEG processing at a site involves creating an XSLT program from the XPATH query, compiling the XSLT program, and then executing the compiled program against the document. As demonstrated in Section 5, there is significant overhead in the compilation step. We now describe our technique for eliminating most of this overhead by directly generating mostly *compiled* XSLT programs.

When a site manager starts up, one of the first things it does is to create and compile an XSLT program using a dummy XPATH query. Once this is done, it identifies the parts of this compiled XSLT query that depend on the XPATH query. Subsequently, when the site manager needs to create the XSLT program for an actual XPATH query, it simply modifies this XSLT program directly to set the query-dependent information. Some compilation is still required, because the query-dependent structures are in XPATH and need to be recompiled. But the cost is much lower than the cost of compiling the entire XSLT program.

5. PERFORMANCE STUDY

In this section, we present a preliminary performance study demonstrating the features of our system, and the flexibility and effectiveness of our architecture. The salient points of our experimental study can be summarized as follows :

- Our flexible architecture allowing arbitrary logical-to-physical

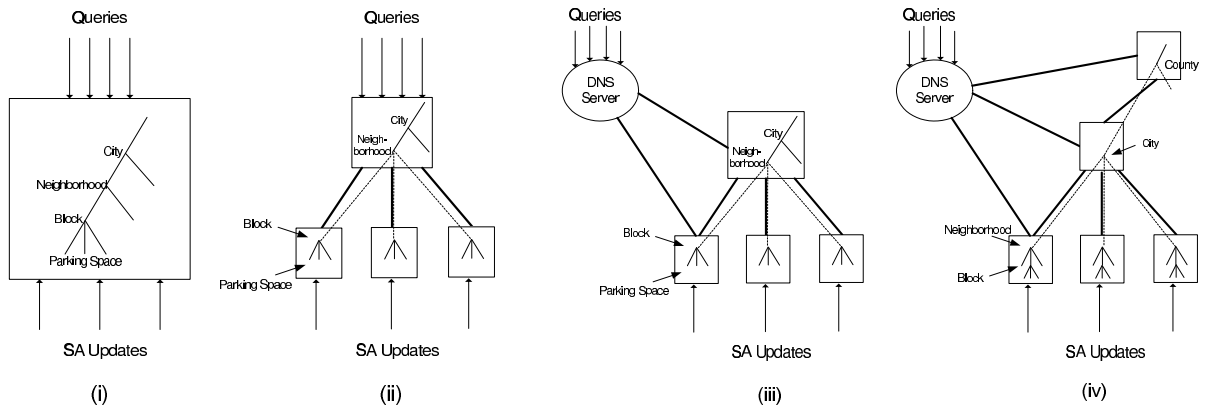

Figure 6: Sensor Database Architectures

Figure 5: Webcams monitoring toy parking lots

mappings can harvest the processing power in the system more effectively than any traditional solution.

- Caching can be very effective in both reducing latencies of queries, and in offloading work to increase overall throughput.
- Compiling XSLT queries directly leads to huge savings in query processing time.

5.1 Experimental setup

For most of our experiments, we use a homogenous cluster of 9 2GHz Pentium IV machines running Redhat Linux 7.3 connected by a local area network. In our current prototype, we have 10 sensor proxies that each have an associated sensor (webcam) that monitors a toy parking lot (Figure 5). For these larger-scale experiments, we simulate as many sensor proxies as required by running fake sensor proxies that produce random data updates. As our backend, we use the Apache Xindice 1.0 [1] native XML database. Xindice currently does not support XSLT processing (though it is a planned feature). Hence, in our current prototype, we use the Xalan XSLT processor [2] for that purpose (Xalan is also used by Xindice for processing XPath).

We use an artificially generated database for our *parking space finder application* consisting of a total of 2400 parking spaces using a hierarchy similar to the one shown in Figure 1. This database models a small part of a nationwide database and contains 2 cities, 3 neighborhoods per city, 20 blocks per neighborhood, and 20 parking spaces per block. We envision that the queries in such a database will typically ask for available parking spaces geographically close to a particular location. As described in Section 3.4, the queries are initially routed to the site manager that owns the lowest common ancestor of the data, and as such, we distinguish between the queries in our workload based on the level in this hierarchy to which they are first routed.

- **Type 1:** These queries ask for data from one block, specifying the exact path to the block from the root.
- **Type 2:** These queries ask for data from two blocks from a single neighborhood.
- **Type 3:** These queries ask for data from two blocks from two different neighborhoods. (Such a query may be asked by a user if her destination is near the boundary of the two neighborhoods.)
- **Type 4:** These queries ask for data from two blocks from two different cities. (The destination is near the boundary between two cities.)

We expect that type 3 and type 4 queries will be relatively uncommon, and most of the queries will be of type 1 or 2. Hence, we will also show results for mixed workloads that include more queries of the first two types.

5.2 Handling sensor updates

An update from a sensing agent is typically routed directly to the site manager that owns the relevant node in the hierarchy. On the site manager side, processing a sensor update involves updating the document at the site with the new availability information, as well as timestamping the data. A single site manager is typically able to handle 200 updates a second in our current prototype. The total number of updates that can be handled by the system scales linearly with the number of site managers among which the data is distributed.

5.3 Architectural comparison

With our first set of experiments, we demonstrate the flexibility of our arhcitecture in harvesting the processing power available in the system. We consider four different architectures each of which

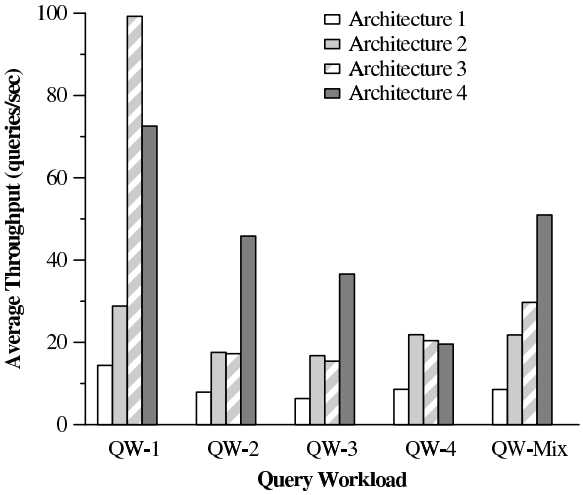

Figure 7: Query Throughputs for Various Architectures

is a viable alternative for our application. We show results for five query workloads, QW-1, QW-2, QW-3, QW-4, consisting of randomly generated queries of types 1 to 4 respectively, and QW-Mix, a mixed query workload that asks 40% queries of type 1 and 2 each, 15% queries of type 3 and 5% queries of type 4.

- **Centralized –** Figure 6(i): In this architecture, all the data is located at a central server, and data updates as well as queries are sent to a central server. Such an architecture can not be expected to scale very well as it can handle very few sensor updates (200 updates per second).
- **Centralized querying, distributed update –** Figure 6(ii): In this scenario, we offload the sensor updates to other machines by distributing the *blocks* among the rest of the machines. The queries are still sent to the centralized server, because the server is the sole repository for the mapping from blocks to physical machines. This scenario is intended to simulate a simple distributed object-relational database, where the blocks form an object-relational table that is distributed and the hierarchy is maintained as another set of tables that are all stored at the central server. Of course, this is not the only possible design using an object-relational database, but most such designs will suffer from similar flaws as this design. (Object-relational sensor databases are discussed further in Section 6.)
- **Distributed querying, distributed update, fixed two-level organization –** Figure 6(iii): This scenario is similar to the above scenario, except that we use the DNS server to store the mapping from blocks to physical machines. This helps in solving the queries of type 1 significantly, but does not help much with other kinds of queries.
- **Distributed querying, distributed update, hierarchical organization –** Figure 6(iv): A more logical organization of the data, considering the nature of the query workloads, would be to arrange it hierarchically in a geographic fashion. We do this by assigning the 6 neighborhoods to 6 different site, assigning the 2 cities to two different sites, and assigning the rest of the hierarchy to one site. This corresponds to the scenario of choice in IrisNet.

Note that all architectures use the same number of sensor proxies, and the latter three architectures use the same number of sites.

Figure 7 shows the query throughputs for these four architectures for the five query workloads. As we can see, the centralized solution does not scale very well for querying either, and can handle very few queries efficiently. Although distributing only the updates (Architecture 2) increases the number of sensor data updates the system is able to handle, it only improves query throughput by a factor of 2 over the centralized solution, because all queries go through the centralized server. Using DNS for self-starting distributed queries (Architecture 3) shows the effectiveness of this technique, as the throughput for type 1 queries increases by nearly an order of magnitude. However, all other queries still go through the central server, which is the bottleneck for the other types of queries, and also for the mixed workload. The hierarchical distribution of data (Architecture 4) turns out to perform the best overall as it can handle all kinds of queries quite well. It does perform 25% worse than Architecture 3 for type 1 queries, because it is using 25% fewer machines in processing these queries. However, it performs at least 66% better than the other architectures on the mixed workload.

Because the mapping of logical nodes in the hierarchy to the physical sites is not fixed in our architecture, our system is able to handle the skewed query workloads arising in our application much more effectively than other architectures. For example, during business hours, a large percentage of the queries may be asking for information for blocks in the *Downtown* neighborhood. Such a skewed query workload can be much better handled by redistributing those blocks across all available machines, as opposed to them being on a single node as in the above mapping. Figure 8 shows the results of a simple experiment where we skewed the query workload to consist of 90% queries targetting a single neighborhood for type 1 and type 2 queries. As we can see, the original distribution of the data (Architecture 4) does not perform very well, whereas a more balanced architecture that distributes the blocks in that neighborhood across all sites has a factor of 4 higher throughput for this workload.

5.4 Dynamic load balancing

As discussed in the earlier section, our system is capable of changing the mapping of logical nodes in the hierarchy to the physical sites dynamically while still being able to answer queries.

We show the effectiveness of dynamic load balancing through an experiment that traced the average throughput of the system over time. For this experiment, we started multiple querying clients all asking queries of type 1 with 90% of the queries directed to a fixed neighborhood X, and 10% of the queries asking for a block in a randomly chosen neighborhood. Figure 9 shows the average throughput of the system over time. As we can see, initially when all the blocks in neighborhood X are located on a single site, the average throughput of the system was quite low. At 206 seconds into the experiment (the first dashed line), we started redistributing the data on that site to other sites by explicitly asking the site manager to delegate its blocks to other nodes one by one. In our current prototype, this has to be done by sending a request for delegating ownership of each block one at a time. These requests were sent at even intervals until 373 seconds (the second dashed line). At that time, the blocks under neighborhood X were distributed across all the machines evenly. As we can see, the average throughput of the system increased by nearly a factor of 3 even with this crude load-balancing scheme, while the system was still able to answer queries.

5.5 Caching

Caching of query results has two benefits: it can be used to re-

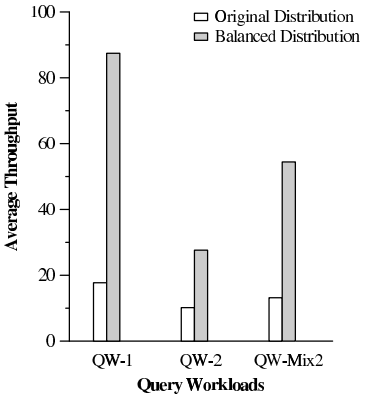

Figure 8: Load Balancing (QW-Mix2 consists of 50% type 1 and 50% type 2 queries)

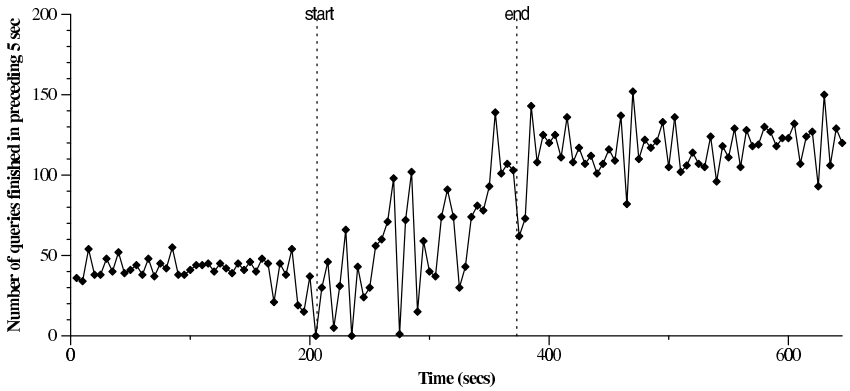

Figure 9: Dynamic Load Balancing

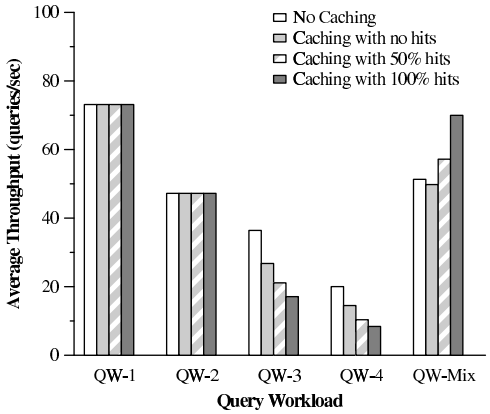

Figure 10: Caching Throughputs (Architecture 4)

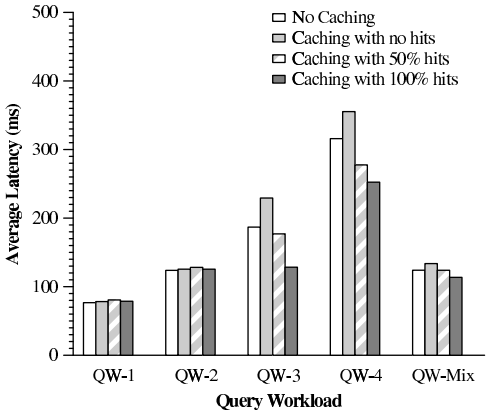

Figure 11: Caching Latencies (Architecture 4)

duce response times by bringing the data close to the queries, and it can be used to off-load work from the sites that own popular data to less loaded sites (auto-tuning). Figure 10 shows how caching can help in increasing overall throughput of the system by offloading work. We use the fourth architecture and show results without caching, with caching but no hits (this demonstrates the overheads of caching), with caching and 50% hits, and finally, with caching and 100% hits. As we can see, caching induces minimal overhead in the system. Caching does not affect the throughputs for type 1 and type 2 in this scenario because these queries are always directed to the machine that has the full data. On the other hand, for type 3 and type 4 queries, we see that as the probability of hits increases, the overall throughput of the system *reduces* significantly. This happens because after the initial few queries, all the queries are completely answered by the top-level sites (sites that are assigned to the city and county nodes), and these nodes become the bottleneck. As we will see in Section 5.6, the time taken to forward a query to another node is much less than the time taken to process the query when the answer is present at a node. This suggests the need for bypassing the cache under heavy load imbalance. On the other hand, caching improves throughput by up to 33% for the more realistic mixed workload, because the otherwise idle top-level sites can absorb some of the load from the lower-level sites.

The effects of caching are more pronounced in terms of latency. Unfortunately, because we are not using a wide area network in our experiments, it is hard to see these effects in our experiments. Figure 11 shows the average latencies for the five workloads under different caching scenarios as discussed above. Even in the case of near-zero network latencies, we can see that query latencies are reduced by 10–33% for type 3 and type 4 queries, and for the mixed workload.

5.6 Micro-benchmarks

To understand how the time to answer a query is distributed amongst various tasks, we ran some micro-benchmarks against our system. Figure 12 shows the breakdown of query processing time depending on at which level in the hierarchy the query was asked. The query used in this experiment was a query of type 1 asking for one particular block. Even though this query will always be routed to the site that owns the neighborhood, we artificially routed this query to the sites higher up in the hierarchy to see the effect of the number of hops taken by the query. Three settings were studied: small database with naive XSLT creation, small database with fast XSLT creation, and large database with fast XSLT creation.

As we can see, for all of the scenarios, the total processing time

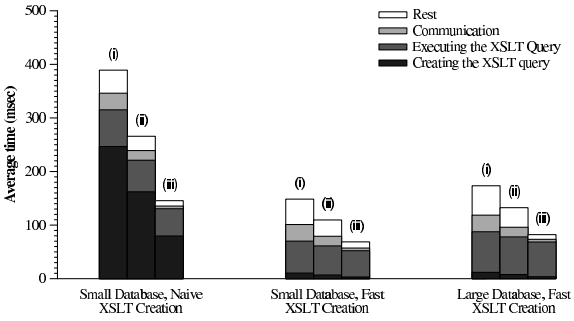

Figure 12: Microbenchmarks: when the query is routed to the site manager that owns (i) the county node, (ii) the city node, (iii) the neighborhood node

consumed by the query is reduced significantly (by over 50%) if the query is routed directly to the site that has the data, once again demonstrating the effectiveness of self-starting distributed queries. This experiment also shows why we chose to optimize the XSLT query creation time. As we can see, if the XSLT query is generated and compiled using traditional interfaces, then the time taken for this completely dominates the overall query processing time. Using direct compilation to XSLT from the original XPATH query reduces the overall query processing time by over 50%!

To see how our query processing mechanism scales with respect to the database size, we increased the total size of the database by a factor of 8 by doubling the number of neighborhoods, the number of blocks in a neighborhood, and the number of parking spaces in a block. As we can see, the processing time increased by less than 20% at each of the nodes!

These micro-benchmarks also show where the bottlenecks in our current prototype are. As we can see, most of the time is spent in executing the XSLT query and during CPU processing for communication. The communication part also includes the cost of constructing and deconstructing the messages. We believe that much of this is because we are using Java 1.3; using JIT (just-in-time compilation) and newer XSLT processing packages should significantly reduce this time.

6. RELATED WORK

Previous work in sensor databases has focused primarily on networks of closely co-located sensors with limited resources [5, 26, 27]. The sensor query processing system operates directly on continuous, never-ending streams of sensor data that are pushed into the query plans for continuous queries. The stream of data is viewed as a streaming relation (e.g., in Fjords [26]) or a time series (e.g., in Cougar [5]). These efforts have developed techniques for answering continuous queries over streaming data and for performing certain query processing tasks in the sensor network itself in order to eliminate communication and extend sensor battery lifetimes.

This paper complements this previous work by addressing fundamental challenges in distributed query processing over wide area sensor databases. Based on the applications we were considering, we sought to provide a more familiar abstraction of the database as the collection of values corresponding to the most recent updates (e.g., the currently available parking spaces). Even in this more traditional setting, there were plenty of challenges to overcome. The distributed database infrastructure in IrisNet shares much in common with a variety of large-scale distributed databases. For example, DNS [28] relies on a distributed database that uses a hierarchy based on the structure of host names, in order to support name-to-address mapping. LDAP [35] addresses some of DNS's limitations by enabling richer standardized naming using hierarchically-organized values and attributes. However, a key difference is that these efforts target a very narrow set of lookup queries (with very limited predicates), not the richness of a query language like XPATH.

Abiteboul et al. [19] present techniques for materializing and maintaining views over semistructured data. Answering queries from views is a hard problem in general. Our caching infrastructure differs significantly from this work in that we do not store or use the *queries* that resulted in the caching of the data, only the data itself. Our approach is more data-driven in nature. We generalize subqueries, tag the data, and maintain certain invariants, all to make our partial-match caching scheme tractable. There has been a lot of work on caching in distributed databases, and object-oriented databases. Franklin and Carey [22] present techniques for client-server caching in object-oriented databases. Various work [13, 12, 30, 3, 25, 4, 6, 29] discusses issues of data replication and replicate management in distributed databases. Much of this work has focused on maintaining replicas consistent with the original data, with various levels of consistency guarantees. In our work, we take the approach of *not providing* any strict guarantees of consistency. We believe that for the kinds of applications for which wide area sensor databases will be used, such strict guarantees are not required. Depending upon the requirements of an application, it is certainly possible to provide such guarantees by maintaining auxiliary information about the replicas of the data in the system. Our approach is more akin to DNS, in that it is based on specifying consistency requirements with the queries, and using a *time-to-live (ttl)* field associated with the cached copies in order to determine staleness.

Recently, there has been a lot of interest in streaming data sources and continuous query processing [9, 20, 18]. This work is related to ours in that the sensor proxies can be thought of as producing data streams, and continuous queries over such streams are quite natural in such a system. To our knowledge, most of this work focuses on a centralized system, whereas *distributed* data storage and query processing is one of our main design goals. As a result, the query processing challenges we face in our system are quite different. Also, these systems assume that the streaming data sources are relational in nature, whereas we use XML as our data model.

Although we propose a hierarchical, native XML storage approach to wide area sensor databases, an alternative would be to use a distributed object-relational database [32] to store the leaves of the XML document (as discussed in Section 5). In our parking space finder application, these would correspond to either the blocks or the parking spaces. The hierarchy information can be maintained either at a central server or along with the leaves themselves. This approach has several critical disadvantages. First, the hierarchy information becomes a bottleneck resource, as demonstrated in our performance study. Approaches to avoid this bottleneck would likely entail mimicking much of our hierarchical approach, and hence would benefit from the techniques presented in this paper. Second, the richness of XML allows transparent schema changes, and the use of highly expressive languages such as XPATH, XSLT or XQuery. Many queries that can be naturally described using these languages are not easily expressible in SQL. Third, use of such a database seriously restricts how data can be partitioned among available sites, limiting opportunities for load-balancing. Our architecture also enables powerful caching semantics naturally; we are not aware of any work on caching in object-relational databases that is equally as powerful. Much work has also been done on storing XML using object-relational databases, and publishing object-relational data as XML [15, 24, 33, 31]. This

work is orthogonal to the issues we discuss here, as the challenges in our query processing come mainly from the single document view of the data, and the distributed nature of our system.

Recent work on peer-to-peer databases [14, 7, 21, 16] is quite closely related to our work. Although our data is organized hierarchically, and for performance reasons, we expect the participating sites to also have a hierarchical organization, this is not required by our architecture. As such, the participating sites can be thought of as peers cooperating with each other to store and query data. In [14], distributed hash tables (DHTs) perform the analogous role of the DNS server in our architecture in that both of them are used to find relevant data satisfying a query. DNS is more attractive in our scenario because of the hierarchical nature of our data. Our work differs considerably in the actual query processing part itself because of our use of XML and the XPATH query language. [21, 16] discuss issues of data placement, and caching in peer-to-peer networks. The OLAP caching framework presented in [16] relates quite closely to our caching framework, but handles different kinds of data and queries.

There is a large body of literature on load balancing techniques for parallel and distributed systems (*e.g.*, [17, 8, 23, 11, 34, 10]). Our current system provides a natural mechanism for performing load balancing, but we have not yet determined effective load balancing policies for our setting.

7. CONCLUSIONS

In this paper, we motivated the view of a wide area sensor database as a distributed hierarchical database with timestamped updates arriving at the leaves. We showed the advantages of using XML as a data representation, constructing a logical site hierarchy matching the XML document hierarchy, mapping this logical hierarchy onto a smaller hierarchy of physical sites, and providing for flexible partitioning and caching that adapts to query workloads. We described the many challenges in providing efficient and correct XPATH query processing in such an environment, and proposed novel solutions to address these challenges in an effective, flexible, unified, and scalable manner. New techniques were presented for self-starting distributed queries, query-evaluate-gather, partial-match caching, and query-based consistency. Experimental results on our IrisNet prototype demonstrated the significant performance advantages of our approach even for a small collection of sites. We anticipate that these advantages will only increase when IrisNet is deployed over hundreds of sites and thousands of miles.

8. REFERENCES

[1] Apache Xindice Database. http://www.dbxml.org.

[2] Xalan-Java. http://xml.apache.org/xalan-j.

[3] D. Agrawal and S. Sengupta. Modular synchronization in distributed, multi-version databases: Version control and concurrency control. *IEEE TKDE*, 1993.

[4] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM TODS*, 1990.

[5] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards sensor database systems. In *MDM*, 2001.

[6] S. W. Chen and C. Pu. A structural classification of integrated replica control mechanisms. *ACM TODS*, 1992.

[7] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *ICDCS*, 2002.

[8] A. Fox et al. Cluster-based scalable network services. In *SOSP*, 1997.

[9] D. Carney et al. Monitoring streams - A new class of data management applications. In *VLDB*, 2002.

[10] D. Ferguson et al. An economy for managing replicated data in autonomous decentralized systems. In *ISADS*, 1993.

[11] D. R. Karger et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM STOC*, 1997.

[12] J. Gray et al. The dangers of replication and a solution. In *SIGMOD*, 1996.

[13] J. Sidell et al. Data replication in mariposa. In *ICDE*, 1996.

[14] M. Harren et al. Complex queries in dht-based peer-to-peer networks. In *IPTPS*, 2001.

[15] M. J. Carey et al. XPERANTO: Publishing object-relational data as XML. In *WebDB*, 2000.

[16] P. Kalnis et al. An adaptive peer-to-peer network for distributed caching of olap results. In *SIGMOD*, 2002.

[17] R. Blumofe et al. Cilk: An efficient multithreaded runtime system. In *PPoPP*, 1995.

[18] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[19] S. Abiteboul et al. Incremental maintenance for materialized views over semistructured data. In *VLDB*, 1998.

[20] S. Chandrasekaran et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[21] S. Gribble et al. What can databases do for peer-to-peer. In *WebDB*, 2001.

[22] M. Franklin and M. Carey. Client-server caching revisited. In *IWDOM*, 1992.

[23] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 1993.

[24] M. Klettke and H. Meyer. XML and object-relational database systems — enhancing structural mappings based on statistics. *LNCS*, 1997:151, 2001.

[25] N. Krishnakumar and A. Bernstein. Bounded ignorance in replicated systems. In *PODS*, 1991.

[26] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, 2002.

[27] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad hoc sensor networks. In *OSDI*, 2002.

[28] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *SIGCOMM*, 1988.

[29] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD*, 2002.

[30] C. Pu and A. Leff. Replica control in distributed system: An asynchronous approach. In *SIGMOD*, 1991.

[31] T. Shimura, M. Yoshikawa, and S. Uemura. Storage and retrieval of XML documents using object-relational databases. In *Database and Expert Systems Applications*, pages 206–217, 1999.

[32] M. Stonebraker and G. Kemnitz. Postgres next generation database management system. *CACM*, 1991.

[33] B. Surjanto, N. Ritter, and H. Loeser. XML content management based on object-relational database technology. In *Web Info. Sys. Eng.*, pages 70–79, 2000.

[34] B. W. Wah. File placement on distributed computer systems. *IEEE Computer*, 1984.

[35] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). Technical report, IETF, 1997. RFC 2251.