Locality Aware Networked Join Evaluation

Yanif Ahmad, Uğur Çetintemel, John Jannotti, Alexander Zgolinski
{yna, ugur, jj, amz}@cs.brown.edu
Brown University, Providence, RI 02912

Abstract

We pose the question: how do we efficiently evaluate a join operator, distributed over a heterogeneous network? Our objective here is to optimize the delay of output tuples. We discuss key challenges involved in the distribution, namely how to partition the join operator, how to place the resulting partitions on the network, and how to route inputs values from sources to our operators. Our model revolves on one simple concept – exploiting locality. We consider data locality in the distributions of input data values, and network locality in the distribution of network distances between sites. We sketch strategies to partition the input data space, and instantiate a structured topology, consisting of operator replicas to whom to route tuples for processing. Finally, we briefly discuss implementation issues that require addressing to enable the networked join proposed here.

1. Introduction

Content-addressable networks, sensor networks, and publish/subscribe systems have all begun to highlight the interplay between data routing and data indexing in today's widely distributed systems. Indeed many classes of distributed systems applications need data routing and indexing functionality at the application layer, including web caching and content delivery networks. These applications frequently require a *rendezvous point*, designating a specific place in the network where a portion of application functionality is executed. For web caching, this involves clients reading from a rendezvous, while updates are sent to, or retrieved by the rendezvous. We abstractly consider this functionality as a join operation, taking inputs from read and write stages of the application pipeline.

In sensor network applications, such as remote triage on a battlefield [26], or intrusion detection monitoring, the application requires notification, whenever a complex set of conditions hold. This corresponds to running a continuous query to detect the condition, by matching events via an application-defined predicate. This matching of events corresponds to a join operation.

In publish/subscribe systems, events generated by publishers are matched against profiles specified by subscribers. Here profiles represent stored state in the system, enabling continuous matching against events as they arrive. Under the relational model, this matching of events and profiles can be considered a join of the events relation and the subscription relation. Publish/subscribe systems have become the de facto communication model in massively multiplayer online games (MMORPGs). In these games, the virtual world is statically divided into a set of zones, with players receiving all events in their current zone. However, the static division of zones results in poor filtering efficiency, especially under scenarios of non-uniform gameplay where players tend to congregate in the virtual world. Accounting for this eventuality requires a more dynamic publish/subscribe functionality.

In this paper, we discuss a framework for the continuous evaluation of a join operator, in a widely distributed manner. We optimize the expected delay of an output tuple, caused by a successful match on two input tuples. For an MMORPG application, a continuous join eliminates the need for static subscriptions, allowing a continual matching of events, and arbitrary subscriptions. To efficiently support such a mechanism, we focus on exploiting two key characteristics of distributed evaluation – *network locality*, and *data locality*. Here network locality is defined as the proximity of the data sources' network locations. Data locality is defined as the similarity between data sources in terms of the input values produced, and the frequency these values are produced at. Data locality also captures temporal properties of the inputs, such as the synchronicities of the input values.

We approach the challenge of deploying a distributed join operator, considering two strategies to optimize for tuple latency. We initially describe a placement primitive, needed for any deployment algorithm.

This tackles the simple case of where to place a single join operator, on the network. The first question we pose for a broader deployment, is how do we determine a parallelization (or *partitioning*) of the join operator, to support distributed evaluation? We outline a strategy considering data locality in probabilistic value distributions for each data source. Our second question is how should we place these partitions of our join operator at network sites to optimize on tuple latency? We address this issue with a replication mechanism that utilizes our placement primitive. We replicate considering network locality in source locations, in conjunction with the semantic behaviour of sources, to construct a tree of join operator replicas. In short, we are attempting to reflect trends in each data source's value distribution, as corresponding trends in the networked execution of the join operator.

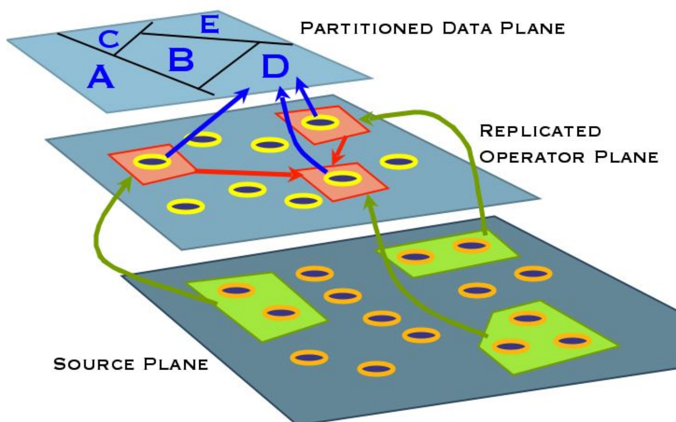

Figure 1. Networked join evaluation overview: to support a distributed deployment of a join operator we i) partition the data space, ii) assign replicas, connected in a tree, to process tuples in each partition. Sources may then send tuples to their nearest replica, via a content-based routing mechanism.

Figure 1 depicts an overview of these challenges and our proposal. Our networked join operates on a partitioned data space, evaluating each partition on a structured operator replica graph. We perform content-based routing to deliver tuples from our data sources to these operator replicas. Throughout this mechanism, we attempt to structure our deployment to rapidly output any successful join of input tuples.

To the best of our knowledge, no present day system considers the combination of network and data locality in their design. Existing distributed hash tables (DHTs), such as Chord [25], Bamboo [13], and CAN [22], provide lookup functionality and act as an index for stored data. Recent systems, such as Mercury [4], have also investigated range queries. These systems primarily focus on load balancing, and do not leverage the structure of participants in the twofold access patterns corresponding to both branches of the join. The PIER system [12] discusses several techniques for widely distributed joins, yet uses these DHTs as an unstructured rendezvous. By hashing the join key attribute, and thus performing the join at an arbitrary site, PIER eliminates any data or network locality in either the attribute or the sources. The DIM data structure [16] answers multi-dimensional range queries in sensor networks. This system considers a simple form of data locality in associating similar values at nearby network locations. However the authors do not consider data or location trends in the values produced by sources, rather assume that data values are produced uniformly at random by every sensor. In contrast our proposal, with the aid of probabilistic models of value distributions, attempts to identify frequently occurring matches, and the network origins of these matches. This information is then used to structure the rendezvous points of inputs from multiple sources, enabling sources to rapidly route their tuples to be joined.

2. System and Data Model

Our framework is designed for an infrastructure model of a distributed system. We assume the availability of a substantial set of heterogeneous network hosts, connected through a wide-area network. In this model, stability, and churn are not as crucial as in peer-to-peer systems. Furthermore, we do not address security issues here, assuming that participants belong to the same administrative domain.

Our data model is a continuous query model, as found in stream management systems ([1, 2, 6, 18]). Under this model, an infinite stream of tuples drives a push-based control flow and evaluation of the join operator. We assume our join operator is associated with a window, and for simplicity, assume each stream maintains its own window. In a distributed scenario, one open issue is how to ensure the maintenance of distributed windows with global semantics.

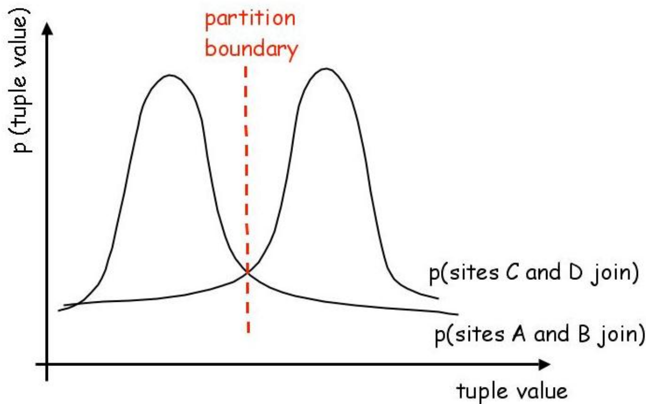

Figure 2. Simplified, motivating example: we propose exploiting data locality. In this scenario, where sites A and B join with high probability on "low" values, and sites C and D join over "high" values, we create two partitions, and place each partition to optimize for its relevant sources.

3. Networked Join Model

We begin with a simple scenario exemplifying how we intend to exploit data and network locality. Consider four sites, A, B, C and D. Sites A and B produce a certain range of values at a high rate, while sites C and D produce a different set of values at a high rate. We claim that processing these tuples in a centralized manner, where all four sites push data to a single network location, does not exploit trends in the underlying value distribution. We propose creating two *instances* of the operator, one placed near sites A and B to handle the range of values frequently emanating from these sources, and another placed near sites C and D. This is abstractly illustrated in Figure 2.

In addition to data locality, we investigate techniques to exploit network locality between the sources. Here we focus on placing replicas of instances, based on the network proximity of sources. Abstractly, we attempt to ensure nearby sources produce output tuples with low latency, at a nearby site, rather than at a single (centralized) partition instance.

3.1. Modelling Data Sources

Our join's sources may be arbitrary network sites. We assume each source maintains a probabilistic model representing the probability density function (pdf) over any values it produces. These pdfs may be obtained as a simple histogram, or with standard learning algorithms and distribution fitting techniques. Our input tuples are comprised of multiple attributes, implying these pdfs are joint distributions. We may marginalize this joint pdf to obtain the pdf for any combination of attributes. With this model, each source is able to provide the probability of producing a specific value.

3.2. Probabilistic Join Evaluation

Our join operates on tuples from all sources based on a join predicate. In the scope of this paper, we consider the equi-join operator, but remark the principles applied here may be generalized to arbitrary joins. We return to this shortly. Using the data source model described above, we may compute the expected output rate of our join operator as follows. For a single pair of sources, this output rate is the product of the total input rate, and the probability of two tuples having identical join key values. Since the join key attributes are likely to be a subset of the joining relations' attributes, this probability is equivalent to a product of each source's joint distributions, marginalized over the join key attributes.

This model of output rates is too simple for a stream-based model, where join operators are defined with windows to support asynchronous inputs. In this model, we abstract away window semantics, such as whether the operator has a window for each stream, or whether the window is a band. We simply assume we are able to ascertain a distribution yielding the probability of a specific value existing in the operator's window. We refer to such a distribution as a window distribution. For a join operator with one window per stream and source, and a sliding policy of simply removing the oldest tuple in the window, the window distribution is equivalent to the nth power of the input distribution, where n is the window size.

With this window distribution, we may obtain our expected output rate of a source pair as a product of one source's input rate, the same source's marginal distribution of join key attributes, and the window distribution of the second source. Clearly any pair of sources may join to produce an output tuple. Furthermore the join occurs over all possible join key values.

The generalization to other types of join operators, with arbitrary join predicates, may be accomplished through the definition of an indicator function yielding whether the values join. One open issue relates to how we could capture the output probability distribution of arbitrary join predicates, since this distribution is likely to be specific to each indicator function.

4. Networked Join Deployment Challenges

We now outline three key problems in deploying networked joins, and sketch their solutions. These solutions all target minimizing the total expected network delay of a tuple. We address: i) how to construct a routing plane so that sources may send their input tuples to an appropriate join operator, ii) how to augment this routing plane with semantic information to construct a join space, and subsequently use this space to identify localization opportunities, and iii) how to partition a join operator by differentiating sources based on their position within this join space.

4.1. Join Routing Topology Construction

Our routing plane connecting the join's sources and the join operator is a routing tree with sources at its leaves, and replicas of the operator as the internal nodes. We replicate operators to reduce network latency between inputs and the location processing these inputs. Operators on our replica tree compute join tuples on different incoming branches, outputting join results as well as forwarding tuples to their parents. This ensures completeness when evaluating the join. We use a hierarchical clustering algorithm to construct our replica tree, initially bootstrapping our infrastructure by clustering on a synthetic coordinate system capturing latencies between sites, such as Vivaldi [7] or NPS [19].

We initialize our cluster hierarchy by choosing our base clusters with a leader election algorithm. The base clusters are (approximately) balanced subdivisions of the synthetic coordinate space. Our algorithm is a standard leader election algorithm with a termination criterion where a site elects itself leader whenever it has knowledge of k other sites (where k is a configuration parameter) or when it has received knowledge of a site at distance d in the synthetic coordinate space (where d is also a configuration parameter).

Base cluster leaders compute a weight based on the total distance between themselves and every other site in their cluster. Following this, base cluster leaders exchange their weights with neighbouring leaders to elect a higher level leader. The termination criterion for leader election is based on a weight threshold or again a distance threshold (whichever is met first). The site meeting the condition then computes a centroid of weights, and declares it a leader for the set of known sites. Note that the centroid chosen as described above is from the set of leaders only (as these are the only sites propagating weights at this level). The centroid selected may be significantly offset from the actual centroid, thus we refine our centroid by propagating weights down the hierarchy rooted by the initial centroid. At each level we recompute the centroid amongst the sites available at that level.

4.2. Join Coordinate Space Construction

In this section we describe our methodology for augmenting this space with semantic information to support the exploitation of both network locality and data locality. With the model of join output probabilities presented previously, we claim that exploiting data locality requires comparing the join output pdfs of pairs of sources. By differentiating the output pdfs for different sets of the attribute domain, we may tailor our operator placement to optimize for the latency of accessing inputs. To this end we attempt to capture features of our sources' join pdfs in our coordinate space to support a search for localization opportunities. We do so by increasing the dimensionality of our coordinate space, to include the first m moments of the source pdfs. Thus each source is placed at a point on a $c+m$ dimensional space (where c is the dimensionality of the network-oriented coordinate space) with m moments of their own pdf. The distance function on this space is a Euclidean distance intended to approximate the similarity of pdfs and the network separation of the respective sources.

In our model, we augment the original coordinate space defined over network latencies alone to this higher-dimensional join space over time, as probability distributions are built up from the input data. We adapt our routing plane to accommodate the changing positions of the sources in the join space, leveraging our hierarchy to determine new parents for sources. When a source changes its position on the join space, we route through its existing ancestors towards the root of our replica tree, stopping at the first ancestor rooting a subtree containing a cluster encompassing the source's new position. This ancestor then propagates the source's new position back down a path in its subtree, updating the source's routing information until we reach the base cluster in which the source now lies.

4.3. Join Partitioning

Recall our objective of minimizing the total expected output delay of a tuple. We partition operators, ensuring partitions process different sets of the attribute domain, and tailor the placement of this partition to suit the sources most likely to produce outputs lying in that set. Thus our partitioning algorithm must

determine how many partitions to create and what values of the attribute domain each partition must process.

Our partitioning algorithm uses a gradient based heuristic to drive the optimization. Specifically, we optimize our objective by differentiating sources in terms of their output distributions, searching for changes in a pair's output pdf that significantly impacts the total output pdf for a particular value in the attribute domain. The point at which the output pdf changes greatly is one candidate for placing a partition boundary. Our high level intuition for placing partition boundaries at points of high pdf gradient is as follows. Given that the probability of output significantly differs in value on either side of the point of large gradient, the relative contribution of the source to the total output of the attribute values near this point also differs. Furthermore, a source's contribution to a value determines its position on the join space. Consequently the source would be best served by operators placed at different network locations (assuming the operator is placed at a cluster centroid as described previously) and using the same operator location would only increase the expected tuple latency.

Our partitioning algorithm performs its search in a distributed manner, first using a clustering technique on the join space to prune the output pdfs considered when searching over the attribute domain. Each cluster chooses a leader as a search coordinator. The search coordinator collects the probability distributions from the other sources in its cluster alone, and computes the pairwise join pdfs of these sources.

We briefly outline our mechanism for exchanging probability distributions. We use existing algorithms for creating and updating wavelet-based histograms as the core of our distribution mechanisms. Sources construct a wavelet-based histogram on their inputs and exchange the h largest wavelet coefficients with neighbours on the coordinate space, such that the value of h is negatively correlated with the distance between sources in the space.

The search coordinator then attempts to find partition boundaries in these pdfs alone. The search for boundaries occurs over the entire attribute domain, greedily selecting points of high gradient. We employ a thresholding technique defining the minimum width of partitions to restrict the number of partitions created. This threshold is based on the minimum separation of sources. The number of partitions created may exceed the number of sites present in the network. To account for this we group partitions based on the characteristics of the distributions within the partitions, and determine operator placements based on the contributions of sources relative to these groups.

Finally we turn to the placement of these partitions. Our strategy here is to use the hierarchical clustering algorithm described earlier. The key difference however is that instead of clustering on a coordinate space of network latencies, we are clustering on a projection of the join space. Specifically, we project the join space corresponding to moments computed on the output probabilities of the attribute values in a given partition. Repeating this process over all partitions constructs our routing forest, connecting all sources to our network deployed operator.

5. Adaptive Partitioning, and Replication

Now, we briefly discuss some of the issues our mechanisms will have to address, to be effective in dynamic, long-running distributed systems. We focus on the question of adaptivity, describing our requirements on operators for adaptive partitioning, and distribution models to cope with varying data distributions over time.

5.1. Maintaining Partitioned State

In order to support dynamic repartitioning, where we may acquiesce operators or partition them further, we rely upon operator implementations supporting these semantics on their specific states. In addition to function calls supporting the bootstrapping, and serialization of state (as found in the Flux system [24]), we require functions to support the injection and extraction of state elements, stored as a side effect of processing inputs of *specific* attribute values. For example, in the case of incremental partitioning, these values correspond to the attribute values of tuples to be routed to the new operator instance. This requires identification of state elements, as part of the extraction process.

Another issue is the blocking effect of adapting the number of partitions. In order to ensure complete processing of inputs during the transitional phase, existing schemes employ buffering techniques. In a wide-area system, modifying the deployment of partitions requires interaction with the underlying content-based routing layer. During the transition period of a partition, the routing layer must support the necessary buffering, raising the question of how to efficiently support distributed buffering, with the added information of the eventual destination of the buffer.

5.2. Time-Varying Value Distributions

In the algorithms above, we measure models of the data, as inputs to our framework, to widely partition and distribute a single operator. In a long-running system, with the semantics of an infinite input stream, leveraging the entire history of tuples may not form a strong basis for the model, especially with distributions that evolve, and change over time. Instead, we may wish to be more selective in the use of historical inputs, choosing relevant tuples based on either a temporal or semantic criteria (e.g, an age function, or by the value's statistical significance over a window).

This selective use of history must coexist with updates to the model in an online manner, as tuples arrive. Recent work on approximate summaries of streams (e.g. sketches, wavelet-based histograms [11]) have highlighted the necessity for single-pass, constant time methods of updating models. Unlike these works, we are leveraging an input model to guide our optimization algorithms, and not to actually approximate the query itself. Given updates to the model coincide with query processing, we envision another optimization problem, investigating the cost of introspection during query processing as its objective.

6. Related Work

The most prominent work on partitioning operators lies in parallel databases research. This literature encompasses topics such as hash-based joins [9], handling workload skew [10], and spatial joins [21]. While all of these works provide a plethora of partitioning techniques and join evaluation algorithms, to the best of our knowledge, none consider the effects of heterogeneously distributed data access, across a wide-area network, to support the operation. This lack of access locality can adversely affect the expected tuple delay.

The presence of a slow (and costly) medium for data access has been investigated in sensor networks. Systems such as Cougar [27], and TinyDB [17] have investigated query processing techniques and both energy and bandwidth optimization mechanisms. Data centric and geographic routing techniques have also appeared in sensor networks [15, 16, 23]. Finally the use of probabilistic models has been proposed to in the context of acquisitional, and distributed inference problems ([8, 20]).

Operator placement itself has been approached in both the distributed stream management domain [3], and the sensor networks domain [5]. These works represent initial steps in our vision of our networked query deployment, since they do not approach the tasks of partitioning or replicating an operator, for the sake of improving network latency.

Note that unlike most approaches in both the parallel systems and sensor networks communities, the discrete optimization problems we have described revolve around system deployment. In both parallel databases, and sensor networks, every site is assumed to be utilized as either a source, or a processing site and thus involved in query evaluation. This is not the case in our infrastructure model.

7. Open Issues

There are several directions we have not even begun to incorporate into this model. Recent work on modelling streams as time series, and capturing the temporal nature of streams in detecting bursts and distribution changes [14] could be leveraged for optimization purposes. Work on self-similarity to construct generative models of time series data could guide the prediction of input tuples, potentially leading to proactive optimization of networked joins.

We also briefly highlight our implementation plans for the ideas discussed here. Our networked join implementation will use the Borealis query processing engine for evaluating the query, while the SAND data management framework will integrate the probabilistic models and distributed optimization algorithms to solve the partitioning and replication problems.

References

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, Jan. 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 2003.

[3] Y. Ahmad and U. Çetintemel. Network-aware query processing for distributed stream-based applications. In *VLDB*, 2004.

[4] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, 2004.

[5] B. J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *IPSN*, 2003.

[6] S. Chandrasekaran, A. Deshpande, M. Franklin, and J. Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, Jan. 2003.

[7] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, Portland, Oregon, USA, August 2004.

[8] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model driven data acquisition in sensor networks. In *VLDB*, Sept. 2004.

[9] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsaio, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.

[10] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, 1992.

[11] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, pages 79–88, 2001.

[12] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, Sept. 2003.

[13] B. Karp, S. Ratnasamy, S. Rhea, and S. Shenker. Spurring adoption of DHTs with OpenHash, a public DHT service. In *IPTPS*, Feb. 2004.

[14] D. Kifer, S. Ben-David, and J. Gehrke. Detecting changes in data streams. In *VLDB*, Sept. 2004.

[15] J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Mobicom*, 2000.

[16] X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multidimensional range queries in sensor networks. In *Sensys*, Nov. 2003.

[17] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An acqusitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 2005 (to appear).

[18] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, Jan. 2003.

[19] T. S. E. Ng and H. Zhang. A network positioning system for the internet. In *USENIX*, Boston, USA, July 2004.

[20] M. Paskin and C. Guestrin. Robust probabilistic inference in distributed systems. In *UAI*, July 2004.

[21] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, 1996.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[23] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: a geographic hash table for data-centric storage. In *WSNA*, 2002.

[24] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, Mar. 2003.

[25] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, 2003.

[26] N. Tatbul, M. Buller, R. Hoyt, S. Mullen, and S. Zdonik. Confidence-based data management for personal area sensor networks. In *DMSN*, Aug. 2004.

[27] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, Jan. 2003.