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*— This position paper addresses the distributed evaluation of join operations across a heterogeneous network. We first argue that distributed joins arise naturally in a variety of networked applications. We then discuss the key challenges involved in the distribution, namely how to partition the join operator and how to place the resulting partitions on the network, in order to minimize output latencies. We then sketch solution strategies that leverage information about input value distributions and network characteristics. We conclude with some implementation issues and open problems.**

I. Introduction

Many emerging distributed systems applications need data routing or indexing functionality at the application layer. In particular, a common requirement of many applications is that of establishing *rendezvous points*, specific network points where portions of application functionality can be locally executed. Such rendezvous points are used to "match" data and/or queries injected from different network locations.

Consider the following application scenarios:

- In many data-centric sensor networks applications (e.g., distributed storage [17] and indexing [12]), lookup operations or query requests can be answered without flooding, if both sensor data and the requests are routed to the same rendezvous locations.
- In massively multiplayer online games (MMORPGs), a publish/subscribe model can be effectively used to perform distributed state management, whereby each player receives only the events that fall into her "area-of-interest" that refers to the portion of the virtual game world with which the player can interact. For example, in most current games, the virtual world is statically divided into a set of zones with players receiving all events in their current zone. An alternative more efficient approach is to adjust the zones dynamically, primarily based on the players' virtual locations. In either case, each player need only interact with servers that maintain the relevant zone(s).

We argue that the distributed matching functionality required by these applications can be generalized and abstracted as a distributed join operation that takes inputs from the read and write stages of the application pipeline. For instance, in the publish/subscribe scenario, 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 as a join of the "events" relation and the "subscriptions" relation. The key benefit of the join abstraction in this context is that this operation can be optimized by leveraging application semantics, encapsulated as a variety of statistics such as input rates, data-value frequencies, etc.

In this paper, we introduce a framework for the continuous evaluation of a join operator in a highly-distributed manner. Our primary optimization criterion is the delay of an output tuple, produced by a successful match on two input tuples. We leverage two key characteristics in our study – *network locality* and *data locality*. Here network locality refers to the proximity among the network locations of the input sources. Data locality refers to the similarity among the data sources in terms of the input values produced, and the frequency at which these values are produced. The latter also captures temporal properties of the inputs, such as the synchronicities of the input values.

We initially describe a placement primitive, needed for any deployment algorithm. This primitive tackles the basic issue 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 solution strategy considering data locality in probabilistic value distributions for each input 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 join replication mechanism that utilizes our basic 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, our solution strives to reflect trends in each data source's value distribution, as corresponding trends in the networked execution of the join operator.

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 process, 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 [19], Bamboo [10], and CAN [16], provide lookup functionality, serving primarily as an index for stored data. The PIER system [9] discusses several techniques for widely distributed joins, yet uses these DHTs as an unstructured

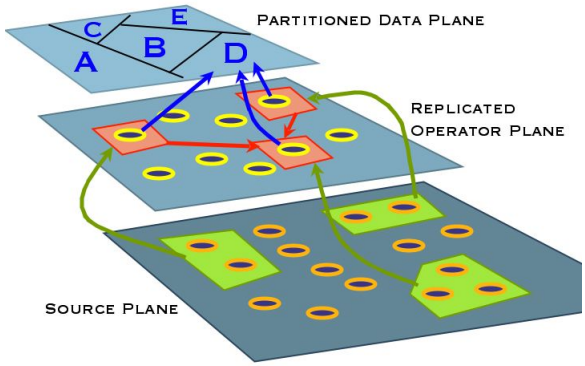

Fig. 1. Networked join evaluation overview: to support a distributed deployment of a join operator we i) partition the data space, ii) assign replicas, connected in a tree, to process tuples in each partition. Sources may then send tuples to their nearest replica, via a content-based routing mechanism.

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. Other relevant systems, such as Mercury [3] and DIM [12], have investigated efficient support for multi-attribute range queries in networked environments. These efforts primarily focus on load distribution and balancing, and do not consider leveraging the location *and* data similarity trends.

In contrast to the previous efforts, 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 and efficiently route their tuples to be joined. The rest of the paper discusses this networked join model, outlining the key challenges and our initial ideas towards an efficient and practical solution.

II. Networked Join Model

In the rest of the paper, we assume an infrastructure model consisting of a substantial set of heterogeneous network hosts, connected through a wide-area network. However, our basic approach and principles are general and applicable to other settings (such as wireless sensor networks) as well.

Our processing follows a continuous query model, as found in stream processing systems (e.g., Aurora [2], Borealis [1]). Under this model, a continuous 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.

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. Hence, we propose creating two *instances*

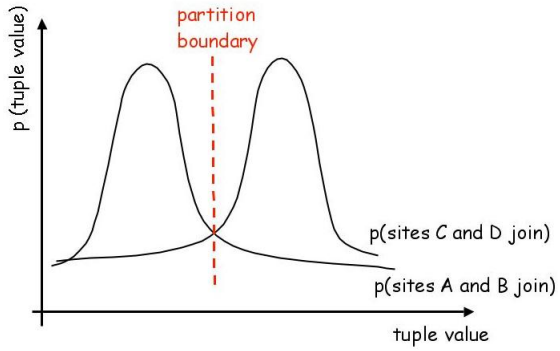

Fig. 2. Simplified, motivating example: we propose exploiting data locality. In this scenario, where sites A and B join with high probability on "low" values, and sites C and D join over "high" values, we create two partitions, and place each partition to optimize for its relevant sources.

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.

In the rest of this section, we first describe the behaviour of our join operator, in terms of input and output rates, and then introduce a probabilistic definition of join selectivity (inspired by [7]).

A. Modelling Data Sources

The join operator'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. The 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.

B. Probabilistic Join Evaluation

The 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 issue 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, the 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.

Given such a window distribution, we may obtain the expected output rate of a source pair as a product of (1) one source's input rate, (2) the same source's marginal distribution of join key attributes, and (3) 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.

III. Challenges in Deploying Networked Joins

We now outline three key problems in deploying networked joins, all targeted at minimizing the total expected network delay of a tuple. We address: (1) how to place an operator in the network; (2) how to identify localization opportunities, by differentiating sources based on the values they produce; and (3) how to select a set of sites to execute our join, as a resource allocation problem.

A. Operator Placement Problem

Our first challenge is to determine a well-suited location for a single join. This placement problem is present in any complex deployment problem, and necessitates a placement primitive. We leverage source probability distributions to determine this location. We assign each source a weight corresponding to its mean output probability, relative to the total for all sources. We refer to this weight as a source's *contribution factor*. We place our operator at a centroid of sources, ensuring that the contribution factor is decreasingly correlated to distance. Clearly, a site meeting this exact property may not exist in our network. Thus, we refer to our centroid as the site minimizing an error function, based on a sum of squares, of the difference in sources' contribution factors, and the fractional distance of sources from the centroid.

B. Join Partitioning Problem

Another challenge involves deciding how many operators to deploy. The issue here is to properly partition the join attribute value domain to minimize the total expected network delay of a tuple. Our grounds for posing this question is as follows. Each pair of sources will have differing expected output rates, across the join attribute value domain. By partitioning the value domain into subsets, we may choose different sites on the network to process tuples with different join attribute values. Provided we then tailor the selection of processing site for each subset, based on the network locations of sources most likely to produce join attribute values in our subset, we claim the desired end effect when compared to a single site processing the entire value domain. In short, the end effect is due to processing tuples with join attribute values that sources are likely to produce, at sites close to these sources.

This solution requires a content-based routing mechanism to route tuples according to the partitioning scheme. The server infrastructure must collaborate to maintain this routing mechanism as an overlay network. Recent literature [3] contains overlay network construction algorithms using a combination of short, and long distance links, without the use of a global hash function.

C. Instance Replication Problem

Following the partitioning of the join attribute value domain, we need to assign sites to evaluate operator instances. Replicating join partitions and executing them in concert present an opportunity to further optimize the expected tuple delay, as we describe below.

The replication challenge is to select sites that are well-suited to execute operator replicas. We consider a site well-suited if it is near to sources producing inputs that are likely to join. We view this challenge as a task to construct a *replica graph*, whose vertices are the sources and the replicas. The graph's edges denote the connectivity of sources to replicas, and replicas to themselves. Our graph must contain at least one root replica, defined as a replica reachable from all sources. Since this root is reachable by all sources, we may if necessary, compute the entire join at the root to ensure correctness. In our model, we place the root replica at the centroid of sources, as described in the operator placement problem.

Each input tuple may then join with other tuples, at any replica on the path between the source, and a root replica. We define the total expected delay as the sum, across all sources, of the expected delay between an input tuple, and each output it causes. The expected delay is a product of network distance from source to replica, and the probability of a tuple joining with state at the replica.

This problem is defined for a general replica graph which may include cycles and multiple root replicas. In the scope of this paper, we consider only tree topologies for replica graphs. We highlight two rules that each replica needs to follow to ensure correctness of the join. First, a replica forwards all tuples it receives towards the root, to ensure correct computation of the join. Second, a replica only considers joining tuples it receives on *different* branches of the replica tree. Overall, a tree topology has the benefit of lower total tuple duplication while forwarding along the replica path.

IV. Partitioning And Replication Strategies

We now briefly describe our initial ideas towards solving the challenges outlined above. First, we discuss our mechanism

for computing and exchanging pairwise source join probability distributions. We then move on to discuss how to determine partitions using these distributions. Finally, we sketch our strategy in selecting source groups for our tree-shaped replica graphs.

A. Probability Distribution Exchange Protocol

Our sources exchange their local models using an epidemic protocol to generate pairwise join probability distributions. This allows our search algorithm to operate in the pairwise probability space, in terms of which our objective function is defined. To improve convergence in this protocol, we propose the following diffusion heuristic. Each source maintains its model as a histogram, and sources order histogram buckets by decreasing value probability. Sources gossip in rounds along this ordering and selectively forward using local probabilities as bounds on pairwise probabilities. The intended effect here is that only pairs of sources with high join probabilities will exchange the majority of their buckets, while sources with low join probabilities will not gossip much.

B. Domain Partitioning Protocol

We now outline a strategy for selecting subsets of the value domain, to solve the aforementioned join partitioning problem. Our goal is to differentiate the sources by their output distributions, as greatly as possible, and optimally place our instances for similar groups of sources.

The distribution exchange protocol described above enables a search algorithm to operate in a search space of pairwise output probability distributions. Our search for partitions attempts to find subsets of the value domain, so as to maximize the difference in pairwise output distributions over all possible combinations of sources pairs.

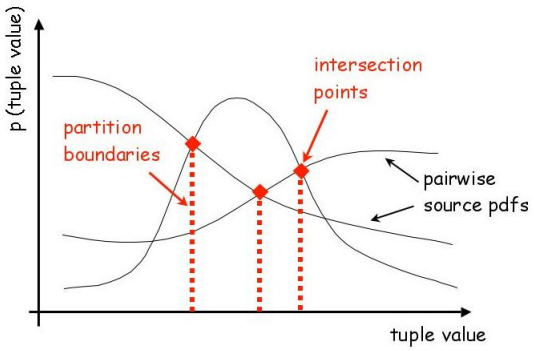

Fig. 3. Partitioning protocol overview: we select our partition boundaries at the intersection points of pairwise source pdfs, in a progressive manner starting at the intersection with greatest probability.

Our first step is to compute the intersection points of pairwise output distributions in a progressive manner. The intuition here is that subdividing partitions that do not contain intersection points will not lead to any further maximization of the total pairwise difference. One simple heuristic for computing intersections in a progressive manner is in decreasing order of the output probability of the intersection point.

Next, we collapse neighbouring partitions, meeting a specific criteria, into a single partition. This criteria requires computing the total gradient of all pairwise distributions within each partition. We choose to coalesce neighbouring partitions with negative total gradient. Partitioning the distribution from its local minimum or maximum enables us to consider two distribution segments independently. For a partition containing a distribution segment of entirely negative gradient, its absolute value ensures a positive contribution to our objective.

Our final step ranks our potential partitions in increasing values of total gradient. Given an upper bound on the number of partitions we may select, m, we choose the first $\frac{m}{2} - 1$ partitions in our ranking, and coalesce the remaining partitions between the selected ones. The upper bound we mention captures our claim that partitioning the value domain too finely will not yield any further optimization. Determining such a bound would prove useful as a termination criteria in our search.

C. Operator Replication Algorithm

Once we have our partitions, we propose the following strategy to solve the instance replication problem. First, we scale pair-wise output distributions by the network separation of the appropriate source pairs. Our algorithm attempts to select groups of sources within this multidimensional space. We prune the search space using a heuristic based on network distances. This heuristic eliminates pairwise distributions, whose sources are separated by a distance greater than the distance between either source and the centroid (of all sources).

Our algorithm then attempts to approximate a Voronoi diagram of this multidimensional space over the point set of all sources and the site representing the centroid of sources. For intuition, this procedure attempts to capture sources that are likely to join, and are nearby to each other. We use the resulting cells in our Voronoi diagram as our source groups. We construct our replica graph as a spanning tree connecting each cell. The root of this spanning tree is chosen as the cell containing the centroid of all sources. The root's children are chosen as the cells neighbouring the root's cell. This process continues until our tree spans all cells. Our algorithm then instantiates replicas attempting to place each replica by as a centroid for each cell.

V. 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.

A. Maintaining Partitioned State

In order to support dynamic repartitioning, where we may acquiesce operators or partition them further, we plan to 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 [18]), 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 transition, 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 thus has to support the necessary buffering.

B. 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 [8]) 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.

VI. Related Work

The most prominent work on partitioning operators lies in parallel databases research. This literature encompasses topics such as hash-based joins [5], handling workload skew [6], and spatial joins [15]. 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 [20], and TinyDB [13] have investigated query processing techniques and both energy and bandwidth optimization mechanisms. Data centric and geographic routing techniques have also appeared in sensor networks [11], [17], [12]. Finally the use of probabilistic models has been proposed to in the context of acquisitional, and distributed inference problems ([4], [14]).

VII. Concluding Remarks

We introduced a framework for in-network join evaluation and discussed pertinent algorithmic and architectural challenges. We are currently detailing the initial solutions outlined here, guided by a network game and a wireless real-time collaboration application. We plan to deploy these applications on top of the Borealis [1] distributed stream processing engine.

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 (to appear).

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

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

[5] 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.

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

[7] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *SIGMOD*, 2001.

[8] 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.

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

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

[11] 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.

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

[13] 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).

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

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

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

[17] 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.

[18] 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.

[19] 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.

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