Connectivity-Tolerant Query Optimization Over Distributed Mobile Repositories*

Sharma Chakravarthy, Aditya Telang†, Mohan Kumar
Mark Linderman‡, Sanjay Madria§, Waseem Naqvi¶

Department of Computer Science and Engineering
University of Texas at Arlington
Arlington, TX, USA

sharma@cse.uta.edu, adtelang@in.ibm.com, kumar@cse.uta.edu,
Mark.Linderman@rl.af.mil, madrias@mst.edu, Waseem_Naqvi@raytheon.com

ABSTRACT

Query processing and optimization in centralized and distributed environments is well-researched. Centralized query optimization focused on minimizing the number of input/output (or I/O) from disk. Distributed query processing focused mainly on maximizing local computation and minimizing data transfer between nodes. Here the distribution of data was pre-determined and both connectivity and bandwidth were pre-defined *and* guaranteed. Work on sensor data acquisition deal with non-join queries without taking mobility and connectivity interruptions into consideration. However, these assumptions are no longer true when queries are executed over repositories stored in mobile aerial vehicles which collect, process, and store data in real-time, and connectivity changes significantly over the duration of interest. Currently, only data in one vehicle can be queried by the ground control.

This paper explores query processing and optimization issues along with concomitant metadata needed for processing/optimizing queries over distributed, mobile, connectivity-challenged environments. Since response-time and fault-tolerance are the main focus, we propose plans using join, semi-join, and replication-based approaches. We propose and evaluate several heuristics for this environment ranging from greedy to cumulative approaches along with the use of replicated copies of data. We have performed elaborate experimental analysis to validate heuristics that work well for this environment. As maintaining replication is a challenge in this environment, we summarize our initial approach. This work on connectivity-tolerant query optimization is part of a larger middleware-based, service-oriented architecture.

1. INTRODUCTION

As part of a larger effort on distributed middleware-based architecture for fault-tolerant computing over distributed repositories, we address query processing and optimization in this paper. A brief description of the larger problem is provided for understanding the context for this work.

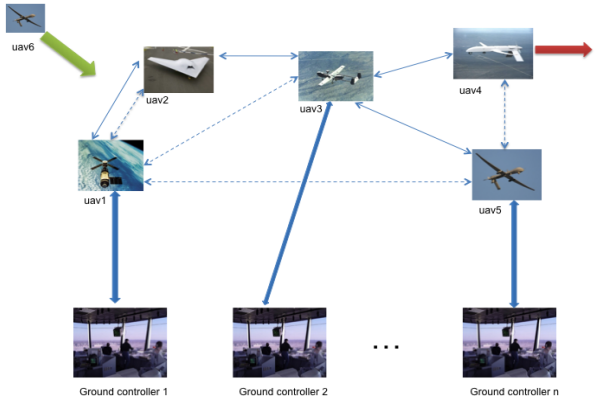

Figure 1: Example of Nodes and Connectivity

The general problem can be stated as follows: Consider a number (2 to 15) of nodes (unmanned aerial or other vehicles termed UAVs in this paper, and ground operators) whose connectivity is dynamically changing, and whose data bandwidth can vary from low to high. In this setting, how do we accomplish a specific task (query, search, subscription notification) that uses data and services from multiple nodes (for computation or collaboration) that are subject to QoS (Quality of Service) requirements (e.g., time to first result, response time). In other words, each node is independently acquiring multiple/different data types (e.g., location, telemetry, and images) and storing them locally. The data is stored in the form of Managed Information Objects

*AFRL has approved this work for public release; distribution unlimited. Case No. 88ABW-2012-5499
†IBM Research, Bangalore, India
‡Air Force Research Labs, Rome, NY
§Missouri University of Science and Technology
¶Raytheon Corporation, MA

The 18th International Conference on Management of Data (COMAD), 14th-16th Dec, 2012 at Pune, India.

(or MIOs) and can be sent on-demand to ground operators, and others nodes based on connectivity. There is also a need for combining (or joining) data from multiple nodes to get a better understanding of the overall situation. Data stored in a node is defined using type, metadata, and the payload. The communication between the nodes is through RF or satcom or other types of links (e.g., Link 16). It is also assumed that the nodes can be of different types based on processing capacity, storage, types of data it can collect, up/down link bandwidths, and latencies of data transfer. Nodes can also play different roles (depending upon the resources available onboard): i) collect data and forward it, ii) collect data, processes it, and forward both collected and processed data, and iii) collect, process, store/hold, and forward data. A single node can play different/multiple roles for different types of data. Their roles may change over time.

A typical scenario consists of a number of Airborne platforms (UAVs, Helos, Fighters, AWACs, etc.) which are traveling at various speeds (100, 200, 500 knots etc.), some in formation and some on independent tracks. Each has an associated *ground platform* that are either stationary or moving. Stationary platforms have semi permanent positions whereas Mobile ones may be on vehicle or foot. The connectivity among all airborne platforms is intermittent based on distance, line-of-sight, obstacles, cloud coverage etc. The transmission bandwidth is different for receiving and sending and depends on a number of factors such as distance, orientation, obstacles along the path etc. Each node (an airborne platform) has storage that is meaningful for the node type. Although computing power varies from node to node, we can assume that it is sufficient to run a local database management system (DBMS). Relational DBMS is assumed. Power is assumed to be a non-issue in this work because these platforms, we were told, have enough juice for the duration of the mission. This general scenario arises in various contexts:

- Disaster management, such as flooding, hurricanes, and evacuation. Information needed: evacuation routes, extent of damage, view of the area affected.
- Cooperative Combat Air Patrol. Mixture of UAVs, manned fighters, and AWACs cooperatively defending a region. Information needed: Signals, Lines of Bearing, contact positions, tracks

The scenario described above is illustrated in Figure 1. It is assumed that ground controllers are always in contact with their respective UAVs. Connectivity of other nodes (or UAVs) depends on dynamic factors. The connectivity (or disruption) of the nodes changes dynamically in this scenario as illustrated by solid lines and broken lines. Each line type (solid or broken) represents a different configuration of the network – one partitioning the nodes into two graphs and the other maintaining reachability for all nodes. The figure also shows new nodes coming into and existing nodes going out of the network.

Currently, it is only possible to process queries on data stored in a single airborne platform. Current state-of-the-art in distributed query processing assumes fixed, hard-wired connectivity among participating nodes. Replication is considered from an availability (of data) viewpoint and not from connectivity viewpoint. Latest work in sensor query processing [17] does not deal with mobile platforms with

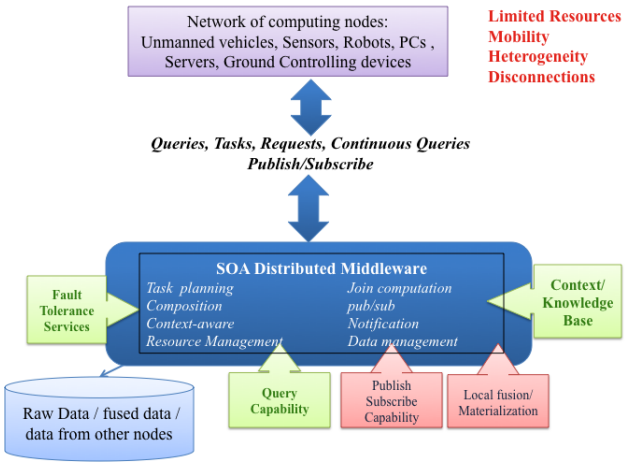

Figure 2: Pluggable Middleware Architecture

resident relational DBMS and intermittent connectivity. It is also possible to download data into ground nodes (from all nodes) and then process queries over those nodes. This results in delays that is not acceptable. Also, the data collected by multiple vehicles for a situation provides a holistic view and hence it is important to have the capability to issue queries *in real-time* that can be processed over all the relevant data available in one or more airborne platforms. Based on the requirements of the situations listed earlier (especially response time), it is important to have the capability to process queries over data in multiple nodes as they are being acquired.

The following are examples of queries that need to be executed on networked, distributed information sources.

1. Get all images taken within last 5 minutes of the area bounded by ⟨latitude1, longitude1⟩ and ⟨latitude2, longitude2⟩.
2. Get all SAM (surface to air missile) locations within 12 NM (nautical miles) of the area bounded by ⟨latitude1, longitude1⟩ and ⟨latitude2, longitude2⟩.

Since each node is autonomous and may belong to a group that needs to solve problems collaboratively, there is a need for two fundamental components that form the core of our overall approach:

- a common middleware component that is common to and present in all the nodes and a context (as a knowledge base or KB) that holds the capabilities, network configuration, and
- current state of the network at each node which is managed and used by the middleware.

The KB will also include the global requirements as capabilities of connected nodes are dynamically gathered. The context information can be customized/tailored either to a node, a task or for a set of tasks. The middleware heavily relies on the context to perform operations, knows the capabilities of self and other nodes, and to perform tasks collaboratively. A service oriented architecture (SOA) for the middleware is used to build larger systems in which this fits as a component seamlessly.

Figure 2 shows the service-oriented architecture (SOA) for the middleware for supporting query processing (and other services) over distributed repositories and accommodate fault tolerance. The overall architecture includes a middleware in each node that has a number of services (based on SOA) for collecting, managing, replicating data and metadata for the purposes of routing and query processing. Each node will have the SOA middleware and as many plugin components as needed. As a node collects data, it is stored in the repository on that node. Connectivity and replication information is periodically exchanged between nodes and stored in the context/kowledge base. For details on other services, please refer to [7] which contains an accessible url.
Contributions: Some of the key contributions of this paper are as follows:

1. Formulation of the distributed query optimization problems for ad-hoc connectivity in the presence of connectivity interruptions,
2. A cost metric that is different from traditional distributed cost metrics,
3. A query processing strategy with partial independent computations in different nodes, and
4. Replication of data for dealing with availability and incorporating it into query optimization

Overall, the novelty is in generalizing distributed query optimization to ad-hoc networks with mobile platforms, intermittent connectivity, and replication.

The remainder of the paper is organized as follows. Section 2 defines the problem being addressed in this paper. Section 3 discusses related work on query processing and metadata management. Section 4 discusses meta data used for query processing and its management. Section 5 briefly summarizes our replication strategy. In Section 6, we discuss our approach for processing queries, plan generation alternatives, and introduce heuristics appropriate for this environment. Section 7 has elaborate experimental analysis and their interpretation. Section 9 has conclusions.

2. PROBLEM STATEMENT

The focus of this paper is on processing SQL queries over distributed repositories collected/stored on each vehicle with the specified constraints on connectivity and reachability. This will allow for holistic queries without even having to know which repository contains what information. Although we are using SQL queries referring to relations in a node in this paper, a GUI can easily generate these queries from an interactive interface. It is meaningful to assume that each node has secondary storage of reasonable size. It is also assumed that each node has enough computing power to support a database management system (DBMS) that can process queries from its local (secondary) storage. A relational DBMS is assumed at each node.

The problem at hand is somewhat different from the traditional query processing that has been developed for centralized, distributed, and federated architectures. Although the computations/operators (e.g., join, semijoin) are the same as that of traditional query processing systems, the environment and the goals of these computations are quite different. Instead of knowing the schema, the data is published using a managed information object (or MIO) that needs to be used efficiently. An MIO (used to represent/encapsulate data) consists of: data type, metadata, and the payload. The metadata could be as simple as schema information or it can consist of additional information, such as range values, organization of data, number and types of objects in a picture etc.

Another difference is the need for replication of data – not from the viewpoint of local processing, but from the viewpoint of accessibility or reachability. Compared with earlier approaches where the nodes at which data was stored (or even replicated) were pre-determined, in the current scenario it is an important decision that has to be made dynamically by the system. As connectivity is not complete, multiple hops may be needed to reach a copy of the data. Furthermore, nodes can even store data that may not be directly useful to that node but is in close proximity for others that need it. When a node moves away (i.e., is not a neighbor anymore), there is a need to decide whether to keep the copy in that node or not. The utility of data and its copies need to be optimized using some metric (or a combination) such as cost of storage, cost of communication, time for data transfer, and longevity of storage.

In this paper, we address the problems of: query plan generation for this environment, relevant heuristics that are meaningful for this architecture, and use of replication for improving query processing. A prototype implementation developed in Java is used for extensive experimental results that validate our approaches and inferences.

3. RELATED WORK

Traditional relational database management systems (DBMSs), consisting of a set of persistent relations, a set of well-defined operations, and highly optimized query processing and transaction management components, have been researched for over several decades and are used for a wide range of applications. Typically, data processed by a DBMS is less frequently updated, and a snapshot of the database is used for processing queries. Abstractions derived from the applications for which a DBMS [5, 1, 15, 12, 18] is intended, such as consistency, concurrency, recovery, and optimization have received a lot of attention.

Query processing is a key consideration in database management systems. For this reason, query optimization has been one of the most active research areas since the advent of relational DBMSs. The acceptance and success of relational systems can be attributed largely to advances in query optimization over several decades [19, 11]. A major advantage of relational systems over earlier technologies is that the users of a relational DBMS are relieved of the need to describe their queries procedurally. More important, users are not required to understand the details of physical representation and its impact on queries posed to the DBMS.

In a distributed (or even a multi-database) environment, queries are decomposed, and query fragments are directed to particular sites (or databases) for processing [3, 2, 16]. Distribution of the database reduces the size of the data stored at each node, increases the locality of reference for the queries processed at a given node. Replicated databases provide an additional opportunity – that of choosing the site (at which a subquery is sent for processing) to increase the probability of overlap with other subqueries. Hence, queries processed at a site may have a lot of overlap of the data they

access.

Other forms of query optimization, such as semantic query optimization [9], multiple query optimization [8, 20], and more recently, continuous query processing [6] have focused on modeling, scheduling, and load shedding strategies. The work presented in this paper is related, but is distinctly different from them. In this work, queries are not transformed using semantics, multiple queries are not batched and optimized, and continuous query processing techniques deal with a different set of metrics and their optimization is very different from what is required for this scenario.

Several middleware architectures have been developed in the recent past to support mobile ad hoc networks (MANETs), sensor networks, and pervasive systems. Boulkenafed and Issarny develop a comprehensive middleware for data sharing in MANETs [4]. The focus of the work however is minimizing energy consumption. Kalasapur et al. developed an elegant middleware for service provisioning in pervasive systems with mobile nodes [13]. Tamhane and Kumar have developed a resource management mechanism for pervasive systems with underlying ad hoc networks [21]. None of these works consider dynamic networks such as that of UAVs, where node mobility is a regular feature rather than a rarity. Christman and Johnson discuss a customized self configuring architecture designed for UAVs [10]. However, they do not deal with on content sharing and query processing. The middleware architecture proposed in [14] attempts to address this important issue in UAV based networks.

4. METADATA AND ITS MANAGEMENT

In order to process queries, minimal information about the schema, connectivity of the nodes, replication information (if any) as well as available cardinality and other statistics need to be available in each node. Furthermore, some of the above need to be kept current in this dynamic environment. At the core of our middleware is the use of graph theoretic and sub-graph matching techniques to ensure network status awareness and data access. A graph structure is created to capture the essence of data objects/services, corresponding computing nodes and the relationship among the data objects as well as the nodes. The associated middleware tools facilitate the response to queries in dynamic heterogeneous environment comprising mobile nodes. The proposed service provisioning framework is flexible in representing metadata and services, and adaptive to changing environments by incorporating the replicated copies. We assume the following information in the form of tables accessible to the local database.

Data at each node is assumed to be a relation with the schema shown in Table 1. R_{ij} corresponds to relation R_i at node j. R_{ii} $(i = j)$ will be used to represent the *primary copy* of a relation at node i. R_{ij} (i <> j) will be used to indicate the replica of R_i in node j. A field 'TimeOfUpdate' is maintained for each update that happens over the Meta Data to estimate the accuracy of data and keep a track of how recently the update has been done.

A number of additional information about the characteristics of each R_{ii} is maintained in a node i (and periodically propagated to all other nodes) for the purpose of query plan generation and cost estimation. If a relation R_i is replicated at this node (j), then for each replicated relation R_{ij}, we need to maintain the same information as in Table 1. The difference is that this information may not be current. Every node maintains a copy of its original relation that is stored at some other node. Currently, replication is assumed to be a single copy and complete for each relation. Network Managed data is maintained and updated by the middle-ware, and accessed for processing by the local query processor for executing intermediate steps of a query plan.

Selectivity for simple and composite conditions are calculated using standard formulas [19, 16] based on the information in Table 2.

A Relation-to-Node mapping table, as shown in the Table 3, is maintained by the message management system at each node which indicates the location of the original and the replica of a Relation. A value of 0 in the replica node column indicates that the replica is not complete at this point in time and hence is not considered for generating a query plan.

Name	Original Node	Replica Node
R1	1	4
R2	2	1
...	...	...
Rn	N	k

Table 3: Relation and Replica Locations

Finally, a Connectivity map is maintained at each node which checks for the existence of a connection between any two nodes and the corresponding bandwidth between them. If the Received Signal Strength (RSS) is zero or below a threshold, then the connection is considered to be 0 and 1 (or present) otherwise. RSS value lies on a scale of 1 to 10. The actual RSS value is used in cost estimation. LSF (Link stability Factor) is a function of rate of change of RSS value over a period of time. LSF, to some extent, measures the stability of the link over a period of time. This is important as the plan is generated once and the execution of steps take some time. A pair is considered for the plan generation if the RSS value at the instant is 1. A sample connectivity map is shown in Table 4. *Note that bi-directional connectivity is maintained as the bandwidth is different between uplink and downlink.* See [14] for network related issues.

5. REPLICATION STRATEGY

In order to ensure accessibility and fault-tolerance, each data object is replicated on other nodes. Currently, there exists only one replica of a given data item. N_s represents the source node, where the original copy of data item D_i was acquired. N_c is the candidate node that will contain a replica of data object D_i. When N_s decides to replicate its contents on another node N_c, a node from the set of the nodes that are immediate neighbors of the source node is selected as candidate nodes for replication. Immediate neighbors are those nodes which are directly connected to the source node. The source node tries to replicate all its tuples on the chosen candidate node. For each of the above selected candidate nodes, a cost function $C(s, c)$ is computed. The node with the lowest cost is selected as a candidate for replication. The cost function to determine the candidate node for replication is dependent on the following factors: $Bandwidth$ defines the closeness of N_c from N_s in terms of bandwidth. Greater bandwidth is desirable; $Linkstability$ is a measure of stability of the link between nodes N_s and N_c. Greater stability of the link between the two nodes implies better longevity; and greater $Degreeofthenode N_c$ indicates

better accessibility of replicated data. Additional details can be found in [14].

6. QUERY PROCESSING AND PLAN GENERATION

Although it is tempting to try to optimize a query from scratch as is done traditionally, we need to take the environment and constraints into account for proposing an appropriate solution. The focus here is to generate a query plan that can complete the execution of a query with minimal data transmission cost and good response time. Hence, a plan generator that tries minimize I/O in each node is not the best way as the local DBMS is likely to do a better job; and we need to leverage that. Hence, we decided to delegate local optimization to the DBMS at each node and concentrate on a plan that minimizes data transfer (or data movement) for processing a query. As a result, a query plan for this scenario is envisioned as numbered sequence of plan steps that can be easily interpreted and executed at any node[1]. Table 5 gives a description of a plan format. Each step includes the operation to be applied, the data items involved, the node where it is applied, the name of the result and the node where it is created.

Unlike traditional query processing, the plan needs to be sent from node to node[2] (or partial plans generated at each node which is not considered in this paper) for the purposes of query processing. A counter, as part of each plan, indicates the next step to be executed and is initialized to 1. An example of a query plan is shown in Table 6.

The plan format described above is sufficient to describe any arbitrary relational query plan involving selects, projects, and joins (also known as an SPJ query). The above format can also accommodate SQL aggregate operators, such as a SUM, COUNT, AVERAGE, MINIMUM, and MAXIMUM. A query is executed as follows. A complete plan is generated at the node where the query is received using the metadata stored in that node. The plan is then sent to the node in which the first operation takes place (if it is different from the node where the query plan is generated) along with the plan step counter. The interpreter in that node uses the plan step counter to execute as many steps as possible in that node. When a move or copy is encountered, it sends the data as well as the plan (actually the remaining portion of the plan to reduce the amount of data transferred) to the next node. This process continues until the last step of the plan is executed. The result of the query will always be sent to the node at which the query was received.

Currently, a complete query plan is generated as follows. Each node in the architecture has the same query plan generator and uses *only the Metadata in that node*. Note that the metadata is updated by the underlying mechanism briefly indicated in Section 4. The query plan is constructed one join/semijoin at a time. Costs of partial plans are computed using well-defined statistics and formulae for computing selectivities for conditions and join. The lowest total cost query plan is used as the final plan after the plan space is explored either exhaustively or using heuristics. This will result in a good plan (or an optimal plan). Several heuristics are explored as part of this project to reduce the total computation required for generating a plan and still generate a good plan[3]. These heuristics are compared experimentally with respect to replication and connectivity scenarios.

The complexity of the optimal plan generation is k^n where n is the number of joins and k is the number of alternatives for each join. Currently, k being used is 18 (three alternatives for join, semijoin, & hybrid alternatives, and the same using replica as well). Note that this is at the logical level. For each logical join alternative, there will be many physical alternatives making the plan space significantly larger. Assuming three joins, we need to explore 5000+ alternative query plans and compute cost for each one of them. For plans with more than three joins, this exhaustive approach is not viable. Hence, we have incorporated some heuristics to limit the number of plans generated by pruning plans carried forward after each join. A query optimizer has been implemented to validate the heuristics and their effectiveness on synthetic data and multi-join queries that simulate actual data sets.

Cost for our plans is mainly data transfer cost which in turn depends on the width of the tuple and cardinality of the relation (intermediate or otherwise). Hence it is important to estimate the number of tuples as well as their width. Statistics in the form of cardinality and domain characteristics are used for this purpose. Join and condition selectivity are inferred from the statistics maintained. Intermediate result sizes are also estimated as its accuracy is important as the choice of the best query plan is primarily based on the cost of data transfer based on availability of connectivity. The statistics used for evaluating the cost of a (partial) query plan is the same as the ones used in traditional and distributed query processing [19, 16]. All of these are well-established for the relational model. We do not include the processing cost for the operation/plan, but only the data transfer cost. Processing cost depends upon the availability of index and other structures and mainly influences the order of join (which we take into account in our plan generation process). As future work, it will be useful to explore what access structures are meaningful and take the processing cost into account as well. In each node, the plan can be executed by converting it into an SQL statement if a relational database is used for storing data in that node.

To improve the accuracy of selectivity, for each attribute of R_{ii} on which a condition has been applied, selectivity information is maintained as follows. Table 7 reflects the **actual** selectivity values for conditions on that relation and will be used when the same or similar condition is encountered in a later query. Otherwise, selectivity formulas are used for calculating the resulting relation cardinality. The conditions are maintained at the component level using a hash table which can be associatively searched using the relation and condition. The intermediate relation cardinality and width are also maintained.

[1]In fact, we assume that at each node, plan steps are combined to generate an SQL query to be processed locally accessing *only* local data.

[2]As an alternative, it is possible to simultaneously send the entire plan or preferably portions of the relevant plan steps to each node. If this alternative is used, a synchronization mechanism is needed to execute plan steps in the correct sequence without any need to transfer plans. It is also possible to dynamically generate plan steps at each node when needed rather than generating the entire plan to start with.

[3]Note that, in general, the objective of query optimization is not as much as generating an optimal plan by spending a lot of resources, but to certainly avoid *bad* plans and do it fast.

Relation	C1	C2	C3
R1	0.2	0.5	
R2		0.6	0.67
R1	0.9	0.1	0.7

Table 7: Selectivity Table

6.1 Plan Generation Implementation

The query plan generator is implemented in Java. A relational database is used for storing metadata (as will be done in each node). A constants Java file is used for conducting experiments and to setup parameters for varying connectivity and replica information (as shown in Figure 3). An interactive option is also available to input query, load metadata from a file, and analyze individually best, worst, or any plan generated. For details of implementation refer to [7].

The generator begins by generating all distinct partial plans (from an initial empty set) for each join. As an exhaustive algorithm, it generates 18^n plans for a query containing n joins. It is evident that this approach is not viable beyond a few joins. This is being done so that we can compare heuristics-based plans with the optimal ones to analyze the effectiveness of heuristics we come up with (e.g., top-k in each iteration, top-k cumulatively, top-k for each type of plan.) for queries with fewer joins. The generator then iterates through the relation list and creates the necessary plan steps. Then all of the attributes required are projected on the output and join condition attributes to minimize the data transfer across nodes which form the bulk of the cost of query processing in this environment. Since most of the plans will use these initial select or project statements (to reduce the width and cardinality of the relation), these same statements are attached to every plan. For plan alternatives using joins the generator moves the required relations to the location of the join and then performs the join. Even for this, projections are applied to reduce the overall width and cardinality of relations moved. The plan class takes care of updating intermediary name, location, and condition information. Then the generator moves on to the next plan.

For plan alternatives using semijoins, the relation that will be semijoined to is copied and projected on the attributes used in the specific join condition to minimize data transfer. Then it is moved to the location of the semijoin. The semijoin is performed. When the semijoin step is added to a plan the plan updates name, location, and condition information and in the case of semijoins the output relation and the relation that still needs to be semijoined to finish the operation is added to a stack to keep track of the remaining semijoins (to generate chained semi join plans). Note that a join can be processed as a sequence of two semijoins. However, when multiple semijoins are performed in a sequence, the second semijoin needs to be performed in reverse order (hence a stack). The next plan is then processed. For multiple joins, after all of the plans have been processed with the first join, all of the joined relations will be projected on the remaining join attributes required and then algorithm will iterate through all the plans again performing the remaining joins and semijoins. After a relation has been joined, its current location is considered to be that of the result of the join even if it currently has a replica, which may cause some of the plans to be the same. After all cases have been exhausted, the algorithm goes through and finishes each case by iterating through the stack of remaining semijoins completing the remaining semijoins in reverse order and then moves the final relation to its output node. During each step of the plan generation, the cost associated with a move or a copy is calculated, if there is no direct connection between nodes the cost is considered prohibitively high and value is automatically forced to a very high level by using a very low bandwidth for the calculation. After calculation the plans can be viewed in sorted form. The plan generator generates a summary of: number of plans generated, lowest and highest cost plan numbers. It is possible to view any of the plans in detail. The same process is used for generating plans using heuristics except that a subset of plans are used in each iteration which are selected based on the specific heuristic.

The plan generator also includes a network component that generates the connectivity matrix using the seed provided. Each element in the matrix represents the cost of the link from node x to node y. Number of connections is also specified as part of the configuration. The connectivity matrix generated is consistent with the bandwidth assumptions for this scenario. The connectivity matrix is updated to simulate movements of the nodes.

6.1.1 Sample Best and Worst Plans

Consider a multi-join query that is sent to node 1 and the results expected back in node 1.

```
Node        TARGET 1
SELECT      *
FROM        U_1_D,U_2_D,U_5_D
WHERE       ((U_5_D.OBJTYPE=1))
        AND ((U_1_D.LAT>U_2_D.LAT))
        AND ((U_2_D.LONG>U_5_D.LONG));
```

Plan Number	Total Cost (in milli secs)	Remarks
253	**493.873**	alternatives 15 (first join) and 2 (second join)
263	585.942	alternatives 15 and 11 (only semijoins)
3	*175117.8*	alternatives 1 and 3

Table 8: Sample Plan Costs

Below, we present Lowest cost, semijoin only cost, and highest cost plans for the above query in Table 8 and additional information about how they were generated in terms of plan combinations for a network configuration. In the above the best plan seems to be a combination of join and semijoin. The worst plan seems to be made of only joins. As can be seen, the difference between the best and the worst plan is significantly large. Hence, it is important to choose plans closer to the *best* plan (i.e., a *good* plan).

6.2 Heuristics-Based plan generation

The purpose of generating an exhaustive plan space as indicated above is to demonstrate the cost differences between the best and the worst plans. The above algorithm is still not exhaustive in that it does not consider all possible join combinations. As can be seen clearly, there is a significant difference between the best and the worst plan. The goal of query optimization is not necessarily to choose the *optimal* plan, but to avoid bad plans and choose a *good* (closer to the optimal and far from the worst) plan.

During testing, we also realized that the connectivity plays a critical role in that if only one way connection is available between nodes, it impairs good plan generation as semijoin-based plans need to finish the second half of join by bringing the results back to that node. In order to generate a plan without exhaustive search of the plan space, we have proposed a number of heuristics to the above algorithm to compare their performance with the optimal plan. We use this prototype implementation to analyze various aspects such as connectivity, bandwidth, as well as selectivity to understand the types of plans generated and the effect of these parameters on total plan cost. We have identified the following heuristics to be useful and have implemented them so that we can compare them to the optimal ones to determine when and which heuristics to use for queries with more joins.

1. **Top-k Iteration:** Plan generation is iterative with respect to joins. For this heuristic, we choose top k (where k can be specified as a parameter) *lowest* cost partial plans in each round of expansion or iteration. Note that each iteration in our approach corresponds to processing a join. The number of iterations is equal to the number of joins. The intuition behind this approach is to use a greedy local selection and hope that it will turn out to be good globally as well. This significantly reduces the size of the explored plan space.

2. **Top-k Cumulative:** For this heuristic, we choose top k lowest *cumulative* cost (up to that point) plans in each round/iteration of expansion. Again, the intuition is that the cumulative cost up to this point is more meaningful (than Top-k-iteration, for example) and this would lead toward a good overall plan. Note that this and the above heuristic will be identical up to two joins. We expect this heuristic to do better than the previous one as the number of joins increase.

3. **Top-k Join-type:** For this heuristic, we categorize plans into join-based, semijoin-based, and hybrid (a combination of join and semijoin). we choose top k *lowest* cost plan from each category for expansion in each round. The Top-k join-type is a different type of heuristic as we have different types of partial plans and their costs are likely to be different. Here, k lowest cost plans from each type is chosen for the next round. In order to compare them in a fair manner, the k value need to be lower (1/3 as we have 3 join types) so that the same number of plans are carried forward in each round. Otherwise, this approach is likely to explore a larger plan space and do better than the other two heuristics.

In addition to the above, a number of other possibilities for plan generation exist: i) incremental plan generation, ii) looking ahead at connectivity and pruning plan alternatives, iii) getting dynamic cost information and then generating partial plans

Note that connectivity, in this context, is likely to play a significant role not only in the generation of a complete plan, but also its cost. If sufficient connectivity does not exist among the nodes that participate in the query (including the nodes that have a replica), a complete query plan may not even be feasible. The presence of replica increases the probability of generating a complete plan and if several exist, heuristics hopefully will choose a good one without having to generate all plans. A heuristic that incorporates connectivity would be very useful for this environment.

The above three heuristics have been implemented in our prototype. The software has two modes: interactive and experimental to make it easy to test and use. In the interactive mode, a query can be given at the prompt (or in a file) and a heuristic specified for its plan generation. The generator will indicate the number of plans generated as well as the lowest and highest cost plans (along with plan number). One can output (or look at) any plan in details by typing the plan number. It is also possible to provide a file input to process multiple queries in this mode. The selectivity and cardinality information is statically initialized. The connectivity is also initialized at the start of the system. This can be easily changed by loading a new or different relations and connectivity information before executing the plan generator.

In the experimental mode, the configuration is set using a Java Constants class (a sample is shown in Figure 3. The input consists of: number of queries to be generated, seed for query generation, number of connectivity configurations to be used in the experiment, seed for configuration generation, and connectivity factor. The generator has a random query generator on the schema stored in the system and generates the desired number of queries for which minimum and maximum number of joins can be specified. The seed is to ensure repeatability of experiments as well as generate a new sequence of pseudo-random queries. The same is true for network configurations and its seed. The connectivity factor is use to control the sparseness of the connectivity matrix. If there are n nodes, the connectivity factor can vary from 0 to (n-1), 0 indicating no connectivity at all and (n-1) indicating complete connectivity. The connectivity itself is generated randomly to satisfy the parameters specified.

The above setup allows one to perform different types of experiments. For each query, connectivity can be changed to determine how the plan cost changes and can also compare the optimal cost with heuristics-based plan costs. It is possible that due to the connectivity, a number of plans cannot be completed resulting in a high cost. Queries or connectivity sequences can be changed, independently, by varying the corresponding seed.

7. EXPERIMENTAL ANALYSIS

In order to test the effectiveness of the heuristics proposed for the query plan generator, we performed several experiments using these heuristics across different connectivity matrices and several different queries. The following Java interface (see Figure 3) was typically used for setting up parameters for all experiments.

A sample set of queries used for experimentation is shown in Figure 4. Two and three join queries with different selection and join conditions have been purposely chosen so that they can be compared with optimal results. This will force the execution of plan in multiple nodes and also brings in the use of replicated relations based on the connectivity. These queries were generated by domain experts who have experience in these scenarios. Finally, the cardinality for all relations ranges from 100000 tuples to 505000 tuples. This cardinality also represents the amount of data acquired during a mission. *We have presented tables instead of plots as the range of values from our experiments is quite large and*

hence is not conducive to plotting.

7.1 Comparison of Heuristics

In this experiment, we tested the five queries (shown in Figure 4) and tested the three heuristics along with the optimal algorithm on the same configuration of the connectivity matrix. Table 9, shows the cost (in milliseconds) incurred by the various approaches towards generating the top-3 best plans (average) for 5 different queries on the same connectivity matrix configuration. Based on the results, it can be observed that the plan generation process depends hugely on the connectivity between the nodes. Among optimal plans, the semijoin ones seem to perform better as expected since data transfer is reduced significantly. Further, amongst the different heuristic approaches, the semijoin approaches (either top-k-iterative or top-k-cumulative) appear to perform better for the current set of connectivity configurations.

7.2 Plans costs with and without replication

In this experiment, we compare the costs of generated plans with and without replication to establish the need for replication and its importance for this environment. In order to test in the *absence* of replication, we selected each query independently and altered the settings of Table 3 such that for the nodes involved in the corresponding query no replica existed. For instance, considering *Query 2*, we modified Table 3 such that replicas for nodes 5 and 10 did not exist in any other node. We then evaluated the optimal as well as heuristic-based plans for each query in the absence of replication, by averaging the costs obtained from the corresponding top-3 plans. We then enabled replication by creating replicas of the nodes, as shown in Table 3, and evaluated the costs in the presence of replication.

Table 11 shows for a specific query (*Query 3*) the costs obtained by each plan in the presence and absence of replication. It is clear that the processing cost without replication is significantly higher (as high as 6 times). We also wanted to understand the behavior of averages. Table 12 displays the average costs obtained across all five queries, when replication was present and absent. We observed that, in the absence of replication, it was difficult to obtain a low cost plan (due to the nature of the connectivity between the different nodes); as a result, a relatively high-cost plan has to be selected. In contrast, replication provides a distinct advantage as a low cost plan, involving the replica nodes can be obtained even though the connectivity between actual nodes involved in the query may not exist. Consequently, the presence of replication yields comparatively low-cost plans, and hence proves to be fruitful in such scenarios where the connectivity between nodes is dynamic and susceptible to frequent changes. This is for a single copy replication. It would be interesting to study the tarde-offs between number of copies and plan costs.

7.3 Impact of Connectivity on Plan Cost

In this experiment, we present a single query (shown below) and computed the top-3 plan cost using the heuristics proposed along with the optimal plan cost on six different configurations of the connectivity matrix. We had to keep the connectivity large; otherwise, no (or not many) plans were generated. Since the connectivity matrix is large in size, we do not show it here. Instead, we have displayed a sample configuration file earlier. We have done this experiment on several queries with similar results.

Method	Replication	No Replication
Optimal Join	63.76	175.73
Optimal semijoin	135.33	326.28
Top-K Cumulative Join	85.57	195.14
Top-K Cumulative semijoin	78.46	179.07
Top-K Iterative Join	70.76	379.54
Top-K Iterative semijoin	129.39	894.21
Top-K Join-type Join	80.76	391.72
Top-K Join-type semijoin	129.39	666.67

Table 11: Replication Vs. No Replication: Effect on costs for Query 3

Method	Replication	NO Replication
Optimal Join	8.91	29.41
Optimal semijoin	29.33	126.81
Top-K Cumulative Join	527.21	143.73
Top-K Cumulative semijoin	279.21	795.41
Top-K Iterative Join	33.11	177.14
Top-K Iterative semijoin	318.25	828.21
Top-K Join-type Join	801.49	935.72
Top-K Join-type semijoin	304.71	899.67

Table 12: Replication Vs. No Replication: Effect on costs across all queries

```
Query 1   target 2
SELECT    * FROM UAV_2_DATA, UAV_4_DATA, UAV_6_DATA
WHERE     ((UAV_2_DATA.NODEID=76)) AND ((UAV_2_DATA.LONG>=804))
          AND ((UAV_6_DATA.LONG<=540) AND
          ((UAV_2_DATA.LAT=UAV_4_DATA.LAT)) AND (UAV_4_DATA.LONG=UAV_6_DATA.LONG));
```

Table 10 shows the cost (in milliseconds) incurred by the various approaches towards generating the top-3 best plans for the given query. Based on the results, it can be observed that the plan generation process depends heavily on the connectivity between nodes. For many network configurations, no plan is generated even in optimal join case. However, amongst the different heuristics, the semijoin approaches (both iterative and cumulative) appear to do better and very close to optimal for the current set of connectivity configurations. However, determining the exact relationship between the type of join and the corresponding costs of plan generation will require further analysis and is beyond the scope of this paper.

7.4 Desiderata

It is very clear from the experiments that the proposed heuristics are meaningful and generate good plans that are not too far from the optimal without exploring the entire plan space. The presence and absence of replication makes a significant difference both for the number of plans available and the cost of the plan. This is only for directly connected replica. If multiple hops are included, reachability will be even better (at the cost of transmission cost). Connectivity of the network certainly plays a central role and more attention needs to be placed on heuristics and optimization to include predicted stability of network and its leveraging. Alternate plan precessing strategies will also be beneficial for this environment. As an example, parallel execution of

plan steps in different nodes is likely to reduce response time substantially.

8. ACKNOWLEDGEMENTS

Authors would like to acknowledge the support from Air Force Research Laboratory (AFRL) for this work. The work presented in this paper is partially supported by Air Force Research Laboratory (AFRL) grant. Authors would also like to acknowledge the contributions of Danny Hua, Nick Steffen, and Chance Eary on the implementation of the query optimizer prototype used for experimental analysis and all students who worked on this project from all participating institutions.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we have explored SQL query processing and optimization in distributed environments where connectivity is changing rapidly. Instead of optimizing the query from scratch, we have relied on local optimization and have used an incremental plan generation approach with several heuristics for processing a query at the granularity of joins and semijoins and concomitant data transfers. Replicated copies are assumed and taken into account in order to alleviate availability of data due to connectivity issues and increase the probability of an available copy during query processing.

A number of extensions are currently being investigated: i) optimum number of replicated copies instead of a single copy, ii) generating the query plan incrementally and dynamically (due to connectivity issues), iii) use of parallel plan evaluation with concomitant complexity to plan generation and evaluation, and iv) various QoS issues pertaining to query results.

10. REFERENCES

[1] Abraham Silberschatz and Henry F. Korth and S. Sudarshan, *Database System Concepts, 3rd Edition.* McGraw-Hill Book Company, 1997.

[2] P. A. Bernstein and N. Goodman, "The theory of semi-joins," Computer Corporation of America, Tech. Rep. Tech Report CCA-79-27, 1979.

[3] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, "Query processing in systems for distributed databases (SDD-1)," *ACM TODS*, vol. 6, no. 4, pp. 602–625, Dec 1981.

[4] M. Boulkenafed and V. Issarny, "Middleware service for mobile ad hoc data sharing, enhancing and data availability," in *ACM Middleware*, vol. 2672, no. 1. LNCS, 2003, pp. 6–25.

[5] C. J. Date, *An Introduction to Database Systems, Volume 2, Sixth Edition.* Addison-Wesley, Reading, 1995.

[6] S. Chakravarthy and Q. Jiang, *Principles of Stream Data Management.* Springer, 2008.

[7] S. Chakravarthy, M. Kumar, S. madria, and W. Naqvi, "A Distributed Middleware-Based Architecture for Fault-Tolerant Computing Over Distributed Repositories," *TR CSE-2011-8, UT Arlington*, Dec 2011, http://www.cse.uta.edu/research/publications/Downloads/CSE-2011-8.pdf.

[8] S. Chakravarthy, "Divide and Conquer: A Basis for Augmenting a Conventional Query Optimizer with Multiple Query Proceesing Capabilities," in *ICDE*, 1991, pp. 482–490.

[9] U. S. Chakravarthy, J. Grant, and J. Minker, "Logic-Based Approach to Semantic Query Optimization," *ACM Trans. Database Syst.*, vol. 15, no. 2, pp. 162–207, 1990.

[10] H. C. Christmann and E. N. Johnson, "Design and implementation of a self-configuring ad-hoc network for unmanned aerial systems," in *AIAA*, 2007.

[11] G. Graefe, "Query evaluation techniques for large databases," *Computing Surveys*, vol. 25, no. 2, pp. 73–170, Jun. 1993, (Survey Article).

[12] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Vol. II.* Computer Science Press International, Inc., MD 20850, 1989.

[13] S. Kalasapur, M. Kumar, and B. Shirazi, "Dynamic service composition in pervasive computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 7, pp. 907–918, July 2007.

[14] M. Kumar, M. L. Sharma Chakravarthy, Sanjay Madria, and W. Naqvi, "Middleware for Supporting Content Sharing in Dynamic Networks," in *MilCom2011, The Military Communication Conference*, November 2011.

[15] M. Stonebraker, Ed., *Readings in Database Systems.* Morgan Kaufman Inc., 1988.

[16] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[17] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1061318.1061322

[18] R. Ramakrishnan, *Database Management Systems.* WCB/McGraw-Hill, 1998.

[19] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," *Proc. of ACM SIGMOD Conference*, pp. 23–34, Jun. 1979.

[20] T. K. Sellis, "Multiple -query optimization," *ACM TODS*, vol. 13, no. 1, 1988.

[21] S. Tamhane and M. Kumar, "Middleware for decentralised fault tolerant service execution using replication in pervasive systems," in *IEEE PerCom, Sixth International Workshop on Middleware Support for Pervasive Computing*, March 2010.

Timestamp	Nodeid	Lat	Long	Obj_type	Obj_desc	Object_ptr
8 bytes	4 bytes	4 bytes	4 bytes	8 chars	Varchar (64)	Pointer (8 bytes)

Table 1: Relation Format

Attr Name	Type	Cardinality	Position	Width	Min Value	Max Value	Unique values in the range
Timestamp	number	1200	1	100	50	140	90
Lat	number	1200	2	4	10	100	90
ObjType	varchar	4000	3	64	20	350	330
ObjPtr	categorical	2000	3	8	Null	Null	10

Table 2: Relation Metadata

Nodei	Nodej	RSS	LSF	Bandwidth	Start-up Cost
1	3	1	5	100	10

Table 4: Connectivity map

Operation n	Parameter	Operand-1	Operand-1 Loc	Operand-2	Operand-2 Loc	Result Name	Result Loc

Table 5: Plan Format

Operation	Param	Operand1	Operand1 Loc	Operand2	Operand 2 Loc	Result Name	Result Loc
Select	A > 100	R1	1	Null	Null	R1'	1
Project	A1, A3, A4	R1'	1	Null	Null	R1"	1
Move or copy	Null	R1"	1	Null	Null	R"	2
Semi Join	A > C	R"	2	R2	2	SR1	2
Join	B = D	R12	2	R2"	2	JR1	2

Table 6: Example Query Plan

```
Configuration File:
package afrl;
public interface afrlConstants {

int     NUMBER_OF_QUERIES     = 1; //queries generated
String  FILE_NAME             = "outputFiles/apr13_queries_exp1.txt"; // file name
String  NETWORK_FILE_NAME     = "outputFiles/network/apr13_network_exp1"; //conn matrix file
int     SEED                  = 4406235; //for query generator
int     NETWORK_DEGREE        = 11; //# of connected nodes
int     NUM_NETWORKS          = 6; //# of connectivity matric to be generated
int     NETWORK_SEED          = 33152035; //seed for connection matrix generator
int     NUM_NODES             = 13; //# of nodes in connection matrix
int     TopKOptimal           = 3; // optimal plans to display; 0 (all)
int     TopKCumulativeCost    = 3; //carry K plans, 0 (not use this heuristic)
int     TopKIterationCost     = 3; //carry K plans, 0 (not use this heuristic)
int     TopKJoinType          = 9; //carry k number of join type heuristic applying
                                   //cumulative heuristic to k/3 of each type
boolean displayNonConnective  = false; //true to display non connective plans
boolean heuristicDebug        = false; //true to dump heuristic execution data to files}
```

Figure 3: A Sample Configuration Specification

Method	Query 1	Query 2	Query 3	Query 4	Query 5
Optimal Join	8.19	55.34	63.76	2.89	2.52
Optimal semijoin	3.29	8.61	135.33	4.74	2.17
Top-K Cumulative Join	20.54	68.07	85.57	62.78	4.70
Top-K Cumulative semijoin	9.91	39.31	78.46	12.21	4.29
Top-K Iterative Join	8.19	59.54	70.76	5.34	5.62
Top-K Iterative semijoin	4.19	9.31	129.39	162.62	4.95
Top-K Join-type Join	174.20	485.37	80.76	7.12	7.70
Top-K Join-type semijoin	11.91	390.31	129.39	3.21	3.29

Table 9: Heuristics Vs. Optimal: Costs incurred across top-3 plans

```
Query 1: target 2
SELECT   *
FROM     UAV_2_DATA, UAV_4_DATA, UAV_5_DATA
WHERE    ((UAV_2_DATA.NODEID=66)) AND((UAV_2_DATA.LONG>=614)) AND ((UAV_5_DATA.NODEID=77))
         AND ((UAV_2_DATA.LAT=UAV_4_DATA.LAT)) AND ((UAV_4_DATA.NODEID=UAV_5_DATA.NODEID));

Query 2: target 5
SELECT   *
FROM     UAV_5_DATA, UAV_10_DATA
WHERE    ((UAV_10_DATA.LAT=609)) AND ((UAV_10_DATA.OBJPTR<=246)) AND ((UAV_5_DATA.OBJPTR=UAV_10_DATA.OBJPTR));

Query 3: target 9
SELECT   *
FROM     UAV_9_DATA, UAV_10_DATA, UAV_5_DATA
WHERE    ((UAV_9_DATA.LONG>351)) AND ((UAV_9_DATA.LAT>=40)) AND ((UAV_5_DATA.LONG<=804))
         AND ((UAV_9_DATA.OBJPTR=UAV_10_DATA.OBJPTR)) AND ((UAV_10_DATA.LAT= UAV_5_DATA.LAT));

Query 4: target 6
SELECT   *
FROM     UAV_6_DATA, UAV_10_DATA, UAV_4_DATA
WHERE    ((UAV_6_DATA.LAT<55)) AND ((UAV_6_DATA.NODEID<=260)) AND ((UAV_4_DATA.NODEID=22))
         AND (((UAV_6_DATA.TIMESTAMP=UAV_10_DATA.TIMESTAMP)) AND (UAV_10_DATA.OBJPTR= UAV_4_DATA.OBJPTR));

Query 5: target 9
SELECT   *
FROM     UAV_9_DATA, UAV_3_DATA, UAV_2_DATA, UAV_4_DATA WHERE ((UAV_9_DATA.TIMESTAMP<=764))
         AND ((UAV_9_DATA.LONG<102)) AND ((UAV_2_DATA.NODEID=66)) AND ((UAV_2_DATA.LONG>=614))
         AND ((UAV_9_DATA.LAT=UAV_3_DATA.LAT)) AND ((UAV_3_DATA.LAT=UAV_2_DATA.LAT))
         AND ((UAV_2_DATA.OBJPTR=UAV_4_DATA.OBJPTR));
```

Figure 4: Sample Queries Used

Method	Network 1	Network 2	Network 3	Network 4	Network 5	Network 6
Optimal Join		8.07				
Optimal semijoin	4.53	4.79	3.67	3.18	3.17	4.01
Top-K Cumulative Join	44.01	20.44	20.44	17.78	13.52	16.05
Top-K Cumulative semijoin	4.51	272.38	272.38	4.61	4.31	272.38
Top-K Iterative Join						
Top-K Iterative semijoin	5.18	5.55	14.48	3.15	4.17	4.47
Top-K Join-type Join	33.31	20.44	16.73	17.78	16.23	14.55
Top-K Join-type semijoin	6.81	4.55	4.47	4.81	4.17	6.06

Table 10: Heuristics V/S Optimal: Costs incurred across different connectivity configurations