The SPRAWL Distributed Stream Dissemination System

by

Yuan Mei

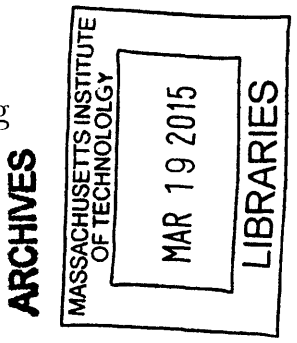
Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2015



Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
January 30, 2015

Signature redacted

Certified by.....
Samuel R. Madden
Professor
Thesis Supervisor

Signature redacted

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

The SPRAWL Distributed Stream Dissemination System

by

Yuan Mei

Submitted to the Department of Electrical Engineering and Computer Science
on January 30, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Many large financial, news, and social media companies process and stream large quantities of data to customers, either through the public Internet or on their own internal networks. These customers often depend on that data being delivered in a timely and resource-efficient manner. In addition, many customers subscribe to the same or similar data products (e.g., particular types of financial feeds, or feeds of specific social media users). A naive implementation of a data dissemination network like this will cause redundant data to be processed and delivered repeatedly, wasting CPU and bandwidth, increasing network delays, and driving up costs.

In this dissertation, we present SPRAWL, a distributed stream processing layer to address the wide-area data processing and dissemination problem. SPRAWL provides two key functions. First, it is able to generate a shared and distributed multi-query plan that transmits records through the network just once, and shares the computation of streaming operators that operate on the same subset of data. Second, it is able to compute an in-network placement of complex queries (each with dozens of operators) in wide-area networks (consisting of thousands of nodes). This placement is optimal within polynomial time and memory complexity when there are no *resource (CPU, bandwidth)* or *query (latency) constraints*. In addition, we develop several heuristics to guarantee the placement is near optimal when constraints are violated, and experimentally evaluate the performance of our algorithms versus an exhausting algorithm. We also design and implement a distributed version of the SPRAWL placement algorithm in order to support wide-area networks consisting of thousands of nodes, which centralized algorithms cannot handle. Finally, we show that SPRAWL can make complex query placement decisions on wide-area networks within seconds, and the placement can increase throughput by up to a factor of 5 and reduce dollar costs by a factor of 6 on a financial data stream processing task.

Thesis Supervisor: Samuel R. Madden
Title: Professor

To my dear husband and parents,

Wen and Guangquan & Yu

Acknowledgments

It's snowing again, just like the winter I started my journey of doctoral study at MIT. This is a moment I have dreamed for years — when my life met setbacks, when my research lost momentums, and when my career faced difficult choices, I always told myself, you must keep faith and stand on. This is an unforgettable moment for me to thank all the people I am deeply indebted to.

First, I want to thank my advisor, Professor Sam Madden, who is beyond comparable. There are so many things I want to thank him that I do not know where to start. Words can not express my gratitude to him with so many years' help. Since the first day I came to MIT, Sam has put enormous trust on my research. Under his leadership, I have been so lucky to witness the emergence of big data as an exciting new field, and from him, I have learned the imperativeness for engineers to integrate academic knowledge with industrial practice. In addition to research, Sam has also provided lots of help on my student life and career development — from the bottom of my heart, he is not only an advisor and a role model, but a good friend and a family member. He has not only taught me the knowledge to be a good researcher, but the virtues to be a great person. I am really proud of having been his student, and sincerely hope one day I can also make him proud.

Second, I want to thank my committee members, Professors Michael Stonebraker and Hari Balakrishnan. As prominent scholars in computer science and engineering, Michael and Hari have very busy schedules, but they can always find time to patiently listen to my research talk and tirelessly help me find the right answers. Every time I met with them, I was amazed by the breadth and depth of their knowledge and skills. It has been such an honor to have both of them on my doctoral committee.

Next, I want to thank my lab mate, Eugene Wu. He kindly provided many details on the SASE project and helped me walk through the slides again and again before my dissertation defense. I also want to thank all other members in the DB group and many of my friends on campus and throughout the world. No matter where your are after graduation, I will definitely remember our friendship and hope our paths can

cross again in the future.

Finally, I want to dedicate my dissertation to my husband, Wen Feng, and my parents, Guangquan Mei and Yu Wang, for their unconditional, endless, and eternal love — my parents gave me life, brought me up, and influenced me to be a person of integrity, and meanwhile, the Cupid made a special arrangement for me to meet the best boy in our best age and at the best place — any language in the world cannot completely express my deep appreciation to and tremendous love for my family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern financial and Internet services need to process and disseminate streams of data to thousands or millions of users spread around the globe. This is accomplished not only via massive centralized compute clusters consisting of hundreds of machines, but by a complex wide-area network of routers and caches. Applications of such networks include real-time financial service systems, news feed systems, and social media networks. News and financial feed services, like Thomson Reuters [7], Bloomberg [3], and Dow Jones have to process and stream massive quantities of data feeds both over the public Internet as well as over their private networks to subscribers who have various requirements as quickly and efficiently as possible. Social media networks such as Twitter [8] and Facebook [4] may receive updates at data centers worldwide, subsets of which need to be processed and disseminated efficiently to users and servers all over the world, using both their own data centers and caches provided by caching services like Akamai [1]. Delivering information while at the same time processing it in a cost effective and efficient manner is of critical importance. In addition to the need for efficient data processing, the cost of simply transmitting this data can be quite significant. For example Amazon charges $.09/GB for data transferred out from EC2 to the public Internet, when transferring more than 1 GB of data per month.

1.1 New Challenges

The applications described above introduce a number of new challenges, including:

- geographically distributed data feeds, users, and network infrastructure,
- global data feeds with potentially high data rates,
- massive numbers of user subscriptions, which may include complex queries like pattern detection and require short latency, and
- heterogeneous wide-area networks with thousands of machines and varied network connectivity.

To address the special challenges of wide-area distributed stream processing and dissemination problems outlined above, we need a simple, effective and scalable solution to deliver results of user subscriptions in a timely manner (satisfy latency requirements), while minimizing usage of network resources and accommodating CPU and bandwidth constraints in order to support as many queries as possible.

1.2 Contributions

In this dissertation, we describe SPRAWL, a data stream distribution layer designed to efficiently distribute data processing across hundreds or thousands of nodes. Specifically,

1. SPRAWL employs a *decomposition and placement (DP)* algorithm similar to *Nonserial Dynamic Programming (NSDP)* [33] that, given a network of servers, with measurements of CPU, bandwidth and latency between servers, and an operator graph, optimizes placement of the operators on the servers to minimize some objective function (e.g., total bandwidth cost or CPU cost). The SPRAWL DP algorithm can guarantee an optimal placement within polynomial time and memory complexity when resources and latency are unconstrained.

2. SPRAWL includes extensions to SPRAWL DP algorithm to deal with cases where resource and latency constraints are included. We experimentally show that these extensions perform near-optimally. This is important because many applications need to run on commodity machines with limited CPU capacity and network bandwidth. In addition, latency concerns are often significant in streaming settings.

3. SPRAWL extends SPRAWL DP algorithm with a distributed version that partitions the query plan and assigns sub-plans to network clusters. Each cluster is responsible for placing its local sub-plan partition, and collaborates with each other to optimize the overall placement. This extension makes SPRAWL scalable to thousands of network nodes and queries each with dozens of operators.

4. SPRAWL includes *multi-query sharing strategies* that identify opportunities to share the transmission of data through the network, as well as the shared execution of operators. SPRAWL extends SPRAWL DP algorithm to support multi-output DAG query plans in this case since a shared operator is very likely to have multiple outputs and the original SPRAWL DP algorithm may not apply any more.

5. Finally, SPRAWL make items 1 – 4 possible in a variety of stream processing systems (single node or distributed) [28, 13, 44, 6]. SPRAWL is not a full featured stream processing system. Instead, we have designed SPRAWL to work with different stream processing engines via a unified interface, as long as the system provides the capability to implement a DAG of stream processing operators and supports the appropriate operator implementations.

1.3 Related Work

There has been prior work on distributed stream processing and in-network multi-query placement [41, 37, 58, 10, 47] that closely related to SPRAWL. However, previous work lacks key features SPRAWL provides. Min-Cut [41] and SODA [58] are both

centralized placement algorithms, and are not scalable to wide-area networks with hundreds or thousands of nodes. In addition, Min-Cut [41] does not handle CPU costs or resource/query constraints. SQPR[37] is more focused on multi-query sharing, and uses a *mixed integer linear program (MILP)* [5] to solve the operator placement problem, which has exponential time complexity if resource and query constraints are considered. Finally, SBON [47] and SAND [10] both provide distributed query placement solutions, but offer no guarantee on the quality of the placements, even in the unconstrained case, and (as we show in our experiments) generate placements that are substantially inferior to those produced by SPRAWL. Besides, SBON and SAND may take long time to converge to a stable placement. More related work will be investigated in Chapter 7.

1.4 Dissertation Outlines & Main Results

Chapter 2 describes the SPRAWL system designs, highlights the special requirements of wide-area networks, and defines the type of queries SPRAWL supports. SPRAWL is designed as a multi-query optimization layer for various stream processing systems over wide-area networks. To achieve this goal, SPRAWL contains two parts: a *central control module* and an *individual control module*. The central control module is responsible for multi-query optimization and placement, while individual control modules provide a unified interface for underlying stream processing systems.

Chapter 3 discusses the cost model SPRAWL used by the optimization and placement algorithms, and provides guidance and benchmarks for cost estimates. SPRAWL's cost objectives are designed to minimize overall network resource usage, while simultaneously satisfying resource and query constraints.

Chapter 4 introduces the design and implementation of the SPRAWL *decomposition and placement (DP)* algorithm, proves the optimality of the SPRAWL DP algorithm without resource and query constraints, provides solutions when constraints are considered, and extends the SPRAWL DP algorithm with a scalable version where no global information is necessary for each local central control module to make opti-

mization and placement decisions.

Chapter 5 introduces the SPRAWL data-oriented *multi-query sharing strategies* to extend SPRAWL DP algorithm for support for multiple queries, and proves that the SPRAWL DP algorithm can always find an optimal placement solution no matter which operator in the query plan is chosen as a root.

Chapter 6 contains four sets of experiments. It shows that in the Amazon Elastic Compute Cloud (EC2) [2], SPRAWL can increase throughput by up to a factor of 5 and reduce dollar costs by a factor of 6 on a financial data feed processing benchmark compared to a random placement strategy. It also demonstrates that SPRAWL can make complex query placement decisions on wide-area networks (with thousands of network nodes) within seconds and reduce the (latency or bandwidth) cost by a factor of 3 relative to an existing scalable distributed relaxation algorithm called SBON [47]. In addition, it experimentally shows that SPRAWL DP with constraints can perform almost as well as an exhaustive algorithm, even in highly constrained settings. Finally, it shares some experiences on how cost estimates are chosen, and how throughput and dollars spent relate to the choices of cost estimates.

Chapter 7 is a literature review, Chapter 8 describes future work and Chapter 9 concludes the dissertation.

Chapter 2

SPRAWL System Design

In this chapter, we provide an overview of the design and architecture of SPRAWL, including the network model and queries it supports. We begin with a few definitions.

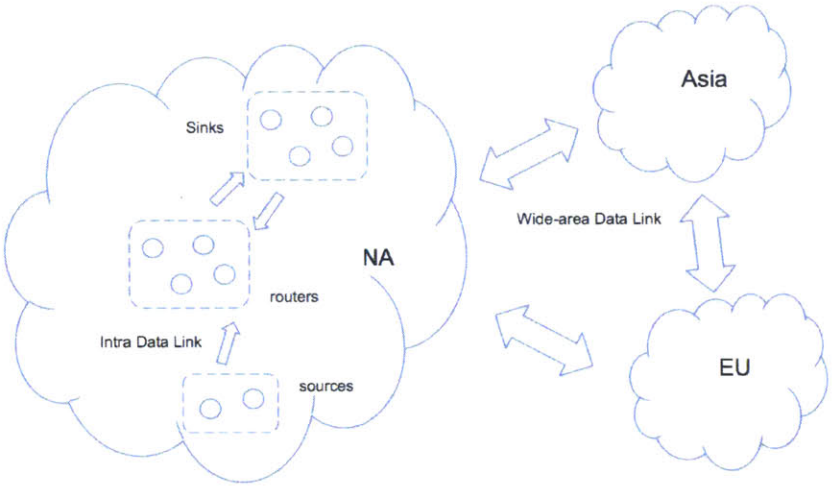
2.1 Definitions

To distinguish networks of physical nodes from graphs of operations, we use the following terms in this dissertation:

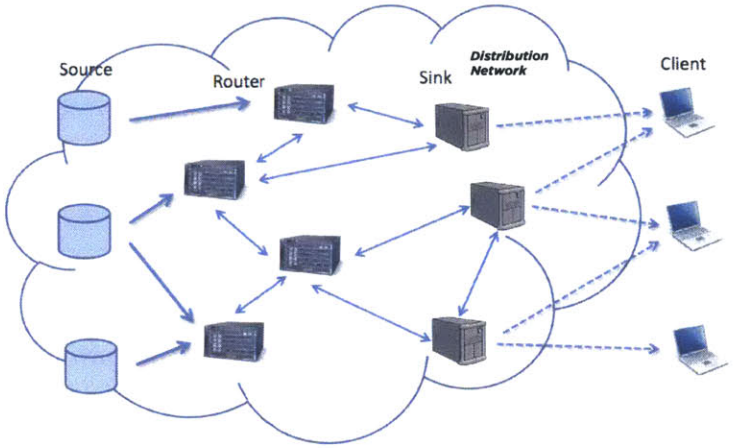
- *Network Node* : a server used to generate and/or process data across the physical network.
- *Network Link* : a physical connection between network nodes.
- *Query Operator* : a block of code that applies a specific operation to data, e.g., "filter", "aggregate", or "join". Some operators are *pinned* to particular network nodes, and others are *unpinned* and are free to execute anywhere in the network.
- *Data Edge* : an edge between two operators that carries data.
- *Query DAG* : a directed acyclic graph composed of operators for query planning and execution. A query DAG is typically a tree for a single query, but can be

non-tree when multiple queries are merged together because shared operators may have multiple outputs.

2.2 Wide-Area Network Properties



(a) Wide-Area Network Topology with Three Clusters



(b) Internal Network Structure in each Data Cluster

Figure 2-1: Illustrative Wide-Area Network Model

Since SPRAWL is designed to be scalable to wide-area networks, we begin with a brief description of the properties of such networks. Wide-area networks have clusters of nodes (e.g., Amazon availability zones, or data centers in large organizations),

connected via wide-area links, as shown in Figure 2-1a. Within each data cluster, network nodes are connected via high-bandwidth, low-latency local-area links, as shown in Figure 2-1b.

Compared to wide-area links, local-area links typically have much lower latency (1 ms or less) and much higher network bandwidth (1-10 Gbps), with very low (or free) per-byte transmission costs. Since wide-area links have to traverse the public Internet, service providers often charge on a per-byte basis, and throttle the maximum allowed data rate per connection. Wide area cluster latencies range from 10s to 100s of milliseconds. Network nodes in wide-area networks can be categorized based on their functionality:

- *sources* (data feeds) produce data to be processed,
- *routers* (switches) process data and disseminate it to other network nodes, and
- *sinks* deliver query results to users.

A network node can be a source, router and sink at the same time.

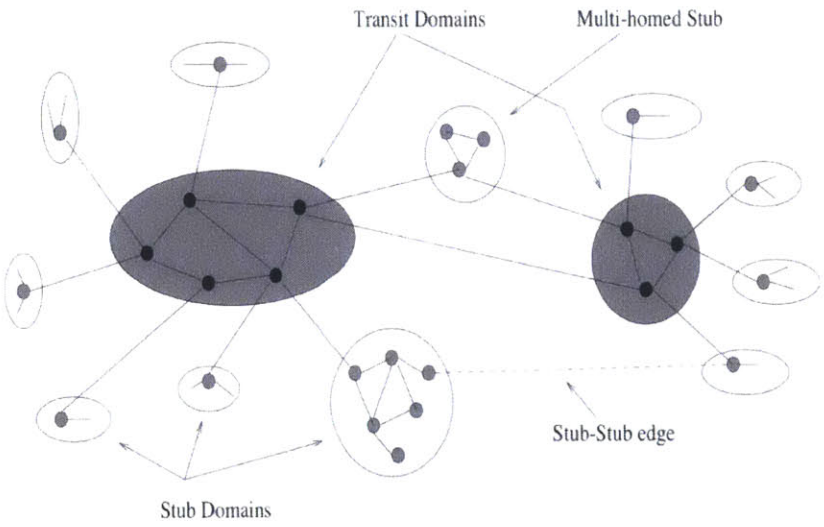


Figure 2-2: Illustrative Transit-Stub Network Model (reproduced from [61])

Wide-area network structures are often modeled by a *Transit-Stub* model, as shown in Figure 2-2 [61]. In a transit-stub model, data clusters are connected via a network

of *border* nodes (transit domains), marked as gray areas in the figure. These border nodes route data from inside the cluster to the wide-area Internet, and from the wide-area Internet into the cluster. SPRAWL distributed sharing strategies and placement algorithms are designed based on this structure.

We assume that network links are symmetric. The routing path between network nodes is the shortest path calculated based on the routing information maintained by border nodes. We assume network topologies are relatively stable, so we do not discuss fault-tolerance issues in this dissertation. The network is not required to be fully connected.

2.3 Input Query DAG

SPRAWL is designed to support a collection of stream operators similar to those that appear in stream processing engines (filters, windowed aggregates, and windowed joins). Currently, SPRAWL only accepts inputs as a DAG of operators rather than as SQL query (i.e., it doesn't have a query parser). Specifically, an input DAG of operators is provided in XML files as was done in Borealis [13].

Query 1 is an example streaming query over a financial data stream written in StreamSQL [6], and its input DAG is illustrated in Figure 2-3. It compares the 5 minute average prices of IBM stock trades in US and Asian markets. Notice that *COM* is a user defined function (UDF) that implements the COMPARE operator. SPRAWL does not need to know implementation details of such UDFs, as long as operator statistics are provided, as we discuss in Chapter 3.

Query 1. Compare (using a user-defined comparison function) the 5 minute average prices of IBM stock trades in US and Asian markets.

```
SELECT  COMPARE(avg(Asia.price), avg(US.price))
WHERE   Asia.symbol = 'IBM' AND US.symbol = 'IBM'
FROM    AsiaStocks as Asia, USStocks as US
WITHIN  5 min
```

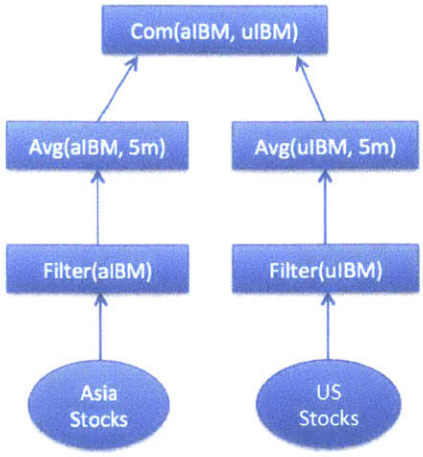


Figure 2-3: DAG Plan for Query 1

As another example, Query 2 finds all stock trades with the same company from Asia, US and European markets, with trading volume greater than X within 1 hour. One of the corresponding query DAG plans is shown in Figure 2-4.

Query 2. Find all stock trades for the same company from Asia, US and European markets, with trading volume greater than X within 1 hour.

```
SELECT   Asia.symbol, US.symbol, EU.symbol
WHERE    Asia.symbol = US.symbol = EU.symbol AND
         Asia.volume > X AND US.volume > X AND EU.volume > X
FROM     AsiaStocks as Asia, USStocks as US, EUStocks as EU
WITHIN   1 hour
```

2.4 System Architecture

SPRAWL provides a unified interface to make multi-query sharing and placement optimization transparent to the underlying stream processing system. As such, a SPRAWL system has two parts: a *central control module*, an instance of which runs in each network cluster, and an *individual control module* that runs on each network node, as shown in Figure 2-5. Central control modules collect and deploy user subscriptions across the network. Individual modules are designed to communicate with

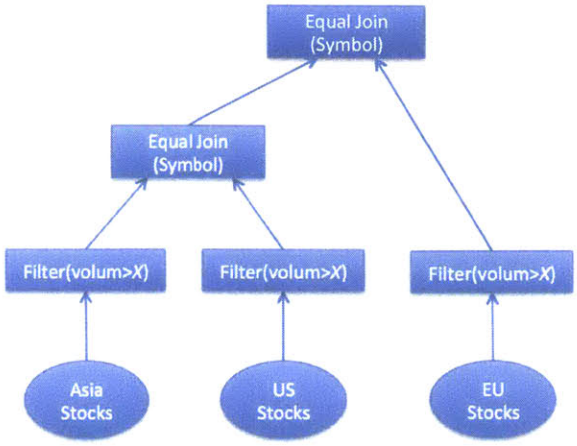


Figure 2-4: DAG Plan for Query 2

the underlying query processors.

2.4.1 Control Module Clusters

A single centralized control module is not a scalable solution for wide-area networks with thousands of network nodes. It may take several minutes for a centralized algorithm to make a placement decision for complex queries over such networks as we will show in Chapter 6, not to mention tracking network resource and routing updates. Hence, SPRAWL partitions wide-area networks into smaller clusters. SPRAWL models wide-area networks based on the Transit-Stub structure [61] described in Section 2.2, in which case each cluster only needs to communicate with neighbor border nodes (transit domains) to decide network routing.

Each network cluster in SPRAWL has a central control module, as shown in Figure 2-5. Central control modules in each network cluster act as peers. Peer central control models collaborate with each other to apply multi-query sharing strategies and make final placement decisions. Each central control model deploys a placement by sending messages to the individual control modules on each network node within the cluster. These messages specify the other nodes in the cluster the node should communicate with and which operators it should instantiate.

An individual control module on each network node is very light-weight. It decodes messages from the central controller, and reconstructs sub-plans to execute on the

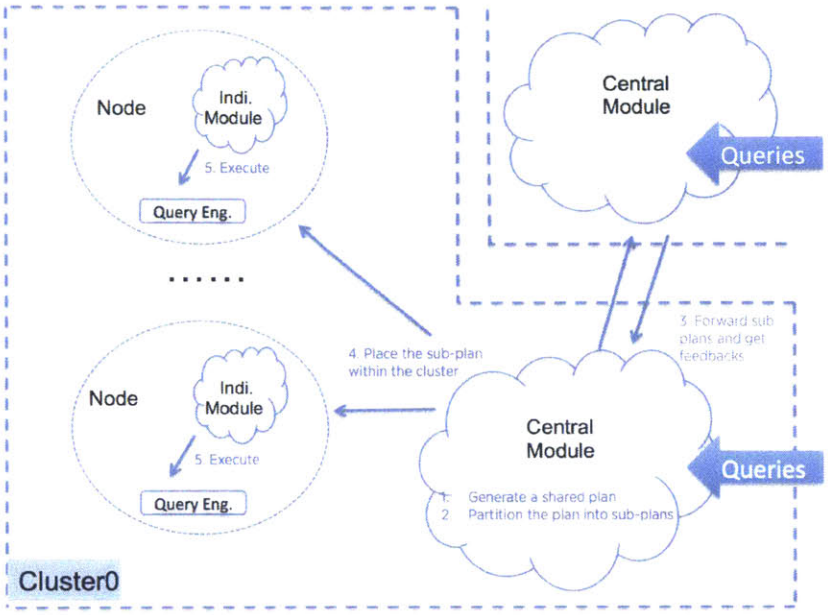


Figure 2-5: SPRAWL System Architecture

underlying stream processing system. Individual control modules support data transmission between network nodes, in the event that the underlying stream processing system is a single-node system which does not support distributed data transfer. In this case, the individual control module redirects the inputs and outputs of sub-plans to its own input and output sockets. In addition, individual control modules track the local computer and network conditions and send updates to the peer central control module within the same cluster. The peer central control module uses these updates to make placement decisions.

Our experiments show that a peer central control module is able to respond within reasonable time delay (seconds) for complex queries in a transit-stub network with fewer than 500 nodes. SPRAWL can scale to thousands of network nodes as long as the average number of nodes in each cluster is bounded (not more than a few hundred).

2.4.2 Example Distributed Query Placement

Figure 2-6 illustrates how SPRAWL places Query 1 over a three-cluster wide-area network. In SPRAWL, queries are registered in the cluster where results are delivered.

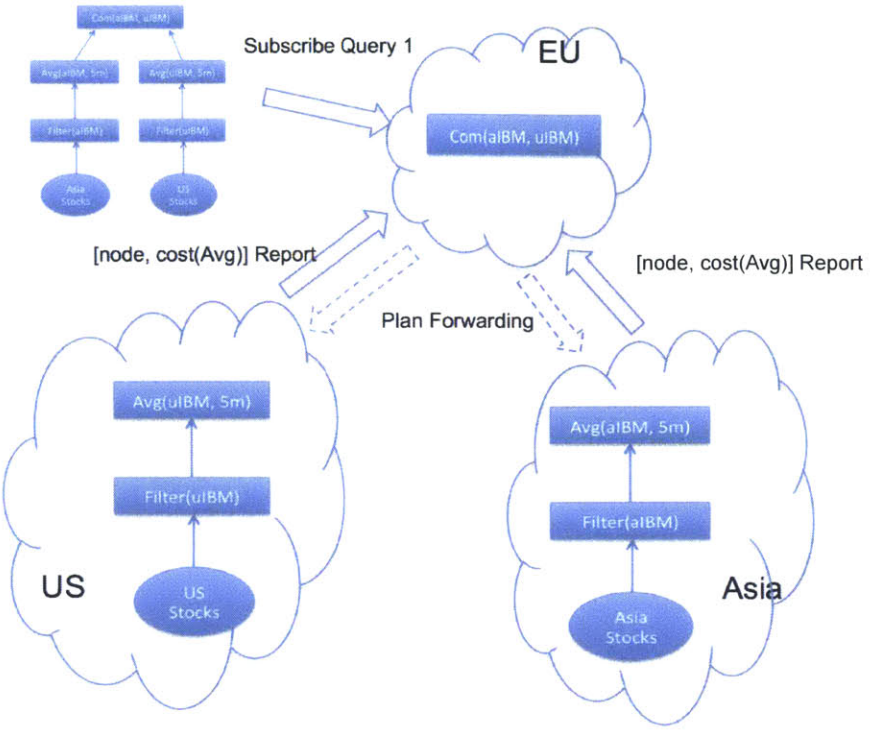


Figure 2-6: Query 1 Placed on Wide-Area Networks with Three Clusters

We call this cluster the *root cluster* for the query. The root cluster is responsible for partitioning registered queries into sub-queries, and deciding which cluster each sub-query should be forwarded to. Query 1 in Figure 2-6 is registered with the EU cluster, so the EU cluster is the root cluster for Query 1. SPRAWL processes Query 1 as follows:

1. The EU central control module accepts Query 1, applies SPRAWL multi-query sharing strategies and partitions Query 1 into three sub-plans.

2. The EU central control module forwards the two *Avg* sub-plans to the US and Asia clusters, respectively, and keeps the *Com* sub-plan for itself.

3. The US and Asia central control modules accept forwarded sub-plans, apply SPRAWL multi-query sharing strategies to these sub-plans the same way as normal input query DAGs, and then use SPRAWL DP algorithm to calculate placement information.

4. The US and Asia central control modules report calculation results of their sub-plans back to the EU root cluster. The reported results are a list of $(node, cost)$ pairs indicating the optimal cost of the sub-plan on each network node within the cluster.

5. The EU root cluster chooses the placement for *Com* based on the calculated results from US and Asia, and notifies US and Asia of its decision.

6. The US and Asia central control modules finalize their local sub-plan placement after receiving these notifications.

A central module only needs knowledge of border nodes from other clusters to compute query partitioning and sub-plan placement, and does not need global information of the entire network.

2.4.3 Central Control Module

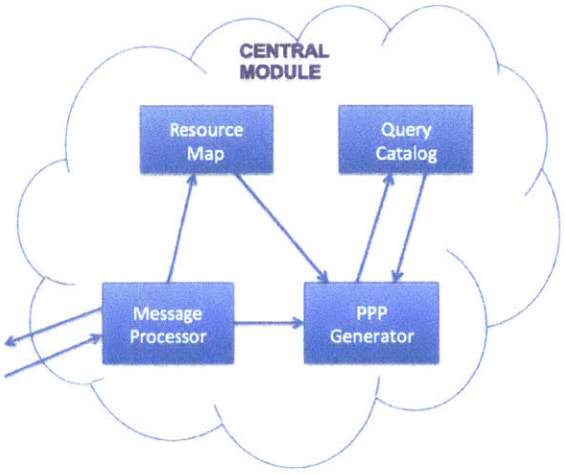


Figure 2-7: Central Control Module

The central control module includes four components, as shown in Figure 2-7:

- *PPP Generator*: a plan/partition/placement generator;
- *Message Processor* : a messenger to communicate with local individual modules and peer central modules in other clusters;

- *Resource Map* : a network resource allocation map for the cluster;
- *Query Catalog* : a query/sub-query catalog recording queries/sub-queries running in the cluster.

The *PPP generator* (abbreviation for plan/partition/placement generator) is the core unit of a central module. It generates a shared query plan by applying SPRAWL multi-query sharing strategies, partitions the plan to sub-plans, and makes placement decisions for the plan. A *message processor* is used to share calculated placement results with other peer clusters and notify individual modules for deployment. As we show in Section 4.5, the number and total bytes of the messages exchanged through the network are small. SPRAWL implements the messenger using efficient *remote procedure calls (RPCs)*. A *resource map* maintains resource allocation information for network nodes and links in the cluster, based on which the PPP generator calculates plan placement. A *query catalog* records queries/sub-queries that are already running in the cluster, based on which the PPP generator decides how to apply multi-query sharing strategies.

2.4.4 Individual Control Module

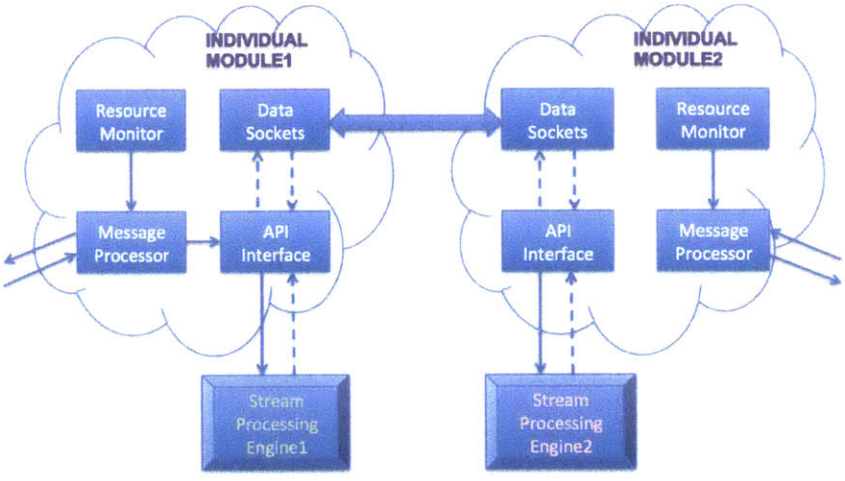


Figure 2-8: Individual Control Module

Each individual control module also contains four parts, as shown in Figure 2-8:

- *API Interface* : an interface for different underlying stream processing engines.
- *Resource Monitor* : a monitor that periodically updates network node and link resource states.
- *Message Processor* : a messenger to communicate with the central module in the cluster.
- *Data Sockets*: for data transmission if the underlying stream processing engine does not support distributed execution.

An individual module decodes deployment messages from a central module, and re-constructs a DAG of stream processing operators executable on underlying stream processing engines through unified API interfaces. We will show how these interfaces work by two cases in Appendix A. An individual module monitors local machine and network conditions through a resource monitor. The monitor periodically samples CPU and network link usages and sends updates to central module through the messenger unit. A data socket unit is necessary when the underlying stream processing system does not support distributed communication, like Wavescope [28]. In that case, input and output data is redirected to the data socket unit.

Chapter 3

Cost Model

Before introducing SPRAWL optimization and placement algorithms, we first present the cost model these algorithms are based on. The SPRAWL cost model captures the total execution "cost" of a particular placement of operators on a network of nodes. It is used to measure the overall quality of a particular placement of operators. The goal of SPRAWL is thus to generate a minimum cost plan and placement. In this chapter, we define a specific cost function, called the *Network Resource Cost Function*, used by default in SPRAWL and suggest cost estimation methods. We note, however, that SPRAWL is suitable to support a family of cost functions and allows users to write their own cost objectives, as we will show later in Section 4.4.3.

3.1 Cost Objective

Cost objectives need to capture a trade-off between the quality of service (QoS) delivered to end users and the operational costs of the service. From an end user's perspective, the service provider should

- correctly deliver results of user subscriptions, and
- deliver results within a certain latency.

Users in financial and news data services are willing to pay more for lower latency. If the query latency is longer than a user wants, he may be unhappy and stop using

the service. Of course, minimizing query latency can be one of the cost objectives, but latency lower than certain thresholds may make no difference for users. Hence, we decide to set query latency as a constraint for the cost objectives.

From a service provider's perspective, SPRAWL is designed to

- minimize the overall usage of network resources (CPU and bandwidth), and
- deliver results with low latency for the majority of end users.

In this way, SPRAWL can satisfy most user requirements while simultaneously supporting as many queries as possible. The *Network Resource Cost Function* is designed based on these criteria.

3.2 Network Resource Cost Function

The *Network Resource Cost Function* is modeled as a combination of multiple network resource criteria. Specifically, we define the *Network Resource Cost Function* as follows:

$$\begin{gathered} \min_{x} \{\sum w_i^{CPU} \times CPU_i + \beta \sum w_j^{BW} \times Lat_j \times BW_j \} \\ \text{Such that, } CPU_i \leq CPU_I,\ BW_j \leq BW_J \text{ and } Lat_q \leq L_q \end{gathered} \tag{3.1}$$

The cost model consists of two sub-expressions: the left models CPU cost ($CPUCost$ for short), and the right bandwidth and latency cost ($BWCost$ for short). Here, x denotes a placement of operators throughout the network.

Although the *Network Resource Cost Function* is designed based on network resources, it also tends to generate query placements with low-latency. Latency depends on a variety of network features, including network congestion, computational load, and the aggregate latency in the overlay network between sources, routers, and sinks. In SPRAWL, we mainly consider three features:

1. *network congestion* — related to network bandwidth consumption.
2. *node load* — related to CPU usage.

3. *path latency* — related to latency of each link in the routing path and the number of hops.

Network traffic will delay network transmission, increasing latency. For the same reason, over-loading a machine may delay data processing, also leading to longer latency. Last but not least, longer routing paths also introduce delays. The *Network resource cost function* guarantees no network nodes or links are over-loaded by enforcing CPU and bandwidth constraints. In addition, SPRAWL always picks up the shortest latency path suggested by border nodes (having routing information).

3.2.1 *CPUCost*

$\sum w_i^{CPU} \times CPU_i$ measures compute costs. CPU_i indicates the total CPU usage on network $Node_i$, which is the sum of the CPU usage for each operator placed on $Node_i$. w_i^{CPU} is the weighted cost of each unit of CPU computation on $Node_i$. This value may vary depending on the type of machines available in the network (e.g., different sized Amazon EC2 instances), and may also vary as a node becomes more or less loaded to encourage the movement of computation between network nodes. For instance, if $Node_i$ is already heavily loaded, we can increase w_i^{CPU} so that the cost to use this node for new subscriptions is relatively high.

3.2.2 *BWCost*

Similarly, $\beta \sum w_j^{BW} \times Lat_j \times BW_j$ measures network costs. w_j^{BW} is the weighted cost of each unit of bandwidth on network $Link_j$. The price for transmitting data over different networks may be hugely variable (e.g., on Amazon Web Services, wide area bandwidth may cost as much as \$0.18/GB, while intra-data center bandwidth is essentially free). Increasing the weight cost (price) of bandwidth can also move communication to other links to avoid network congestion.

Lat_j denotes the latency of $Link_j$. BW_j denotes the sum of bandwidth consumption for each operator transmitting data over $Link_j$. $Lat_j \times BW_j$ measures the saturation of $Link_j$. It increases if the latency or bandwidth consumption of $Link_j$

increases. Of course, the cost functions can also be a simple weighted sum of latency and bandwidth, rather than a product. We choose product instead of sum in this dissertation in keeping with previous research [10, 47] that used a similar methodology. Finally, β is a parameter to vary the relative weight of $CPUCost$ and $BWCost$.

3.2.3 Constraints

$CPU_i \leq CPU_I$ is a *CPU constraint* to ensure $Node_i$'s CPU usage does not exceed its capacity, $BW_j \leq BW_J$ is a *bandwidth constraint* to ensure that $Link_j$ is not saturated, and $Lat_q \leq L_q$ is a *latency constraint* to keep the latency of a query q within a user's specified latency requirement L_q.

3.3 Cost Estimates

In this section, we discuss how to estimate the parameters in Formula 3.1.

3.3.1 $BWCost$ Estimates

Lat_j can be estimated and updated easily by periodically pinging between pairs of network nodes. If needed more sophisticated network measurement techniques could be employed, e.g. Vivaldi [20].

BW_j can be estimated by measuring each source's data rate and each operator's selectivity, using standard database cost estimates. For example, the output bandwidth consumed by a filter with selectivity s and input data rate R tuples/s is $R*B*s$ bytes/s, where B is the bytes used by each input tuple.

w_j^{BW} is the weighted unit cost for bandwidth, which is a provider-dependent value. For example, in Amazon EC2, data transmission between different AWS regions costs \$0.02/GB for first 1 GB of data, and \$0.18/GB up to 10 TB every month. We will also show how we experimentally determine this weight in Section 6.5.2.

3.3.2 $CPUCost$ Estimates

CPU_i is more complicated, because it depends not only on the data and logical query, but also on the query operator implementation. Much of the previous research on network-aware stream processing [10, 47, 41] has excluded CPU cost and CPU constraints due to either methodological limitations or to simplify the optimization formulation. This may be reasonable when running queries with light computation demands. However, in other cases, CPU cost does matter. Specifically, many clouds consist of commodity machines and service providers charge for machines by hour. For example, an Amazon EC2 M3.Xlarge machine (64-bit 4 Core, 3.25 units each core, 13 units in total and 15 GB Memory) costs $\$0.500/hour$. The SPRAWL cost model uses the percentage of a machine's CPU used when running an operator to measure the operator's CPU usage. This method is easy to model and extend if an operator's input and output data rates are known. w_i^{CPU} is the weighted unit cost for CPU, and can be set similarly in the way w_i^{BW} is set.

3.3.3 β Estimates

Finally, we need to decide β in Formula 3.1. β is the relative weight of $CPUCost$ and $BWCost$, varying in different networks and stream processing systems. Users need to profile their own β value based on network conditions and stream processing systems they are using. SPRAWL provides profiling benchmarks to help users choose their own β value.

SPRAWL profiles β based on dollars spent. Suppose we pay $\$_{CPU}$ to reserve a machine for a period of *time*, and $\$_{BW}$ to transmit 1 unit of data through the network ($\$_{CPU}$ and $\$_{BW}$ can be quoted easily from cloud providers). Suppose that during a certain period of *time*, the total amount of data transmitted through network is G. We then get:

$$\begin{aligned} CPUCost &\sim \$_{CPU} \\ \beta \times BWCost &\sim \$_{BW} \times G \end{aligned} \tag{3.2}$$

The left side of Formula 3.2 is the cost calculated from Formula 3.1, and the right side of Formula 3.2 is the corresponding US dollars paid. We say $CPUCost$ and $BWCost$ is equivalent if the same amount of money is paid for computation and for data transmission. Hence,

$$\begin{aligned} \frac{CPUCost}{\beta \times BWCost} &= \frac{\$_{CPU}}{\$_{BW} \times G} \\ \Longrightarrow \qquad \beta &= \frac{\$_{BW} \times G \times CPUCost}{\$_{CPU} \times BWCost} \end{aligned} \tag{3.3}$$

3.3.4 Example β Estimation Benchmark

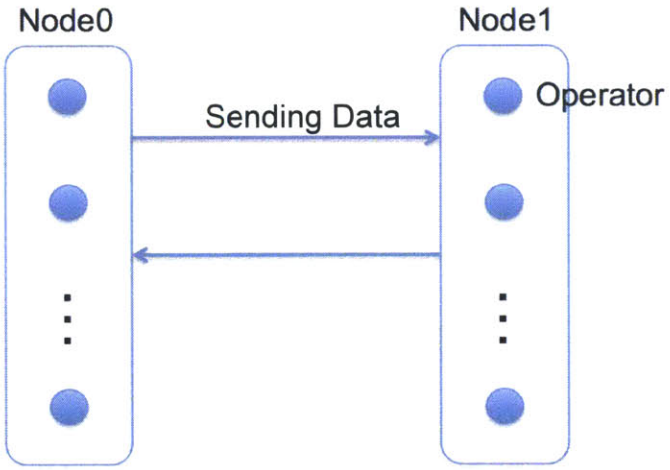


Figure 3-1: Example β Estimation Benchmark

Figure 3-1 illustrates an example benchmark setting for β estimation. We run simple $source \rightarrow filter$ queries on network $Node_0$ and $Node_1$, with sources configured to produce random data at a maximum rate. The price to reserve a machine for a period of $time$ is $\$_{CPU}$. During the period of $time$, the $source \rightarrow filter$ queries send $G\ GB$ of data between $Node_0$ and $Node_1$. The price to transmit 1 GB of data between $Node_0$ and $Node_1$ is $\$_{BW}$. So, we can estimate β easily according to Equation 3.3:

$$\beta = \frac{\$_{BW} \times G \times (CPUCost_0 + CPUCost_1)}{(\$_{CPU0} + \$_{CPU1}) \times BWCost_{0,1}} \tag{3.4}$$

$CPUCost$ is measured as the percentage of CPU units used. Assume the $source \rightarrow filter$ queries use $p_0\%$ and $p_1\%$ CPU capacity of $Node_0$ and $Node_1$ respectively, then

$$\beta = \frac{\$_{BW} \times G \times (p_0\% + p_1\%)}{2\$_{CPU} \times BWCost_{0,1}} \tag{3.5}$$

The remaining job is to calculate $BWCost$ between $Node_0$ and $Node_1$:

$$\begin{aligned} BWCost_{0,1} &= Lat \times BW \\ &= Lat \times G/time \end{aligned} \tag{3.6}$$

Finally,

$$\beta = \frac{\$_{BW} \times time \times (p_1\% + p_2\%)}{2\$_{CPU} \times Lat} \tag{3.7}$$

Chapter 4

Query Plan Decomposition and Placement

In this chapter, we consider the problem of placing a query plan (tree) on a physical network topology. The problem of multi-query plan generation will be discussed in Chapter 5. Even in the single query case, however, there are an exponential number of operator placements, since any operator can be placed on any node, and we have to consider all such possible placements. In this chapter, we show that it is possible to efficiently compute optimal placements within polynomial time and memory complexity, and with few network messages.

This chapter is organized as follows: Section 4.1 introduces the SPRAWL *decomposition and placement* (SPRAWL DP for short) algorithm; Section 4.2 proves optimality of SPRAWL DP without constraints; Section 4.3 extends SPRAWL DP to include network resource and query constraints; Section 4.4 extends SPRAWL DP to a distributed version so that SPRAWL can scale to wide-area networks with thousands of network nodes; finally, Section 4.5 provides an analysis of the time, memory and message complexities of SPRAWL DP.

4.1 SPRAWL Decomposition and Placement Algorithm

The placement problem can be formulated as *discrete programming* that can be solved efficiently by algorithms like *Non-serial Dynamic Programming (NSDP)*. NSDP is a general technique that aims to solve optimization problems in stages, with each stage calculated from the result of previous stages. SPRAWL employs an NSDP-like algorithm we call *decomposition and placement* (DP for short) that is based on the observation that the total cost of each plan can be accumulated from its sub-plans.

4.1.1 Sub-Plan Cost Accumulation

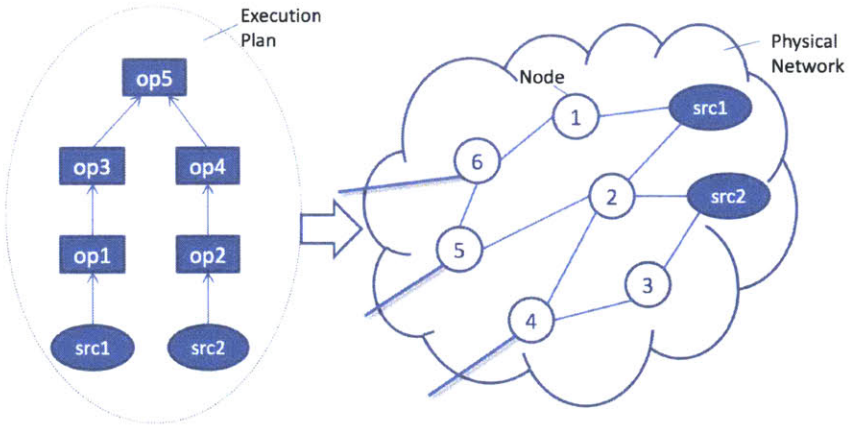


Figure 4-1: Query 1 Placed onto a Physical Network

Figure 4-1 shows how Query 1 is placed onto a physical network. For notational purposes, we denote a subtree by its root operator. For example, subtree op_3 in Figure 4-1 refers to "$src_1 \rightarrow op_1 \rightarrow op_3$", and subtree op_5 refers to the whole tree. If a subtree's root operator op_i is placed on $Node_j$, we denote it as the subtree op_i on $Node_j$. Using the *network resource cost function* given in Formula 3.1, the cost of the subtree op_i on $Node_j$ is the sum of:

- the total cost of op_i's children's subtrees,

- the total $BWCost$ from each child to op_i, and

- the $CPUCost$ of op_i on $Node_j$.

For example, in Figure 4-1, given that op_3 is on $Node_1$ and op_4 is on $Node_2$, the subtree cost of op_5 on $Node_5$ is equal to the sum of:

- *Child subtree cost* : the subtree cost of op_3 on $Node_1$ + the subtree cost of op_4 on $Node_2$

- *BW cost* : $\beta \times [w_{1,5}^{BW} \times Lat_{1,5} \times BW_{(3,5)} + w_{2,5}^{BW} \times Lat_{2,5} \times BW_{(4,5)}]$
  Here, $w_{i,j}^{BW}$ indicates the unit weight cost to transfer data from $Node_i$ to $Node_j$, $Lat_{i,j}$ is the latency from $Node_i$ to $Node_j$, and $BW_{(p,q)}$ indicates the data rate from op_p to op_q.

- *CPU cost* : the CPU cost to process op_5 on $Node_5$

4.1.2 SPRAWL Decomposition Function

We can write the decomposition function as follows:

$$
\begin{aligned}
&OPCost_{i,j} = \\
&\underset{x \in X}{\text{Min}}\{\textstyle\sum_{t \in T}(\underset{[a]}{SubCost_{t,x(t)}} + \underset{[b]}{BWCost_{x(t),j}}) + \underset{[c]}{CPUCost_{i,j}}\} \\
&\text{where } X \text{ is the set of all possible node assignments for} \\
&\text{each operator, } T \text{ is the set of } op_i\text{'s children, and } x(t) \text{ is} \\
&\text{the physical placement of each child } t \text{ in } T.
\end{aligned}
\tag{4.1}
$$

$OPCost_{i,j}$ denotes the optimal subtree cost of the subtree op_i on $Node_j$, $SubCost_{t,x(t)}$ denotes the subtree cost of op_t on $Node_{x(t)}$, given a assignment x; $BWCost_{x(t),j}$ represents *latency* $\times$ *bandwidth* cost ($BWCost$ for short) of sending data from op_i's child op_t to op_i via network link from $Node_{x(t)}$ to $Node_j$; $CPUCost_{i,j}$ stands for CPU cost ($CPUCost$ for short) of executing op_i on $Node_j$.

4.2 Optimality of SPRAWL DP without Constraints

We first consider the case where there are no bandwidth, CPU, or latency constraints on network nodes, links or queries, i.e., if the constraint clauses in Formula 3.1 are not considered. In this case, the decomposition function has an optimal substructure.

4.2.1 Optimal Substructure

Theorem 4.2.1. *For a given tree-structured plan with op_i on $Node_j$, its optimal subtree placement with cost $OPCost_{i,j}$ must contain an optimal placement of each of its children's subtrees.*

Proof. We prove this by contradiction. Suppose we have an optimal subtree placement with cost $OPCost^*_{i,j}$. Consider one of op_i's children op_t on $Node_{x(t)}$. If op_t's subtree on $Node_{x(t)}$ (with subtree cost $SubCost_{t,x(t)}$) does not have an optimal cost, we can replace $SubCost_{t,x(t)}$ with $OPCost_{t,x(t)}$ to get a better solution for $OPCost_{i,j}$, since no constraints apply. This contradicts the fact that $OPCost^*_{i,j}$ is optimal. □

Hence, in the unconstrained case, $SubCost_{t,x(t)} = OPCost_{t,x(t)}$, and we can rewrite Equation 4.1 as follows:

$$OPCost_{i,j} = \underset{x \in X}{\text{Min}}\{\sum_{t \in T}(\underbrace{OPCost_{t,x(t)}}_{[a]} + \underbrace{BWCost_{x(t),j}}_{[b]}) + \underbrace{CPUCost_{i,j}}_{[c]}\} \tag{4.2}$$

Equation 4.2 indicates $OPCost_{i,j}$ can be accumulated using its immediate children's optimal subtree cost (Part [a]). Part [b] of Equation 4.2 is the $BWCost$ from op_t to op_i via the link from $Node_{x(t)}$ to $Node_j$, which can be estimated easily as long as we know the output data rate from the child op_t to op_i. The output data rate can be estimated based on the plan structure and data source rate. Part [c] represents the $CPUCost$ of executing op_i on $Node_j$, which can also be estimated easily because it is only related to the number of CPU units needed to process operator op_i on $Node_j$.

Without the CPU constraint clauses in Formula 3.1, $CPUCost$ (Part [c]) is independent from the rest of Equation 4.2. Hence, Equation 4.2 can be rewritten as:

$$OPCost_{i,j} = \underset{x \in X}{\text{Min}}\{\sum_{t \in T}(\underset{[a]}{OPCost_{t,x(t)}} + \underset{[b]}{BWCost_{x(t),j}})\} + \underset{[c]}{CPUCost_{i,j}} \tag{4.3}$$

Without bandwidth constraint clauses, each of op_i's children's optimal subtree costs plus $BWCost$ (Part[a] + Part [b]) are independent of each other. So we can further reduce Equation 4.3 to:

$$OPCost_{i,j} = \sum_{t \in T}\{\underset{x \in X}{\text{Min}}(OPCost_{t,x(t)} + BWCost_{x(t),j})\} + CPUCost_{i,j} \tag{4.4}$$

Equation 4.4 indicates that $OPCost_{i,j}$ can be calculated as long as every child's $OPCost_{t,x(t)}$ is known.We can ensure this property easily by calculating each $OPCost_{i,j}$ in a postorder traversal of the query tree to make sure that before calculating $OPCost_{i,j}$, $OPCost_{t,x(t)}$ has already been calculated and cached for each of op_i's children op_t, on each possible placement of op_t in X. This property guarantees the optimality of the entire query tree placement generated by SPRAWL DP algorithm.

Continuing with the example shown in Figure 4-1, a feasible postorder traversal of the qeury plan (shown in the left) is "src_1, op_1, op_3, src_2, op_2, op_4, op_5". When calculating the optimal subtree cost $OPCost_{op_5,j_5}$ of the subtree rooted from op_5 on each possible network $Node_{j_5}$ ($Node_1$ ... $Node_6$), $OPCost_{op_3,j_3}$ and $OPCost_{op_4,j_4}$ for each $j_3, j_4 \in (1...6)$ have already been calculated and cached. To calculate each $OPCost_{op_5,j_5}$ with $j_5 \in (1...6)$, we only needs to calculate op_3's (and op_4's) network position $x(op_3)$ that minimizes the optimal subtree cost rooted from op_3 plus bandwidth cost $BWCost_{x(op_3),j_5}$.

4.2.2 Pseudocode for SPRAWL DP Algorithm

The SPRAWL DP algorithm for the unconstrained case is shown in Algorithm 1 (pseudocode). The plan $Tree$ to be deployed and the physical network Net (in-

Algorithm 1: SPRAWL DP Algorithm with No Constraints

Input: operator plan $Tree$ and physical network Net
Output: optimal placement OPT

```
1  Initialize two dimensional matrices OPCost, and COPT
2  foreach operator op ∈ Tree (in postorder) do
3      if op is leaf then
4          foreach n ∈ Net do
5              OPCost[op][n] = CPUCost_{op,n};
6          continue;
7      foreach n ∈ Net do
8          OPCost[op][n] = CPUCost_{op,n}; COPT[op][n] = ∅;
9          foreach c ∈ children of op do
10             MinC = ∞; MinP = −1;
11             foreach k ∈ Net do
12                 if MinC > OPCost[c][k] + BWCost_{k,n} then
13                     MinC = OPCost[c][k] + BWCost_{k,n}; MinP = k
14             OPCost[op][n]+ = MinC;
15             COPT[op][n] ⟵^{insert} (c, MinP)
16 MinRC = ∞; MinRP = −1
17 foreach n ∈ Net do
18     if MinRC > OPCost[root][n] then
19         MinRC = OPCost[root][n]; MinRP = n
20 return deploy(root, MinRP, COPT)
21 def deploy(op, node, COPT):
22     OPT ⟵^{insert} (op, node)
23     if op is leaf then return OPT;;
24     foreach c ∈ children of op do
25         merge(OPT, deploy(c, COPT[op][node].c, COPT))
26     return OPT
```

cluding topology and resource information) are input parameters to the algorithm. The output is an optimal placement of $Tree$ over Net. Algorithm 1 traverses $Tree$ in postorder, from leaf to root. If the operator is a leaf, the minimal cost to put operator op on $Node_n$ is just the corresponding CPU cost (lines 3 – 6). If an operator op can not be put on some particular $Node_n$ in the network (e.g., source operators may only be allowed to put on the source nodes), we can make $CPUCost_{op,n} = \infty$ to avoid executing op on $Node_n$.

Non-leaf operators are considered starting from line 7. $OPCost[op][n]$ is used to store the minimal cost of the subtree rooted from operator op on $Node_n$. $COPT[op][n]$ is used to store each of op's children's positions when the minimal cost is reached. This is used later by *deploy* (line 20) to recursively lookup each child's optimal placement when the subtree root's position is determined. Lines 9 – 15 calculates $MinC$ (the

minimal subtree cost plus $BWCost$) for each child of the current operator op on $Node_n$ under consideration, and records the minimal-cost position of each child as $MinP$. Note that $OPCost[c][k]$ must have already been calculated when op is under consideration, since the algorithm visits operators in postorder.

After the entire tree has been traversed, all the optimal cost information is recorded in $OPCost$, and all the optimal placement information is recorded in $COPT$. Lines 16 – 19 uses this information to search for $MinRC$ (the minimal subtree cost from $root$) and $MinRP$ ($root$'s optimal position). After fixing the root's optimal position, we can use information stored in $COPT$ to recursively find the entire tree's optimal placement, as shown in lines 21 – 26.

4.3 SPRAWL DP with Constraints

When the constraint clauses in Formula 3.1 are considered, Algorithm 1 may no longer guarantee an optimal solution if any of the constraints is reached. For example, in Figure 4-1, when considering the placement of $op5$, it may not be feasible to place children $op3$ and $op4$ on nodes chosen by Algorithm 1 because of the constraint clauses. Constraints in Formula 3.1 can be categorized into two types: *resource constraints* (CPU and bandwidth) and *query constraints* (latency).

Resource constraints can be checked in each calculation stage. To avoid violating resource constraints, we can gradually increase the (weight) cost of the resource unit as the resource becomes scarce or make the resource unavailable when it is fully consumed. In contrast, latency constraints are query oriented. They cannot be checked until the entire plan is built. We show how SPRAWL handles these two types of constraints in this section.

4.3.1 SPRAWL DP with Resource Constraints

To check resource constraints, we maintain an extra $ResourceMap_{i,j}$ to record the CPU and link bandwidth usage for each subtree rooted from op_i on $Node_j$. $ResourceMap_{i,j}$ is calculated and accumulated the same way as $OPCost_{i,j}$. CPU

Algorithm 2: SPRAWL DP Algorithm with Resource Constraints

Input: operator plan $Tree$ and physical network Net
Output: optimal placement OPT

```
1  Initialize two dimensional matrices OPCost, COPT, and ResourceMap;
2  foreach operator op ∈ Tree (in postorder) do
3      place_OK = false;
4      foreach n ∈ Net do
5          ResourceMap[op][n].addCPU(n, op);
6          if ResourceMap[op][n].valid() then
7              NetC <-insert- n; place_OK = true;
8      if place_OK == false then return null;
9      if op is leaf then
10         foreach n ∈ NetC do
11             OPCost[op][n] = CPUCost_{op,n};
12         continue;
13     foreach n ∈ NetC do
14         OPCost[op][n] = CPUCost_{op,n}; COPT[op][n] = ∅;
15         foreach c ∈ children of op do
16             MinC = ∞; MinP = −1; cplace_OK = false;
17             foreach k ∈ Net do
18                 if !ResourceMap[c][k].valid() then continue ;
19                 tempRM = ResourceMap[c][k] + ResourceMap[op][n];
20                 tempRM.addBW(c, k, op, n);
21                 if !tempRM.valid() then continue ;
22                 cplace_OK = true;
23                 if MinC > OPCost[c][k] + BWCost_{k,n} then
24                     MinC = OPCost[c][k] + BWCost_{k,n}; MinP = k
25             if cplace_OK == false then
26                 ResourceMap[op][n].valid = false; break;
27             OPCost[op][n]+ = MinC; COPT[op][n] <-insert- (c, MinP)
               ResourceMap[op][n].add(ResourceMap[c][MinP]);
               ResourceMap[op][n].addBW(c, MinP, op, n);
28 place_OK = false;
29 MinRC = ∞; MinRP = −1
30 foreach n ∈ Net do
31     if ResourceMap[root][n].valid()
32     and MinRC > OPCost[root][n] then
33         MinRC = OPCost[root][n]; MinRP = n;
34         place_OK = true;
35 if place_OK == false then return null;
36 else return deploy(root, MinRP, COPT);
```

and bandwidth resources can be allocated only if the allocation does not violate resource constraints.

Algorithm 2 shows the pseudocode for SPRAWL DP with resource constraints. Lines 3 – 8 check whether each $Node_n$ can provide enough CPU units for operator *op* to run on it. Lines 9 – 12 initialize the cost of leaf operators just as in Algorithm 1. Non-leaf operators are considered starting from line 13. When calculating operator *op*'s children's minimal cost position, the corresponding $ResourceMap$ must also be validated, as shown in line 18 and line 21. If any of operator *op*'s children cannot be placed because of resource violations (line 25), then *op* can not be placed on $Node_n$ (line 26). Finally, the $ResourceMap$ of operator *op* on $Node_n$ is updated if the child's minimal cost position is found (line 27). Lines 28 – 30 look for the optimal node position for the root of $Tree$. The function *deploy* in line 36 is the same as in Algorithm 1.

Instead of setting hard constraints on resources as we did in Algorithm 2, we can adjust the weights w^{CPU} and w^{BW} defined in Formula 3.1 to encourage the movement of computation and network traffic between network nodes and links. If a network node or link is heavily loaded, we can increase its w^{CPU} or w^{BW}. Similarly, we can decrease w^{CPU} and w^{BW} when nodes or links are underutilized. In reality, machines don't stop running when their resources are over-utilized but simply become slower. Network links do not stop transmitting data when they are congested but lead to longer latency. By making resource constraints *soft*, we can still achieve optimality. This does, however, require choosing w^{CPU} and w^{BW} carefully.

4.3.2 SPRAWL DP with Latency Constraints

Latency constraints are different from resource constraints because they are query-oriented instead of network-oriented. They cannot be checked until the entire plan is built. For CPU and bandwidth constraints, if a server or a link is saturated, we can avoid violating constraints by not using the server or link in a later deployment. However, we can not avoid latency constraints in a similar way since latency continues to increase as the deployment proceeds. To solve this problem, we pre-allocate a

latency bound to each subtree of a query, called *latency bound pre-allocation*. Before discussing details of latency bound pre-allocation, we first describe how latency is calculated within the placement procedure.

Latency Calculation

Latency calculation is based on Equation 4.5, and can be done while SPRAWL DP traverses the query tree, so no additional pass of tree traversal is needed.

$$Lat_q = \underset{sub \in q}{\text{Max}}\{Lat_{sub} + Lat_{link}\} + Lat_{proc} \tag{4.5}$$

Equation 4.5 indicates that the latency of a query tree q is accumulated recursively as the maximum latency of any sub-tree *sub* of q (Lat_{sub}) plus the link latency between the root of *sub* and the root of q (Lat_{link}), plus the processing latency of the root of q (Lat_{proc}). In most cases, Lat_{proc} can be treated as a constant, and is often negligible, especially with simple operators in wide-area networks where network latency dominates.

Latency Bound Pre-Allocation

To satisfy latency constraints, we pre-allocate a latency bound to each subtree of a query q based on q's overall latency constraint. The pre-allocation of latency bounds must be done carefully. Setting each subtree's latency constraint too tightly could lead to a bad placement, while setting it too loosely may lead to a placement that violates q's overall latency constraint. For each subtree *op* placed on $Node_n$, its latency bound pre-allocation is set as follows:

$$Lat_{op,n} \leq \underset{q \in Q}{\text{Min}}\{Lat_q - Lat_{n,client} - Lat_{proc}\} \tag{4.6}$$

As illustrated in Figure 4-2, Lat_q is the overall query latency requirement, $Lat_{n,client}$ is the network latency from $Node_n$ to the client node $Node_{client}$ on which query q is delivered, and Lat_{proc} is the operator processing time (from *op* to q's root).

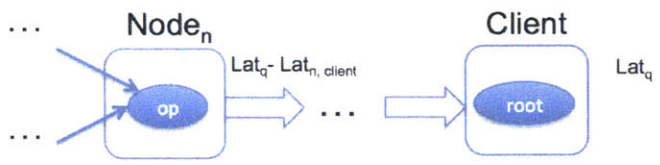


Figure 4-2: Latency Bound Pre-Allocation

As above, in most cases, Lat_{proc} can be treated as a constant, and is often negligible. If multiple queries share the same *op*, the minimal latency bound is chosen.

This latency bound pre-allocation criterion is chosen based on two observations:

1. Every query placement that does not violate the latency constraint and that places *op* on $Node_n$ must also satisfy the subtree latency requirement $Lat_{op,n}$ set by Equation 4.6. This is because the latency between the node of the subtree root operator's placement $Node_n$ and the node where query results are delivered $Node_{client}$ is $Lat_{(n,client)}$, leaving a quota of at most $Lat_q - Lat_{(n,client)}$ latency for the subtree.

2. Subtrees *op* on $Node_n$ that satisfy the subtree latency requirement $Lat_{op,n}$ can obtain at least one latency-valid query placement, simply by putting operators between *op* and the query root onto $Node_{client}$.

The two observations ensure the latency criteria set by Equation 4.6 is neither too tight (every placement that does not violate latency constraints will have already followed the criteria) nor too loose (at least one placement can be found). Latency bound pre-allocation can be done at the same time as the SPRAWL DP traverses the DAG for placement calculation, and does not require an extra traversal of the tree.

Pseudocode for the SPRAWL DP algorithm with latency constraints is shown in Algorithm 3. Algorithm 3 works for latency constraints by including subtree latency bound pre-allocation and latency checks in *ResourceMap.valid*(). Line 28 updates the latency of the subtree *op* on $Node_n$ to its longest path latency.

Algorithm 3: SPRAWL DP Algorithm with Constraints

Input: operator plan $Tree$ and physical network Net
Output: optimal placement OPT

```
1  Initialize two dimensional matrices OPCost, COPT, and ResourceMap;
2  foreach operator op ∈ Tree (in postorder) do
3      place_OK = false;
4      foreach n ∈ Net do
5          ResourceMap[op][n].addCPU(n, op);
6          ResourceMap[op][n].lat = 0;
7          if ResourceMap[op][n].valid() then
8              NetC <-insert- n; place_OK = true;
9      if place_OK == false then return null;
10     if op is leaf then
11         foreach n ∈ NetC do
12             OPCost[op][n] = CPUCost_{op,n};
13         continue;
14     foreach n ∈ NetC do
15         OPCost[op][n] = CPUCost_{op,n}; COPT[op][n] = ∅;
16         foreach c ∈ children of op do
17             MinC = ∞; MinP = −1; cplace_OK = false;
18             foreach k ∈ Net do
19                 if !ResourceMap[c][k].valid() then continue ;
20                 tempRM = ResourceMap[c][k] + ResourceMap[op][n]; tempRM.addBW(c, k, op, n);
21                 tempRM.lat = ResourceMap[c][k].lat + linklat(k, n) if !tempRM.valid() then
                   continue ;
22                 cplace_OK = true;
23                 if MinC > OPCost[c][k] + BWCost_{k,n} then
24                     MinC = OPCost[c][k] + BWCost_{k,n}; MinP = k
25             if cplace_OK == false then
26                 ResourceMap[op][n].valid = false; break;
27             OPCost[op][n]+ = MinC; COPT[op][n] <-insert- (c, MinP)
               ResourceMap[op][n].add(ResourceMap[c][MinP]);
               ResourceMap[op][n].addBW(c, MinP, op, n);
               templat = ResourceMap[c][k].lat + linklat(k, n)
28             if ResourceMap[op][n].lat < templat then
29                 ResourceMap[op][n].lat = templat;
30 place_OK = false;
31 MinRC = ∞; MinRP = −1
32 foreach n ∈ Net do
33     if ResourceMap[root][n].valid()
34     and MinRC > OPCost[root][n] then
35         MinRC = OPCost[root][n]; MinRP = n;
36         place_OK = true;
37 if place_OK == false then return null;
38 else return deploy(root, MinRP, COPT);
```

4.4 Distributed SPRAWL DP

The observation that SPRAWL DP divides the query placement problem into subcomponents, which can be solved and combined to arrive at a globally optimal solution inspires a natural extension to make the SPRAWL DP algorithm distributed: each central control module works on a subset of the plan, and then collaborates with other central control modules to determine the final placement.

As shown in Section 2.4.2, queries are registered in the cluster where results are delivered in SPRAWL. We call this cluster the *root cluster* for the query. The root cluster is responsible for partitioning registered queries into sub-queries, and deciding which cluster each sub-query should be forwarded to. Figure 2-6 illustrates how Query 1 is placed in a three-cluster wide-area network. Query 1 is registered with the EU cluster. The EU control module partitions Query 1 into three sub-plans, placing the two *Avg* sub-plans in the US and Asia clusters, respectively, and keeps the *Com* sub-plan for itself. We will describe this partitioning problem in Section 4.4.1.

To determine the placement of each operator, each cluster (US, Asia and EU) applies the SPRAWL DP algorithm on its assigned sub-plans in parallel. The results of the computation on the US and Asia clusters are sent to the parent (root) cluster (EU), which uses the results of US and Asia to determine the optimal placement of its own operators, and to report back to the child US and Asia clusters the final placement they should use. The details of this algorithm are given in Section 4.4.2.

4.4.1 Query DAG Partition and Assignment

We start with discussing how plans are partitioned and assigned. Distributed SPRAWL DP is designed for wide-area networks where data transmission through wide-area links is costly. Hence our goal is to minimize wide-area bandwidth consumed. The DAG partitioning and assignment problem is similar to the discrete programming problem discussed in Section 4.2, with operators assigned to clusters instead of individual network nodes. Hence, we can design the objective function as

follows:

$$\min_{p} \{\sum w_j^{BW} \times BW_j \} \tag{4.7}$$

Formula 4.7 minimizes the overall bandwidth sent through wide-area data links. p denotes a partition of operators over all clusters, BW_j stands for the sum of bandwidth consumption for each operator transmitting data over wide-area $Link_j$, and w_j^{BW} is the weighted cost of each unit of bandwidth on $Link_j$. The partitioning problem with cost objective Formula 4.7 can be solved using a DP algorithm similar to Algorithm 1. If desired, SPRAWL also allows users to write their own cost objectives in unconstrained cases, as we will discuss in Section 4.4.3.

Wide-area network structures can be modeled by a *Transit-Stub* model [61], where clusters are connected via a network of *border* nodes as discussed in Section 2.2. The border nodes route data from inside the cluster to the wide-area Internet, and from the wide-area Internet into the cluster. Hence, the control module in each cluster only needs knowledge of its border nodes to compute partitioning, and does not need global information of the whole network.

4.4.2 SPRAWL Distributed DP Algorithm

After a query plan is partitioned and sub-plans are assigned to clusters, the distributed placement process can begin.

Consider a sub-plan S with its root operator r placed on a child cluster C, with an edge to an operator o in its parent cluster P. To calculate $OPCost_{o,k}$ for $Node_k$ in cluster P, the central control module in P needs to know $OPCost_{r,j}$ for each $Node_j$ in cluster C as well as the cost to transmit data from r to o, that is $BWCost_{(r,o)}$, as shown in Formula 4.8:

$$\begin{aligned} &OPCost_{o,k} = \\ &\text{Min}\{\textstyle\sum_{r \in o.children()}(OPCost_{r,j} + BWCost_{(r,o)}) + CPUCost_{o,k}\} \\ &\text{for each } Node_j \in r.cluster() \end{aligned} \tag{4.8}$$

In distributed SPRAWL, $BWCost_{(r,o)}$ from operator r to operator o can be divided

into three parts: BWCost from r on $Node_j$ to the border node in C, between the border node of C and P, and from the border node of P to o on $Node_k$. The first part of this BWCost will be added to $OPCost_{r,j}$ (denoted as $OPCost'_{r,j}$) and sent to cluster P. All of these quantities can be computed locally by the central module of C simply by running Algorithm 1 (or Algorithm 3) on S. Hence, the message sent to P is an array of $OPCost'_{r,j}$ for each $Node_j$ in cluster C, indicating the optimal subtree costs of placing S at each $Node_j$ in C. This process continues upward until the root cluster collects all the messages from its child clusters. The root cluster then sends the placement decision back to each of its child clusters, which in turn send placement decisions to their children, and so on.

The message sent back to each child C includes the node information where C's sub-plan should be rooted. This backward process is similar to the *deploy*() function in Algorithm 1. The deployment process is completed when each child cluster receives the decision made by its parent cluster.

Resource constraint checking and latency bound pre-allocation can be done similarly as described in Section 4.3. Again, the central control module in each cluster only needs to know the local sub-plan and networks within the cluster and does not need any global information.

4.4.3 User-Defined Cost Objectives

The SPRAWL DP algorithm decomposes the placement problem into stages,with costs calculated (lines 12 – 13 in Algorithm 1) and accumulated (line 14) in each stage. SPRAWL allows users to define cost objectives by writing their own `Calculate` and `Accumulate` functions. SPRAWL implements a template for users to *instantiate* (line 5), *calculate* and *accumulate* $OPCost$, and provides APIs to access CPU, bandwidth and latency estimates in unconstrained cases.

As an example, if we want to apply SPRAWL DP to the cost objective shown in Formula 4.7, we should initiate $OPCost = 0$ since no CPU cost is considered, and keep the rest of the algorithm unchanged, except that $BWCost$ calculation should be adjusted accordingly. Users must be careful when designing cost objectives to make

sure their cost objectives still have optimal substructure.

4.5 Complexity Analysis

In this section, we analyze time, memory and network message complexity of SPRAWL DP with and without constraints as well as the distributed SPRAWL DP.

4.5.1 Time Complexity

The time complexity of SPRAWL DP without constraints (Algorithm 1) and with constraints (Algorithm 3) are bounded by the four loops (lines 2 – 11 in Algorithm 1, lines 2 – 18 in Algorithm 3). The third loop (line 9 in Algorithm 1, line 16 in Algorithm 3) is negligible because, although arbitrary size queries are allowed, the number of children each operator has is limited. This is true because machines can not handle an unlimited number of input or output sockets. Hence, the entire time complexity is bounded by $O(MN^2)$, where M is the number of the operators in a query plan and N is the number of physical nodes in a network. SPRAWL has better time complexity than Min-Cut [41] ($O(N^3)$) in large networks since M is much smaller than N in such cases.

For the distributed SPRAWL DP algorithm, the time complexity is $\Theta(c * M_s N_c^2)$, where c is the number of clusters which can be considered as a constant, M_s is the average number of operators in each sub-plan and N_c is the average number of nodes in each cluster. Although the upper bound is still $O(MN^2)$, distributed SPRAWL DP works much better in reality because it divides the wide-area network into smaller clusters, and allows computation in parallel within these clusters.

4.5.2 Memory Complexity

The memory complexity of SPRAWL DP with no constraints (Algorithm 1) is related to two matrices $OPCost$ and $COPT$. The memory complexity for $OPCost$ is $\Theta(MN)$, and $O(MN)$ for $COPT$. Hence, the total memory complexity of the

algorithm is bounded by $O(MN)$.

SPRAWL DP with constraints (Algorithm 3) has one more data structure *ResourceMap*. *ResourceMap* is with $\Theta(MN)$ memory complexity, so the total memory complexity is still bounded by $O(MN)$.

The memory complexity of distributed SPRAWL DP is bounded by $O(M_s N_c)$. The message complexity is negligible, as will be shown in the next section.

4.5.3 Message Complexity

For centralized SPRAWL DP, messages are sent between the central control module and each individual module. The central control module sends sub-plan configurations to each corresponding individual module, with message format similar to the API code shown in Appendix A. The overall number of messages sent from the central module to individual modules is bounded by $O(M)$ per query. Each individual module sends machine and network resource updates to the central module after each sub-plan is deployed. Hence the number of network resource update messages is also bounded by $O(M)$.

Besides the messages sent between central modules and individual modules, distributed SPRAWL DP sends additional messages between central modules in different clusters. The number of messages sent between clusters is equal to the number of sub-plans the query DAG is partitioned into, which is bounded by the number of clusters c. The message sent from a child cluster to a parent cluster consists of an array of costs, one for each $Node_j$ in the child cluster, which is around $O(N_c)$ bytes. N_c is normally smaller than 1000, so the size of each message is less than 1KB, and can be implemented efficiently via RPCs. The message sent back from a parent cluster to a child cluster is a single id number that is used to identify the sub-plan root's placement. Hence, the overall message cost is negligible.

Chapter 5

Multi-Query Plan Generation

In Chapter 4, we addressed the problem of operator placement assuming a query DAG (tree) is given. In this chapter, we show how query DAGs are constructed for multiple concurrent queries.

Previous work [31, 30, 50] in database research on multi-query sharing focused mainly on sharing operator processing and reusing intermediate results, since data transmission between operators (except for disk I/O) is not costly in traditional local-area/single-node databases. However, in a wide-area streaming scenario, saving bandwidth is critically important. Because of this, our sharing strategy is oriented around sharing data transmission (reducing bandwidth consumption), rather than just sharing operators. This is significant because it means that even if operators share no computation, they can still share data (tuples) sent over the network, if they operate on the same input streams. Of the related distributed stream processing systems, SQPR [37] also shares streams between operators. However, it uses a mixed integer linear program (MILP) solver, which is not scalable as the number of operators per query and the network size increase.

5.1 Definitions

Before introducing the SPRAWL multi-query sharing strategy, we introduce a few definitions (similar to those in [50], but modified to be data oriented):

Definition 1. *An operator o_i is* **covered** *by operator o_j ($o_i \Rightarrow o_j$) iff o_i's result is a subset of o_j's result.*

Definition 2. *An operator o_i is* **equivalent** *to operator o_j ($o_i \equiv o_j$) iff $o_i \Rightarrow o_j$ and $o_j \Rightarrow o_i$.*

From the definitions given above, we can easily derive two more properties:

Property 1. *If operator $o_i \Rightarrow o_j$ and operator $p_i \Rightarrow p_j$, then $o_i \wedge p_i \Rightarrow o_j \wedge p_j$*

Property 2. *If operator $o_i \Rightarrow o_j$ and operator $p_i \Rightarrow p_j$, then $o_i \vee p_i \Rightarrow o_j \vee p_j$*

Here, $\wedge$ and $\vee$ represent `AND` and `OR`, respectively.

5.2 Multi-Query Sharing Strategy

As mentioned in Chapter 2.3, queries are input as directed acyclic graphs (DAGs) of operators. The SPRAWL multi-query sharing strategy identifies sharing opportunities between these DAGs in three steps:

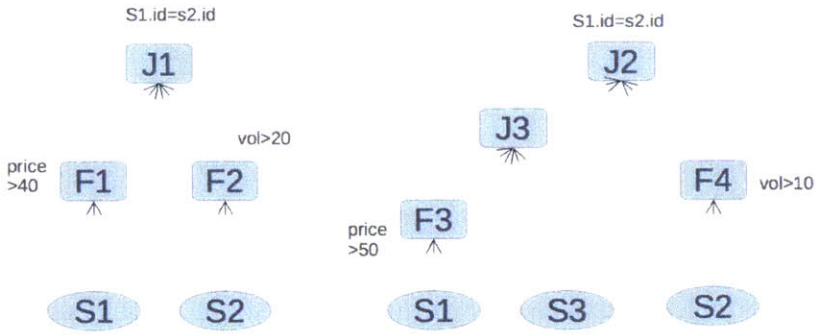
1. SPRAWL uses standard optimization techniques (e.g., a Selinger-style optimizer [49]) to locally optimize these DAGs.

2. SPRAWL identifies operators with the same inputs that perform the same computation in these locally optimized DAGs, and merges them together.

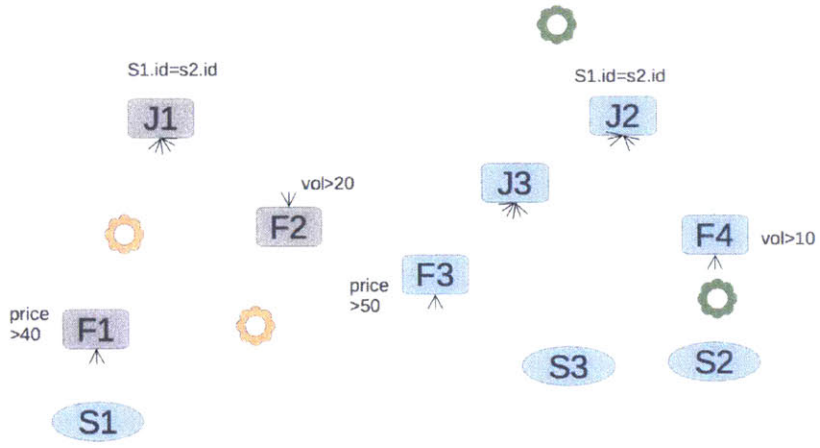
3. SPRAWL identifies opportunities to reuse existing data links.

Steps 1 and 2 are either well-known or straightforward [18, 43], so we focus here on Step 3. Step 3 enables a data-oriented sharing strategy. It not only captures operators that have shared computation, but also avoids disseminating unnecessary copies of data even when operators don't perform shareable computation.

Figure 5-1a shows two locally optimal example plans for two join queries. Assuming the left plan is currently running in SPRAWL, and the right plan is newly inserted, we show how SPRAWL can merge the right plan into the left, and how the corresponding costs are updated so that the SPRAWL DP algorithm can work with the SPRAWL sharing strategy.



(a) Two Locally Optimal Query Plans



(b) Merging the Right Plan with the Left

Figure 5-1: SPRAWL Multi-Query Sharing Strategy

5.2.1 Covered and Equivalent Operators Identification

First, SPRAWL finds all the *covered* and *equivalent* operators in these two plans. This can be done using a bottom-up search for common subexpressions, similar to what is

done in prior work on finding subexpressions [25, 40, 48].

In our example, $F3 \Rightarrow F1$, so we can add an edge from $F1$ to $F3$ and remove $F3$'s original input $S1$, as shown in Figure 5-1b. Also notice that $F2 \Rightarrow F4$, but $F2$ from the left plan has already been deployed. Hence, adding an edge from $F4$ to $F2$ must not affect the placement of either $F2$ or $F4$, since we do not want interfere with existing running plans.

5.2.2 Cost Adjustment

Second, we identify all data links that carry the same data, shown as orange and green flower markers in Figure 5-1b, and adjust the costs of these shared data links so the SPRAWL DP algorithm can perform placement properly.

For example, when calculating $OPCost_{F3,j}$, operator $F3$'s input can come from $F1$ as shown in the figure, but could also be provided by $J1$ or $S1$. Thus, $F3$'s bandwidth cost should be updated to $Min\{BWCost_{(F1,F3)},\ BWCost_{(J1,F3)},\ BWCost_{(S1,F3)}\}$, where $BWCost_{(i,j)}$ denotes the bandwidth cost to transmit output of op_i to op_j. The amount of data sent between operator $(F1, F3)$ and $(J1, F3)$ is equal to the size of output of $F1$, and that between $(S1, F3)$ is the amount of source data produced by $S1$. Note that if $F3$ is placed on the same $Node_j$ as $F1$ or $J1$, then $BWCost_{F3,j} = 0$, since the $BWCost$ to put $F3$ on $Node_j$ has already been "paid" by $F1$ or $J1$. This indicates $OPCost_{F3,j} = CPUCost_{F3,j}$ in this case. Furthermore, since $F3$ shares some of the computation from $F1$ ($price \leq 40$ already been filtered out) if $F3$ uses $F1$'s output as input, $CPUCost_{F3,j}$ should also be adjusted accordingly; specifically, if $F3 = F1$, then $CPUCost_{F3,j} = 0$.

As a second example, consider $OPCost_{F4,j}$. Here, operator $F4$ can get input data from $S2$ or $F2$, so the bandwidth cost is updated to $Min\{BWCost_{(S2,F4)},\ BWCost_{(F2,F4)}\}$. If $F4$ is placed on the same $Node_j$ as $F2$, $BWCost_{F4,j} = 0$ because of link sharing.

5.2.3 Operator Placement

After *CPUCosts* and *BWCosts* are adjusted according to the query sharing strategies introduced above, the SPRAWL DP algorithm can be applied to complete the placement process. In Figure 5-1b, rectangles in gray represent pre-existing operators, ovals stand for sources, and blue rectangles are new operators to be placed. The SPRAWL DP algorithm traverses these blue rectangle operators in postorder and computes an optimal placement for them, as described in Chapter 4.

5.2.4 Plan Reordering

Note that the right plan with three joins in Figure 5-1a may not be the best single-node plan for merging with the left plan since $J1$ and $J2$ have the same join predicate, and have input data from the same source. Currently, SPRAWL only shares operator computations if they have covered or equivalent relations. So, if $F3 \Rightarrow F1$, $F4 \Rightarrow F2$, and thus $J2 \Rightarrow J1$, we decide to share the computation of $J1$ and $J2$. In the case when we decide to share $J1$ and $J2$, the right plan p of Figure 5-1a needs to be reordered to $p\prime$: join $J2$ first, and then $J3$. We then apply SPRAWL placement algorithm to both p and $p\prime$, and pick up the one having smaller optimal operator placement cost.

5.2.5 Distributed SPRAWL Multi-Query Sharing Strategy

The SPRAWL sharing strategies can also work in a distributed setting. For each input DAG, its root cluster's central control module first applies SPRAWL sharing strategies on it, and then partitions the generated shared plan. Other peer clusters only need to apply SPRAWL sharing strategies on the sub-plans assigned to them, in the same way they deal with normal input DAGs. Neither the root cluster nor any peer clusters needs global information.

5.2.6 SPRAWL Multi-Query Plan Generation and Placement Summarization

In summary, a central control module applies SPRAWL multi-query sharing and placement strategies following four steps:

1. Find all the *covered* and *equivalent* relations from the newly subscribed query's locally optimal plan p with all existing plans in the deployed network. Add or remove data links between them if necessary.

2. If there are *join* covered or equivalent relations, p needs to be reordered to $p\prime$ so that join computation can be shared.

3. Mark all data links that send the same copy of data, and adjust $CPUCost$, $BWCost$ and $OPCost$ accordingly.

4. Apply SPRAWL DP algorithm to p (and $p\prime$ if generated in step 2), and pick the optimal placement.

5.3 Multi-Output Plans

SPRAWL DP algorithm calculates $OPCost[op][n]$ during a postorder traversal of a tree plan, and decides the optimal placement in a top-down fashion once the root's optimal position is determined. However, if a plan has multi-output operators, Algorithm 1 – 3 can no longer be applied directly. Multi-output plans arise when several queries share an operator after the SPRAWL multi-query sharing strategy is applied. Such queries have to split results from a shared operator somewhere in the plan, leading to a multi-output structure.

The left side of Figure 5-2 shows an example of a multi-output plan. User 1 and user 2 share the join operator, and then split the outputs after the join. Hence, there are two roots delivering results in the direction of data flow in this DAG: $user1$ and $user2$. If we traverse this DAG in postorder just as in Algorithm 1 – 3, both

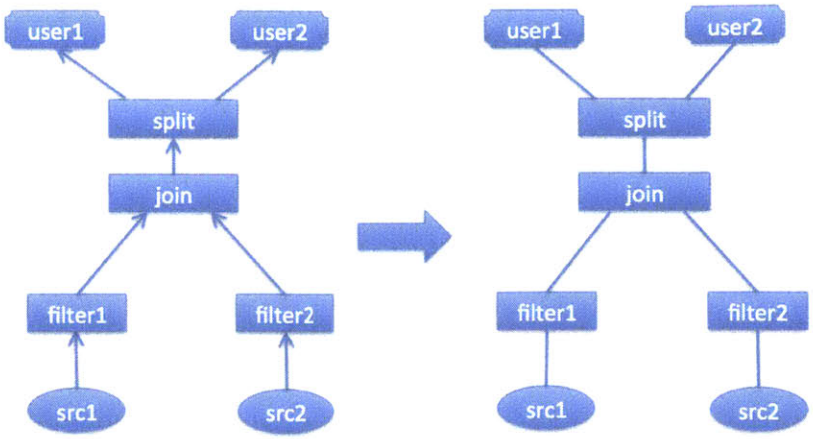


Figure 5-2: Example of Multi-Output Plans

$OPCost[user1][*]$ and $OPCost[user2][*]$ depend on the position of *split*, where $*$ indicates any physical network node. If $OPCost[user1][*]$ and $OPCost[user2][*]$ choose a different placement for *split*, then we need a way to decide which position should be chosen for *split*. In addition it is possible neither of these is an optimal choice.

5.3.1 Naive Solution

An intuitive solution based on Algorithm 1 – 3 is to consider $user1$ and $user2$ together, as follows:

$$\begin{aligned} &OPCost_{(u1,i)(u2,j)} = \\ &\quad \underset{x \in X}{\text{Min}}\{OPCost_{split,x(split)} + BWCost_{x(split),i} + BWCost_{x(split),j} \\ &\quad + CPUCost_{u1,i} + CPUCost_{u2,j}\} \end{aligned} \tag{5.1}$$

where $OPCost_{(u1,i)(u2,j)}$ stands for the optimal sub-DAG cost when $user1$ is on $node_i$ and $user2$ is on $node_j$. $OPCost_{(u1,i)(u2,j)}$ is composite of:

- the optimal sub-DAG cost rooted from *split*,
- $BWCost$ from *split* to $Node_i$ and $Node_j$, and
- $CPUCost$ of $user1$ on $Node_i$ and $user2$ on $Node_j$.

However, this naive solution has two limitations:

1. The complexity of this solution is exponential as the number of users U sharing the same operator goes up: $O(N^U)$. This is because we need to consider each user's possible physical position together as in Equation 5.4.

2. If the outputs of $user1$ and $user2$ in Figure 5-2 are further shared and split, it is difficult to decide the set of operators that should be considered together.

5.3.2 Undirected Graph Solution

We develop an *undirected graph solution* to the multi-output plan problem based on the observation that the cost accumulation (subgraph $\rightarrow$ entire graph) is independent of the direction of data flow. The direction of data flow only affects the cost calculation of each edge (w_j^{BW} may be different in different directions) and latency accumulation (we need direction information to calculate latency). Once the cost is calculated, it does not matter which direction the cost is accumulated from. Hence, we do not need to consider the data flow direction when accumulating the cost of the plan, as shown on the right side of Figure 5-2. This plan is an undirected graph and any node can be picked up as the unique root (e.g., *user1* or *user2*). The SPRAWL DP can work on this undirected graph after picking a root and traverse the undirected graph in a postorder, as long as remembering the edge direction for purposes of cost and latency calculation.

5.3.3 Undirected Graph with Different Roots

Now a new question arises as we have several possible choices of root: will an undirected graph with different roots end up with the same optimal subtree cost? Figure 5-3 shows two corresponding undirected trees of the query DAG in Figure 5-2. On the left side is a tree rooted from $user2$, and on the right side is a tree rooted from *split*. We claim that $OPCost^{user2} = OPCost^{split}$ in unconstrained cases, where $OPCost^i$ means the optimal subtree cost rooted from op_i with op_i as the entire tree root.

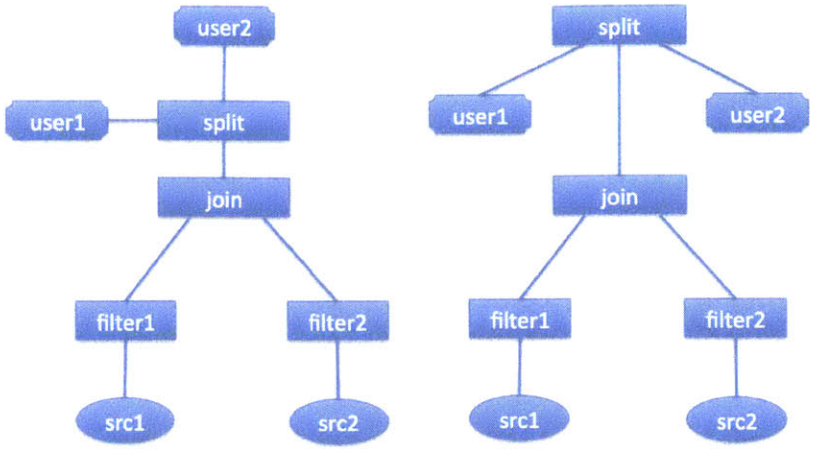


Figure 5-3: Multi-Output Plan with Different Roots

Theorem 5.3.1. *Suppose T_1 and T_2 are two undirected query trees derived from an acyclic DAG plan T, rooted at operators r_1 and r_2, respectively. Then $OPCost^{r_1} = OPCost^{r_2}$ in unconstrained cases, where $OPCost^i$ is the cost of the optimal subtree rooted at op_i with op_i as the entire tree root.*

Proof. We prove this by contradiction. Suppose the theorem is not true; e.g., that (without loss of generality) $OPCost^{r_1} > OPCost^{r_2}$. Setting r_2 as root, and using Algorithm 1, we can find an optimal placement x for each operator: $\{x(1), x(2), ..., x(r_1), ..., x(r_2)\}$, with total cost equal to $OPCost^{r_2}$. According to the cost function, Formula 3.1, the total cost of a placement is just the sum of $CPUCost$ and $BWCost$. Thus, given a placement x, the total cost of the placement will be the same, no matter what order costs are accumulated in.

Now, suppose we use the same placement configuration x on each operator, but calculate and accumulate cost in the order that results from setting r_1 as the root (in postorder just as line 2 in Algorithm 1). Because the order in which we add up the costs does not affect the overall sum, we will get $Cost^{r_1}(x) = OPCost^{r_2}$, where $Cost^{r_1}(x)$ stands for the cost of the subtree rooted at r_1 with r_1 as the entire tree root, given the placement x. Since $OPCost^{r_1} > OPCost^{r_2}$, this means $OPCost^{r_1} > Cost^{r_1}(x)$, which contradicts the fact that $OPCost^{r_1}$ is the optimal subtree cost rooted at r_1, with r_1 as the entire tree root. □

5.3.4 Undirected Graph Solution with Postponed Latency Calculation

Latency calculation is sensitive to the choice of root. For example, in Figure 5-3, choosing $user2$ versus *split* as the root will affect how latency is accumulated. As discussed in Chapter 4.3.2, in a tree-based plan, the latency of a subtree rooted from *op* placed on $Node_n$ is calculated as:

$$Lat_{op,n} = \underset{t}{\text{Max}}\{Lat_{t,MinP} + Lat_{MinP,n}\} \tag{5.2}$$

where t iterates through the set of children of *op*, $Node_{MinP}$ is the minimal-cost network node for t's position calculated by line 13 in Algorithm 1, and $Lat_{MinP,n}$ is the latency between $Node_{MinP}$ and $Node_n$.

In the example, Formula 5.2 indicates the latency of the subtree $user1$ in the right plan should be 0 because it is a leaf (and has no children). However, the real latency of the query rooted on $user1$ is the latency of the subtree *split* plus the latency between $user1$ and *split*. We can address this problem by storing the direction of data flow from the original DAG, and postponing latency calculation. When the SPRAWL DP visits the operator *split*, it notices *split* has a child whose direction of data flow is different from the traversal order. It must then return to $user1$ (and $user2$) to re-update the latency estimate for $user1$ (and $user2$).

Postponed Latency Calculation

The latency calculation problem arises when a query root r is different from the root r^{SP} chosen for SPRAWL DP traversal, in which case some of the edge (data flow) directions are different from the direction of traversal order. We solve this problem by postponing latency calculation for operators with the opposite direction until SPRAWL DP reaching r^{SP}.

Figure 5-4 shows how postponed latency calculation works. The query plan is the same as that in Figure 5-2, but using *split* (left) and $filter2$ (right) as the traversal

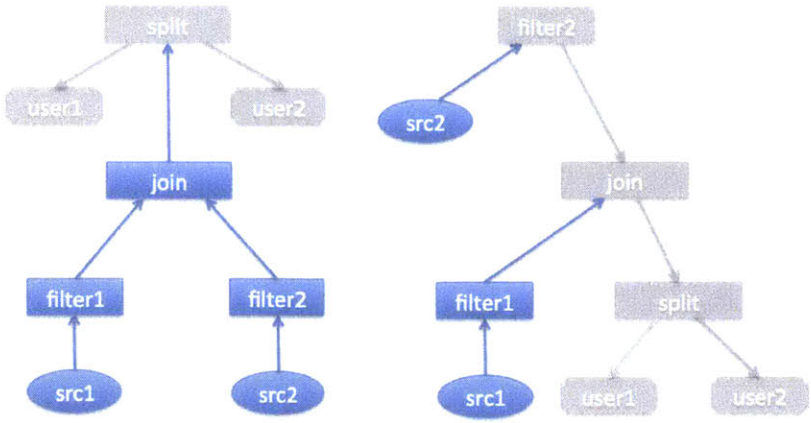


Figure 5-4: Illustration of Postponed Latency Calculation

root r^{SP} respectively. Subtrees with edges whose data flow direction is different than the cost accumulation direction are shown in gray.

For the left DAG, SPRAWL DP processes the subtree rooted from *join* as usual since each edge in the subtree rooted from *join* agrees with the SPRAWL DP traversal direction. In contrast, $user1$ and $user2$'s latency calculations are postponed till root *split* is reached by SPRAWL DP because they are connected to edges with opposite direction. After *split* is reached, $user1$ and $user2$'s latencies are updated as follows :

$$\begin{aligned} Lat_{split} &= Lat_{join} + Lat_{(join,split)} \\ Lat_{u1} &= Lat_{split} + Lat_{(split,u1)} \\ Lat_{u2} &= Lat_{split} + Lat_{(split,u2)} \end{aligned} \tag{5.3}$$

where Lat_{op} stands for the latency of the subtree rooted from operator op, and $Lat_{(op_1,op_2)}$ stands for the latency between operator op_1 and op_2.

As a more complicated example, the right DAG of Figure 5-4 chooses $filter2$ as the traversal root. Similarly, SPRAWL DP processes the blue subtrees as before. Gray operators' latency calculation is delayed until the root $filter2$ is reached. After that, gray operators' latencies are updated backwards as follows:

$$
\begin{aligned}
Lat_{filter2} &= Lat_{src2} + Lat_{(src2,filter2)} \\
Lat_{join} &= Max\ \{\ Lat_{filter1} + Lat_{(filter1,join)}, \\
&\quad\ Lat_{filter2} + Lat_{(filter2,join)}\ \} \\
Lat_{split} &= Lat_{join} + Lat_{(join,split)} \\
Lat_{u1} &= Lat_{split} + Lat_{(split,u1)} \\
Lat_{u2} &= Lat_{split} + Lat_{(split,u2)}
\end{aligned}
\tag{5.4}
$$

5.4 Query Adaptation

The SPRAWL DP Algorithms discussed in Chapter 4 and the multi-query sharing strategies introduced in this chapter can initiate an (near) optimal placement on wide-area networks. However, as new queries are added/deleted, the existing query deployment over the network may not be suitable any more. We can re-optimize the deployment by applying SPDP algorithms on the existing queries. However, when and how often to initiate the re-optimization are difficult to decide, and we leave these problems to future work.

In addition, varied input data rates and link bandwidth availability may lead to violation of network/latency constraints, and network topology changes may cause existing routing paths and network nodes to become unavailable. Hence we need to fix the constraint violation issues in such situations. We will discuss the solution in Section 5.4.1.

5.4.1 Fixing Constraint Violation

In Formula 3.1, the network constraints are calculated based on the estimated average input data rates. However, input data rates may vary to a great extent with time. If the input data rate is higher than expected at some point, some of the bandwidth constraints may not hold any more, and the corresponding network links may become bottlenecks. Similarly, as the network topology changes or network node/link availability varies, the original placement may not work any more.

One solution is to re-optimize the existing queries based on the new statistics

and re-deploy them. However, re-optimization of the entire query plan is very costly. Hence, SPRAWL only re-optimizes the operators and data links that violate the constraints.

Based on feedback it receives from individual modules, the central module of each cluster periodically identifies operators and data links that violate network constraints, removes the violated operators and data links from existing query plans, and re-deploys the removed parts using SPDP algorithms discussed in previous chapters. We can also use this method to deal with network link or node failures.

5.5 Query Deletion

Deleting a query from the network is a little tricky in SPRAWL. Since Queries share operator computation and data transmission amongst each other, operators and data links can only retire after the last query uses the operators (or links) is unsubscribed. We define a query *uses* an operator if the query gets data from the operator. As an example, if the left query in Figure 5-1a stops running, $J1$ and $F2$ can be marked as deleted, while $F1$ can not because the right query in Figure 5-1a uses data from $F1$ in the shared plan. If an operator can not retire, all the operators that have a data flow path connecting to the operator can not retire either.

Hence, the query deleting process starts from the root operator in a pre-order traversal of the query plan (top-down). If the visiting operator is not shared by other queries, we can mark the operator as deleted and continues to its children. Otherwise, the entire subtree rooted from the visiting operator can not be deleted.

Chapter 6

Experiments

In this chapter, we describe experiments we performed to evaluate three aspects of SPRAWL:

1. The effectiveness of the SPRAWL DP algorithm and multi-query sharing techniques in a real world setting, in terms of its ability to optimize system performance, resource usage, and cost (in dollars).

2. The efficiency and effectiveness of the SPRAWL DP algorithm for complex queries in very large (wide-area) networks.

3. How close to optimal the SPRAWL DP algorithm can be when constraints are reached.

Key takeaways of our experiments include:

1. The SPRAWL DP algorithm can reduce real-world (dollar) costs versus a random deployment strategy by a factor of 5 or more, when considering deployments on wide-area Amazon AWS networks.

2. The SPRAWL cost model accurately predicts the dollar cost of SPRAWL deployments.

3. Versus previous systems that perform wide-area operator placement [47], SPRAWL can perform 2–3x better.

4. The SPRAWL distributed DP algorithm performs nearly as well as the SPRAWL centralized DP algorithm in wide-area networks, and allows SPRAWL to place complex queries on networks of thousands of nodes in just a few seconds while the centralized algorithm takes several minutes.

5. In a resource constrained setting, where SPRAWL's DP is not provably optimal, SPRAWL performs nearly as well as an exhaustive algorithm, while scaling to much larger networks and more complex queries.

6.1 Experiment Settings

Our first set of experiments was run on Amazon EC2 clusters [2], using M3.Xlarge (64-bit 4 Core, 3.25 units each core, 13 units in total and 15 GB Memory) instances in several different "availability zones" around the world (US, Asia, and Europe). The underlying streaming processing system used in these experiments is ZStream [44], a stream processing system for efficient composite event pattern matching.

The second set of experiments runs on randomly generated, simulated Transit-Stub networks generated by GTITM [61]. We configure these networks to be similar in structure and size to the network settings used in the experimental evaluation of SBON [47]. SBON is a distributed placement algorithm that uses a relaxation algorithm for placing operators, which we compare directly against below. See Section 6.3 for a more detailed qualitative comparison between SPRAWL and SBON.

The third set of experiments runs on smaller random networks, in order to allow us to generate an optimal baseline in constrained cases for comparison with an optimal placement (generated by using an ILP solver or simply exhaustive search). These optimal algorithms are very slow and cannot scale to larger networks. Simulations were run on a MacBook Pro, with 2.9 GHz Intel Core i7 Processor and 8 GB memory.

Finally, we share some experiences in Section 6.5 on how cost estimates are chosen, how throughput and dollars spent relate to the choice of cost estimates, and how to quantify this relationship based on studies over Amazon EC2 clusters.

We ran queries over stock market data with the schema:

$$stocks : (timestamp, name, volume, price)$$

We generate synthetic data according to this schema and measure the maximum rate at which operators can process it. Queries are random combinations of selects, joins and aggregations; we explicitly vary the mixture and number of operators to control plan complexity and the amount and degree of operator sharing that is possible. Details of the structure and complexity of queries are given in individual experiments below.

The SPRAWL DP algorithm and multi-query sharing strategies are implemented in Java and complied as a library that can be called externally. Control Modules (central/individual) are implemented in C++, and run on Linux or MacOS.

6.2 Amazon EC2 Experiment

In our first experiment, we run an end-to-end test on Amazon EC2 clusters to demonstrate the overall effectiveness and performance of SPRAWL.

6.2.1 Network Settings

The network topology is similar to that shown in Figure 2-1a. Data is generated, transmitted and processed through three EC2 clusters: NA.Virginia, Asia.Singapore and EU.Ireland. In each cluster, there are 6 network nodes that act as sources (2 in each cluster), 24 as routers of which 12 can also be sinks (30 nodes in total). Although this is not a particularly large network, this real-world wide area configuration will allow us to assess the overall effectiveness of SPRAWL's cost estimates, DP algorithms and multi-query sharing strategies, and to measure our ability to minimize real-world operating costs (like dollars). We evaluate scalability on simulated networks in the next section.

As described in Section 2.2, nodes within the same cluster are connected by local links, and nodes between different clusters are connected by wide-area links. These

differ in two ways:

1. wide-area links have lower single-link bandwidth capacity than local links, and
2. wide-area links are more expensive (in terms of $/GB data sent) than local links.

These differences affect the weighted price w^{BW} used in Formula 3.1. We discuss data links and weighted price estimation in more detail in Section 6.5.

We do not compare with SBON in this experiment because SBON relies on network node coordinates to map from the cost space to corresponding network nodes, which likely will not be accurate in the EC2 virtualized environment.

6.2.2 Query Settings

The experiment is run as follows: a new, randomly generated query subscribes to each cluster every 120 seconds, until 100 queries have been created. Queries are random combinations of selects and joins that filter on *name* and *price* attributes, and perform equi-joins over *name*, each with 8-12 operators.

Streams are generated at the maximum rate from all 6 network source nodes. Source nodes produce data with varying distributions on *name* and *price*. We choose queries such that at least 30% of the queries can be shared. For each shareable query, it can be shared with at most 15% of the other shareable queries.

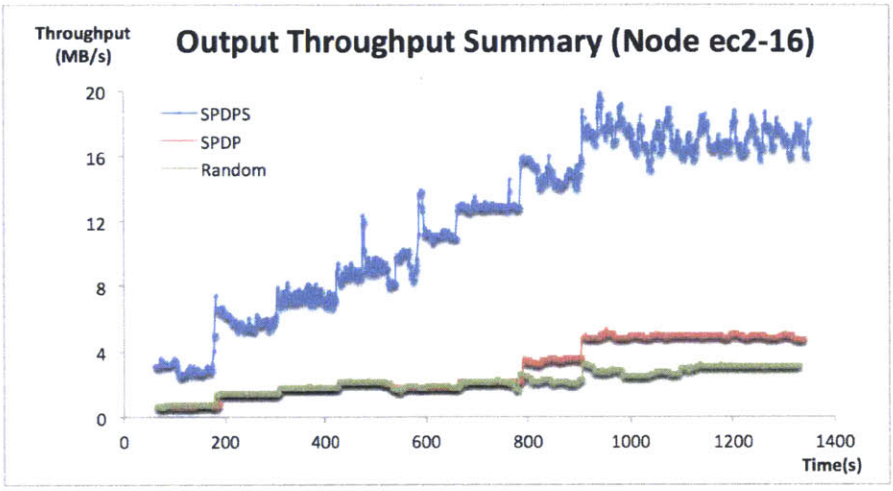
6.2.3 Deployment Settings

We experiment with three different deployments:

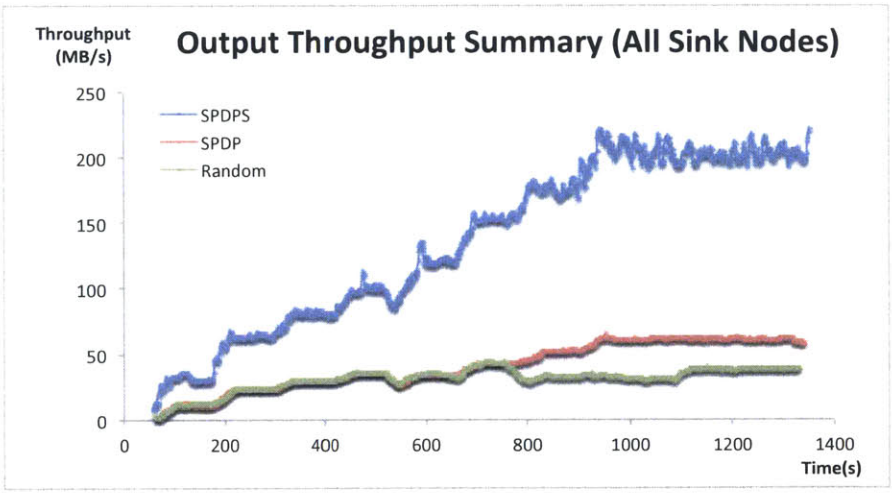
1. *SPDPS* uses both SPRAWL DP algorithm and the SPRAWL multi-query sharing strategy;
2. *SPDP* only uses the SPRAWL DP algorithm without sharing strategies;
3. *Random* is a random placement strategy where operators are initially placed on the source node, then on a router in the cluster of the source, then on a router

in the cluster of the destination, and finally on the destination node, where transitions between these different nodes are randomly chosen.

6.2.4 Output Throughput Performance



(a) Output Throughput Summary of Node EC2-16



(b) Output Throughput Summary with all Sink Nodes

Figure 6-1: Output Throughput of Wide Area Experiment

Figure 6-1a shows the output throughput of a single sink node, Figure 6-1b shows the output throughput in aggregate over all sink nodes.The X-axis shows the elapsed time running of the experiment. Throughput climbs as new queries start, one after

the other.

Figure 6-1 indicates that *SPDPS* placement outperforms the other two placements significantly, obtaining overall throughput that is roughly 3x *SPDP* and 5x *Random*. We also notice that *SPDP* performs roughly the same as *Random* at the beginning, but outperforms *Random* significantly after about 70 queries are added to the system. This is because initially network resources are roughly symmetric, e.g., each machine has about the same CPU and network capacity and weights. However, after a number of queries have been added, when some of the machines are more loaded than others, and data links are at different levels of saturation, the SPRAWL DP algorithm is better able to make use of available resources, even when no sharing strategy is used. Overall, *SPDP* is able to achieve about 1.8x output throughput of *Random*.

6.2.5 Dollar Cost

SPDPS	SPDP	Random
$20.85	$98.62	$135.33

Table 6.1: US Dollars Paid for Running 60GB Data

Table 6.1 shows the US dollars we paid for processing all of these queries. We have 6 source nodes, 10 GB of input data, amounting to about 60 GB total data sent per query, with 100 queries running in total on AWS. These experiments were run in AWS during 2011; these costs are dominated by wide-area networking costs (at the time inter-zone bandwidth cost about $.20/GB). From the table we can see that *SPDPS* placement costs 6x fewer dollars in comparison to the *Random* strategy. We discuss how our cost model is related to the system performance and to the dollars paid in detail in Section 6.5.

6.3 SPRAWL on Wide Area Networks

In this section, we compare the centralized and distributed SPRAWL DP algorithm with our own implementation of the placement algorithm described in the SBON

paper [47] on wide area networks.

6.3.1 Network Settings

We simulate the placement algorithms on 5 randomly generated transit-stub networks with 1550 nodes each. These networks each have 10 transit domains, each with 5 transit nodes. Each transit node has 3 stub domains, each with 10 stub nodes. This configuration is similar to that used in the original SBON paper [47].

6.3.2 Query Settings

Input queries are random combinations of selects, joins and aggregations of varying number of operators per query (between 4 and 12). Source and sink operators are uniformly distributed throughout the 10 clusters. Queries are registered to the cluster where the results are delivered one by one until the network is slightly saturated (we describe a test with different levels of network saturation in Section 6.4).

6.3.3 Deployment Settings

We compare against four types of deployment in this experiment:

1. *SP-Central* is the centralized SPRAWL DP algorithm;
2. *SP-Distribute* is the distributed SPRAWL DP algorithm;
3. *Relaxation* is the placement algorithm used in SBON;
4. *Random* is the same as that in the *Amazon EC2 experiment* (in Section 6.2).

Relaxation is one of the classic algorithms used to solve discrete optimization problems. We ran the relaxation algorithm in these experiments until it converged. For fairness of comparison, we do not employ the SPRAWL multi-query sharing strategies (since SBON employs different multi-query strategies, targeting reusing existing operators) and adjust our cost objectives to be the same as SBON's, since SBON does not consider CPU cost, as follows:

$$\begin{aligned} &\min_{x}\{\sum w_j^{BW} \times BW_j \times Lat_j^2\} \\ &\text{such that, } CPU_i \leq CPU_I,\ BW_j \leq BW_J, \text{ and } Lat_q \leq L_q \end{aligned} \tag{6.1}$$

SBON applies the relaxation placement on a virtualized cost space, and then maps the placement on the cost space onto the real physical network nodes. Since the real routing path is different from the virtual path in the cost space, it is difficult to include weights and constraints in the virtual cost space. This is the reason we choose to stop registering more queries when the network becomes slightly saturated.

6.3.4 Placement Cost on Wide-Area Networks

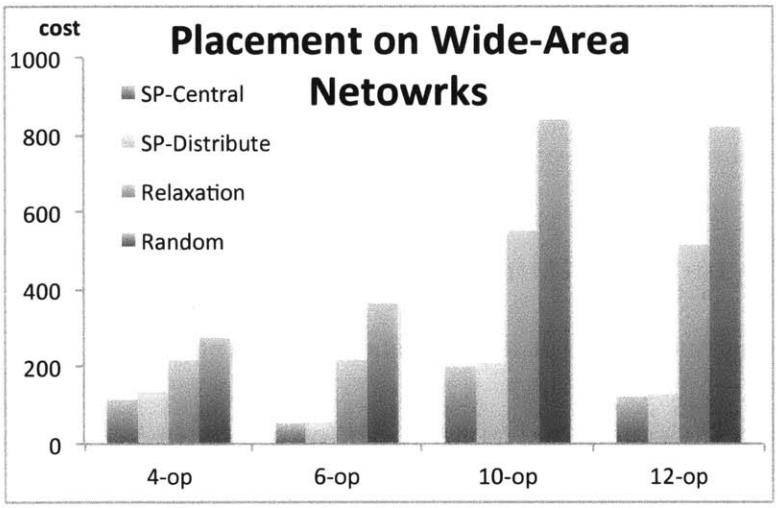


Figure 6-2: Cost Per Query on 1550-Node Transit-Stub Networks

Figure 6-2 shows the normalized average placement cost of each query placed by the four deployments. The cost is calculated based on the adjusted cost objectives in Formula 6.1. The X axis shows the number of operators in each query. We ensure that there is at least one join in *4-op* and *6-op* queries and at least three joins in *10-op* and *12-op* queries, because pure single input/output operators are not interesting in a cost model that only considers bandwidth cost (we can simply put them on one network node and avoid network transmission as much as possible). Note that the 6-op queries outperform the 4-op queries because both usually have just one join, and

the 6-op queries include additional filters. The 12-op queries do better than the 10-op queries for similar reasons.

Figure 6-2 shows that *SP-Distribute* can almost always make placement decisions as well as *SP-Central*. This is because the cost objectives used for *SP-Distribute* sub-plan partitioning are well-suited to the wide-area network (*Transit-Stub*) structure. For a random network topology, *SP-Distribute* may not work as well.

Figure 6-2 also shows that *SP Distribute* performs 2–3 times better than *Relaxation*, and 2–5 times better than *Random*. We notice that *Relaxation* is very sensitive to the number of edges in a network, as we will discuss more in Section 6.3.7. For a complete graph, it is only $10\% - 15\%$ worse than *SP Distribute*, but for transit-stub networks that usually have low edge connectivity, it does not work well. This is because the mapping from cost space to network nodes is more accurate in a more highly connected network, and because relaxation has more degrees of freedom when the network is highly connected.

6.3.5 Similarity of SP-Central & SP-Distribute

	Same Placement %
4-op	51.7%
6-op	83.0%
10-op	27.6%
12-op	37.0%

Table 6.2: Similarity of SP-Central & SP-Distribute placement

If a query plan is partitioned *correctly*, *SP-Distribute* should ideally have the same placement as *SP-Central*. *Correctly* means if a sub-plan is partitioned by *SP-Distribute* to a cluster C, every operator in the sub-plan should be placed into C by *SP-Central*.

Table 6.2 shows the percentage of the queries for which *SP-Distribute* chooses the same placement as *SP-Central*. The rates shown are not high, which means the sub-plan partitions are not *too* good, suggesting that the second-level SPRAWL DP in

SP-Distribute is doing something useful. Especially as, overall, the actual placement found by the SPRAWL DP algorithms is not far from the optimal.

6.3.6 Placement Time on Wide-Area Networks

	SP-Central	SP-Distribute
4-op	0.21 s	0.02 s
6-op	1.66 m	1.68 s
10-op	3.11 m	3.57 s
12-op	5.26 m	4.96 s

Table 6.3: Placement Time Per Query on 1550-Node Transit-Stub networks

Table 6.3 shows the average placement time per query on the 1550-node transit-stub networks by centralized SPRAWL DP and distributed SPRAWL DP, respectively. We simulate a 150 ms round-trip message time on wide-area links to account for increased communication cost.

The placement time increases rapidly as the number of operators in a query increases. The placement time of the centralized SPRAWL DP increases to 3 – 5 minutes when deploying a 10-op query, which is not acceptable in real time streaming processing networks. In contrast, the distributed SPRAWL DP only takes 3 – 4 seconds to make a placement decision. This is exactly the reason why we need the distributed version of SPRAWL. As we will show in Table 6.4 in Section 6.4, an exhaustive algorithm searching for optimal placement takes more than 5 hours to place a 7-op query on a network with only 30 nodes. We expect exhaustive placement time is much longer with larger networks.

Our evaluation of SBON is a centralized simulation of a distributed algorithm and takes only a few seconds to converge; SBON does not report the distributed version convergence time in the paper. In reality, we believe the distributed relaxation algorithm would take much longer to execute.

6.3.7 Network Edge Connectivity

As stated in Section 6.3.4, the relaxation algorithm is sensitive to edge connectivity and it does not perform well on networks with low edge connectivity. Transit-Stub networks usually have edge connectivity around 0.01. Hence in this section, we study the placement performance of SPRAWL and the relaxation algorithm on more connected networks.

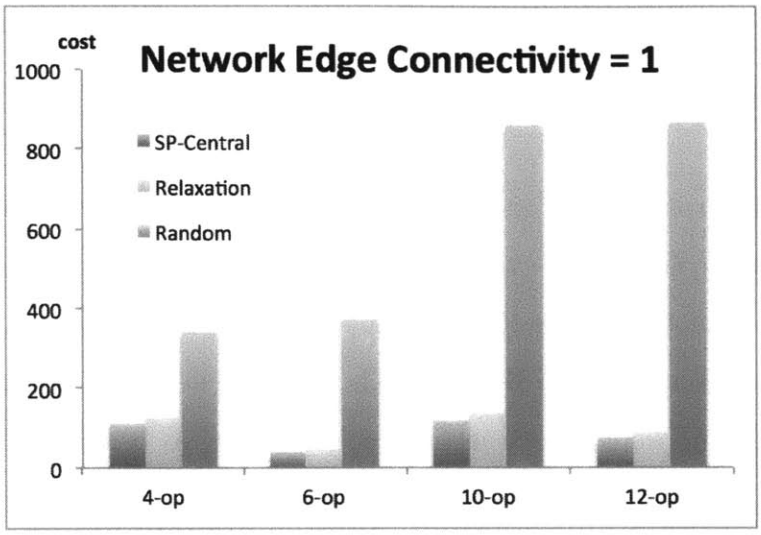
We use GTITM to generate 50-node random networks in 100×100 scale in this experiment. A fully connected network has $N(N-1)/2$ edges, where N is the number of network nodes. For example, a 1550-node network has more than 1.1 million edges, which may not fit in the memory. Hence, we choose small networks in this experiment.

Query settings and deployment settings are the same as before, except that *SP-Distribute* is not tested in this case because random networks do not have border nodes to provide routing information and it is difficult to partition random networks reasonably. We ran 10 randomized trials for each edge connectivity rate.

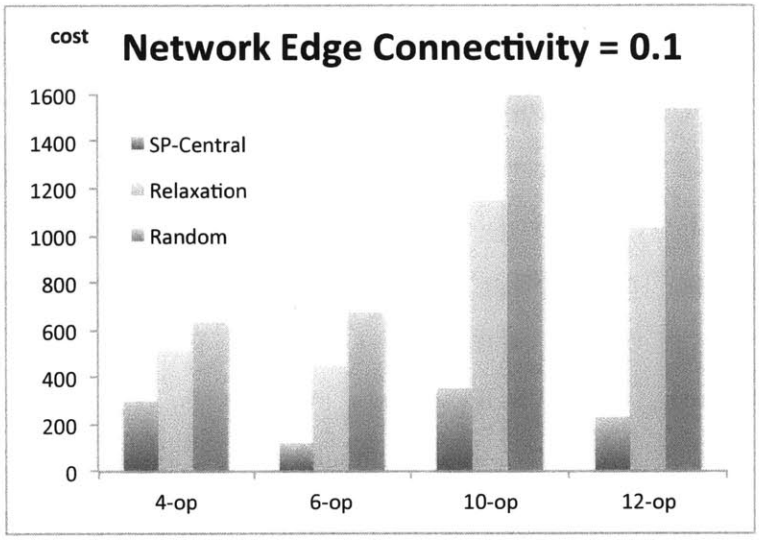
Figure 6-3a and Figure 6-3b show normalized average placement cost of each query placed by the *SP-Central*, *Relaxation* and *Random* deployment, with edge connectivity rate equal to 1.0 and 0.1 respectively. In the case of a fully connected network (Figure 6-3a with edge connectivity rate equal to 1.0), *Relaxation* performs much better than it does on Transit-Stub networks as shown in Figure 6-2. It uses 15% more network resources per query than *SP-Central* on average, which is consistent with the results reported in the SBON paper [47]. In the case of a less connected network as shown in Figure 6-3b (edge connectivity rate equal to 0.1), *Relaxation* performs similar to that of Transit-Stub networks. It uses 2x – 3x more network resources per query than *SP-Central*.

6.4 SPRAWL With Constraints

As noted in Section 4.2, SPRAWL can generate an optimal placement without constraints on CPU, bandwidth, or latency (Algorithm 1). We showed this can be ex-



(a) Cost Per Query with Network Edge Connectivity Rate Equal to 1



(b) Cost Per Query with Network Edge Connectivity Rate Equal to 0.1

Figure 6-3: Placement Cost with Different Network Edge Connectivity

tended to handle the case with constraints in Section 4.3 (Algorithm 2 – 3). In our third set of experiments we measure how well SPRAWL performs in a resource constrained setting in comparison to an exhaustive algorithm.

We compare SPRAWL DP to the exhaustive algorithm in two settings:

1. In a resource-constrained network where resources are easily violated, checking if SPRAWL can fit as many queries into the network as an exhaustive enumeration algorithm.

2. In a wide area network where a number of queries are to be deployed, checking if SPRAWL can adequately allocate resources amongst queries.

The optimal algorithm we compare SPRAWL DP to is simply an exhaustive search (Exhaustive for short) that iterates through each possible placement and chooses the optimal one. The time complexity of Exhaustive is $\Theta(N^M)$, where N is the number of network nodes, and M is the number of unpinned operators. Table 6.4 shows the runtime of SPRAWL and Exhaustive to generate a placement over a 3 cluster wide-area network similar to that we use in Section 6.2. Exhaustive's running time increases rapidly as the number of unpinned operators grows.

	SPRAWL	Exhaustive
6-op	2 ms	56 ms
10-op	4 ms	1 m
12-op	10 ms	> 5 h

Table 6.4: Runtime of SPRAWL vs Exhaustive on 3-Cluster Network

6.4.1 Resource-Constrained Network

In this experiment, we deploy a single query onto a resource-constrained network to compare the percentage of queries we can place and the placement cost between SPRAWL and Exhaustive. Placement cost is calculated according to Formula 3.1.

Query Settings

As illustrated in Table 6.4, Exhaustive's running time becomes very long when the number of unpinned operators exceeds 5. Hence in this experiment, we choose a 10-operator query as our benchmark: 3 unpinned joins, 1 unpinned filter, 1 unpinned aggregation, 4 pinned sources, and 1 pinned sink. Exhaustive can not work on any bigger queries, while smaller queries are easier to fit in and are less interesting.

Operator connections and filter selectivity (in the range [0,1]) on these connections, as well as latency bounds are all randomly generated. Latencies are chosen to range between the shortest to the longest path latency from network source to sink.

Network Settings

The network topology we use in these experiments are similar to a single cluster of Figure 2-1a. Network node CPU capacity, link capacity, weight(1, 3), and latency are also randomly selected. We choose these so that each network node can support 1 to 10 source operators, and each network link can handle the output of 0.5 to 3 sources. Network source links (from a source to other nodes) are guaranteed to have capacity to support at least one source link.

Deployment Settings

We ran 5 randomized trials, placing 1000 queries per trial, and computed the average percentage of successfully placed queries and corresponding placement cost for SPRAWL and Exhaustive. Numbers in this experiment were selected to make the placement quite difficult, such that only a small fraction of the total queries could actually fit. We expect in practice that real deployments would not be so heavily constrained.

Query Fit Rate & Placement Cost

Table 6.5 and 6.6 shows the average percentage of queries fit onto the network and average per query placement cost. Table 6.5 is with all types of constraints and

	SPRAWL	Exhaustive
Avg. Query Fit Rate	29.7%	34.2%
Per Query Placement Cost	236.4	226.3 (232.3)

Table 6.5: Query Fit Rate & Placement Cost in Resource-Constrained Networks with all Constraints

	SPRAWL	Exhaustive
Avg. Query Fit Rate	37.4%	43.0%
Per Query Placement Cost	235.8	226.8 (236.1)

Table 6.6: Query Fit Rate & Placement Cost in Resource-Constrained Networks with CPU, BW Constraints

Table 6.6 is only with CPU and BW constraints. Since Table 6.5 is with more constraints, the fraction of queries that fit is lower than that in Table 6.6.

In a resource-constrained network (only 34.2% queries are successfully placed), SPRAWL can fit in about 88% (SPRAWL rate/ Exhaustive rate) of the queries Exhaustive can fit. The number in brackets in Table 6.5 and 6.6 is the real per query cost for Exhaustive. However, Exhaustive may fit in some costly queries that can not be fit by SPRAWL, so we adjust this number by excluding those queries that SPRAWL cannot fit.

From Table 6.5 and 6.6, we can see that the SPRAWL per query cost is just 4% higher than Exhaustive, and that it is able to place most queries, even in this demanding, highly constrained case. We say a network is *heavily resource-constrained* to a type of query if the network has high probablity ($> 50\%$) not able to find a placement for a single query of that type by Exhaustive. We show in the next experiment that SPRAWL can do as well as Exhaustive when the network is not so heavily resource-constrained.

6.4.2 Network Resource Allocation

In this experiment, we want to show that SPRAWL can efficiently allocate network resource to multiple queries, and show that SPRAWL can do as well as Exhaustive when the network is not heavily resource-constrained.

Query & Network Settings

Queries are continuously deployed onto a wide area network similar to Figure 2-1a, until no more queries can be supported by the network. We compare the number of queries successfully placed by SPRAWL and Exhaustive, and compare their average placed query cost. We use the same type of queries used in last experiment. Again, all queries are generated randomly, and no sharing strategies are considered in this case. To make the network *not heavily resource-constrained*, we only choose randomly generated networks able to fit more than 100 queries by Exhaustive.

Number of Queries Supported & Placement Cost

	SPRAWL	Exhaustive
Total No. Query Supported	128	129
First Stop Avg. Cost	258.7	257.8
Five-seq. Stop Avg. Cost	297.1	295.8

Table 6.7: Multi-Query With All Constraints

	SPRAWL	Exhaustive
Total No. Query Support	128	127
First Stop Avg. Cost	271.6	270.7
Five-seq. Stop Avg. Cost	297.1	296.5

Table 6.8: Multi-Query With CPU and BW Constraints

Table 6.7 shows results with all types of constraints and Table 6.8 shows results without latency constraints. Table 6.7 and 6.8 show that SPRAWL and Exhaustive can support roughly the same number of queries. This indicates SPRAWL can allocate resources as efficiently as Exhaustive.

First Stop Avg Cost shows the average deployed query cost until either SPRAWL or Exhaustive fails to find a placement for a query. This indicates the network is slightly saturated. *Five-seq. Stop Avg Cost* shows the average query cost when either SPRAWL or Exhaustive first fails to place five successive queries. In this case, the

network is near saturation. We terminate the experiment when the system fails to place 20 successive queries.

As we can see from Table 6.7 and 6.8, the average placed query cost of SPRAWL and Exhaustive are almost always the same. This is because most queries placed do not saturate any constraints, in which case, SPRAWL generates the same optimal placement as Exhaustive. Even if a query reaches constraints, SPRAWL can often generate a good placement, as we have already demonstrated in the previous experiment (Section 6.4.1).

First Stop Avg. Cost in Table 6.7 is smaller than that in Table 6.8 because Table 6.7's first stop is due to latency constraints, while Table 6.8's is due to CPU or BW saturation. Before *first stop*, the sets of queries supported by SPRAWL and Exhaustive are the same. From *first stop* to *five-seq stop*, the network becomes more and more resource-constrained, and the sets of queries supported by these two algorithms differ. Note, however, that the two algorithms can support roughly the same number of queries with the same average placement cost.

6.5 Amazon EC2 Cost Estimates Study

In this section, we share some experiences on how cost estimates are chosen, and how throughput and dollars spent relate to the choice of cost estimates.

6.5.1 Join Placement

The first experiment is designed to place a join query in between amazon EC2 zones from different areas to check the impacts of different input rates and join selectivities.

Query & Network Settings

The join query is similar to that shown in Figure 2-3, with one source ($src1$) located in US.Virginia and the other ($src2$) in Asia.Singapore, and results delivered to the Asia.Singapore availability zone. The ratio of input data rate of src1 to src2 is around

10 : 1. We vary the selectivity of the join by varying the number of *src*2 tuples that join with each *src*1 tuple (we call this the *window size*).

Deployment Settings

SPRAWL identifies two optimal deployments (PlaceA and PlaceB). PlaceA is optimal when windowsize=20, and PlaceB is optimal when windowsize=1. The difference between the two placements is the node on which the join operator is placed. In PlaceA, the join runs in the Asia.Singapore zone. In PlaceB, the join runs in the US.Virginia zone. PlaceA is better than PlaceB if the cost of transmitting join output and src2 over the wide-area network is higher than that of transmitting src1.

Throughput & Bandwidth Usage

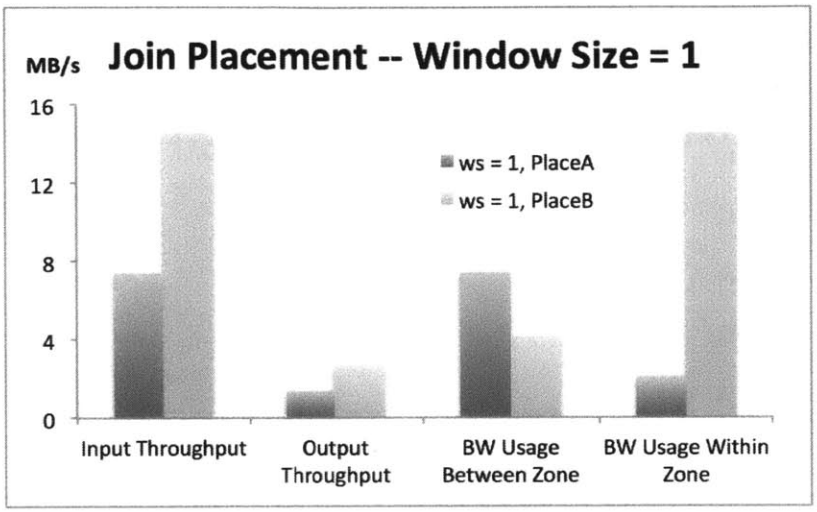
Figure 6-4 shows the input and output throughput as well as intra and inter bandwidth usages of these two placements, with window size equal to 1 in Figure 6-4a and 20 in Figure 6-4b.

In Figure 6-4a, PlaceB uses much more total bandwidth than PlaceA, but since most of its bandwidth usage is within zones, its overall cost is low because intra-link bandwidth costs are negligible in AWS. Since data transmission between different zones is much slower and more expensive than within the same zone, PlaceB is actually cheaper than PlaceA in this case.

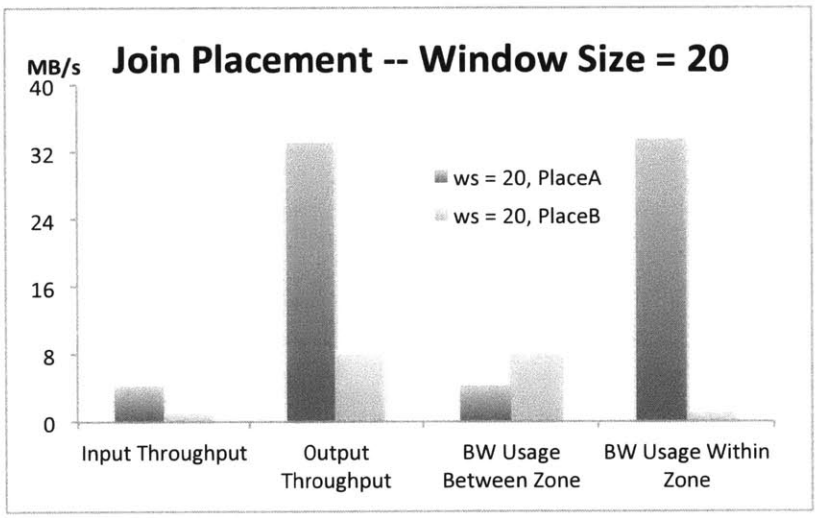
In Figure 6-4b, when the join window size is set to 20, the join output size is about 7.6x the size of src1. In this case, PlaceA has much lower bandwidth usage between zones than PlaceB, making it preferable.

Estimate Cost & Dollar Cost

Figure 6-5a shows the bandwidth cost estimated from our cost model (Function 3.1). Here, the weight w^{BW} for each link within the same zone is the same. We show how to determine the weight of links between different zones and within the same zone by

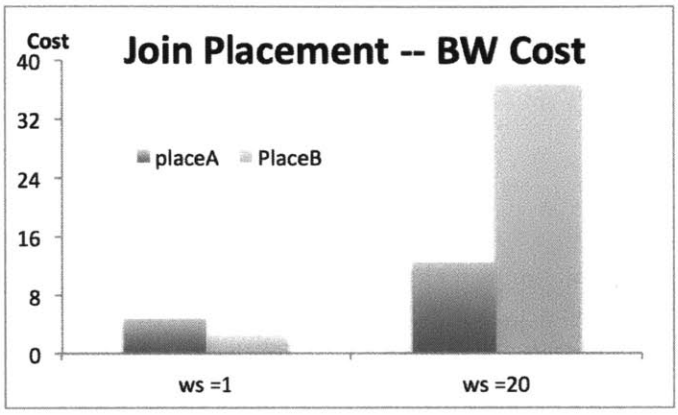


(a) Join Placement with Window-Size Equal to 1

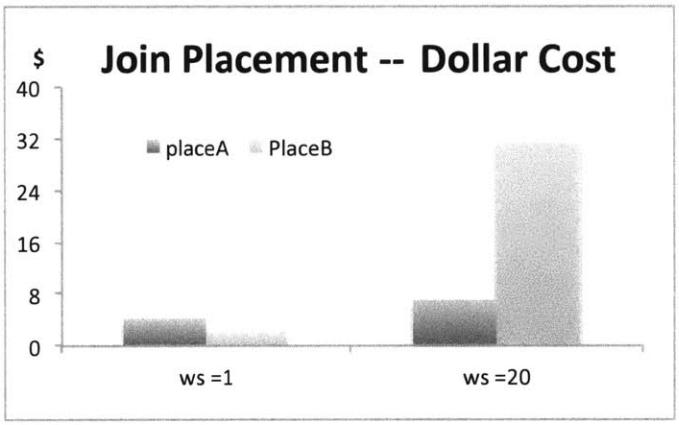


(b) Join Placement with Window-Size Equal to 20

Figure 6-4: Join Placement Performance in between Different Zones



(a) Bandwidth Cost Estimated by Cost Model



(b) US Dollars Paid for Processing 50GB Data

Figure 6-5: Join Placement Cost in between Different Zones

using results in the next experiment (Section 6.5.2).

As we will show in Section 6.5.2, a local link within Asia or US can provide about 16.3 MB/s bandwidth, and a wide-area link between Asia and US about 7.2 MB/s. Hence, the weight of a link between zones is $16.3 \div 7.2 = 2.26$. We also need to adjust this weight according to the real price to send data through links. For example, if the price to send data between Asia and US is twice as much as that to send data within Asia or US, the weight of wide area link versus a local link is $2.26 * 2 = 4.52$x.

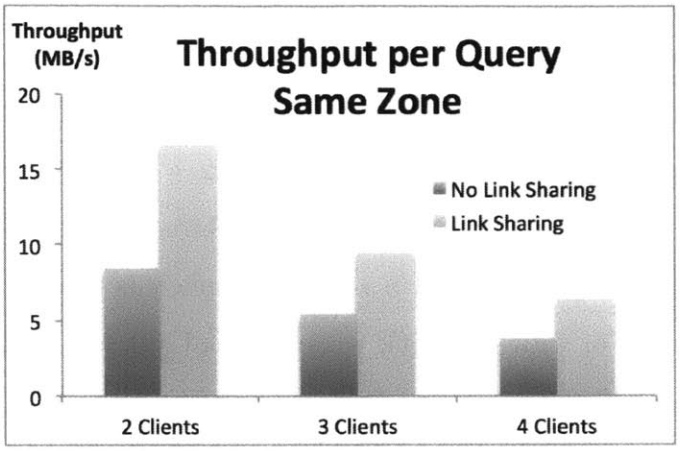
Figure 6-5b illustrates the cost in dollars for running the join experiment and processing around 50 GB src1 input data. By comparing Figure 6-5b to Figure 6-5a and Figure 6-4, we can see that our cost model does a good job of tracking system performance and total money cost.

6.5.2 Link Sharing

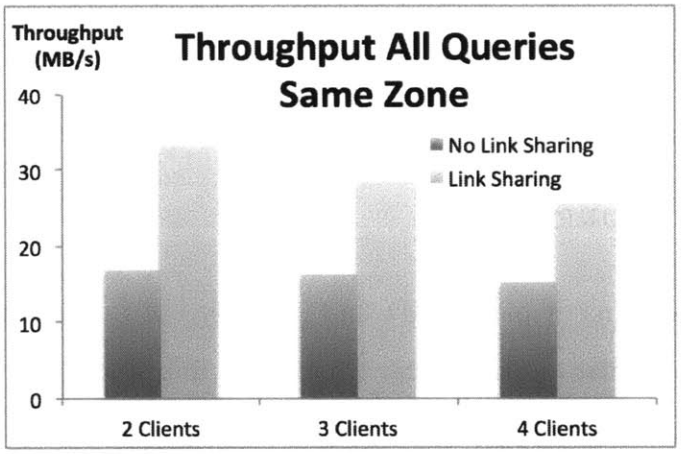
The goal of this experiment is to study and quantify the performance improvement of local and wide-area data link sharing, where data link sharing means only a single copy of each tuple is transmitted to a node even if many operators process it. We can use the results from this section to calculate network parameters in the cost model, as we did in Section 6.5.1.

Query, Network and Deployment Settings

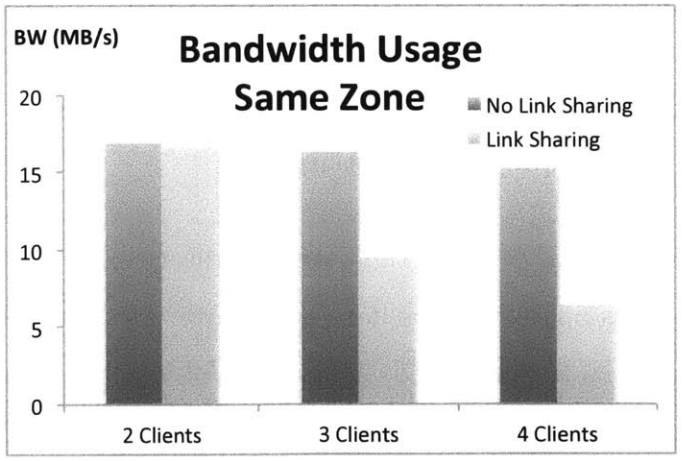
The experimental setup is as follows: the data generator runs on one EC2 node, and produces data as fast as possible. This data is sent over the network to another EC2 node, which runs a variable number of clients, each of which subscribes to the same set of stock symbols. Data is either sent without link sharing (*No Link Sharing*), resulting in multiple copies being sent, or with sharing (*Link Sharing*), where one copy is sent and then split amongst the clients once it arrives. This experiment is network-bound (neither node's CPU is overloaded).



(a) Throughput per Query in the Same Zone



(b) Throughput all Query in the Same Zone



(c) Bandwidth Usage in the Same Zone

Figure 6-6: Link Sharing Within the Same Zone

Throughput & Bandwidth Usage within the Same Zone

Figure 6-6 shows the performance improvement of local link sharing within the same zone. Figure 6-6a shows the average throughput on a single client in MB/sec; Figure 6-6b shows the overall system throughput; and finally, Figure 6-6c shows bandwidth usage.

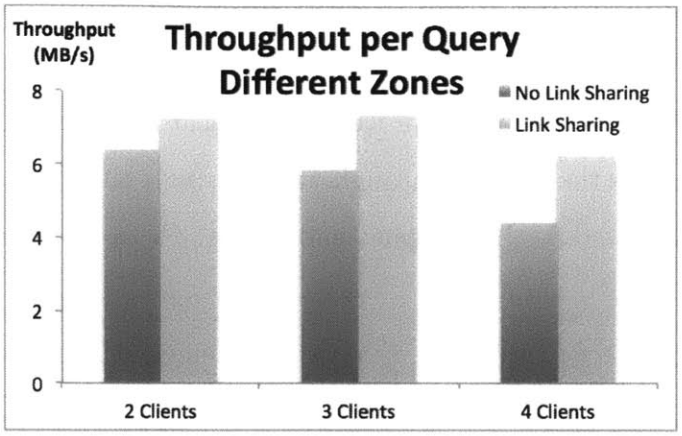
From these three graphs, we can see that the throughput is improved significantly by local link sharing, both from the perspective of a single client and the entire system. Figure 6-6b shows that the overall throughput of the *No Link Sharing* strategy in all cases (2-4 clients) is roughly the same, about 16.3 MB/s. This is because the *No Link Sharing* strategy sends multiple copies of data through network, and is bounded by the network bandwidth. This experiment shows that local link bandwidth is capped at 16.3 MB/s.

Throughput & Bandwidth Usage in between Different Zones

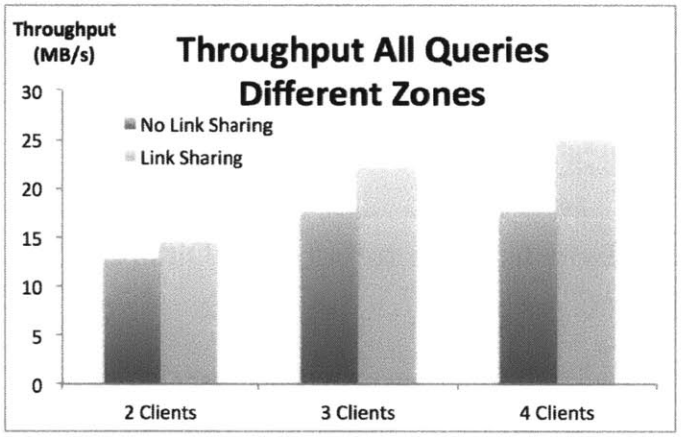
Figure 6-7 shows the results of the same experiment setting as in Figure 6-6, but with wide-area link sharing. As expected, the results are somewhat different than with local link sharing.

Comparing Figure 6-6 and Figure 6-7, we can see that wide-area link sharing does not offer as much performance improvement as local link sharing when sending two or three copies of data (with two or three clients). This is because transmitting data through a *single socket* between different zones is limited by Amazon at around 7.2 MB/s, but multiple sockets can achieve higher throughput (up to the 16.3 MB/s as measured in the previous experiment). Hence, the throughput of the *Link Sharing* and *No Link Sharing* strategies is similar until the *No Link Sharing* strategy reaches network saturation (16.3 MB/s).

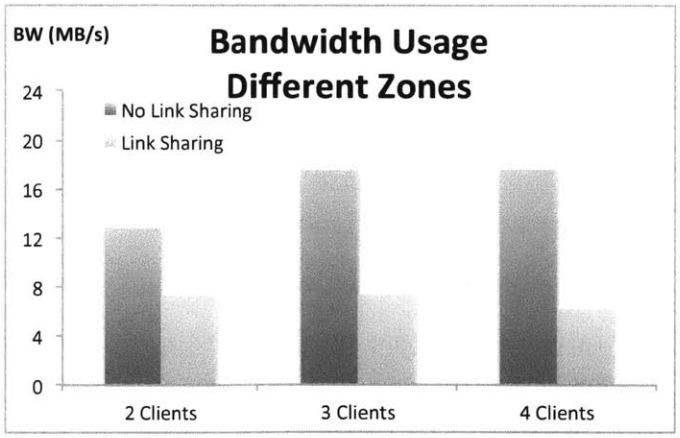
As shown in Figure 6-7a, the throughput of both the *No Link Sharing* and *Link Sharing* strategies is around 7.2 MB/s when sending two copies of data. The throughput gap between those two strategies increases quickly as more copies of data are sent



(a) Throughput per Query in between Different Zones



(b) Throughput all Query in between Different Zones

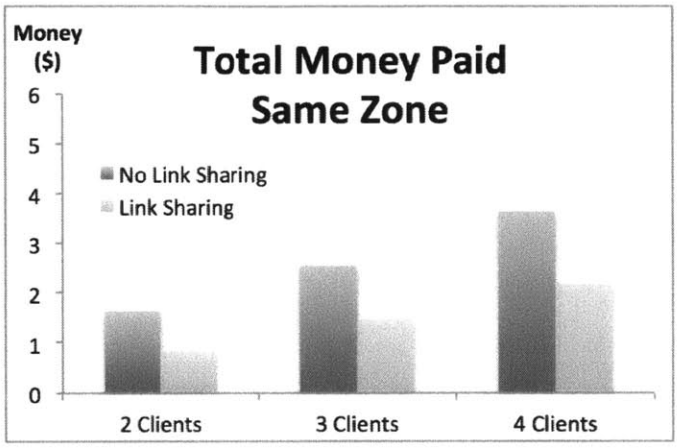


(c) Bandwidth Usage in between Different Zones

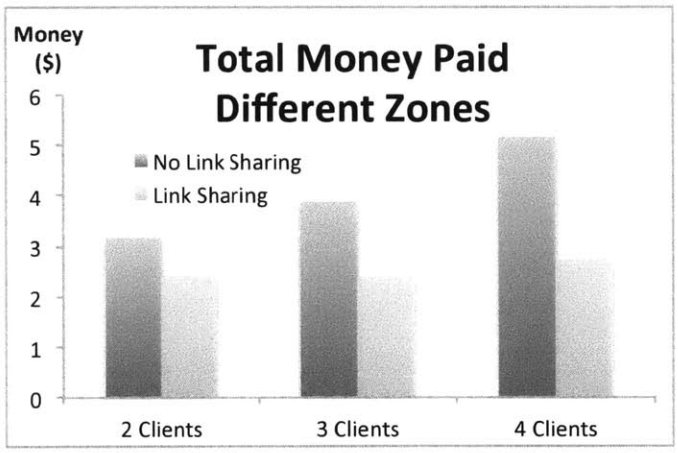
Figure 6-7: Link Sharing in between Different Zones

(when the network is saturated). Figure 6-7c shows the bandwidth usage of the *No Link Sharing* and *Link Sharing* strategies between different zones. Since data transmission between zones is expensive, the *No Link Sharing* strategy is particularly costly in this case.

Total US Dollar Cost



(a) US Dollar Cost within the Same Zone



(b) US Dollar Cost in between Different Zones

Figure 6-8: Cost in US Dollars for Processing 50 GB Data

Figure 6-8 shows the total US dollars we paid to process about 50 GB of data in this experiment. The cost consists of two parts: the cost to rent machines, and the cost to transmit data. System throughput determines how long the machines need to

be reserved, and bandwidth usage determines the payment for data transmission.

Chapter 7

Related Work

Distributed database systems have been studied extensively since the 1970s [57, 14, 53]. At that time, the research topics are mainly focused on distributed data management collaborating at different locations for large corporations. Ever since then, many different architectures and technologies have been proposed and developed, like distributed query optimization [26, 29], stream processing [9, 13, 12, 17, 18], sensor network data propagation [42, 16, 22], publish/subscribe systems [15, 24, 45], graph partitioning [38, 32, 11], overlay networks [52, 35, 62], and network awareness data/stream processing [10, 47, 39, 46].

In this chapter, We will first have a brief review of such systems, and then describe and compare with SPRAWL the five closely related systems mentioned in Chapter 1 (Min-Cut [41], SBON [47], SAND [10], SQPR [37] and SODA [37]) in detail.

7.1 Distributed Query Optimization

Distributed query optimization becomes important ever since distributed data management is popular and commercialized [26, 29]. The distributed query optimization techniques are designed to solve the client-sever site selection problem for query operators in the context of heterogeneous environment. They also includes special join techniques(like horizontally partitioned joins, semijoins [14], double-pipelined hash joins [56]), intra-query parallelism architectures(MapReduce [21]), and dynamic data

replication algorithms [19, 59] to reduce communication costs and improve query performance.

However, none of these researches consider resource-aware in-network processing. In addition, all these techniques are designed for stored data, hence many of them are not suitable for dealing with streaming data.

7.2 Stream Processing Systems

A number of stream processing systems [9, 13, 12, 17, 18] have been developed as real time data processing becomes a critical requirement for applications like financial data services, social networks, and real time monitoring. However, most of these systems are centralized: data is streamed from different data sites into a central cluster or warehouse where these streams are processed according to the continuous queries issued by users.

Some of the stream processing systems have been extended to distributed versions to solve the problem of adaptive load-shedding for long-running continuous queries, like Borealis [13, 54] and Flux [51].

Other distributed stream processing systems address the problem of supporting huge number of queries in a large scale Internet. NiagaraCQ [18] achieved so by grouping continuous queries sharing similar structures. However, it is not network resource-aware either.

7.3 Sensor Networks

In-network data dissemination has been studied in sensor networks [42, 16, 22] for a long time because resource-efficient data propagation is critical in sensor networks. Sensors usually have very limited resources (like power and bandwidth). Hence, data propagation efficiency determines the period of time sensor networks can last and the overall amount of data sensor networks can collect.

However, different from normal distributed data systems, sensor networks are

usually self-organized, and highly dynamic, with each sensor node having limited resources and restricted network topology information. In addition, sensor nodes themselves are unstable and can be unavailable at any time.

Hence, data dissemination research in sensor networks is mainly focused on issues like optimal routing path search based on limited information and fault-tolerance if some sensor nodes are suddenly broken. Such issues are not the focus in this dissertation since wide-area data centers as shown in Chapter 2.2 are configured in advance and stable most of the time.

7.4 Pub-Sub Systems

Pub-sub systems are stream processing systems specifically designed for message updating [15, 24, 45]. They collect data from publishers, and deliver updated results to subscribers according to their subscriptions in a timely fashion. Most of these pub-sub systems are centralized. Some of them are extended to distributed model to solve Internet-scale message delivering [23].

Pub-sub systems mainly focus on massive message updating. Generally it supports *XPath* queries or simple XML-based filtering instead of computational-heavy operators like joins and aggregations. Besides, the first importance in pub-sub systems is to deliver message updates as fast as possible. Hence pub-sub systems are not network resource-aware either.

We can also consider SPRAWL as a distributed network resource-aware pub-sub system.

7.5 Graph Partitions in Parallel Computation

The problem of partitioning irregular graphs and allocating distributed tasks in a heterogeneous environment has been long studied in the parallel computation community [38, 32, 11]. Graph partitions in parallel computation is designed for compute-intensive applications in scientific and engineering domains, like computational fluid

dynamics(CFD) [55] and VLSI testing [36].

These applications are modeled as a weighted undirected graph $G = (V, E)$, referred to as a *workload graph*. Each vertex has computational weight which reflects CPU units it needs, and each edge has a communication weight which reflects data transit on it. So the partition problem is to find a good partition for graph G so as to minimize the resulting data flow between partitions.

Graph partitions is network resource-aware, however, it differs from our scenario in three ways:

- it does not allow application-specific features like selectivity and data rate of streams,
- it does not designed for wide-area networks (it is designed for parallel computation), and
- it does not support pinned vertices, while stream data sources and sinks are usually pinned.

7.6 Overlay Networks

Overlay networks [52, 35, 62] like peer-to-peer networks focus on the problem of locating the network node that stores a particular data item in large-scale dynamic networks as nodes are added to or leave networks.

Message routing in overlay networks is similar to data flow routing in SPRAWL. However, instead of supporting expressive and complicated operators, overlay networks only support limited message processing. We can consider SPRAWL as working on top of such overlay networks so as to support more complicated optimal "message routing".

7.7 SPRAWL Vs. Network Resource-Aware Data Processing Systems

Network awareness data processing systems [47, 41, 10, 37, 58, 39, 46] are closely related to SPRAWL. However, SPRAWL has fundamental advantages in three aspects overall:

1. SPRAWL supports distributed decision making for query sharing and placement and does not need global network information, while most existing systems do it in a centralized way [41, 37, 58, 46, 39], and are not scalable for wide-area networks. As shown in Chapter 2.2, a centralized placement algorithm may spend 3-4 minutes to place a query in a 1550-node wide-area network, not to mention tracking global network resource and topology changes.

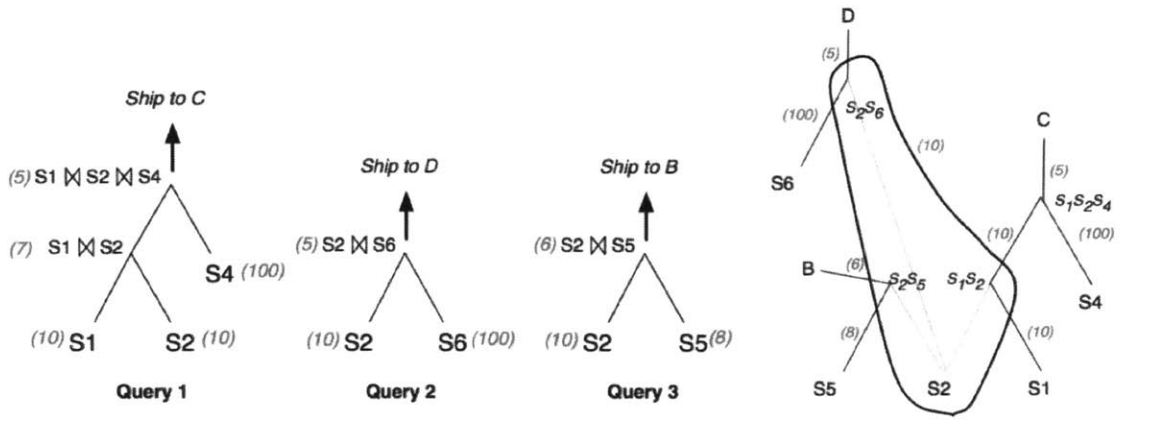
2. SPRAWL supports flexible cost objectives, including CPU cost, resource constraints, and query constraints, as discussed in Chapter 3. Most systems [41, 47, 10, 39] only consider network bandwidth cost and do not include constrained clauses due to either methodological limitations or to simplify the optimization formulation.

3. Compared to existing distributed multi-query placement systems [47, 10], SPRAWL has better placement performance and converge time as shown in Chapter 2.2.

We will discuss and compare SPRAWL with these network resource-aware systems in detail in this section.

7.7.1 Min-Cut

Min-Cut [41] is a centralized algorithm that generates an operator assignment that minimizes the overall communication cost for distributed queries. The algorithm generates a hyper-graph $\mathcal{H}_D$ for a set of input query plans, and computes a minimal



(a) Example Queries in Min-Cut (b) Hyper-graph for Query 1-3

Figure 7-1: Example Hyper-graph in Min-Cut (reproduced from [41])

cut on $\mathcal{H}_D$ for each link (x, y) in the communication (physical) network $\mathcal{G}_C$. Figure 7-1b is a hyper-graph constructed for all three queries in Figure 7-1a. The circle in Figure 7-1b is a hyper-edge for $S2$ since all three queries share the same data source $S2$.

The minimal communication cost incurred over the link (x, y) is exactly the transmission cost over the cut. After finding all the locally optimal solution for each link in $\mathcal{G}_C$, it proves and generates a global optimal solution under the assumption that $\mathcal{G}_C$ is a tree and each $\mathcal{H}_D$ for link (x, y) has a unique solution (minimal cut). Min-Cut algorithm has a $O(log(n))$ approximation for a general non-tree network $\mathcal{G}_C$.

Min-Cut is theoretically sound, but with strong restrictions, which may not be practical in real world:

1. Min-Cut needs global information to merge each local minimal cut to a global optimal solution, and is not straightforward to extend to a distributed version, so it may not be suitable for wide-area networks.

2. To obtain an optimal solution, Min-Cut requires the communication (physical) network to be a tree, which is not the case in practice.

3. Min-Cut only considers network communication cost, and it is not clear how to

include CPU cost, latency requirements, and resource constraints since Min-Cut is based on edge cut costs.

7.7.2 SBON

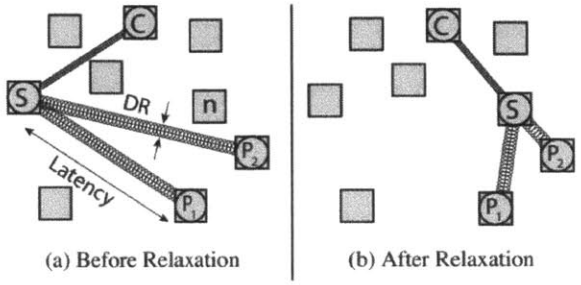


Figure 7-2: Spring Relaxation in SBON Cost Space (reproduced from [47])

SBON (Stream-Based Overlay Network) [47] is a decentralized adaptive framework for operator placement using a spring relaxation algorithm to minimize network impacts. In SBON, the query plan is considered as a collection of operators connected by springs (operator edges), and properties of physical networks (latency for example) is mapped onto a cost space.

As shown in Figure 7-2, the average force of spring i is $\vec{F_i} = \frac{1}{2}k_i\vec{s_i}$ where s_i is latency Lat_i and k_i is the data rate DR_i (similar to *Latency* and BW in SPRAWL). Based on this, the optimal operator placement is one with the lowest energy state of the system (i.e., with the lowest sum of the potential energies stored in the system). If a spring has higher energy than necessary, unpinned operators will be pulled/pushed to lower the overall system energy. After a number of relaxation iterations, the system gradually converges to a low and stable energy state. Then SBON maps the converged placement from the cost space onto underlying physical networks.

SBON scales well in wide-area networks, however,

1. The convergence properties of SBON is unclear. In the case where multiple operators are unpinned, it is hard to chose which operator to migrate and making

an incorrect choice can result in back and forth migration, causing the system to converge slowly or not at all.

2. Mapping the placement from the cost space to the real physical network is not accurate. It is also difficult to include link weights and resource constraints in the cost space since real routing path can not be mapped onto the cost space.

3. Since each SBON instance does not have global view, it is easy to 'stuck" in local minima. In addition, the cost function for relaxation algorithm must agree with the form $\vec{E_i} = \vec{F_i} \times \vec{s_i}$, making SBON's cost model restricted.

7.7.3 SAND

SAND (Scalable Adaptive Network Database) includes a set of distributed operator placement strategies to develop a highly-scalable and adaptive network-oriented database system on top of a Distributed Hash Table (DHT) [52, 62].

SAND is a greedy algorithm that deploys a query plan bottom-up in postorder, similar to ours. When encountering a new operator, SAND chooses to place the operator in one of the following four candidate locations,

- one of its children's locations,
- a *common* location,
- the end-user's location, or
- a location meeting a certain distance criterion.

The total cost of a query plan is the sum of each edge cost. Here edge costs refer to costs transmitting data between operators, and is calculated as $BW \times Distance$. BW is the output date rate of that edge, and $Distance$ is the network distance of the physical link the edge uses to send data.

A common location refers to a physical node where a placed operator and all its children can potentially be co-located. A location meeting a certain distance criterion

is used to select configurations to optimize for total cost by reducing distances between operator-child mappings. SAND has proved the probability it improves over the baseline bandwidth cost theoretically. The baseline deployment is a centralized one which places all operators on client(sink) side.

SAND experiments on top of Tapestry using Transit-Stub network topologies obtained from the GTITM [61], and show effects of different placement strategies on *bandwidth consumption ratio* (the ratio of the overall bandwidth) and latency *stretch* (the ratio of the longest path length on the network to the longest path length from the sources to the sink).

However,

1. SAND does not always generate a good placement and has no guarantee upon the efficiency of placement solution.

2. The distributed version of SAND placement does not provide a solution for sub-query placement and assignment. Instead, it only investigates one extreme of subtree assignment, a single operator is a subtree.

3. SAND does not include multi-query sharing because sharing may introduce multi-output query DAG, which can not be handled by the SAND placement algorithm.

7.7.4 SQPR

SQPR (Stream Query Planning with Reuse) [37] combines query admission, operator allocation and reuse together as a single inter-related constrained optimisation problem. To achieve so, it proposes an query planning model with four objectives:

1. O_1: maximize the number of satisfied queries

2. O_2: minimize the system-wide network usage

3. O_3: minimize the usage of computational resources, and

4. O_4: potentially balance the load between network hosts

Due to the conflicting nature of these objectives, SQPR generates Pareto efficient solutions by maximizing a weighted sum of

$$\lambda_1 O_1 - \lambda_2 O_2 - \lambda_3 O_3 - \lambda_4 O_4$$

subject to constraints

for some constants $\lambda_1, \lambda_2, \lambda_3, \lambda_4 >= 0$

The constraints mentioned in the above function include *demand constraints*, *availability constraints*, *resource constraints* and *acyclicity constraints*. The first three constraints are similar to ours described as CPU constraints and BW constraints in Chapter 3. Acyclicity constraints requires no self-feedback loops. SQPR formulates the query assignment problem as a mixed integer linear program (MILP), which can be solved by standard branch and bound algorithm [34].

The discussion and analysis of query planning and placement in SQPR is complete. However, the solution to the optimisation problem is based on mixed integer linear program (MILP), which is exponential in time and memory as the size of communication networks and the number of query operators grow. This MILP solution is not at all scalable even to hundreds of network nodes. Their simulation results are based on 50 hosts and one (two-way/three-way/four-way) join per query.

7.7.5 SODA

SODA (Scheduling Optimizer for Distributed Applications) [58] is an optimizing scheduler for System S [27], a streaming processing system that assumes offered load exceeds system capacity most of the time. Hence, SODA is designed to fully utilize each processing node in System S.

SODA scheduler divides the problem into two stages:

1. *admission control stage*: decides which jobs to admit and which template to choose for each job.

2. *operator placement stage*: generates placement for jobs admitted and templates chosen.

SODA uses *Non-Serial Dynamic Programming (NSDP)* [33] to solve operator placement problem, and treat objective functions as "black boxes", but no mention of how objective functions are constructed, how constraints are solved, and how network latency can be included. In addition, SODA is a centralized scheduler which is not scalable for wide-area networks.

7.7.6 Other Network Awareness Systems

Papaemmanouil *et al* [46] designed a network awareness system focusing on making all queries running in the system satisfy network QoS requirements. Initially, a new operator is placed on the node closest to it's input. This placement may cause violation of some network QoS requirements (similar to *constraints* in SPRAWL). Then operators are periodically moved around to adjust to the QoS requirements and network condition changes. The experiment is taken on PlanetLab deployment of S^3 [60]. It shows that by adaptively updating the global plan and deployment, they can improve the number of queries that meet QoS expectations by 39%–58%, on average. The system is target for highly dynamic networks and QoS-based services, however, it does not guarantee how good the adjusted placement is, and how long the adjustment procedure may take.

Kumar *et al* [39] constructs a three-layer architecture for the placement problem: application layer, middleware layer and underlay layer. Application layer is a declared data-flow graph expression of the query needed to be placed. Middleware layer compiles data-flow graph into separated operators and getting network resource information. Underlay layer partitions the underlay network to recursively place the operators. However, the paper neither mentions how to compile the declared data-flow graph into separate operators in details, nor describes how cost estimates are assigned. [39] includes a simulation with real data from Delta Airlines, showing that dynamic operator deployment can save on average around 15 - 18% end-to-end delay time. However, the data only contains 4 airport (network) nodes.

Chapter 8

Future Work

There are several interesting directions in our future research. As proved in Theorem 5.3.1, an undirected graph is able to achieve the same optimal cost placement no matter which operator is chosen as a root in the unconstrained case. However, in constraint cases, this theorem does not hold any more and we need to find suitable roots for the undirected graph. For example, traversing the most costly subtree first, and setting an operator with minimal $CPU + \beta * BW$ as the root may result in a smaller cost because higher-cost subtrees may have more chances to be optimally placed. In addition, choosing different operator as roots will affect individual querys cost. Hence how to balance the cost between different user queries is a challenge.

Another interesting direction is how to partition the network and queries. Currently, SPRAWL partitions networks based on the Transit-Stub network model. Although most of the wide area networks are based on this model, We would like to extend SPRAWL to more generalized networks without this Transit-Stub limitation.

As stated in Section 5.4, when more and more queries are added into the networks, the existing deployment may no longer be (near) optimal To achieve better performance, we need to re-optimize the deployment by applying SPDP algorithms on the existing queries. Re-optimization may lead to operator migration, which in most cases is costly. The benefits of re-optimization will depend on how much the new deployment saves the overall cost and on how long this strategy remains optimal for. How to measure and balance between the benefits and the costs is a challenge.

Chapter 9

Conclusions

In this dissertation, we presented SPRAWL, a resource-aware stream processing optimization and placement layer to optimize stream processing programs distributed across wide area networks. SPRAWL is able to compute an optimal in-network placement of operators in a given query plan that can minimize the overall plan cost when no constraints are present. We presented extensions to make this work in constrained cases.

We also showed how to distribute SPRAWL to optimize placements on networks with thousands of nodes, significantly beyond what the centralized algorithm can handle in reasonable time limits.

SPRAWL is also able to *share* execution of operators and data transmission, transmitting records through the network just once, and combining streaming operators that operate on the same subset of data.

Overall, we showed that this can increase throughput by up to a factor of 5 on complex mixtures of stream processing queries, and reduce overall costs (using AWS prices as a guide) by a factor of 6.

Appendix A

API for Underlying Stream Processing Systems

As mentioned in Chapter 1, SPRAWL is a stream dissemination layer that can make query optimization and placement transparent to stream processing systems. SPRAWL achieves this through unified API interfaces. In this chapter, we use two cases to show how these interfaces are used, one for distributed stream processing engines, and the other for single-node engines.

A.1 ZStream

SPRAWL uses ZStream [44] as its default stream processing system. ZStream is a distributed stream processing engine that allows stream processing operators to be connected and compiled as an executable DAG. It supports basic stream processing operators such as, *filter*, *windowed join*, and *windowed aggregation* as well as more complicated operators like *sequence*, *conjunction*, *kleene closure*, and *negation* for pattern detection.

Figure A-1 is an example operator placement for Query 1: the left *Avg* sub-plan is placed on $Cluster_0.Node_0$, the right sub-plan is on $Cluster_1.Node_1$, and *COM* is on $Cluster_2.Node_2$. We show example API calls generated by each individual module below.

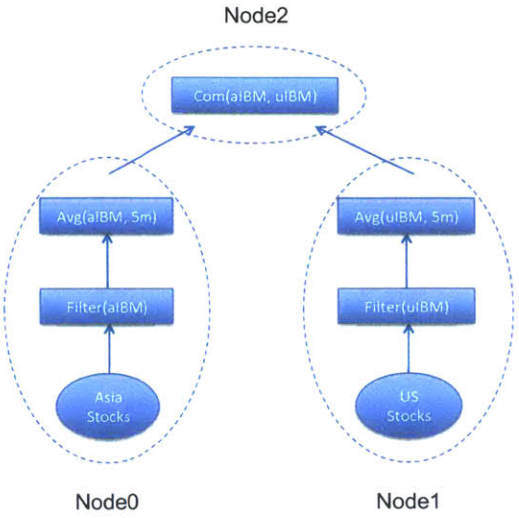


Figure A-1: An Example Placement for Query 1

A.1.1 Example API Calls for $Node_0$

```
> plan = createplan.init(0)
> plan.addop('0', 'src', 'Asia Stocks', null, '0')
> plan.addop('1', 'filter', 'name:=:aIBM', '0', '1')
> plan.addop('2', 'agg', 'avg:price:5m', '1', '2')
> plan.addop('3', 'output', null, '2', 'ip2:port0')
> plan.start()
> ......
> plan.end()
```

- `createplan.init(nodeid)` :

  `Createplan` creates and initializes a new plan on a physical network node, given *nodeid*.

- `addop(opid, optype, oppara, opinput, opoutput)` :

  `Addop` attaches an operator to the plan. *Opid* indicates the internal operator id, *optype* indicates the operator type, *oppara* is a list of operator parameters, *opinput* is a list of input data edge ids, and *opoutput* is a list of output data

edge ids.

- `start(time)` :

  It starts the plan with the running period specified as *time*. If no *time* is specified, the plan will keep running forever.

- `end()` :

  It terminates a plan execution.

All parameters of the APIs are strings. A query plan(sub-plan) is created and initialized with a physical network *nodeid*, and constructed by adding operators one by one. If two operators are connected, the downstream operator's input id is set to the upstream's output id. For example, the *Avg* operator's input id is equal to the *filter* operator's output id on $Node_0$. Finally, the *output* operator delivers results from $Node_0$ to $Node_2(ip2 : port0)$. A plan (continuous query) keeps running forever if no time constraint is specified. It can be terminated explicitly by an `end()` function.

A.1.2 Example API Calls for $Node_1$

```
> plan = createplan.init(1)
> plan.addop('0', 'src', 'US Stocks', null, '0')
> plan.addop('1', 'filter', 'name:=:uIBM', '0', '1')
> plan.addop('2', 'agg', 'avg:price:5m', '1', '2')
> plan.addop('3', 'output', null, '2', 'ip2:port1')
> plan.start()
> ......
> plan.end()
```

The API calls for $Node_1$ are similar to that for $Node_0$, since the right sub-plan is symmetric to the left.

A.1.3 Example API Calls for $Node_2$

```
> plan = createplan.init(2)
```

```
> plan.addop('0','input', '0:3', 'port0', '0')
> plan.addop('1','input', '1:3', 'port1', '1')
> plan.addop('2', 'compare', null, '0:1', '2' )
> plan.start()
> ......
> plan.end()
```

On $Node_2$, two *input* operators accept data streams coming from $Node_0$ and $Node_1$ respectively. The parameters of the *input* operator specifies where the input stream comes from ('Nodeid:operatorid'). The *compare* operator has two inputs ('0:1'), with one connected to the first *input* with outputid 0, and the other to the second *input* with outputid 1.

A.2 Wavescope

In this section, we show how SPRAWL interfaces work with a single-node stream processing system Wavescope [28] in this section. WaveScope is a system for developing high-rate stream processing applications using a combination of signal processing and database (event processing) operations developed in *MIT CSAIL Database Group*. Example API calls are shown below. We use the same query placement example as in Figure A-1.

A.2.1 Example API Calls for $Node_0$

```
> plan = createplan.init(0)
> plan.addop('0', 'src', 'Asia Stocks', null, '0')
> plan.addop('1', 'filter', 'name:=:aIBM', '0', '1')
> plan.addop('2', 'agg', 'avg:price:5m', '1', '2')

> datasocketsend = createsenddatasocket.init(plan.id)
> datasocketsend.bind(ip2, port0)
```

```
> datasocketsend.start()
> plan.start()
```

The API calls are similar to that of ZStream, except output of sub-plans are redirected to a socket provided by the SPRAWL individual module on $Node_0$.

- `createsenddatasocket.init(planid)` :
  to create and initialize a data socket to transmit output of a plan with *planid.*
- `bind(ip, port)` :
  to bind the socket to the given *ip* and *port*
- `start()` : to start sending data

A.2.2 Example API Calls for $Node_1$

Similarly, API calls for $Node_1$ are as follows:

```
> plan = createplan.init(1)
> plan.addop('0', 'src', 'US Stocks', null, '0')
> plan.addop('1', 'filter', 'name:=:uIBM', '0', '1')
> plan.addop('2', 'agg', 'avg:price:5m', '1', '2')

> datasocketsend = createsenddatasocket.init(plan.id)
> datasocketsend.bind(ip2, port1)
> datasocketsend.start()
> plan.start()
```

A.2.3 Example API Calls for $Node_2$

Correspondingly, the *COM* operator reads input data from sockets provided by the individual module on $Node_2$.

```
> datasocketrecv0 = createrecvdatasocket.init()
```

```
> datasocketrecv0.listen(port0)
> datasocketrecv0.start()

> datasocketrecv1 = createrecvdatasocket.init()
> datasocketrecv1.listen(port1)
> datasocketrecv1.start()

> plan = createplan.init(2)
> plan.addop('0','src', 'datasocketrecv0.id', 'port0', '0')
> plan.addop('1','src', 'datasocketrecv1.id', 'port1', '1')
> plan.addop('2', 'compare:price:left.name=right.name', '0:1', '2' )
> plan.start()
```

- `createrecvdatasocket.init(planid)` : to create and initialize a data socket to receive data.

Bibliography

[1] Akamai. http://www.akamai.com/.

[2] Amazon elastic compute cloud. http://aws.amazon.com/ec2/.

[3] Bloomberg. http://www.bloomberg.com/.

[4] Facebook. https://www.facebook.com/.

[5] Ilog cplex. https://www.software.ibm.com/.

[6] sqlstream. http://www.sqlstream.com/.

[7] Thomson reuters. http://thomsonreuters.com/.

[8] Twitter. https://twitter.com/.

[9] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal(The International Journal on Very Large Data Bases)*, 12(2):120–139, August 2003.

[10] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *VLDB '04 Proceedings of the 30th international conference on Very Large Data Bases*, volume 30, pages 456–467, Toronto, Canada, August 2004.

[11] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing - Parallel data-intensive algorithms and applications*, 28(5):749–771, May 2002.

[12] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *ACM SIGMOD Record(ACM Special Interest Group on Management of Data)*, 30(3):109–120, September 2001.

[13] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In

Proceedings of the 2005 ACM SIGMOD international conference on Management of data (SIGMOD '05), pages 13–24, Baltimore, MD, USA, June 2005.

[14] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and Jr. James B. Rothnie. Query processing in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems (TODS)*, 6(4):602–625, Dec 1981.

[15] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87 Proceedings of the 11th ACM Symposium on Operating systems principles*, pages 123–138, Austin, Texas, USA, November 1987.

[16] Boris Jan Bonfils and Philippe Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *IPSN '03 Proceedings of the 2nd international conference on Information processing in sensor networks*, pages 47–62, Palo Alto, California, USA, April 2003.

[17] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD '03 Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, San Diego, California, USA, June 2003.

[18] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. In *SIGMOD '00 Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 379–390, Dallas, Texas, USA, May 2000.

[19] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data placement in bubba. In *SIGMOD '88 Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 99–108, Chicago, Illinois, USA, June 1988.

[20] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of SIGCOMM*, 2004.

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM—50th anniversary issue: 1958–2008*, 51(1):107–113, January 2008.

[22] Amol Deshpande, Suman Nath, Phillip B. Gibbons, and Srinivasan Seshan. Cache-and-query for wide area sensor databases. In *SIGMOD '03 Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, San Diego, California, USA, June 2003.

[23] Yanlei Diao, Shariq Rizvi, and Michael J. Franklin. Towards an internet-scale xml dissemination service. In *VLDB '04 Proceedings of the 30th international conference on Very large data bases—Volume 30*, pages 612–623, Toronto, Canada, August 2004.

[24] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD '01 Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 115–126, Santa Barbara, California, USA, June 2001.

[25] Sheldon Finkelstein. Common expression analysis in database applications. In *SIGMOD '82 Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 235–245, Orlando, Florida, USA, June 1982.

[26] Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In *SIGMOD '96 Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 149–160, Montreal, Quebec, Canada, June 1996.

[27] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *SIGMOD '08. Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1123–1134, Vancouver, Canada, June 2008.

[28] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. Xstream: a signal-oriented data stream management system. In *ICDE '08. Proceedings of the 24nd international conference on Data Engineering, 2008*, pages 1180–1189, Cancun, Mexico, April 2008.

[29] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *VLDB '97 Proceedings of the 23rd international conference on Very Large Data Bases*, pages 276–285, Athens, Greece, August 1997.

[30] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Query processing of multi-way stream window joins. *The VLDB Journal(The International Journal on Very Large Data Bases)*, 17(3):469–488, May 2008.

[31] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB '03 Proceedings of the 29th international conference on Very Large Data Bases*, pages 297–308, Berlin, Germany, September 2003.

[32] Bruce Hendrickson and Robert Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 28–41, San Diego, California, USA, December 1995.

[33] Toshihide Ibaraki and Naoki Katoh. *Resource allocation problems: algorithmic approaches.* MIT Press, Cambridge, Massachusetts, USA, 1988.

[34] IBM. "ilog cplex", 2010.
http://www.ibm.com.

[35] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and Jr. James W. O'Toole. Overcast: Reliable multicasting with an overlay network. In *OSDI'00 Proceedings of the 4th conference on Symposium on operating system design & implementation - Volume 4*, pages 14–29, San Diego, California, USA, October 2000.

[36] Wen-Ben Jone and Christos A. Papachristou. A coordinated circuit partitioning and test generation method for pseudo-exhaustive testing of vlsi circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(3):374–384, March 1995.

[37] Evangelia Kalyvianaki, Wolfram Wiesemann, Quang Hieu Vu, Daniel Kuhn, and Peter Pietzuch. Sqpr: Stream query planning with reuse. In *ICDE '11. Proceedings of the 27nd international conference on Data Engineering, 2011*, pages 840–851, Hannover, Germany, April 2011.

[38] Shailendra Kumar, Sajal K. Das, and Rupak Biswas. Graph partitioning for parallel applications in heterogeneous grid environments. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pages 167–173, Fort Lauderdale, Florida, USA, April 2002.

[39] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Resource-aware distributed stream management using dynamic overlays. In *Proceedings of 25th IEEE international conference on Distributed Computing Systems, 2005. ICDCS 2005*, pages 783–792, Columbus, Ohio, USA, June 2005.

[40] Per–Åke Larson and H. Z. Yang. Computing queries from derived relations. In *VLDB'85, Proceedings of 11th international conference on Very Large Data Bases*, pages 259–269, Stockholm, Sweden, August 1985.

[41] Jian Li, Amol Deshpande, and Samir Khuller. Minimizing communication cost in distributed multi-query processing. In *ICDE '09. Proceedings of the 25nd international conference on Data Engineering, 2009*, pages 772–783, Shanghai, China, March 2009.

[42] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review(OSDI '02: Proceedings of the 5th Symposium on Operating systems design and implementation)*, 36(SI):131–146, December 2002.

[43] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD '02 Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 49–60, Madison, Wisconsin, USA, June 2002.

[44] Yuan Mei and Samuel Madden. Zstream: A cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD international conference on Management of data (SIGMOD '09)*, pages 193–206, Providence, Rhode Island, USA, June 2009.

[45] Benjamin Nguyen, Serge Abiteboul, Grégory Cobena, and Mihaí Preda. Monitoring xml data on the web. In *SIGMOD '01 Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 437–448, Santa Barbara, California, USA, June 2001.

[46] Olga Papaemmanouil, Sujoy Basu, and Sujata Banerjee. Adaptive in-network query deployment for shared stream processing environments. In *ICDEW 2008. IEEE 24th international conference on Data Engineering Workshop*, pages 206–211, Cancun, Mexico, April 2008.

[47] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE '06. Proceedings of the 22nd international conference on Data Engineering, 2006*, pages 49–60, Atlanta, GA, USA, April 2006.

[48] Daniel J. Rosenkrantz and Harry B. Hunt III. Processing conjunctive predicates and queries. In *VLDB '80 Proceedings of the 6th international conference on Very Large Data Bases*, volume 6, pages 64–72, Montreal, Quebec, Canada, October 1980.

[49] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79 Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, Boston, Massachusetts, USA, May 1979.

[50] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)*, 13(1):23–52, March 1988.

[51] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE '03. Proceedings of the 19th international conference on Data Engineering, 2003*, pages 25–36, Bangalore, India, March 2003.

[52] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01 Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, San Diego, California, USA, August 2001.

[53] Michael Stonebraker. *The Design and Implementation of Distributed INGRES*, pages 187–196. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, USA, 1986.

[54] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *VLDB '07 Proceedings of the 33rd international conference on Very Large Data Bases*, pages 159–170, Vienna, Austria, September 2007.

[55] Joe F. Thompson, Bharat K. Soni, and Nigel P. Weatherill. CRC Press, Boca Raton, Florida, 1999.

[56] Tolga Urhan and Michael J. Franklin. Xjoin: Getting fast answers from slow and bursty networks. *Technical report CS-TR-3994(Feb.), University of Maryland, College Park; UMIACS-TR-99-13*, February 1999.

[57] R. Williams, D. Daniels, L. M. Haas, G. Lapis, B. G. Lindsay, P. Ng, R. Obermarck, P. G. Selinger, A. Walker, P. F. Wilms, and R. A. Yost. R*: An overview of the architecture. In *JCDKB 1982 Proceedings of the 2nd international conference on Databases*, pages 1–27, Jerusalem, Israel, June 1982. Reprinted in: M. StoneBraker(ed.), *Readings in Database Systems*, Morgan Kaufmann Publishers, 1994, pages, 515–536.

[58] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. Soda: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware '08. Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, pages 306–325, Leuven, Belgium, December 2008.

[59] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)*, 22(2):255–314, June 1997.

[60] Praveen Yalagandula, Puneet Sharma, Sujata Banerjee, Sujoy Basu, and Sung-Ju Lee. s^3: a scalable sensing service for monitoring large networked systems. In *INM '06 Proceedings of the 2006 SIGCOMM workshop on Internet network management*, pages 71–76, Pisa, Italy, September 2006.

[61] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to model an internetwork. In *INFOCOM'96 Proceedings of the 15th annual joint conference of the IEEE computer and communications societies conference on The conference on computer communications*, volume 2, pages 594–602, San Francisco, California, USA, March 1996.

[62] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.