Budapest University of Technoloy and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Optimization of Incremental Queries in the Cloud

Scientific Student's Association Report

Author:

József Makai

Advisors:

Gábor Szárnyas
dr. István Ráth
dr. Ákos Horváth

2014

Contents

Kivonat

Az adatbázis- illetve modell-lekérdezések alapvető szerepet játszanak az adatvezérelt alkalmazásokban. Hatékony kiértékelésük kulcsa a végrehajtás optimalizációja. A modellvezérelt szoftvertervezés (model-driven engineering, MDE) folyamatai nagymértékben támaszkodnak a modellek hatékony lekérdezésére. Azonban a modellezendő rendszerek komplexitásának növekedése a modellek több 10-100 milliós méretűre növekedését okozta, így a modell-lekérdezések és transzformációk komplexitása gyakran jelentős skálázhatósági kihívást jelent a jelenlegi MDE eszközök számára.

A skálázhatósági problémára megoldást nyújtanak az elosztott, felhőalapú modell-lekérdező rendszerek. Ilyen rendszer a Hibatűrő Rendszerek Kutatócsoport által fejlesztett IncQuery-D inkrementális gráf lekérdező keretrendszer, melynek célja, hogy biztosítsa a nagy modellek feletti közel lineárisan skálázódó lekérdezéseket. Az elosztott környezetben történő lekérdezés optimalizáció a klasszikus egygépes adatbázis kezelő és modell-lekérdező rendszerekhez képest új aspektusokat és kihívásokat rejt az egyes erőforrások korlátai, az adatok hálózaton történő átvitele és az elosztott rendszer költségei miatt.

Dolgozatom célja, hogy bemutassak és az IncQuery-D elosztott modell és gráf lekérdező rendszerhez kidolgozzak olyan optimalizációs módszereket, amelyek a modellek mérete és a lekérdezések jellege alapján heurisztikák segítségével képesek allokálni a rendszer számítási csomópontjait a rendelkezésre álló erőforrásokra úgy, hogy az több különböző szempont szerint optimalizált végrehajtást eredményezzen.

Az optimalizáció rendszer teljesítményére gyakorolt hatását mérési eredményekkel kívánom igazolni, melyek áttekintésében segítséget nyújt az általam készített monitorozó rendszer. A dolgozat további fontos eredménye, hogy mind a monitorozó, mind az optimalizáló módszerek az IncQuery-D rendszerhez tartozó fejlesztői környezetbe teljes mértékben integráltak.

Abstract

Database and model queries are the foundations of data-driven applications and query optimization is essential for their efficient evaluation. Fast model queries are of primary importance in the workflows of model-driven software engineering (MDE). As modelled systems are significantly increasing in complexity, big software models have already reached and in many cases exceeded 10-100 million model elements in size, therefore traditional MDE tools often encounter scalability issues because of the complexity of queries and transformations. These issues limit the efficiency of the development process.

Distributed, cloud-based model query engines aim to solve the scalability issues. IncQuery-D is an incremental graph query framework which aims to provide scalable queries over large graph models and is developed by the Fault Tolerant Systems Research Group. However these systems require query optimization in order to evaluate queries efficiently. Query optimization in distributed environments brings new aspects and challenges – including the capacity limit of resources, network communication between computers and the cost of the infrastructure – compared to the traditional single workstation model and database query engines.

In this report, we aim to address the challenges and provide a implementation in the IncQuery-D distributed model and graph query engine. These methods take account of the model size and consider the nature of the query to use heuristics. These heuristics are important in the allocation process of the computation nodes to the available resources in order to provide optimized query evaluation by different aspects.

We wish to justify the impact of the optimization techniques on the query engine performance by measurements. These results can be viewed by the IncQuery-D monitoring system that we created for the query engine. Further achievement is that the monitoring system and the optimization facilities are fully integrated to the IncQuery-D's own development environment.

Chapter 1

Introduction

1.1 Context

Model-driven software engineering (MDE) plays an important role in the development processes of critical embedded systems. Advanced modeling tools provide support for a wide range of development tasks such as requirements and traceability management, system modeling, early design validation, automated code generation, model-based testing and other validation and verification tasks.

Scalability issues in MDE However, models representing sensor data, reverse engineered software models (e.g. abstract syntax trees of existing source code) and geospatial models can contain well over 10^9 modeling elements [34]. Furthermore design and system models (e.g. AUTOSAR [4]) are also exceeding the more millions elements in size. Modeling toolchains are facing scalability challenges and these issues limit the efficiency of the development process.

IncQuery-D: Incremental query evaluation in the cloud The scalability problems originate from the facts that design models constantly increase in size and the complexity of the adherent MDE tools has also evolved. The cloud infrastructure and big data databases ("scaling out") offers scalability solutions for the storage of big models. On the other hand these technologies require an extending solution which supports the incremental evaluation of complex queries with typical MDE workloads. To solve this challenge, IncQuery-D [36] is an incremental graph and model query framework which aims to provide scalable queries over large graph models. The contributions of this report are extensions of earlier results, focused on the allocation optimization of the distributed system in order to provide better resource utilization and runtime performance.

1.2 Problem Statement and Requirements

In the context of distributed systems, resource allocation is a persistent problem. Allocation is basically a sizing problem of the distributed system, which aims to provide resources for the different components in a feasible way. In many practical cases, it is difficult to predict the amount of resources required for the system to work and to allocate the components of the system the way to provide scalability. In today's cloud computing environments where computing resources are publically available, one may also wish to consider other factors (such as cost) for the optimization. The problems discussed in this report are related to the sizing and allocation problems of the IncQuery-D distributed query engine.

Allocation problems in IncQuery-D The IncQuery-D distributed incremental query system also holds several difficult allocation problems. The distributed computation nodes and processes of the system run memory intensive tasks and generate high volume network traffic as they exchange data. Therefore we have to separate these tasks efficiently in order to avoid the overuse of the available resources, but we also have to ensure reasonable utilization to reduce the costs of the computation infrastructure.

Furthermore, the allocation can have serious impact on the runtime performance of the system. We have to avoid the thrashing of the query evaluation due to memory exhaustion, and minimize the amount of expensive network communication in order to maximize the runtime performance of the system and reduce query processing time.

Our proposed solution In order to address these challenges, we propose a complex, multi-dimensional optimization approach. Our proposal should provide stable and reliable allocation variants which avoid the overuse of the available resources. It also supports optimization according to different aspects, like infrastructure cost and network communication. Finally, we strive to ensure that the optimization is automated as much as possible, in order to reduce the amount of necessary human input to a minimum.

1.3 Objectives and Contributions

Heuristics-driven Combinatorial Optimization Methods We aim to provide combinatorial optimization based solutions for the allocation problems, which use heuristics-driven methods for the estimation of process memory usage and network communication. The provided algorithms can find solutions were the different optimization objective functions (overall network communication, overall computation infrastructure cost) have optimum, while the resources (especially the memory) of the computers are not exhausted. Therefore we solve multi-dimensional combinatorial optimization problems to take both the network communication and overall cost into account in order to provide good runtime performance for the system while the cost is kept as low as possible.

Telemetry Data for Resource Usage Estimation We recognized the importance of system runtime observability in regards of the allocation optimization, since the heuristics are based on system telemetry data such as memory usage and network connection parameters. Therefore the creation of a runtime monitoring subsystem is a precondition for the allocation optimization problems.

Contributions

- In order to reach the proposed objectives, the first task was the design and implementation of a *runtime monitoring subsystem* for IncQuery-D, which collects all the relevant performance data from the system.

- We also created a web-based *monitoring dashboard* in order to visualize the system components and their relevant metrics. With this facility, the engineer can precisely observe the behaviour of the system in real time.

- As the optimization problems require estimations for the memory consumption of processes and their network communication, we elaborated *heuristics-based methods* for precise predictions.

- In order to be able to create algorithms for the allocation omptimization problems, we created *mathematical formalizations* for the problems and analyzed them in terms of *computational complexity.* Furthermore we provide formal proofs regarding their complexity class, in order to justify the choice of constraint satisfaction-based [39] methods (instead of dedicated algorithms). We also implemented these *CSP-based algorithms.*

- As a further contribution, the *runtime monitoring* and *allocation optimizer* subsystems were *fully integrated* to the IncQuery-D framework and to its integrated development environment.

- Finally, we tested the impacts of the solutions on the system performance with *benchmark measurements.*

1.4 Structure of the Report

The report is structured as follows. Chapter 2 introduces the background technologies for this report's contributions and examines the related work. Chapter 3 provides an overview of the problems and the motivation for allocation optimization. Furthermore it introduces the new components of the IncQuery-D framework related to the allocation. Chapter 4 provides the formalization of the allocation optimization problems and proves their computational complexity. Chapter 5 proposes the CSP-based solutions for the optimiaztion problems. This chapter also brings case studies for the allocation optimization problems and their solutions. Furthermore this chapter will introduce the integrated allocation optimizer and runtime monitoring subsystem in regards of the case studies. Chapter 6 presents measurement results about the effect of allocation optimization. Chapter 7 concludes the report and presents our future plans.

Chapter 2

Background and Related Work

The purpose of this section is to introduce the background technologies and works of this report's contributions. Finally, we take a look at the related work.

2.1 A Motivating Case Study, the Train Benchmark

The Train Benchmark [30] was designed at the Fault Tolerant Systems Research Group to measure the efficiency of model queries and manipulation operations in different tools. The Train Benchmark is primarily targeted for typical MDE workloads, more specifically for well-formedness validations.

In a typical MDE model validation workload, the user validates the model which contains some errors. According to the results, the user tries to fix some (but typically not many) of the errors by the modification of model elements and revalidates the model. The revalidation determines the impact of the change and provides feedback to the user. The Train Benchmark programatically simulates this typical workload with relatively small changes.

The Train Benchmark is built around an imaginary railroad system. The system's network is composed of typical railroad items, including signals, segments, switches and sensors. The complete metamodel is shown in Figure 2.1. A subgraph of an instance model is shown in Figure 2.2.

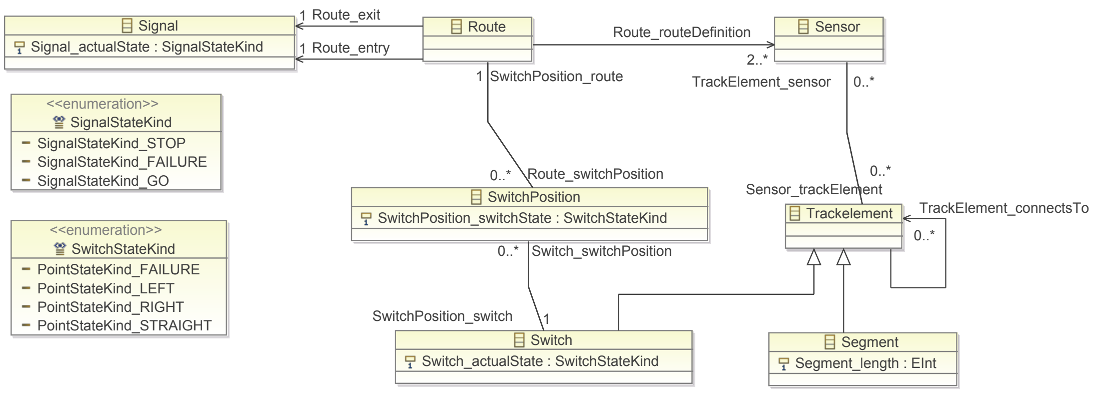


Figure 2.1. Metamodel of Train Benchmark.

The benchmark defines four distinct phases, also shown in Figure 2.3.

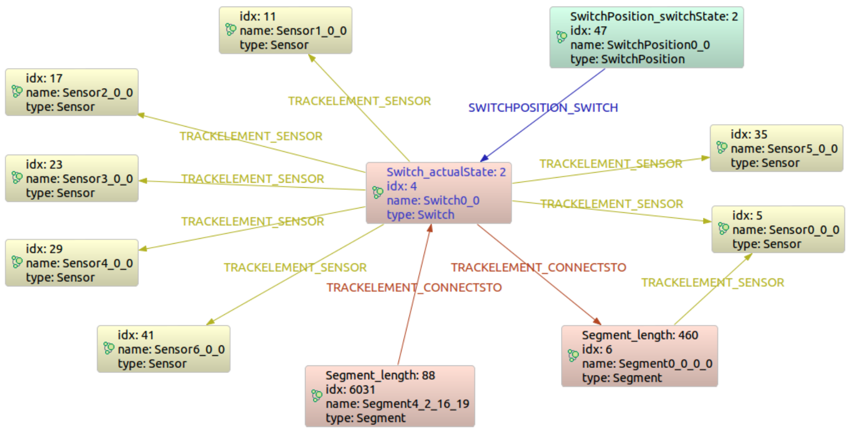


Figure 2.2. A subgraph of a railroad system visualized.

1. *Load:* load the serialized instance model to the database.
2. *First validation:* execute the well-formedness query on the model.
3. *Transformation:* modify the model.
4. *Revalidation:* execute the well-formedness query again.

RouteSensor

Description. The *RouteSensor* well-formedness constraint requires that all sensors that are associated with a switch that belongs to a route must also be associated directly with the same route. Therefore, the query (Figure 2.4) looks for sensors that are connected to a switch, but the sensor and the switch are not connected to the same route.

Goal. This pattern checks for the absence of circles, so the efficiency of the join and the antijoin operations is tested.

Transformation. Randomly selected invalid *sensors* are disconnected from the *switch*, which means that the constraint will no longer apply (Figure 2.6).

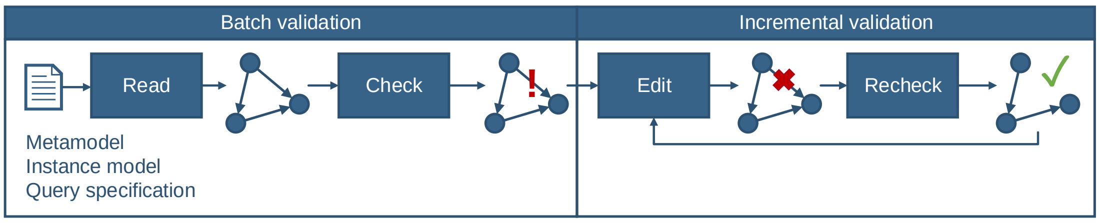


Figure 2.3. Workflow in Train Benchmark.

SwitchSensor

Description. The *SwitchSensor* well-formedness constraint requires that every switch must have at least one sensor connected to it. Therefore, the query (Figure 2.5) checks for switches that have no sensors associated with them.

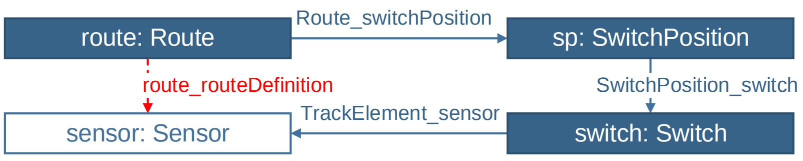


Figure 2.4. Graph pattern of the RouteSensor Query.

Goal. This query checks whether an object is connected to a relation. This pattern is common in more complex queries, e.g. it is used in the *RouteSensor* queries.

Transformation. Random elements are selected from the set of invalid *switches* and are connected to newly created *sensors* (Figure 2.7).

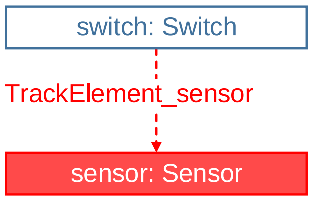


Figure 2.5. Graph pattern of the SwitchSensor Query.

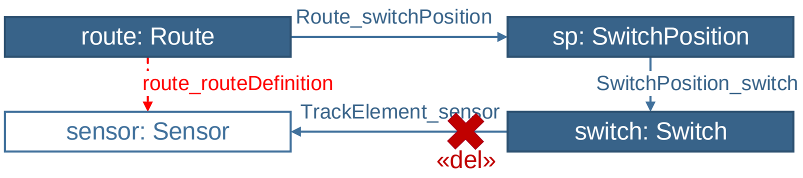


Figure 2.6. Modification in the RouteSensor Query.

2.2 4store

4store [1] [27] is an open-source, distributed triplestore [21] written in C. 4store is primarily applied for semantic technology projects.

Architecture 4store was designed to work in a cluster with high-speed networks. 4store server instances are capable of discovering each other using the Avahi configuration protocol [5]. 4store offers a command-line and an HTTP server interface.

Data Model 4store's data model is an RDF [18] graph. It supports the RDF/XML [17] input format.

Sharding 4store distributes the RDF resources evenly across the cluster. 4store also supports replication by mirroring tuples across the cluster.

Query Language and Evaluation 4store uses the Rasqal RDF Query Library [16] to supports SPARQL [20] queries.

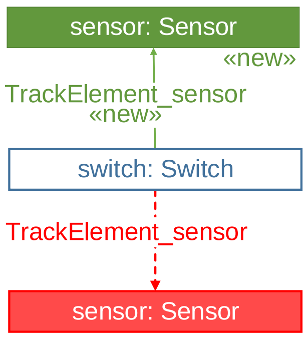


Figure 2.7. Modification in the SwitchSensor Query.

2.3 The Rete Algorithm

In the following, we provide an overview of the *Rete algorithm* [26]. The algorithm has primary importance in the context of this report as IncQuery-D is built on the foundations of this algorithm.

Numerous algorithms were invented for the purpose of incremental pattern matching. Mostly, these algorithms originate from the field of rule-based expert systems. One of the most well-known is the *Rete algorithm*, which creates a propagation network. The network stores the partial matches found in the graph.

2.3.1 Overview of the Rete Algorithm

The basis of the algorithm is an asynchronous network of communicating nodes, which is shown in Figure 2.8. This is essentially a dataflow network. First, the network computes the set of pattern matches in the graph. The main feature of the algorithm is that it is capable of incrementally maintaining the match set by propagating *update messages*. In order to do that, each network node stores the computed partial results in a cache and will maintain this cache according to the changes. Creating new graph elements (vertices or edges) results in *positive update messages*, while removing graph elements results in *negative update messages*.

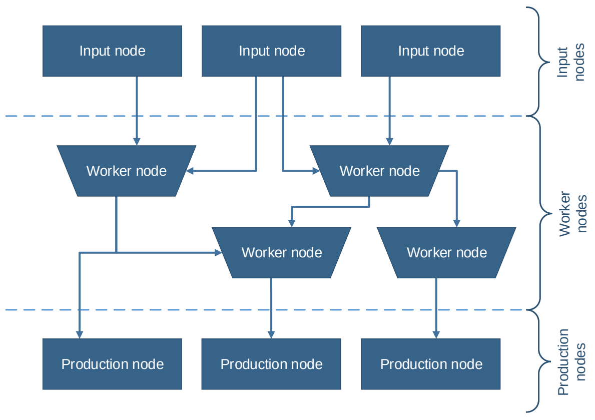


Figure 2.8. The structure of the Rete propagation network.

There are three types of nodes in the network:

- *Input nodes* are responsible for indexing the model by type, they store the different edge and vertex types of the graph. They are also responsible for creating and propagating the *update messages* to the *worker nodes.*
- *Worker nodes* perform operations on the output of their parent node(s) and propagate the results. The worker nodes store partial query results in their own memory.
- *Production nodes* are terminators that provide an interface for fetching the query results.

2.3.2 Data Representation in the Rete Algorithm

Definition 1 (Tuple). A *tuple* is an ordered list of elements. It has arity, which is the number of elements (attributes) in the *tuple.* The items in a *tuple* are referenced by their index. The first element has an index of 0. Tuples can be $\langle 16, 34, 8\rangle$ or $\langle 1, '\text{hello}', \text{true}\rangle$. ▪

In order to use tuples for graph pattern matching, the vertices and edges in the graph have to be mapped to tuples. We presume that each vertex in the graph has a unique identifier.

Mapping Vertices to Tuples The following solution can be used to represent the vertices with *tuples.* We create a *tuple* for each vertex with its identifier and its attributes $\langle id, attribute_1, attribute_2, \ldots\rangle$. We will store the *tuples* of each vertex type in different *relations.* For example to store *Route* 2.1 typed vertices we can use the *relation Route* $= \{\langle 1\rangle, \langle 3\rangle, \langle 4\rangle, \ldots\}$

Mapping Edges to Tuples Edges are mapped in a straightforward way: each (directed) edge is represented by a $\langle$*source vertex id*, *target vertex id*$\rangle$ *tuple.*

2.3.3 Worker Nodes of the Rete Algorithm

In the following we present some of the most important types of *worker nodes* and briefly describe their operations. Their operations come from *relational algebra*, which can be found in any computer science textbook related to relational databases [38]. However, we do not discuss their *relational algebraic* form in detail here.

Alpha Nodes

Alpha nodes have one input slot. They usually filter the content of the parent node according to some criteria. The incoming relation is denoted by r and the outgoing relation is denoted by t. Hereby we present the most important *Alpha nodes* in regards of this report.

Check Node The *Check node* implements the *selection* operation of the *relational algebra.* It filters the incoming *tuples* which do not satisfy a certain condition. This condition is usually one of the mathematic relations $(=, \neq, \geq, \leq, \ldots)$. As an example we can use the *Check node* to express that one attribute of a graph vertex should be equal with 20. This node type does not require memory in order to work as it will only filter and propagate the incremental changes. The notation of the *Check node* is shown in Figure 2.9.

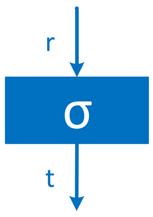


Figure 2.9. Rete Check node.

Trimmer Node The *Trimmer node* implements the *projection* operation of the *relational algebra*. It removes the appropriate attributes of the incoming *tuples*. It also filters duplicate *tuples* because even if the incoming *relation* contained unique *tuples*, the new *relation* with fewer attributes can contain duplicates. The notation of the *Trimmer node* is shown in Figure 2.10.

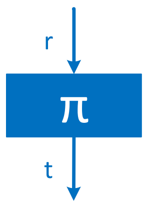


Figure 2.10. Rete Trimmer node.

Beta Nodes

Beta nodes have two input slots, the *primary (p)* and the *secondary (s)*. Beta node implementations typically store the input relations in *indexers*. This will be required for their incremental operations. The relation representing the result *tuples* is denoted with *t*.

Join Node The *Join node* implements the *theta join* operation of the *relational algebra*. It matches the corresponding *tuples* from its parent *relations*. In our context, the most common use of this node is to match the source and target vertices of the edges. This node type requires memory for its incremental operation as the new (or removed) *tuples* of the *changeset* should be matched with all foregoing *tuples* in order to get correct results. The notation of the *Join node* is shown in Figure 2.11.

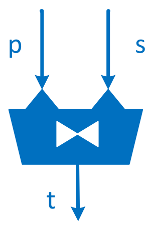


Figure 2.11. Rete Join node.

Antijoin Node The *Antijoin node* looks for *tuples* in its *primary (p) relation* which do not have a matching pair in the *secondary (s) relation*. This operation is useful for finding missing edges and other *negative conditions* in the graph. This node also stores its parent *relations* for the incremental operation capability. The notation of the *Antijoin node* is shown in Figure 2.12.

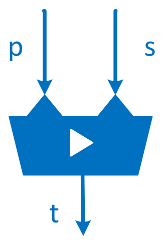


Figure 2.12. Rete Antijoin node.

2.4 IncQuery-D

IncQuery-D [29] [36] is an incremental distributed graph and model query engine. Its aim is to provide scalable query processing for large graphs and models with more 10 - 100 million elements. Here we give a brief introduction of the system.

2.4.1 Architecture

The high-level architecture of IncQuery-D is shown in Figure 2.13.

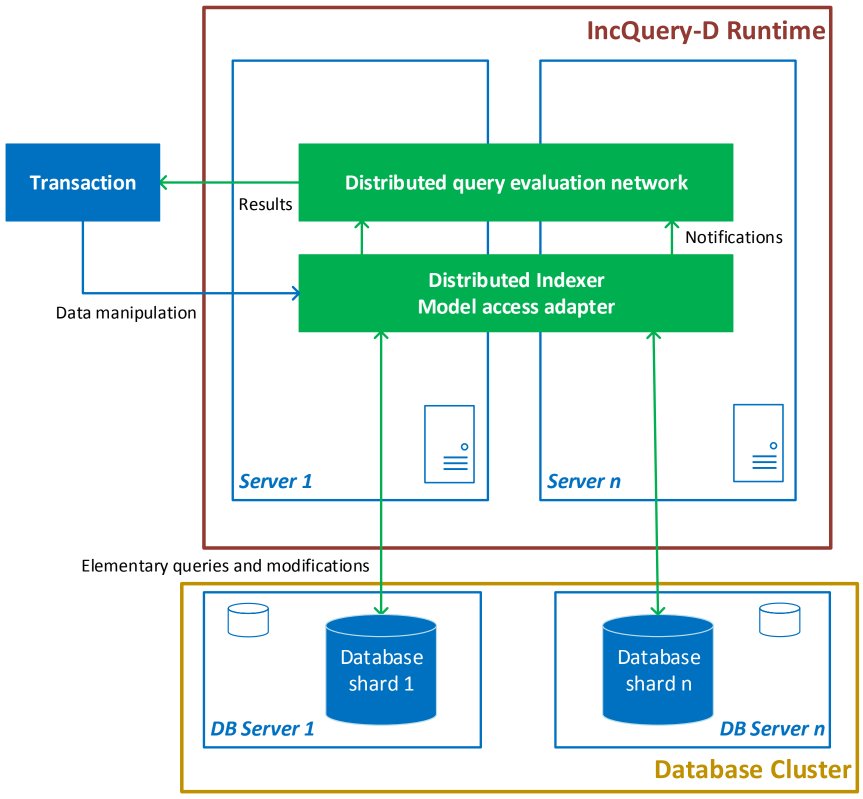


Figure 2.13. Architecture of IncQuery-D.

IncQuery-D's architecture consists of three layers, the *storage layer*, the *distributed model indexer and adapter*, the *distributed query evaluation network* (based on the *Rete propagation network*).

The *storage layer* is a distributed database which is responsible for persisting the model. The primary technology for storing the graph models is the *4store* [1] triplestore.

The *distributed model indexer and adapter layer* is responsible for maintaining type-instance indexes so that all instances of a given type (both edges and graph nodes) can be enumerated quickly. It is also responsible for providing a unique identifier for each model element. Finally, as *model change notifications* are required for incremental query evaluation, this layer is also responsible for accepting *change notifications* and propagating *model changes* to the *query evaluation layer.* This layer achieves this by providing a facade for model manipulation operations.

The *distributed query evaluation network* is responsible for evaluating the query and providing the results incrementally. To achieve this, this layer implements a distributed version of the *Rete propagation network* (Section 2.3). The importance of the layer is the capability of distributing the query evaluation to more processes and computers of a cluster in order to provide scalability.

2.4.2 The Coordinator

It is a general concept for distributed systems to have a *coordinator* process which supervises, coordinates and controls the operations of the system.

In the IncQuery-D system the *coordinator* is responsible for the creation and supervision of the *Rete network* and its *Rete nodes.* It also helps in the coordination of the distributed query evaluation. Finally, the query results can be retrieved with the help of the *coordinator* from the *Production nodes*, once they are calculated.

2.4.3 Remote Message Sending with Akka

In distributed systems it is essential that the distributed processes communicate with each other through the network. In the IncQuery-D system we use the *Akka* messaging framework [2]. We present the most important relations between IncQuery-D and *Akka* as the report will have closely related parts to the remote communication of the system.

The most important concept of *Akka* is the *actor. Actors* can communicate with each other, both locally and remotely through the network. The distributed *Rete nodes* of IncQuery-D will be implemented as (remote) communicating *actors.* The *coordinator* will be an *actor* as well. The *actors* (except the *coordinator*) will run in so-called *Akka microkernels* [3]. Basically, these are containers which run in a separate Java Virtual Machine. The *microkernel* provides a convenient way for deploying and running *actors.* It will be the responsibility of the *coordinator* to create the appropriate *actors* for the *Rete nodes* in the *microkernels.*

2.5 Query Optimization for the Rete Algorithm

Query optimization is important part of every data query engine. Its aim is to provide short query evaluation time as the user expects to see the results immediately. Therefore it tries to predict the fastest possible execution plan for the queries.

2.5.1 Rete Layout Optimization

There are also several optimization opportunities in case of the Rete algorithm. One of them is the layout optimization of the Rete network. Consider the following simple Rete network with a Join and a Check node depicted in Figure 2.14. Suppose that the Check node filters some mutual attributes used by the parent Join node. In that case the filtering by the Check node could happen before the Join operation like it is shown in Figure 2.15. This way the Join node should deal with fewer amount of data tuples causing faster query evaluation.

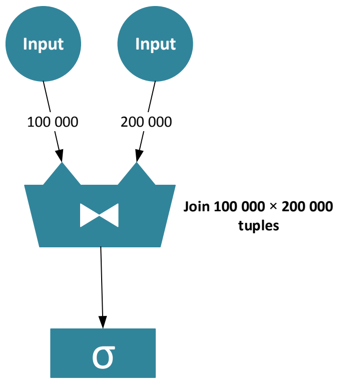


Figure 2.14. Rete layout without Optimization.

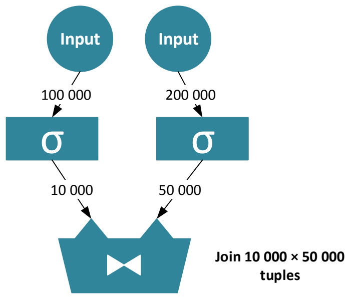


Figure 2.15. Rete layout with Optimization.

Note that this was just a simple example for the Rete layout optimization opportunities. There are sophisticated algorithms [40] created for that problem. However, this report will not discuss this problem as we aim to solve orthogonals problems.

2.5.2 Rete Allocation Optimization

Optimization of the *Rete algorithm* has other aspects as well. If we think about the fact that the Rete network will be distributed to different processes on different computers, we can realize that the allocation problem is not trivial. By allocation we mean the assignment of Rete nodes to processes and the mapping of processes to host computers as it is depicted in Figure 2.16 and 2.17.

One could derive the conclusion that there are a lot of allocation variants available even in case of a small Rete network. These different variants may cause different effects on the performance of the system. So rather than doing the allocation in an ad-hoc way, there is a valid motivation for creating optimized allocation. This allocation problem will be proposed in details in Chapter 3.

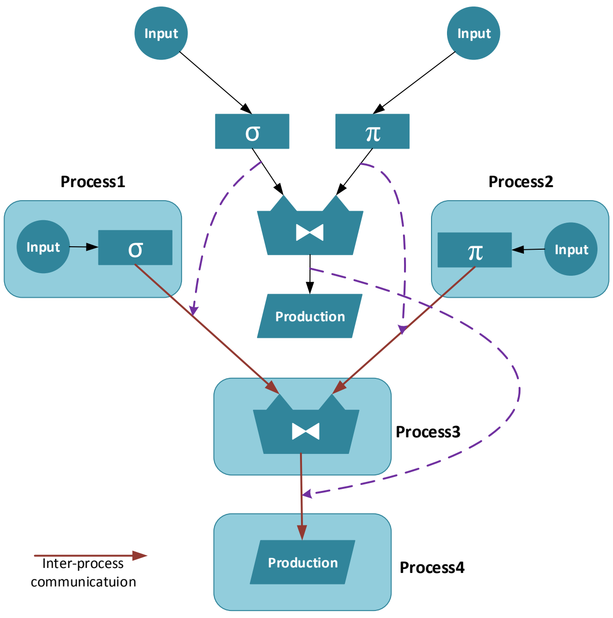


Figure 2.16. Allocation of Rete nodes to processes.

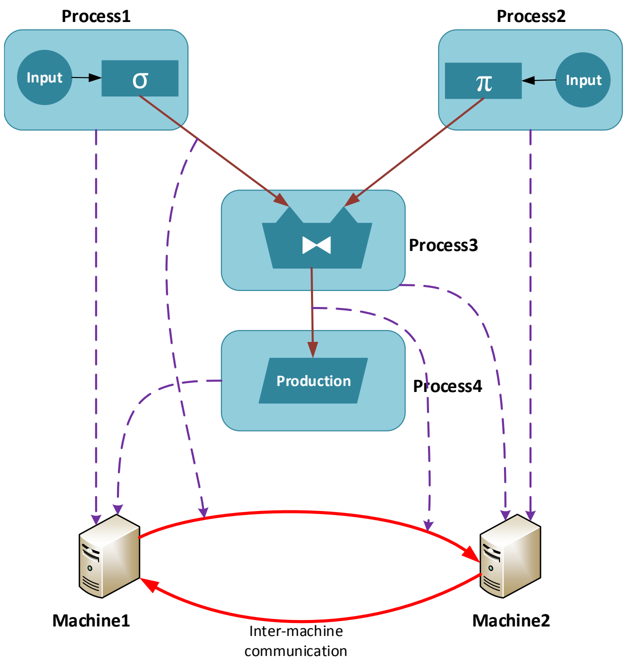


Figure 2.17. Allocation of processes to machines.

2.6 Related Work

The following paragraph with its references is based on the IncQuery-D paper [29].

A wide range of special languages have been developed to support graph based representation and querying of computer data. The Resource Description Framework (RDF) [18] is developed to support the description of instances of the semantic web, assuming sparse, ever-growing and incomplete data. Semantic models are built up from triple statements, which can be queried using the SPARQL [20] graph pattern language with tools like Sesame [19] or Virtuoso [13]. Property graphs provide a more general way to describe graphs by annotating vertices and edges with key-value properties. They can be stored in graph databases like Neo4j [12] which provides the Cypher query language. Even though big data storage (usually based on MapReduce [25]) provides fast object persistence and retrieval, query engines realized directly on these data structures do not provide dedicated support for incremental query evaluation. In the context of event-based systems, distributed evaluation engines were proposed earlier, scaling up in the number of rules rather than in the number of data elements. As a very recent development, Rete-based caching approaches have been proposed for the processing of Linked Data (bearing the closest similarity of our approach). INSTANS [33] uses this algorithm to perform complex event processing (formulated in SPARQL) on RDF data, gathered from distributed sensors. Diamond [32] evaluates SPARQL queries on Linked Data, but it lacks an indexing middleware layer so their main challenge is efficient data traversal.

Up to our best knowledge, none of the aforementioned works had related work in regards of the novel approach of the allocation optimization problem proposed in this report.

Chapter 3

Overview

Query optimization is of key importance for fast query evaluation and all mature query engines must provide solutions for it. This especially applies for query engines that address scalable query evaluation. However, optimization in a distributed environment has several new aspects and challenges. In this chapter, we overview these challenges and also discuss our proposed solution.

3.1 Allocation in Distributed Query Optimization

Distributed systems are built in order to distribute computation tasks and their resource usage between computers of a cluster. It is usually not efficient or not possible to place all computation task to one process on one machine. Therefore the computation tasks have to be assigned to different processes and these processes have to be allocated to different computers. However, the different allocation alternatives cause significant performance difference compared to each other.

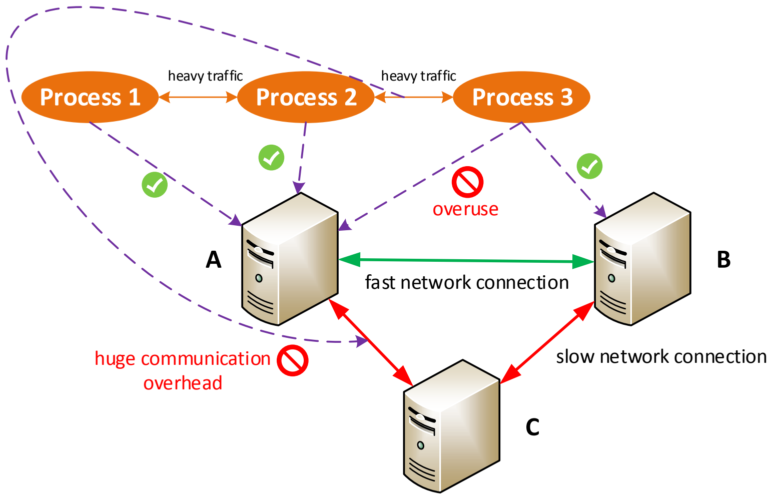


Figure 3.1. Overview of allocation challenges for distributed query engines.

Figure 3.1 depicts the challenges of allocation in the distributed architecture of IncQuery-D. In this particular scenario, we allocate process 1 and process 2 to machine A. It would make sense to allocate process 3 there as well, but this would exhaust the resources of machine A. Therefore, we can allocate it to either machine B or C. On the other hand, we know that the communication link between A and B is faster than the one between A and C, and we expect intense traffic between process 2 and 3. Therefore, we place process 3 to machine B in order to avoid huge communication overhead.

The architecture of IncQuery-D is an asynchronous dataflow network where the nodes of the network, the Rete nodes, run memory- and CPU-intensive computation tasks. The purpose of the allocation optimization is the maximization of the dataflow network's throughput. In order to reach that, the possible bottlenecks of the processing should be avoided by preventing significant resource overuse. On the other hand the expensive remote communication between processes should be reduced as much as possible.

Distributed query engines generate intense network traffic as the distributed computation nodes and processes propagate the partial results of the query. This traffic can sometimes be very high, reaching 10-100 million bytes, depending on the size of the queried database. If we think in the cloud environment then the computers may be in different data centers or sometimes even on different continents. Furthermore, on public networks the available bandwidth can be used by others as well. So if we send vast amount of data on a slow connection link, then we can experience huge overhead in the query processing time.

3.1.1 Allocation Challenges

The problems proposed above are important part of the IncQuery-D allocation process as well. To better understand these problems in the allocation process of IncQuery-D, it will be discussed in details with the following example.

Assume we have the following 3 machines with the given memory capacities to use for the system.

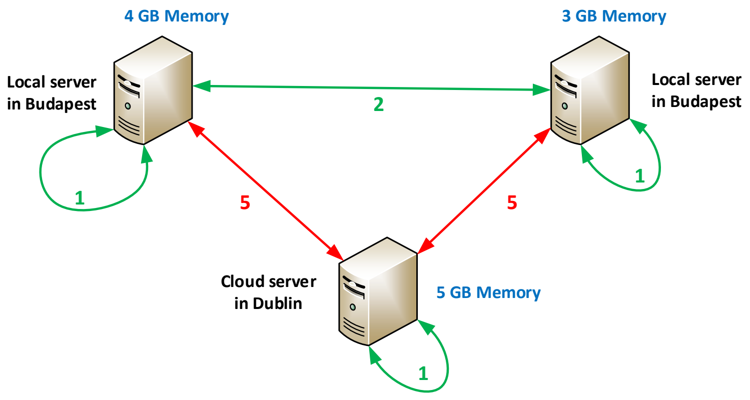


Figure 3.2. Sample infrastructure.

The numbers on the edges describe the communication overhead of the connections between the computers. Those will be taken into account as multipliers for the amount of communication between processes.

Minimizing Network Traffic

In order to justify the rationale of using such an overhead multiplier number, we conducted measurements with the messaging framework used in IncQuery-D, Akka [2]. We had two machines in this measurement. Both machines were in Budapest, one was a local machine and the other was created in a private cloud service. We measured the communication time with increasing amount of tuples between processes on the same host machines and between processes on different remote hosts.

Definition 2 (Process). In the context of IncQuery-D a process means the Java Virtual Machine and all its corresponding utilities as the IncQuery-D system is implemented on top of the Java platform. ▪

Definition 3 (Normalized tuple). Given a set tuples (with the same arity), *normalized tuple* is defined as the product of the tuple arity and the number of tuples. Using normalized tuple assumes that we have the same type of data in the tuples. We use the concept of normalized tuple as a simplifying assumption. ▪

Example 1 (Normalized tuple). *If we have 10000 tuples with two numbers* ($\langle 1, 2 \rangle$) *then normalized tuple will be 20000.*

We have created a plot from the measurement results which are shown in Figure 3.3. Next to the curves, we also put their linear regression approximations. The purpose of this regression curve is to show that the transmission time can be extremely well approximated by the normalized tuples and a constant multiplier as the correlations between the corresponding curves are very close to 1. Therefore the communication overheads between machines can be indeed very well characterized by these numbers.

The importance of these overhead numbers lies in their ratio compared to each other. This way, we can say that the local communication has 5.2 times better performance parameters compared to the remote.

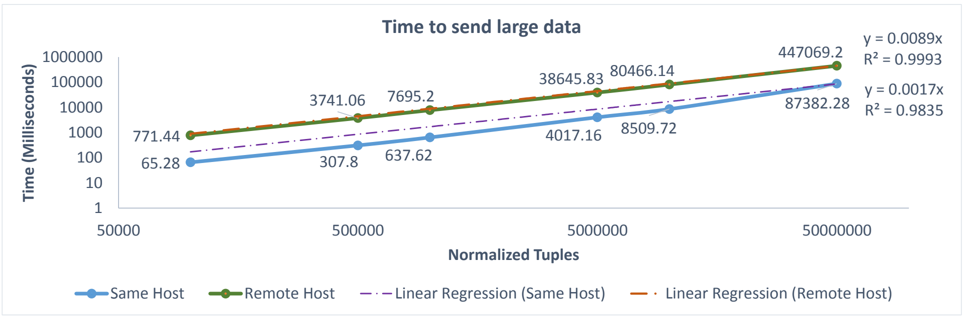


Figure 3.3. Measurements of large volume data traffic.

This measurement also justifies another important fact: the communication between remote machines has significant overhead compared to the communication on the local interface of a machine.

Both axes of the above plot have logarithmic scale. One may observe that there is a shift between the curves. This means that there is approximately one order of magnitude difference (it lowers a bit as the number of normalized tuples increases) between them. It can also be observed that at 5 millions of tuples the difference is already 34 seconds and it increases dramatically after that. So it shows that network traffic minimization is indeed a valid optimization target.

Note that above results contain data serialization time as well.

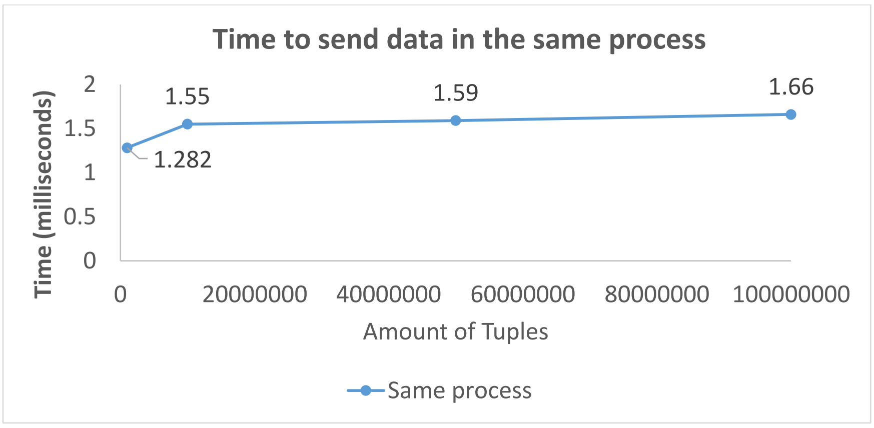


Figure 3.4. Time to send data in the same process.

We also measured the communication time in the same process.

As it is shown in Figure 3.4, it is fast as the communication happens through the memory and the data is not serialized. With this fact in mind we can ignore the overhead of communication within the same process so we can focus on the inter-process communication. This will be an assumption from now on.

Avoiding Local Resource Exhaustion

The Problem The first problem of the allocation is the assignment of computation nodes, the Rete nodes to JVMs. The input of the problem is the constructed Rete network of the query. The purpose of this problem is the determination of the appropriate number of processes the query will run in, and furthermore the allocation of Rete nodes to the processes. The output of the problem will be the processes with their estimated memory consumption and the Rete nodes assigned to them. Furthermore, we will have the inter-process communication edges with their estimated intensity.

JVM Memory Properties As the Rete nodes run memory intensive computation tasks we have to keep some facts in mind about the JVM memory in order to operate the system efficiently. Most of the time it is not recommended to use too much (above 8–16 GB) heap space for one JVM. In case of such large heap space consumption the garbage collector could run quite frequently in order to try free some space. However, when the heap is large then the runtime of the garbage collection will take more time as it has to scan larger object nets. As a conclusion if the heap consumption of a process gets close to its limit then we can experience thrashing. This means that the garbage collector runs in the biggest part of the time and finally the process will be shut down. Rete nodes of IncQuery-D in case of big models can consume 8 GB heap space or more, so allocating more of them to one JVM would case serious performance degradation.

3.1.2 Multi-Dimensional Optimization

So far we discussed the potential of network communication minimization in the context of IncQuery-D allocation optimization. However, the optimization may have other dimensions than the network communication.

For the IncQuery-D allocation optimization process we decided to support the cost of the distributed system as another optimization dimension. The reason behind cost-based optimization is that the system can be used in public cloud services as well, where we have to pay for the used resources. Therefore, the user can decide between the optimization targets but the other dimension will be taken into account as well.

If we have more than one optimization dimension, then the Pareto front [14] of the solutions can be drawn. The Pareto front represents the characteristics of the solution space considering the different optimization dimensions.

A sample Pareto front is shown in Figure 3.5. One can observe that the curve describes a negative correlation between communication and the cost as if we want to improve on one dimension the other has to be worse.

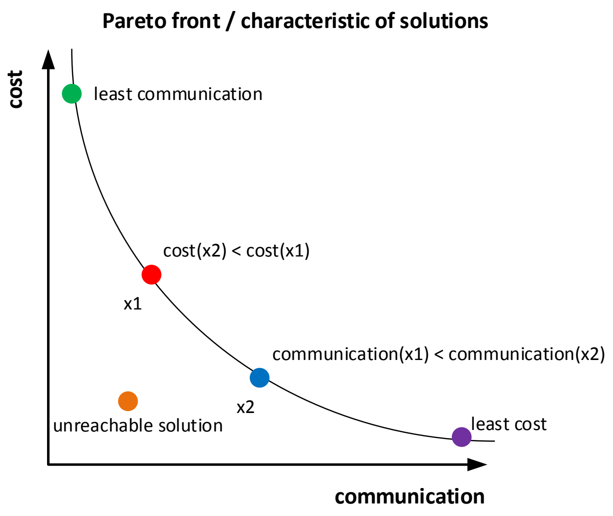


Figure 3.5. Pareto front in IncQuery-D multi-dimensional optimization.

Pareto front is interesting in the context of the IncQuery-D allocation because the found solution is lexicographically further optimized in the other dimension as well. For example if the user has chosen the communication as the primary optimization target, there are possibly more optimal solutions with the particular value. Therefore the optimizer looks further for a solution within that solution set where the cost is minimal. This concept is shown in Figure 3.6.

In case of IncQuery-D allocation optimization the Pareto front does not always look like as the above ones because there is no real negative correlation between the dimensions. The allocation sometimes has a globally optimal solution which means that a solution is feasible where both the communication and cost dimensions have minimum. Figure 3.7 illustrates such a scenario.

3.2 The IncQuery-D Allocator

The purpose of this section is to overview our proposed solution. We first discuss how the IncQuery-D architecture had to be extended with the components, required by

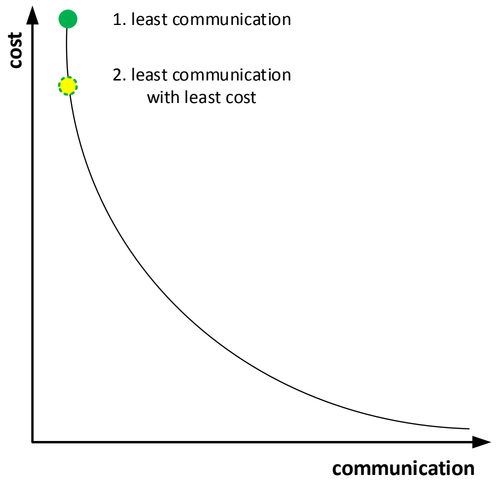


Figure 3.6. Lexicographical optimization.

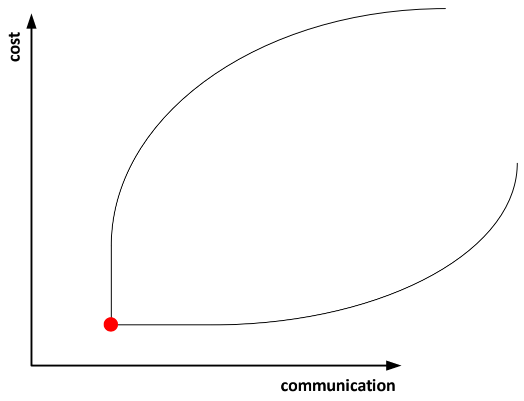


Figure 3.7. Pareto front when the global optimum exists.

allocation optimization, including a new subsystem called Monitoring to gather telemetry data necessary to drive the optimizer. Then we show the algorithmic approach to the optimization problem. Finally we discuss the heuristical prediction methods used by the optimization.

3.2.1 Extended IncQuery-D Architecture

The purpose of this section is to introduce the allocation related new components of the IncQuery-D system.

Figure 3.8 shows the extended architecture. IncQuery-D Tooling is a bundle of tools integrated into a development environment. The purpose of these tools is the convenient usage of the IncQuery-D system by providing different administration facilities for control and monitoring, and development facilities for query evaluation. Tooling is a client side component, and runs on the machine of the user.

One important part of the tooling is the query editor. IncQuery-D has a high-level graph pattern language and the queries can be formulated above a metamodel. The metamodel describes the structure of the data (e.g. the types of nodes and edges) and is defined in RDF [18]. After the query is written, the compiler transforms it to a Rete recipe. Basically, the Rete recipe is an abstract model of the query which describes the

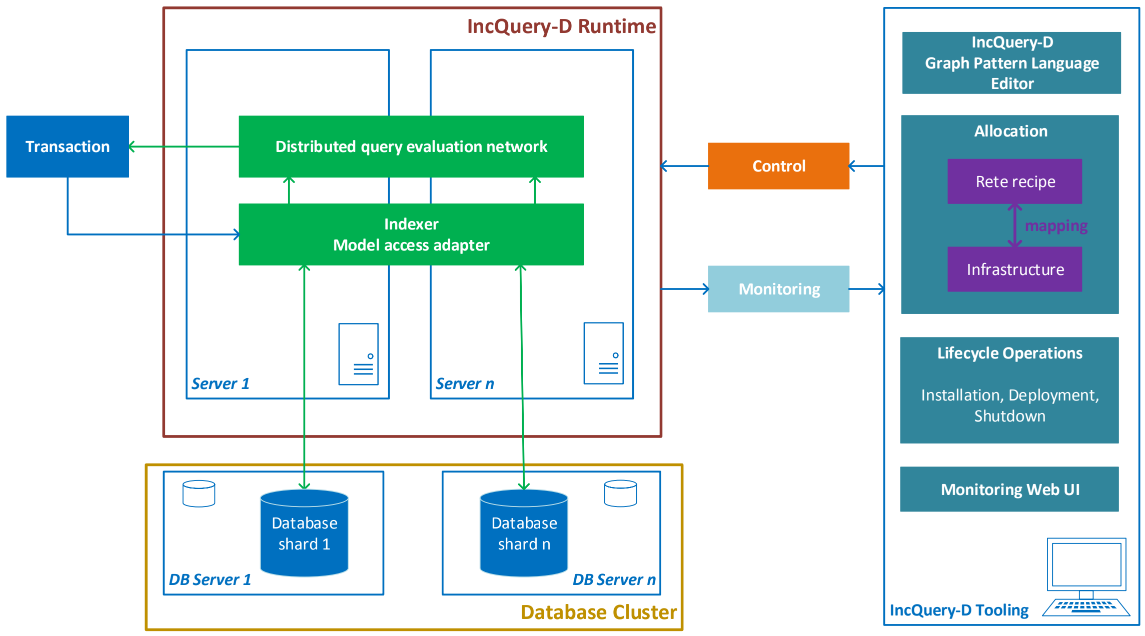


Figure 3.8. Extended Architecture of IncQuery-D.

layout of the Rete network with the Rete nodes and their connections. This workflow is illustrated in Figure 3.9.

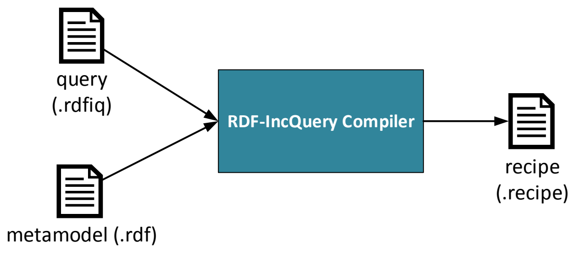


Figure 3.9. Compilation of the query.

The allocation component has the role of mapping the aforementioned Rete recipe to the available infrastructure. The functionality of this component will be further discussed in Section 3.2.2.

The Tooling component also provides commands for the lifecycle operations of the system. The user can install the IncQuery-D system to the server machines, queries can be deployed, their processing can be started, and finally the system can be shutdown.

The last important part of the tooling mentioned here is the Monitoring Web UI. This user interface provides monitoring data about the runtime state of the system. The Monitoring subsystem will be also discussed in more details in Section 3.2.3.

3.2.2 The Allocation Optimizer Subsystem

Allocation Strategy

Grouping of Rete Nodes From the perspective of allocation there are two groups of Rete nodes. The first group consists of the Memory nodes, which possess memory caches and are considered as memory-intensive nodes. The memory consumption of these nodes can grow large with the rise of the model size. These node types are the Input node, the Production node and the Beta nodes.

The other group contains the Non-Memory nodes, which do not require memory caches in order to work and are not considered as memory-intensive nodes. These node types are the Alpha nodes.

The purpose of this distinction is the separation of memory intensive computation nodes.

Rules of the Allocation The Rete node allocation is done by some simple rules. The main guideline is that the memory intensive Rete nodes should be separated from each other, therefore processes can contain only one Memory node. This way we avoid having processes with too large memory space. The prediction of process memory consumption will also be easier as we have to calculate with the possible memory increase (as the size of the model can grow) of one node only. Therefore, we can create memory efficient and good performance processes.

On the other hand, we want to minimize the number of processes used, therefore we will create exactly as many processes as many Memory nodes we have, one for each node.

The other important rule considers the placement of Non-Memory Rete nodes. As those do not consume much memory, they can be placed to processes with Memory nodes. This is reasonable since we assumed that in-process communication is fast. The general property of Non-Memory nodes is that they can produce less (or equal) data as output than they got as input. Therefore those are allocated to the same processes as their parents in order to reduce inter-process communication. If there are more of them in a sequence, then the whole sequence – until a Memory node is found – is allocated to the same process as this particular Memory node.

The application of these rules are shown in Figure 3.10.

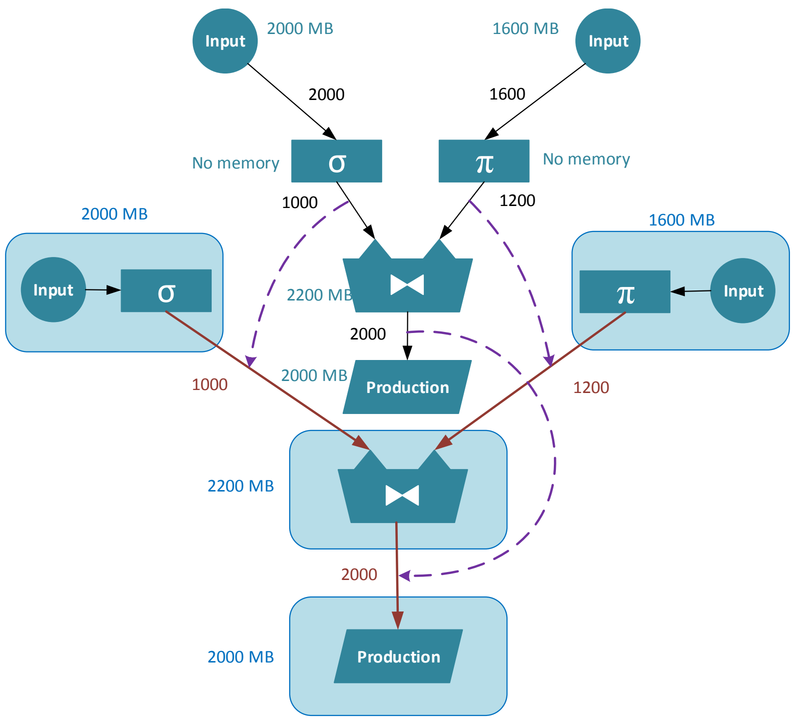


Figure 3.10. Allocation of sample Rete net to processes.

Estimations In this allocation phase there is another critical problem. Once the processes are given, we have to calculate the estimated memory consumption of the processes and the estimated traffic volume for the inter-process communication edges as these are important inputs for the next phase of the allocation. Therefore, we must try to predict these values according to statistics about the data. The heuristic methods used for this problem were discussed in Section 3.2.4. Now we assume that these parameters are already given.

Process Allocation Problem

If the processes and inter-process communication edges are given, then the next phase of the allocation is the mapping of processes to the available machines. However, the allocation of processes has resource related constraints as it was mentioned before. The expected memory usage of the processes on one machine should not exceed the memory capacity of this particular machine. With the satisfaction of these constraints, usually there are still a lot of allocation variants available for an allocation algorithm.

Figure 3.11 shows one possible solution with the deployment of processes to machines. If the processes are mapped to machines then the weight of inter-process communication edges can be determined by the multiplication factor coming from the communication overhead of the concerned hosts. With all of these information given, the total weight of communication in the system can be calculated.

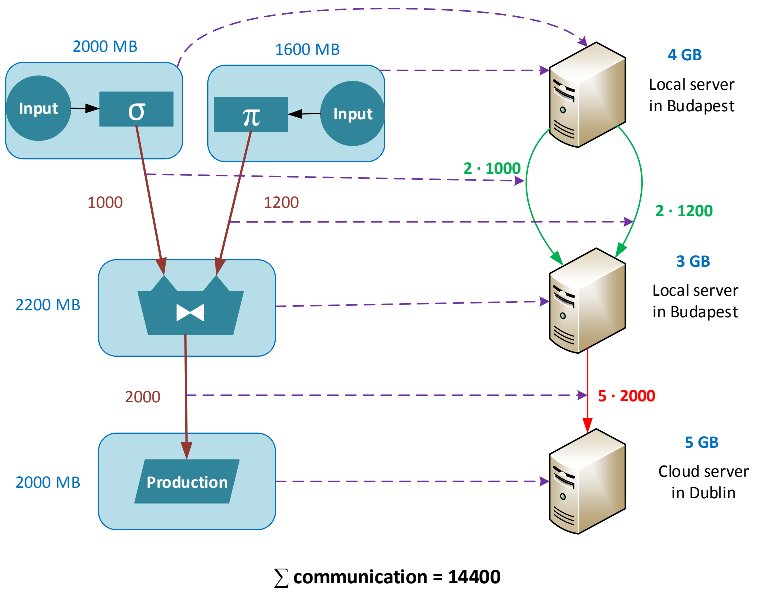


Figure 3.11. Sample allocation to server machines.

Figure 3.12 shows another possible allocation alternative.

One may observe the differences between the two configurations. The first difference is that in the latter, not all machines are used. The other difference is that the aggregate weight of communication is less in the case of the second configuration.

As it was mentioned earlier, the allocation has the functionality of mapping the Rete network to the available infrastructure. However, we saw in Section 3.1.1 that the allocation entails important optimization opportunities. Therefore the Allocation Optimizer subsystem has the role of providing an optimal allocation for a particular query.

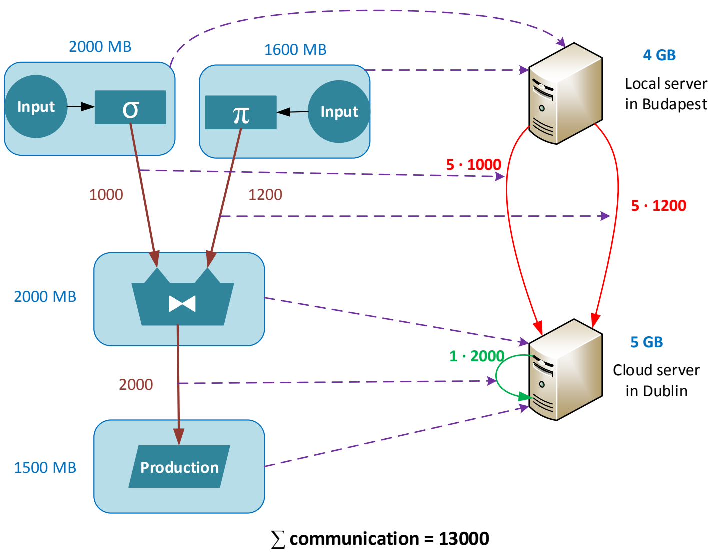


Figure 3.12. Better allocation of the configuration.

Inventories In order to create an allocation, the user has to provide information about the available infrastructure besides the queries. In the IncQuery-D terminology this is called *inventory*. The inventory is basically a resource description language and it has a metamodel (Figure 3.13) defined in EMF (Eclipse Modeling Framework) [8].

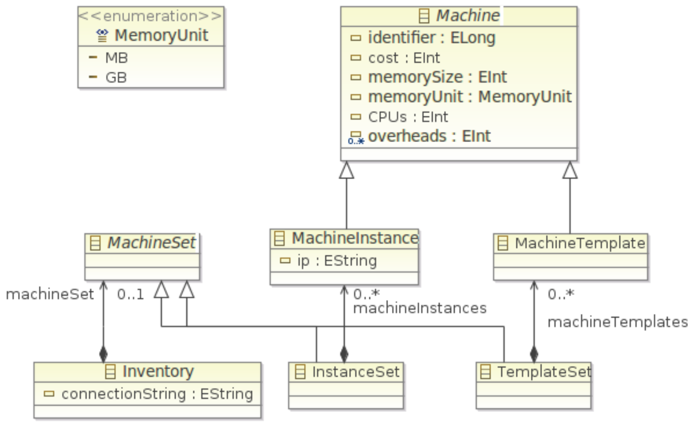


Figure 3.13. Metamodel of inventory description language.

The main element of the metamodel is the Inventory class. It has a connection string attribute which defines the database cluster IncQuery-D will work with. The Inventory can contain 1 machine set that can be either an InstanceSet or a TemplateSet. The difference is that the InstanceSet can contain MachineInstance objects and the TemplateSet can contain MachineTemplate objects. While MachineInstance represents an existing host computer, the MachineTemplate is basically a template. These templates are supported for user convenience as in a cloud infrastructure people can reserve more machines of different types. This way the user can select the appropriate templates and can run the allocation in order to find out how many instances should be reserved later.

The machines have different parameters like memory capacity and the number of CPUs. The mentioned communication overhead numbers between machines can be defined in this model as well. These numbers can be described with a matrix-style notation, the convention being that the numbers have to be given for each machine in the order they were defined in the model.

Figure 3.14 and Figure 3.15 show sample instance models with machines and templates.

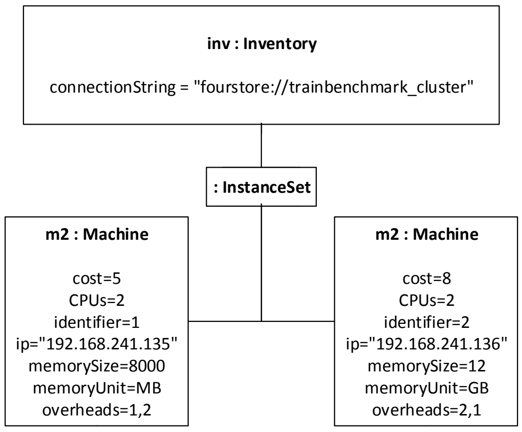


Figure 3.14. Inventory model with machines.

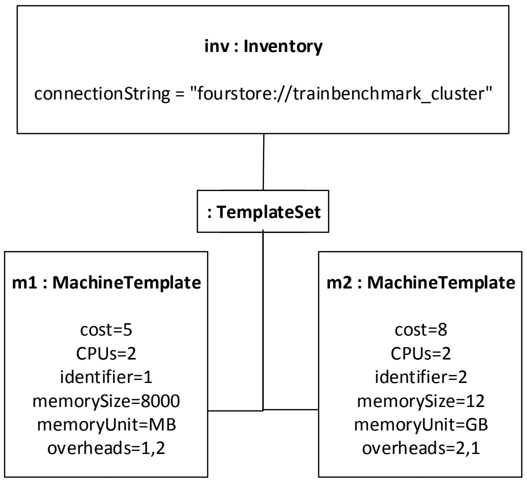
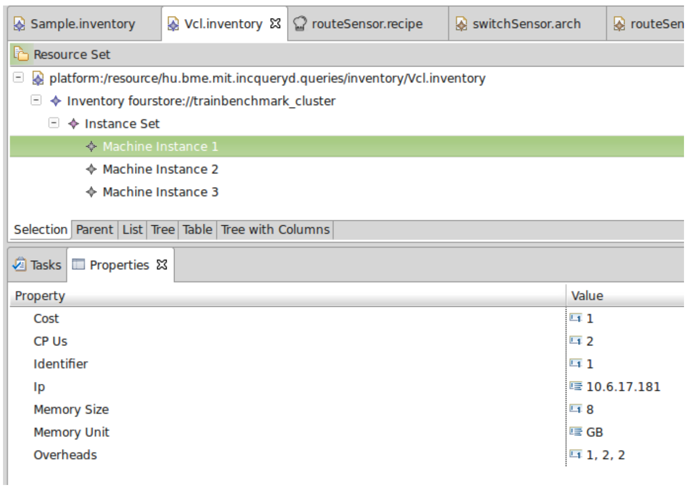


Figure 3.15. Inventory model with machine templates.

IncQuery-D Tooling also provides an editor for an inventory model that is shown in Figure 3.16.

If the user has provided the inventory model and has written the query then the allocation can be invoked. The Allocation Optimizer component takes the generated Rete recipe containing the structure of the Rete network and the inventory model containing the available resources as inputs. On the other hand, we would like to optimize the allocation, therefore the Allocation Optimizer also requires statistics about the data the queries will be run against. The values of the statistics come from the model containing databases. The statistics contain the cardinality of model elements coming from the Input nodes in order to know the amount of data which will enter the system. According to these statistics, the optimizer will use heuristics to estimate the memory consumption of JVMs and the intensity of data propagation between the connected Rete nodes.

In the allocation process, the Allocation Optimizer has to decide according to its inputs, which process a particular Rete node should belong to and which machine the processes will


Sample.inventory | Vcl.inventory | routeSensor.recipe | switchSensor.arch | routeSen

Resource Set
- platform:/resource/hu.bme.mit.incqueryd.queries/inventory/Vcl.inventory
  - Inventory fourstore://trainbenchmark_cluster
    - Instance Set
      - Machine Instance 1
      - Machine Instance 2
      - Machine Instance 3

Selection | Parent | List | Tree | Table | Tree with Columns

Tasks | Properties

Property	Value
Cost	1
CP Us	2
Identifier	1
Ip	10.6.17.181
Memory Size	8
Memory Unit	GB
Overheads	1, 2, 2

Figure 3.16. Inventory model editor.

be started on in order to get an optimal layout of the system. Therefore this component implements algorithms for the aforementioned allocation problems in order to solve them. The elaboration of this component will be discussed in details and illustrated with case studies in Chapter 5. The workflow can be inspected in Figure 3.17.

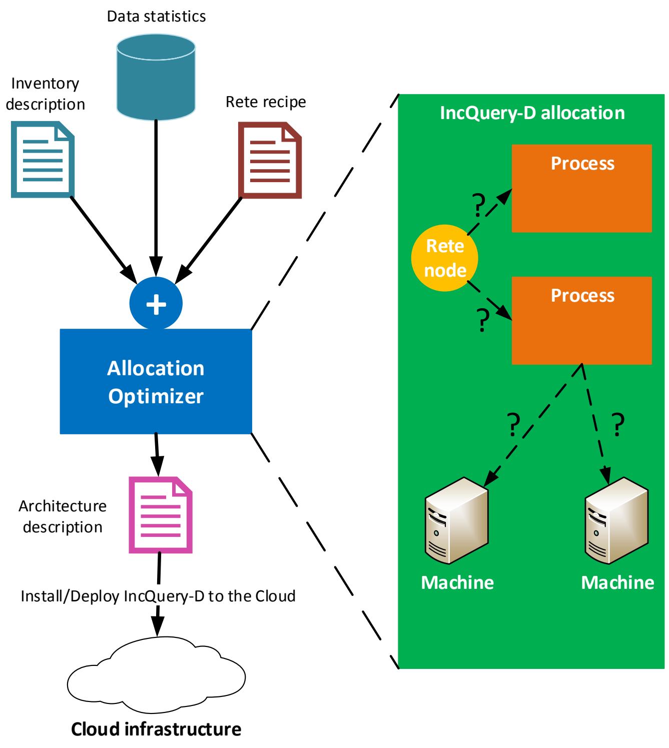


Figure 3.17. The Allocation Optimizer subsystem.

The output of the optimizer is an architecture description model. This model contains the aforementioned Rete recipe and the mappings to processes and machines.

Once the architecture description is created, the system can be deployed to the host computers, the processes will be started, the Rete nodes will be created in the appropriate processes and the query processing can be started.

3.2.3 Runtime Monitoring Subsystem

The purpose of the Monitoring subsystem is to provide real-time performance data about the IncQuery-D system as it can be extremely useful for us to see the runtime utilization of the resources and the behaviour of processes. It had an important role in the creation of heuristics (Section 3.2.4) for the allocation optimization problems.

The relation of the Monitoring subsystem to the IncQuery-D Runtime system is shown in Figure 3.18.

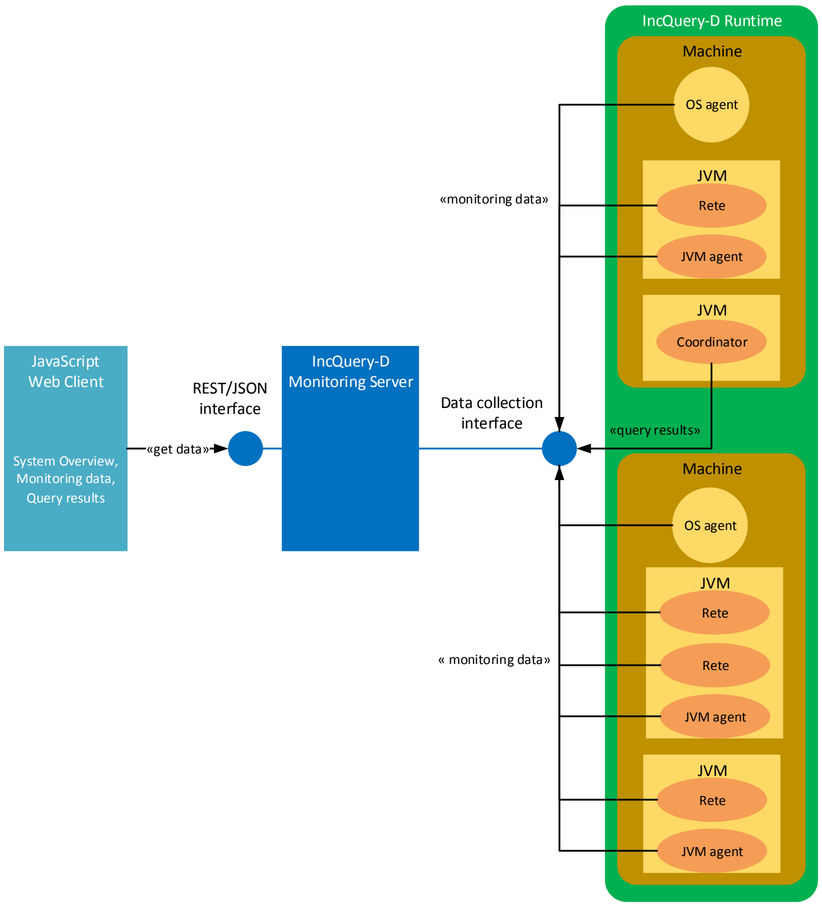


Figure 3.18. The IncQuery-D Monitoring subsystem.

The heart of the Monitoring subsystem is the central monitoring server. The task of this server is to collect the monitoring data from all sources and make them available on a convenient REST interface. The primary client of the server is the web based monitoring dashboard. This is a thin client web application created for the human inspectors of the system. This application and the related resources are hosted on the web server component of the monitoring server as static contents.

Infrastructure monitoring One of the most important tasks of the monitoring is the collection of host machine resource utilization and process behaviour data. The collected metrics for a machine include CPU, memory utilization and disk, network usage. The data is provided by an agent installed on each host. The collected metrics for Java Virtual Machine processes are CPU, memory usage and garbage collection related data. This data comes from a JVM agent created in each process. Figure 3.19 depicts what we can see on the monitoring user interface.

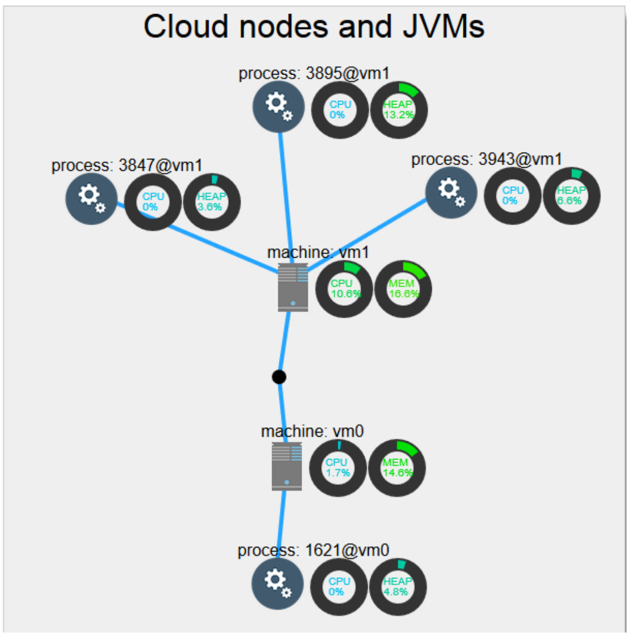


Figure 3.19. The Infrastructure on the Web UI.

In the above picture only the most important metrics are shown in little gauges next to the monitored components. The other metrics of the components become available in a heatmap-like view if we click on their names. The heatmap view is designed in order to be able to spot overused resources or high-intense resource usages easily. Figure 3.20 and 3.21 show these views.

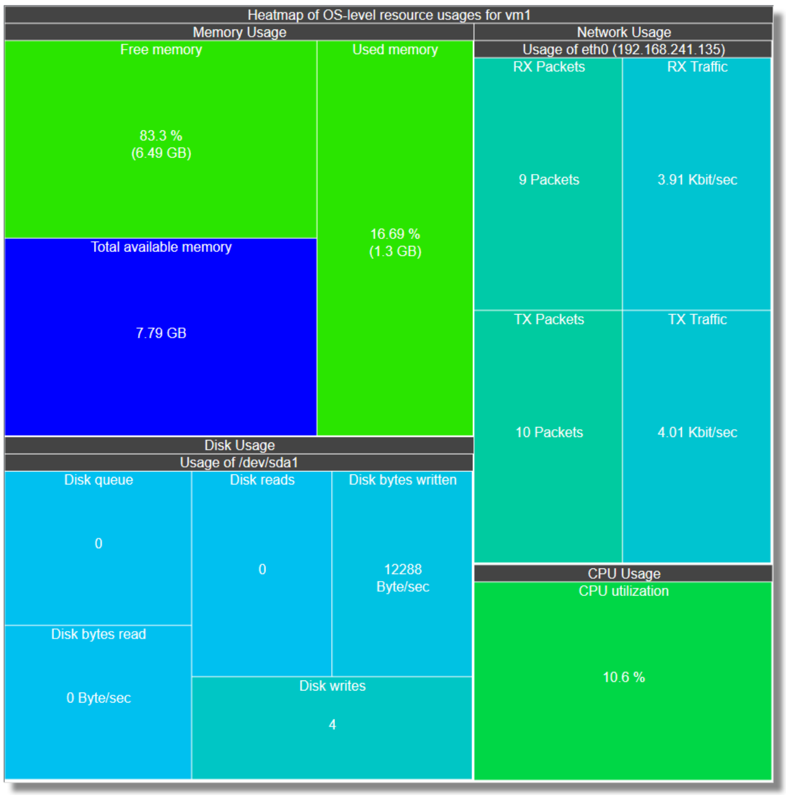


Figure 3.20. Operating System metrics Heatmap on the Web UI.

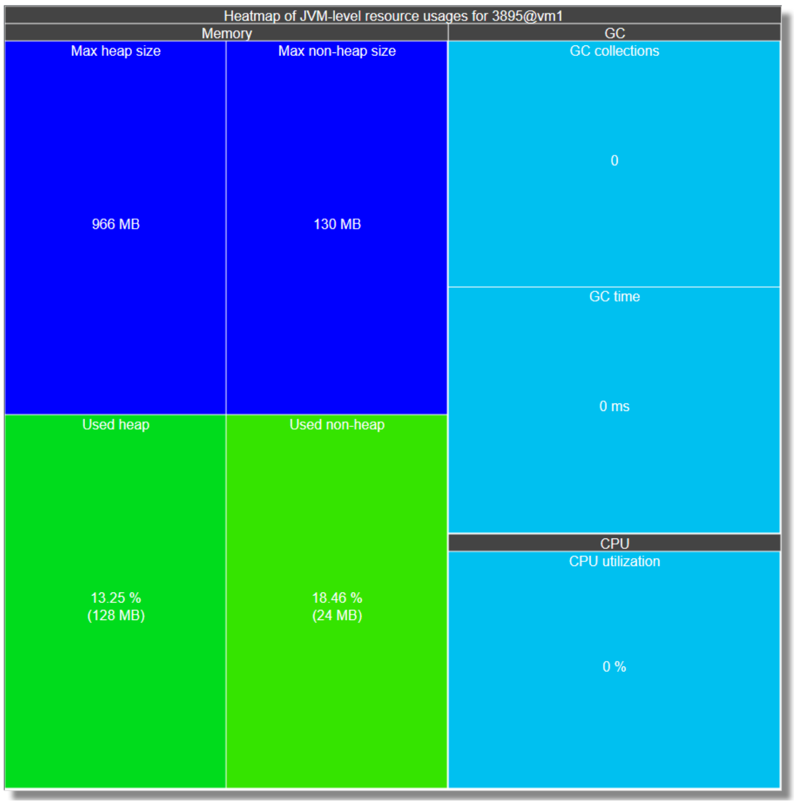


Figure 3.21. Process(JVM) Heatmap on the Web UI.

Rete node monitoring The other important task is the monitoring of Rete nodes. The most important characteristics of the Rete nodes are their memory usage including their collection sizes (if they have any) and the data propagation they do during the query evaluation. These characteristics are of key importance in the heuristics. The Rete layout of a query and the related metrics are shown in Figure 3.22.

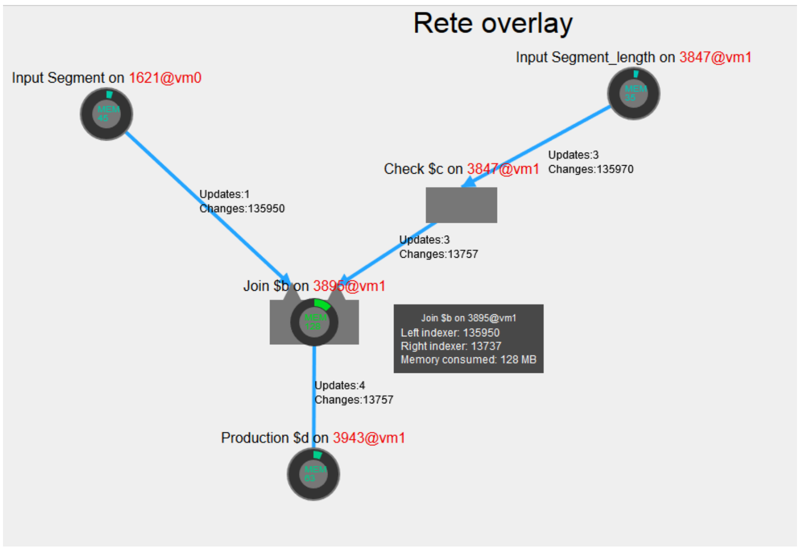
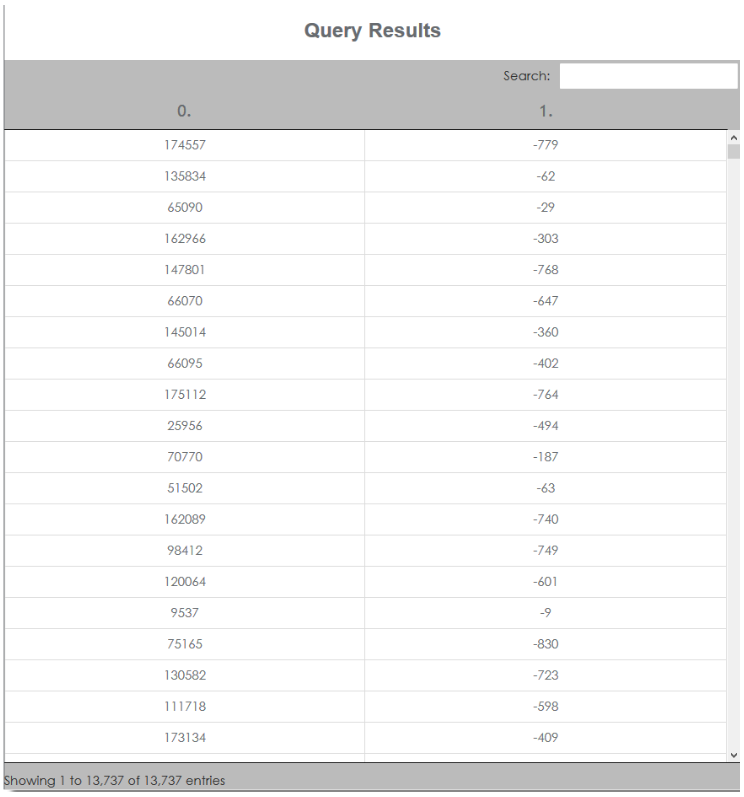


Figure 3.22. The Rete layout on the Web UI.

Query Results view The last important view on the monitoring web UI is the Query Results view. This view shows the actual results of a query.



Query Results

Search:

0.	1.
174557	-779
135834	-62
65090	-29
162966	-303
147801	-768
66070	-647
145014	-360
66095	-402
175112	-764
25956	-494
70770	-187
51502	-63
162089	-740
98412	-749
120064	-601
9537	-9
75165	-830
130582	-723
111718	-598
173134	-409

Showing 1 to 13,737 of 13,737 entries

Figure 3.23. Query results on the Web UI.

3.2.4 Heuristics in the Optimization

As it was mentioned before, the Allocation Optimizer requires the memory consumption of processes and the inter-process data traffic volume in order to be able to optimally allocate the processes to the available machines. However, these values can not be known prior to the runtime operation of the system. Therefore we have to use heuristic methods in order to make predictions for them. As the good approximation of these values is critical for the allocation optimization, we created measurements with the *Runtime Monitoring Subsystem* and used statistical methods in order to find good heuristics for our purposes. We will also use the *Runtime Monitoring Subsystem* in Chapter 5 to justify the correctness of these heuristics with examples.

Estimation of Process Memory Consumption

The processes of the IncQuery-D system are Java Virtual Machines. In order to create good heuristics for the memory consumption we have to keep some facts about the JVM memory in mind. The largest part of the memory is the heap space for dynamic memory allocations. Before the version 8 of Java, the maximum size of the JVM heap space must be set when we start the process and the JVM will not be able to exceed this limit during its lifetime. Therefore it is critical to allocate enough memory for the process. On the other hand we can not allocate too much memory for one process because that way the utilization of resources would be bad.

The JVM heap memory has two dominant parts, the so-called *Old Gen Space* and the *Eden Space*. The *Old Gen Space* contains the long-living objects which are needed by the process for a long period of time. For the processes of IncQuery-D these are the tuples stored by the *Rete nodes*. The *Eden Space* contains the short-living objects. The space of these objects is frequently freed by the *Garbage Collector*. These objects are the temporary objects produced by the code.

The memory usage characteristics of a process can be analyzed by profiler tools. Figure 3.24 shows a sample profiling with the *YourKit*[23] profiler. The typical memory usage of the JVM can be observed, the orange coloured space is the *Old Gen Space*. The blue coloured space is the *Eden Space*, it has typical sawtooth characteristics as these objects are created and destroyed frequently.

With the memory consumption heuristics we have to predict the size of the *Old Gen Space* and the size of the *Eden Space* used by our processes.

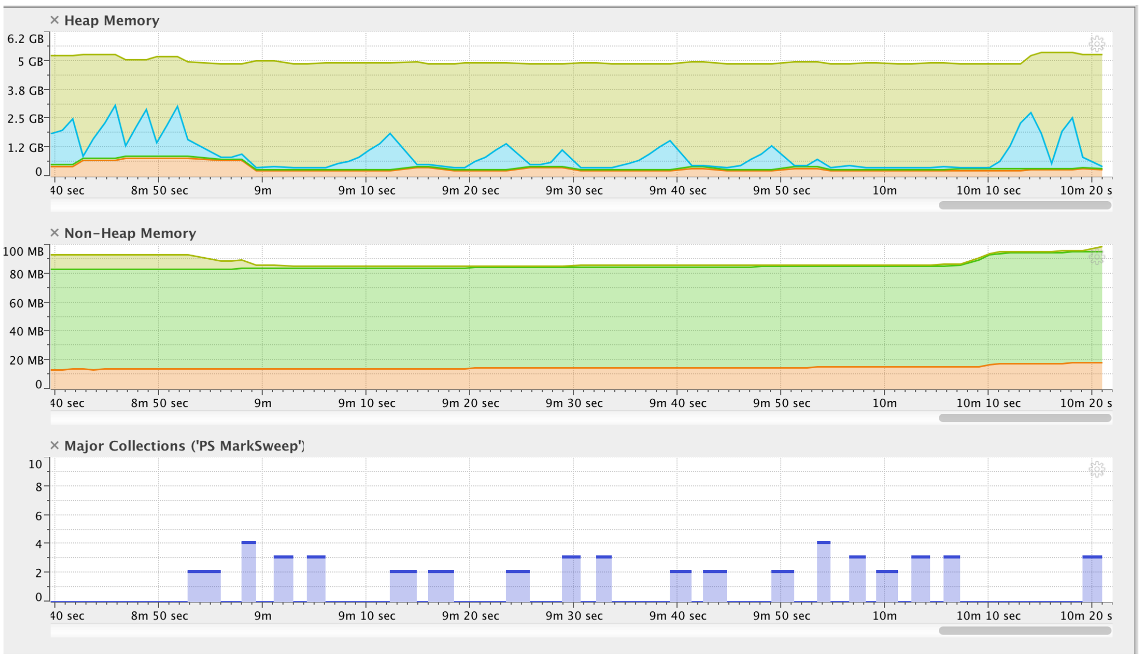


Figure 3.24. Memory Profiling.

We have also made measurements with the *Runtime Monitoring Subsystem* of IncQuery-D. We measured the heap memory consumption of the processes with different *normalized tuples* stored by their *Rete nodes*. We have created a scatterplot from the results which is shown in Figure 3.25. In order to create heuristics for the memory consumption, we fit a linear regression curve and determined a linear equation which will be the base of our prediction. The equation of this particular curve is $y = 0.0003 \cdot x + 52.969$. Basically this means that we need 0.0003 megabytes memory to store each normalized tuple, this will be the consumption of the *Old Gen Space*. The 52.969 shift is the estimated size of the *Eden Space*.

As we can see, there are some points above the regression curve. On the other hand, we also have to consider the increase of the model size. Therefore we allocate an additional 40% heap memory for the processes. As we do not want to have too small processes, the minimum amount of heap memory we allocate is 128 megabytes.

Therefore, for estimating the memory consumption, the formula is:

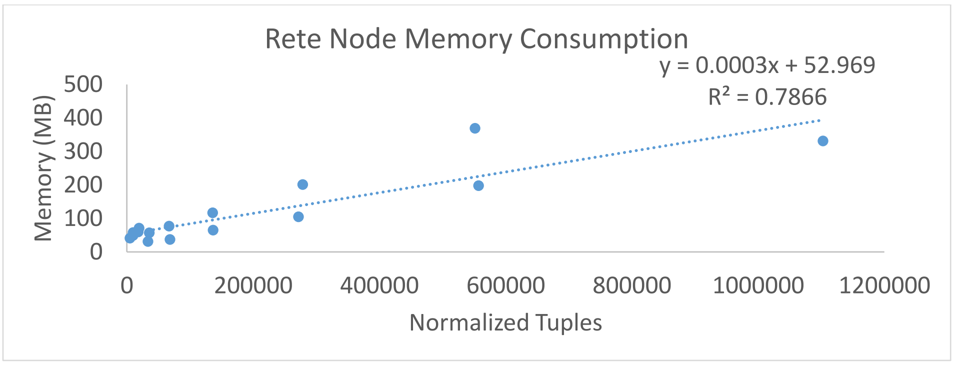


Figure 3.25. Memory Prediction for Rete Nodes.

$$\max = \{128, (0.0003 \cdot x + 52.969) \cdot 1.4\}$$

Estimation of Communication Intensity

We will use the expected normalized tuples sent between *Rete nodes* as the estimated communication intensity between processes. As we have seen in Section 3.1.1, multiplying the normalized tuples with the communication overhead approximated the communication time very well. Therefore, the normalized tuples will be good heuristics for our purposes.

Fortunately, we know how many elements are in the model from each type by querying the underlying database of IncQuery-D. This will give us how many tuples will enter the system through the *Input nodes*.

In order to create the heuristics we have to predict how many tuples will leave the different *Rete nodes*. The algorithm is simple, we have to go through the *Rete network*, starting by the *Input nodes* and we have to predict the outgoing normalized tuples by the incoming amount. We will also use the normal tuple definition here as *Rete nodes* work with tuples. The estimation will be based on the behaviour of the different *Rete nodes* and the fact that models are usually sparse graphs.

Estimation for Input Nodes The estimation for *Input node* communication is simple. They will propagate all of their tuples with all of their attributes.

Estimation for Alpha Nodes

Estimation for Check Node We expect the *Check node* to filter 90% of the tuples. The reason behind this is that we usually look for abnormal elements with such filter expressions, and usually there are not many of them in big models. Therefore, we expect to send the 10% of tuples and all of their attributes.

For example, if we have 10000 tuples with two attributes, then we expect to send 1000 tuples with both attributes and the normalized tuple amount will be 2000.

Estimation for Trimmer Node As we know which attributes of tuples will be removed by the *Trimmer node* we can be sure that only the remaining attributes will be propagated. As *Trimmer node* also removes the duplicate tuples after the trim, we expect to propagate the 90% of the trimmed tuples.

For example, if we have 10000 tuples with two attributes and the second attribute will be removed, then we expect to propagate 9000 tuples with the first attribute only and therefore the sent normalized tuple amount will be 9000.

Estimation for Beta Nodes

Estimation for Join Node The upper bound for the tuple number produced by *Join node* is the product of tuple number coming from its left parent and the tuple number coming from the right parent. As we usually join nodes and edges this would mean that all edges have all other nodes as endpoint which is not realistic. Because of this fact, and considering sparse graphs we expect to propagate only 1% of the possible matches.

For example, if we join a node type – 500 tuples with one attribute – with an edge type – 100 tuples with two attributes – and the first attribute of the edges must match with the one attribute of the nodes, then we expect to propagate 500 tuples. However, the propagated tuples will only have two attributes as we only use one of the matching attributes (as those are the same), therefore the amount of propagated normalized tuples will be 1000.

Estimation for Antijoin Node The upper bound for the tuple number produced by *Antijoin node* is the tuple number coming from its left parent as it will only propagate elements from there. The *Antijoin node* looks for graph elements that have no corresponding element from the other set. As this usually checks for abnormal structure, we expect to propagate only 10% of the tuples. The arity of the tuples will be the same.

For example, if we have 5000 tuples (with one attribute) coming from the left parent and 10000 elements coming from the right parent, then we expect to propagate 500 tuples with one attribute, therefore the amount of propagated normalized tuples will be 500.

Considering Incremental Behaviour As IncQuery-D is an incremental query engine we have to keep in mind the incremental behaviour of the queries.

The heuristics mentioned before gave approximations for the first evaluation of the query and not the incremental reevaluations. However, the possibility of modification is higher where there are more model elements. Therefore, the communication characteristics of *Rete nodes* usually remain similar compared to each other, just with a lower volume as the evaluation of model modifications involve less data propagation than the first evaluation of the query on the whole model. On the other hand we have to note that the incremental behaviour fully depends on the particular workload, therefore it can not be estimated precisely.

Chapter 4

Formalization of the Allocation Optimization Problems

The purpose of this section is to describe the optimization problems of the IncQuery-D allocation process. These problems are the Communication Minimization and the Cost Minimization problem. Once the problems are formalized, they will be analyzed in terms of computational complexity in order to find the proper way to solve them.

4.1 The Communication Minimization Problem

As it was shown in Figure 3.3 the transmission of large data between remote machines can have significant time overhead compared to transmission between processes on the same host. Since the communication overhead significantly differs between processes on local and remote computers, minimizing the overall volume of remote communication in the network is an important optimization target.

Fundamental assumptions The purpose of the allocation is the placement of processes to the available computers since the mapping of Rete nodes (Section 2.3) to processes is done in a preceding step. The fundamental rule is to place exactly one Rete node with internal memory to one process because those can consume vast amount of memory in case of big models. Rete nodes with no internal memory are placed to the same process as their parent node since in-process communication is fast.

The computers as resources are characterized with their memory capacity but also other parameters e.g. CPU usage could be taken into account. The parameters of communication channels (e.g. link quality, distance) between computers are characterized with a scalar and can be called overhead. This value represents the likelihood of communication between two machines as if the value is large then this channel will not be preferred (especially for large amount of data). The ratio of these numbers is more important than the numbers themselves. For example if we wish to express that the connection between ith and jth machines has 2 times better parameters than the connection between kth and jth machines then we can write $o_{i,j} = x$ and $o_{k,j} = 2x$, where $o_{i,j}$ and $o_{k,j}$ are the overhead values.

The resource consumption of processes is described by their memory requirements. This is the amount of memory that is sufficient for the process to run. As Rete nodes in the processes communicate with each other, there will be data transfer between processes. This is represented with an edge between the processes which has a weight. This weight describes the amount of data transfer between the processes and is measured in *normalized tuples* (3). The memory usage and the communication volume of the processes can not be determined exactly in allocation time. Therefore we use heuristics to predict these values from the characteristics of the Rete net and nodes.

During the allocation, the capacity of the machines should not be overused by the processes while the overall volume of communication in the network should me minimized.

4.1.1 Formalization of the Communication Minimization Problem

Input:

Memory requirements of processes

$$S = (s_1, s_2, \cdots, s_n) \tag{4.1}$$

Predicted memory consumption of the processes measured in megabytes.

Memory capacity of machines

$$C = (c_1, c_2, \cdots, c_m) \tag{4.2}$$

Memory capacity of machines that can be used by the processes measured in megabytes.

Communication edges between processes

$$E_{n,n} = \begin{bmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,n} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{n,1} & e_{n,2} & \cdots & e_{n,n} \end{bmatrix} \tag{4.3}$$

The matrix E contains the amount of communication between processes measured in *normalized tuples*. For the ith process the ith row describes these values. For example, 50000 *normalized tuples* communication intensity between ith and jth processes is described as $e_{i,j} = 50000$.

Communication overheads between machines

$$O_{m,m} = \begin{bmatrix} o_{1,1} & o_{1,2} & \cdots & o_{1,m} \\ o_{2,1} & o_{2,2} & \cdots & o_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ o_{m,1} & o_{m,2} & \cdots & o_{m,m} \end{bmatrix} \tag{4.4}$$

The matrix O contains the overhead multipliers between machines. For the ith machine the ith row describes these values. This matrix is usually symmetric to its main diagonal.

The convention is that the ith row of E and the s_i (ith element of S) weight belong to the ith process and the jth row of O and the c_j (jth element of C) capacity belong to the jth machine. This way the processes and machines can be identified by their number.

The following W matrix contains the calculated communication weights between processes.

$$W_{n,n} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,n} \end{bmatrix} \tag{4.5}$$

The capacity constraints of the machines can be described with the following linear inequality system.

$$\begin{aligned} s_1 \cdot x_{1,1} + s_2 \cdot x_{1,2} + \cdots + s_n \cdot x_{1,n} &\le c_1 \\ s_1 \cdot x_{2,1} + s_2 \cdot x_{2,2} + \cdots + s_n \cdot x_{2,n} &\le c_2 \\ &\vdots \\ s_1 \cdot x_{m,1} + s_2 \cdot x_{m,2} + \cdots + s_n \cdot x_{m,n} &\le c_m \end{aligned} \tag{4.6}$$

$$\forall j : \sum_{i=1}^{m} x_{i,j} = 1, x_{i,j} \in \{0,1\} \tag{4.7}$$

The first inequalities state for each machine that the memory capacity cannot be exceeded. The last equation states for each process that it must be allocated to exactly one machine. So it is guaranteed that the resources of a machine will not be overused while all processes will be allocated. The communication weights are calculated with the following formula:

$$\forall i, j, k, l, k \neq l : x_{i,k} + x_{j,l} \geq 2 \rightarrow w_{k,l} = e_{k,l} \cdot o_{i,j} \tag{4.8}$$

It tells that if two processes are being allocated to the ith and jth machines then the communication volume between them has to be multiplied with the overhead expected between these two machines.

Output:

$$\text{weight} = \min \left\{ \sum_{\substack{1<i<n \\ 1<k<n}} w_{i,k} \right\}, \tag{4.9}$$

while we assume that $w_{i,k}$ is 0 if there is no communication edge between the ith and kth processes. The computed optimal allocation can be extracted from the $x_{i,j}$ variables where the 1 value means that the jth process is allocated to the ith machine.

4.1.2 The Complexity of the Communication Minimization Problem

The Communication Minimization is a combinatorial optimization [7] problem because it needs to find an optimal solution from a finite domain. Not only in algorithm theory, but also in practice, it is a good idea to examine the complexity of the problem before trying to find an effective algorithm to solve it. This rule especially applies for combinatorial optimization problems where the problem often turns out to be $\mathcal{NP}$-hard [37].

With the fact of $\mathcal{NP}$-hardness in mind, the algorithm designer often has to decide between an optimal solution and an efficient algorithm. If the algorithm designer is satisfied with feasible solutions then the use of approximation algorithms [28] might be a good decision. However, if not, the designer can try to reduce the problem effectively to SAT (Satisfiability) [6] or CSP (Constraint Satisfaction Problem) [39] problems. The advantages of this idea are that efficient (though not polynomial time) solvers exist for these problems and the problem can be described declaratively.

We now prove that the Communication Minimization problem is $\mathcal{NP}$-hard. $\mathcal{NP}$-hardness can be proved by Karp-reduction [37]. The basic idea is to show that the input of a well-known $\mathcal{NP}$-hard problem can be transformed to the input of our problem and the output vice versa so that a hypothetic or existing algorithm for our problem would always give the correct solution for the other problem. The transformation of the inputs and outputs must be done in polynomial time. This shows that our problem is at least as hard as the well-known $\mathcal{NP}$-hard problem because if we have an efficient algorithm for our problem, then we have efficient algorithm for the other problem as well. This is usually denoted as $X \prec Y$, which means that Y problem is at least as hard as the X.

Theorem 1. The Communication Minimization problem is $\mathcal{NP}$-hard.

Proof 1. To prove this theorem we will reduce the well-known $\mathcal{NP}$-hard Knapsack problem to the Communication Minimization, Knapsack $\prec$ Communication Minimization will be shown.

The Knapsack Problem [11]

Input:

Knapsack capacity

$$W \tag{4.10}$$

Item values

$$(v_1, v_2, \cdots, v_n) \tag{4.11}$$

Item weights

$$(w_1, w_2, \cdots, w_n) \tag{4.12}$$

Given the set of n items, each with a weight and value, it has to be ensured that the weights of the items will not exceed the capacity of the knapsack:

$$\sum_{i=1}^{n}(w_i \cdot x_i) \leq W, \qquad x_i \in \{0, 1\}, \tag{4.13}$$

while the aggregate value of the items put into the knapsack should be maximized:

$$\mathrm{V} = \max\left\{\sum_{i=1}^{n}(v_i \cdot x_i)\right\}, \tag{4.14}$$

where V is the output, the $x_i = 1$ means that the ith item is put into the knapsack and $x_i = 0$ means that it is not.

Reduction of the Knapsack Problem

The first problem is the difference between the objective functions: in Knapsack it is maximization, in Communication Minimization it is minimization. However, it is fairly easy to create a minimization problem from a maximization one. Only the cost function should be multiplied by -1, since if a solution had maximal value of the cost function then its negative should have minimal.

The transformation of the inputs will be the following. In this context n will be the number of items in the Knapsack problem.

$$(w_1, w_1, w_2, w_2, \cdots, w_n, w_n) \mapsto (s_1, s_2, \cdots, s_{2n-1}, s_{2n}) \tag{4.15}$$

This means that for each items in the Knapsack problem, the transformation will order 2 processes with the same weights. The ith item will be mapped to the $(2i\text{th}, 2i - 1\text{th})$ process pairs.

$$\left(W, W, 2 \cdot \sum_{i=1}^{n} w_i\right) \mapsto (c_1, \cdots, c_m) \tag{4.16}$$

There will be 3 machines with the capacities $(W, W, 2 \cdot \sum_{i=1}^{n} w_i)$. The last value is 2 times the total weight of items in the Knapsack problem to ensure that all processes can be allocated to 1 machine as this is a constraint in the Communication Minimization problem, otherwise the problem can not be satisfied.

$$\begin{array}{c} \\ 1 \\ 2 \\ \vdots \\ 2i-1 \\ 2i \\ \vdots \\ 2n-1 \\ 2n \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & \cdots & 2i-1 & 2i & \cdots & 2n-1 & 2n \end{array} \\ \begin{pmatrix} 0 & -v_1 & \cdots & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & -v_i & \cdots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & -v_n \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 \end{pmatrix} \end{array} \mapsto E_{2n,2n} = \begin{bmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,2n} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,2n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{2n,1} & e_{2n,2} & \cdots & e_{2n,2n} \end{bmatrix} \tag{4.17}$$

There will be n edges, 1–1 between each process pairs with the negative weight of the item represented by the pair. So this means there will be $-v_i$ weighted communication edge between $2i - 1$th and $2i$th processes, $\forall i \in \{1, \cdots, n\}$. Using the negative of the weight has an important role in mapping the maximization problem to a minimization one.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mapsto O_{3,3} \tag{4.18}$$

Between the 2 W capacity machines, the overhead will be 1, all the other overhead values are 0.

We will show, that if we transform the input of the Knapsack problem like this then a hypothetic algorithm for the Communication Minimization problem will solve it correctly.

Figure 4.1 helps to review how the algorithm operates on the aforementioned input.

Pick the $2i-1$th , $\forall i \in 1, \cdots, n$ processes (one of each pair). The subsets of this process set will fit into a W capacity machine exactly when the knapsack item sets represented by them fits into the knapsack. This is fairly obvious because of the same sizes and capacity. The same is true for the other n $(2i\text{th}, \forall i \in 1, \cdots, n)$ processes and the other W capacity machine. It is also obvious that all n communication edges go between these disjoint process sets. However, the algorithm will prefer allocating the endpoints of the edges to the W machines because this way it can reduce the objective function, in every other case the value of the edge would be multiplied by 0.

Note that in case of a positive edge value, the algorithm will allocate the two sides to the "∞" machine because that way it counts with 0, but this is is not a problem because a positive edge value would mean a negative item value for the knapsack problem. These items can be trivially left out of the knapsack as well because they would reduce the total value.

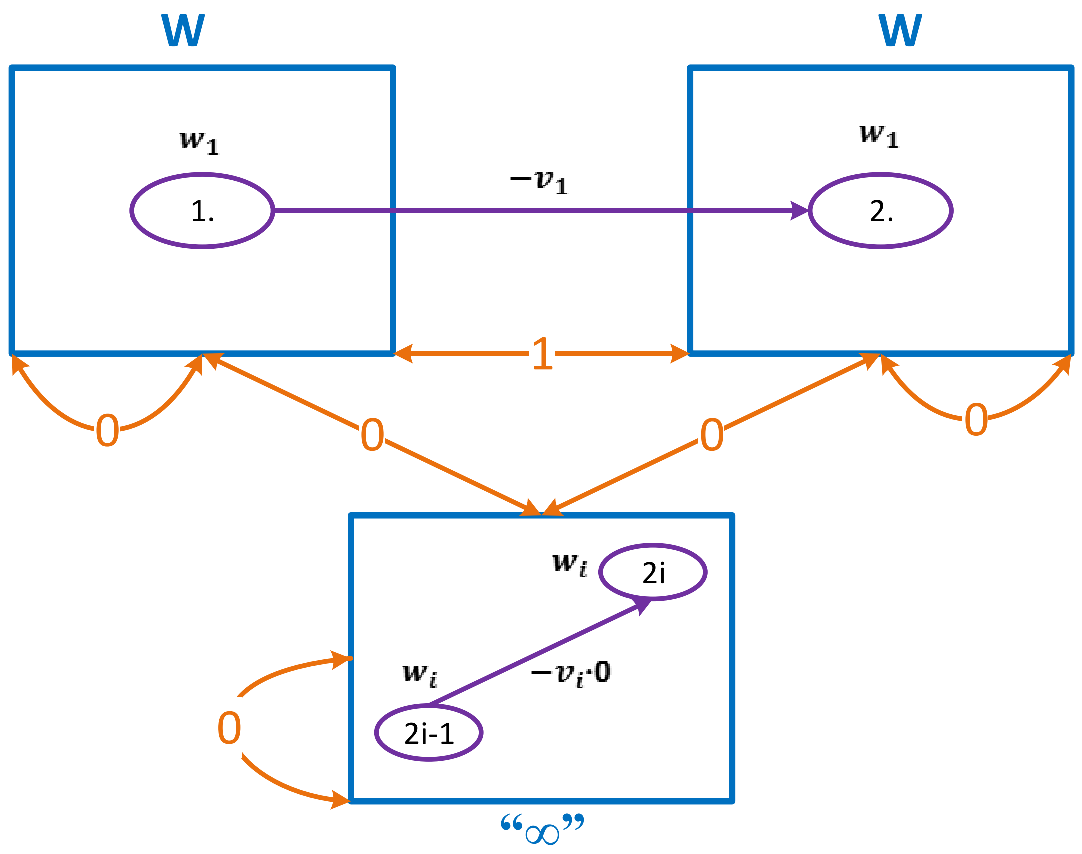


Figure 4.1. Reducing Knapsack to Communication Minimization.

After these observations, we can make the following two statements. Mark the minimum of the Communication Minimization problem with A and the maximum of the reduced Knapsack problem with B.

Statement. $-A \leq B$ **.**

Proof 2. Indirect. Suppose that we have reduced the Knapsack problem to the Communication Minimization as described above and $-A > B$. $-A = -(-v_i - \cdots - v_j - \cdots - v_l)$, because the minimum will be produced from some negative weighted edges multiplied by 1 as illustrated in Figure 4.2.

However, if such solution exists for the Communication Minimization it means that the $ith, \cdots, jth, \cdots, lth$ items with $v_i, \cdots, v_j, \cdots, v_l$ values fit into knapsack. So there is a solution for the Knapsack problem where the objective function takes the value $-A = (v_i + \cdots + v_j + \cdots + v_l)$ which is contradiction, the statement is true.

Statement. $-A \geq B$ **.**

Proof 3. If we can show a solution of the Communication Minimization where the objective function takes the value of $-B$ then the statement is true.

Suppose that the Knapsack problem has maximum with the ath, bth, $\cdots$, kth items. Then $B = v_a + v_b + \cdots + v_k$.

But then the $2a - 1$th, $2b - 1$th, $\cdots$, $2k - 1$th processes in the transformed Communication Minimization can be allocated to one W capacity machine and $2a$th, $2b$th, $\cdots$, $2k$th processes to the other W machine. Between these process pairs

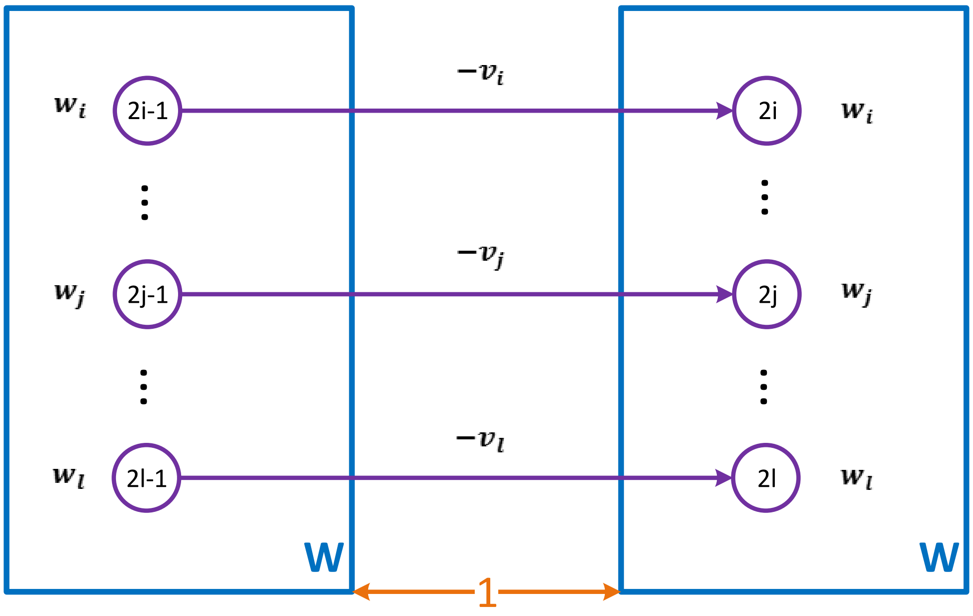


Figure 4.2. Solution for the Communication Minimization.

there are $-v_a, \cdots, -v_b, \cdots, -v_k$ weighted edges. If these process pairs are allocated to the two W machines then the objective function takes the $-v_a - \cdots - v_b - \cdots - v_k = -(v_a + v_b + \cdots + v_k) = -B$. That solution is exactly what we were looking for. ∎

If both $-A \leq B$ and $-A \geq B$ are true then $-A = B$. So if the output of the transformed Communication Minimization problem is multiplied by -1 then it will give good solution for the original Knapsack problem.

The last step to show is that the transformations have polynomial complexity. The creation of the S array (process memory requirements) has the complexity of $\Theta(n)$. The creation of the C array (machine capacities) has the complexity $\Theta(n)$. The creation of the E matrix (process communication edges) has the complexity $\Theta(n^2)$ and it takes $\Theta(1)$ steps to produce the O matrix (machine communication overheads). The tra nsformation of the output is $\Theta(1)$. So the overall complexity is $\Theta(n) + \Theta(n) + \Theta(n^2) + \Theta(1) + \Theta(1) = \Theta(n^2)$.∎

4.2 The Cost Minimization Problem

While in the previous section we formalized the communication minimization it is often an important requirement to allocate our system with the least possible cost. Furthermore as we saw in Figure 3.6, even if the system is primarily optimized for communication, it can be further optimized lexicographically in the cost dimension. This section describes the Cost Minimization problem if the costs of using the machines are given.

The same assumptions are made as in Section 4.1.

4.2.1 Formalization of the Cost Minimization Problem

Input:

Memory requirements of processes

$$S = (s_1, s_2, \cdots, s_n) \tag{4.19}$$

Memory capacity of machines

$$C = (c_1, c_2, \cdots, c_m) \tag{4.20}$$

Cost of machines

$$B = (b_1, b_2, \cdots, b_m) \tag{4.21}$$

The monetary cost of each machines.

Given the vector w which contains the cost of each machines in the system. Assume that the vector w contains the cost of the machine if it is used or 0 if it is not used.

$$w_m = [w_1, w_2, \cdots, w_m] \tag{4.22}$$

We may establish the following constraints as in the Communication Minimization problem:

$$\begin{aligned} s_1 \cdot x_{1,1} + s_2 \cdot x_{1,2} + \cdots + s_n \cdot x_{1,n} &\leq c_1 \\ s_1 \cdot x_{2,1} + s_2 \cdot x_{2,2} + \cdots + s_n \cdot x_{2,n} &\leq c_2 \\ &\vdots \\ s_1 \cdot x_{m,1} + s_2 \cdot x_{m,2} + \cdots + s_n \cdot x_{m,n} &\leq c_m \end{aligned} \tag{4.23}$$

$$\forall j : \sum_{i=1}^{m} x_{i,j} = 1, \qquad x_{i,j} \in \{0,1\} \tag{4.24}$$

The cost of the machine is calculated with the following formula:

$$\forall i \in \{1, \cdots, m\} : \sum_{j=1}^{n} x_{i,j} \geq 1 \rightarrow w_i = b_i \tag{4.25}$$

This describes for each machine that if at least 1 process is allocated to it, then it should be calculated with its cost. Otherwise the w_i value will remain 0.

Output:

$$\text{cost} = \min \left\{ \sum_{i=1}^{m} w_i \right\}, \tag{4.26}$$

while we assume that w_i is 0 if the ith machine is not used. The computed optimal allocation can be extracted from the $x_{i,j}$ variables where the 1 value means that the jth process is allocated to the ith machine.

4.2.2 The Complexity of the Cost Minimization Problem

Theorem 2. The Cost Minimization problem is $\mathcal{NP}$-hard.

Proof 4. To prove this theorem we will reduce the well-known $\mathcal{NP}$-hard Bin Packing problem to the Cost Minimization, so formally Bin Packing $\prec$ Cost Minimization will be shown.

The Bin Packing Problem [37]

Input: List of n items with positive sizes $(a_1, a_2, \cdots, a_n)$ to pack, and set $B = \{1, \cdots, m\}$ of bins with capacity V.

$$\sum_{j=1}^{n}(a_j \cdot x_{i,j}) \leq V \cdot y_i, \qquad \forall i \in \{1, \cdots, m\} \tag{4.27}$$

This means that the capacity of the bins can not be exceeded with the sizes of items packed into them.

$$\sum_{i=1}^{m} x_{i,j} = 1, \qquad \forall j \in \{1, \cdots, n\} \tag{4.28}$$

But all items should be packed into one bin.

$$y_i \in \{0, 1\}, \qquad \forall i \in \{1, \cdots, m\} \tag{4.29}$$

$$x_{i,j} \in \{0, 1\}, \forall i \in \{1, \cdots, m\} \text{ and } \forall j \in \{1, \cdots, n\}, \tag{4.30}$$

where $y_i = 1$, if bin i is used and $x_{i,j} = 1$, if item j is put into bin i.
Output:

$$\mathrm{C} = \min \left\{ \sum_{i=1}^{m} y_i \right\} \tag{4.31}$$

The number of bins used for the packing should be minimized.

The Reduction of the Bin Packing Problem

We give the transformations of inputs, outputs and show their correctness. The basic idea is to map the items of Bin Packing to the processes of Cost Minimization with the same weights. The machines in Cost Minimization will play the role of the bins. As the bins, they also have uniform capacity, and have the cost of 1 as used bins count with 1 in the objective function.

The sizes of n items are mapped to the memory requirements of n processes:

$$(a_1, a_2, \cdots, a_n) \mapsto (s_1, s_2, \cdots, s_n) \tag{4.32}$$

The capacities of m bins are mapped to the memory capacities of m machines:

$$V \mapsto c_i, \qquad \forall i \in \{1, \cdots, m\} \tag{4.33}$$

The cost of each machine is 1:

$$1 \mapsto b_i, \qquad \forall i \in \{1, \cdots, m\} \tag{4.34}$$

▪

If we run the hypothetic algorithm for the Cost Minimization problem with the aforementioned input and map its output to the Bin Packing output (cost $\mapsto C$) then the reduced problem will be solved as well. This can be easily seen as the items are mapped to the processes with the same sizes and the bins are mapped to machines with the same capacities. As the cost of each machine is 1 and the capacity is V, the optimal solution will give the amount of machines used for the allocation. This is exactly the number of bins that the items could be packed into.

The last step is to show that the input and output transformations are computed in polynomial time. This is also easy to see, we had to iterate through the items and bins once while the outputs where the same. So the complexity of the transformations is $\Theta(n + m) + \Theta(1) = \Theta(n + m)$.

Chapter 5

Elaboration

The purpose of this chapter is to illustrate the internal operations of the Allocation Optimizer (Section 3.2.2) subsystem of IncQuery-D with examples. For the sake of clarity these examples will be simpler than the ones used in the benchmark (6).

5.1 Algorithms for the Problems

It was shown in Section 4.1.2 and Section 4.2.2 that the allocation problems of IncQuery-D are $\mathcal{NP}$-hard. Because of that we decided to use CSP [39] as an approach to solve these problems. The advantage of this approach is that the problems can be stated in a declarative manner. The formal description of the problems in Section 4.1.1 and Section 4.2.1 are precise enough for a solver as those work with a system of mathematical and logical formulas. In the IncQuery-D allocation process we use the Google OR-Tools [10] solver.

So the allocation algorithm consists of the placement of Rete nodes to processes, calculating heuristics for process memory usage and communication volume, and transforming the problem to a system of mathematical formulas in order to be able to give it to the solver as an input. After the solver found the optimal solution, the allocation can be extracted from the variables as it was shown.

5.1.1 OR-Tools API

The purpose of this section is to show the API of the OR-Tools and how the different mathematical and logical formulas of our problems can be stated for the solver. OR-Tools has an object-oriented style API, it basically works with solver, constraint and variable objects, the statement of constraints happens with function calls. In order to be able to state the constraints in a more declarative way, we created a functional style wrapper API for the solver. The other advantage of this wrapper is that the solver will be easily changeable for our algorithm as only these formulas should be written for the new solver. The wrapper is written in the Xtend [22] functional language.

Statement of Mathematical and Logical Formulas

For the description of our problems we need variables. We use integer variables as OR-Tools does not support real variables, fortunately integer numbers perfectly fit our requirements. We can have bounded variables, it can take its value from a finite interval, and we can have enumerated variables which can take its value from an enumerated list of numbers.

```
def static bounded(String name, long lower, long upper, Solver solver){
        return solver.makeIntVar(lower, upper, name);
}

def static enumerated(String name, long[] values, Solver solver){
        return solver.makeIntVar(values, name);
}
```

We will need a contraint we can express that the sum of variables can not exceed the value of a scalar (used to express (4.6), (4.23)).

```
def static SUM_LE(IntVar[] row, int scalar){
       return solver.makeSumLessOrEqual(row, scalar);
}
```

We will need a constraint in order to express that the sum of variables should be equal with the value of a scalar (used to express (4.7), (4.24)).

```
def static SUM_EQ(IntVar[] row, int scalar){
       return solver.makeSumEquality(row, scalar);
}
```

We will need a constraint in order to express that the sum of variables should be greater than or equal with the value of a scalar (used to express the preconditions for (4.8), (4.25)).

```
def static SUM_GE(IntVar[] row, int scalar){
       return solver.makeSumGreaterOrEqual(row, scalar);
}
```

We will need to express that if a constraint is satisfied then a postcondition will be true. The operator $\Rightarrow$ will be used for that. Unfortuantely, OR-Tools does not offer support for that directly. Therefore we have to reduce this problem to lower level mathematical formulas. Fortunately, the logical value (satisfied or not) of the constraints can be used. Mark with $a \in \{0,1\}$ variable whether the constraint as a precondition is statisfied or not. Mark with $b \in \{0,1\}$ variable whether the postcondition can be true or not. We use the $a \leq b \Longrightarrow a - b \leq 0$ formula to expess our requirements.

a	b	$a-b \leq 0$
0	0	1
0	1	1
1	0	0
1	1	1

As it can be seen, if the precondition is not satisfied ($a = 0$) then the postcondition can be true or false. This means we allow the satisfaction of the postcondition by other precondition constraints as well. On the other hand if the precondition is satisfied ($a = 1$) then the postcondition will be neccessarily true (used to express the postconditions for (4.8), (4.25)).

```
// => operator
def static operator_doubleArrow(Constraint c, Pair<IntVar, Integer> pair){
       return solver.makeLessOrEqual(c.^var, solver.makeEquality(pair.key, pair.value.intValue).^var);
}
```

Furthermore we will require the minimization of the sum of variables (used to express (4.9), (4.26)).

```
def static SUM_MIN(IntVar[] row){
       var IntVar sumOfRow = solver.makeSum(row).^var()
       return solver.makeMinimize(sumOfRow, 1)
}
```

Finally the constraints can be posted for the solver with the >> operator.

```
// >> operator, post constraint
def static operator_doubleGreaterThan(Solver s, Constraint c){
       solver.addConstraint(c);
}
```

5.2 Case Study: Evaluation of the SwitchSensor Query

In this section we will discuss a case study of the allocation optimization for the *Switch-Sensor* query. We will show in details what happens in the background of the allocation optimization.

5.2.1 The Query

We have to write the query in the pattern language of IncQuery-D as the first step. The syntax is the *IQPL* graph pattern language [24].

```
vocabulary <railway.rdf>

base <http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#>

pattern switchSensor(aSwitch) = {
       Switch(aSwitch);
       neg find hasSensor(aSwitch);
}

pattern hasSensor(Trackelement) = {
       TrackElement_sensor(Trackelement, Target);
}
```

We have to define the metamodel for the query. The metamodel is defined in RDF format [18]. The *Railway metamodel* is publicly available [15].

```
vocabulary <railway.rdf>
```

The main pattern is the *switchSensor* pattern. The result contains those *Switch* graph node elements in the model which do not satisfy the *hasSensor* pattern. The *hasSensor* pattern checks whether a *Switch* node has a *TrackElement_sensor* outgoing edge or not. This query looks for missing edges of *Switch* nodes. As a simplification we do not specify that the endpoint of the *TrackElement_sensor* outgoing edge should be *Sensor* typed as this comes from the structure of the metamodel.

```
pattern switchSensor(aSwitch) = {
       Switch(aSwitch);
       neg find hasSensor(aSwitch);
}

pattern hasSensor(Trackelement) = {
       TrackElement_sensor(Trackelement, Target);
}
```

After we defined the query, a corresponding Rete recipe (see Section 3.2.1) will be generated. The constructed Rete network can be seen in Figure 5.1. We get two Input nodes as we used two different types in the query. As we look for those Switch nodes which do not have *TrackElement_sensor* edges, we use an Antijoin node. However, we only need the Switch end point of the edge so a Trimmer node will project to the first attribute prior to the Antijoin operation.

After the Rete network is ready, the Rete nodes can be assigned to processes. In this layout we have four Memory nodes and one Non-Memory node. Therefore we will have four processes, each Memory node goes to one separate. The one Trimmer node will be placed to the same process as its parent.

The other important task is the prediction of Rete node, process memory consumption and the inter-process communication intensity. We will use the heuristic methods introduced in Section 3.2.4.

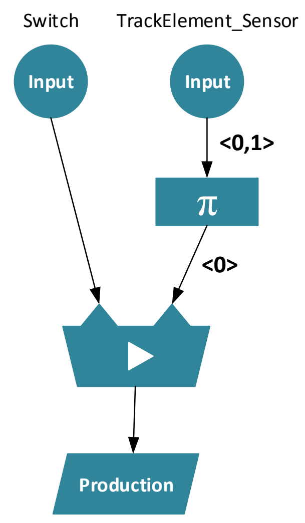


Figure 5.1. Rete Recipe of the SwitchSensor Query.

Prior to the allocation we gather statistics about the cardinality of different node and edge types from the model containing databases. Currently there are 2604750 *TrackElement_sensor* edges in the model and there are 97442 *Switch* nodes.

From the Input nodes all tuples will be propagated. The Trimmer node will remove one of the two attributes. As it also checks for duplicates (it means that those *Switch* nodes has more then one *TrackElement_sensor* edges) it predicts that 10% of the tuples (considering sparse graphs) will be removed. Finally the Antijoin node predicts that 10% of *Switch* nodes will not have outgoing edges.

The estimated memory consumption of Rete nodes will be calculated from the incoming amount of normalized tuples. The memory consumption of a process will be the memory consumption of the contained Memory node with the 40% spare space.

The intensity of inter-process communication can be determined after we ignore the in-process communication edges.

The estimated values calculated by the algorithm are shown in Figure 5.2.

The next step will be the allocation of processes to machines. This will be discussed in regards of both communication and cost minimization.

5.2.2 Communication Minimization

The first allocation of the query will be optimized for minimal network communication. In order to be able to allocate the query, we need an inventory model with the capacity of the available machines and with their communication overhead multipliers. We created two virtual machines with 3 GB memory capacity. The machines could communicate on a local network, so the overhead between machines was predicted to be 3 times greater than the overhead of the communication on the local interface of a machine. This layout can is shown in Figure 5.3.

Computation of the Optimal Allocation

According to the constructed Rete network and the given *Inventory* model, the inputs of the algorithm will be the following (the number of the processes can be seen in Figure 5.2 and the order of the machines is *vm0, vm1*):

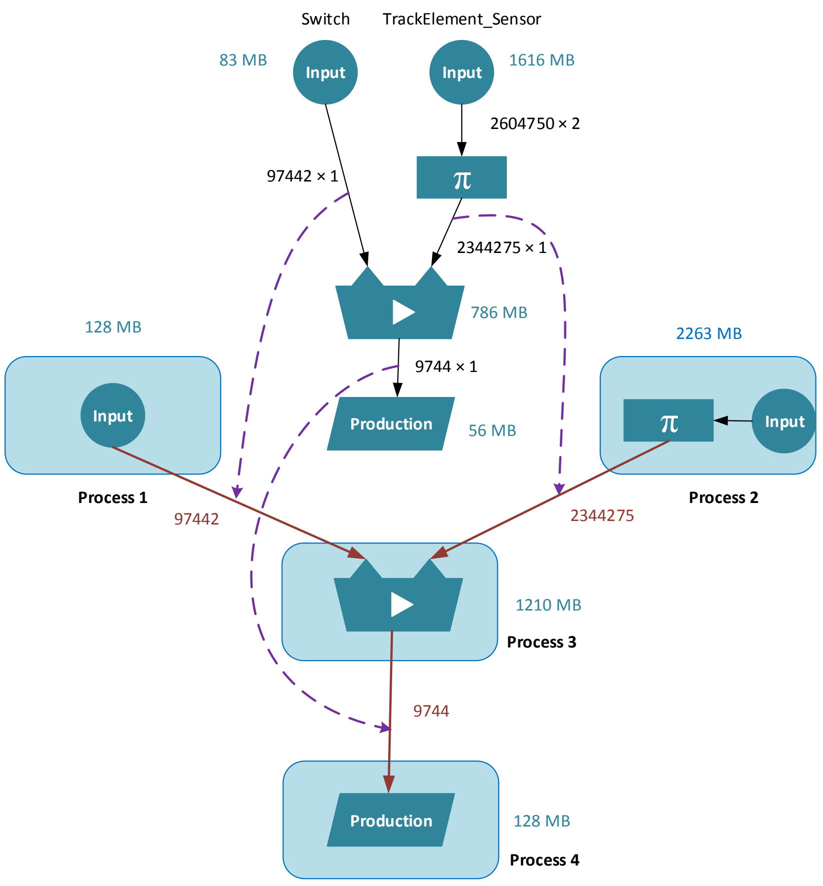


Figure 5.2. Rete Node Allocation of the SwitchSensor Query.

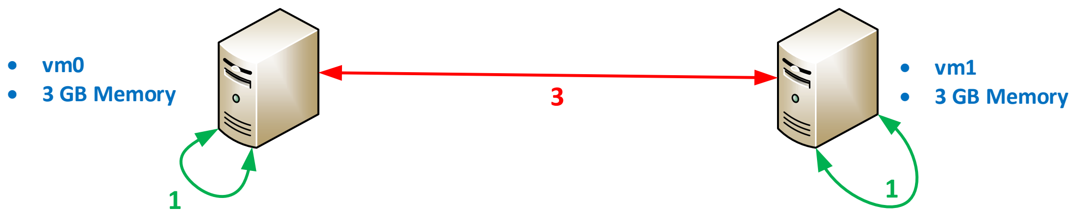


Figure 5.3. Inventory for the SwitchSensor Query Communication Minimization.

$$S = (128, 2263, 1210, 128) \tag{5.1}$$

$$C = (3072, 3072) \tag{5.2}$$

$$E_{4,4} = \begin{bmatrix} 0 & 0 & 97442 & 0 \\ 0 & 0 & 2344275 & 0 \\ 0 & 0 & 0 & 9744 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.3}$$

$$O_{2,2} = \begin{bmatrix} 1 & 3 \\ 3 & 1 \end{bmatrix} \tag{5.4}$$

We can create the constraint set according to the input. The memory usage constraints will be the following:

$$\begin{aligned}128 \cdot x_{1,1} + 2263 \cdot x_{1,2} + 1210 \cdot x_{1,3} + 128 \cdot x_{1,4} \leq 3072\\128 \cdot x_{2,1} + 2263 \cdot x_{2,2} + 1210 \cdot x_{2,3} + 128 \cdot x_{2,4} \leq 3072\end{aligned} \tag{5.5}$$

The process placement costraints are as follows:

$$\begin{aligned}x_{1,1} + x_{2,1} = 1\\x_{1,2} + x_{2,2} = 1\\x_{1,3} + x_{2,3} = 1\\x_{1,4} + x_{2,4} = 1\end{aligned} \tag{5.6}$$

The wights of communication edges will be calculated with the formulas below. Only non-zero communication edges should be taken into account as multiplying zero by anything will be zero. These constraints determine the cost of the communication edges if their endpoints are placed to different machines.

For the edge between 1st and 3rd processes:

$$\begin{aligned}x_{1,1} + x_{1,3} \geq 2 \rightarrow w_{1,3} = 97442 \cdot 1\\x_{1,1} + x_{2,3} \geq 2 \rightarrow w_{1,3} = 97442 \cdot 3\\x_{2,1} + x_{1,3} \geq 2 \rightarrow w_{1,3} = 97442 \cdot 3\\x_{2,1} + x_{2,3} \geq 2 \rightarrow w_{1,3} = 97442 \cdot 1\end{aligned} \tag{5.7}$$

For the edge between 2nd and 3rd processes:

$$\begin{aligned}x_{1,2} + x_{1,3} \geq 2 \rightarrow w_{2,3} = 2344272 \cdot 1\\x_{1,2} + x_{2,3} \geq 2 \rightarrow w_{2,3} = 2344272 \cdot 3\\x_{2,2} + x_{1,3} \geq 2 \rightarrow w_{2,3} = 2344272 \cdot 3\\x_{2,2} + x_{2,3} \geq 2 \rightarrow w_{2,3} = 2344272 \cdot 1\end{aligned} \tag{5.8}$$

For the edge between 3rd and 4th processes:

$$\begin{aligned}x_{1,3} + x_{1,4} \geq 2 \rightarrow w_{3,4} = 9744 \cdot 1\\x_{1,3} + x_{2,4} \geq 2 \rightarrow w_{3,4} = 9744 \cdot 3\\x_{2,3} + x_{1,4} \geq 2 \rightarrow w_{3,4} = 9744 \cdot 3\\x_{2,3} + x_{2,4} \geq 2 \rightarrow w_{3,4} = 9744 \cdot 1\end{aligned} \tag{5.9}$$

The objective function will be (as we can ignore the edges with 0 weight):

$$\text{weight} = \min \{w_{1,3} + w_{2,3} + w_{3,4}\} \tag{5.10}$$

The optimizer algorithm found that the objective function had optimum with the following variable values:

$x_{1,1} = 0$	$x_{1,2} = 1$	$x_{1,3} = 0$	$x_{1,4} = 0$
$x_{2,1} = 1$	$x_{2,2} = 0$	$x_{2,3} = 1$	$x_{2,4} = 1$

In that case

$$\begin{aligned}
x_{2,1} + x_{2,3} &\geq 2 \rightarrow w_{1,3} = 97442 \\
x_{1,2} + x_{2,3} &\geq 2 \rightarrow w_{2,3} = 7032816 \\
x_{2,3} + x_{2,4} &\geq 2 \rightarrow w_{2,3} = 9744
\end{aligned} \tag{5.11}$$

will be true so weight $= 97442 + 7032816 + 9744 = 7140002$.

This means that 1st, 3rd and 4th processes will be started on the *vm1* machine, while 2nd process will be started on the *vm0* machine.

The allocation optimization procedure is continued in the cost dimension in order to find the cheapest solution among the solutions with the least network communication. However, this time we have to use both machines as the processes exceed one's capacity. Therefore the cost optimization is trivial in that case.

The optimal allocation determined by the optimizer is illustrated in Figure 5.4. We can think of it as a visual interpretation of the architecture description model.

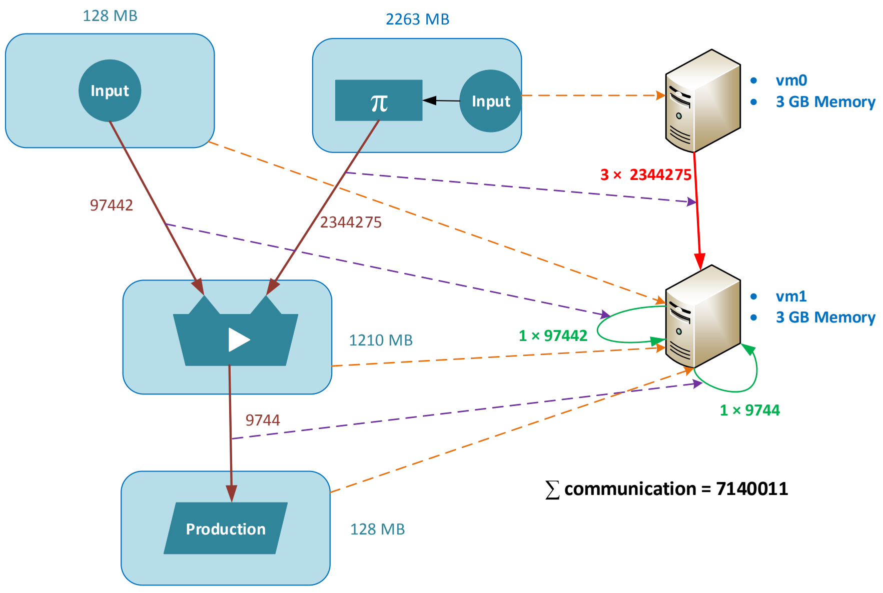


Figure 5.4. Process Allocation of the SwitchSensor Query.

Running and Monitoring the Query

After the allocation has been computed and the architecture description is ready, we can deploy the system and the query can be started. We will use the monitoring dashboard to inspect the runtime behaviour of the query. We will check the utilization of resources, the memory consumption and the communication intensity of Rete nodes and processes in order to justify that the allocation is working as intended.

In Figures 5.5, 5.6, 5.7 we can see the utilization of machines and the resource usage of each JVM. We can observe that the memory usage of JVMs was estimated very well. It is also a good sign that the garbage collector was not run frequently. The memories of the machines are also utilized well, though the memory of *vm0* is a little overused.

We can also note that with the good allocation, IncQuery-D was able to run a query on two machines that could not be run only on one of them.

We can check the consumed memory and communication intensity of the *Rete nodes* in Figure 5.8. We can see that the traffic generated by the *Input nodes* could be estimated

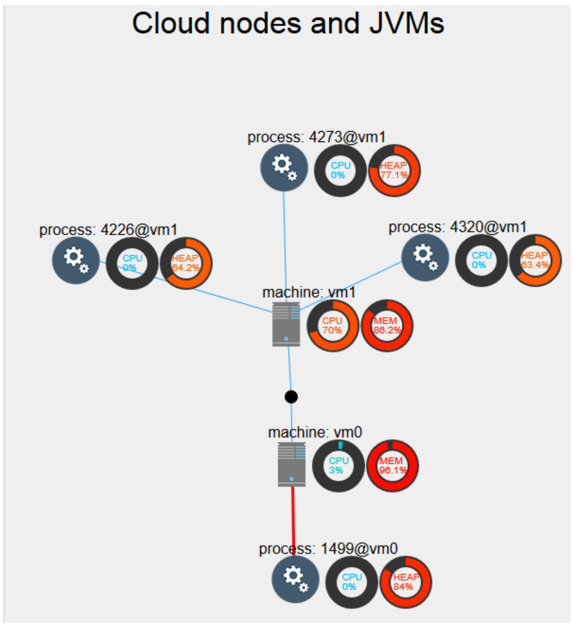


Figure 5.5. SwitchSensor Query Infrastructure on the Dashboard.

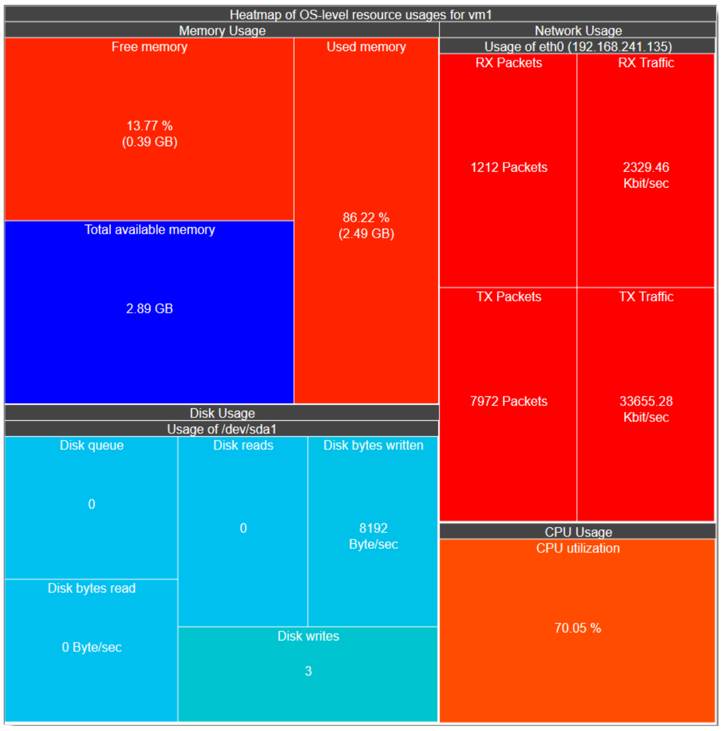


Figure 5.6. Machine Heatmap for VM1 on the Dashboard.

precisely. We can also see incremental changes to the *TrackElement_Sensor* edges, as 10 new edges were added. The *Trimmer node* estimated that 10% of elements will be duplicates, therefore 2344275 tuples will be propagated. We can see that this prediction was quite close to the reality. The *Antijoin node* expects to propagate 10% of its left parent. However, we can see much more tuples in the figure. This is because the tuples from the left parent arrived first and all of them were propagated as none of those had a matching pair among the tuples of the right parent (as it was empty). When the tuples from the right parent arrived, the node found a matching pair for 87000 *Switch* tuples, therefore

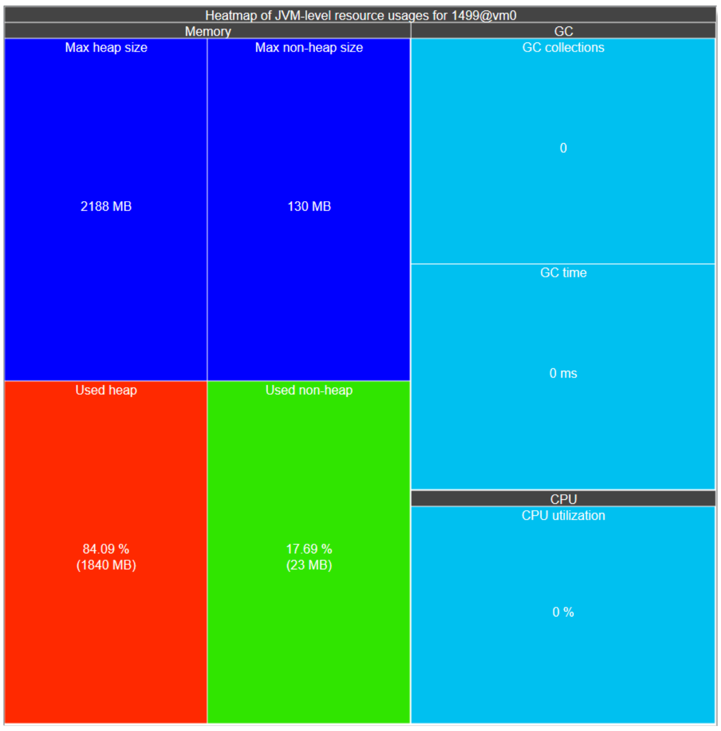


Figure 5.7. JVM Heatmap on the Dashboard.

those were propagated as negative changes. However, the 10% is a good prediction for the steady-state of the system.

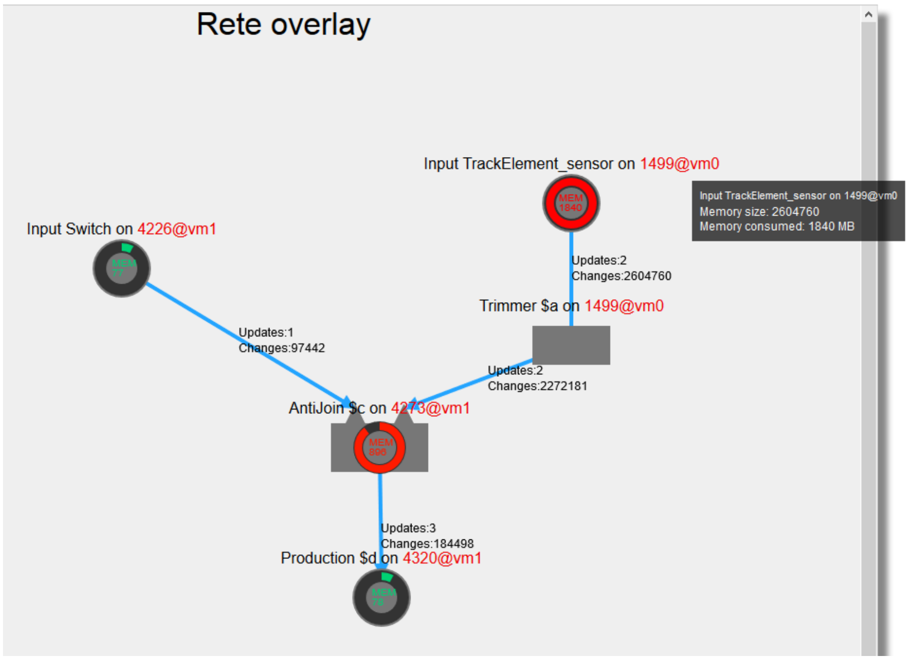


Figure 5.8. Monitored Rete Network of SwitchSensor Query.

5.2.3 Cost Minimization

The second allocation of the query will be optimized for minimal cost.

This time we will use a different *Inventory* model. We will use two cheap, 10$ machines with low capacities and one expensive 1000$ machine with very good parameters. The layout of the infrastructure can be seen in Figure 5.9.

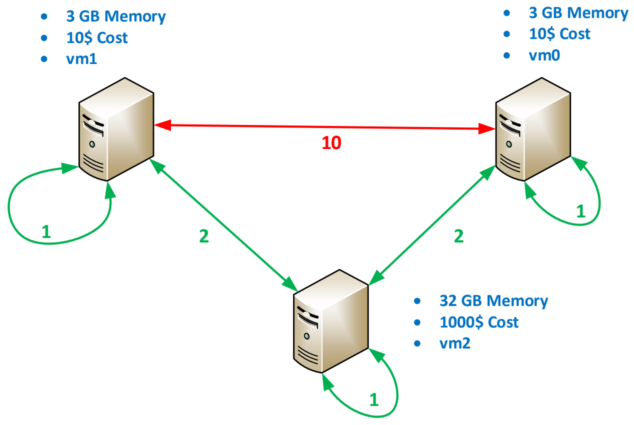


Figure 5.9. Inventory for the SwitchSensor Query Cost Minimization.

Computation of the Optimal Allocation

The inputs of the algorithm will be the following (the order of machines is *vm0, vm1, vm2*):

$$S = (128, 2263, 1210, 128) \tag{5.12}$$

$$C = (3072, 3072, 32768) \tag{5.13}$$

$$B = (10, 10, 1000) \tag{5.14}$$

The memory usage constraints will be the following:

$$\begin{aligned} 128 \cdot x_{1,1} + 2263 \cdot x_{1,2} + 1210 \cdot x_{1,3} + 128 \cdot x_{1,4} &\leq 3072 \\ 128 \cdot x_{2,1} + 2263 \cdot x_{2,2} + 1210 \cdot x_{2,3} + 128 \cdot x_{2,4} &\leq 3072 \\ 128 \cdot x_{3,1} + 2263 \cdot x_{3,2} + 1210 \cdot x_{3,3} + 128 \cdot x_{3,4} &\leq 32768 \end{aligned} \tag{5.15}$$

The process placement constraints are the following:

$$\begin{aligned} x_{1,1} + x_{2,1} + x_{3,1} &= 1 \\ x_{1,2} + x_{2,2} + x_{3,2} &= 1 \\ x_{1,3} + x_{2,3} + x_{3,3} &= 1 \\ x_{1,4} + x_{2,4} + x_{3,4} &= 1 \end{aligned} \tag{5.16}$$

The price of the machines will be determined with the following formulas:

$$\begin{aligned} x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} \geq 1 &\rightarrow w_1 = 10 \\ x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} \geq 1 &\rightarrow w_2 = 10 \\ x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} \geq 1 &\rightarrow w_3 = 1000 \end{aligned} \tag{5.17}$$

The objective function will be:

$$\text{cost} = \min\{w_1 + w_2 + w_3\} \tag{5.18}$$

The optimizer found an optimal solution with the following values:

$x_{1,1} = 1$	$x_{1,2} = 1$	$x_{1,3} = 0$	$x_{1,4} = 1$
$x_{2,1} = 0$	$x_{2,2} = 0$	$x_{2,3} = 1$	$x_{2,4} = 0$
$x_{3,1} = 0$	$x_{3,2} = 0$	$x_{3,3} = 0$	$x_{3,4} = 0$

In that case

$$\begin{aligned} 0+1+0+0 \geq 1 \rightarrow w_1 = 10 \\ 1+0+1+1 \geq 1 \rightarrow w_2 = 10 \end{aligned} \tag{5.19}$$

will be true and

$$0+0+0+0 \geq 1 \rightarrow w_3 = 1000 \tag{5.20}$$

will not be true, the cost will be 20.
So the expensive machine, *vm2* will not be used.

However, the algorithm will terminate as it does multi-dimensional optimization, therefore we will further optimize the solution in the communication dimension. On the other hand, we know that the optimal allocation will not contain *vm2* machine, because we found a solution with only 20 cost.

We will invoke the *Communication Optimization* algorithm with the following inputs (S and C vectors will be the same):

$$E_{4,4} = \begin{bmatrix} 0 & 0 & 97442 & 0 \\ 0 & 0 & 2344275 & 0 \\ 0 & 0 & 0 & 9744 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{5.21}$$

$$O_{2,2} = \begin{bmatrix} 1 & 10 \\ 10 & 1 \end{bmatrix} \tag{5.22}$$

The output of the algorithm is:

$x_{1,1} = 0$	$x_{1,2} = 1$	$x_{1,3} = 0$	$x_{1,4} = 0$
$x_{2,1} = 1$	$x_{2,2} = 0$	$x_{2,3} = 1$	$x_{2,4} = 1$

According to that solution, the provided *Architecture* of the query is shown in Figure 5.10.

This example shows that optimization for cost makes sense when we prefer using cheap, off-the-shelf hardware instead of expensive server machines.

5.3 IncQuery-D IDE

In this section, we will show the IncQuery-D IDE and its functionalities in regards of the allocation. The purpose is to illustrate the workflow of a query, that was introduced in Section 3.2, with an example. We will see the main stages and artifacts of the workflow from the user's point of view and will give an insight about how the workflow is implemented in practice.

The IncQuery-D IDE is implemented on the Eclipse Platform [9].

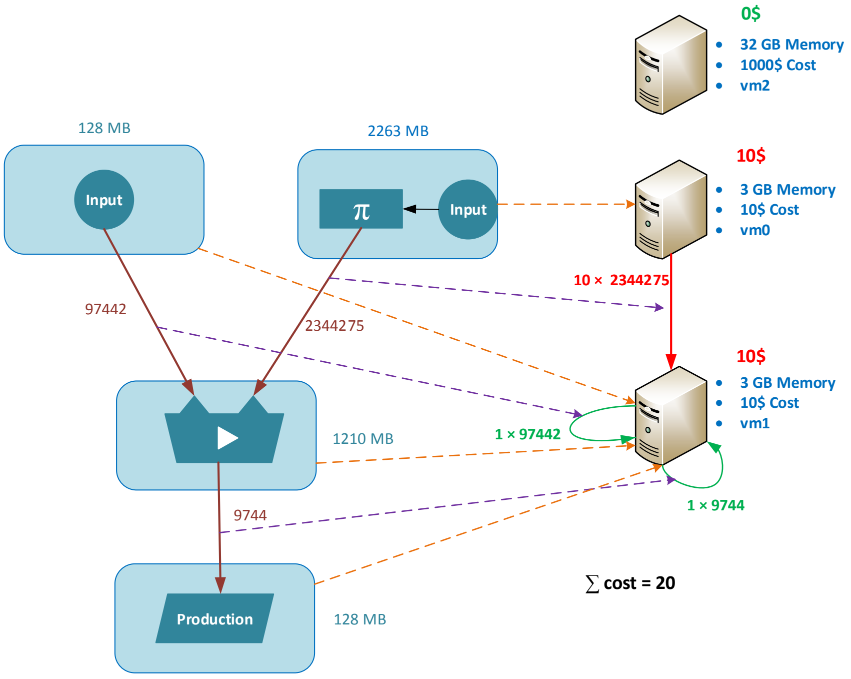
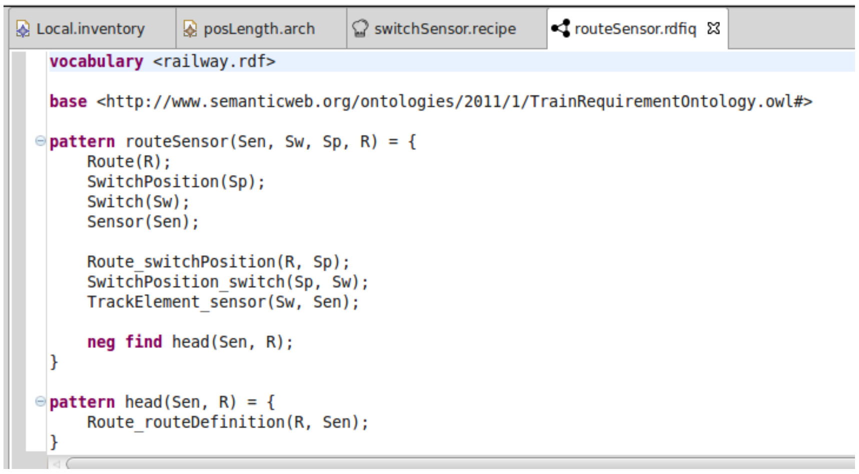


Figure 5.10. Process Allocation of the SwitchSensor Query.

The RouteSensor Query This time we will use a more complex query as an example, the *RouteSensor* query. The query defined in the editor is shown in Figure 5.11. The editor provides syntax highlight and content assist for query editing.



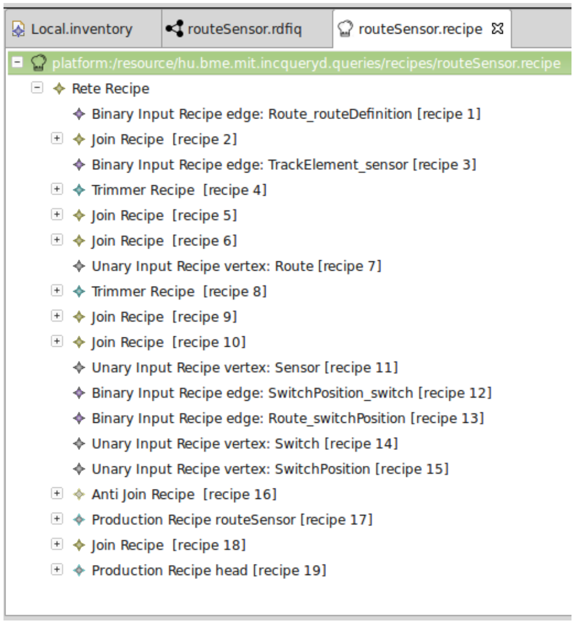
```
vocabulary <railway.rdf>

base <http://www.semanticweb.org/ontologies/2011/1/TrainRequirementOntology.owl#>

pattern routeSensor(Sen, Sw, Sp, R) = {
    Route(R);
    SwitchPosition(Sp);
    Switch(Sw);
    Sensor(Sen);

    Route_switchPosition(R, Sp);
    SwitchPosition_switch(Sp, Sw);
    TrackElement_sensor(Sw, Sen);

    neg find head(Sen, R);
}

pattern head(Sen, R) = {
    Route_routeDefinition(R, Sen);
}
```

Figure 5.11. The RouteSensor Query in the Editor.

Once the query is written and compiled we get the generated *Rete recipe* defining the *Rete network* for the query. The *recipe* file (with *.recipe* extension) and the *Rete nodes* generated for the query are shown in Figure 5.12. This abstract model containing file of the query will be one input for the *Allocation Optimizer*.

5.3.1 The Allocation Optimizer Subsystem

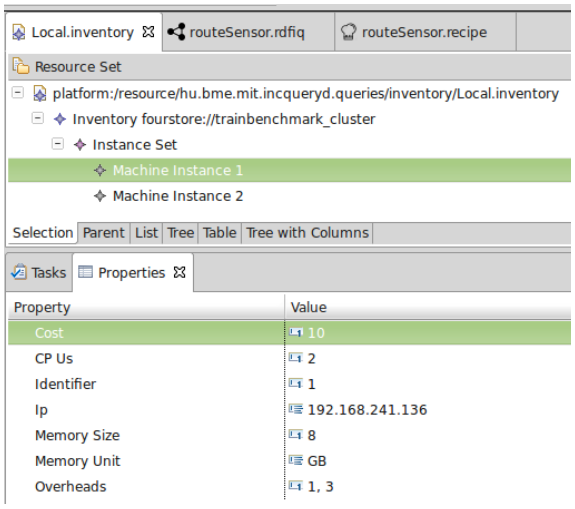
One other important input for the *Allocation Optimizer* is the *inventory* description model. This model is contained in a file with *.inventory* extension.


Local.inventory | routeSensor.rdfiq | routeSensor.recipe

platform:/resource/hu.bme.mit.incqueryd.queries/recipes/routeSensor.recipe
- Rete Recipe
  - Binary Input Recipe edge: Route_routeDefinition [recipe 1]
  - Join Recipe [recipe 2]
  - Binary Input Recipe edge: TrackElement_sensor [recipe 3]
  - Trimmer Recipe [recipe 4]
  - Join Recipe [recipe 5]
  - Join Recipe [recipe 6]
  - Unary Input Recipe vertex: Route [recipe 7]
  - Trimmer Recipe [recipe 8]
  - Join Recipe [recipe 9]
  - Join Recipe [recipe 10]
  - Unary Input Recipe vertex: Sensor [recipe 11]
  - Binary Input Recipe edge: SwitchPosition_switch [recipe 12]
  - Binary Input Recipe edge: Route_switchPosition [recipe 13]
  - Unary Input Recipe vertex: Switch [recipe 14]
  - Unary Input Recipe vertex: SwitchPosition [recipe 15]
  - Anti Join Recipe [recipe 16]
  - Production Recipe routeSensor [recipe 17]
  - Join Recipe [recipe 18]
  - Production Recipe head [recipe 19]

Figure 5.12. Rete Recipe of the RouteSensor Query.

We have created a model (with *Local.inventory* name) in the *Inventory editor* of the IDE which is shown in Figure 5.13. We have to specify a connection string for the database which stores the model. In this example we use 4store clustered databases with the *fourstore://trainbenchmark_cluster* connection string. Two virtual machines will run the query in this example, both have 8 GBs of memory space. We also defined the communication overheads between the machines with a matrix-style notation (for each machine in order).


Local.inventory | routeSensor.rdfiq | routeSensor.recipe

Resource Set
- platform:/resource/hu.bme.mit.incqueryd.queries/inventory/Local.inventory
  - Inventory fourstore://trainbenchmark_cluster
    - Instance Set
      - Machine Instance 1
      - Machine Instance 2

Selection | Parent | List | Tree | Table | Tree with Columns

Tasks | Properties

Property	Value
Cost	10
CP Us	2
Identifier	1
Ip	192.168.241.136
Memory Size	8
Memory Unit	GB
Overheads	1, 3

Figure 5.13. Creating Inventory for the RouteSensor Query.

Once the *Rete recipe*, the *inventory* are created and the database has loaded the data, we have all inputs for the allocation and the user can create an allocation for the query from the IDE. In order to do that, the user has to right click on the *recipe* file and choose the optimized allocation from the context menu as it is shown in Figure 5.14.

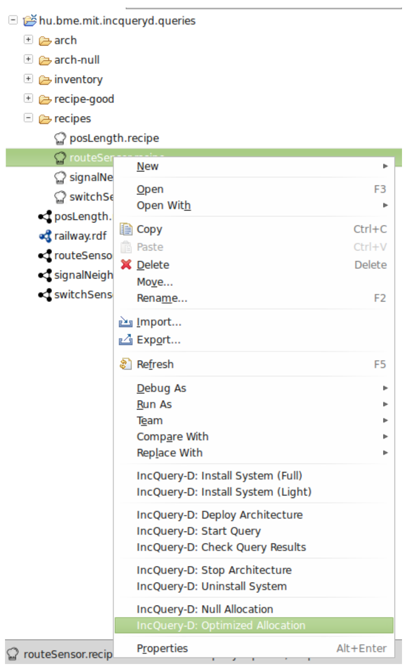


Figure 5.14. Optimized Allocation for the RouteSensor Query.

However, the allocator does not know which *inventory* file it should use for the allocation. Therefore it offers to choose one from the file system of the local machine. The allocator does not know either which objective function it should primarily optimize for, therefore the user has to specify that, too.

Once all of the inputs were specified by the user, the allocation optimizer can start its work.

It queries the database in order to determine how many data tuples the queries will run against. After that it creates processes for the *Rete nodes* and calculates the heuristical values for process memory consumption and inter-process communication. Finally, it will assign the processes to machines optimally.

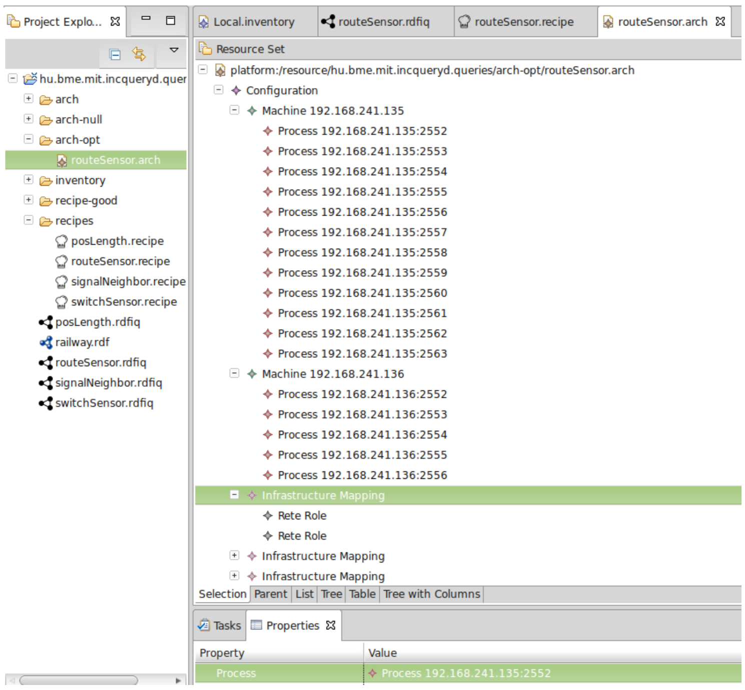
The output of the allocation will be the optimized *architecture* model. In this *Route-Sensor* query example we optimized for the network communication. The generated *architecture* file in the IDE is shown in Figure 5.15.

It contains the used machines with their IP addresses. The machines contain their processes with their port numbers and estimated memory consumption. The processes contained by one particular machine will be started on that machine.

The model also contains *InfrastructureMapping* elements. The role of this element is the mapping of *Rete nodes* to processes. In Figure 5.15 we can see that the process *192.168.241.135:2552* (identified by IP and port number) will contain two *Rete nodes.*

5.3.2 IncQuery-D IDE Lifecycle Operations

In this section we will see how we can run the query and manage its lifecycle if we have the *architecture* description prepared for it. The lifecycle operations for the query can be reached from the context menu of an *architecture* file.



```
Project Explo...
hu.bme.mit.incqueryd.quer
  arch
  arch-null
  arch-opt
    routeSensor.arch
  inventory
  recipe-good
  recipes
    posLength.recipe
    routeSensor.recipe
    signalNeighbor.recipe
    switchSensor.recipe
  posLength.rdfiq
  railway.rdf
  routeSensor.rdfiq
  signalNeighbor.rdfiq
  switchSensor.rdfiq
```

```
Local.inventory | routeSensor.rdfiq | routeSensor.recipe | routeSensor.arch
Resource Set
platform:/resource/hu.bme.mit.incqueryd.queries/arch-opt/routeSensor.arch
  Configuration
    Machine 192.168.241.135
      Process 192.168.241.135:2552
      Process 192.168.241.135:2553
      Process 192.168.241.135:2554
      Process 192.168.241.135:2555
      Process 192.168.241.135:2556
      Process 192.168.241.135:2557
      Process 192.168.241.135:2558
      Process 192.168.241.135:2559
      Process 192.168.241.135:2560
      Process 192.168.241.135:2561
      Process 192.168.241.135:2562
      Process 192.168.241.135:2563
    Machine 192.168.241.136
      Process 192.168.241.136:2552
      Process 192.168.241.136:2553
      Process 192.168.241.136:2554
      Process 192.168.241.136:2555
      Process 192.168.241.136:2556
    Infrastructure Mapping
      Rete Role
      Rete Role
    Infrastructure Mapping
    Infrastructure Mapping
Selection | Parent | List | Tree | Table | Tree with Columns

Tasks | Properties
Property    Value
Process     Process 192.168.241.135:2552
```

Figure 5.15. Optimized Architecture Model for the RouteSensor Query.

Installing the IncQuery-D System The first step is to install the IncQuery-D system to the computers which will run the query. A script will install all the binaries, including the dependencies, and other resources needed by the IncQuery-D system to the host machines used in the particular *architecture* description. The installation has two variants, *full* and *light*. The difference between them is not relevant in the context of this report.

The *coordinator* with its binaries will also be installed to one machine.

The installation will also include the monitoring components. The *operating system monitoring agents* will be installed to all machines while the *monitoring server* will be installed to one machine.

In this example the *coordinator* and the *monitoring server* will run on the machine with the *192.168.241.135* IP address. These information are also described in the *architecure model*.

This step can be skipped if the system is already installed to the hosts.

Deploying the Query Deployment is a preparation for the query processing. In this phase we start the *Rete node* containing processes of the *architecture* file (these will be the *Akka microkernels*) on the appropriate hosts with their estimated memory sizes and their ports used for communication reserved for them.

We start the *coordinator* process as well, it also reserves its ports used for communication.

Finally the monitoring components will be started, the *agents* on each host contained in the *architecture* and the *monitoring server* on the assigned host.

Starting the Query In this phase, we give a command for the *coordinator* and it will start the processing of the query. The *coordinator* requires the prepared *architecture* description for this operation. The *coordinator* will create the *Rete actors* for the *Rete nodes* in the appropriate processes according to the *architecture* description.

The *Input nodes* will load the appropriate data from the database and the results will be propagated through the *Rete network*.

In this phase, the *Rete nodes* can already be observed on the *monitoring dashboard*. Figure 5.16 shows the network with the propagation for the *RouteSensor* query. We can observe that 21107 tuples reached the *Production node*. These tuples are the recent results of the query.

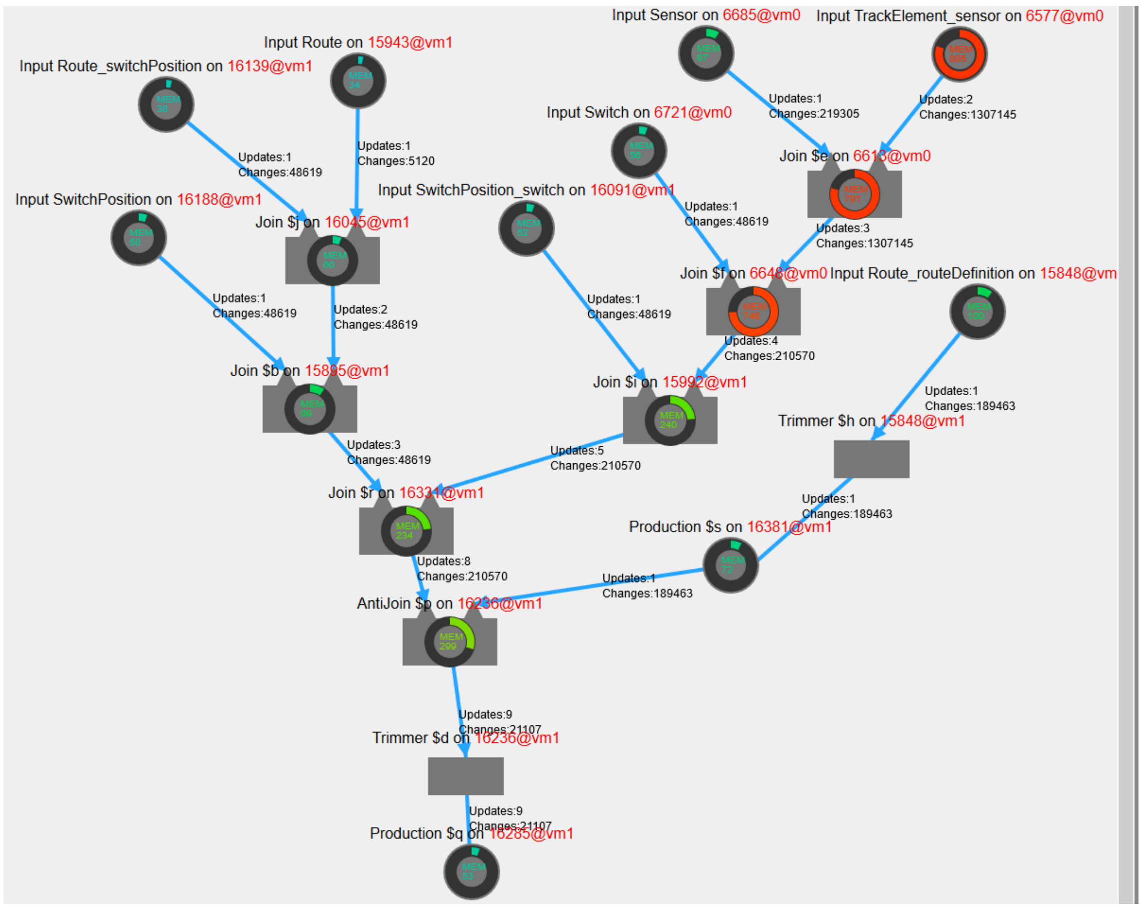


Figure 5.16. Rete Layout for the RouteSensor Query.

Check the Query Results With this command the *coordinator* gets the results from the *Production node*. The results will be propagated to the *monitoring server* in order to be able to visualize on the *monitoring dashboard* as we have seen in Section 3.2.3.

Stop the Query This command terminates the query by stopping all the running processes mentioned previously.

Uninstall the IncQuery-D System This command removes all the aforementioned binaries and resources used by the IncQuery-D system from the used machines.

Chapter 6

Evaluation of the Optimization

In this chapter we wish to evaluate the performance of the optimally allocated system with benchmark measurements. We conducted multiple measurements on models of different sizes and in various environments.

6.1 Purpose of the Benchmark

The purpose of the benchmark is to measure the impact of the allocation on the performance of the system. In order to achieve this, we utilized the Train Benchmark framework, introduced in Section 2.1. The Train Benchmark has been designed with the specific goal of evaluating the performance of graph pattern matching tools. In previous work [35], the Train Benchmark has been extended to accomodate performance measurements in a distributed environment. We build on these results to evaluate the differences between various allocation strategies.

Experiments To assess the impact of optimization, we designed two experiments. In the *first experiment*, the impact of memory allocation optimization is assessed by comparing an optimized configuration to naïve setup that a) uses the default heap size for the Java Virtual Machines, and b) assigns close to the maximum RAM available to the JVMs. In the *second experiment*, we assess the impact of network traffic optimization by comparing the optimized configuration to a *counter-optimized* setup that maximizes communication along remote connections. To emphasize the effect of network communication on overall performance, we configured the cloud environment to use lower speed (10 Mbit) connections that simulate a system under load. In both experiments, we follow the Train Benchmark specification in the *model transformation* (*Xform*) scenarios and run the *RouteSensor* query on instance models of increasing sizes, up to 2 million nodes and 11 million edges.

Hardware and software setup The benchmark software configuration consisted of an extended Train Benchmark setup using the 4store (version 1.1.5) database in a clustered environment. We ran three virtual machines (VMs) on a private cloud powered by Apache VCL, each VM was running 64-bit Ubuntu Linux 14.04 on Intel Xeon 2.5GHz CPUs and 8GBs of RAM. We used Oracle 64-bit Java Virtual Machines (version 1.7.0_72), and Akka 2.1.4. We integrated the Monitoring Subsystem into the benchmark environment in order to record telemetry data at each execution stage of the benchmark. The benchmark results were automatically processed by the R scripts provided by the Train Benchmark framework.

6.2 Results and Analysis

For the sake of brewity, we present the most important aspects of the measurements, focusing on 1) *read and first check* and 2) *transformation* phases of the Train Benchmark[1]. The *read and first check* phase includes the combined execution time of loading data into the IncQuery-D system and propagating update tokens corresponding to the initial data to construct the complete Rete network. The *transformation* phase includes the combined execution time of model manipulation sequences that involve propagating a smaller number of update tokens through the Rete network.

6.2.1 Expected Results

As memory allocation primarily affects the JVMs ability to scale up with memory demands, we expect the memory optimization to provide a more efficient configuration for higher instance model sizes. In fact, it is foreseen that default heap size parameterization (which corresponds to $\min(1\text{ GB}, 0.25 \times \text{RAM})$ according the Oracle specification[2]) will not allow the measurements to execute successfully for larger model sizes.

For network traffic optimization, we foresee that the reduced amount of remote communication will yield more efficient execution for the *read and first check* phase, as that phase involves the (comparatively) large amount of updates propagating through the Rete network.

6.2.2 Measurement Results

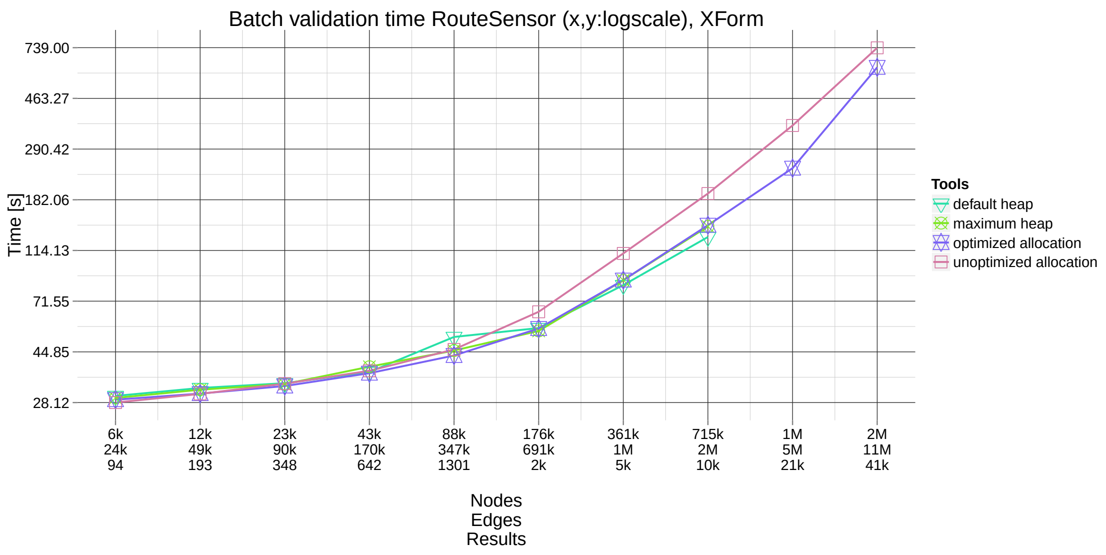


Figure 6.1. Runtime of the read and first check phases on a slow (10 Mbit) network.

Figure 6.1 shows combined measurement data from both experiments for the *read and first check* phase. Note that both axes are scaled logarithmically. The x axis shows model and query result sizes (number of nodes, number of edges, number of result entries from top to bottom, respectively). The y axis shows execution time in seconds. The

[1]The complete measurement data is available from `http://trainbenchmark.inf.mit.bme.hu`

[2]`http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html#par_gc.ergonomics.default_size`

chart corresponds to a post-processed data set taking the minimum of five measurement runs into account, in order to minimize noise stemming from load variations in the cloud environment.

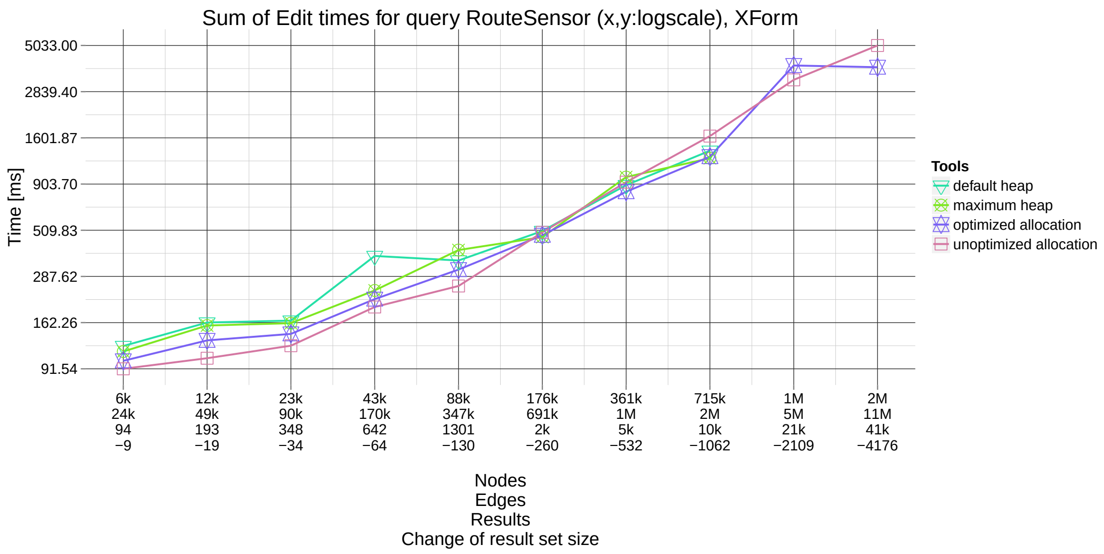


Figure 6.2. Runtime of the transformation phase on a slow (10 Mbit) network.

Figure 6.2 shows combined measurement data from both experiments for the *transformation* phase. Note that both axes are scaled logarithmically, while the y axis now shows execution time in milliseconds, and the bottom labels on the x axis now show *changes* in the query result sets (which correspond to delta tuples received from the production node).

6.2.3 Result analysis

Experiment 1: Memory optimization

As it can be seen in Figures 6.1 and 6.2, both the “default heap size” variant and the “maximum heap size” variants failed to execute successfully for the largest instance model sizes, both executions reporting out of heap space errors in the JVMs running one of the Rete nodes.

These observations are explained by telemetry data recorded during the execution runs. The memory measurements are illustrated in Figures 6.3 and 6.4, which show the system state after the *load and first check* phase is complete for the 1M+5M instance model. As it can be seen on the memory gauges (shown for VM processes in Figure 6.3 and for Rete nodes in Figure 6.4), the memory usage for the memory intensive nodes have been estimated correctly by the heuristics – all the join nodes indicate a memory usage within the target range (orange color) close to 1GBs. As the “default heap size” variant uses 1GB heap limits, the JVM may get into a thrashing state under such loads and cause a timeout during measurement which the Train Benchmark framework registers as a failed run. Similarly, in the case of the “maximum heap size” variant, due to the lack of a reasonable upper limit, the JVMs may interfere with each other’s memory allocations resulting in a runtime error.

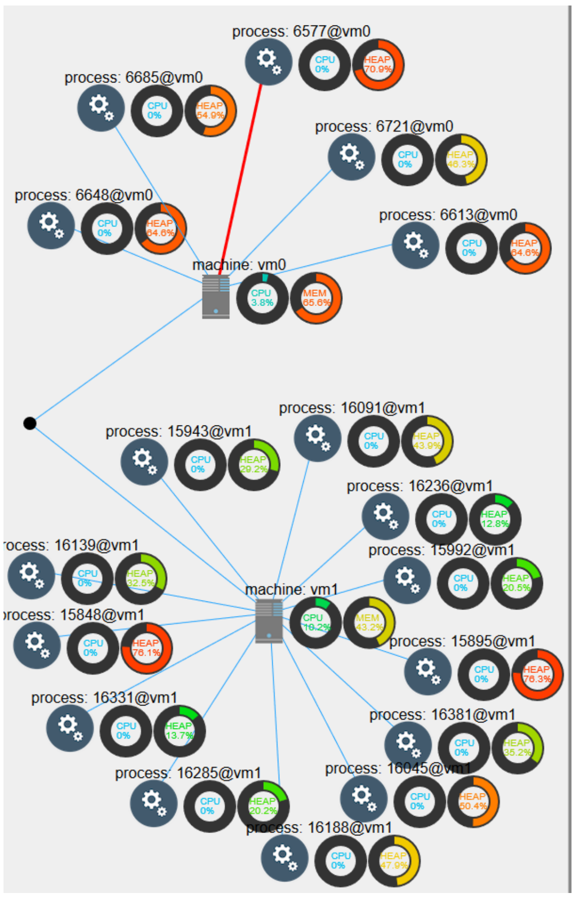


Figure 6.3. RouteSensor measurement infrastructure monitoring.

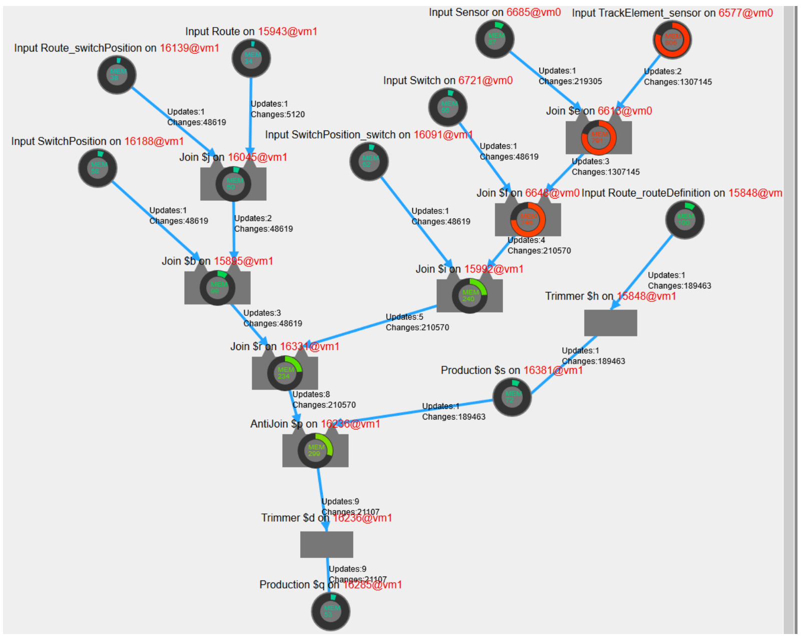


Figure 6.4. RouteSensor measurement Rete network monitoring.

Experiment 2: Network optimization

Figure 6.1 compares the *optimized* variant to a “counter-optimized” (shown as *unoptimized*) as described in Section 6.1. We may observe that while the overall characteristics are similar, the *optimized* variant shows a constant-multiplier advantage, running approximately 15-20% faster than the *unoptimized* variant.

Similar observations may be made by looking at Figure 6.2. As the numbers are comparatively small (which is consistent with the Train Benchmark specification), the *optimized* variants advantage is within the measurement error range.

These observations are explained by telemetry data recorded by the Monitoring Subsystem during the measurement runs. The network traffic measurements are summarized in Figure 6.5, which compares the network traffic volume recorded for the "optimized" and "unoptimized" variants. It can be seen that while the overall volume is practically equivalent, its distribution between local and remote links is characteristically different (i.e. in the "optimized" case, the overall remote volume is approximately 15% lower than in the "unoptimized" case).

	Unoptimized			Optimized		
Traffic [MBytes]	vm0	vm1	vm2	vm0	vm1	vm2
Remote RX+TX	300	349	371	248	280	347
Local RX+TX	14	2	74	24	20	190
SUM Remote	1020			875		
SUM Local	90			234		
Total traffic	1110			1109		

Figure 6.5. Network traffic statistics.

6.3 Threats to Validity

For our experiments, we considered the following internal and external threats to validity.

- Transient and unknown *background load* in the cloud environment. As a countermeasure, we performed *several execution runs* and considered their minima for the result plots.

- We strived to *avoid systematic errors* in the experiments (e.g. incorrect queries or workloads) by *cross-checking all results* with the specification of the Train Benchmark.

- The validity of the analysis and especially the generalizability of the results to real-world workloads has been thoroughly investigated in several academic papers on the Train Benchmark (e.g. [36, 31]). We believe that our measurements are faithful extensions of the Train Benchmark and thus the results of these previous works apply to our contributions as well.

Chapter 7

Conclusion

This chapter summarizes the contributions presented in the report.

7.1 Summary of Contributions

We presented a novel approach for query optimization in the context of a distributed incremental query engine, the IncQuery-D, and introduced the concept of allocation optimization. We created a runtime monitoring subsystem and an allocation optimizer component for the IncQuery-D framework, and we fully integrated our implementation results to the Eclipse-based development tools for IncQuery-D.

7.1.1 Scientific Contributions

Hereby we summarize the scientific contributions of this report.

- We introduced the novel concept of allocation optimization in the context of distributed query optimization.
- We formalized the proposed allocation optimization problems as combinatorial optimization problems.
- We proved the $\mathcal{NP}$-hardness of the formalized problems.

7.1.2 Practical Accomplishments

Hereby we summarize the practical accomplishments of this report.

- We implemented a new runtime monitoring subsystem for IncQuery-D, which collects all the relevant performance metrics of the system. Furthermore, a web based GUI has been created which visualizes all the important monitoring data, the components of the system and the query results.
- We implemented allocation algorithms for the proposed problems, which are based on CSP (Constraint Satisfaction Problem) solving.
- We integrated the allocation optimizer and monitoring components to IncQuery-D runtime, and to the Eclipse-based development environment of IncQuery-D.
- We extended the Train Benchmark environment, by collecting and storing monitoring data of the IncQuery-D system.

7.2 Achieved Results

Overall, we believe that the new IncQuery-D allocation optimizer shows very promising results. We evaluated the effects of allocation optimization on the runtime performance of the IncQuery-D system, and we found that the allocation optimizer has a critical role in allocating memory for the processes.

We could provide efficient memory allocation solutions, where the processes had enough memory to operate while the RAM resources of the computers were not overused. These tasks are critical as a human user can not be realistically expected to perform similar optimization for more complex configurations.

In our initial experiments, we realized that the communication optimization had little effect on the runtime performance of the system in the case when we measured in a private cloud with fast and dedicated Gigabit Ethernet connections for relatively small models and simple queries. However, this is not surprising as sending relatively small amount of data on a fast communication link has no significant overhead compared to the other computations of the system.

However, we found that in an environment with lower speed network communication (or where the network connections are not dedicated to the query engine but are under load from outside the system), communication optimization caused significant performance improvement in the query evaluation. Even for small models the evaluation time could be 20% less than in an unoptimized case. Therefore this way of optimization has key importance in providing scalable query evaluation in environments with worse conditions (e.g. in a public cloud infrastructure).

On the other hand, we believe that in case of really large models and complex queries, communication optimization will cause evident performance improvement even in case of fast communication links. Furthermore, the overall idea behind network traffic optimization can be extended to incorporate other optimization aspects such as CPU load and multiple cores, providing a straightforward direction for future research.

7.3 Limitations and Future Work

The current implementations of the provided solutions has some known limitations:

- The performance of the allocation optimization algorithms is not yet capable of solving problems with large models and complex queries efficiently.
- Our heuristics are based on certain assumptions that are specific to software modeling (e.g. that the graphs have low edge density) that may not always hold.

For future work, we plan to address the following challenges:

- Providing performance improvements for the allocation algorithms, e. g. by trying out alternative solvers or implementing different solution approaches.
- Providing precision improvements for the process memory consumption and network communication heuristics.
- Creating dynamic reconfiguration and reallocation solutions for the IncQuery-D system based on a feedback-loop coming from the runtime monitoring data and this way making the system autonomous.

Acknowledgement

I would like to say huge thanks to my supervisors Gábor Szárnyas, Dr. István Ráth and Dr. Ákos Horváth for their limitless help and enthusiasm. I am also grateful to Dr. Dániel Varró, Benedek Izsó and Dr. Gábor Bergmann for their friendly advice in regards of my work. Furthermore I would like to extend my appreciation to all other colleagues in the Fault Tolerant Systems Research Group who provided numerous valuable suggestions and help.

Last but not least, I would like to express my deep thankfulness to my family and friends for their continuous support and appreciation which helped me all along the way.

Bibliography

[1] 4store. http://4store.org/. Accessed: 2014-10-22.

[2] Akka Framework. http://akka.io/. Accessed: 2014-10-14.

[3] Akka Microkernel. http://doc.akka.io/docs/akka/snapshot/scala/microkernel.html. Accessed: 2014-10-22.

[4] AUTOSAR. http://www.autosar.org/. Accessed: 2014-10-22.

[5] Avahi protocol. http://avahi.org/. Accessed: 2014-10-22.

[6] Boolean Satisfiability Problem. http://en.wikipedia.org/wiki/Boolean_satisfiability_problem. Accessed: 2014-10-10.

[7] Combinatorial Optimization. http://en.wikipedia.org/wiki/Combinatorial_optimization. Accessed: 2014-10-10.

[8] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/. Accessed: 2014-10-14.

[9] Eclipse Platform. https://www.eclipse.org/. Accessed: 2014-10-22.

[10] Google OR-Tools. https://code.google.com/p/or-tools/. Accessed: 2014-10-12.

[11] Knapsack Problem. http://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf. Accessed: 2014-10-11.

[12] Neo Technology, Neo4j. http://neo4j.com/. Accessed: 2014-10-22.

[13] OpenLink Software: Virtuoso Universal Server. http://virtuoso.openlinksw.com/. Accessed: 2014-10-22.

[14] Pareto Efficiency. http://en.wikipedia.org/wiki/Pareto_efficiency. Accessed: 2014-10-14.

[15] Railway RDF Metamodel. https://github.com/FTSRG/trainbenchmark-rdf/blob/master/metamodel/railway.rdf. Accessed: 2014-10-22.

[16] Rasqal RDF Query Library. http://librdf.org/rasqal/. Accessed: 2014-10-22.

[17] RDF/XML. http://www.w3.org/TR/rdf-syntax-grammar/. Accessed: 2014-10-22.

[18] Resource Description Framework. http://www.w3.org/RDF/. Accessed: 2014-10-14.

[19] Sesame: RDF API and Query Engine. http://rdf4j.org/. Accessed: 2014-10-22.

[20] SPARQL Query Language. http://www.w3.org/TR/rdf-sparql-query/. Accessed: 2014-10-22.

[21] Triplestore. `http://en.wikipedia.org/wiki/Triplestore`. Accessed: 2014-10-22.

[22] Xtend. `http://www.eclipse.org/xtend/`. Accessed: 2014-10-19.

[23] YourKit. `http://www.yourkit.com/`. Accessed: 2014-10-22.

[24] Gábor Bergmann, Zoltán Ujhelyi, István Ráth, and Dániel Varró. A Graph Query Language for EMF models. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, volume 6707 of *Lecture Notes in Computer Science*, pages 167–182. Springer, Springer, 2011. Acceptance rate: 27%.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[26] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.

[27] Steve Harris, Nick Lamb, and Nigel Shadbolt. 4store: The design and implementation of a clustered RDF store. In *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*, pages 94–109, 2009.

[28] Dorit S Hochbaum. *Approximation algorithms for NP-hard problems.* PWS Publishing Co., 1996.

[29] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. IncQuery-D: Incremental Graph Search in the Cloud. In *Proceedings of the Workshop on Scalability in Model Driven Engineering*, page 4, Budapest, Hungary, 2013. ACM, ACM.

[30] Benedek Izsó, Gábor Szárnyas, István Ráth, and Dániel Varró. Train Benchmark Technical Report. 2014.

[31] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth, and István Ráth. Towards Precise Metrics for Predicting Graph Query Performance. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 412–431, Silicon Valley, CA, USA, 11/2013 2013. IEEE, IEEE. Acceptance Rate: 23%.

[32] Daniel P Miranker, Rodolfo K Depena, Hyunjoon Jung, Juan F Sequeda, and Carlos Reyna. Diamond: A SPARQL query engine, for linked data based on the rete match. In *Workshop on Artificial Intelligence meets the Web of Data*, 2012.

[33] Mikko Rinne, Esko Nuutila, and Seppo Törmä. INSTANS: High-Performance Event Processing with Standard RDF and SPARQL. In *11th International Semantic Web Conference ISWC 2012*, page 101. Citeseer, 2012.

[34] Markus Scheidgen, Anatolij Zubow, Joachim Fischer, and Thomas H Kolbe. *Automated and transparent model fragmentation for persisting large models.* Springer, 2012.

[35] Gábor Szárnyas. Superscalable Modeling. Master's thesis, Budapest University of Technology and Economics, Budapest, 12/2013 2013.

[36] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. IncQuery-D: A Distributed Incremental Model Query Framework in the Cloud. In *ACM/IEEE 17th International Conference on Model Driven Engineering*

Languages and Systems, MODELS 2014, Valencia, Spain, 2014. Springer, Springer. Acceptance rate: 26%.

[37] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. Introduction to Algorithms. `http://en.m.wikipedia.org/wiki/Introduction_to_Algorithms`. Accessed: 2014-10-11.

[38] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.

[39] Pascal Van Hentenryck, Helmut Simonis, and Mehmet Dincbas. Constraint satisfaction using constraint logic programming. *Artificial intelligence*, 58(1):113–159, 1992.

[40] Gergely Varró and Frederik Deckwerth. A Rete Network Construction Algorithm for Incremental Pattern Matching. In Keith Duddy and Gerti Kappel, editors, *Theory and Practice of Model Transformations*, volume 7909 of *Lecture Notes in Computer Science*, pages 125–140. Springer Berlin Heidelberg, 2013.