QEF - Manual do Usuário

Douglas Ericson
Fabio Porto

Data Extreme Laboratory (DEXL)
Laboratório Nacional de Computação Científica

Setembro 2010

1 INTRODUCTION

Purpose of the manual

This document, QEF *User Manual*, provides reference information about the QEF (Query Evaluation Framework) system. The document covers the software full installation, configuration and utilization. QEF is a framework for developing query processing systems that has been extended in a couple of different application domains, such as Scientific Visualization. Interested reader may find more detailed information concerning the CoDIMS architecture and algorithms at [4,5].

QEF overview

CoDIMS (Configurable Data Integration Middleware System) is a middleware environment for the generation of adaptable and configurable data integration middleware systems. Data integration systems were designed to provide an integrated global view of data and programs published by heterogeneous and distributed data sources. Applications benefit from these types of system by transparently accessing resources independently of their localization, data model and original data structure.

We derived from CoDIMS a Query Evaluation Framework (QEF), i.e. a framework where users can define and execute their requests. Afterwards, we improved the framework to run Scientific Visualization requests.

2 QEF

The framework

Main idea

The QEF framework provides a Query Execution Environment Framework that helps users define and execute several types of requests (Figure 1). By "request", we mean a set of user defined operations or functions. Our system manages the execution of the request in a distributed environment (Grid environment), the communication between query execution components, and the access to heterogeneous data sources.

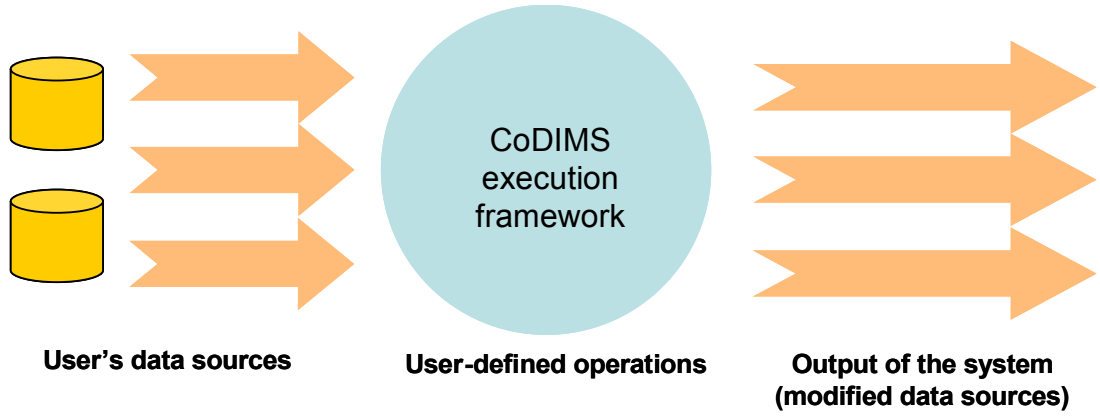

Figure 1: QEF Execution Framework approach.

Data sources

In QEF, users can access transparently heterogeneous data stored at different locations. The idea is to provide a DataSource interface between the data and user's applications. This interface abstracts the format and the location of the data and eases the way the user accesses his data (files, databases, URLs, etc.), i.e. the user reads uniformly from DataSource wrappers without considering types or locations.

Operations and Workflows

The framework provides an easy way for users to execute requests. In QEF, a request is a set of user-defined algebraic operations, communicating with each other and aiming to produce a result. These operations are represented by a Query Execution Plan (QEP). A QEP is represented as a tree where the nodes are operators, the leaves are the data sources, and the edges are the relationship between operators in the producer-consumer form. We consider trees presenting linear or bushy topologies.

Figure 2 shows a QEP composed of four operators (A, B, C and D). Each operator consumes a tuple[1] and produces a modified tuple. Hence, during the QEP execution tuples flow from one operator to another one in the tree; we call this dataflow the producer-consumer model.

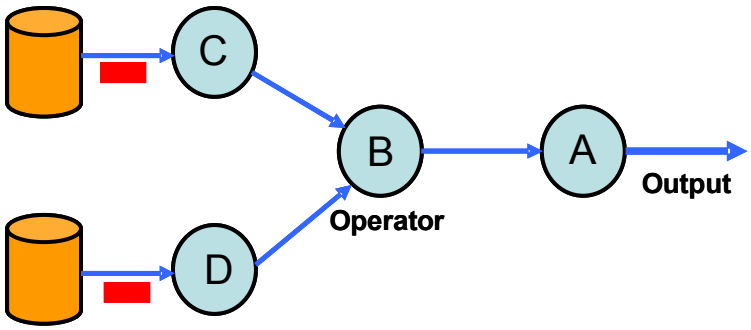

Figure 2 Example of Query Execution Plan.

The operators implement the following operations (iterator execution model [Greaf]): Open - prepares the operator to produce data; Next - produces a data under demand of the consumer operator; and Close - finalizes the execution of the user. Calling one of these operations, at the root operator, will propagate it to its children operators and so on, until reaching the data sources (leaves). Using the Next operation, requests producers to produce a data item (a tuple) to be consumed by a consumer operator, building an execution chain. The result is a pipeline execution of operators synchronized through the Next call between pairs of producer-consumer operators. The pipeline execution mode allows the production of results as soon as the first tuple reaches the pipeline root operator (first-tuple first.). Within the advantages of this technique is the extensibility to

[1] As traditionally adopted in the relational data model, we name the data structure consumed and produced by operators as tuple.

new operators and towards different implementations of the same operator, by just requiring the implementation of the iterator interface.

Concretely, the QEP is an XML file composed of a list of operator templates, where each operator is defined by an id, a name, a list of producers and a list of parameters (Figure 3). With this template we can construct a tree of operators (Figure 2).

```
<op:Operator id="1" prod="2">
        <Name>Operator A</Name>
        <ParameterList>
                <!—Operator's parameters -->
        </ParameterList>
</op:Operator>
```

Figure 3: Operator's definition in the QEP

There are two types of operators in QEF: Algebraic and Control. Algebraic operators implement the algebra of a data model; they act on the content of a tuple, processing according to the desired semantics. Control operators, on the other hand, are Meta-operators that implement an execution characteristic, associated with the tuple dataflow. Combining those two types of operators, applications can support different execution workflows and allows the system to transparently decide on an execution strategy keeping intact the execution semantics.

Distribution and Parallelization

One important aspect in QEF is the distribution of the execution over a Grid environment. Applications benefit from this type of mechanism by reducing query (request) evaluation time.

In QEF, a Query Optimizer decides which operators of the QEP should be parallelized and the set of available grid nodes to be used, based on the cost of the operator on each remote node. Figure 4 shows an operator parallelized over three nodes.

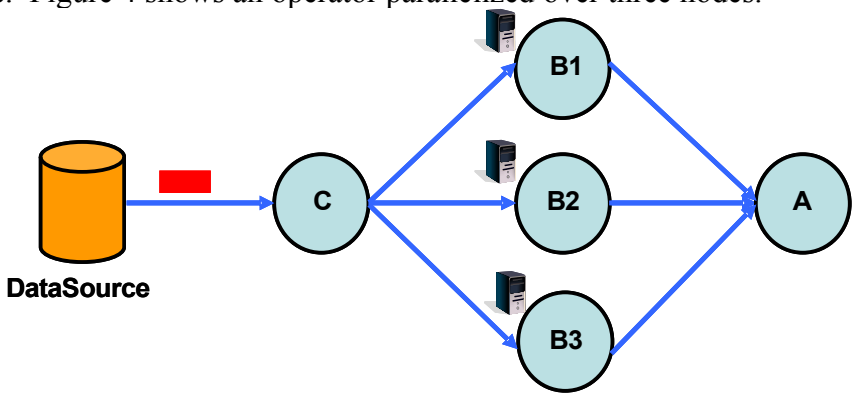

Figure 4 : Operator distribution over three nodes.

When an operator is parallelized over remote nodes, the management of the communications between the nodes is performed by the local node (Master node). A Query Optimizer runs on the local node and calls G2N[1] to take the parallelization

[1] Grid Greedy Node scheduling algorithm

decision. The main idea behind this algorithm is that given a set of remote nodes and a set of tasks it defines a subset of nodes on which the task should be parallelized. G2N also manages the communication between the remote nodes and the central one, i.e. it decides how many tuples should be sent to each node for processing.

In order a distributed executing to happen, the initial QEP is transformed, by the introduction of new control operators. Figure 5 shows an initial QEP transformed to handle the distribution of operator B over two remote nodes. Control operators are represented in green. In this case, the Optimizer adds the following control operators:

- Receiver (R) and Sender (S): allows exchanging of data between nodes
- Instance to Block (I2B) and Block to Instance Converters (B2I): aggregate and disaggregate tuples into blocks to optimize data transfer
- Split (Sp) and Merge (Me) operators: to send and receive blocks from multiple nodes.

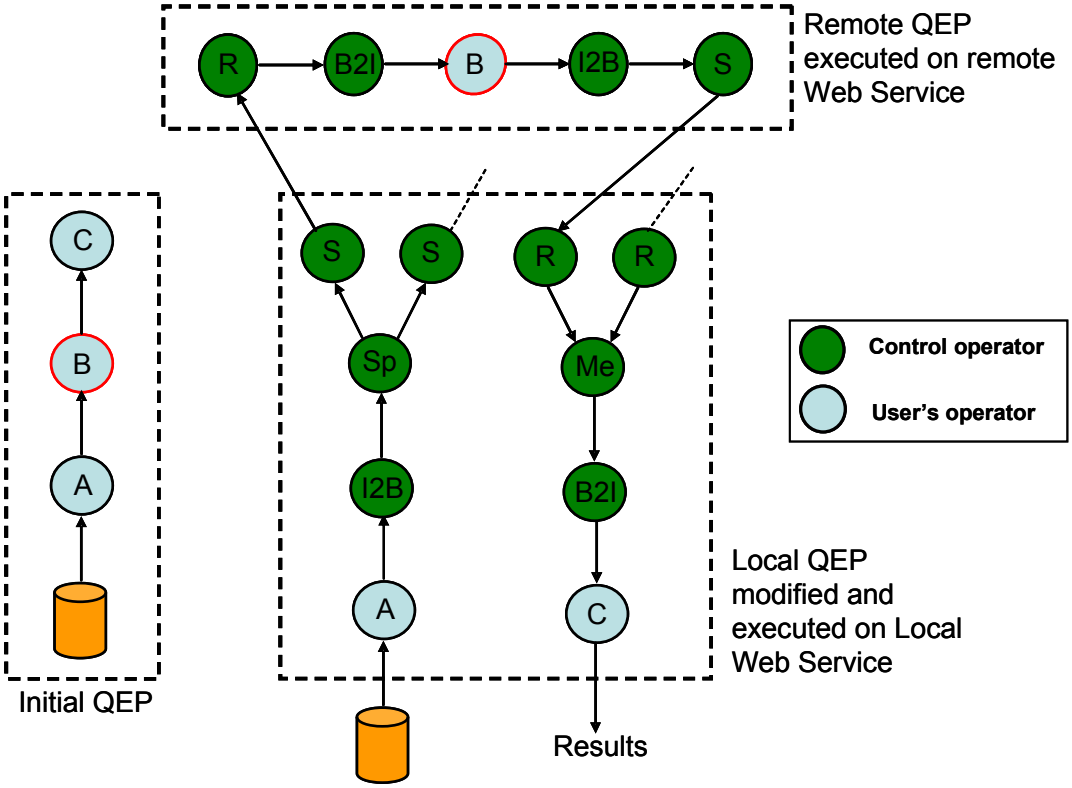

Figure 5: Transformation of an initial QEP and creation of remote QEPs.

3 IMPLEMENTATION

Architecture

The following figure briefly illustrates the architecture of the framework. We can categorize QEF components (Java classes) into three main groups: DataSource management components, Execution and Workflow management components,

Distribution and Parallelization management components. Each of these is described in Table 1.

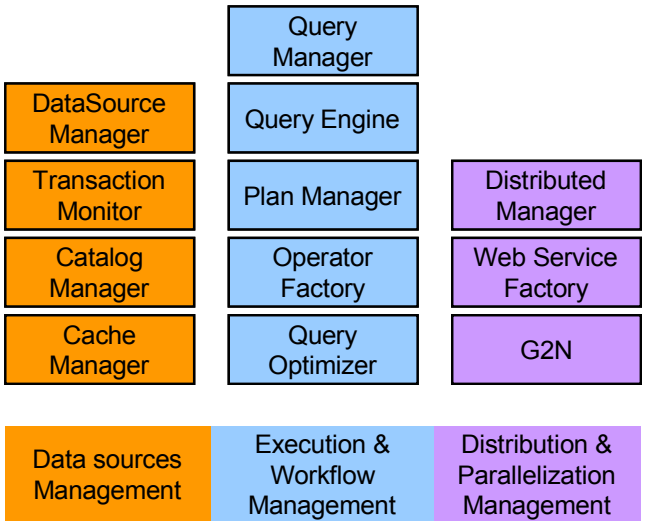

Figure 6: Components of the QEF framework

Table 1: QEF main component's description

Class	Definition
DataSource Manager	Instantiates the Data sources.
Transaction Monitor	Handles the communication with databases.
Catalog Manager	Handles the communication with the QEF internal Catalog.
Cache Manager	Accesses data stored in registered data sources and cache them for repetitive demands. Mainly used by operators requiring direct access to databases and other data sources.
Query Manager	Executes the requests (in centralized or distributed mode). Runs on the local node.
Query Engine	Core component of the execution. It may run on the local node (centralized execution) or on remote ones (distributed execution). In the latter case, a query engine is also run in the local node, managing the distributed execution.
Plan Manager	Creates a query execution plan.
Operator Factory	Instantiates the operators given their class definition (uses Java reflection).
Query Optimizer	Optimizes a query execution plan.
Distributed Manager	Instantiates the remote web services and prepare the distributed environment.
Web Service Factory	Creates GT4 Web Services instances.
G2N	Manages the distribution of tuples among remote nodes.

Grid environment

Globus

Our software uses the Globus Toolkit to build a Grid environment . GT4 is a set of software components that implement Axis web services for building distributed

systems, among other grid specific functionality, such as authentication, file transfer, etc... The web services are hosted by containers and we interact with them by starting the containers.

Communication scheme

We distinguish three components in our distributed architecture:

- The client who can be viewed as an interface between the user and the web services. It performs the initialization and termination of QEF, communicates with the user requests, and manages the communication with the web services. It also executes the request when there's no distribution to be made.
- The local web service which executes the non-distributed part of the query execution plan. It holds G2N algorithm, interact with remote nodes to send and recover tuples, and communicates with the client to return the final result of the request.
- The remote web services which executes the distributed operators.

Each of these components could be running on a separate node or on the same node, called the local node (see Figure 7 and Figure 8).

The client and the local web service often run on the local node. The reason for that is that the local web service and the client do not share the same resources, and do not execute at the same time, therefore the performance is not decreased. On the other hand the remote web services usually run on separate remote nodes, unless you want to test the software and execute the web services on local host's instances.

Finally, we can also run all the instances of the web services and the client on the same node, each of these listening on a different port (). This is usually the case, when we want to test the application or when we do not have an available cluster of remote nodes.

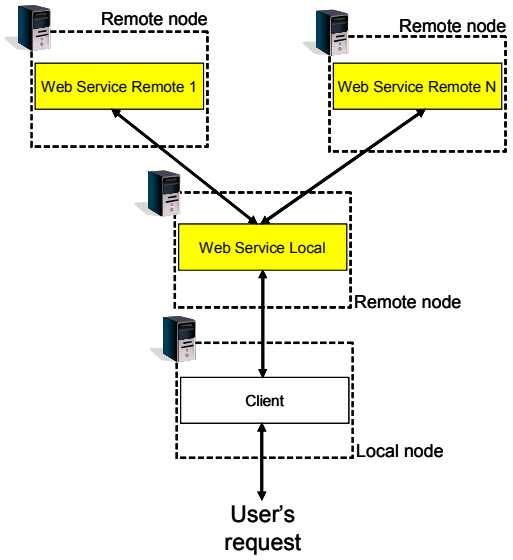

Figure 7: Scenario 1 – Web service's interactions. Each web service is running on a separate node.

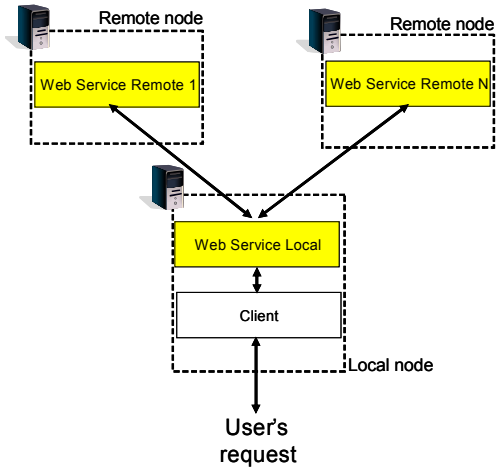

Figure 8: Scenario 2 – The client and the local web service are running on the local node.

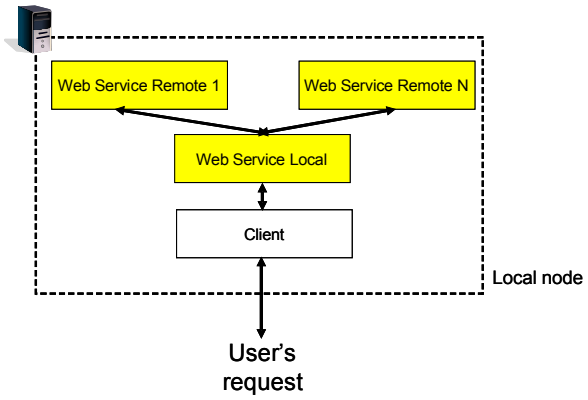

Figure 9: Scenario 3 – The client and all the web services running on the local node.

4 APPLICATION

In this chapter, we explain:

- How to use QEF from Java code;
- How to prepare a Grid environment.

If you find a problem in the installation (or in the installation manual), please contact us. We are also pleased to receive feedbacks and comments.

Prepare the Grid environment

In this section, we'll discuss how to prepare a distributed Grid environment so you can parallelize the execution of the applications in QEF over multiple nodes.

First, you have to choose a cluster of remote nodes (called the environment). **If you do not have an available cluster, you can test the parallelization using local host nodes**. We recommend using UNIX/Linux nodes for your cluster, as it is more convenient to call remote scripts on these machines.

Once you have your cluster you are ready to download the customized version of Globus for QEF on each node (including the central one). Observe, however, that if you simulate the parallel execution environment by running multiple web services in a single machine (i.e. local web services), you will only need to proceed once with the installation described here.

- Download qef.zip from http://???? under the Download page. This file contains: The Globus Java Core Source, the QEF Web Service, The QEF jar file, the *CODIMS_HOME* directory and some logging information.
- Create a srv folder in / directory and unzip qef.*zip* there.
- Set an environment variable called *GLOBUS_LOCATION* that points to that directory

You will now register the environment within QEF and provide some configuration parameters; for doing that, **edit the *file CODIMS_HOME/codims.env*** in the local node (Figure 10).

```
# Set and environment id of your choice
ENVIRONMENT_ID = 1

# Define your environment by giving a list of machines and ports
# Check that the port number is free (netstat -a)
NODES=localhost:8083

# Set the address of your local web service (Could be running on local host or on remote node)
LOCAL_WEB_SERVICE = localhost:8082

# Set the path to CODIMS_HOME
CODIMS_HOME = /srv/QEF/CoDIMS-tcp/build/classes/codims-home
```

Figure 10: The codims.env configuration file

To commit the changes execute codimsEnv.bat or codimsEnv.sh (in *CODIMS_HOME*). These scripts run a Java class (`ch.epfl.CodimsEnv.java`) that modifies the Catalog and the property files. The class reads the one *codimsTemplate.properties* (in *CODISM_HOME/Scripts*), modifies him and generates one new configuration file *codims.properties* (in *CODIMS_HOME*).

Running the application

Now that you have downloaded and installed QEF and GLOBUS on each node, you are ready to run the application.

First, you have to start the Globus container on each remote node (otherwise QEF Web Service won't be available). To start the container, type the following command:

```
globus-start-container –nosec –p portNumber
```

The portNumber field is the port number on which QEF Web Services will listen for incoming requests; and the *nosec* arguments indicates that we are running the Globus container without any security mechanism.

If you want to run *n* instances of a GT4 web service on your local node, you will open three command windows and type the command above *n* times giving different port numbers.

We have also provided scripts for starting and stopping a list of web services. See under *CODIMS_HOME/scripts/start.sh* and *CODIMS_HOME/scripts/stop.sh*. You can use these scripts to start Web Services running under UNIX or Linux machines. Note that, these scripts make use of SSH. So make sure that you have SSH installed on each machine or replace the command by RSH (which requires fewer configurations).

Use QEF to run any application from Java

In this section, we show how to install QEF in order to run the any application process from Java.

Pre-requisites

Use your favorite Java IDE to create a new project and import the classes. Add all the jar files located in *CODIMS_PROJECT/lib* to your project. Afterwards, prepare your Grid environment, if it is not yet installed (see section); edit the file codims.env. Finally, start the containers on each node of your environment (as described in section).

Request execution

Overview

The class `QueryManagerImpl` (in `package ch.epfl.codimsd.qeef`) defines an interface for communicating with QEF:

- `QueryManagerImpl getQueryManagerImpl()`
- `RequestResult executeRequest(Request request)`
- `void shutdown()`
- `long executeAsync(Request request)`
- `ExecutionState getExecutionState(long requestId)`
- `RequestResult getRequestResult(long requestId)`

See the `main` method for a full example of utilization.

Prepare the request

In QEF, we query demand is considered more generically as a *request*. Thus a user builds a request object and submits it to QEF for execution. A request object comprises the following structure:

- The type of the request (it's ID). This one should be defined in the Catalog.
- A RequestParameter object (Package ch.epfl.codimsd.query) which encapsulates a `HashMap` object storing what the application needs to process. The user may want to add to the `HashMap` parameter values that the operators might need to access during the execution.
- Some tuning parameters added to the `RequestParameter HashMap`, described in Table 2.

Table 2: Request parameters

Parameter	Definition
LOG_EXECUTION_PROFILE	When set to "TRUE", this parameter allows logging the execution so you can see the progression of the execution on the Web interface.
NO_DISTRIBUTION	When set to "TRUE" this parameter forces QEF to run in a centralized mode.

Initialize QEF

Before, running the application we need to start QEF by calling the method `getQueryManagerImpl` of the class `QueryManagerImpl`. Afterwards, the user may execute multiple queries without re-initializing.

Executing a request

There are two modes for starting a request execution in QEF, single-user or multiple-user mode. In a single user mode, request execution blocks QEF until the latter finishes the evaluation. In this mode, a new request has to wait for the previous to end. In the multi-user mode, requests run asynchronously, freeing QEF to accept and run new requests. In the multi-user mode, all common data-structures stored in the local node are shared between concurrent requests, optimizing the overall performance.

The single-user mode is obtained by invoking the *`executeRequest`* method of the class *`QueryManagerImpl`*. On the other hand, a call to the *`executeAsync`* method introduces a asynchronous evaluation of the requests, allowing concurrent query evaluations. When you call this method you get a *request id* that you can use afterwards to know the state of your execution and to get the final results. For doing this, you can periodically call *`getExecutionState(yourID)`* of class *`QueryManagerImpl`* and get an *`ExecutionState`* object encapsulating execution state information. If the *`ExecutionState`* indicates that the execution is finished (`isFinished()`) method of class `ch.epfl.codimsd.qeef.ExecutionState`), you can use `getRequestResult` to retrieve the complete result.

Obtaining the request result

When the *ExecutionState* returns a isFinished indication, the execution has been successfully terminated and the results can be obtained from a `RequestResult` object. This object is composed of:

- ResultSet object containing a list of tuples. Each tuple is a single object containing the result.
- The metadata of the tuple, that helps you reading the object.
- The time of the execution.
- A result integer code.

Finally, when you finish with QEF you can *close()* the system by calling the `shutdown` method.

5 CONFIGURATION

In this section, we present in more detail the configuration parameters.

Derby Catalog

The Catalog is an Apache Derby database where we store some configuration parameters needed by QEF. These parameters could be classified in two categories: environment parameters and request parameters.

You will find under *CODIMS_HOME/SQL Requests/Catalog.txt* the script used to create and fill the tables.

Why Derby?

Derby is a relational database implemented in Java. We have chosen to use this database to build the Catalog for the following reasons:

- Derby is based on java, JDBC, SQL and offers a JDBC embedded driver, which allows embedding the database in a Java-based solution.
- We can easily install and deploy the database by adding the derby jar file to the project (note: Derby is already supplied within the QEF bundle and you should not worry about its installation, as far as QEF execution and requests evaluation are concerned.)
- It supports concurrent access
- It supports client/server mode with the Derby Network Client JDBC driver and Derby Network Server . This solution is useful when an operator is running on a remote machine and requires to access to the local Catalog.

System parameters

Table 3 defines the tables used to store system configuration parameters. These parameters are not related to a request and should not be changed unless you want to modify the behavior of the internal components.

Table 3: Environment parameters stored in the Catalog

Table name	Definition
initialConfig	Stores parameters used in some Java classes (size of buffers, size of caches, timeouts, maximum number connections to

	databases, etc.)
executionProfile	Used by the Web interface. It stores the state of the execution. Basically, each time a node processes a tuple, we write the number of processed tuples in the table and we print the new number to the screen.
initialExecutionProfile	Stores initial execution configuration (number of remote nodes, id of request, rates of each node and number of initial tuples to process).

Request parameters

In this section, we present the tables used to store parameters specific to a request type. **When a user creates its request within QEF, he should update the Catalog with all these parameters.** Table 4 presents the parameters.

Table 4: Request parameters stored in the Catalog.

Table name	Definition
Datasource	Name and the class name of QEF datasources.
Ds_database	Connection parameters to databases (IRI[1], username, password, driver name, server).
RequestType	The name and id of the requests.
Template	The path to the QEP on the local machine.
RequestTypeTemplate	Bind a request type to a query execution template.
OperatorType	The name and class name of QEF operators.
Ds_Table	Stores the number of tuples for each datasource
Environment	Defines the environments available in QEF.
Node	List of nodes for each environment.
AppNodeRate	The rate (in milliseconds) of the operators on each environment nodes. If you don't know the approximate rate of your operator on each machine, you cant set the value to 500 ms.

CatalogManager class

The CatalogManager class (package *ch.epfl.codimsd.connection*) manages the communications with the Catalog.

To create the Catalog, edit the file *CODIMS_HOME/SQLRequest/Catalog.txt* and run the `setup` method of the `CatalogManager` class.

File Properties

[1] The IRI (Internationalized Resource Identifier) is the unique identifier of this database. Choose one and check its uniqueness.

QEF properties

We have provided a Java property file named codims.properties and located in *CODIMS_HOME* that defines some additional parameters needed for the request execution. Next table presents those parameters.

Table 1: codims.properties parameters

Parameter	Definition
IRI_CATALOG_DERBY	This field holds information on how to access the Derby Catalog (Path to the catalog, driver, username and password).
ENVIRONMENT_ID	The id of the available remote environment (Should be defined in the Catalog).
LOCAL_WEB_SERVICE	Address of the local Web Service.
IS_DERBY_STARTED	If this value is set to "TRUE" the Derby server is started. Set it to "TRUE" when running QEF from Tomcat and to "FALSE" otherwise (Java code).

6 CONCLUSION

In this manual we presented QEF a query processing system. The execution is supported by a query optimizer that selected available nodes and schedule algebraic operators to run in parallel in those nodes. QEF brings to the applications the following benefits: extensibility, performance, transparency. It provides extensibility by modeling an application process as an algebraic expression, implemented through standardized operator interface. New operators can easily be integrated and existing ones may offer various implementations.

Finally, QEF manages distribution transparently to the user. The decision on nodes schedule, the deployment of necessary software and the execution control is done without any extra effort from the user.

This manual presents installation procedures to be followed in order to run QEF. We present two different execution scenarios: java application based installation, and IDE based execution. Required configuration is explained and necessary parameters are introduced.

The authors of this manual would acknowledge any information that may help to enhance its content and motivate users to give feedback on their experiences by mailing the authors.

Known issues

Time delay when loading the web services

REFERENCES

[1]. Barbosa ACP, Porto F., Melo RN, Configurable data integration middleware system. *Journal of the Brazilian Computer Society* 2002; **8**:12–19.

[2]. Fabio Porto, Vinicius F. V. da Silva, Marcio L. Dutra, and Bruno Schulze. An adaptive distributed query processing grid service. In *Proceedings of the Workshop on Data Management in Grids, VLDB, Trondheim, Norway, 2-3 September 2005*, 2005.

[3]. http://www.globus.org/toolkit/

[4]. http://java.sun.com/j2se/1.5.0/

[5]. http://db.apache.org/derby/

[6]. http://db.apache.org/derby/papers/DerbyTut/ns_intro.html

[7]. Goetz Graef, "Volcano - An Extensible and Parallel Query Evaluation System", IEEE Trans. Knowl. Data Eng.,V(6),N(1),1994.

java -classpath CoDIMS.jar ch.epfl.CodimsEnv