An Extensible Query Execution Engine for Supporting New Query Execution Models

Fausto Ayres[1], Fabio Porto[2], and Rubens Melo[3]

[1] Centro Federal de Educação Tecnológica do Rio de Janeiro (CEFET), Brazil
fausto@cefet-rj.br
[2] EPFL - Ecole Polytechnique Fédérale de Lausanne
School of Computer and Communication Sciences
Database Laboratory, 1015 Lausanne, Switzerland
fabio.porto@epfl.ch
[3] Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Brazil
rubens@inf.puc-rio.br

Abstract. Query processing in traditional Database Management Systems (DBMS) has been extensively studied in the literature and adopted in industry. Such success is, in part, due to the performance of their Query Execution Engines (QEE) for supporting the execution of traditional queries. With the advent of the web and its semi-structured data model, new query scenarios were created, suggesting new execution models such as: adaptive, continuous, and stream based. To support these models, the traditional QEE must be extended, resulting in a great development effort as the one recently seen to support the XML data model. This paper proposes the design and construction of an extensible QEE adapted to new execution models and our approach is to implement each execution model as a combination of execution modules. Thus, adding new modules to this QEE, new execution models will be supported. To achieve this goal, we use a software framework technique to produce a framework, named QEEF (Query Execution Engine Framework).

1 Introduction

The adoption of Relational Database Management System (RDBMS) in management of databases is a reality in the industry. The success of those systems was achieved, in part, by the efficiency of their query processing, in witch has been widely studied by the database community. That efficiency is due to the Query Optimizer, that produces optimized Queries Execution Plans (QEP), and to the Query Execution Engine (QEE), which receives and executes a QEP according to a query execution model. We consider a "query execution model" as a set of execution techniques, called "execution characteristics", to produce a data flow that attends to the requirements of a set of query applications, generalized as "application scenarios". For example, the traditional scenario is represented by SQL queries in RDBMS, and its execution model uses the execution characteristics of parallelism, distribution, and demand-driven control flow. With the advent of the Internet computational model and new data models

(such as semi-structured model [1]), new query execution scenarios were produced, such as:

1. Queries that have to deal with few (or no) data statistics or with variations in query execution conditions, such as: unpredictable delays to access data sources, selectivity and lack of memory ([2], [3], [6], [7] and [18]).
2. "Publish/Subscribe" type queries that need to be re-evaluated continuously, according to the user needs ([9], [10]).
3. Queries over data streams, where data are processed upon arrival ([11], [19] and [21]).

Such scenarios produce new execution characteristics, such as: adaptability of the QEP and data-driven control flow, both not supported by the traditional QEEs. Furthermore, these scenarios suggest new characteristics that will be required by new query applications, resulting in new extensions of traditional QEEs. On the other hand, those extensions can demand a considerable development effort, as seen recently in commercial DBMS to support the XML data model [22].

The proposal of this work consists on the design and the implementation of an extensible QEE that supports different execution models. In our approach, each execution characteristic's state is implemented by an execution module, so that, a particular combination of these modules in a QEP permits the QEE to support a particular query execution model. Therefore, the specification of new modules provides the extension of QEE supporting new execution models. The idea of using execution modules as QEP operators is not new, following the example of the Exchange [16] and Eddies [3] operators, that encapsulate respectively, the parallelism and the adaptability of the algebraic operators. However, we don't know in literature a QEE that combines and extends modules in a flexible and uniform way to support different execution models.

In order to design the extensible QEE, we use the software framework technique [14], producing the QEEF (Query Execution Engine Framework) [4]. As a case study, we instantiate the engine AQEE (Adaptive Query Execution Engine) to support the execution model of the scenario 1 above.

This work is organized as follow: Section 2 describes the execution scenarios mentioned above; Section 3 presents the new concepts: execution characteristics, modules, and models; Section 4 specifies the architecture of the extensible QEE; Section 5 shows the AQEE case study, and, finally, in Section 6 we present our conclusions.

2 Query Execution Scenarios

In this section we will detail the scenarios presented in the introduction of this work as well as the main related works.

2.1 Preliminaries

A query execution engine (QEE) is responsible for the execution of a Query Execution Plan (QEP) submitted, in general, by the DBMS optimizer. A QEP is composed

by a set of related operators aiming to produce query results. Most DBMs represent a QEP by 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. These trees present linear or bushy topology.

During the QEP execution, tuples of data flow through the QEP's operators in the following manner: first, a tuple is created by an operator that access its data source, and then it is processed by the others operators until it is discarded or it is sent to the output. This dataflow is in according to the producer-consumer model, in witch tuples flow from producers to consumers' operators. Furthermore, this dataflow must be in according to the techniques needed by the own applications.

Although different architectures of QEE are found in literature, is not known a study that classifies them according to their functionalities. Additionally, it is not easy to find a clear separation between the QEEs and the other components of a query processing system. However, we define the architecture of a QEE by the following elements:

- Execution Unit: corresponds to an operator;
- Data Unit: corresponds to a data tuple processed by any operator;
- Data Structure: corresponds to a persistent or volatile structure necessary to support the operators execution (ex: lists, bags, sets and trees);
- Data Model: corresponds to an algebra implemented by the algebraic operators;
- Execution Model: corresponds to a set of techniques to produce a data-flow required by a given scenario.

2.2 Traditional Execution Scenario

The state of art in traditional QEEs is based on the iterator interface [17], in which the operators implement the following operations: Open - it prepares the operator to produce data; Next - it produces a data under demand of the operator consumer; and Close - it finalizes the execution of the user.

Calling one of these operations, at the root operator, will propagate it to its operators' children and so on, until reaching the data sources (leaves). Using the Next operation, each operator will produce a data when its consumer operator requests it. As result, connecting each pair of operators related to a plan, it is able to execute any plan in pipeline, increasing the performance and producing output results as soon as possible (first-tuple first.) Within the advantages of this technique is the extensibility to new operators and to different implementations of the same operator [5]. Another feature of the traditional execution model is the use of the Exchange operator [16], for parallel execution, and Send and Receive operators [20], for distributed execution.

2.3 Adaptive Execution Scenario

In this scenario, the QEE adapts QEP operators according to changes in certain conditions that can occur during execution, since:

1. They have their performance slow down, due to the unpredictable delays in access time of QEP's data sources.

2. They detect an internal problem.

A query processing system is considered adaptive if (a) it receives information from its environment, (b) uses this information to determine its behavior and (c) processes in a continuous way, generating an interaction between the environment and the system. In this way, the adaptive QEE offers a dynamic optimization (in run time), in contrast with the traditional process of optimization that is static.

The adaptive execution of operators in a QEP can be done in terms of tuple or in terms of fragment. In the first case, each tuple is always sent to the operator of higher performance, until complete all the operations necessary to be sent to the output. In the second case, the execution will occur in a traditional form inside of a QEP fragment until it is detected some problem, and in that case, the fragment execution is interrupted and the optimizer is called to generate another alternative fragment. The main solutions found in literature are namely: Query Scrambling [2], Eddies [3], Dynamic Query Scheduling [6], Hybrid [7] and Tukwila [18].

2.4 Continuous Execution Scenario

The query execution in a continuous way permits the operators to obtain new results of a database without submit the same query, repeatedly. In general, the control flow is data-driven, or in other words, as new data arise, new results are produced. Besides that, the execution time of a query can be very long. Examples of related works are namely: TelegraphCQ [9] and NiagaraCQ [10].

2.5 Streams-Based Execution Scenario

This execution scenario is characterized by the following elements:

- The stream data arrives online;
- The QEE does not have control about the order in which the stream elements stream must be processed;
- A stream can have unlimited size;
- The data are processed at once due to the space limitations in memory.

Examples of application for this scenario include financial application, sensor network application and click-streams application. Examples of related works are [11], TurboPath [18] and Continuous Eddies [21].

We use these scenarios above as examples of extensibility of a QEE to support different and new requirements of application, in an incremental way. In the next section we detail these execution scenarios.

3 The Supporting of Different Query Execution Models

Traditionally, a QEE is built to support a specific query execution model, present in a certain scenario. In this context, the PECs submitted to a QEE are executed always

using the same execution model that, in general, is implicit into the QEE's implementation. For example, the traditional QEE executes QEPs based on the traditional model.

The first step toward the supporting of different execution models by a QEE, was to analyze the scenarios previously presented, and capture new execution concepts, named of:

- Execution Characteristic: is a type of execution technique that produces the desired data flow control needed by a (or part of) QEP. Each characteristic can have different states.
- Execution Module: is a high level operator that implements an execution characteristic state by inserting control operators between algebraic operators, based on some strategy.
- Execution Model: is the result of the combination of execution modules to attempts the requirement of a given execution scenario.

The Table 1 relates these and others execution concepts in composition levels (from 1 to 6) and in abstraction levels (Physical and Logical) where the physical concepts implement the logical concepts.

Table 1. The composition and abstraction levels of query execution concepts

Level	Physical Concept	Logical Concept
1	QEE	Query Execution Scenario
2	QEP	Query Execution Model
3	Module	Query Execution Characteristic's State
4	Control Operator	Meta-Operator
5	Algebraic Operator	Data Model's Algebra
6	Tuple	Data Model's Data Structure

At level 1 (the highest) we have a customized QEE implementing each query applications scenario. At level 2, a QEP implements an execution model using different models. At level 3, model implements a certain state of a characteristic of execution, matching one or more control users. In level 4, we have a control operator implementation for each type of communication between operators. At level 5, we have an algebraic operator implementation based on data model's semantic operations. In level 6 (the lowest) we have a tuple implementation based on a data model's data structure.

The extensible QEE proposed in this paper, named QEEF, is based on this different composition levels and, in particular, on the combination of execution modules in a QEP, to support different execution models. In this case, the user must specify a QEP containing these modules. Permitting this specification, we propose the meta-model QUEM (QUery Execution Meta-Model.) that defines the rules for combination of existing modules and the new ones. In the following sections, we detail these execution concepts and the meta-model QUEM.

3.2 Query Execution Characteristics

From an analysis of the scenarios described in the previous sections, we observe some abstractions, named “query execution characteristics”, that can be presented in one or more execution models and they can assume different states depending on the model considered. Some of these characteristics were introduced to [17]. They are:

- Synchronism: it defines the state of the execution of an operator when he requires service or data from another operator. There are three possible states: wait (synchrony pipeline), wait-all (sequential synchrony) and no-wait (asynchrony);
- Parallelism: it occurs when two operators process data in an independent way, in other words, the consumption of the data is not synchronized with its production. There are two states: Intra and Inter-operator;
- Distribution: it concerns about distributed execution of the operators in a QEP to support the consumption of data from a remote operator; the alternatives are: Local or Remote;
- Data Flow: it specifies the order in which data are processed. There are two states:: Fixed, where the tuples flow through the operators in the same way, and Adaptive, where the tuples flow in different ways, depending on the performance of these operators;
- Control Flow: it refers to the kind of control between operators producing a chain of execution between consumers and producers’ operators. The alternatives are: Demand-Driven and Data-Driven;
- Response Time: it refers to the moment in which the query results will be available to the user application. There are two states: First-Tuple first, that requires data to be sent to the user immediately after they are produced, and Last-Tuple first, that requires data to be sent to the user only when all the tuples have been produced.

3.3 Query Execution Modules

The Table 2 suggests the most common relationship between the execution modules and models based on the scenarios described in Section 2, including the traditional scenario. This table shows, for each model, the correspondent modules that must be implemented by the QEE. Note that each characteristic permits several alternative modules in one or more models, each model permits several alternative modules of the same characteristic. In the following sections, we describe some of these modules.

Table 2. Relationships between Modules and Models.

Execution Characteristic	Module	Execution Model			
		Traditional	Adaptive	Continuous	Stream based
Synchronism	Wait	*	*		X
	Waitall		X		X
	Nowait	X	X	*	*
Distribution	Remote	X	X	X	X

Parallelism	Local	*	*	X	X
	Inter	*	*	X	X
	Intra	X	X	X	*
Data Flow	Fixed	*		X	*
	Adaptive		*	*	X
Control Flow	Demand-Driven	*	*	X	X
	Data-Driven		X	*	*
Response Time	First-Tuple	*	*	*	*
	Last-Tuple		X		X

Legend: (*) is a more relevant configuration and (X) is a possible configuration.

Wait, Nowait, and Waitall Modules. The figure 1 shows the *Wait, Nowait and Waitall* modules, in which are implemented through the homonym control operators. In this figure, the arrows indicate the producer-consumer data flows, between the algebraic operators op1 and op2, and the control operators *Wait, Nowait* and *Waitall*.) The module *Wait* uses the *wait* operator for synchronizing the op1 and op2 operators in a way that op2 waits for a tuple to be produced by op1. In an analogous way, the module *Nowait* uses the *Nowait* operator to decouple op1 and op2 in a way that data consumption is independent of its production, in two different situations:

- When the consumption rate of op2 is lower than the production rate of op1, the data that arrive in the *Nowait* operator are temporally stored in a buffer while they are waiting for being consumed by *o*p2. The limitation of the memory buffer size can control the *Nowait* production rate;
- When the consumption rate of op2 is higher than the production rate of op1, the *Nowait* operator behaves as a *Wait* operator*;*

The module *Waitall* uses the *Waitall* operator to consume (and materialize) all the tuples produced by op1, before producing the first tuple to op2. Although this technique can have an importance in the execution of certain QEPs, it's not efficient as a whole, because the input/output cost could be too expensive.

Demand-Driven and Data-Driven Modules. The Figure 2 shows the *Demand-Driven* module, in which a consumer requires data to its producer through a Get method call, and the *Data-Driven* module, in which a producer sends data to its consumer through a Put method call. Although most of execution models need only one of these modules in the same PEC, it's possible to combine them through control flow schedulers, implemented by the *passive* and *active* control operators (see Figure 2). In this case, the *active* operator requires data from its producer (by get method) and sends it to its consumer (by put method). The *passive* operator receives data from its producer (by put method) and these data are sent (by get method) to its consumer.

Adaptive Module. The Figure 3 shows the *Adaptive* module for a QEP fragment that needs to adapt the op1 and op2 operators, by inserting the *distributor* control operator, in which distribute tuples based on some distribution policy*,* and inserting both *wait* and *active* control operators*,* to scheduler op1 and op2, in the follow manner: each *wait* operator consumes a tuple from the *distributor*'s input queue, and sends it to its consumer operator (op1 or op2) that process it and, if is not discarded, sends it to the correspondent *active* operator to send it back to the *distributor*'s input queue, to be processed by another operator or, in the case of it has been processed by all internal operators, to be sent to the *distributor*'s output queue. For example, the tuples processed by the QEP fragment in the Figure 3 would flow two possible ways: op1→op2 and op2→op1.

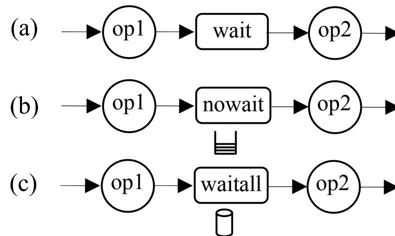


Fig. 1. Modules *Wait* (a), *Nowait* (b) e *Waitall* (c).

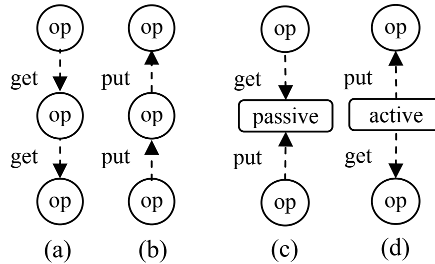


Fig. 2. The (a) *Demand-Driven* and (b) *Data-Driven* Modules*.* These modules can be combined through the (c) *passive* and (d) *active* control operators.

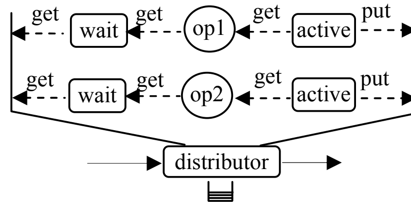


Fig. 3. *Adaptive* Module

3.3 Query Execution Models

In this section we show examples of the combination of the modules to produce the Traditional, Continuous, and Adaptive models. It's important to be noted that these examples are based on the same QEP fragment.

Traditional Model. The Figure 4 shows, on the left side, an example of QEP with the modules *Fixed*, *First-tuple*, *Demand-Driven* and *Wait*. The solid arrows indicate the direction of data flow. The data sources were omitted by simplicity. This Figure illustrates, on the right side, the QEP to be executed, where the dashed arrows indicate the control flow direction. As a consequence of this execution process, the operators, op1, op2, and op3, process tuples in a synchrony way. As the application requires new tuples, they are produced by op3. Each tuple flows in the fixed way: op1→op2→op3.

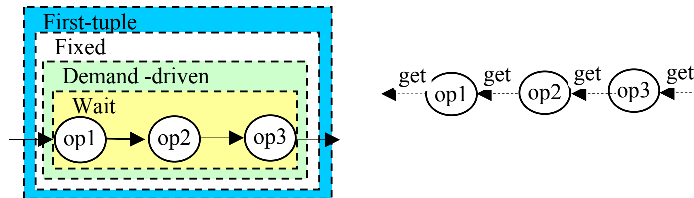


Fig. 4. Example of the Traditional Execution Model

Continuous Model. The Figure 5 shows, on the left side, an example of QEP with the *Fixed, First-tuple, Data-driven,* and *Nowait* modules, and shows, on the right side, the QEP to be executed.

Adaptive Model. The Figure 6 shows, on left side, the modules Adaptive, Fixed, First-tuple, Demand-Driven and Wait modules, and shows, on the right side the QEP to be executed. When the application requires a tuple from the distributor, it consumes tuples from op1 and sends them to the adaptable execution of the op2 and op3 operators, until one of them produce a result tuple. As a consequence of this execution process, the three operators, op1, op2 and op3 process tuples in a synchrony way. As the application requires new tuples, the distributor produces these. The tuples flow in different way depending on its adaptability.

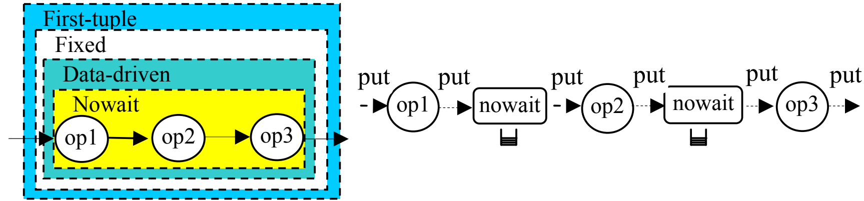


Fig. 5. Example of the Continuous Execution Model

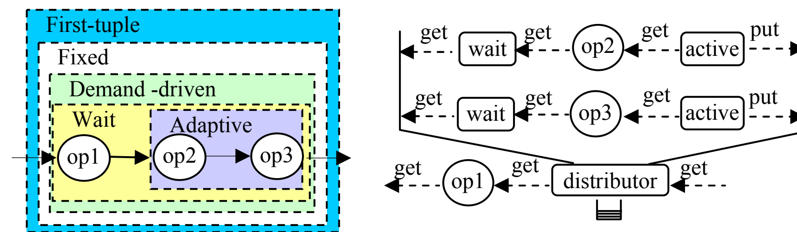


Fig. 6. Example of the Adaptive Execution Model

3.4 Query Execution Meta-Model

A QEP, submitted to the extensible QEE, is specified in a XML document, according to the traditional approach or our approach (named meta-QEP), in which is based on an execution meta-model, named QUEM (QUery Execution Meta-model). In the traditional approach, the QEP's descriptions contain algebraic and control operators. In our approach, the meta-QEP's description contains algebraic operators (algebraic QEP) and modules, combined by QUEM in a producer-consumer form. A XML document for a meta-QEP is specified by the QUEM DTD presented in the Figure 7. A similar DTD, without modules, is used to specify a traditional QEP.

A meta-QEP is pre-processed by a meta-QEE, resulting in a final QEP that will be executed by a QEE, as the traditional QEP would be. The Figure 8 illustrates the forms to submit a QEP to the extensible QEE. The Figure 9 illustrates the two execution levels for a meta-QEP.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT   METAPLANO (listoperator, MODULO)>
<!ELEMENT   MODULO (FIXED | ADAPTIVE | DEMAND-DRIVEN | DATA-DRIVEN |
            REMOTE | LOCAL | INTRA | INTER | LASTTUPLE | FIRSTTUPLE | WAITALL |
            NOWAIT | WAIT | DEFAULT)>
<!ELEMENT   ADAPTIVE ((MODULO | ALGEBRICO), (MODULO | ALGEBRICO)+)>
<!ELEMENT   DATA-DRIVEN (MODULO | ALGEBRICO)>
<!ELEMENT   DEMAND-DRIVEN (MODULO | ALGEBRICO)>
<!ELEMENT   FIRSTTUPLE (MODULO | ALGEBRICO)>
<!ELEMENT   FIXED (MODULO | ALGEBRICO)>
```

```
<!ELEMENT   INTER ((MODULO | ALGEBRICO)+, ALGEBRICO)>
<!ELEMENT   INTRA (MODULO | ALGEBRICO)>
<!ELEMENT   LASTTUPLE (MODULO | ALGEBRICO)>
<!ELEMENT   LOCAL (MODULO | ALGEBRICO)>
<!ELEMENT   NOWAIT (MODULO | ALGEBRICO)>
<!ELEMENT   REMOTE (MODULO | ALGEBRICO)>
<!ELEMENT   WAIT (MODULO | ALGEBRICO)>
<!ELEMENT   WAITALL (MODULO | ALGEBRICO)>
<!ELEMENT   DEFAULT (ALGEBRICO)>
<!ELEMENT   ALGEBRICO (MODULO | ALGEBRICO)*>
<!ATTLIST   ALGEBRICO name CDATA #REQUIRED ref CDATA #REQUIRED>
<!ELEMENT   listoperator (operator+)>
<!ELEMENT   operator (parameter+)>
<!ATTLIST   operator id CDATA #REQUIRED  type CDATA #REQUIRED
            name CDATA #REQUIRED>
<!ELEMENT   parameter (itemparameter+)>
<!ATTLIST   parameter  name  CDATA #REQUIRED>
<!ELEMENT   itemparameter >
<!ATTLIST   itemparameter type CDATA #REQUIRED value CDATA #REQUIRED >
```

Fig. 7. The QUEM DTD for a meta-QEP specification

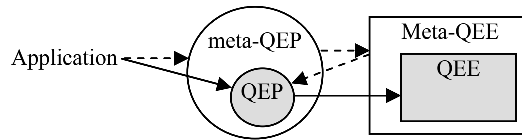


Fig. 8. The two different approaches to submit a QEP

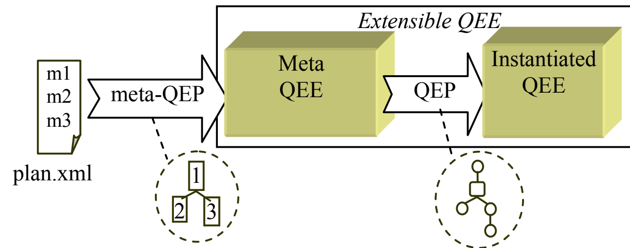
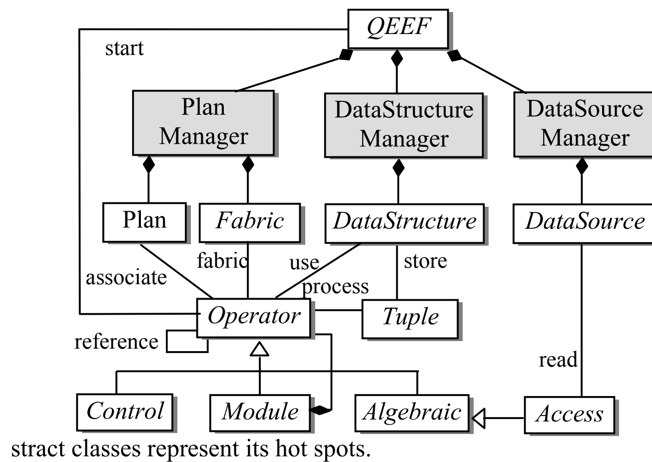


Fig. 9. The two execution levels for a meta-QEP. At the first level, the QEEF behaves like a meta-QEE, executing the modules m1,m2, ..., and mN, of the meta-QEP. At the second level, the QEEF behaves like an instantiated QEE, executing the algebraic and control operators of the final QEP.

4 The Extensible Query Execution Engine

In this section, we present the specification of the Extensible QEE, named QEEF (Query Execution Engine Framework), a white-box software framework [14] implemented in Java language, to be instantiated according to an execution and data models, producing a QEE to support these models. Additionally, the QEEF can be instantiated for different execution and data models producing different QEEs, in an orthogonal way. The Figure 10 shows the QEEF architecture specified in a (high level) UML diagram, where the concrete classes represent its frozen-spots and the ab-



stract classes represent its hot spots.

Fig. 10. The architecture of the QEEF

The QEEF class implements the Facade design pattern [15] and it interfaces a QEE and the user application, in which submits a QEP (or a meta-QEP) specified in a XML document. The Plan Manager class implements the Singleton pattern and manages the construction of a plan (Plan class). In the following way: upon receiving the XML description of a meta-QEP, the algebraic operators (Algebraic class) are created, and then the modules (Module class) are created and combined according to the QUEM meta-model. The Module class implements a Decorator pattern. Each module creates its own control operators (Control class) and inserts them into a QEP, according to the logic of composition of each module, producing a final plan object composed only by algebraic and control objects related in the consumer-producer manner.

The Factory class is used to create all operators. During the execution, an operator can use many data structures (Structure class) for persistence of its tuples (Tuple class). The Singleton class DataStructureManager that manages the data persistence creates all data structures. The Access class represents the algebraic operators that access a data source. The DataSource class encapsulates all access to data sources and need to be specialized (via adapter pattern) to the data source's interfaces. The Singleton class DataSourceManager creates all data sources, based on their metadata. The

Operator class implements, to all operators, an iterator-based interface consisting of the following operations: Open - it prepares the operator to produce data; Hasnext - it produces a tuple; Getnext - it sends a tuple under demand of the consumer; Putnext - it sends a tuple to the consumer; and Close - it finalizes the operator.

The initialization of a plan is made through an open method call from its root operator, in which can propagates it to the others operators, depending on the execution model. The execution of an operator is made through the hasnext and getnext methods calls (in a demand-driven control flow) or a Putnext method call (in a data-driven control flow). The data of the tuples being produced can be printed out by the toString() method in the Tuple class.

5 A Case Study

We present in this section a case study, motivated by the adaptive scenario, instantiating the QEEF to the adaptive execution model, producing the AQEE (Adaptive Query Execution Engine) to be used by the web data integration application described in the follow. In this case, we consider a local data source (represented by L1) containing the initial list of products and three sites of supermarkets with information published in the web (represented here by the data sources S1, S2 and S3). The exported relational scheme is composed of the relations: L1 (cod, name), S1 (cod, price), S2 (cod, price) and S3 (cod, price). A data integration application that aims at recovering the prices of products of the three sites above would be expressed by the SQL query below:

```
Select L1.name, S1.price, S2. price, S3. price
From   L1, S1, S2, S3
Where L1.cod = S1.cod and L1.cod = S2.cod and L1.cod = S3.cod
```

The predicate p = (L1.cod=Si.cod) illustrates the need of providing a search criterion to access published data by the site Si. In this case, each site requires the association of a product code to the price of this product. This way, the data source L1 should be the first to be accessed to obtain the initial list of products. It defined then, a partial order between the operators that will compose the QEP. There is, to deep left trees, the following possible orders of evaluation of the predicates: $O_1 = L1\rightarrow S1\rightarrow S2\rightarrow S3$, $O_2 = L1\rightarrow S1\rightarrow S3\rightarrow S2$, $O_3 = L1\rightarrow S2\rightarrow S1\rightarrow S3$, $O_4 = L1\rightarrow S2\rightarrow S3\rightarrow S1$, $O_5 = L1\rightarrow S3\rightarrow S1\rightarrow S2$, and $O_6 = L1\rightarrow S3\rightarrow S2\rightarrow S1$. Other possible plans would include versions to bushy plans. The determination of the best plan to be executed in this scenario is a difficult task considering the great variations in the answer time to the sites involved. Purely static strategies aren't capable to model such variations. Adaptive strategies already can combine several possible plans during the execution of a query. For this query example, AQEE will execute the QEP equivalent to the $O_1 \cup O_2 \cup O_3 \cup O_4 \cup O5 \cup O_6$.

The Figure 11 illustrates the meta-QEP of the query above, consisting of the relational operators Scan, BindAccess (Ba), BindJoin (Bj) [13] and Project, and the modules *Fixed, Demand-drive, Wait, Adaptive,* and *First-Tuple.* The final QEP produced

by the QEEF's meta-engine is showed in the Figure 12, through the object diagram, composed only by algebraic and control operators.

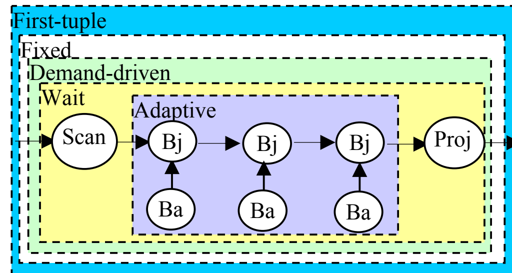


Fig. 11. Example of QEP using Adaptive Execution Model

The combination of modules in that meta-QEP occurs in the follow manner: The *Fixed* module contains the Scan operator (associated to L1,) the Project operator, and the *Adaptive* module that contains the three *BindJoin* operators that uses the correspondent BindAccess operator to submit *bindings* to the sites S1, S2, and S3, respectively. The *distributor* operator manages the adaptability of the tuples, putting them into a queue. Each tuple has a priority that is incremented every time its return to the queue. The tuple with highest priority is sent to output as soon as possible.

The Figure 13 shows the Java implementation of the AQEE's application main method, in which calls the following QEEF methods: Open (to initialize the engine and the data sources listed in the metabase), Execute (to submit a QEP specified in XML), Hasnext (to produce a tuple), Getnext (to get the tuple), and Close (to finalize all operations). It's possible to re-execute the same QEP by calling the Reexecute method.

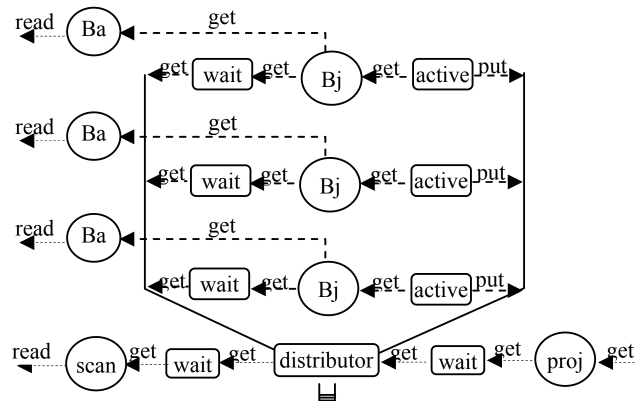


Fig. 12. Operators of the QEP, represented by the object diagram

```
public static void main (String[] args)  {
          AQEE engine = new AQEE();
          engine.open("Metabase.txt");
          engine.execute("Plan.xml");
          while (engine.hasnext()){
             Tuple t = engine.getnext();
             System.out.println(t.toString());
          }
          engine.close();
}
```

Fig. 13. The application main method

6 Conclusions

In this work, we presented an extensible query execution engine (QEE), named QEEF, to support new query execution models. We defined a query execution model as being a combination of execution modules, which is a high level operator that inserts some control operators into the algebraic query execution plan (QEP). The QEEF architecture was obtained using a software framework technique, very used in the construction of systems with high flexibility and that can be adapted rapidly to attend the new application requirements, being, therefore, adequate to our context. We develop a case study simulating a web data integration application, where the adaptability is associated to the unpredictability of the access time to these data. Applications as those present challenges as the access time to data sources register variations and there is no statistical information.

The main contributions of this work are: an analysis of the different execution scenarios, modules and models; the QEEF specification and its meta-model QUEM that makes possible the combination of modules in a QEP; and the AQEE case study. Besides that, we do not find in literature an extensible QEE with a support to new execution models.

As future work, we will instantiate the QEEF to new case studies such as a combination of Relational and XML data models in a QEP. The use together of these models has particular importance in applications of web data integration (like [12, 23]).

References

1. S. Abiteboul, P. Buneman, and D. Susciu. Data on the Web. Morgan Kaufman, 1999.
2. L. Amsaleg, M. J. Franklin, A. Tomasic and T. Urhan, Scrambing Query Plans to Cope with Unexpected Delays, In Proc. PDIS Conf., Miami, USA, 1996.
3. R. Avnur, J. Hellerstein, Eddies: Continuously Adaptive Query Processing, In Proc. ACM Intl. Conf. on Management of Data (SIGMOD), Dallas, Texas, 2000.

4. F. Ayres, et al, Um Framework para Construção de Máquinas de Execução de Consultas Relacionais, In Proc. Congresso Argentino de Ciencias de la Computacion - CACIC´02, Buenos Aires, Argentina, 2002.
5. J. Bercken, et al. XXL- A Library Aproach to Supporting Efficient Implementations of Advanced Database Queries. In Proc. Intl. Conf. on Very Large Databases (VLDB), Roma, Italy, 2001.
6. L. Bouganim, F. Fabret, C. Mohan, P. Valduriez, Dynamic Query Scheduling in Data Integration Systems. In Proc. 16o ACM. Intl. Conf. on Data Engineering (ICDE), San Diego, CA, 2000.
7. L. Bouganim, F. Fabret, F. Porto e P. Valduriez, Processing Queries with Expensive Functions and Large Objects in Distributed Mediator Systems, In Proc. 17o ACM. Intl. Conf. on Data Engineering (ICDE), Heidelberg, Alemanha, 2001.
8. S. Babu, and J. Widom, Continuous Queries Over Data Streams. ACM SIGMOD Record, September, 2001.
9. S. Chandrasekaran, et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In Proc. Intl. Conf. on Innovative Database Research (CIDR), 2003.
10. J. Chen, et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In Proc. ACM Intl. Conf. on Management of Data (SIGMOD), 2000.
11. A. Dobra et al. Processing Complex Aggregate Queries over Data Streams. In Proc. ACM Intl. Conf. on Management of Data (SIGMOD), p61-73, Madison, Wisconsin, USA,2002.
12. M. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima and W-C. Tan. SilkRoute: A Framework for Publishing Relational Data in XML. In Transactions on Database Systems (TODS), v27, n4, p438-493, 2002.
13. D. Florescu, I. Monolescu, A. Levy and D. Suciu, Query optimization in the Presence of Limited Access Patterns, In Proc. ACM Intl. Conf. on Management of Data (SIGMOD), Filadelfia, USA, 1999.
14. M. Fayad, D. Schmidt, and R. Johnson, Building Application Frameworks – Object-Oriented Foundations of Frameworks. John Wiley & Sons, Inc. 1999.
15. E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns, Addison-Wesley pub., 1995.
16. G. Graefe, and W. McKenna. The Volcano Optmizer Generator: Extensibility and Efficient Search. In Proc. ACM Intl. Conf. on Database Engineering (ICDE)., Vienna, 1993.
17. G. Graef, Query Evaluation Techniques for Large Databases, In ACM Computing Surveys, 25(2), 1993.
18. Z. G. Ives, D. Florescu, M. Friedman, A. Levy and D. Weld, An Adaptive Query Execution System for Data Integration, In Proc. ACM Intl. Conf. on Management of Data, Filadelfia (SIGMOD), USA, 1999.
19. V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams, In Proc. ACM. Intl. Conf. on Data Engineering (ICDE), 2002.
20. D. Kossmann, The State of the Art in Distributed Query Processing. In ACM Computing Survey, v32, n4, 2000.
21. S. Madden et al. Continously Adaptive Continuous Query over Streams. In Proc. ACM Intl. Conf. on Management of Data (SIGMOD), p171-183, Madison, Wisconsin, USA, 2002.
22. J. McCann. The Database Machine: Old Story, New Slant. In Proc. Intl. Conf. on Innovative Database Research (CIDR), 2003.
23. Manolescu, D. Florescu., D. Kossmann, F. Xhumari and D. Olteanu. Agora: Living with XML and Relational. In Proc. Intl. Conf. on Very Large Databases (VLDB), Cairo, Egypt, 2000.