Implementation of Parallel and Distributed Query Processing in the AutoMed Heterogeneous Data Integration Toolkit

Dimitris Fourkiotis

MSc Web Information Management project report

School of Computer Science and Information Systems

Birkbeck, University of London

September, 2007

Supervisor Alexandra Poulovassilis

Contents

List of Figures

Abstract

AutoMed is a heterogeneous data transformation and integration system, capable of handling virtual, materialized and indeed hybrid data integration across multiple data models. Queries submitted in a global schema are transformed and evaluated against a set of data sources, exploiting the functionality of the various components of the AutoMed Global Query Processor (AQP). Our motivation for this project is to enhance the AQP functionality, adding parallel and distributed query processing, in order to improve its overall performance.

Keywords: {*data integration, BAV, AutoMed, IQL, functional languages, parallel query processing, distributed query processing*}

Chapter 1

Introduction

Heterogeneous data integration can be described as the process of combining data residing at different data sources and conforming to possibly different data models, under an integrated schema. As the volume and the need to share existing data explode, data integration appears with increasing frequency. Integrating heterogeneous data sources under one query interface is not something new and the scientific community has proposed different solutions to this problem.

One approach is *data warehousing* [14], where data from several data sources are extracted, transformed and loaded into a single database and can be queried under a single interface. As a result, a data warehouse offers good query processing performance at the cost of maintaining the warehouse and returning stale data to user queries.

Another approach, trying to loosen coupling between data, is to provide a query interface over a *mediated schema (virtual database)*, with the data residing in the original data sources. Queries are then transformed into specialized queries over the original databases.

We can consider each of the data sources to be a view over the mediated schema (LAV approach) or we can express the mediated schema as a view over the sources (GAV approach). Depending the approach used to define the views between the mediated and the data source schemas, different algorithms must be used to rewrite queries over the data sources. A comprehensive survey on answering queries using views is presented in [12].

Comparing GAV and LAV from the point of view of query processing, it is well known that processing queries in LAV approach is a difficult task. In general, the complexity of query rewriting in LAV is NP-complete, but with a relatively small space of rewrites this is not a problem for integration systems [12]. On the other hand, processing queries in GAV approach is based on a simple unfolding strategy. This is because the mapping directly specifies the source query which corresponds to the elements of the mediated schema.

AutoMed is a heterogeneous data transformation and integration system, which offers the capability to handle virtual, materialized and indeed hybrid data integration across multiple data models. It supports the Both-as-View (BAV) data integration approach, capturing all the information that is present in GAV and LAV derivation rules. Queries submitted to a global schema are transformed and evaluated against a set of data sources, exploiting the functionality of the various components of the AutoMed Global Query Processor (AQP). This project enhances the AQP functionality, adding parallel and distributed query processing.

The rest of this thesis is organized as follows: Chapter 2 is a short introduction to the AutoMed framework, discussing its most important aspects. Chapter 3 discusses our approach for parallelizing the query evaluation components of the AQP. Since AutoMed is implemented using Java, Chapter 4 gives an introduction to Java threads to the extent necessary for this thesis. Chapter 5 discusses the implementation details for parallelizing the query processor and Chapter 6 presents a performance evaluation of the new evaluator. Chapter 7 presents the design and implementation details for distributed query processing, and Chapter 8 gives our concluding remarks and discussion for future work.

Chapter 2

The AutoMed framework

2.1 The AutoMed approach to data integration

In data integration [19], different databases are integrated to form a single virtual database, conforming to an associated global schema. The two common data integration approaches are the local as view (LAV) and the global as view (GAV).

AutoMed[1] supports a new approach to data integration [24] called *both as view (BAV)*. In this approach the integration of schemas is specified as a sequence of bidirectional transformation steps incrementally adding, deleting or renaming constructs, in order to map one schema to another. It can be shown that GAV and LAV rules can be derived from a BAV pathway [24]. A key advantage of BAV over GAV and LAV approaches is that it readily supports the dynamic evolution of both local and global schemas.

Another feature of AutoMed is that it is not restricted to one modelling language as its *Common Data Model (CDM)*; instead it makes use of a low level *hypergraph data model (HDM)* to express constructs of higher level models. AutoMed supports a functional query language as its *intermediate querying language (IQL)*, although it would be possible to use any other query language, for example SQL or XQuery.

The rest of this chapter discusses the core components of the AutoMed toolkit. Section 2.2 talks about the hypergraph data model, gives an example of mapping a simple relational model into HDM and presents the set of

[1]The AutoMed project is a research project, jointly run by Birkbeck and Imperial Colleges. Software and documentation are available from the AutoMed website `http://www.doc.ic.ac.uk/automed/`.

primitive transformations used to map one schema to another. Section 2.3 introduces the intermediate query language and Section 2.4 the AutoMed architecture. Finally, Section 2.5 explains how query processing is done.

2.2 The HDM data model

The basis of AutoMed integration toolkit is its low level hypergraph data model (HDM) [22, 29]. HDM defines a low-level modelling language which is based on a hypergraph data model, together with a set of constraints. There are facilities provided which can be used to define constructs of higher level models in terms of the HDM. An HDM schema consists of some combination of nodes, edges and constraints.

A *schema* in the HDM is a triple $\langle Nodes, Edges, Constraints\rangle$. A *query* q over *a schema* $S = \langle Nodes, Edges, Constraints\rangle$ is an expression whose variables are members of $Nodes \cup Edges$. $Nodes$ and $Edges$ define a labelled, directed, nested hypergraph. It is nested in the sense that edges can link any number of both nodes and other edges. $Constraints$ is a set of boolean-valued queries over S. Nodes are uniquely identified by their names. Edges and constraints have an optional name associated with them.

Constructs of any higher level modelling language M can be classified as either *extensional constructs* or *constraint constructs*, or both. Extensional constructs represent sets of data values from some domain. Each such construct in M must be built using the extensional constructs of HDM. The kinds of extensional constructs are:

- *nodal* constructs may be present in a model independent of any other constructs. The *scheme* of each construct uniquely identifies the construct. An example of a nodal construct in a higher level model, would be a *table* in the relational model.

- *linking* constructs associate constructs with each other and can only exist when these constructs exist. Linking constructs map into edges in HDM. A *relationship* in the ER model would be an example of a linking construct.

- *nodal-linking* constructs are nodal constructs that can only exist when certain other constructs exist, and that are linked to these constructs. Nodal-linking constructs map into a combination of a node and an edge in the HDM. An example of a nodal-linking construct would be an *attribute* in a relationship.

Finally, constraint constructs represent restrictions on the extents of the extentional constructs of M.
After the definition of a modelling language M in terms of the HDM, a set of primitive transformations is available for transforming schemas defined in M.

2.2.1 Transformation pathways in AutoMed

As discussed above, each construct of a modelling language M must be expressed in terms of HDM constructs. Once the constructs of M have been defined in this manner, mappings between schemas expressed in M can be described as a *pathway* of *primitive transformation* steps, each adding, deleting or renaming one construct of M. There exist five types of primitive transformations for transforming schemas:

1. $addC(\langle\langle s\rangle\rangle, q)$ applied to a schema S produces a new schema S' that differs from S in having a new C construct identified by the scheme $\langle\langle s\rangle\rangle$. The extent of $\langle\langle s\rangle\rangle$ is given by the query q on schema S.

2. $extendC(\langle\langle s\rangle\rangle, q_l, q_u)$ applied to a schema S produces a new schema S' that differs from S in having a new C construct identified by the scheme $\langle\langle s\rangle\rangle$. The minimum extent of $\langle\langle s\rangle\rangle$ is given by query q_l and the maximum by query q_u.

3. $delC(\langle\langle s\rangle\rangle, q)$ applied to a schema S produces a new schema S' that differs from S in not having a C construct identified by the scheme $\langle\langle s\rangle\rangle$. The extent of $\langle\langle s\rangle\rangle$ may be restored by the query q on schema S'.

4. $contractC(\langle\langle s\rangle\rangle, q_l, q_u)$ applied to a schema S produces a new schema S' that differs from S in not having a C construct identified by the scheme $\langle\langle s\rangle\rangle$. The lower bound query q_l and upper bound bound query q_u on schema S' have the same semantics as for extend.

5. $renameC(\langle\langle s\rangle\rangle\langle\langle s'\rangle\rangle)$ applied to a schema S produces a new schema S' that differs from S in not having a C construct identified by the scheme $\langle\langle s\rangle\rangle$ and instead having $\langle\langle s'\rangle\rangle$, differing from $\langle\langle s\rangle\rangle$ only in its textual name.

In addition to the above primitive transformations, AutoMed uses the *id* transformation to state the equivelance of two constructs in two different schemas. *Id* transformations can be used to define the integration semantics.

Integration semantics define the way data from separate data sources are combined to form the extent of global schema constructs. The following options are provided: (i) *choose* semantics, where values from one data source are returned; which data source is chosen depends on the `ChooseOptimiser` or, if optimization is not done, the `choose` function of the IQL language; (ii) *append* (++) semantics, where all values from all data sources are returned in order; (iii) *intersect* semantics where the common data values from the data sources are returned; and (iv) *union* semantics where all *distinct* data values are returned.

For example in Figure 2.1, if each local schema LS has a construct A which also appears untransformed in the global schema GS, then a query on A over the GS would return an append (++) of the extends of A in LS_1, LS_2 and LS_3.

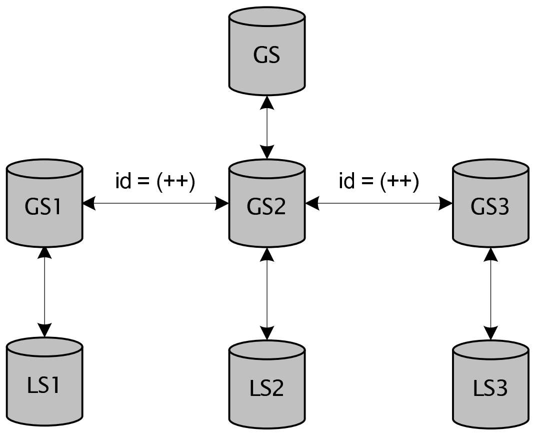


Figure 2.1: Defining integration semantics using *id* transformation.

2.2.2 A relational model represented in HDM

The following example aims at better understanding the usage of the HDM, by showing how a simple relational model can be mapped into the hypergraph data model. Given the relation $course(\underline{id}, cname, \#semesterID)$, where $\#semesterID$ is a reference to a foreign key, this would mapped into a relation construct $\langle\langle course\rangle\rangle$, three attribute constructs $\langle\langle course, id\rangle\rangle$, $\langle\langle course, cname\rangle\rangle$ and $\langle\langle course, semesterID\rangle\rangle$, a primary key construct $\langle\langle course\_pk, course, \langle\langle course, id\rangle\rangle\rangle\rangle$ and a foreign key construct $\langle\langle course\_fk, course, \langle\langle course, semesterID\rangle\rangle, semester, \langle\langle semester, semID\rangle\rangle\rangle\rangle$, assuming there is also a relation $semester(\underline{semID})$.

2.3 The IQL query language

IQL [27, 28] is a typed, functional language. IQL provides a common query language that queries written in higher level languages can be translated into and out of. This section first introduces functional programming languages and then presents the IQL.

2.3.1 Functional programming languages

Functional programming is so called because a program consists of a series of function applications, each of which receives input through its argument(s) and delivers output via its result[2].

The key characteristics of functional languages are summed up as follows. Functional programs *do not contain assignment statements*, so variables, once bound to a value, never change. Functional programs contain no side effects: a function call can only compute its result. This eliminates a major source of bugs, and also makes the order of execution irrelevant[3], since no side effects can change the value of an expression. This freedom helps make functional programs easier to analyze and optimize than their imperative counterparts.

Another fundamental advantage of functional programming is *modularization.* There are two features in functional languages that aid modularization: the first is the ability to write higher order functions on data structures (e.g. *flatmap* on collections[4]) and the second is *lazy evaluation.* Lazy evaluation is the technique of delaying a computation until such time as the result of the computation is known to be needed [9]. For example, consider the case where a function f takes as input the output of another function g. This

[2]In this context, functions are like simple mathematical functions and can be defined using equations.

[3]Also known as *referential transparency.*

[4]Function flatmap applies a function to each item of a collection and concatenates the results. For more details see Section 3.3.

could be implemented by computing the output of function g, storing it in memory, and then passing it to f. The problem with this approach is that the result of g may occupy so much memory, that the two functions cannot be composed. Since with lazy evaluation argument(s) of a function are evaluated as needed, we can compose the two functions, with the output from g being evaluated incrementally only as it is needed by f. This allows us to incrementally process infinite lists (also called *streams*) without infinite loops or size matters interfering in the computation.
For more details on functional languages the reader is referred to [13].

2.3.2 Datatypes, variables & functions

IQL supports integer and float numbers, strings (enclosed in single quotes) and datetime objects. Integers and floats are primitive Java types whereas strings and datetime objects are Java String objects. Tuples (i.e. $\{1, 2, 3\}$), lists (i.e. $[1, 2, 3]$), sets and bags are also supported. The tuple, list, set and bag constructors can be arbitrarily nested. Throughout the rest of this thesis, we assume list semantics, except for where explicitly otherwise stated.
Variables and functions are represented by identifiers starting with a lowercase character. A number of primitive built-in functions are available, which can be easily extended, adding new abilities to the language. New, anonymous, functions can be defined using *lambda abstractions*. For example:

$$lambda\ \{x, y, z\}\ ((*)\ ((+)\ x\ y)\ z)$$

defines a function which takes a triple, adds the first two components and multiplies by the third one.
IQL also supports *variable unification* (variables having the same name, are treated as implicit joins); i.e. the following query evaluates to $[\{3, 5\}]$:

$$[\{a, c\} \mid \{a, b\} \leftarrow [\{1, 2\}, \{3, 4\}];\ \{b, c\} \leftarrow [\{4, 5\}, \{6, 7\}]]$$

2.3.3 Higher level constructs

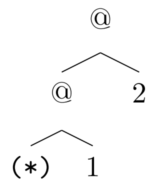
IQL supports *let* expressions and list, bag and set *comprehensions*.
let expressions assign an expression to a variable and this variable can then be used within other expressions. For example, the query

$$let\ f\ =\ ((*)\ 100\ 200)\ in\ ((+)\ f\ f)$$

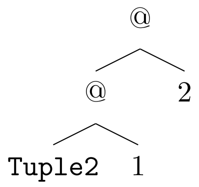
returns 20000 + 20000 = 40000.
Comprehensions are of the form $[h \mid q_1;...q_n]$ where h is an expression termed the head and $q_1...q_n$ are *qualifiers*, with $n \geq 0$. A qualifier may be either a filter or a generator. Generators are of the form $p \leftarrow e$ and iterate a *pattern* p over a collection-valued expression e. A pattern may be either a variable or a tuple of patterns. Filters are boolean-valued expressions that act as filters on the variable instantiations generated by the generators of the comprehension.
A description of how comprehensions are internally translated into applications of the `flatmap` operator, is given in Section 3.3 below.

2.3.4 Abstract representation

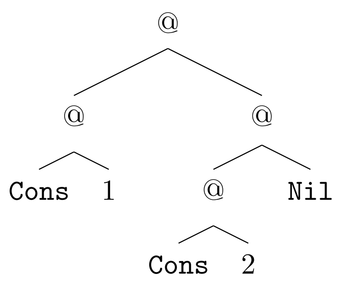
IQL queries in AutoMed are represented internally using a full binary *directed acyclic graph (DAG).* All non-leaf cells can be either apply cells (@) or lambda cells (λ). An apply cell denotes the left child being applied to the right one. For example the query $(*)$ 1 2 is represented as:



Leaf cells may be constants, variables or function names. They may also be *constructors* of tuples (`Tuple1, Tuple2,...`), lists (`Cons`), bags (`BCons`) and sets (`SCons`). For example, the following tree represents tuple $\{1,2\}$:



Lists are represented using the general form (*Cons head tail*), where *head* is the first element and *tail* is the rest of the list. An empty list is represented using the `Nil` constructor. For example, the list $[1,2]$ is represented as follows:



The above internal abstract representations are implemented by the *abstract syntax graph (ASG)* class in the AutoMed toolkit. The various components of the AutoMed handle ASG representations of queries.
More details on the internal representation of IQL queries as an abstract syntax tree and examples of the Query Processor API can be found in [15].

2.3.5 An IQL example

Consider the schema constructs discussed in Subsection 2.2.2. The following query:

$$[\{y\} \mid \{x, y\} \leftarrow \langle\langle course, cname \rangle\rangle]$$

returns the name of each course and this one:

$$gc\ count[\{y, x\} \mid \{x, y\} \leftarrow \langle\langle course, semesterID \rangle\rangle]$$

first groups on *semesterID* (*gc*) and then counts (*count*) the number of courses in each semester.

2.4 The AutoMed architecture

The AutoMed toolkit consists of several components, as illustrated in Figure 2.2[5]. The query processor is responsible for processing queries and will be discussed in the next section. The *schema matching tool* can be used to identify related objects in various data sources and the *XML schema transformation tool* can generate transformation pathways from data source schemas to a global schema. A *GUI* is also available for interacting with these components, as well as with the Model Definition Repository (MDR) and Schema Transformations Repository (STR) components of the AutoMed metadata repository.

[5]Adopted from [21].

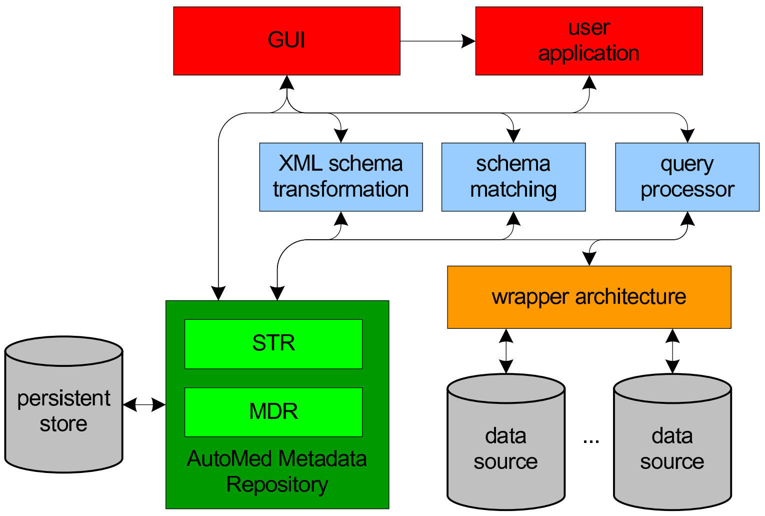


Figure 2.2: The AutoMed architecture.

2.4.1 The AutoMed Metadata Repository

The AutoMed metadata repository [6] forms a platform for other components of the AutoMed to be implemented upon. The current implementation uses a RDBMS to provide persistent storage for data modelling language descriptions in the HDM, database schemas and transformation pathways between schemas.

The AutoMed repository has two main logical components that can be accessed via the AutoMed API. The *Model Definitions Repository (MDR)* is used to describe how a data modelling language is represented as combinations of nodes, hedges and constraints in the HDM. It is used to configure AutoMed so that it can handle a particular data modelling language. The *Schema Transformation Repository (STR)* is used to define schemas in terms of the data modelling concepts in MDR. It is also used to specify transformations between these schemas. AutoMed users will need to update this repository using the AutoMed API as new databases are added to the AutoMed repository, or these databases evolve [23]. The MDR and STR may be held in the same persistent storage, or separately.

2.5 Query processing in AutoMed

2.5.1 The AutoMed Query Processor

The *AutoMed Query Processor (AQP)* consists of several components and is used to evaluate queries submitted to a global schema against a set of data sources. Figure 2.3[6] illustrates the AQP architecture. The AQP in-

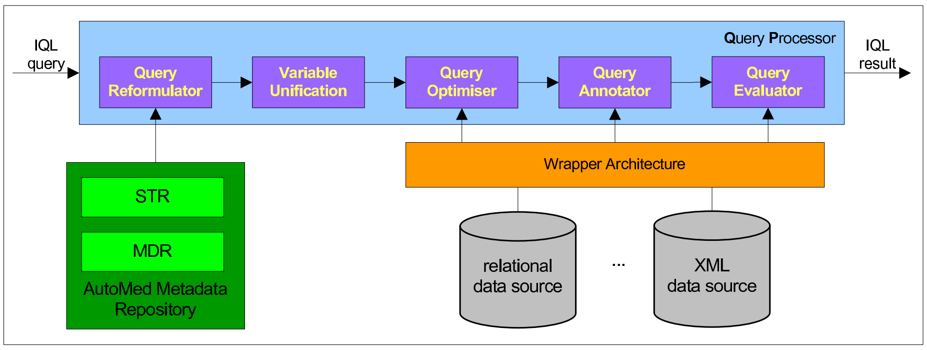


Figure 2.3: AutoMed Query Processor.

stance that will be used during the query process, can be configured using the `QueryProcessorConfiguration` component (see below).
An IQL query is first reformulated, according to the transformation pathways stored in the AutoMed repository, using the `QueryReformulator` component. The reformulated query, which now contains only data source constructs, is then optimized by the `QueryOptimiser` component. The `QueryAnnotator` component inserts AutoMed wrapper objects within the optimized query. The `QueryEvaluator` component is finally used to evaluate the annotated query. The rest of this section discusses the major aspects of the AQP, to the level of detail needed for this thesis.

2.5.2 Query Processor Configuration

The `QueryProcessorConfiguration` component allows the user to set a number of parameters that configure the internal mechanisms of the AQP. For example, the user can choose whether the GQP will perform query optimization or not, or choose which implementation will be used for each component, as each of the components of Figure 2.3 may have multiple

[6]Taken from [15].

implementations. For a detailed presentation of the existing options, see Section 3.4 of [15].

2.5.3 Query reformulation

`QueryReformulator` component is capable of reformulating queries submitted to a global virtual schema, using GAV, LAV or BAV reformulation.
When using GAV reformulation the query processor uses only those portions of BAV pathways that define virtual schema constructs in terms of data source constructs (`delete, contract` and `rename` transformations). When using LAV, the query processor uses only the portions of BAV pathways that define data source constructs in terms of virtual schema constructs (`add, extend` and `rename` transformations). In BAV reformulation it uses all the available information contained in BAV pathways.

2.5.4 Logical optimization

After the reformulation of the initial query into a new one, which only contains data source schema constructs, the `QueryOptimiser` component performs various optimizations at the logical level. This process has two goals: first, to simplify the query by performing algebraic optimizations, and second, to build the largest possible subqueries that can be pushed down to the local data sources (see [15] for more details of query optimization in AutoMed).

2.5.5 Query annotation

After the reformulation and optimization of the query, *wrapper objects*, responsible for evaluating IQL queries, are inserted within the query. The `QueryAnnotator` component traverses the tree representation of the query and identifies the largest possible subquery that can be pushed down to the data sources. The ability of a wrapper to identify queries which it is capable of evaluating relies on a parser associated with each wrapper.

2.5.6 Query evaluation

After the query has been annotated, the evaluation process takes place. Evaluation is performed by a `QueryEvaluationProvider` instance.
Evaluating a query expressed in a functional language consists of performing

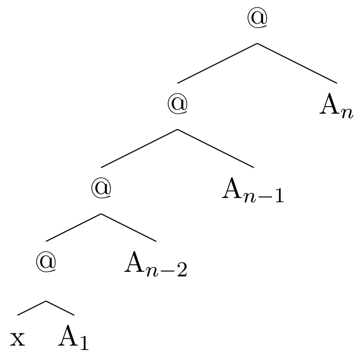
reductions on reducible expressions until no more reductions are applicable. It is then in the so called *normal form*. The order in which these reductions are performed makes no difference to the results of the query.

The `QueryEvaluator` component

When a `QueryEvaluator` component is instantiated, it is passed a `Query ProcessorConfiguration` instance. As a result, it has access to a `Function Table` instance which contains the set of IQL built-in functions that are to be used for the evaluation.
Regarding reducing expressions the `QueryEvaluator` component always reduces the leftmost outermost reducible expression first; this is known as *normal-order* reduction. In normal-order reduction, function arguments are not evaluated unless they are actually needed for the function to return a result.
The `QueryEvaluator` component contains methods for full evaluation (reduction to *normal form*) and partial evaluation (reduction to *weak-head normal form*). For furhter details on *normal form* and *weak-head normal form* see [16].

The query evaluation flow

The entry point for evaluation is the `evaluate(...)` method. This calls the `weakHNF(...)` method, which, starting from the root of the subtree passed to it as parameter, goes to the tip of the spine, passing over aplly (@) cells, until it finds a *special*, or a *function*, or a *lambda* cell. The following tree represents a spine to which arguments (A_1-A_n) are attached, with x being a special, or a function, or a lambda cell.



If the cell is a special cell (e.g. a *generator*) or a built-in function cell,

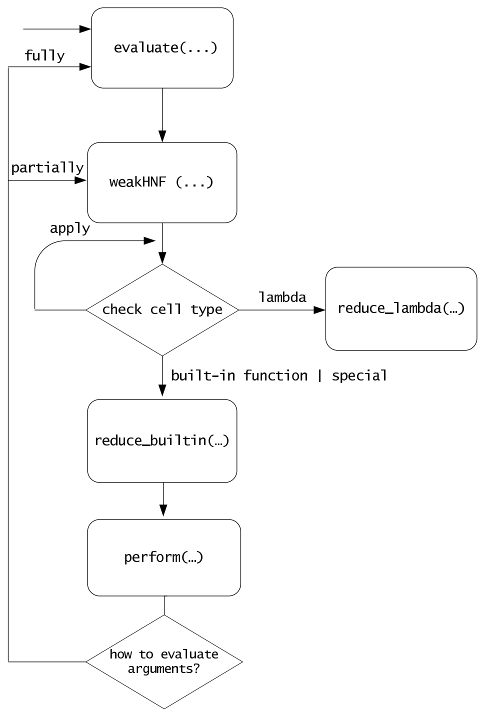


Figure 2.4: Query evaluation flowchart.

weakHNF calls the reduce_buitin(...) method to evaluate the function or the special cell accordingly. If it is a lambda cell, weakHNF calls the reduce_lambda() method to evaluate the lambda expression.
When the evaluate method eventually returns, the whole tree is fully evaluated. weakHNF method does not guarantee this, so the arguments of the spine should be again checked by evaluate after weakHNF returns; if they include a function or a constructor the method weakHNF is called again. Through these iterative calls to weakHNF, eventually the whole tree representing the query is fully evaluated.

Chapter 3

A parallel approach to IQL query evaluation

3.1 Introduction

The current implementation of the `QueryEvaluator` component takes a serial approach to query evaluation. For example, suppose we have the simple IQL query:

$$\langle\langle ... \rangle\rangle \; intersect \; \langle\langle ... \rangle\rangle$$

which intersects two schemes (naming details make no difference for this example). This query is internally translated into calls to a number of built-in functions, but we will focus on the `intersect` function. `Intersect` is a binary function which evaluates its arguments and outputs the common elements. The current implementation of the `QueryEvaluator` evaluates the first argument (which includes a call to a `Wrapper` object for retrieving the data), then evaluates the second argument (which, in our case, is also a call to a `Wrapper` object) and finally the `intersect` function applies its own logic.
However, the arguments of the `intersect` function could be evaluated in parallel, speeding up the whole process. The benefits of parallel execution are more obvious when instead of having simple operations as the arguments of a function, these contain nested functions further down.

3.2 Parallelizing the query processor

A first question to ask when it comes to parallelizing the AutoMed Query Processor is what should be parallelized? Is it enough to parallelize the `QueryEvaluator` component or should other components of the AQP also be parallelized? We argue that `QueryReformulator`, `QueryAnnotator` and `QueryOptimiser` components all perform operations that, if parallelized, will not significantly speed-up query processing. The only component that will offer a notable performance speed-up if it is parallelized is the `QueryEvaluator` component and more precisely the evaluation of functions with more than one argument.
The goal of this parallelization is of dual nature: first, we expect to obtain a speed-up due to parallelism itself; second, after integrating parallelization with *incremental query processing*[1], a drop in the time in which the first result is presented to the user may be achieved.

3.3 The IQL built-in functions

This section presents the IQL built-in functions, focusing on their potential for parallelization (see [15] for a detailed list of IQL functions).
While reading this section, the reader should have in mind that only functions with arity greater than one will be parallelized.

- **Arithmetic** functions: this group contains functions like `Add`, `Divide`, `Multiply` and so on. One could argue that the simplicity of these functions does not justify their parallelization. However consider the following IQL query: *gc count*[...] + *gc count*[...]. It is clear that it may be beneficial to evaluate the two arguments of the + function in parallel, using two threads. Our implementation gives the ability to the user to select if arithmetic functions are to be parallelized via a "threading level mechanism" (discussed in Section 3.5).

- **Collections** functions: this group contains functions whose arguments are collections, like `Append`, `Average`, `Choose`, `Count`, `Flatmap`, `Fold`, `Foldl`, `Foldr`, `Intersect`, `Map`, `Max`, `Min`, `Member`, `Union`, `Monus`, `Group`

[1]This is an on going effort for incrementally returning results as soon as they are available and not waiting until the complete result has been constructed. This work is being done independently of this thesis.

and `Sum`. The general approach to parallelizing these functions is to create as many threads as the collection-valued arguments and then evaluate each such argument using a separate thread. (Therefore, only functions that have more than one collection-valued argument will be parallelized.)

- **Comparison** functions: this group contains the comparison functions, like `Less` and `Equals`, with parallelization of them being straight forward, as with the arithmetic functions.

- **Date** functions: this group contains the two functions `getMonth` and `now`. The first function returns the current month and is of arity one (thus no parallelization); and the second one, returns a date time representation and is of arity zero (also no parallelization).

- **Internal** functions: this group includes functions `Comprehension` and `CallToWrapper`. `CallToWrapper` is responsible for submitting a query to a data source for evaluation. This function is not parallelized because we cannot intervene in the internal query processing of the data source.
  A comprehension is internally translated by the built-in function `Comprehension` according to following equivelances, for subsequent evaluation:

$$\begin{array}{ll} [e \mid p \leftarrow s; Q] & \Longrightarrow\ flatmap\ (lambda\ p\ [e \mid Q])\ s \\ [e \mid e'; Q] & \Longrightarrow\ if\ e'\ [e \mid Q]\ [\,] \\ [e \mid] & \Longrightarrow\ [e] \end{array}$$

  The *if* function takes three arguments and returns the second if the first is true, otherwise the third. The *flatmap* function, when it operates on lists, is defined as follows:

$$\begin{array}{lcl} flatmap\ \ f\ [\,] & = & [\,] \\ flatmap\ \ f\ (Cons\ x\ xs) & = & (f\ x)\ ++\ (flatmap\ f\ xs) \end{array}$$

  Serial execution of comprehensions follows the above translation. In the first step, a comprehension is translated into a combination of the *flatmap* and *lambda* functions. Due to this translation, when a comprehension is evaluated, a nested-loops evaluation strategy is adopted over the collections in its generators.

In parallel execution, we instead evaluate all generators of a comprehension expression in parallel, and then translate into the flatmap/ lambda representation and proceed with the evaluation as with serial execution. As a result we have a performance speed-up in comprehension evaluation, as shown by the experimental results (see Chapter 6).

- **Logical** functions: this is the group of logical functions of IQL. Although some of these functions, like `And` and `Or`, are appropriate for parallelization, we have not parallelized them. Suppose we have the following logical expression:

  $$(x \ and \ (y \ or \ z))$$

  Suppose also that we have parallel execution of these functions. If, let's say the thread executing x ends, evaluating to $false$, before threads executing y and z end, then they should stop executing as the whole expression will evaluate to $false$. This would require to have access to these two threads, something that is not provided by the `ThreadPool Executor` class we are using.

- **Other** functions: this group contains several general-purpose functions. All functions in this group are of arity one and thus they are not parallelized.

- **String** functions: this group contains string associated functions. Except those that are of arity one, the rest are parallelized.

- **TypeConversion** functions: this is a collection of functions which allow converting between different AutoMed types (`List2Bag`, `Set2Bag` and more). None of these functions is parallelized as all are of arity one.

3.4 Parallel functional languages in practice

When it comes to parallelizing a functional language, it may be sometimes necessary to diverge from the purely lazy functional model. Indeed, in [11] the author argues that parallel functional languages should be strict[2]. Lazy

[2]In strict evaluation, arguments of a function are evaluated before it is called. In contrast, in lazy evaluation, arguments of a function are passed to the function unevaluated and the function itself determines when they are to be evaluated.

evaluation is sequential; in contrast, parallel programs arrange computation on multiple processors and thus require some eager evaluation. For example, consider our parallelization of comprehensions. Arguments of the `Flatmap` function are evaluated before the calls to the functions themselves. This is eager evaluation, improving performance in parallel execution, at the expense of laziness.

Tuning the parallel performance of such programs may also require some code restructuring. In [20] the authors introduce a *clustering* technique for the parallelization of functions with collection-valued arguments. A naive approach would fully evaluate each collection argument of the function in parallel, yielding very fine task granularity. Clustering improves task granularity by introducing fewer tasks, each operating on a subset of the collection. For example, consider the `Flatmap` function defined in Section 3.3, which applies a function f to each element of a collection. Using a different thread to apply f to each item of the collection would delay query evaluation, due to the cost of managing the different threads and is an example of fine task granularity[3]. Using a clustering technique `Flatmap` would use a different thread for multiple items of the collection.

We have adopted an intermediate approach: parallelizing the evaluation of comprehension generators as described in Section 3.3; but then not further parallelizing the application of the flatmap function (since they have one collection-valued argument).

3.5 Implicit and explicit programming models

The basic problem in a parallel program is that, in addition to specifying *what* value the program should compute, it is necessary to know *how* the machine should organize the computation. One approach is to rely on the run-time system to manage the parallel execution without any programmer input (*implicit* approach). Another is to delegate most management tasks to the runtime system, but allow the programmer the opportunity to *give advice* on a few critical aspects (*explicit* approach).

The *Glasgow Parallel Haskell (GpH)*[4] [30] language follows an intermediate approach between purely implicit and purely explicit approaches. The runtime system manages most of the parallel execution, only requiring the programmer to indicate those values that might be useful to be parallelized.

[3]The current implementation evaluates `Flatmap` function serially.

[4]See `http://www.cee.hw.ac.uk/~dsg/gph/` for more information on GpH.

GpH uses *Haskell*[5] [17] as its computational language. Parallelism is introduced by the `par` combinator, which takes two arguments that are to be evaluated in parallel. The expression `p 'par' e` indicates that `p` could be evaluated by a new thread, with the parent thread continuing evaluation of `e`. Operator `seq` indicates that two expressions must be evaluated serially. Consider for example the parallel computation of Fibonacci numbers. If `n` is greater than 1, then a new thread is created to compute `pfib(n-1)` and the parent thread continues to evaluate `pfib(n-2)`. The following is a code snippet illustrating the usage of `par` and `seq` combinators:

```
pfib n
  | n <= 1 = 1
  | otherwise = n1 'par' n2 'seq' n1+n2+1
                where
                   n1 = pfib(n-1)
                   n2 = pfib(n-2)
```

Our approach is a combination of the explicit and implicit approaches, like in GpH. We do not have special IQL syntax to indicate which arguments should be parallelized. Instead, we annotate cells of the DAG that represent built-in functions or roots of comprehensions with a *threading level*. There are a number of predefined threading levels and each level contains a different set of IQL built-in functions. The user then can choose between these levels, for example, when requests a query to be evaluated. Suppose we have the following IQL query:

$$[\ \{x, y\}\ |\{x\} \leftarrow \langle\langle scheme_1\rangle\rangle;\ \{y\} \leftarrow \langle\langle scheme_2\rangle\rangle\ ]$$

After reformulation and optimization this query will contain, among other functions, a call to `Comprehension` function. The user can choose the threading level that the evaluator operates, and thus if the `Comprehension` function is to be evaluated in parallel or not. For more details, see Chapter 5.

[5]Reference [31] is a comprehensive survey for parallel and distributed functional languages that are using Haskell as their computational language.

Chapter 4

Introduction to Java Threads

In this chapter we give a short introduction to Java Threads and more general topics concerning threaded programming to the extent necessary for this thesis. Section 4.1 explains what is a thread, benefits and problems that could appear. Section 4.2 discusses thread support in Java and Section 4.3 explains how to create and manage threads in Java. Section 4.4 discusses race conditions and synchronization issues, finishing with deadlocks and starvation phenomena (Section 4.5). Then follows a reference to thread performance, and more specific about how stack and heap size can affect performance (Section 4.6.1), as well as the role that the underlying Operating System and Java Virtual Machine play (Section 4.6.2). In Section 4.6.3 there is a comparison between thread pools and thread spawning on demand.

4.1 What is a thread, benefits and problems

A *thread* is an application task that is executed by a host computer. For example suppose that one wrote a program that performed two separate tasks: one calculated the factorial of a number and one calculated the root of that number. These are two separate tasks, so they could be written as two separate threads executing *in parallel.* Roughly speaking threads are a series of instructions that can be executed in parallel at multiple processors or at the same processor through *time slicing.*
Threads exist within a process and share memory space with their parent, so they all have access to class member variables (shared variables that are defined as members of a class and not being local to methods), whereas each

thread has its own program counter and stack. Multiple threads may run concurrently within the same process.
Threads provide light, efficient way for concurrent programming. When a single-threaded process blocks, the whole program hangs; when a thread blocks (i.e. on I/O), another can run. As a result, threads are an excellent choice for multitasking applications (Web servers, file servers). Less operating system (OS) resources are used up, i.e. memory, buffers, kernel data structures, and creating/destroying a thread is much cheaper than creating/destroying a process. On the other hand, threads increase programming complexity, for example non-deterministic behavior in terms of CPU scheduling (over multiple executions threads may be executed in different order), need to be synchronized, usually difficult to debug. There are also portability issues due to different implementations.

4.2 Java thread support

Upon the release of Java in 1995, Sun promoted the language as being among other things, robust, safe, architecture-neutral, portable, object-oriented, simple and threaded [10]. Although the two last seemed contradictory, it turns out that the Java threading system is simple, at least relative to other threading systems. In early versions of Java, this simplicity came with some trade-offs; some of the advanced features that are found in other threading systems were not available in Java. Java 2 Standard Edition changed this: a large number of thread-related classes (under the `java.util.concurrent` package) make the task of writing multithreaded programs much easier. Classes that support lock-free, thread-safe programming on single variables (`java.util.concurrent.atomic` package) are available, a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors (`java.util.concurrent.locks` package) and utility classes commonly useful in concurrent programming, are also available. This collection of classes has been introduced with JDK 1.5: for previous releases, third party classes that provide the same or similar functionality must be used.

4.3 Thread creation and management

Java threads can be created in two different ways: by extending the `Thread` class[1] or by implementing the `Runnable` interface[2]. Each class extending the `Thread` class must overload the `run()` method, adding its own logic; eventually when the new class is instantiated and the thread is created, it begins its execution from this method. The `run()` method can be thought as the `main()` method of a standalone Java application: the `main()` method is where the thread starts its execution. Below follows an example of creating a thread using the first method, adopted by [26]. In this example a new listener is associated with a start button and a new random character is generated whenever the associated event occurs.

```
public class RandomCharacterGenerator extends Thread{
      ...
      public void run(){
            for(;;){
                  nextCharacter();
                  try{ Thread.sleep(getPauseTime()); }
                  catch(InterruptedException ie){ return; }
            }
      }
}

public class SwingTypeTester{
      ...
      private void InitComponents(){
            ...
            startButton.addActionListener(new ActionListener(){
                  ...
                  producer = new RandomCharacterGenerator();
                  producer.start();
                  ...
            });
      }
      ...
}
```

Using the `Runnable` interface the programmer can separate the implementation of a task from the thread used to run the task. For example instead of

[1]By extending a class, the new one inherits the behavior and methods of the base class.
[2]An interface forms a contract between the class and the outside world.

using the `Thread` class, `RandomCharacterGenerator` could implement the `Runnable` interface and change the way in which the thread is constructed.

```
public class RandomCharacterGenerator implements Runnable{
       ...
      public void run(){
            for(;;){
                  nextCharacter();
                  try{ Thread.sleep(getPauseTime()); }
                  catch(InterruptedException ie){ return; }
            }
      }
}
public class SwingTypeTester{
      ...
      startButton.addActionListener(new ActionListener(){
            ...
            producer = new RandomCharacterGenerator();
            Thread t = new Thread(producer);
            t.start();
            ...
      });
      ...
}
```

Choosing between the two alternatives depends on whether the new class should inherit behavior from the `Thread` class or from other classes. The `Thread` class also provides a collection of methods that are not available by the `Runnable` interface.
A thread is considered to exist once it has been instantiated. Although it does not start its execution unless the `start()` method is invoked, other threads can interact with it. When the `start()` method returns, two threads are executing in parallel; the original thread (which has just returned from the `start()` method) and the newly started (which is executing the code in its `run()` method). When a thread finishes executing the `run()` method it exits and it is considered to be dead and ready for garbage collection.
There are more functions that allow a programmer to manage a thread, like `sleep()`, `wait()` and `yield()`. The first two suspend a thread untill it is notified by another one (they differ in the way they release thread's locks) and `yield()` removes the thread from the executing queue giving

the chance for other threads to be executed. For more details the reader is referred to [25]. Figure 4.1 shows the lifecycle of a thread.

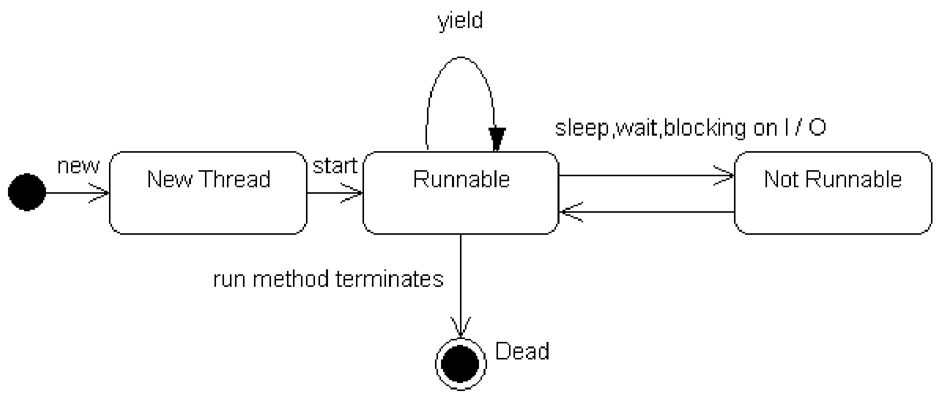


Figure 4.1: Thread's lifecycle.

4.4 Data synchronization and race conditions

As discussed in Section 4.1, threads share memory space with their parent, so they all have access to class member variables (shared variables that are defined as members of a class and not being local to methods). Sharing data between threads can be problematic due to what is known as a race condition. A *race condition* occurs every time more than one thread is trying to access shared variables and special programming techniques need to be used in order to ensure the right behavior of the program, and Java provides certain mechanisms that deal with this problem. The `synchronized` keyword is provided and is very similar to a *mutex lock*. The concept of *synchronization* is simple: when a method is declared to be synchronized, a thread that wants to execute the method must acquire a lock on the object containing the method. This can be done if and only if there is no other thread that holds a lock to that object. The acquisition and release of a lock is guranteed to be a mutual process by the JVM. Choosing to lock the whole method instead the block of code where the racing condition occurs is known as *coarse-grained synchronization* and can reduce the parallelization degree of a program. For example, assume there is a class defining two vectors and a method that among other things adds elements to these vectors. Instead of defining the whole method as synchronized and preventing concurrently execution of non mutual exclusive operations, one could only synchronize access to shared objects.

```
pubic class Example {
          ...
          Vector x;
          Vector y;

          private methodX(void){
                    ...
                    synchronized(x){ x.put(...); }
                    synchronized(y){ y.put(...); }
          }
}
```

Java also provides the `volatile` keyword to solve problems relating with the scope of a lock[3] and explicitly locking through the `Lock` interface.

4.5 Deadlocks and starvation phenomena

A *deadlock* occurs whenever two threads are waiting for a lock to be freed and the programming logic is such that the lock is never freed. Deadlocks between threads is one of the hardest problems to solve in any threaded program and is the responsibility of programmer to prevent them, as JVM does not take any action preventing such phenomena. A naive way to solve the problem would be: When a lock is held on an object never call a method that needs other locks on the same object, something that in reality is impractical as, for example, many useful Java classes are synchronized and one will want to use them from synchronized methods. another way would be to lock some higher-order object that is related to many lower-order objects we need to use, something that often leads to coarse-grained locking.

Whenever multiple threads compete for a resource, there is the danger of *starvation*, where the thread never gets the resource. The operating system and its approach to thread management can assist in avoiding or encouraging this problem. There are different preventing techniques. At the CPU

[3]There are cases where a lock is grabbed and never released, i.e. by synchronizing the `run()` method, causing deadlock phenomena.

level, all multi-threaded operating systems must allocate CPU cycles fairly and efficiently. Scheduling algorithms must schedule thread usage of the CPU. Each one has strengths and weaknesses, including the potential for starvation.

4.6 Thread performance

There are a lot of factors that might affect the performance of a multi-threaded program, like the stack or heap size, the underlying operating system, the JVM itself, synchronization issues and so on. Although these factors may affect performance, there are cases where developers believing that synchronization is inherently expensive, write complex code which is difficult to maintain and more prone to bugs than the simpler, synchronized, version. The rest of this section discusses some of the more crucial factors affecting performance.

4.6.1 Stack and heap size

The memory stack is where a thread stores its local variables, the program counter which indicates which statement in the method is currently executing and other internal information. All these together determine the size of the stack. The size is platform dependent; the space needed to store the local variables differs across various platforms, although the local variables must have the same size, and furthermore the various internal information is dependent on the Java implementation. The size of the stack impacts Java's memory usage, causing stack overflows or out-of-memory errors and when one is scaling to large number of threads (in the thousands) there is a significant waste of memory usage due to the quite large default stack size. The programmer can specify the stack size using the particular class constructor, but this can lead to unportable programs. Instead, one could specify the stack size for all threads passing the command line argument `-Xss` to the JVM. Java also provides a stack associated API [25].
Heap is a portion in memory where dynamically created objects are stored. The size of the heap can affect performance. As the heap gets full, the garbage collector is called, and this can cause performance related problems. The amount of heap size can be controlled via the `-Xms` and `-Xmx` command options.

To optimize the memory allocation, each thread has a dedicated area of the heap, known as TLH (thread local heap), and that thread can allocate from its local heap area. This technique works well for small objects and number of threads, but when the number of threads increases, the thread local heap can consume a significant amount of the heap, causing frequent calls to the garbage collector. For an introduction on how memory management can affect performance the reader is referred to [2] and [1].

4.6.2 Underlying OS and JVM

Java is inherently multi-threaded and because of this the underlying OS can significantly affect performance. Different operating systems show different performance for thread creation and synchronization, while faster chips affect execution time as well. Different supported thread models (one-to-one, many-to-one, many-to-many)[4] and different stack sizes are all factors depending on the operating system. The specific implementation of the Java Virtual Machine that is used and its configuration is also important. For example, synchronization performance is a factor that is strictly associated with the JVM implementation used. Two resources for JVM and how it is related to performance are [1] and [4] whereas in [3] one can find specific information on Java threading in Solaris.

4.6.3 Thread creation on demand vs. thread pools

A *thread pool* is a collection of previously created threads that sit idle until there is a task to perform. As the program has tasks for execution, it creates threads for them instantiating a `Runnable` object and passing it to the thread pool executor for execution. One thread that sits idle in the pool takes the task and executes it. As a thread pool might have fewer threads than tasks to execute, it might have to wait for an available thread to run it stored in a waiting queue. Java's implementation of a thread pool is provided through the `Executor` interface and it comes with two different kinds: the `ScheduledThreadPoolExecutor` and the `ThreadPoolExecutor`.
There are a number of reasons for choosing to use a thread pool, rather than creating threads on demand. First, there is a common assumption that the overhead of creating a thread is high and by using a pool we gain a lot in

[4]According to how user threads are mapped into OS threads.

performance. The degree to which this is true depends on the program. The extra time used to create a thread, usually has an upper of a few hundred microseconds [26], is usually not important for some programs. Second, using a thread pool the programmer can concentrate on the program logic rather than writing code for thread creation and manipulation. Finally, thread pools provide some performance benefits; one for example, could throttle the number of threads so they don't flood the system. Further details on thread pools in general and Java's support can be found in Chapter 10 of [26] and in [25] accordingly. Concluding, choosing between creating threads on demand and dynamically creating a thread depends upon the program and is subject to trade offs.

Chapter 5

Parallelization of the AutoMed Query Processor

This section describes in detail the implementation of the new evaluator regarding its parallelization. During the design phase, it was a high priority to maintain the modularity of the application and create a component that is easily maintainable and extendable. Whenever it was not possible to create a new class to encode the required logic, we tried to modify existing code as little as possible, keeping the two logics clearly separated.

5.1 `ParallelEvaluator` design

`ParallelEvaluator` is the new implementation of `QueryEvaluationProvider` for executing a query in parallel. Figure 5.1 shows the class diagram for the new component. It extends the `Evaluator` class, keeping the same

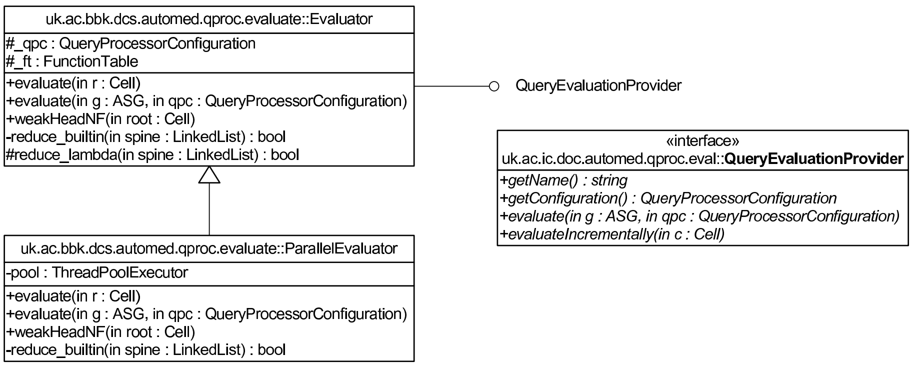

uk.ac.bbk.dcs.automed.qproc.evaluate::Evaluator
#_qpc : QueryProcessorConfiguration
#_ft : FunctionTable
+evaluate(in r : Cell)
+evaluate(in g : ASG, in qpc : QueryProcessorConfiguration)
+weakHeadNF(in root : Cell)
-reduce_builtin(in spine : LinkedList) : bool
#reduce_lambda(in spine : LinkedList) : bool
QueryEvaluationProvider
«interface»
uk.ac.ic.doc.automed.qproc.eval::QueryEvaluationProvider
+getName() : string
+getConfiguration() : QueryProcessorConfiguration
+evaluate(in g : ASG, in qpc : QueryProcessorConfiguration)
+evaluateIncrementally(in c : Cell)
uk.ac.bbk.dcs.automed.qproc.evaluate::ParallelEvaluator
-pool : ThreadPoolExecutor
+evaluate(in r : Cell)
+evaluate(in g : ASG, in qpc : QueryProcessorConfiguration)
+weakHeadNF(in root : Cell)
-reduce_builtin(in spine : LinkedList) : bool

Figure 5.1: `ParallelEvaluator` class diagram.

functionality and adding a new, threaded, logic for reducing built-in functions. The `QueryEvaluator` component illustrated in Figure 2.3 corresponds to the `QueryEvaluationProvider` interface which is implemented by the `Evaluator` class.
The new implementation is based on the following observation: evaluating a function has three discrete steps; a *pre-processing* step, where function-associated operations before the actual evaluation are performed, the *evaluation* step (includes reduction to normal form or to weak-head form of the arguments of the functions in parallel) and a *post-processing* step, where function-associated operations after the evaluation take place.
The new evaluator overrides function `reduce_buitin` and changes the way in which it evaluates built-in functions. With the new design of the IQL functions (see Section 5.4 below), each function encodes information on (a) whether its arguments should be processed in parallel or not, and (b) whether an argument is to be reduced to normal form or weak-head normal form. The evaluator then decides if it should use a new thread or not, based on this information. Below is a code snippet of `reduce_builtin`:

```
...
CountDownLatch cond=new CountDownLatch(arity);
Object tasks[]=new Object[arity];

for(int i=0;i<arity;i++){
  if((function.getEvaluationMeta().get(i)).toString().equals("evaluate"))
  {
    tasks[i]=new EvaluatorEvaluateThread(this,args[i],cond);

    try{ getPool().execute((EvaluatorEvaluateThread)tasks[i]); }
    catch(RejectedExecutionException ex){ }
   }
   ....
   cond.await();
   redex.mimicThisCell(function.post_processing(args));
}
```

Above, the number of tasks that are to be created depends on the arity of the function. The parent thread creates as many `Object` ojects as the arity of the function, and a `CountDownLatch` object is also initialized with the arity of the function. The `Object` objects are the tasks that will be executed and the `CountDownLatch` object is used to synchronize the parent thread with its children. The parent thread waits for the children to end, calling `cond.await()`, and upon completion calls the `post_processing()` method of the current built-in function. In the `cond.await()` call, the parent thread is waiting until the latch has counted down to zero. Every time a child thread finishes, it calls the `CountDownLatch.countDown()` method, of the `CountDownLatch` object passed as a parameter from the parent, decreasing the count of the latch. Eventually, when all the children threads finish, the parent thread continues with its execution.

5.2 Annotating queries

In order to support our semi-explicit model, we have to annotate queries and indicate the level to which each cell belongs to. This class was created to include all annotation constants needed and also includes some previously defined constants related to IQL queries. The semantics of each constant is

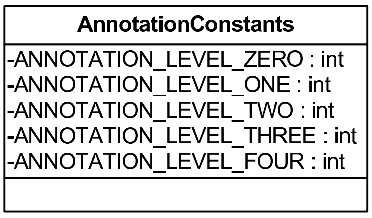

AnnotationConstants
-ANNOTATION_LEVEL_ZERO : int
-ANNOTATION_LEVEL_ONE : int
-ANNOTATION_LEVEL_TWO : int
-ANNOTATION_LEVEL_THREE : int
-ANNOTATION_LEVEL_FOUR : int

Figure 5.2: `AnnotationConstants` class.

explained in the next section.
During the annotation phase, the query tree is traversed and wrapper objects are inserted. In serial execution, the same wrapper object is used to access the database. In parallel execution this is not efficient, because race and synchronization issues may occur. For this reason a new class, named `QueryAnnotationProviderForParallelEvaluation`[1], was created. This class extended so it can be used for annotating the query in parallel evaluation.

5.3 The threading level mechanism

The user can choose which functions are to be evaluated in parallel, by modifying the *threading level* in which the evaluator operates. The threading level mechanism is controlled via the `threadingLevel` member variable

[1]Provided by Lucas Zamboulis.

of the `QueryProcessorConfiguration` class. When evaluator operates for example, in `ANNOTATION_LEVEL_THREE`, functions that contained in lower levels are also evaluated in parallel. Currently there are five predefined levels:

- `ANNOTATION_LEVEL_ZERO:` in this level there is no parallelism and the parallel evaluator operates like the serial one.
- `ANNOTATION_LEVEL_ONE:` in this level the basic collection functions are parallelized. It includes functions `Append`, `Intersect`, `Monus` and `Union`.
- `ANNOTATION_LEVEL_TWO:` this level encodes the parallelization of the `Comprehension` function.
- `ANNOTATION_LEVEL_THREE:` this level includes all arithmetic and comparison functions.
- `ANNOTATION_LEVEL_FOUR:` this level includes all other functions except those that are of arity one.

The number of levels and the functions assigned to each one, were decided based on the expected improvement that we would have parallelizing each function. Our grouping is verified by the experimental results, as discussed in Section 6.4.
New levels can be easily defined by adding or removing functions. The threading level mechanism therefore introduces a new grouping of IQL functions based on the threading level they belong.
With the above technique, we support a semi-explicit model, in which some predefined functions are parallelized and the user chooses between them, tuning the evaluator for each different query.

5.4 The new design of IQL functions

Creating new classes for the parallel version of the functions, would require to duplicate a lot of code, causing redundancy problems and making code maintenance harder. Therefore, we modified the abstract class `BuiltInFunction` that all functions override. A new function, `post_processing(Cell args[], Evaluator e):Cell`, was added, which every function that follows the *argument evaluation/post-processing* model, must override, adding its own logic.

A member value, `evaluationMeta:ArrayList` was also added to encode the evaluation mode. The possible values for the evaluation mode are:

- **evaluate:** for evaluation to normal form.
- **weakHNF:** for evaluation to weak head normal form.
- **perform:** meaning that the argument must be evaluated without creating a new thread, calling the `perform` function instead of the `post_processing`.
- **none:** meaning that the argument is not to be evaluated.

It is important to note that for each argument of the function there is a corresponding value for the evaluation mode. For example, consider function intersect, which is of arity two and evaluates both its arguments to normal form before performing their intersection. The constructor of the class is like this:

```
public Intersect() throws ParseException {
  ...
  evaluationMeta.add(0,new String("evaluate"));
  evaluationMeta.add(1,new String("evaluate"));
...
}
```

This is a generic design and interferes the least with the serial evaluation. Functions that follow the pre-processing, evaluation, post-processing model, just need to override the `post_processing` function and add their own logic. New functions can be also easily added without the need for code modifications in the parallel evaluator.
Note also that currently there is no function that needs a pre-processing step.

5.5 `ThreadPoolExecutor` configuration

Instead of creating new threads each time we needed one, we chose to use a thread pool. There are various reasons for this choice: first, we have a lot of short-running tasks, so each thread evaluates the arguments of a function finishing in a short period of time. Thus, we can reuse previously created threads from the pool, sitting idle, rather than having the cost of creating a new one each time. Second, using a thread pool, we delegate all the thread

management to the pool itself.
For this implementation the `ThreadPoolExecutor` class of Java 5 was used. To use this pool, two things are needed: first, the tasks that the pool is to run must be created, and second, the pool itself must be instantiated and configured.
The tasks are simple `Runnable` objects, following the standard approach to threading discussed in Section 4.3. Two runnable objects were created: the `EvaluatorEvaluateThread`, which fully evaluates its arguments and the `EvaluatorWHNFThread`, which partially evaluates its arguments.
The pool is an instance of the `ThreadPoolExecutor` class. Below, some important points of creating and configuring the pool, are discussed:

```
package java.util.concurrent;
public class ThreadPoolExecutor implements ExecutorService {
  public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue);
  ...
 }
```

The *core pool size*, *maximum pool size*, *keep alive time* and *work queue* control how the threads within the pool are managed. The thread pool is created with *M* core threads and *N* maximum threads. When a task enters the pool for execution and the pool has fewer than *M* threads, a new thread is created to handle the request, even if there are idle threads. If there are more than *M* threads but less than *N*, a new thread will be created only if the queue is full and all threads are busy; otherwise the task will be placed on the queue, or if it is full, a new thread will be created to serve the request. If the pool has *N* threads and the queue is full, the task is rejected; otherwise it is placed on the queue for later execution. Idle threads die after *unit* time period, in an attempt to reduce the total number of threads in the pool. In our case the pool was created calling:

```
ThreadPoolExecutor(4,50,10L,TimeUnit.SECONDS,
                              new SynchronousQueue<Runnable>());
```

As we expect to have a lot of short-running tasks, we create a small number of core threads (four) so that the pool quickly reaches this number and start reusing threads. We also have fifty maximum threads and a `Synchronous` queue, which does not have an internal capacity. Thus, tasks are never stored in the queue and if all threads are busy, always a new one is created to serve the request, as we do not want to have the storing delay. Idle threads die after ten seconds.

5.6 Using the new evaluator

The programmer can enable the new evaluator by calling the `setParallelPro cessing(int level):void` method of the `QueryProcessorConfiguration` class. This method sets the threading level to the parameter's value and also changes the annotator and the evaluator that will be used. Supposing we have a `QueryProcessorConfiguration` object, `qpc`, the following code enables the parallel evaluator for parallel execution of the basic collection functions:

```
qpc.setParallelProcessing(AnnotationConstants.ANNOTATION_LEVEL_ONE);
```

Chapter 6

Performance evaluation

This chapter presents the performance evaluation of our parallel evaluator. Section 6.1 discusses measuring performance of a Java application in general. Section 6.2 presents the execution environment used to perform the experiments and Section 6.3 discusses some code-related points observed during evaluation process, which are strongly connected with the overall performance of AutoMed, and do not only affect parallel evaluation, but serial evaluation as well. Finally, Section 6.4 presents and comments the experiments conducted.

6.1 Measuring performance

Measuring performance of a Java program, particularly measuring performance of isolated tasks, presents certain difficulties. JVMs perform just-in-time compilation of the Java byte-code. This means that the longer an application runs, the more efficient the application becomes: more code becomes compiled, more methods become inlined and so on. A second complication is introduced by the garbage collector. In our case we are measuring the performance of discrete operations, the time that query execution takes, and when the garbage collector is running may interfere with this timing.
Platform-specific factors may also affect performance. Different operating systems and virtual machines produce different results (Section 4.6.2) and heap and thread stack sizes can also affect performance (Section 4.6.1). The number and timing of processors are also important factors in performance evaluation.
Results presented in this chapter are coupled to the configuration discussed

in the next section.

6.2 The test environment

Table 6.1 lists the characteristics of the computers used for the experiments.

Table 6.1: Technical characteristics.

Name	Memory	CPU(s)	Operating System	DBMS
PC_1	2GB	2 x P4 3.0Ghz	Win XP Pro	Postgres 8.0
PC_2	1GB	2 x P4 3.0Ghz	Win XP Pro	MySQL 4.1
PC_3	1.5GB	P4 2.4Ghz	Win XP Pro	MySQL 4.1

AutoMed was installed on PC_1. The AutoMed repository was stored in a Postgres relational database, using the `pg74.214.jdbc3` jdbc driver for communication with the database. MySQL 4.1 was installed in the two other computers, each of which had two different databases used to submit queries. `MySQL Connector/J 5.0.7` driver was used for communication with these databases, supporting full parallelism for database access. Table 6.2 lists the tables of each database and their size. Conducting the experiments with larger tables was not possible due to memory restrictions placing an upper limit on the size of query results that can be constructed.

Table 6.2: Tables used and their size.

Database						
gpmdb	table	proseq	protein	aa	peptide	species
	size	884	1568	9818	19696	59553
pepseeker	table	proteinhit	peptidehit			
	size	137191	186873			

Stabilizing the execution environment was an important goal. First, we assigned fixed values to the heap size, so the JVM does not allocate memory dynamically. Using flags `-Xms` and `-Xmx` for the initial and maximum heap size accordingly, we created a heap size of 1200 mbytes[1]. Also a thread stack

[1]This is a large heap size necessary only for the experiments and not in normal execution of the AutoMed toolkit.

frame of 10mbytes was created using the flag `-Xss`. Second, we tried to minimize the effects of garbage collection. After the end of each experiment we call the `System.gc()` method, to give an indication to the JVM that the garbage collector should be called, and the executing thread sleeps for three seconds, in order to avoid interfering with the garbage collector. A new instance of the `Evaluator` and `ParallelEvaluator`[2] classes is also created to ensure the same execution conditions for each experiment. Third, the thread pool was instantiated with a fixed number of threads, all of them being pre-started. None of these threads are destroyed even if they are idle for a long time. Thus, we have a fixed number of threads, always available to execute a task. The reason for this choice was that we did not want to count the added cost of creating-destroying a thread in the experiments[3]. The program was compiled using Java 1.5 and executed in Java HotSpot 1.5 virtual machine. All the experiments ran outside the IDE, from the command prompt. As the IDE maintains its own JVM, we had a notable improvement when running from the command prompt, both in serial and parallel executions.

6.3 Performance associated code observations

During the evaluation process we observed and modified parts of the code affecting performance, discussed below:

- Cache inside `CallToWrapper` class: each time a `CallToWrapper` function is used to evaluate an IQL query against a data source, it checks whether it can use internally cached data to fulfill the request[4]. In order to operate correctly, each time data is retrieved from or stored in the cache, a copy is created, using the method `Cell.copyOfGraph()`. With the size of data increasing, this process can take a significant amount of time, delaying query execution. This cache was designed to improve a specific category of queries, joins and cartesian products. As it is out of the scope of this thesis to investigate the general behavior

[2]In normal execution, the same instance of the `Evaluator` or `ParallelEvaluator` classes is used for evaluation of multiple queries.

[3]The cost of having varying number of threads inside the pool was counted and does not affect the results of the experiments.

[4]Note that this cache operates within a *single* query and not across queries i.e. the cache is initialized whenever a new query is submitted to the AQP.

of the cache, all experiments were conducted with the cache disabled.

- Retrieving data using the same wrapper object: until now, the same wrapper object was used to access data, causing race conditions and a need for synchronization techniques with parallel execution. This is not desirable as concurrent data retrieval is fully supported from the jdbc driver used, and synchronizing the threads would add unnecessary delays. To solve this problem we create a different wrapper object for each subquery. This is not the perfect solution, as it uses multiple connections to access the database. Instead we can have one wrapper object per database and use a connection pool to access the database. Since this part of code is developed by the Imperial College, we tried to interfere as little as possible with their code, and a new query annotator was developed to produce different wrapper objects to access data.

- The `SQLWrapper` class: `SQLWrapper` class extends the `AutoMedWrapper` abstract class and is responsible for retrieving SQL data. The whole class was carefully restructured in order to efficiently support the parallel query evaluation. The previous implementation included calls to functions (i.e. static methods `toASGList()` or `toASGTuple()`), that although created to simplify the code, were causing delays to parallel execution.

- Static methods and member variables: every time there is a call to a static method, threads try to acquire the lock of the class to which the method belongs, causing delays. Throughout the AutoMed code there are calls to static methods, or static member variables, that significantly delay query evaluation. For example, a static `QProcLogger` object within the `Cell` and `ASG` classes is used to log various debugging information. This is a bottleneck for parallel execution and was removed. As a second example, the `Cell` class had a static member called `id` used to uniquely identify each cell. Every time a new `Cell` was created, `id` value was incremented by one, causing a race condition (as multiple threads may try to read and modify its value each time). As this member was not used anywhere in AutoMed, it was removed.

6.4 Experimental results

This section presents and discusses our performance evaluation of the new evaluator. For each experiment, we give the initial query and the query that is submitted to the query evaluator after reformulation, optimization and annotation. Each experiment is conducted using both the serial and the parallel evaluator, performing one hundred iterations for each one. We have two different fixed databases and we use the seven tables discussed above, except where explicitly stated otherwise. Thus, each point in the following graphical representations corresponds to the average execution time of the query over one hundred iterations, for each of the seven tables.
The `Append` function `(++)` is used for the integration semantics and for all the experiments[5]. Append is commonly used for integration semantics in AutoMed and optimizers that take it into account already exist.

First experiment

Here the initial query has nested calls to the `(++)` function and, after the optimization, the number of calls to the append function and nesting increases. Queries with nested calls to functions is something usual in AutoMed applications (e.g. ISPIDER[6]) and with this experiment we test whether there is an improvement in parallelizing collection operators like `(++)`. The parallel evaluator operates on level `ANNOTATION_LEVEL_ONE` and there are two target schemas (S_1 and S_2 respectively).

Query:

$$([\{x\}|\{x\} \leftarrow \langle\langle ... \rangle\rangle] \ ++ \ [\{x\}|\{x\} \leftarrow \langle\langle ... \rangle\rangle]) \ ++$$
$$([\{x\}|\{x\} \leftarrow \langle\langle ... \rangle\rangle \ ] \ ++ \ [\{x\}|\{x\} \leftarrow \langle\langle ... \rangle\rangle])$$

Query submitted for evaluation:

```
((++)
    ((++)
        ((++) ($Q_1$) ($Q_2$) )
        ($Q_3$) )
    ((++)
        ($Q_4$) ($Q_5$) )
```

[5]Other collection functions (like `Intersect` or `Union`) can be equally used, exploiting other optimizers or writing new ones.

[6]ISPIDER is a project that aims to create a platform to support distributed data analysis using Grid-based technologies. See Section 7.1 for more details.

)
where

$Q_1 \equiv \$wrapper\ MySQLWrapper@1dcc2a3$

$((++)\ \ L[\{x\}|\{x\} \leftarrow:: S_1 : \langle\langle ... \rangle\rangle :]\ \ L[\{x\}|\{x\} \leftarrow:: S_1 : \langle\langle ... \rangle\rangle :])$

$Q_2 \equiv \$wrapper\ MySQLWrapper@14eaec9\ \ L[\{x\}|\{x\} \leftarrow:: S_2 : \langle\langle ... \rangle\rangle :]$

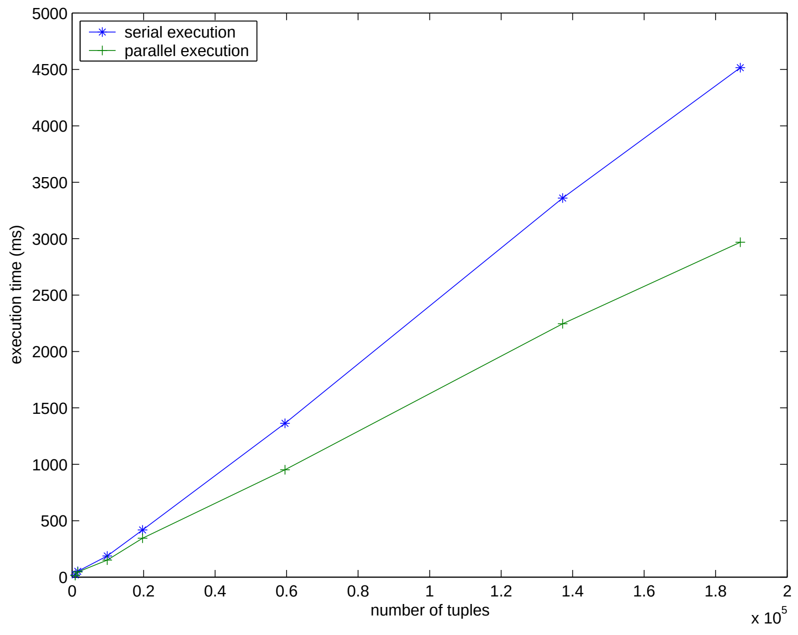
$Q_3 \equiv \$wrapper\ MySQLWrapper@d67067$

$((++)\ \ L[\{x\}|\{x\} \leftarrow:: S_2 : \langle\langle ... \rangle\rangle :]\ \ L[\{x\}|\{x\} \leftarrow:: S_2 : \langle\langle ... \rangle\rangle :])$

$Q_4 \equiv \$wrapper\ MySQLWrapper@e22f2b$

$((++)\ \ L[\{x\}|\{x\} \leftarrow:: S_1 : \langle\langle ... \rangle\rangle :]\ \ L[\{x\}|\{x\} \leftarrow:: S_1 : \langle\langle ... \rangle\rangle :]$

$Q_5 \equiv \$wrapper\ MySQLWrapper@17cf6b6\ \ L[\{x\}|\{x\} \leftarrow:: S_2 : \langle\langle ... \rangle\rangle :]$



This experiment confirms the anticipated speed-up due to parallelism. Executing the arguments of nested functions in parallel, drops the execution time up to thirty five percent for the given table sizes. Note that this difference becomes more obvious as the number of tuples increases.

Second experiment

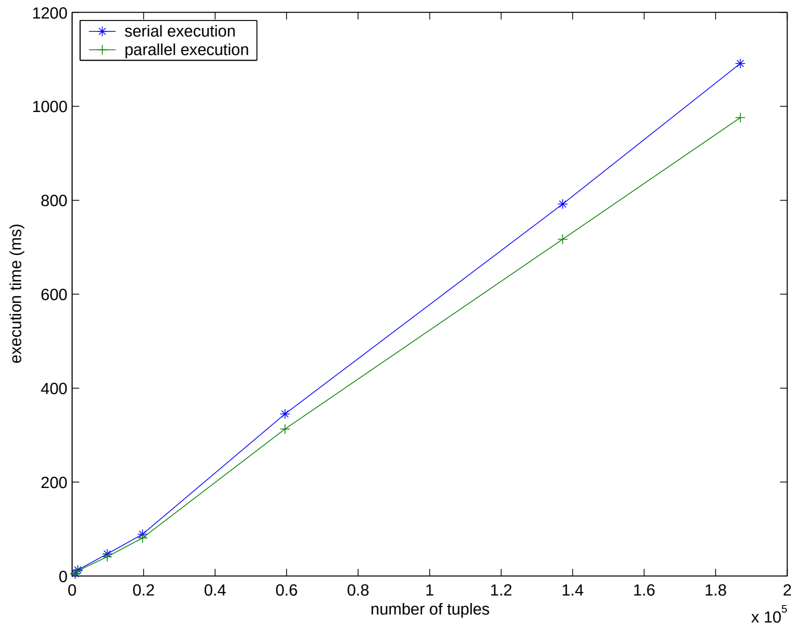
Here the initial query is a projection with two target schemas (S_1 and S_2 respectively) and the query submitted for evaluation is a single `(++)` function. The parallel evaluator operates on level `ANNOTATION_LEVEL_ONE`.

Query:

$[\ \{x\}\ |\ \{x\} \leftarrow \langle\langle ... \rangle\rangle\ ]$

Query submitted for evaluation:

$$((++)$$
$$(\$wrapper\ MySQLWrapper@ea48be\ L[\{x\}|\{x\} \leftarrow:: S_1 : \langle\langle ... \rangle\rangle :])$$
$$(\$wrapper\ MySQLWrapper@12aea3e\ L[\{x\}|\{x\} \leftarrow:: S_2 : \langle\langle ... \rangle\rangle :]))$$



Results of this experiment shows that the drop in execution time is strongly coupled with the type of query. This query has fewer calls than the first one to nested functions, thus fewer subtrees can be evaluated in parallel. But as observed, even with a simple query like this, parallel execution achieves a drop in execution time of up to ten percent.

Third experiment

In accordance with the previous experiment, here we have a projection but with an added condition (half of each table is selected) and optimization is turned off. With optimization switched on, the selection condition is pushed down into the wrappers and this query would not differ from the previous one. The query submitted for evaluation is a simple comprehension with a single generator whose body is a `(++)` function. The comprehension has also a filter, selecting the half of each table. The parallel evaluator operates on level `ANNOTATION_LEVEL_ONE`. There are two target schemas (S_1 and S_2 respectively).

In this query, the added cost for checking for the condition is comparable to the cost of retrieving data and dominates the overall cost, eliminating the benefits from fetching data in parallel.

Query:

$[\ \{x\} \mid \{x\} \leftarrow \langle\langle ... \rangle\rangle; x > \mathit{table\_size}/2\ ]$

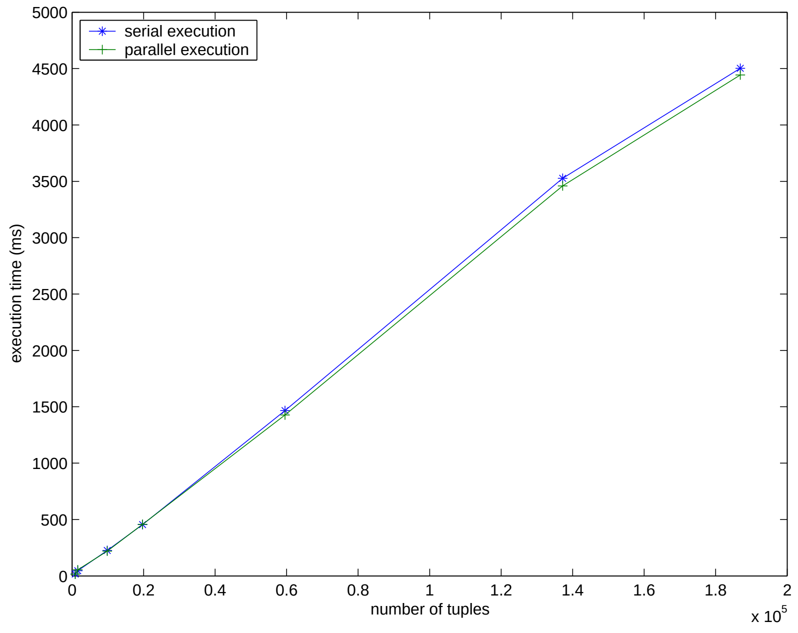
Query submitted for evaluation:

$L[\{x\}|\{x\} \leftarrow ((++)$

$(\$wrapper\ MySQLWrapper@12bf419\ \ L[\{k1\}|\{k1\} \leftarrow:: S_1 : \langle\langle ... \rangle\rangle :])$

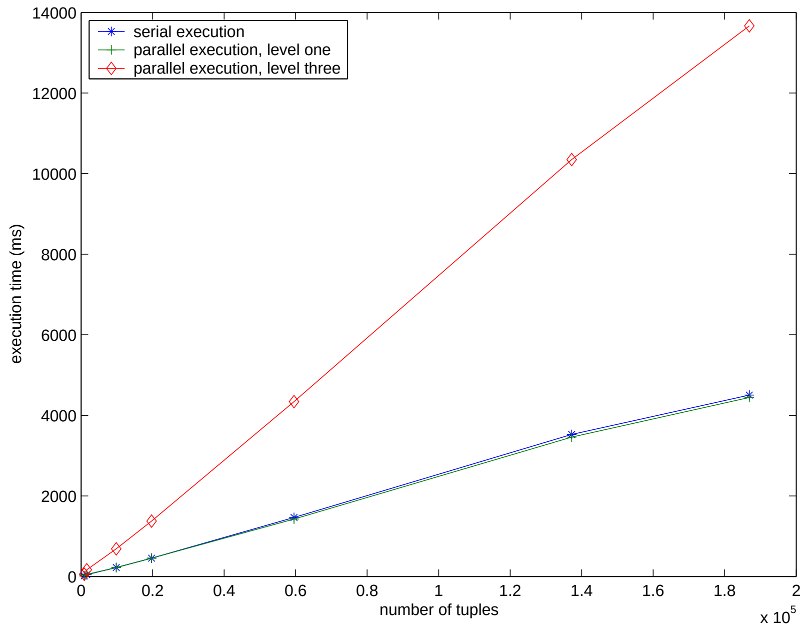
$(\$wrapper\ MySQLWrapper@1c89f29\ \ L[\{k1\}|\{k1\} \leftarrow:: S_2 : \langle\langle ... \rangle\rangle :])\ );$

$((>)\ x\ \mathit{table\_size}/2)\ ]$



Fourth experiment

This is the same experiment as the previous one, but this time parallel evaluator operates on level `ANNOTATION_LEVEL_THREE`, evaluating the arguments of arithmetic and comparison functions in parallel. With this experiment we show that, usually, it is not beneficial to have simple comparisons evaluated in a threaded manner. Checking the condition for each element of the collection using a new thread causes a very fine grained parallelization, delaying rather than improving performance.



Fifth experiment

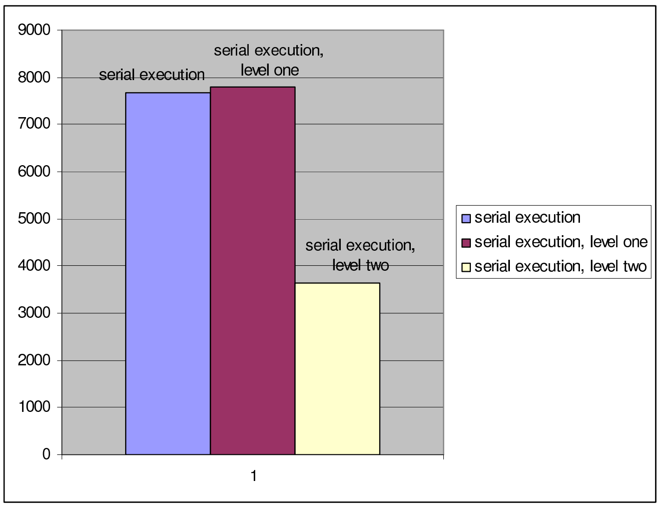
This is an example of a cartesian product. Optimization is disabled and the experiment is conducted with the parallel evaluator operating first on level `ANNOTATION_LEVEL_ONE` and then on `ANNOTATION_LEVEL_TWO`, with only one target schema (S_1) and table (*proseq*). With this configuration we also investigate parallelization of comprehensions.

Query:

$[\ \{x,y\}\ |\ \{x\} \leftarrow \langle\langle proseq\rangle\rangle;\ \{y\} \leftarrow \langle\langle proseq\rangle\rangle;\ ]$

Query submitted for evaluation:

$L[\{x,y\}|$
$\{x\} \leftarrow (\$wrapper\ MySQLWrapper@9a92b5\ \ L[\{k1\}|\{k1\} \leftarrow:: S_1 : \langle\langle proseq\rangle\rangle :]);$
$\{y\} \leftarrow (\$wrapper\ MySQLWrapper@13803d1\ L[\{k1\}|\{k1\} \leftarrow:: S_1 : \langle\langle proseq\rangle\rangle :])]$



Currently cartesian products is a category of queries that is not adequately supported by AutoMed. This is because this category of queries are translated and executed in a nested loops evaluation strategy. Thus we have a complexity of $O(n^2)$ and as all operations are done in memory, even for small tables we get out of memory errors. For the given configuration, the only table that could be used was table *proseq* with size 884 tuples.
This experiment adds to the argument that when it comes to parallelizing a functional language, it is beneficial to diverge from the lazy evaluation model and support some kind of eager evaluation. With the different approach that we adopt in translating the `Comprehesion` function for parallel evaluation, we achieve a drop of fifty percent in execution time.
The same behavior has been observed with joins. The parallel evaluator, operating on level `ANNOTATION_LEVEL_TWO`, is faster than the serial evaluator.

6.5 Conclusions

The above experiments tested the performance of the new evaluator in different scenarios. During the performance evaluation we ignored the network cost. We can safely assume that in a setting where a number of data sources are integrated and parallel query processing is performed, there would be a dedicated network that minimizes delays and is only used for communication between the machines hosting the data.
Our experiments show that the new evaluator is faster for all queries (pro-

vided an appropriate threading level is chosen), with this improvement varying according to the type of query. The speed-up observed varies between ten percent (second experiment) and fifty percent (fourth exeperiment), for the given data sizes and queries. In queries with nested collection functions, a common category of queries in AutoMed, we have observed a drop of up to thirty five percent. Furthermore, the benefits of parallel evaluation are connected with the size of tables: the larger the size of the tables, the greater the difference in execution time.

However, the fourth experiment shows that if the threading level is not carefully chosen, the parallel evaluator can be a lot slower. For the majority of the queries, operating on level `ANNOTATION_LEVEL_TWO` is sufficient. Indeed, moving to a higher threading level, yields very fine task granularity, significantly affecting performance. Even parallelizing the basic collection functions (level `ANNOTATION_LEVEL_ONE`) is enough to offer a notable improvement.

Cartesian products and joins are a category of queries that are still not evaluated efficiently. Both require a large amount of memory and even though joins do not cause out of memory errors, we cannot have an acceptable response time in large data sets.

Chapter 7

Distributed query processing

7.1 Introduction

A *Distributed Database System (DDS)* [7] consists of several databases stored at different locations, communicating using a network. Each database is managed by and running under an independent *Database Management System (DBMS).* These servers can cooperate in executing *global queries* and *global transactions*, offering transparent data distribution.
Distributed query processing is not something new and has been engrossing the scientific community since the release of the experimental database system, System R* [8]. Since then, solutions that address the challenges of distributed query processing have been proposed (e.g. *data slicing and scattering, data duplication, distributed joins, semi joins and Bloom filters*), with articles appearing in many conferences and scientific journals. Investigation and presentation of these topics is out of the scope of this thesis.
With the emergence of heterogeneous data integration, new challenges are appearing. In addition to the need for schema translation and integration, there are new challenges regarding global query processing and optimization: A query expressed on a global schema now needs to be translated into the constructs of the data source schemas, and this translation is likely to be more complex than the unions and joins of fragments in relational DDBs. Local databases will in general support different query languages, hence the types of queries that each data source can handle must be taken into account. The cost of processing local queries is likely to be different on different data sources, complicating the task of deriving a global cost model.
There are various solutions, proposed from the academic community and re-

search centers. ISPIDER [32] combines the Grid data access (OGSA-DAI)[1], Grid distributed querying (OGSA-DQP)[2] and AutoMed tools, to support distributed data analysis. RoDEX[3] is another project, developed at Imperial College, for supporting data integration and distributed query processing in a peer-to-peer network [18]. Reference [5] is a proposal from Microsoft for distributed query processing in a heterogeneous environment for SQL server.

Our approach, discussed in the following sections, follows a simplified distributed model. It is a first attempt to natively support distributed query processing using the AutoMed Query Processor.

7.2 The architecture

With our approach, we follow a hierarchical distributed model. Figure 7.1 illustrates this architecture. In this approach, we have a central node to

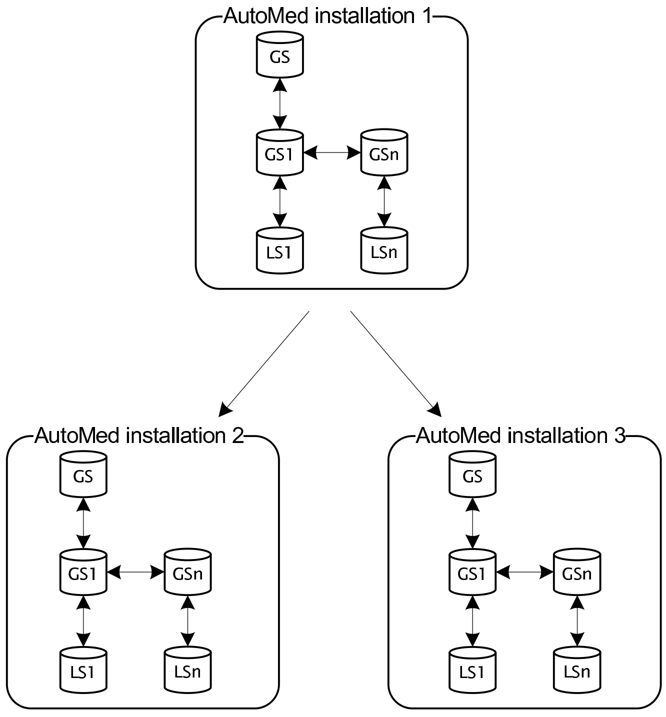


Figure 7.1: Distributed architecture.

[1]See `http://www.ogsadai.org.uk`.

[2]See `http://http://www.ogsadai.org.uk/about/ogsa-dqp/`.

[3]See `http://www.doc.ic.ac.uk/automed/rodex.html`.

which the query is submitted. Each node in the Directed Acyclic Graph (DAG), is an autonomous AutoMed installation, which has its own global schema, integrating a number of data sources and may communicate with other AutoMed installations. Queries submitted to each node are expressed in IQL, thus there is no query language translation issue. The whole process for answering a query submitted to a node is orchestrated by wrappers created for this reason.

In order to be able to answer a query, each node in the DAG must be aware of the global schema that its neighbors are exposing. Thus, the first step is to wrap the global schemas of remote installations. This process must be performed in each node that has a child. For example, in Figure 7.1 node 1 wraps the schemas of each of its children (nodes 2-3). The programmer must know the name of the global schema that each of the children expose.

Suppose for example, that a query Q is submitted to the node 1 in Figure 7.1. This query first has to be reformulated according to the source and target schemas. Suppose that one of the target schemas contain a remote schema; this makes no difference to query reformulation as this schema is available and stored in the local repository. Query reformulator does not distinguish between local and remote schemas.

The query after reformulation is optimized and must then be annotated. In this phase, we have to indicate that the constructs of the subtree that the wrapper can translate are located in a remote installation. For this reason a new annotator, `QueryAnnotationProviderForDistributedEvaluation`, was created. This annotator extends the `QueryAnnotationProviderFor ParallelEvaluation` inserting wrapper objects in the optimized query. This annotator does not check for query language translation and does not have a parser associated with it. When the evaluation phase begins and the `CallToWrapper` function is called, the wrapped subquery is sent to the remote installation for evaluation. In this node, the query may be evaluated against the local data sources, or the query may need to be remotely evaluated. Note, that because each node is an autonomous installation, the evaluator may operate in a serial mode or a parallel mode with a different threading level than the node from which the query was received. Each node can independently decide the configuration that it will follow, based for example on the available memory.

This architecture has following advantages: First, it supports a flexible DAG architecture. Queries can submitted to any node and the nodes may be connected creating a DAG. With this approach we also avoid possible problems with cycles in query processing. Second, we offer a *layered abstraction* to integrated data sources. In order to answer a query, it is not needed to have detailed knowledge of the schemas of the data sources: the global schema that each installation exposes is enough. Third, it changes the way that queries are answered, *balancing the workload* between the nodes. Whereas in the current implementation, data must be transfered from the source databases and stored centrally in the memory of the machine where the logic of the built-in IQL functions is applied, with distributed execution the whole process is performed in different nodes, balancing the resource consumption.

7.3 The `AutoMedHost` wrappers

As discussed above, the distributed query processing is orchestrated by wrappers. These wrappers are the `AutoMedHostWrapperFactory` and the `AutoMedHostWrapper`.
During the wrapping of remote schemas, the method `populateSchema(AutoMedWrapper wrapper)` of the `AutoMedHostWrapperFactory` is called. This then calls the method `retrieveSchema()` of the `wrapper` instance to create a new connection to the remote AutoMed installation and retrieve the remote schema. The retrieved schema is then stored in the local repository.
The `AutoMedHostWrapper` is responsible for handling communication between different AutoMed installations and services the requests for schema retrieval and query execution. The first time the class is loaded into the JVM of an AutoMed installation, the wrapper registers itself with an associated protocol and driver, so it will be possible to be used by the query annotator. The following code snippet shows this process:

```
static {
   AutoMedWrapper.registerWrapper(AutoMedHostWrapper.class,
     Protocol.assertProtocol("AutoMedHostProtocol"),
       "AutoMedHostDriver","AutoMedHostProtocol:");
  }
```

When the query is evaluated, function `executeIQL(ASG q):ASG` of the `AutoMedHostWrapper` is called. This method is responsible for executing the query

in the remote node and encodes the logic for handling the communication between the two cooperating nodes.
Currently, `AutoMedHostWrapper` only supports the relational data model. Supporting different data models would require a complicated communication protocol to transfer the metadata of the model between the nodes. Due to time restrictions we chose to only support the relational model for the current implementation.

7.4 Communication between AutoMed installations

In order for a query to be evaluated in each node, a running instance of a simple server that would accept and serve connections from other nodes must exist, as well as, a communication protocol which all nodes obey.
`AutoMedHost` is a simple, threaded server that was developed for serving requests from other nodes. Thus, an AutoMed installation is implemented by an instance of the `AutoMedHost` class. `AutoMedHost` uses the `ServerSocket` class to create and bind a socket that is used for communicating with remote clients. Class `ServerSocket` establishes a connection between two AutoMed-Hosts using the TCP/IP protocol. Using the TCP protocol, we do not have to manually control communication (acknowledge packets, check for the order that packets arrive, retransmit lost packets and so on) that the UDP protocol would require. When a new connection arrives, the `AutoMedHost` server creates a new thread to serve this request and then waits again for a new one. The following is the code snippet of this process:

```
while(keepAlive){
  try {
    Socket accept = server.accept();
    new Dispatcher(accept).start();
  } catch (IOException ex) { ex.printStackTrace(); }
}
```

The `Dispatcher` class extends the `Thread` class, and the new thread executes the code within the `run()` method of this class. This method, based on the type of the request that was received, either retrieves a schema or executes a query. Below is the code snippet:

```
public void run(){
   PrintWriter out = null;
   BufferedReader in = null;
   String response = null;

   try {
      // get the streams that are used
      // to write to and read from the socket
      out = new PrintWriter(new OutputStreamWriter(
                                   socket.getOutputStream()));
      in = new BufferedReader(new InputStreamReader(
                                   socket.getInputStream()));

      //get the request
      response=in.readLine();

      //check the type of the request
      switch(new Integer(response).intValue()){
         case DistrComConstants.GET_SCHEMA_REQUEST:
              //retrieve schema
              ...
              break;
         case DistrComConstants.EXECUTE_QUERY_REQUEST:
              //execute a query
              ...
              break;
      }
   }
   ...
}
```

When a node needs to retrieve a schema it must send a `GET_SCHEMA_REQUEST` and when it needs to execute a query an `EXECUTE_QUERY_REQUEST`. All the communication variables are included in class `DistrComConstants`. Figure 7.2 illustrates the communication diagram of our protocol.

7.5 Using the evaluator for distributed query processing

Using the new evaluator for distributed querying processing requires more involvement from the user than the parallel execution. First, an instance of the `AutoMedHost` class should be running in each machine (except the

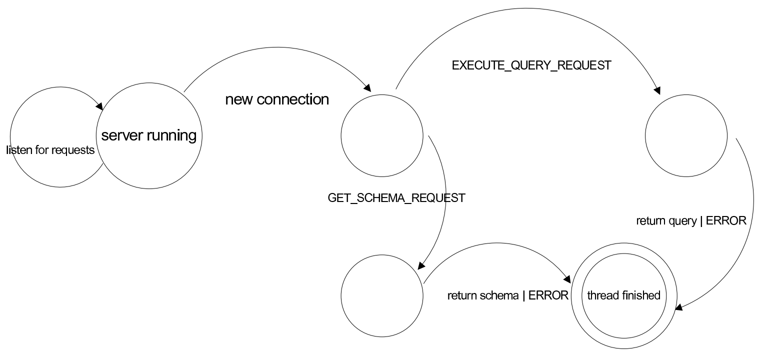


Figure 7.2: Communication state diagram.

machine to which the query is submitted). Then the user must wrap all the remote schemas that each machine wants to be aware of. The following code snippet shows how this can be performed:

```
...
AutoMedWrapper fstw=AutoMedWrapper.selectNewAutoMedWrapper("","",null,
   "AutoMedHostDriver","AutoMedHostProtocol://193.61.44.26/gpmdb1",
        "gpmdb3",new AutoMedHostWrapperFactory());

AutoMedWrapper sndw=AutoMedWrapper.selectNewAutoMedWrapper("","",null,
   "AutoMedHostDriver","AutoMedHostProtocol://193.61.44.40/gpmdb1",
        "gpmdb4",new AutoMedHostWrapperFactory());
...
```

In the above example, the user chooses to wrap and store into the local repository the global schemas with name `gpmdb1`, stored in the machines with IP addresses 193.61.44.26 and 193.61.44.40 accordingly. The name that these two schemas would have in the local repository is `gpmdb3` and `gpmbd4` and the driver and protocol that should be used are the `AutoMedHostDriver` and the `AutoMedHostProtocol` accordingly. After the wrapping, the schemas can be integrated and queries that use these schemas can be written as usual. Note also that distributed query processing requires the use of the `QueryAnnotationProviderForDistributedEvaluation` annotator. Appendix A includes a full example of distributed query processing.

Chapter 8

Conclusions and future work

This project extends the AutoMed Query Processor component to support parallel and distributed query processing.
In order to meet the goals for this project, I had to gain an understanding of the AutoMed theoretical aspects. It was necessary to understand the abstract representation of IQL queries and the IQL language itself. As IQL is a functional language, an investigation of functional languages and how they can be parallelized was also conducted. After this, all the intermediate steps that comprise the query evaluation process were studied, focusing on the query annotation and query evaluation, as these are the main topics I was involved with. Understanding the source code of AutoMed was another challenge. I studied all the main classes that comprise the query processor and after that, I developed the new parallel evaluator by extending the existing one. During the evaluation phase, I studied the code in more detail, locating and changing parts that were causing delays, synchronization issues and possible race conditions. The help of Lucas Zamboulis, the lead developer of the AutoMed project at Birkbeck, was crucial to understand in depth the source code of the AutoMed.
The second requirement for the AQP was to support distributed query processing. For this reason, a threaded server that accepts new connections and serves requests from client was developed to model an AutoMed installation. New wrappers that will be responsible for the communication between the cooperating nodes were developed. For this purpose I studied the `AutoMedWrapper` and `AutoMedWrapperFactory` classes and implemented the new wrappers. A simple communication protocol was also developed to synchronize the communicating nodes.

The new evaluator achieves a significant drop in execution time in certain queries. Despite this, there is still work that can be done to improve its performance and functionality. First, special IQL syntax could be created that would be used to define explicit parallel evaluation policies and improve the flexibility of our semi-explicit model. Second, with the current evaluation strategy, even if arguments of a function are evaluated in parallel, the parent thread must wait for its children to end. Changing the evaluation strategy, small independent subtrees could be located and submitted to a thread for evaluation. This requires extensive changes to the way that evaluation and annotation are performed. Third, changing the way that the function `Flatmap` is evaluated according to the model discussed in Section 3.4, may achieve a further speed-up. This requires to investigate topics like the size of the cluster, what a cluster could be considered to be and so on. Fourth, a deep analysis and restructuring of the code could be done, as there are parts of the code that can be changed (like the cache in the `CallToWrapper` function), improving overall performance.
The current implementation of the distributed query processor is the first step to support distributed query processing in AutoMed. With the current evaluator, we can answer queries in a distributed manner, but in a very simple way: for example, we cannot support distributed joins. An extension to this implementation would be to support communication between the cooperative nodes and perform a distributed join. Extending the communication protocol to support more data models than the relational model is of high priority. Having a caching policy in distributed query execution and using metadata for load balancing between nodes would also be a beneficial extension to the current implementation. Allowing objects, like the `QueryProcessorConfiguration` class, to be serialized and passed between AutoMed installations, would make the new evaluator more flexible.

Bibliography

[1] *The Java Virtual Machine Specification.* Second edition.

[2] Overview of memory management. Available at http://publib.boulder.ibm.com/infocenter/javasdk/v5r0/topic/com.ibm.java.doc.diagnostics.50/html/gcover.html.

[3] Solaris threading. Available at http://java.sun.com/docs/hotspot/threads/threads.html.

[4] The Java HotSpot virtual machine. Available at http://java.sun.com/products/hotspot/docs/whitepaper.

[5] J. A. Blakeley, C. Cunningham, N. Ellis, B. Rathakrishnan, and M.C. Wu. Distributed/Heterogeneous Query Processing in Microsoft SQL Server. In *Proc. of 21st International Conference on Data Engineering (ICDE'05)*, pages 1001–1012, 2005.

[6] M. Boyd, C. Lazanitis, S. Kittivoravitkul, P. McBrien, and N. Rizopoulos. An overview of the AutoMed repository. Technical draft, Dept. of Computing, Imperial College, 2004.

[7] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Addison Weshley, third edition, 1999.

[8] D. D. Chambelin et al. A History and Evaluation of System R*. *Communications of the ACM*, 24(10):632–646, 1981.

[9] A. J. Field and P. G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[10] J. Gosling and H. McGilton. The Java language environment, a white paper. White paper, Sun Microsystems, 1995.

[11] G. Hains. Parallel Functional languages should be Strict. *Workshop on GPPP - World Computer Progress*, pages 527–532, 1994.

[12] A.Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[13] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[14] W. H. Inmon. *Building the Data Warehoouse*. John Wiley, New York, 1992.

[15] E. Jasper, A. Poulovassilis, L. Zamboulis, H. Fan, and S. Mittal. Processing IQL Queries and Migrating Data in the AutoMed toolkit. Technical draft, School of Computer Science and Information Systems, Birkbeck College, 2007.

[16] S. L. Peyton Jones. *Implementing Functional Programming Languages*. Prentice-Hall International, 1992.

[17] S.L. Peyton Jones and J. Hughes (editors). Haskell98: A non-strict, Purely Functional Language, February 1998. http://www.haskell.org.

[18] D.M. Le, C. Lazanitis, and P.J. McBrien. AutoMed P2P Data Integration. Technical Report, Draft, Dept. of Computing, Imperial College, 2007.

[19] M. Lenzerini. Data Integration: A Theoritical Perspective. *PODS*, pages 243–246, 2002.

[20] H-W. Loidl, P.W. Trinder, and C. Butz. Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4):471–486, 2001.

[21] P.J. McBrien. AutoMed in a Nutsell. Dept. of Computing, Imperial College.

[22] P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. of International Conference CAiSE99*. Springer, 1999.

[23] P.J. McBrien and A. Poulovassilis. Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proc. of International Conference CAiSE02*. Springer-Verlag, 2002.

[24] P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. of International Conference ICDE03*. IEEE, 2003.

[25] Sun MicroSystems. Java API documentation. Available at http://java.sun.com/j2se/1.5.0/docs/api/.

[26] S. Oaks and H. Wong. *Java Threads.* O'Reilly, third edition, 2004.

[27] A. Poulovassilis. The AutoMed Intermediate Query Language. AutoMed working document, Dept. of Computer Science, Birkbeck College, 2001.

[28] A. Poulovassilis. A Tutorial on the IQL Query Language. AutoMed technical report, Dept. of Computer Science, Birkbeck College, 2004.

[29] A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28:47–71, 1998.

[30] P. W. Trinder, K. Hammond, H-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

[31] P.W. Trinder, H-W. Loidl, and R.F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(5):469–510, 2002.

[32] L. Zamboulis, H. Fan, K. Belhajjame, J. Siepen, A. Jones, N. Martin, A. Poulovassilis, S. Hubbard, S. M. Embury, and N. W. Paton. Data Access and Integration in the ISPIDER Proteonomics Grid. In *Proc. of 3rd International Workshop on Data Integration in the Life Sciences (DILS)*, pages 3–18.

Appendix A

Source code