

Query Performance Evaluation of an Architecture for Fine-Grained Integration of Heterogeneous Grid Data Sources[☆]

Lucas Zamboulis^{*,a,b}, Nigel Martin^a, Alexandra Poulouvassilis^a

^a*School of Computer Science and Information Systems, Birkbeck,
University of London, WC1E 7HX, U.K.*

^b*Department of Biochemistry and Molecular Biology, University College London,
University of London, WC1E 6BT, U.K.*

Abstract

Grid data sources may have schema- and data-level conflicts that need to be addressed using data transformation and integration technologies not supported by the current generation of Grid data access and querying middleware. We present an architecture that combines Grid data access and distributed querying with fine-grained data transformation/integration technologies, and the results of a query performance evaluation on this architecture. The performance evaluation indicates that it is indeed feasible to combine such technologies while achieving acceptable query performance. We also discuss the significance of our results for the further development of query performance over heterogeneous Grid data sources.

Key words:

data integration, query processing, Grid computing, bioinformatics

[☆]Research funded by the ISPIDER project, BBSRC Grant No. BBSB17220.

*Corresponding author.

Email addresses: lucas@dcs.bbk.ac.uk (Lucas Zamboulis), nigel@dcs.bbk.ac.uk (Nigel Martin), ap@dcs.bbk.ac.uk (Alexandra Poulouvassilis)

URL: <http://www.dcs.bbk.ac.uk/~lucas> (Lucas Zamboulis),
<http://www.dcs.bbk.ac.uk/~nigel> (Nigel Martin), <http://www.dcs.bbk.ac.uk/~ap>
(Alexandra Poulouvassilis)

1. Introduction

Grid computing technologies are becoming increasingly important, as they enable distributed computational and data resources to be accessed in a service-based environment. This is particularly of value for applications requiring access to complex combinations of computational and data resources, as in the Life Sciences, for example.

To realise the potential of Grid computing, standardisation is important both for the Grid middleware technologies and for the application areas in which distributed access to computational and data resources is required. For example, OGSA-DAI [3] is an open-source, standard Grid data access middleware, with an additional component OGSA-DQP [2] providing distributed query processing capabilities over OGSA-DAI enabled data sources. In parallel, initiatives have been taken in application areas such as the Life Sciences to provide standardised data formats and nomenclatures. For example, the FuGE object model [30] supports the standardised modelling and exchange of data related to experiments in functional genomics, while the Gene Ontology [52] defines a controlled vocabulary of terms relating to genes and gene products.

While the combination of standardisation in Grid middleware and in the application areas themselves goes some way towards easing the development of applications requiring distributed querying and analysis of Grid data sources, significant problems remain. Many data sources do not follow application area standards, and even for those that do the plethora of different standards for different sub-areas within a general application area means that a data source designed in the context of one sub-area may be incompatible at the schema and/or instance level with the requirements of an application designed for a different sub-area. Data integration technology supporting fine-grained resolution of schema and instance level conflicts can provide an effective approach to tackling such problems. Hence, a promising area for investigation is the combination of Grid data access/distributed querying middleware and data integration technologies.

Given a set of data sources, *data integration* is the process of creating an integrated resource combining data from the data sources, in order to support queries and analyses that could not be supported by the individual data sources alone. The data integration process may create and maintain a *materialised* integrated resource (i.e. a data warehouse) or a *virtual* integrated resource. The materialised approach is usually chosen for query performance reasons: distributed access to remote data sources is avoided and sophisticated query optimisation techniques can be applied to queries submitted to the data warehouse. However, maintaining a data warehouse can be very complex and costly, and virtual integration is the preferred option if these maintenance costs are prohibitively high; virtual integration is also the preferred option if access to the latest versions of the data sources is required.

Many systems have been developed which create and maintain virtual integrated resources: for example, in the Life Sciences domain, significant systems include DiscoveryLink [26], K2/Kleisli [11], Tambis [22], SRS [58]. The aim of these early systems was to provide users with the ability to formulate queries on the integrated resource. The technologies for doing so differ between systems: DiscoveryLink and Kleisli utilise views over “wrapped” data sources, Tambis incorporates biological ontologies that serve as a global schema over heterogeneous data sources, while SRS adopts a portal-based approach rather than supporting a global integrated schema, an approach extended more recently [4] with schema mappings relating attributes in the linked data sources. Other more recent work [23, 53, 45, 47, 34, 9, 8] has focussed on querying distributed resources, or virtual integrated resources, explicitly within a Grid environment. The technologies supporting this are based on distributed querying processing over wrapped data sources with, in some cases, modelling of ontology-based metadata.

However, none of this work on querying heterogeneous distributed Grid-enabled data sources has so far supported *fine-grained* data transformation and integration of the source data — by fine-grained we mean transforming the values of individual attributes of data source entities and combining them to

form the extent of a new attribute of an entity in the virtual global schema. Therefore, the systems described above can only support simple data transformation queries between the entities of the data source schemas and those of the global schema e.g. it is not possible to define a global schema entity as the join of two data source entities. This is significant in terms of query performance because more complex mappings lead to more complex reformulated queries (i.e. queries referencing only data source entities — see Section 3), and more complex reformulated queries may take longer to evaluate. In this paper, we will show that our architecture provides acceptable performance in the context of select-project-join queries on the global schema and select-project-join-union data transformation queries.

In our own recent work [54], we have developed an architecture for virtual integration of Grid-enabled data sources that leverages the functionalities of the AutoMed heterogeneous data integration system and the OGSA-DQP service-based distributed query processor. AutoMed provides fine-grained data transformation and integration functionality, while OGSA-DQP complements AutoMed’s centralised query processor by providing efficient distributed query processing over Grid-enabled data sources. This work has been undertaken as part of the ISPIDER (*In Silico Proteome Integrated Data Environment Resource*) project¹, which has developed a platform of proteomics-related resources, using existing Grid middleware and leveraging standards from the proteomics and bioinformatics application areas.

There is currently no Grid-enabled middleware supporting fine-grained data transformation/integration, and hence our decision to investigate combining existing data integration and Grid data access and querying middleware in order to meet the ISPIDER project’s data integration requirements. However, while our work has been motivated by ISPIDER, the architecture we have developed is generic. To our knowledge, ours is the first investigation of extending Grid data access and querying middleware with fine-grained data transforma-

¹<http://www.ispider.man.ac.uk>

tion/integration functionality.

In this paper, we investigate the performance of an architecture that combines Grid data access, distributed querying and fine-grained data integration technologies. We investigate these issues for the OGSA-DAI and OGSA-DQP open-source Grid access and querying middleware combined with the AutoMed heterogeneous data transformation/integration system. We present an architectural framework we have developed for investigating the combination of these technologies, and the results of a query performance study we have undertaken. We discuss the significance and applicability of our results for the further development of query processing capabilities over heterogeneous Grid data sources.

Sections 2 and 3 of the paper discuss the OGSA-DAI/DQP and AutoMed middleware, respectively, to the level of detail necessary for this paper. Section 4 presents our architecture, from [54], for combining the respective capabilities of these middleware systems, and also two variant architectures that we have used for comparison with this in our evaluation. Section 5 presents the query performance evaluation we have conducted over the three architectures. This evaluation has been carried out using an integrated resource that has been developed as part of the the ISPIDER project, which is discussed in Section 5.1. All three of our architectures are assessed through benchmarking against a suite of queries posed on the ISPIDER integrated resource. Some of the queries have been provided by proteomics domain experts as representative of queries which would be of value to their research community, while additional queries have been developed by us in order to benchmark particular aspects of our architectures. Section 6 gives our conclusions and directions of future work.

2. OGSA-DAI and OGSA-DQP

The **Open Grid Services Architecture (OGSA)** [19] defines a set of capabilities that address the key concerns of Grid computing systems, and provides a basis for standardisation in the provision of such capabilities through a service-oriented architecture.

OGSA-DAI (OGSA - Data Access and Integration)² is an open-source middleware product for wrapping data sources within the OGSA service-oriented architecture [3]. OGSA-DAI supports a variety of relational and XML DBMSs and text data sources, and its objective is to standardise data access, transport, integration and metadata services for the Grid (other OGSA initiatives are focussing on data derivation, consistency and replication services).

The motivation for OGSA-DAI is to develop middleware that interfaces between existing DBMSs and the OGSA architecture, with the expectation that over time vendors will embed this functionality within their DBMS products, thus simplifying application structure and improving performance. Thus, one of the motivations for OGSA-DAI is to expose and formulate requirements for integrating data management functionality into a Grid, and our work here can be seen as an extension of this aim in application scenarios where sophisticated data transformation and integration capabilities are required.

Figure 1 illustrates the OGSA-DAI architecture. Our own architectures presented in Section 4 below do not make use of steps 1 or 2 of Figure 1, use step 3 during set-up and steps 4 and 5 for querying OGSA-DAI data sources.

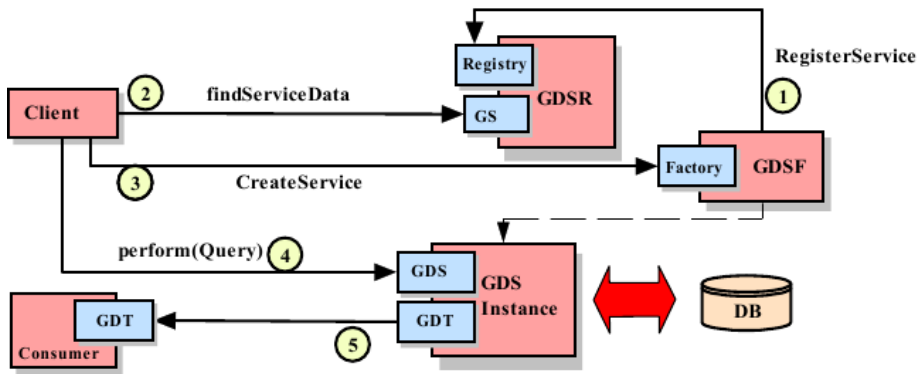


Figure 1: The OGSA-DAI Architecture (taken from [2]).

²<http://www.ogsadai.org.uk>

OGSA-DQP (OGSA - Distributed Query Processor)³ is a service-based distributed query processor over OGSA-DAI enabled resources [2]. OGSA-DQP aims to support efficient query processing over OGSA-DAI enabled resources by offering parallel and distributed query execution. OGSA-DQP offers two services, the **Grid Distributed Query Service (GDQS)**, or **Coordinator**, and the **Grid Query Evaluation Service (GQES)**, or **Evaluator**.

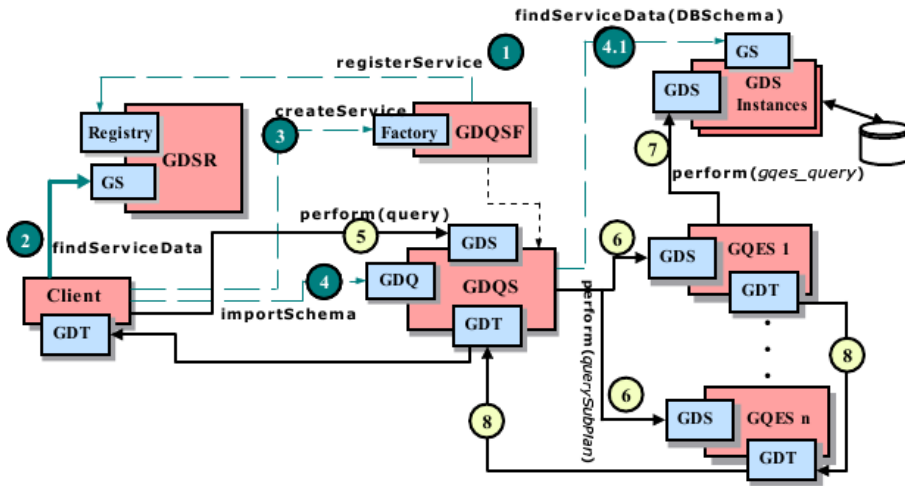


Figure 2: The OGSA-DQP Architecture (taken from [2]).

The Coordinator uses resource metadata and computational resource information to compile, optimise and partition an input query into a distributed query execution plan, and schedule each one of the partitions of this plan for execution on one of the Grid’s nodes [50]. In particular, the Coordinator’s logical optimiser undertakes query unnesting, fusion of multiple selection operations, and pushing of projection operations as close to scan operations as possible; its physical optimiser then produces a query plan by performing cost-based optimisation, if such metadata is available from the Grid data access component (which is not the case with OGSA-DAI); its partitioner component then frag-

³<http://www.ogsadai.org.uk/about/ogsa-dqp>

ments the query plan across multiple Grid nodes using parallelisation operators (e.g. parallel evaluation of a join operation); and finally its scheduler component assigns each query plan fragment to a suitable Grid node based on the CPU and memory resources of the available Grid nodes. The logical optimisation and parallelisation capabilities of the Coordinator are based on [17, 18] and [25], respectively, and we refer the interested reader to these sources for further details.

The Evaluator implements a physical algebra over OGSA-DAI data services and evaluates a partition of the query execution plan assigned to it by a Coordinator. The set of Evaluators participating in a query form a tree through which data flows from the leaf Evaluators, which interact with OGSA-DAI services, up the tree of Evaluators towards its root — the Coordinator.

Figure 2 illustrates the OGSA-DQP architecture. Our own architectures presented in Section 4 do not make use of steps 1 or 2, use steps 3 and 4 during set-up and steps 5-8 for querying OGSA-DAI data sources.

Performance studies of OGSA-DAI and OGSA-DQP have focussed on the impact of alternative data transfer formats and query processing models. [14] presents a performance evaluation of executing SQL queries using OGSA-DAI version 2.2 (also the version used for our performance evaluation here), and using two different data formats for transferring results to the client, `WebRowSet` (XML) and `CSV` (comma-separated values). Although the less verbose `CSV` data format is likely to have better performance due to lower data transfer costs [14], we have used the `WebRowSet` data format in our architecture and performance evaluation here since this supports more metadata information and will thus be advantageous in the longer term within architectures supporting cost-based query optimisation. For example, the availability of information about table sizes can allow optimisation of join operations e.g. deciding to use the smaller table as the outer table in a nested-loops join.

[33] presents a more extensive performance evaluation of OGSA-DAI version 2.2. Of particular interest are the experiments comparing JDBC performance with OGSA-DAI in terms of “synchronous” and “asynchronous” evaluation of

SQL queries. With the former, a query is fully executed by the DBMS and the query results are returned to the client in a single “packet”. With the latter, the results of a query submitted to the DBMS are returned to the client one packet at a time, which may or may not have fixed size. In particular, OGSA-DAI synchronous evaluation results in a slightly higher data throughput compared to OGSA-DAI asynchronous evaluation, but JDBC outperforms both. In this paper, to avoid confusion with the more commonly understood meanings of the terms “synchronous” and “asynchronous”, we use the terms “complete evaluation” and “incremental evaluation”, respectively. Also, unless otherwise stated, “evaluation” refers to “complete evaluation”.

[1] presents a performance evaluation of OGSA-DQP version 2 and [37] of OGSA-DQP version 3.2. Neither investigates the effect of network speed on performance, even though OGSA-DQP does take account of data transfer costs in its query planning. Of particular interest in [1] is the comparison between complete and incremental query processing, with incremental query processing requiring at least twice the time compared to complete query processing — note that OGSA-DQP 3.1 and 3.2 support only incremental query processing, and not complete. [37] suggests that this aspect of OGSA-DQP can be significantly improved, and this is supported by our experiments in this paper. To this end, OGSA-DQP could adopt a binary data format, such as SOAP-binQ [48], and could also support techniques for determining the optimal packet size at runtime [24].

The most recent release of OGSA-DQP — version 3.2 — supports SQL queries. These may include nested sub-queries in the `WHERE` clause, but no nesting is permissible in the `FROM` clause. The previous version, OGSA-DQP 3.1, supports OQL [6] queries⁴. These may include nesting in both the `FROM` and the `WHERE` clauses. However, unlike version 3.2, set operators, and in particular the `UNION` operator, are not supported by version 3.1.

⁴The Object Query Language (OQL) is a query language for object databases developed by the Object Data Management Group (ODMG) — see <http://www.odbms.org/odmg/>.

In our context, the performance advantage of support for nested queries in the `FROM` clause outweighs that of support for set operators: we discuss this further in Section 5 where we evaluate the performance of our test queries. Therefore, we have used OGSA-DQP 3.1 rather than version 3.2 in our performance study here. The next release of OGSA-DQP is expected to support all features from the previous versions, and more, and we plan to migrate to this new version when it becomes available.

3. AutoMed

AutoMed⁵ is a system supporting fine-grained transformation and integration of heterogeneous data, offering the ability to handle virtual, materialised, and indeed hybrid data integration across multiple data models. It supports a low-level Hypergraph-based Data Model (HDM) and provides facilities for specifying higher-level modelling languages in terms of this HDM [38]; in particular, this is achieved using the API of AutoMed’s **Model Definitions Repository (MDR)**, see Figure 4 below). Thus, the set of modelling languages handled with respect to either data source schemas or global schemas is extensible. For any modelling language, \mathcal{M} , specified in terms of the HDM, AutoMed provides a set of primitive schema transformations that can be applied to schema constructs expressed in \mathcal{M} . In particular, for every construct of \mathcal{M} there is an `add` and a `delete` primitive transformation which adds to/deletes from a schema an instance of that construct. For those constructs of \mathcal{M} which have textual names, there is also a `rename` primitive transformation.

AutoMed schemas can be incrementally transformed by applying to them a sequence of primitive transformations, each adding, deleting or renaming just one schema construct. A sequence of primitive transformations from one schema S_1 to another schema S_2 is termed a *pathway* from S_1 to S_2 . All source, intermediate, and integrated schemas, and the pathways between them, are stored in

⁵<http://www.doc.ic.ac.uk/automed>

AutoMed’s **Schemas & Transformations Repository (STR)**, see Figure 4 below).

Each **add** and **delete** transformation is accompanied by a query specifying the extent of the added or deleted construct in terms of the rest of the constructs in the schema. This query is expressed in a functional query language, IQL (see Section 3.1 below). Also available are **extend** and **contract** primitive transformations which behave in the same way as **add** and **delete** except that they state that the extent of the new/removed construct cannot be precisely derived from the other constructs present in the schema⁶.

The queries supplied with primitive transformations are used by the **AutoMed Query Processor (AQP)**, see Figure 4) to reformulate user queries expressed on a global schema by means of *query unfolding* (for global-as-view query processing) or by means of *rewriting queries using views* (for local-as-view query processing) — we refer the reader to [39, 40, 41] for details of this process in AutoMed, and to [35] for a general discussion of global-as-view (GAV) and local-as-view (LAV) data integration.

As discussed in [39], this means that AutoMed is a both-as-view (BAV) data integration system: the **add/extend** steps in a transformation pathway correspond to GAV rules [35] since they incrementally define global schema constructs in terms of source schema constructs; and the **delete** and **contract** steps correspond to LAV rules [15, 36] since they define source schema constructs in terms of global schema constructs.

To illustrate, consider a relational data source schema S_1 with a table *StudentDetails*(*id*, *name*, *address*) containing personal details of all the students in a university, where the attribute *id* is a university-wide unique student identifier. Consider another relational data source schema S_2 with a table

⁶More specifically, each **extend** and **contract** transformation takes a pair of queries that specify a lower and an upper bound on the extent of the construct. The lower bound may be **Void** — semantically equivalent to the empty collection — and the upper bound may be **Any** — semantically equivalent to the largest possible collection of the type of the construct.

$MathsStudents(id, degree, tutor)$ which is maintained by the Maths department and contains information about students in that department; and similarly a third relational data source schema S_3 with a table $CSStudents(id, degree, tutor)$ which is maintained by the Computer Science department and contains information about students in that department. Consider also an integrated schema IS containing the table $Students(id, name, degree, tutor, dept)$ giving the name, degree being studied, tutor and department of each student in the university. Then, a possible transformation pathway from S_1 to IS is as follows⁷:

```

addRel(⟨⟨Students⟩⟩, ⟨⟨StudentDetails⟩⟩)
addAtt(⟨⟨Students, id⟩⟩, ⟨⟨StudentDetails, id⟩⟩)
addAtt(⟨⟨Students, name⟩⟩, ⟨⟨StudentDetails, name⟩⟩)
extend(⟨⟨Students, degree⟩⟩, Range Void Any)
extend(⟨⟨Students, tutor⟩⟩, Range Void Any)
extend(⟨⟨Students, dept⟩⟩, Range Void Any)
delAtt(⟨⟨StudentDetails, id⟩⟩, ⟨⟨Students, id⟩⟩)
delAtt(⟨⟨StudentDetails, name⟩⟩, ⟨⟨Students, name⟩⟩)
contractAtt(⟨⟨StudentDetails, address⟩⟩, Range Void Any)
delRel(⟨⟨StudentDetails⟩⟩, ⟨⟨Students⟩⟩)

```

and a possible transformation pathway from S_2 to IS is as follows:

```

extendRel(⟨⟨Students⟩⟩, Range ⟨⟨MathsStudents⟩⟩ Any)
extendAtt(⟨⟨Students, id⟩⟩, Range ⟨⟨MathsStudents, id⟩⟩ Any)
extendAtt(⟨⟨Students, name⟩⟩, Range Void Any)
extendAtt(⟨⟨Students, degree⟩⟩, Range ⟨⟨MathsStudents, degree⟩⟩ Any)

```

⁷In this transformation pathway, terms of the form $\langle\langle r \rangle\rangle$ and $\langle\langle r, a \rangle\rangle$ are *schemes*, which respectively identify a relation r of an AutoMed relational schema, and an attribute a of a relation r . The standard AutoMed encoding of relational schemas decomposes each table $R(A_1, \dots, A_n)$ into n binary relationships, $R(\bar{K}, A_1), \dots, R(\bar{K}, A_n)$, where \bar{K} is the subset of the attributes A_1, \dots, A_n comprising the primary key of R . The extent of an AutoMed scheme $\langle\langle r \rangle\rangle$ is the projection of table r onto its primary key attributes. The extent of an AutoMed scheme $\langle\langle r, a \rangle\rangle$ is the projection of table r onto its primary key attributes plus its attribute a . See [39] for details.

```

extendAtt(⟨⟨Students, tutor⟩⟩, Range ⟨⟨MathsStudents, tutor⟩⟩ Any)
extendAtt(⟨⟨Students, dept⟩⟩, Range [{s, 'Maths'} | s ← ⟨⟨MathsStudents⟩⟩] Any)
delAtt(⟨⟨MathsStudents, tutor⟩⟩,
      [{s, t} | {s, t} ← ⟨⟨Students, tutor⟩⟩; {s, d} ← ⟨⟨Students, dept⟩⟩; d = 'Maths'])
delAtt(⟨⟨MathsStudents, degree⟩⟩,
      [{s, deg} | {s, deg} ← ⟨⟨Students, degree⟩⟩; {s, d} ← ⟨⟨Students, dept⟩⟩; d = 'Maths'])
delAtt(⟨⟨MathsStudents, id⟩⟩,
      [{s, i} | {s, i} ← ⟨⟨Students, id⟩⟩; {s, d} ← ⟨⟨Students, dept⟩⟩; d = 'Maths'])
delRel(⟨⟨MathsStudents⟩⟩,
      [s | s ← ⟨⟨Students⟩⟩; {s, d} ← ⟨⟨Students, dept⟩⟩; d = 'Maths'])

```

We see that the **add** and **extend** steps are defining global schema constructs in terms of source schema constructs; and that the **delete** and **contract** steps are defining local schema constructs in terms of global schema constructs. The definitions are queries expressed in the syntax of IQL, which we explain in Section 3.1 below. The transformation pathway from S_3 to IS is identical to that from S_2 to IS , but with the relation name `CSStudents` replacing `MathsStudents` everywhere and the string `'CompSci'` replacing the string `'Maths'`.

An in-depth comparison of BAV with other data integration approaches can be found in [39, 40, 41]. For our architectural framework discussed here, which aims to couple Grid data access and distributed querying with fine-grained data transformation/integration technologies, we have used so far only the GAV data integration and querying capabilities of the AutoMed system. Thus, in principle, any other data integration system that supports fine-grained data transformation and integration via GAV rules and GAV query processing could be used instead of AutoMed within our framework.

3.1. The IQL Query Language

IQL is a comprehension-based query language, like the languages that underpin the K2/Kleisli [11] and Tambis [22] systems, for example. Comprehension-based languages (including IQL) subsume query languages such as SQL-92 and OQL in their expressiveness (see [5, 17]). For example, it is possible to express

the *powerset* operation in IQL (see [51]).

The purpose of IQL within the AutoMed system is to provide a common query language that queries written in various high-level query languages can be translated into and out of. For example, a query on a relational global schema can be expressed in SQL; it will be translated into IQL, reformulated to target the data source schemas, and then translated by the data source wrappers to the appropriate query language for each data source. Below we give a brief overview of IQL, to the level of detail required for Section 5. The interested reader is referred to [44] for a tutorial on IQL and to [55] for implementation details.

IQL supports string, boolean, number, date and tuple data types, and set, bag and list collection types. There are several polymorphic primitive operators for manipulating sets, bags and lists. In particular, the binary operator `++` performs set union and bag union on sets and bags, respectively, and appends two lists. The binary operator `--` performs set difference and bag difference on sets and bags, respectively; for lists, it treats its two argument lists as bags and returns the output in ascending sorted order. The operator `flatmap` applies a collection-valued function `f` to each element of a collection and applies `++` to the resulting collections.

The operator `flatmap` can be used to specify *comprehensions* over collections. These are of the form `[h | Q1; ...; Qn]` where `h` is an expression termed the *head* of the comprehension, and `Q1, ..., Qn` are *qualifiers*, with $n \geq 0$. Each qualifier is either a *filter* or a *generator*. A generator has syntax `p ← e` where `e` is a collection-valued expression and `p` is a *pattern* i.e. an expression involving variables and tuple constructors only. The variables of `p` are successively bound by iterating through `e`. Any variables appearing in the head, `h`, inherit these bindings. A filter is a boolean-valued expression, which must be satisfied by the values generated by the generators in order for these values to contribute to the final result of the comprehension.

Thus, the head of a comprehension corresponds to the `SELECT` clause of an SQL or OQL query, the generators to the `FROM` clause, and the filters to the

WHERE clause. Comprehension syntax can thus be used to express select-project-join (SPJ) queries. Comprehensions are a convenient high-level syntax and add no extra expressiveness to functional languages such as IQL (because they can be translated into successive applications of `flatMap`).

IQL supports unification of variables appearing in the patterns of generators within the same comprehension. For example, the following IQL query undertakes a join of tables `r` and `s` over their `a` and `b` attributes:

$$[\{x, y\} \mid \{x, z\} \leftarrow \langle\langle r, a \rangle\rangle; \{y, z\} \leftarrow \langle\langle s, b \rangle\rangle]$$

and is equivalent to the following query:

$$[\{x, y\} \mid \{x, z1\} \leftarrow \langle\langle r, a \rangle\rangle; \{y, z2\} \leftarrow \langle\langle s, b \rangle\rangle; z1 = z2]$$

3.2. Query Optimisation

Query optimisation in AutoMed is performed by the `QueryOptimiser` component (see Section 4.1) which serves as a “policy” class and coordinates the application of a number of individual optimisers. AutoMed users are free to create their own custom `QueryOptimiser` components and their own custom individual optimisers. The termination and confluence of the query optimisation process depend on the specification of the `QueryOptimiser` and the individual optimisers that it invokes.

Below, we discuss six of the individual optimisers provided by AutoMed’s AQP which we have used to create two different optimisation policies for the architectures we present in Section 4. Both these policies are terminating (i.e. the `QueryOptimiser` terminates) and confluent (i.e. if there is a choice at any time as to which individual optimiser to apply to the current query expression, then the final expression output is independent of the choice made). Our individual optimisers apply well-known algebraic equivalences, and we will see that these are sufficient for providing acceptable query performance in our setting. Providing cost-based optimisation could have a further beneficial effect on performance in the longer term. However, at present OGSA-DAI and OGSA-DQP do not expose the necessary metadata to support it. We refer the interested

reader to [55] for details of query optimisation in AutoMed, to [5, 17, 43, 12] for more general discussions of query optimisation in comprehension languages, and to [32, 28, 13] for discussions of query optimisation in distributed and heterogeneous database systems.

Optimiser 1. This optimiser unnests nested comprehensions c.f. unnesting nested SQL queries. It does this by repeatedly applying the following rule to the query tree until there are no more matching instances of the left-hand side of the rule in the tree. The rule renames the variables appearing in the Q_i by matching the patterns p_1 and p_2 , and then eliminates these patterns:

$$[h \mid e_1; p_1 \leftarrow [p_2 \mid Q_1; \dots; Q_n]; e_2] \Rightarrow [h \mid e_1; Q'_1; \dots; Q'_n; e_2]$$

There is a proviso to applying this rule, in that p_1 and p_2 must match i.e. p_1 can be obtained from p_2 by variable renaming. Each Q'_i is obtained from Q_i by applying the same renaming. For example, application of this rule to

$$[\{x, y, z\} \mid x \leftarrow \langle\langle r \rangle\rangle; \{y, z\} \leftarrow [\{v, w\} \mid \{v, i\} \leftarrow \langle\langle s, a \rangle\rangle; \{w, j\} \leftarrow \langle\langle t, b \rangle\rangle]; z > 100]$$

gives:

$$[\{x, y, z\} \mid x \leftarrow \langle\langle r \rangle\rangle; \{y, i\} \leftarrow \langle\langle s, a \rangle\rangle; \{z, j\} \leftarrow \langle\langle t, b \rangle\rangle; z > 100]$$

Optimiser 2. This optimiser rewrites comprehensions containing generators that iterate over an expression which is an append (i.e. $++$) of a number of sub-expressions, into a set of simpler comprehensions. It does this by repeatedly applying the following rule to the query tree until there are no more matching instances of the left-hand side. The rule replaces a comprehension containing a generator that iterates over an append of n sub-expressions by n comprehensions each of which contains a generator iterating over one of the sub-expressions:

$$[h \mid e; p \leftarrow (e_1 ++ \dots ++ e_n); e'] \Rightarrow [h \mid e; p \leftarrow e_1; e'] ++ \dots ++ [h \mid e; p \leftarrow e_n; e']$$

For example, application of this rule to

$$[\{x, y, z\} \mid x \leftarrow \langle\langle r \rangle\rangle; y \leftarrow \langle\langle s \rangle\rangle ++ \langle\langle t \rangle\rangle; z \leftarrow \langle\langle u \rangle\rangle; z > 100]$$

gives:

$$\begin{aligned} & [\{x, y, z\} \mid x \leftarrow \langle\langle r \rangle\rangle; y \leftarrow \langle\langle s \rangle\rangle; z \leftarrow \langle\langle u \rangle\rangle; z > 100] ++ \\ & [\{x, y, z\} \mid x \leftarrow \langle\langle r \rangle\rangle; y \leftarrow \langle\langle t \rangle\rangle; z \leftarrow \langle\langle u \rangle\rangle; z > 100] \end{aligned}$$

We note that this is the equivalent of distributing selections and projections over UNION in the relational algebra. The benefit of this optimisation is that some of the resulting subqueries may refer only to a single data source, and therefore can be sent for full evaluation at that data source. The disadvantage of this optimisation is that the number of subqueries generated for a given input query is $\prod_{i=1}^n m_i$, where m_i is the number of sub-expressions of the i^{th} generator of the input query.

Optimiser 3. This optimiser eliminates comprehensions that it can infer will return empty results because they are undertaking a join over attributes with non-overlapping extents:

$$[h \mid e; p_1 \leftarrow e_1; e'; p_2 \leftarrow e_2; e''] \Rightarrow []$$

Here, the patterns p_1 and p_2 need to have one or more variables in common, and the values within e_1 and e_2 corresponding to these variables need to be known to be non-overlapping. For example, this optimisation can be applied over attributes that are known to have globally unique values over the data sources being integrated e.g. if we know that attributes a and b of relations r and s , respectively, have no values in common, then this expression:

$$[y \mid \{x, y\} \leftarrow \langle\langle r, a \rangle\rangle; \{z, y\} \leftarrow \langle\langle s, b \rangle\rangle]$$

can be replaced by $[]$.

Optimiser 4. This simplifies applications of collection operators to eliminate empty collections, e.g., for any expression Q :

$$Q ++ [] \Rightarrow Q$$

Optimiser 5. This optimiser rewrites a comprehension whose generators refer to j data sources into a comprehension containing j subcomprehensions, each one referencing schema constructs from just a single data source. This grouping of generators and filters that reference the same data source into one subquery allows the query processor to subsequently send that subquery as a whole to the data source for evaluation. As an example, consider the following query over data source schemas S_1 and S_2 , where relations r and t are in S_1 and relation s is in S_2 :

$$[\{y, w, j\} \mid \{x, y\} \leftarrow \langle\langle r, a \rangle\rangle; \{v, w\} \leftarrow \langle\langle s, b \rangle\rangle; \{i, j\} \leftarrow \langle\langle t, c \rangle\rangle; y > 100; w > 200; j > w]$$

This optimiser rewrites the above query into the following:

$$\begin{aligned} &[\{y, w, j\} \mid \{x, y, i, j\} \leftarrow [\{x, y, i, j\} \mid \{x, y\} \leftarrow \langle\langle r, a \rangle\rangle; \{i, j\} \leftarrow \langle\langle t, c \rangle\rangle; y > 100]; \\ &\quad \{v, w\} \leftarrow [\{v, w\} \mid \{v, w\} \leftarrow \langle\langle s, b \rangle\rangle; w > 200]; \\ &\quad j > w] \end{aligned}$$

We note that this is the equivalent of pushing selections and projections through join operations in the relational algebra.

Optimiser 6. Similarly, the operands of collection operators such as $++$ and $--$ may be expressions that refer to different data sources. Grouping together operands that refer to the same data source may enable the query processor of the data integration component to send larger subqueries to the data sources for evaluation. As an example, assuming the same data source schemas and relations as in the previous example, the following query:

$$[y \mid \{x, y\} \leftarrow \langle\langle r, a \rangle\rangle] ++ [w \mid \{v, w\} \leftarrow \langle\langle s, b \rangle\rangle] ++ [j \mid \{i, j\} \leftarrow \langle\langle t, c \rangle\rangle]$$

would be rewritten to:

$$[y \mid \{x, y\} \leftarrow \langle\langle r, a \rangle\rangle] ++ [j \mid \{i, j\} \leftarrow \langle\langle t, c \rangle\rangle] ++ [w \mid \{v, w\} \leftarrow \langle\langle s, b \rangle\rangle]$$

We observe that the first four optimisers are applicable to any query processing setting, while the last two are driven particularly by the requirements of a data integration setting and they aim to produce the largest possible subqueries that can be sent for local evaluation at the data sources.

The first four optimisers operate in tandem, as will become apparent in Sections 5.3 and 5.4. In particular, it is common after the query reformulation stage for comprehensions to contain generators that iterate over an expression that appends a number of

further comprehensions. Optimiser 2 can be applied to split such queries into a number of simpler subqueries. Optimiser 1 can then be applied to eliminate any nested comprehensions within these subqueries. Then Optimisers 3 and 4 can be applied to prune the number of subqueries. An example of the optimisation process is given in Section 5.3.

Optimisers 5 and 6 aim to minimise the amount of post-processing required by the query processor of the data integration component. They are useful for architectures that do not contain a distributed query processing component, as we will see in the next section.

4. Our Architectures

We have developed an architecture, first presented in [54], for the virtual integration of OGSA-DAI enabled data sources. Our architecture uses AutoMed to produce the schema of the integrated resource and to reconcile the data sources both at a schema and at a data level, and uses OGSA-DQP to undertake distributed query processing over the integrated resource. We describe this architecture again here, for completeness, and we then describe also two variants of it that firstly drop the OGSA-DQP middleware, and secondly drop also the OGSA-DAI services. The purpose of these two variants is to enable investigation of the performance impact of each component of our overall architecture. In particular, Section 5 describes a range of experiments conducted on all three architectures and compares between them.

4.1. Complete Architecture

Figure 3 illustrates our complete architecture. In general, each data source is located on a different Grid node, together with an OGSA-DAI service and an OGSA-DQP Evaluator service (GQES) deployed over it. AutoMed and OGSA-DQP's Coordinator service (GDQS) should preferably be deployed on the same Grid node to avoid data transfer costs between them. This node could be one of the data source nodes, or a separate node.

We now first discuss the integration of data sources using AutoMed and OGSA-DAI, and then discuss query processing using AutoMed, OGSA-DAI and OGSA-DQP. The integration process is common in all three architectures, and is therefore omitted in Sections 4.2 and 4.3.

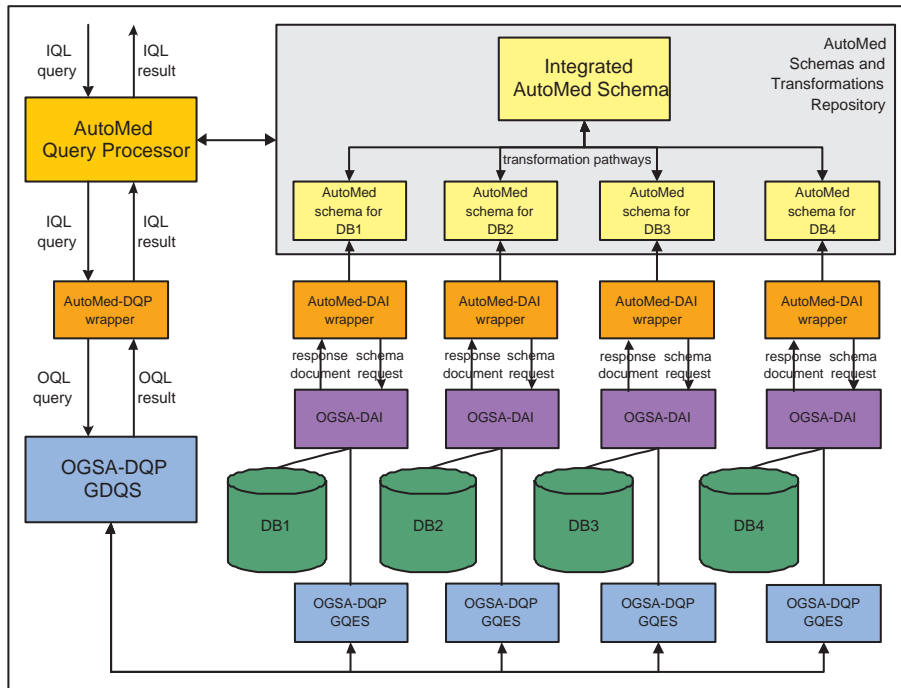


Figure 3: OGSA-DAI/OGSA-DQP/AutoMed Architecture

Each data source is wrapped by an OGSA-DAI service allowing retrieval of schema metadata and submission of queries to the associated data source. Each AutoMed-DAI Wrapper interfaces between an OGSA-DAI data source and AutoMed's Schemas & Transformations Repository (STR), in order to automatically retrieve schema metadata from a data source, via the OGSA-DAI service, and create the corresponding AutoMed schema in the STR. These data source schemas are then transformed into an integrated schema either by manual submission of the appropriate transformation pathways (via the API of AutoMed's STR) or by using schema matching [46] or schema transformation tools [56] that semi-automatically generate transformation pathways. The integrated schema may be defined *a priori*, or may be created incrementally e.g. by starting off with one of the data source schemas and extending this as necessary.

Adding a new data source to a set of already integrated resources would require an OGSA-DAI and an OGSA-DQP GQES service to be deployed over it, and also a new OGSA-DQP GDQS service to be deployed that accesses also the new data source

and the newly deployed GQES service. An instance of the AutoMed-DAI wrapper would be created accessing the newly deployed OGSA-DAI service, and this would be used to retrieve the schema metadata of the new data source and automatically create the corresponding AutoMed schema in the STR. The appropriate transformations from this schema to the existing global schema would then be issued (via the API of AutoMed’s STR). If necessary, the existing global schema would be extended with new entities and/or new attributes for existing entities (again, this would be via the API of AutoMed’s STR) in order to encompass new constructs contributed by the new data source. In this case, the transformation pathways from the existing data source schemas to the original global schema would be extended by the necessary **extend** primitive transformations in order to link them to the new global schema (we recall from Section 3 that **extend** transformations are used when it is not possible to derive any information about a global schema construct from a source schema).

Queries can be submitted to AutoMed for evaluation against an integrated schema. Such queries can be expressed directly in IQL, or can be expressed in a high-level query language — SQL and XQuery are currently supported — in which case they are first translated by AutoMed into IQL⁸.

An IQL query, Q , on an integrated schema is processed by AutoMed’s **Query Processor (AQP)** as follows (see Figure 4). First, the AQP’s **QueryReformulator** component reformulates Q , using the transformation pathways stored in the STR, into an equivalent query, Q_{ref} , referencing only data source schema constructs. For our performance study below only GAV query reformulation (i.e. query unfolding) is deployed, since the ISPIDER integrated schema is defined using GAV rules only. To illustrate this reformulation, returning to the example schemas and transformation pathways in the introductory section of Section 3, the following query Q could be posed on the integrated schema IS to find the names and departments of all students:

$$[\{n, d\} \mid \{s, n\} \leftarrow \langle\langle \text{Students}, \text{name} \rangle\rangle; \{s, d\} \leftarrow \langle\langle \text{Students}, \text{dept} \rangle\rangle]$$

Applying GAV query reformulation uses the **add** and **extend** steps of the transformation pathways from the data source schemas to IS to replace references to constructs of IS in Q by their definitions in terms of the data source schemas (note that in the

⁸The SQL-to-IQL translator supports nested select-project-join-union queries, aggregation functions, and GROUP BY. The XQuery-to-IQL translator supports FLWR queries.

reformulated query all substituted definitions are prefixed by their data source schema name):

$$\begin{aligned} [\{n, d\} \mid \{s, n\} \leftarrow (S1 : \langle\langle \text{StudentDetails}, \text{name} \rangle\rangle) ++ S2 : [] ++ S3 : []); \\ \{s, d\} \leftarrow (S1 : [] ++ S2 : [\{s, \text{'Maths'}\} \mid s \leftarrow \langle\langle \text{MathsStudents} \rangle\rangle]) ++ \\ S3 : [\{s, \text{'CompSci'}\} \mid s \leftarrow \langle\langle \text{CSStudents} \rangle\rangle)] \end{aligned}$$

The **VariableUnification** component then makes variable equality explicit within Q_{ref} , as discussed in Section 3.1. Applied to the above reformulated query, this would result in:

$$\begin{aligned} [\{n, d\} \mid \{s1, n\} \leftarrow (S1 : \langle\langle \text{StudentDetails}, \text{name} \rangle\rangle) ++ S2 : [] ++ S3 : []); \\ \{s2, d\} \leftarrow (S1 : [] ++ S2 : [\{s, \text{'Maths'}\} \mid s \leftarrow \langle\langle \text{MathsStudents} \rangle\rangle]) ++ \\ S3 : [\{s, \text{'CompSci'}\} \mid s \leftarrow \langle\langle \text{CSStudents} \rangle\rangle)] \\ s1 = s2] \end{aligned}$$

Next, the **QueryOptimiser** component optimises Q_{ref} by applying Optimisers 1-4 of Section 3.2, resulting in a query Q_{opt} — we give an example of this process for a query over the actual ISPIDER resource in Section 5.3. The **QueryAnnotator** component then annotates subqueries with details of the appropriate wrapper object, creating a single query plan, Q_{ann} , which is passed to the **QueryEvaluator** component for evaluation — again, we give an example of this process in Section 5.3.

For the specific architecture of Figure 3, there is only one kind of wrapper available to the AQP, namely the AutoMed-DQP wrapper, as we are aiming to deploy the distributed and parallel query processing capabilities of OGSA-DQP over the integrated resource: the AQP does not directly interact with the data sources, only OGSA-DQP does.

Each subquery submitted to an AutoMed-DQP wrapper object is first translated by it into OQL (the query language supported by OGSA-DQP 3.1) and is then submitted to OGSA-DQP for evaluation. In particular, OGSA-DQP's GDQS processes the input OQL query and produces a query plan, which is evaluated using one or more GQES services. The result of this OQL query is then translated into an IQL result by the AutoMed-DQP wrapper and this is returned to the **QueryEvaluator**⁹. This is re-

⁹Note that the performance impact of the translation of an OQL query into an IQL query and of the OQL results back to IQL is not substantial compared to the overall performance impact of OGSA-DQP.

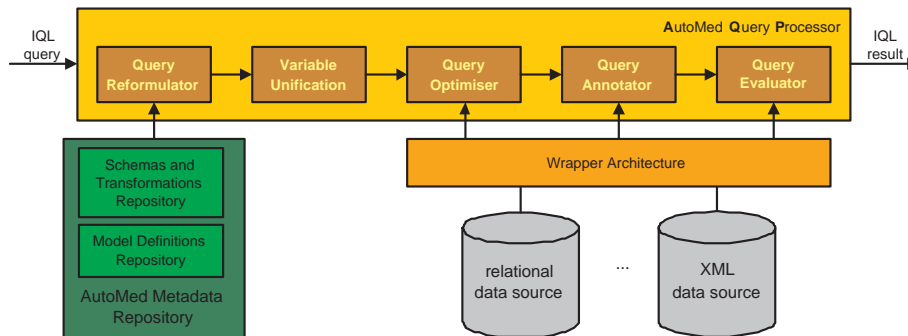


Figure 4: The AutoMed Architecture

responsible for any necessary post-processing of the wrapper results to produce the overall query result returned to the client application: because OGSA-DQP 3.1 supports a subset of the OQL query language¹⁰, and IQL is more expressive than this subset, the original query Q submitted to AutoMed’s AQP may not be fully translatable into a single OQL query, and hence there may be a need for such post-processing.¹¹

We finally note that the AQP undertakes centralised query processing whereas OGSA-DQP offers decentralised query processing: subqueries within a query plan produced by AutoMed’s `QueryAnnotator` component are evaluated by AutoMed-DQP wrapper objects that are located within AutoMed; in contrast, with OGSA-DQP each query partition is evaluated by a GQES service and these distributed services are able to interact with each other and with the GDQS service.

4.2. Architecture using AutoMed and OGSA-DAI

This architecture is similar to the previous one, but without the OGSA-DQP GDQS and GQES services — see Figure 5. Thus, the integration process is the same,

¹⁰The limitation of this subset of OQL that is relevant here and that required the development of the optimisers of Section 3.2 is support for only one level of nesting. For a discussion on the queries allowed by OGSA-DQP 3.1, please see http://www.ogsadai.org.uk/documentation/ogsa-dqp_3.1/query-guide.txt.

¹¹The same would apply for OGSA-DQP 3.2, since this supports a subset of SQL and IQL is more expressive than this subset. For example, the SQL subset supported in OGSA-DQP 3.2 does not support nested queries with correlated variables in the WHERE clause and also does not support the AS keyword, which further limits the ability to express nested queries.

but query processing is performed solely by AutoMed’s AQP in a centralised fashion. In particular, an IQL query submitted to an integrated schema is processed by the AQP in the same way as before, producing queries Q_{ref} , Q_{opt} and Q_{ann} . But each subquery within Q_{ann} is now submitted to an AutoMed-DAI wrapper, rather than an AutoMed-DQP wrapper. The AutoMed-DAI wrapper translates the subquery into an SQL query, and submits it to an OGSA-DAI service for evaluation. As before, AutoMed’s `QueryEvaluator` component post-processes the results returned by the wrappers and produces the overall query result.

Since this architecture does not use a distributed query processing component, we employ a different optimisation policy than in the previous architecture. This policy uses all six optimisers of Section 3.2, aiming to send larger subqueries to the data sources for local evaluation, and thus reduce the amount of post-processing required by AutoMed’s AQP. Therefore, AutoMed’s `QueryOptimiser` component may produce a different Q_{opt} query for the same reformulated query Q_{ref} compared to the previous architecture.

OGSA-DAI services pass on SQL queries to the data source DBMSs and do not impose any constraints on these queries — so query language translation only depends on the translation capabilities of the AutoMed-DAI wrapper. This is able to translate IQL comprehensions that are arbitrarily nested, and the `++` IQL operator, and so outputs possibly nested SPJU SQL queries.

4.3. Architecture using only AutoMed

This architecture is similar to the previous one, but accesses the data sources directly, rather than through OGSA-DAI services — see Figure 6. The integration process is the same as before, and the difference in query processing is that the appropriate AutoMed wrappers for the data sources are deployed, rather than AutoMed-DAI wrappers.

AutoMed currently supports relational data sources via JDBC, XML data sources via the XML:DB API, OWL-Lite and RDF/S documents, and structured text files (e.g. CSV files). In particular, the AutoMed-JDBC wrapper is able to translate possibly nested SPJU IQL queries into possibly nested SPJU SQL queries (but it could also be extended support additional SQL extensions for particular DBMSs). We therefore note that the AutoMed-JDBC wrapper and the AutoMed-DAI wrapper

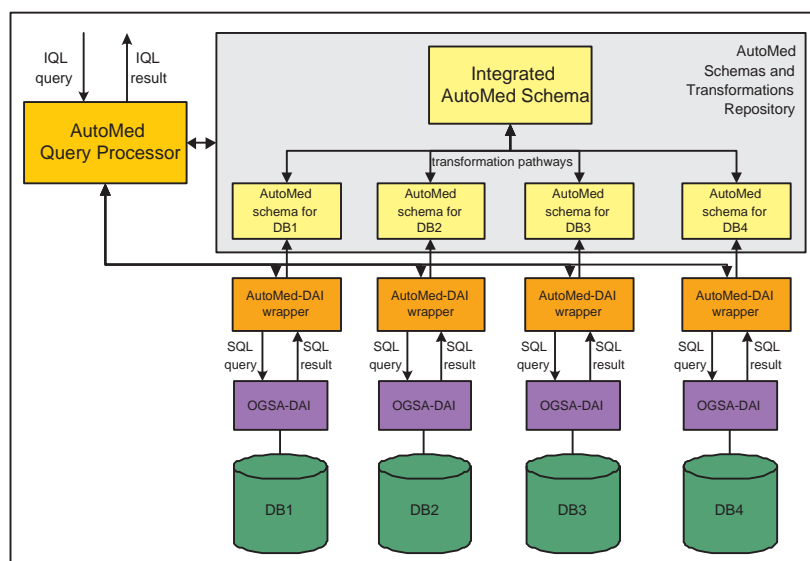


Figure 5: OGSA DAI/AutoMed Architecture

discussed in Section 4.2 have the same translation capabilities. Since OGSA-DAI services encapsulate JDBC functionality, the only notable difference between these two architectures (as we will see below) is in performance, due to the overhead introduced by OGSA-DAI services.

5. Performance Evaluation

This section presents our performance evaluation of the three architectures discussed above in the context of a real-world biological data integration project, namely ISPIDER. We first describe the ISPIDER integration setting and then discuss the experiments performed and results obtained.

In the following, A_1 refers to the AutoMed-only architecture of Section 4.3, A_2 to the architecture of Section 4.2 combining AutoMed and OGSA-DAI, and A_3 to the full architecture of Section 4.1.

5.1. Case Study: The ISPIDER Integrated Resource

The In Silico Proteome Integrated Data Environment Resource (ISPIDER) project has developed an integrated platform of proteome-related resources, using existing

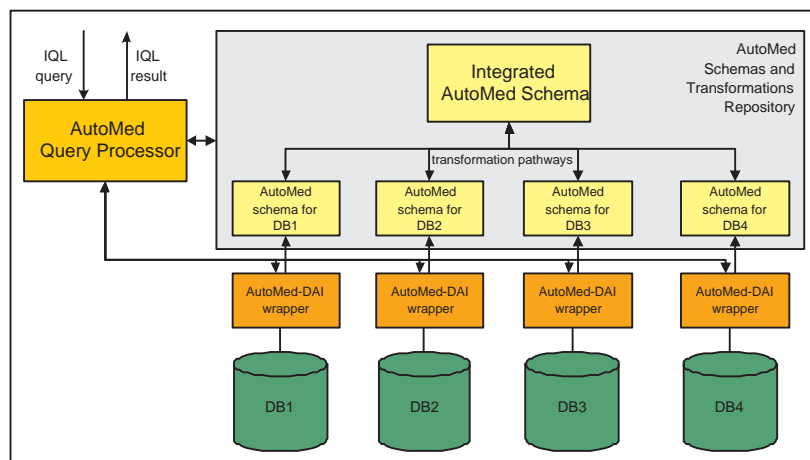


Figure 6: AutoMed-only Architecture

standards from proteomics, bioinformatics and e-Science [49]. As part of this Grid platform, we have developed an integrated resource of four data sources containing experimental proteomics data. As discussed in [54], the integration of these data sources is beneficial for proteomics researchers for a number of reasons: having access to more data leads to more reliable analyses, e.g. by reducing the chances of false negatives in user queries; bringing together data sources containing different but closely related data increases the breadth of information the biologist has access to; and finally the virtual integration of these data sources, as opposed to merely providing a common interface for accessing them, enables data from a range of experiments, tissues, or different cell states to be brought together in a form which may be analysed by a biologist in spite of the widely varying coverage and underlying technology of each data source.

The four autonomous proteomics databases (all MySQL) that we have integrated are as follows:

- **PEDRo** [21] provides descriptions of experimental data sets in proteomics. The PEDRo version used for ISPIDER contains a modest number of experiments (2.5Mb of data), but was significant in the ISPIDER project because of its comprehensive schema, which served as a starting point for the integration process. More generally, PEDRo is also used as a format for exchanging proteomics

data, and in this respect has influenced the standardisation activities of the PSI (Proteomics Standards Initiative, <http://psidev.sourceforge.net>).

- **gpmDB** [10] provides a wealth of peptide/protein identifications from a range of different laboratories and instruments. For ISPIDER, the research version of gpmDB was used, which contains over 41,000,000/7,000,000 peptide/protein hits (over 1,000,000/330,000 distinct ones) within a total of 5.6Gb of data.
- **PepSeeker** [42] captures identification allied to peptide sequence data, coupled with the underlying ion series. ISPIDER used the first version of PepSeeker, containing over 185,000/135,000 peptide/protein hits (over 49,000/47,000 distinct ones) within a total of 4.2Gb of data.
- **PRIDE** [31] is a publicly available repository for proteomics data, containing protein and peptide identifications and post-translational modifications identified on individual peptides. The PRIDE version used in ISPIDER currently contains a modest number of experiments (2.5Mb of data), but it is ultimately expected to mirror all public data of the European Bioinformatics Institutes's PRIDE database, as well as data from the University of Manchester.

Figure 7 illustrates the integrated schema of the ISPIDER virtual integrated resource. The design of this schema started with a copy of the PEDRo schema as a first version, IS_1 . Corresponding attributes from PEDRo and the other three schemas were mapped to the attributes of IS_1 by creating the necessary AutoMed transformation pathways. The entities of IS_1 were then extended with additional attributes not present in the PEDRo schema but derivable from one or more of the other three schemas, again creating the necessary AutoMed transformation pathways and resulting in schema version IS_2 . Finally, additional entities and their attributes not present in PEDRo but derivable from the other schemas were added to IS_2 , again creating the necessary transformation pathways and resulting in the final integrated schema.

This integrated resource is a typical example of data integration in the Life Sciences domain: there are multiple schema-level and data-level modelling conflicts in the data sources, and these need to be resolved by transforming and combining the extents of individual attributes of the data source entities to form the extent of each attribute of each entity in the integrated schema; there is a mixture of large, community data sources (gpmDB, PepSeeker) and of smaller data sources produced by smaller groups of researchers (PEDRo, PRIDE); and there is an absence of commonly agreed identifiers

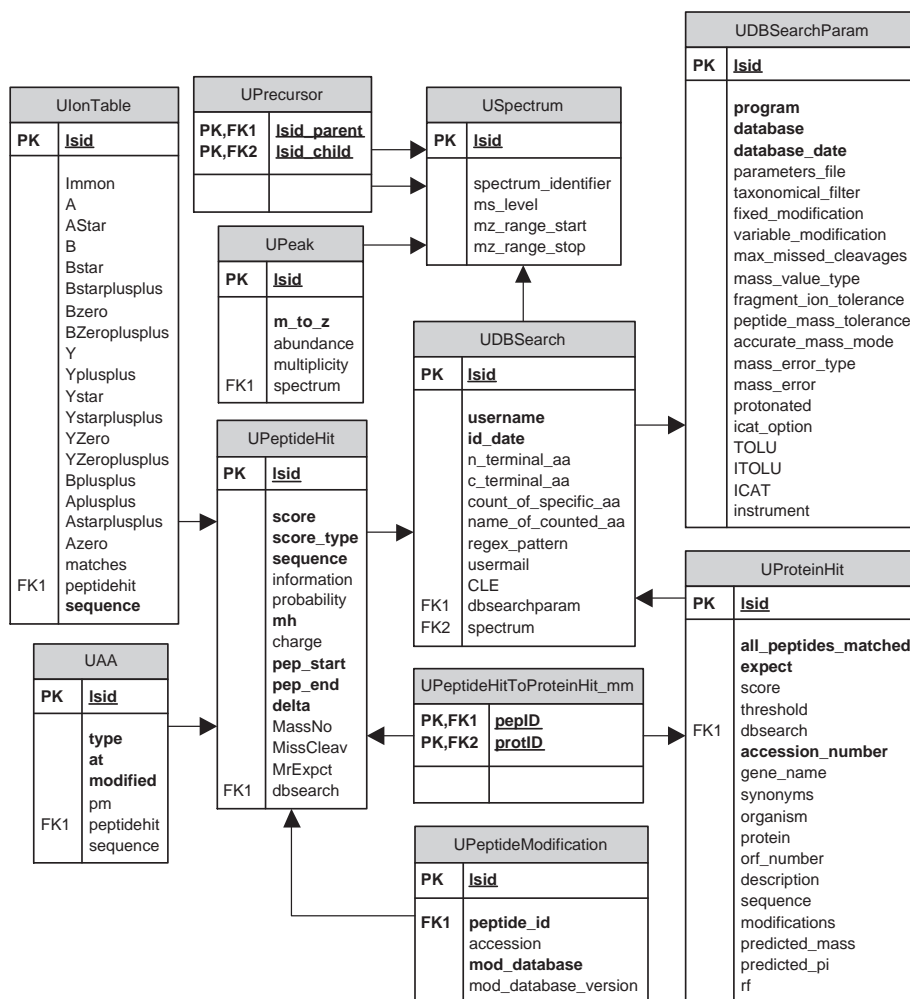


Figure 7: The ISPIDER Integrated Schema

for instances of the biological entities in the different data sources.

This last characteristic is a long-standing problem in the Life Sciences, where it is common practice to use identifiers which are unique only within the specific resource. To identify entity instances in our integrated schema, we therefore generated life science identifiers, *LSIDs* [7], from the data source-specific identifiers: LSID is a uniform resource name (URN) specification providing a standardised naming scheme for entities in the life sciences domain. For example, the LSID *URN:LSID:ispider.man.ac.uk:pedro.proteinhit:99*

refers to the row with primary key value *99* in the table `proteinhit` of the PEDRo database, where *ispider.man.ac.uk* denotes the LSID issuing authority.

A consequence of using LSIDs for preventing conflicts at the level of the integrated schema is that joins over the primary key attributes of integrated schema tables can only result in matches *within* data sources — and not *between* data sources. Thus, the conditions for applying Optimiser 3 (discussed in Section 3.2) hold for such joins, which is significant in terms of query processing and query optimisation, as will be discussed shortly.

5.2. Experimental Set-Up

We have conducted an evaluation of the query performance of all three architectures presented in Section 4. Table 1 summarises the hardware and software characteristics of the Grid used for our evaluation. Each of the MySQL databases is hosted on a different node, and is accessed via JDBC for A_1 , and via OGSA-DAI hosted on Apache Tomcat for A_2 and A_3 . Each of the Grid nodes is provided with enough memory to avoid paging during query execution, whose effects would otherwise dominate the performance impact of the architectural differences we wish to evaluate. AutoMed and OGSA-DQP are hosted on the same Grid node, although this is not mandatory in general, to avoid data transfer costs between them. This node is equipped with a dual core processor and 4Gb of RAM. Due to the software requirements imposed by OGSA-DQP, Java 1.4.2 has been used, which comes only in a 32-bit version, and so it has not been possible to exploit the full power of the 64-bit processor. Throughout the performance study, each pair of Grid nodes is linked with a 100Mbps connection, apart from our study of the impact of network speed, which is reported in Section 5.7.

Our performance evaluation has been conducted using three different sets of queries, investigating one performance factor at a time. For each query, we measure the time taken by:

- (i) AQP set-up: this includes the time spent initialising the AQP, establishing connections with the data sources using the appropriate wrapper instances, and reformulating the query;
- (ii) AQP optimisation: this includes the time spent by the `QueryOptimiser` and `QueryAnnotator` components;

Table 1: Evaluation PCs: Hardware, Software and Operating System Characteristics

OS	Hardware	Software
Linux (Fedora Core 4)	Athlon Dual Core (64-bit, 2.2GHz/ 4Gb RAM)	AutoMed toolkit (Java 1.4.2, 32-bit) AutoMed Repository (PostgreSQL 8) OGSA-DQP 3.1 (Apache Tomcat 5)
MS Windows (XP prof.)	Pentium 4 Dual Core (3GHz/2Gb RAM)	gpmDB (MySQL 4.1) OGSA-DAI 2.2 (Apache Tomcat 5)
MS Windows (XP prof.)	Pentium 4 (2.4GHz/1.5Gb RAM)	PepSeeker (MySQL 4.1) OGSA-DAI 2.2 (Apache Tomcat 5)
MS Windows (XP prof.)	Pentium 4 (2.4GHz/1.5Gb RAM)	PRIDE (MySQL 4.1) OGSA-DAI 2.2 (Apache Tomcat 5)
MS Windows (XP prof.)	Pentium 3 (864MHz/256Mb RAM)	PEDRo (MySQL 4.1) OGSA-DAI 2.2 (Apache Tomcat 5)

- (iii) AQP evaluation: this is the time spent by the AQP’s **QueryEvaluator** to post-process subquery results;
- (iv) the wrappers: this is the time spent by wrapper instances to evaluate the subqueries delegated to them, including the time spent on subquery evaluation and by OGSA-DQP, OGSA-DAI and the data sources themselves.

Each query is executed 10 times, and the medians of the times (i) - (iv) are computed. Prior to these 10 executions, a “warm-up” query is first run, allowing the initialisation of several internal caches, including those of the **QueryReformulator** (retrieving and caching the transformation pathways from AutoMed’s Schemas & Transformations Repository and generating the GAV view definitions of the global schema constructs) and the **AutoMedWrapper** instances (creating connections with the data sources used for connection pooling), but not precomputing or caching any query results. This warm-up query takes less than 10 seconds. As a result of this warm-up query, the set-up time for all the evaluation queries is close to zero and the time spent in the **QueryAnnotator** component does not include any creation of **AutoMedWrapper** instances. After each of the 10 query executions, all AQP objects are marked for deletion and the JVM garbage collector is invoked. As a result, only the transformation pathways and the connections to the data sources remain cached between successive

query executions.

In the remainder of this section, Section 5.3 presents our performance evaluation using a set of biologically meaningful queries provided by the ISPIDER domain experts. These user queries do not require any distributed join processing and so in Section 5.4 we present experimental results using a second set of queries that do require joins between different data sources. Section 5.5 then repeats the experiments with these two query sets but this time using the incremental query processing capabilities of AutoMed, OGSA-DAI and OGSA-DQP. Section 5.6 next evaluates the effect of parallel query processing using the two query sets as well as a third set of queries that are more suitable for investigating the impact of parallel execution. Section 5.7 investigates the impact of network speed on query processing in all three architectures. Finally, Section 5.8 summarises our findings and also discusses their more general applicability.

5.3. User-Provided Queries

Our first set of experiments was with a set of queries provided by our biologist and bioinformatician partners in the ISPIDER project:

- Q^1 Retrieve all protein identifications for a given protein accession number
- Q^2 Retrieve all protein identifications for a given group of proteins
- Q^3 Retrieve all protein identifications for a given organism
- Q^4 Retrieve all protein identifications given a certain peptide
- Q^5 Retrieve all identifications of a given protein given a certain peptide
- Q^6 Retrieve all peptide-related information for a given protein identification

The IQL encodings of these queries are listed in Table 2. In IQL, schema constructs are identified by means of their *scheme* enclosed within double chevrons $\langle\langle \dots \rangle\rangle$. For example, query Q^1 iterates through the `accession_number` attribute of the global table `UProteinHit`, and returns the `lsid` value corresponding to a specific accession number. Query Q^2 returns the `accession_number` and `lsid` values of entries from `UProteinHit` whose description value matches the string `'%Actin%'`. Similarly Q^3 returns the `accession_number` and `lsid` values of entries from `UProteinHit` whose organism value matches `'%gallus%'`. We note that queries Q^2 and Q^3 check for membership within a nested subquery rather than undertaking a join. This is because reformulation of the join versions of Q^2 and Q^3 would result in self-joins on the data source tables and OGSA-DQP

Table 2: User-provided Queries

Q^1 : $\{\{an, lsid\} \{lsid, an\} \leftarrow \langle\langle UProteinHit, accession_number \rangle\rangle; an = 'ENSP00000339074'\}$
Q^2 : $\{\{an, lsid\} \{lsid, an\} \leftarrow \langle\langle UProteinHit, accession_number \rangle\rangle;$ $member [lsid \{lsid, d\} \leftarrow \langle\langle UProteinHit, description \rangle\rangle; like d '%Actin%'] lsid]$
Q^3 : $\{\{an, lsid\} \{lsid, an\} \leftarrow \langle\langle UProteinHit, accession_number \rangle\rangle;$ $member [lsid \{lsid, o\} \leftarrow \langle\langle UProteinHit, organism \rangle\rangle; like o '%sapiens%'] lsid]$
Q^4 : $\{\{pr, sc\} \{lsid1, pr\} \leftarrow \langle\langle UProteinHit, protein \rangle\rangle;$ $\{lsid2, seq\} \leftarrow \langle\langle UPeptideHit, sequence \rangle\rangle; seq = 'ATLTSDK';$ $\{peplD, protID\} \leftarrow \langle\langle UPeptideHitToProteinHit_mm \rangle\rangle;$ $lsid2 = peplD; lsid1 = protID;$ $\{lsid2, sc\} \leftarrow \langle\langle UPeptideHit, score \rangle\rangle]$
Q^5 : $\{\{an, lsid1, sc\} \{lsid2, seq\} \leftarrow \langle\langle UPeptideHit, sequence \rangle\rangle; seq = 'LVNELTEFAK';$ $\{lsid1, an\} \leftarrow \langle\langle UProteinHit, accession_number \rangle\rangle; an = 'gi-229552';$ $\{peplD, protID\} \leftarrow \langle\langle UPeptideHitToProteinHit_mm \rangle\rangle;$ $lsid2 = peplD; lsid1 = protID;$ $\{lsid2, sc\} \leftarrow \langle\langle UPeptideHit, score \rangle\rangle]$
Q^6 : $\{\{an, seq, sc, pr, dbs\} \{lsid1, an\} \leftarrow \langle\langle UProteinHit, accession_number \rangle\rangle;$ $lsid1 = \{'URN:LSID:ispider.man.ac.uk:pedro', 1069\};$ $\{peplD, protID\} \leftarrow \langle\langle UPeptideHitToProteinHit_mm \rangle\rangle;$ $lsid1 = protID;$ $\{lsid2, seq\} \leftarrow \langle\langle UPeptideHit, sequence \rangle\rangle; lsid2 = peplD;$ $\{lsid2, sc\} \leftarrow \langle\langle UPeptideHit, score \rangle\rangle;$ $\{lsid2, pr\} \leftarrow \langle\langle UPeptideHit, probability \rangle\rangle;$ $\{lsid1, dbs\} \leftarrow \langle\langle UProteinHit, dbsearch \rangle\rangle]$

3.1 does not support self-joins¹². Query Q^4 iterates through (i) the protein attribute of `UProteinHit`, (ii) the sequence and score attributes of `UPeptideHit`, selecting a specific sequence, and (iii) the key attributes of `UPeptideHitToProteinHit_mm`; and joins these three tables on their key attributes. Queries Q^5 and Q^6 are similar.

To illustrate query processing over the integrated ISPIDER resource, consider query Q^1 . After reformulation, this becomes query Q_{ref}^1 below¹³. The IQL variables starting with a dollar character are system-generated variables generated by the `VariableUnification` or the `QueryOptimiser` component:

¹²For queries such as Q^2 and Q^3 , it is also necessary to enable in the Data Integration component Optimiser 5 discussed in Section 3.2, so as to create a second level of nesting and prevent their subsequent unnesting and self-join generation by DQP, which DQP would then not be able to evaluate.

¹³Here, we use the shorthand 'pride' rather than the full LSID 'URN : LSID : ispider.man.ac.uk:pride' for presentational clarity, and similarly for the other data sources.

```

[{$an, $lsid}|{$lsid, $an} ← ({{{'pride', $k}, $x}|{$k, $x} ← pride : <<identification, accession_number>>)}
  ++ [{{{'gpmdb', $pid}, $x}|{$pid, $proseqid} ← gpmdb : <<protein, proseqid>>)}
    {proseqid1, $x} ← gpmdb : <<proseq, label>>)}
    proseqid = $proseqid1)
  ++ [{{{'pedro', $phid}, $x}|{$phid, $pid} ← pedro : <<proteinhit, protein>>)}
    {pid1, $x} ← pedro : <<protein, accession_num>>)}
    pid = $pid1)
  ++ [{{{'pepseeker', $d}, $x}|{$d, $x} ← pepseeker : <<proteinhit, ProteinID>>)}];
an = 'ENSP00000339074']

```

We notice that the reformulated query contains a union of four subqueries (we recall that ++ is IQL's union operation), each of which undertakes a select-project-join (SPJ) query on one of the data sources. This is because the queries within the transformation pathways that populate the ISPIDER integrated resource from the data sources are themselves all SPJ queries (there was no need to use grouping or aggregation operators for the data source integration, although these operators are supported by AutoMed and IQL).

Query Q_{ref}^1 is then optimised by the **QueryOptimiser** component using the optimisers discussed in Section 3.2. The resulting query Q_{opt}^1 , which is an improvement over Q_{ref}^1 because it is not nested, is given below:

```

[{$x2, {'pride', $k1}|{$k1, $x2} ← pride : <<identification, accession_number>>)}
  $x2 = 'ENSP00000339074'] ++
[{$x6, {'gpmdb', $pid3}|{$pid3, $proseqid4} ← gpmdb : <<protein, proseqid>>)}
  {proseqid5, $x6} ← gpmdb : <<proseq, label>>)}
  proseqid4 = $proseqid5; $x6 = 'ENSP00000339074'] ++
[{$x10, {'pedro', $phid7}|{$phid7, $phid8} ← pedro : <<proteinhit, protein>>)}
  {pid9, $x10} ← pedro : <<protein, accession_num>>)}
  $phid8 = $pid9; $x10 = 'ENSP00000339074'] ++
[{$x12, {'pepseeker', $d11}|{$d11, $x12} ← pepseeker : <<proteinhit, ProteinID>>)}
  $x12 = 'ENSP00000339074']

```

The **QueryAnnotator** then traverses Q_{opt}^1 and identifies the largest subqueries that can be evaluated by **AutoMedWrapper** instances. The annotated query Q_{ann}^1 is then submitted to the **QueryEvaluator**. For architectures A_1 and A_2 , this annotated query contains four **AutoMedWrapper** instances, one for each of the comprehensions appearing in query Q_{opt}^1 above, since each such subquery refers to a different data source. For architecture A_3 the annotated query similarly has four instances of the OGSA-DQP wrapper: OGSA-DQP 3.1 does not support UNION and thus four separate queries need to be submitted to AutoMed-DQP wrapper instances by AutoMed's **QueryEvaluator**.

The other user-provided queries result in similarly annotated queries, i.e. containing up to four comprehensions being unioned together. Some queries contain fewer comprehensions, since not all data sources contribute to all integrated schema constructs. For example, Q^2 results in a single comprehension since only PEDRo contributes to the construct $\langle\langle \text{UProteinHit}, \text{description} \rangle\rangle$.

Figure 8 shows the running times for the user-provided queries using all three architectures, and splitting the execution times into the four parts discussed earlier. The result sizes of queries Q^1 to Q^6 in terms of the number of tuples returned are 261, 4, 8, 35, 64 and 1, respectively. We see that architecture A_2 is slightly slower than A_1 , which is to be expected since the only difference between them is that A_2 is service-based and wraps JDBC functionality using OGSA-DAI, whereas A_1 uses JDBC directly to access the data sources. Architecture A_3 is significantly slower than the other two. This can be attributed to the relatively inefficient performance of OGSA-DQP version 3.1’s incremental query processing, as noted in Section 2. This set of user-supplied queries do not require distributed join processing and thus cannot exploit this significant aspect of DQP’s capabilities.

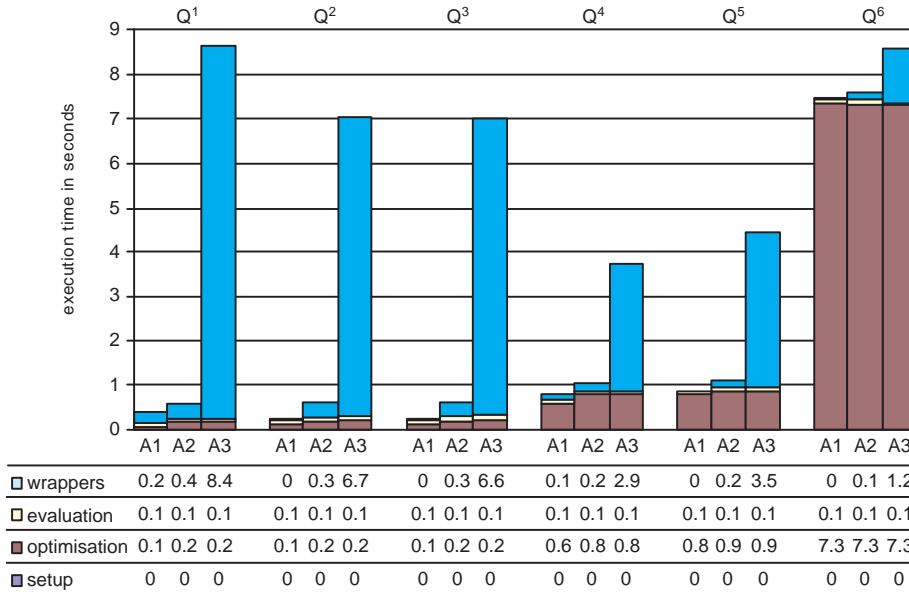


Figure 8: Evaluation of User Queries for Architectures A_1 (left), A_2 (middle) and A_3 (right).

The reformulated and optimised versions of these user-supplied queries generally make use of the IQL ++ (union) operator, which is not supported by OGSA-DQP 3.1 (but is supported by version 3.2). However, we see that the post-processing time spent in AutoMed’s AQP (i.e. the evaluation times in Figure 8) is generally low, and so use of OGSA-DQP 3.2 would result in only slightly better query performance for this set of queries. In contrast, Section 5.4 below explores a set of queries whose reformulated and optimised versions include nested subqueries within the generators of comprehensions (i.e. within **FROM** clauses). For such queries, the use of OGSA-DQP 3.2 (which does not support nested subqueries within **FROM** clauses) would have had a significantly adverse effect on performance.

We finally note that optimisation for query Q^6 takes more than 7 seconds. This query involves a join over 6 schema constructs in the integrated schema. After reformulation, these generators are sourced from 4, 3, 4, 2, 3 and 2 data sources respectively. If optimisation was not performed, all data for each of these constructs would be retrieved from these data sources, incurring a huge data transfer and evaluation cost (much greater than 7 seconds). However, the use of LSIDs means that joins over primary key attributes in the integrated schema (as is the case with Q^6) can only result in matches *within* individual data sources — not *between* different data sources. Opt. 1 and Opt. 3 (discussed in Section 3.2) do indeed eliminate those join combinations that would return an empty result, but take several seconds to do so, because Opt. 2 produces $\prod(4 * 3 * 4 * 2 * 3 * 2) = 576$ subqueries.

This finding, and also the non-negligible optimisation times for Q^4 and Q^5 , points to the need for more work on improving query optimisation performance in AutoMed, e.g. by caching previous/intermediate optimisation results, and by parallelising the successive traversal and rewriting of query trees during the optimisation process.

5.4. Distributed Join Queries

The first set of queries above suggests that all three architectures are suitable for evaluating SPJU queries on the global schema, but that using OGSA-DQP does introduce a performance penalty. However, this set of queries is not appropriate for fully evaluating the query performance of OGSA-DQP because none of the user-provided queries requires distributed joins between the data sources. We therefore devised a second set of queries, shown in Table 3, which, after reformulation and optimisation, do require the evaluation of equijoins between different data sources.

Query Q^{7a} results in a query performing an equijoin over the PRIDE and PepSeeker data sources, query Q^{8a} results in a query performing an equijoin over gpmDB and PEDRo, and Query Q^{9a} results in an equijoin over queries Q^{7a} and Q^{8a} . Queries Q^{7b}/Q^{7c} , Q^{8b}/Q^{8c} and Q^{9b}/Q^{9c} are similar to Q^{7a} , Q^{8a} and Q^{9a} , respectively, but the number of tuples that are input to the join operator after the application of the filters within the comprehensions is larger, as illustrated in Table 4. For example, the join operation in query Q^{7a} is passed 4 tuples from PRIDE, after the application of filter $\text{spi} = 1645$, and 559 tuples from PepSeeker, after the application of filter $m > 1600$, whereas the join operation in query Q^{7b} is passed 1,374 tuples from PRIDE, after the

Table 3: Distributed Join Queries

Q^{7a} : $\{ \{m\} \{sp, spi\} \leftarrow \langle \langle \text{USpectrum}, \text{spectrum_identifier} \rangle \rangle; \text{spi} = 1645;$ $\{ \{peph1, peph2\}, m \} \leftarrow \langle \langle \text{UPeptideHit}, \text{MassNo} \rangle \rangle; m > 1600; m = \text{spi} \}$
Q^{7b} : $\{ \{m\} \{sp, spi\} \leftarrow \langle \langle \text{USpectrum}, \text{spectrum_identifier} \rangle \rangle; \text{spi} > 1600; \text{spi} < 2365$ $\{ \{peph1, peph2\}, m \} \leftarrow \langle \langle \text{UPeptideHit}, \text{MassNo} \rangle \rangle; m > 1600; m = \text{spi} \}$
Q^{7c} : $\{ \{m\} \{sp, spi\} \leftarrow \langle \langle \text{USpectrum}, \text{spectrum_identifier} \rangle \rangle; \text{spi} > 1600; \text{spi} < 3740$ $\{ \{peph1, peph2\}, m \} \leftarrow \langle \langle \text{UPeptideHit}, \text{MassNo} \rangle \rangle; m > 1600; m = \text{spi} \}$
Q^{8a} : $\{ \{at\} \{ \{aa1, aa2\}, at \} \leftarrow \langle \langle \text{UAA}, at \rangle \rangle; aa2 < 50000; at > 1600; at < 1650;$ $\{ \{peph1, peph2\}, \{d1, d2\} \} \leftarrow \langle \langle \text{UPeptideHit}, \text{dbsearch} \rangle \rangle;$ $d2 > 1600; d2 < 1700; at = d2; \text{peph2} > 6200; \text{peph2} < 6251 \}$
Q^{8b} : $\{ \{at\} \{ \{aa1, aa2\}, at \} \leftarrow \langle \langle \text{UAA}, at \rangle \rangle; aa2 < 50000; at > 1600; at < 2100$ $\{ \{peph1, peph2\}, \{d1, d2\} \} \leftarrow \langle \langle \text{UPeptideHit}, \text{dbsearch} \rangle \rangle;$ $d2 > 1600; d2 < 1700; at = d2; \text{peph2} > 6200; \text{peph2} < 6376 \}$
Q^{8c} : $\{ \{at\} \{ \{aa1, aa2\}, at \} \leftarrow \langle \langle \text{UAA}, at \rangle \rangle; aa2 < 50000; at > 1600;$ $\{ \{peph1, peph2\}, \{d1, d2\} \} \leftarrow \langle \langle \text{UPeptideHit}, \text{dbsearch} \rangle \rangle;$ $d2 > 1600; d2 < 1700; at = d2; \text{peph2} > 6200; \text{peph2} < 6501 \}$
Q^{9a} : $\{ \{k\} \{k\} \leftarrow [\{m\} \{sp, spi\} \leftarrow \langle \langle \text{USpectrum}, \text{spectrum_identifier} \rangle \rangle; \text{spi} = 1645;$ $\{ \{peph1, peph2\}, m \} \leftarrow \langle \langle \text{UPeptideHit}, \text{MassNo} \rangle \rangle; m > 1600; m = \text{spi}];$ $\{k\} \leftarrow [\{at\} \{ \{aa1, aa2\}, at \} \leftarrow \langle \langle \text{UAA}, at \rangle \rangle; aa2 < 50000; at > 1600; at < 1650;$ $\{ \{peph1, peph2\}, \{d1, d2\} \} \leftarrow \langle \langle \text{UPeptideHit}, \text{dbsearch} \rangle \rangle;$ $d2 > 1600; d2 < 1700; at = d2; \text{peph2} > 6200; \text{peph2} < 6251 \}] \}$
Q^{9b} : $\{ \{k\} \{k\} \leftarrow [\{m\} \{sp, spi\} \leftarrow \langle \langle \text{USpectrum}, \text{spectrum_identifier} \rangle \rangle; \text{spi} > 1600; \text{spi} < 2365$ $\{ \{peph1, peph2\}, m \} \leftarrow \langle \langle \text{UPeptideHit}, \text{MassNo} \rangle \rangle; m > 1600; m = \text{spi}];$ $\{k\} \leftarrow [\{at\} \{ \{aa1, aa2\}, at \} \leftarrow \langle \langle \text{UAA}, at \rangle \rangle; aa2 < 50000; at > 1600; at < 2100$ $\{ \{peph1, peph2\}, \{d1, d2\} \} \leftarrow \langle \langle \text{UPeptideHit}, \text{dbsearch} \rangle \rangle;$ $d2 > 1600; d2 < 1700; at = d2; \text{peph2} > 6200; \text{peph2} < 6376 \}] \}$
Q^{9c} : $\{ \{k\} \{k\} \leftarrow [\{m\} \{sp, spi\} \leftarrow \langle \langle \text{USpectrum}, \text{spectrum_identifier} \rangle \rangle; \text{spi} > 1600; \text{spi} < 3740$ $\{ \{peph1, peph2\}, m \} \leftarrow \langle \langle \text{UPeptideHit}, \text{MassNo} \rangle \rangle; m > 1600; m = \text{spi}];$ $\{k\} \leftarrow [\{at\} \{ \{aa1, aa2\}, at \} \leftarrow \langle \langle \text{UAA}, at \rangle \rangle; aa2 < 50000; at > 1600;$ $\{ \{peph1, peph2\}, \{d1, d2\} \} \leftarrow \langle \langle \text{UPeptideHit}, \text{dbsearch} \rangle \rangle;$ $d2 > 1600; d2 < 1700; at = d2; \text{peph2} > 6200; \text{peph2} < 6501 \}] \}$

Table 4: Data Source Tuples Retrieved From Each Data Source and Tuples Produced, for Queries in Table 3

Data Source	Q^{7a}	Q^{7b}	Q^{7c}	Q^{8a}	Q^{8b}	Q^{8c}	Q^{9a}	Q^{9b}	Q^{9c}
PRIDE	4	1,374	2,750				4	1,374	2,750
PepSeeker	559	559	559				559	559	559
gpmDB				37	430	968	37	430	968
PEDRo				50	175	300	50	175	300
Tuples output	2,236	2,236	2,236	28	112	194	15,652	15,652	15,652

application of filters $\text{spi} > 1600$ and $\text{spi} < 2365$, and 559 tuples from PepSeeker, after the application of filter $m > 1600$.

The annotated queries for architectures A_1 and A_2 will contain 2 wrapper objects for queries $Q^{7a} - Q^{7c}$ and $Q^{8a} - Q^{8c}$ and 4 wrapper objects for queries $Q^{9a} - Q^{9b}$ — one for each data source involved. However, for architecture A_3 , all the annotated queries will contain just a single wrapper object (because for this set of queries there is no UNION operation to prevent this, as was the case in the first set of queries earlier).

The running times for this second set of queries using the three different architectures are shown in Figure 9. We see that while architectures A_1 and A_2 are faster for queries that join a very small amount of data, A_3 outperforms them when more data is involved, and is able to yield a result in a reasonable amount of time for queries Q^{9a} , Q^{9b} and Q^{9c} compared to architectures A_1 and A_2 . This is because (unlike AutoMed’s AQP), OGSA-DQP supports distributed join processing via its deployment of multiple GQES services on different Grid nodes; also, OGSA-DQP supports a distributed hash-join implementation (unlike AutoMed’s AQP which supports only centralised nested-loops join).

The results of this and the previous section indicate that AutoMed’s AQP is able to adequately handle all stages of query processing for SPU queries, plus also joins within data sources, but that the OGSA-DQP distributed query processing capabilities are necessary in order to efficiently evaluate queries which contain joins between data sources.

These findings validate the design of our overall architecture A_3 and show that, as intended, it does indeed combine the respective advantages of AutoMed, for fine-grained data transformation and integration, and OGSA-DQP, for Grid-based distributed query processing.

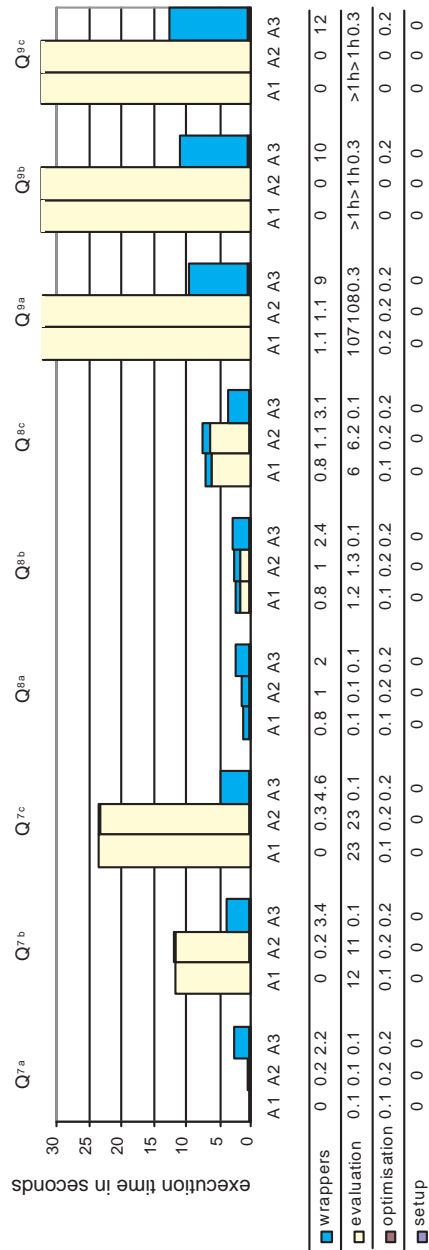


Figure 9: Evaluation of Queries Q^{7a} - Q^{9b} for Architectures A_1 (left), A_2 (middle) and A_3 (right)

5.5. Incremental Query Evaluation

AutoMed’s AQP, JDBC and OGSA-DAI all support both complete and incremental query evaluation. OGSA-DQP 3.1 (and 3.2) operates in incremental mode only and does not support complete evaluation. Thus, the query processing “pipelines” of all three architectures are able to support incremental evaluation.

In general, incremental evaluation requires less memory on both the data source and the query processor Grid nodes, since intermediate and final results do not need to be stored in memory in full. However, it does incur additional communication costs between the various query processing components.

We repeated the performance evaluations described in Sections 5.3 and 5.4 above, this time using incremental rather than complete evaluation within the AQP. The results confirmed our expectations, in that the running times for all queries and architectures are only slightly slower (up to 10% slower) than when using complete evaluation in the AQP (we recall from Section 5.2 that, in our experiments, all Grid nodes are provided with enough memory to avoid paging during query execution).

This positive finding is critical in data integration settings, where results from queries on the data sources and/or intermediate results in the AQP or OGSA-DQP can be of significant size. If the size of such results were larger than the available memory on a Grid node, then with complete evaluation the result sets would need to be paged out to disk. This I/O would incur a performance penalty significantly greater than the minor performance penalty incurred by incremental evaluation.

To confirm this, we repeated the experiments of Sections 5.3 and 5.4 by successively reducing the RAM made available to the OGSA-DAI and OGSA-DQP Evaluator services (we did not reduce the amount of RAM made available to the DBMSs as that would require a performance investigation of the particular DBMS used in the experiments). Architectures A_2 and A_3 showed significant slow-down in query execution speeds (up to an order of magnitude slower when using 10% of the original memory in some cases), that far outweighed the slight increase caused by incremental evaluation. We note though that the relative performance of the two architectures, discussed in Sections 5.3 and 5.4, was unchanged.

Our overall conclusion therefore is that, since the extra cost incurred is small, incremental query processing should be enabled for all queries submitted to all three of our architectural variants.

5.6. Parallel Evaluation

Up to now, we have assumed that the AQP evaluates queries serially. This section investigates the performance of our three architectures if the parallel version of the `QueryEvaluator` component of the AQP is used — this version currently supports parallel complete, but not parallel incremental, evaluation. This `ParallelQueryEvaluator` [20] evaluates in parallel the operands of operators with at least two collection-valued arguments (e.g. `++`). In a multicore/multiprocessor environment, this results in intra-operator parallelism and, if the operands are submitted different data sources for evaluation, this also results in distributed parallelism. In this experiment, we have used a single processor, single core CPU and therefore the benefits of parallelisation are due to the latter type of parallelism only.

We first investigated the performance using the two previous sets of queries, but the results were inconclusive. For most queries, the timings were similar to those obtained with serial evaluation, if somewhat slower, whereas for a few queries parallel evaluation showed a marginal speed-up. Therefore, we devised a set of queries that are amenable to parallel evaluation, in order to investigate the potential benefit of parallel evaluation within the AQP.

The first of these new queries is:

```
[{lsid, id}|{lsid, id} ← ⟨UPeptideHit⟩; id < 5000]
```

and the corresponding query after reformulation and optimisation is:

```
[{$d1, 'URN:LSID:ispider.man.ac.uk:pepseeker'}|$d1 ← pepseeker : ⟨peptidehit⟩; $d1 < 5000]  
+ + [{$d2, 'URN:LSID:ispider.man.ac.uk:pedro'}|{$d2, $e} ← pedro : ⟨peptidehit⟩; $d2 < 5000]  
+ + [{$d3, 'URN:LSID:ispider.man.ac.uk:gpmdb'}|$d3 ← gpmdb : ⟨peptide⟩; $d3 < 5000]  
+ + [{$d4, 'URN:LSID:ispider.man.ac.uk:pride'}|$d4 ← pride : ⟨prideppeptide⟩; $d4 < 5000]
```

Looking at this optimised query, we see that the annotated query will contain 4 wrappers for all three architectures. The other five queries in this new query set contain a different constant within the filter in the comprehension, using a step of 5,000 from 5,000 up to 30,000, which has the effect of selecting an increasing number of tuples from the four data sources.

Figure 10 shows the running times for these six new queries. The results for A_1 are the same for serial and parallel evaluation. This can be explained by the fact that A_1 performs well for SPU queries and, for this set of queries, any benefit from parallelising calls to JDBC is roughly offset by the increased costs of thread management and thread

related issues, such as lock acquisition for shared resources. Parallel evaluation results in a small benefit for A_2 and a significant speedup for A_3 . The impressive benefit for A_3 is a result of parallelising interactions with the wrappers. As discussed in Section 5.3 and illustrated in Figure 8, A_3 is quite slow compared to A_1 for SPU queries, and therefore there is a clear benefit in parallelising calls to OGSA-DQP. Similarly, A_2 uses OGSA-DAI, which has a small but noticeable effect on query evaluation.

Given these results, we can conclude that a parallel implementation of the `QueryEvaluator` component can safely replace the serial one for all classes of queries. In some cases there will be no benefit, but we can identify two cases in which there would be: firstly, if data retrieval in two or more subqueries of the overall query is costly, e.g. because of the complexity of the subqueries or because of slow network links at the data sources, in which case parallelisation of these subqueries will reduce the overall query execution time; secondly, if the post-processing by the AQP is amenable to parallelisation e.g. a query of the form Q_1++Q_2 , where Q_1 and Q_2 are subqueries sent to AutoMed wrappers for evaluation, will benefit significantly by parallel processing of the $++$ operator.

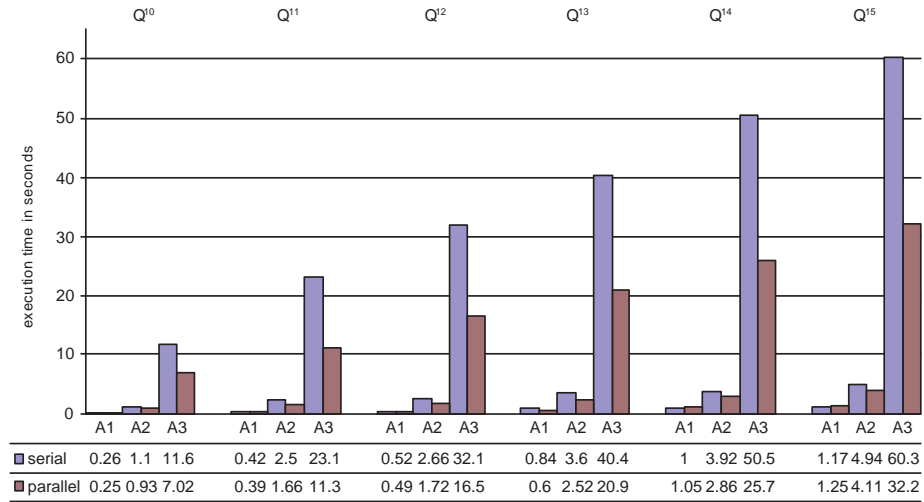


Figure 10: Serial vs. Parallel Evaluation of Queries Q^{10} - Q^{15} for Architectures A_1 , A_2 and A_3 .

5.7. Network Speed

In all our experiments discussed up to now, the network links between all pairs of Grid nodes were 100Mbps. Given that such link speeds are typical only in local data integration scenarios, and that link speeds can have a dramatic effect on query processing performance, we also connected the Grid nodes to a Layer-3 switch and repeated the user query experiments using different link speeds.

Our initial aim was to use three different link speeds, 1Mbps, 10Mbps and 100Mbps, between pairs of nodes; however the switch only supported two modes of operation, 10Mbps and 100Mbps. Another solution was to use the 100Mbps mode with 1%, 10% and 100% rate limiting¹⁴ to achieve the desired network speeds. However, the switch only supported rate limiting between 10% and 100% — not lower. Thus, we have used the following four different link speed options for the experiments discussed in this section:

L_1 : 1Mbps, by selecting the 10Mbps mode and setting rate limiting to 10%

L_2 : 10Mbps, by selecting the 100Mbps mode and setting rate limiting to 10%

L_3 : 10Mbps, by selecting the 10Mbps mode and setting rate limiting to 100%

L_4 : 100Mbps, by selecting the 100Mbps mode and setting rate limiting to 100%

We decided to keep both the L_2 and L_3 methods of obtaining the 10Mbps link speed in order to determine whether there is a difference between them.

We first evaluated the performance of the user-provided queries, Q^1 - Q^6 for each of the three architectures, A_1 , A_2 and A_3 , and for each of the four link speed options, L_1 , L_2 , L_3 and L_4 . Figure 11 shows the timing results. We see that the performance difference varies from no difference (for query Q^6 for architecture A_1 between link speeds L_4 and L_3 or L_2) to 3.7 times slower (queries Q^2 and Q^3 for architecture A_2 between link speeds L_4 and L_1 ¹⁵), and so drawing overall conclusions is not straightforward.

¹⁴Rate limiting is a method of controlling traffic at a certain switch port. Traffic exceeding a predefined limit is either delayed or is dropped and has to be retransmitted.

¹⁵Note that this slowdown is a result of these queries employing a membership sub-query, rather than a self-join, due to the inability of OGSA-DQP 3.1 to handle self-joins — see Section 5.3. Containment is ultimately evaluated by the AQP, resulting in more messages between AQP and OGSA-DAI than if a self-join were used, in which case the join would be evaluated internally by the data sources themselves.

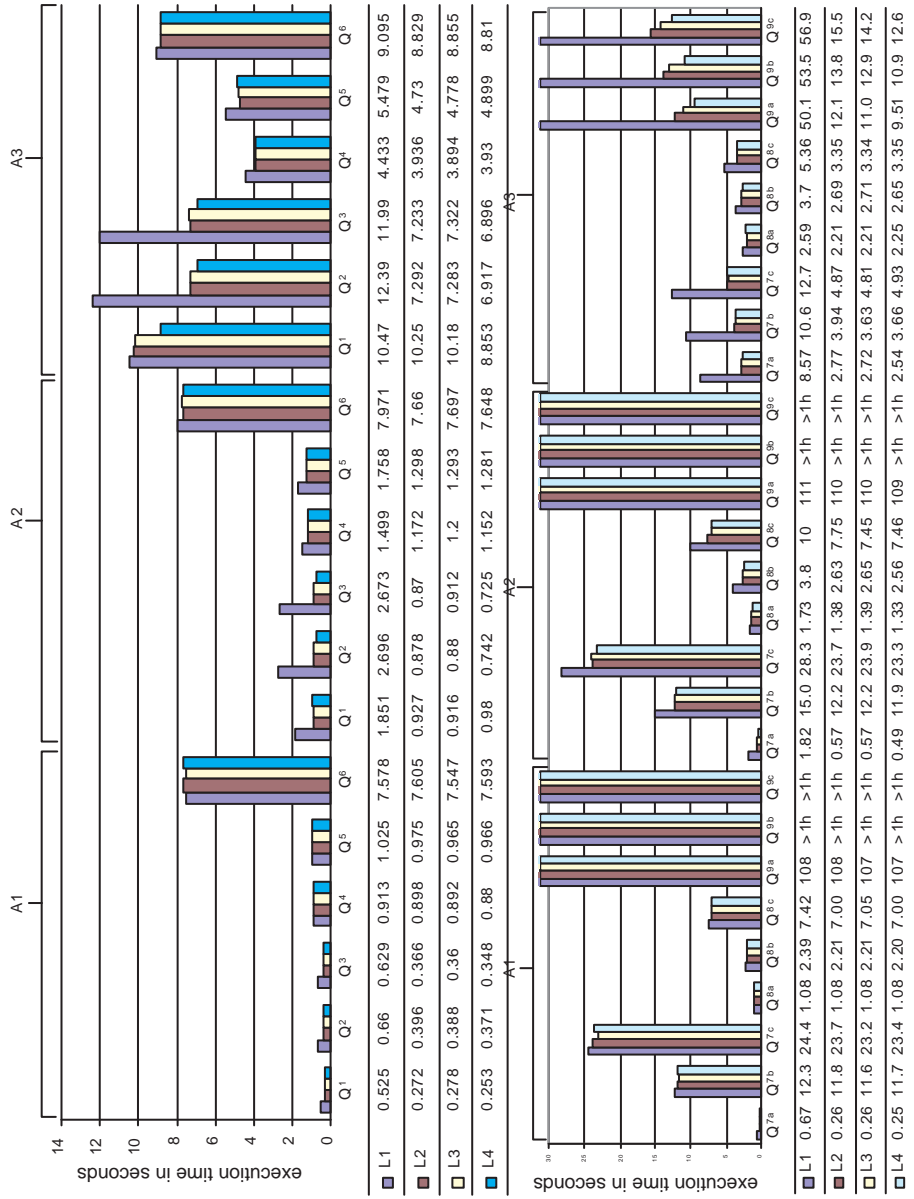


Figure 11: Left: Complete Evaluation of Queries Q^1 - Q^6 for Architectures A_1 , A_2 and A_3 and Link Speeds L_1 - L_4 . Right: Complete Evaluation of Queries Q^{7a} - Q^{9c} for Architectures A_1 , A_2 and A_3 and Link Speeds L_1 - L_4 .

We next evaluated the performance of the second set of queries Q^{7a} - Q^{9c} , for each of the three architectures, A_1 , A_2 and A_3 , for each of the four link speed options, L_1 , L_2 , L_3 and L_4 . The results are again shown in Figure 11. We see that architecture A_1 shows no difference of performance for most queries, and a small but noticeable difference for query Q^{7c} between link speed L_1 and the other link speeds. Taking a closer look at the time spent in each query processing component (set-up time, optimisation, evaluation and wrappers), we see that the times for the first three components are the same, but there is a small but noticeable difference in the time spent in the wrappers, clearly resulting from the time spent to transmit results from the data sources to the AQP. The same applies for the running times for architecture A_2 , but this time the effect is noticeable for all queries between link speed L_1 and the other link speeds. This can be explained by the extra OGSA-DAI layer and the extra messaging it incurs which, even though it goes unnoticed for higher link speeds between Grid nodes, it is evident for the low speed of 1Mbps. Finally, in architecture A_3 , where the use of OGSA-DQP incurs yet another layer of (verbose XML) messaging, the difference in performance is even worse for L_1 , to the point that L_4 is more than 5 times faster than L_1 . The link speed is so significant in this architecture, that there is also a clear difference in performance between the other link speeds as well for queries Q^{9a} , Q^{9b} and Q^{9c} .

These results highlight the need for any deployed architecture to be able to determine the speed characteristics of the connections between Grid nodes and to support cost optimisers that provide alternative query plans based on these characteristics. A closer examination of this aspect of query optimisation is outside the scope of this paper and a subject for future work. As a first measure, however, the verbosity of the messaging of OGSA-DAI and OGSA-DQP services needs to be addressed, and would significantly increase performance for settings with low network speed.

Similarly, other issues for future work include an analysis of the effect of network speed on parallel, incremental, and combined parallel and incremental query processing.

5.8. Summary of Findings

Previous work on querying heterogeneous distributed Grid-enabled data sources has not supported fine-grained data transformation and integration of the source data, but these capabilities are needed in order to support distributed querying and

analysis of complex data sources, as arising in the Life Sciences for example. Since there is currently no Grid-enabled middleware supporting fine-grained data transformation/integration, we have investigated combining the capabilities of existing data integration and Grid data access and querying middleware in order to meet the ISPIDER project’s data integration requirements. While our work has been motivated by ISPIDER, the architectural framework we have developed is generic and can be applied to the integration of other sets of Life Sciences data sources, and indeed to other application areas.

Grid data access is supported in our architecture using OGSA-DAI. Our evaluation confirms previous findings of [29, 33] that report an additional cost in terms of performance when compared to direct data access using JDBC. However, our evaluation indicates that this cost is small in common query processing tasks, and is in many cases negligible compared to the overall cost of query processing in our setting of Grid data integration.

Distributed query processing over Grid data sources is performed using OGSA-DQP in our architecture. Our evaluation shows that OGSA-DQP has acceptable query performance for queries arising in a heterogeneous data integration setting i.e. queries arising from the reformulation of queries expressed on the global schema via fine-grained data transformation/integration mappings. Particular performance issues identified have been communicated to the OGSA-DQP development team, including performance issues in non-distributed query processing (also reported by the OGSA-DQP team in [37]) and the verbosity of the data exchange format, which severely affects settings with low-bandwidth network links.

Fine-grained data integration is achieved in our architecture using the AutoMed heterogeneous data integration system. AutoMed has shown its ability to perform fine-grained data integration for real-world application settings here, as well as in other settings [57].

In any data integration system, the queries resulting from reformulation can be quite complex due to the integration logic encoded in the mappings, and the reformulated queries need to be optimised. AutoMed provides an extensible query processing and optimisation framework that can accommodate well-established optimisation techniques, as discussed in Section 3.2.

The findings of Sections 5.3 and 5.4 validate the design of our architecture and

show that, as intended, it does indeed combine the respective advantages of AutoMed, for fine-grained data transformation and integration, and OGSA-DQP, for Grid-based distributed query processing.

We have also seen that enabling parallel and incremental evaluation within the query post-processing undertaken by the AutoMed query processor can offer considerable advantages in terms of overall query performance and memory consumption without incurring significant overheads.

Turning now to more general conclusions that may be drawn from these findings, the architecture of Figure 3 makes very few assumptions about its components and these perform functionality that, in principle, could be provided by a variety of other systems/products.

For example, access to Grid data sources can be provided by any service-based product that supports the types of data sources used in a particular application setting in terms of retrieving schema information and query processing. While it is the case that OGSA-DAI and OGSA-DQP are tightly coupled, this is not a requirement of our architecture, and the development of bridging functionality between other Grid distributed query processing and Grid data access middleware would be straightforward. Although AutoMed offers advantages in terms of its extensibility with new modelling languages (via its HDM) and its support for schema evolution [16], data integration functionality within our architecture could be provided by any data integration system that supports:

- (i) an extensible wrapper architecture that would allow development of the necessary wrappers for interacting with the Grid distributed query processing and Grid data access components,
- (ii) fine-grained data transformation and integration via GAV rules,
- (iii) a query optimisation framework supporting the optimisation capabilities necessary to complement those of the Grid distributed query processor component, and
- (iv) query processing functionality that is able to post-process subquery results for those global queries that the Grid distributed query processor component does not support.

The key general finding from our experiments is that it is indeed feasible to perform fine-grained data transformation/integration in the context of heterogeneous Grid data

sources and achieve acceptable query performance, by using our proposed architectural framework.

With respect to point (iii) above, in the context of SPJ global queries and SPJU data transformation queries, we have found that applying just four well-known equivalences within the data integration system (Optimisers 1-4 of Section 3.2) is sufficient to provide acceptable query performance in conjunction with OGSA-DQP’s query optimisation and query planning capabilities (we discussed these in Section 2 and refer the reader to [50] for further details). In the longer term, supporting cost-based optimisation could be expected to have a further beneficial effect on query performance (at present OGSA-DAI and OGSA-DQP do not expose the necessary metadata, and neither would AutoMed be able to exploit it — this is clearly an important area of future work).

With respect to point (iv) above, supporting parallel and incremental evaluation within the data integration system can be expected to offer significant advantages in terms of overall query performance.

Having summarised our findings and the more general conclusions that may be drawn from them, we conclude by outlining a general methodology for building a system that offers the same functionality as our architecture:

First, three different types of functionality are required: Grid data access (Grid-DA), distributed query processing over Grid data sources (Grid-DQP), and fine-grained data integration of heterogeneous data sources (fine-grained DI).

Second, if different components are used to provide the fine-grained DI and Grid-DQP functionalities, care must be taken if the data transformation language or the query language (or both) of the fine-grained DI component is more expressive than the query language of the Grid-DQP component. In such a case, the DI component needs to provide a query optimiser that rewrites reformulated queries into subqueries which are suitable for evaluation by the Grid-DQP component, and a query processor that is able to post-process subquery results returned by Grid-DQP component.

Third, query language translation functionality may be necessary between the fine-grained DI and the Grid-DQP component if the two use different query languages. Similarly, query language translation functionality may be required between the Grid-DQP and the Grid-DA component.

Fourth, the combination of Optimisers 1-4 in Section 3.2 and OGSA-DQP’s query

optimisation and planning capabilities is required in order to give acceptable query performance in the context of SPJ global queries and SPJU data transformation queries. In our architecture, these query optimisation and planning capabilities are provided by the AutoMed and OGSA-DQP systems as we have described earlier, but a single component that provides all of them, or a pair of components that provide them in a different combination would also be possible.

Fifth, the architecture must provide query planning capabilities that include the parallel and distributed evaluation of the operands of collection and join operators (as per the discussion in Section 2 on the query planning capabilities of OGSA-DQP and our discussion in Section 5.6).

6. Conclusions and Future Work

In earlier work, we described an architecture for the virtual integration of heterogeneous Grid resources that provides fine-grained data transformation and integration capabilities coupled with distributed query processing over heterogeneous data sources. To achieve this, we combined existing technologies in data integration, namely the AutoMed system, and in Grid access and query processing, namely OGSA-DAI and OGSA-DQP. There is currently no Grid-enabled middleware supporting fine-grained data transformation/integration, and hence our architecture is novel in combining this with distributed query processing within a Grid environment.

In this paper, we have presented an extensive evaluation of query performance in our architecture, investigating a number of performance factors and comparing it with two other architectures that do not use OGSA-DQP and OGSA-DAI/OGSA-DQP, respectively, in order to ascertain the impact of each component on query processing performance. To our knowledge, this is the first such investigation in a Grid-based environment combining fine-grained data transformation/integration with distributed query processing.

Our investigation has demonstrated that our full architecture is able to efficiently evaluate select-project-join-union queries over integrated heterogeneous Grid data sources in a real-world biological setting. Combining data transformation/integration capabilities, as exemplified by AutoMed, with data access and distributed query processing middleware, as exemplified by OGSA-DAI and OGSA-DQP, has thus been demonstrated to be feasible from a query performance perspective.

Because our architecture makes very few assumptions about its components and these components perform functionality that could be provided by different systems and products, we believe that our findings will be of benefit to others wishing to combine similar capabilities in supporting Grid applications that require sophisticated data integration and querying over distributed heterogeneous data resources.

For the future, we have noted the need for OGSA-DQP to support both nested queries and the `UNION` operator, and also OGSA-DQP's relative inefficiency regarding SPJ queries, and these findings have been communicated to the OGSA-DQP team. Further improvements in query performance within our architecture would be likely to arise if OGSA-DAI were able to extract additional metadata from the data sources (if available) so as to allow more effective cost-based optimisation (e.g. information about dataset sizes, distributions and access paths), and again we have communicated this to the OGSA-DAI team.

Our own future work includes: investigating the query performance impact in our architecture of more complex transformation/integration logic than SPJU queries and more complex global queries than SPJ queries; providing JDBC and OGSA-DAI interfaces over the AutoMed Query Processor, in order to promote interoperability between AutoMed and other Grid middleware e.g. Taverna [27]; and improving query optimisation performance in AutoMed and extending it with cost-based query optimisation capabilities. The ISPIDER integrated resource itself will be made publicly available by deploying AutoMed as a service within the ISPIDER Central website [49].

Acknowledgements

We would like to thank our ISPIDER project collaborators from the University of Manchester for our joint work reported in [54], and for providing the initial set of queries for the performance study reported here. We would also like to thank Dimitris Fourkiotis and Jamie Walters for providing the initial implementations of AutoMed's parallel evaluator and SQL-to-IQL translator, respectively, and Steven Lynden, Arijit Mukherjee and Ally Hume for their help with OGSA-DAI and OGSA-DQP.

References

- [1] M. Alpdemir, A. Gounaris, A. Mukherjee, D. Fitzgerald, N. Paton, P. Watson, R. Sakellariou, A. Fernandes, and J. Smith. Experience on performance evaluation

- with OGSA-DQP. In *Proc. U.K. e-Science All Hands Meeting (AHM'05)*, 2005.
- [2] M. Alpdemir, A. Mukherjee, N. Paton, P. Watson, A. Fernandes, A. Gounaris, and J. Smith. Service-based distributed querying on the Grid. In *Proc. International Conference on Service Oriented Computing (ICSOC'03)*, pages 467–482, 2003.
- [3] M. Antonioletti, M. Atkinson, R. Baxter, A. Borley, N. Chue Hong, B. Collins, N. Hardman, A. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead. The design and implementation of Grid database services in OGSA-DAI. *Concurrency - Practice and Experience*, 17(2–4):357–376, 2005.
- [4] A. Budara, P. Cudré-Mauroux, and K. Aberer. From bioinformatics web portals to semantically integrated Data Grid networks. *Future Generation Computer Systems*, 23(3):485–496, 2007.
- [5] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
- [6] R. Cattell and D. Barry. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.
- [7] T. Clark, S. Martin, and T. Liefeld. Globally distributed object identification for biological knowledgebases. *Briefings in Bioinformatics*, 5(1):59–70, 2004.
- [8] C. Comito, A. Gounaris, R. Sakellariou, and D. Talia. A service-oriented system for distributed data querying and integration on Grids. *Future Generation Computer Systems*, 25(5):511–524, 2009.
- [9] C. Comito and D. Talia. XML data integration in OGSA Grids. In *Proc. Workshop on Data Management in Grids (DMG at VLDB'05)*, pages 4–15, 2005.
- [10] R. Craig, J. Cortens, and R. Beavis. Open source system for analyzing, validating, and storing protein identification data. *Journal of Proteome Research*, 3(6), 2004.
- [11] S. Davidson, J. Crabtree, B. Brunk, J. Schug, V. Tannen, G. Overton, and C. Stoeckert Jr. K2/Kleisli and GUS: Experiments in integrated access to genomic data sources. *IBM Systems Journal*, 40(2):512–531, 2001.

- [12] S. Davidson, C. Overton, V. Tannen, and L. Wong. BioKleisli: A digital library for biomedical researchers. *International Journal on Digital Libraries*, 1(1):36–53, 1997.
- [13] A. Deshpande and J. Hellerstein. Decoupled query optimization for federated database systems. In *Proc. International Conference on Data Engineering (ICDE'02)*, pages 716–727, 2002.
- [14] B. Dobrzelecki, M. Antonioletti, J. Schopf, A. Hume, M. Atkinson, N. Chue Hong, M. Jackson, K. Karasavvas, A. Krause, M. Parsons, T. Sugden, and E. Theocharopoulos. Profiling OGSA-DAI performance for common use patterns. In *Proc. U.K. e-Science All Hands Meeting (AHM'06)*, 2006.
- [15] O. Duschka and M. Genesereth. Answering recursive queries using views. In *Proc. ACM Symposium on Principles of Database Systems (PODS'97)*, pages 109–116, 1997.
- [16] H. Fan and A. Poulouvasilis. Schema evolution in data warehousing environments — a schema transformation-based approach. In *Proc. International Conference on Conceptual Modeling (ER'04)*, pages 639–653, 2004.
- [17] L. Fegaras and D. Maier. Towards an effective calculus for object query languages. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 47–58, 1995.
- [18] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, 2000.
- [19] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The Open Grid Services Architecture, Version 1.5. Technical Report GFD-I.080, Open Grid Forum, September 2006. Available at <http://www.ogf.org/documents/GFD.80.pdf>.
- [20] D. Fourkiotis. Implementation of parallel and distributed query processing in the AutoMed heterogeneous data integration toolkit. Master's thesis, Birkbeck College, University of London, 2007. Available at <http://www.doc.ic.ac.uk/automed/publications/Fou07.pdf>.

- [21] K. Garwood, T. McLaughlin, C. Garwood, S. Joens, N. Morrison, C. Taylor, K. Carroll, C. Evans, A. Whetton, S. Hart, D. Stead, Z. Yin, A. Brown, A. Hesketh, K. Chater, L. Hansson, M. Mewissen, P. Ghazal, J. Howard, K. Lilley, S. Gaskell, A. Brass, S. Hubbard, S. Oliver, and N. Paton. Pedro: A database for storing, searching and disseminating experimental proteomics data. *BMC Genomics*, 5(1), 2004.
- [22] C. Goble, R. Stevens, G. Ng, S. Bechhofer, N. Paton, P. Baker, M. Peim, and A. Brass. Transparent access to multiple bioinformatics information sources. *IBM Systems Journal*, 40(2):532–551, 2001.
- [23] A. Gounaris, C. Comito, R. Sakellariou, and D. Talia. A service-oriented system to support data integration on data grids. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 627–635, 2007.
- [24] A. Gounaris, C. Yfoulis, R. Sakellariou, and M. Dikaiakos. Robust runtime optimization of data transfer in queries over web services. In *Proc. International Conference on Data Engineering (ICDE'08)*, pages 596–605, 2008.
- [25] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD'90)*, pages 102–111, 1990.
- [26] L. Haas, P. Schwarz, P. Kodali, E. Kotlar, J. Rice, and W. Swope. Discoverylink: A system for integrated access to life sciences data sources. *IBM Systems Journal*, 40(2):489–511, 2001.
- [27] D. Hull, K. Wolstencroft, R. Stevens, C. A. Goble, M. Pocock, P. Li, and T. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(2):729–732, 2006.
- [28] Z. Ives, A. Halevy, and D. Weld. Adapting to source properties in processing data integration queries. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*, pages 395–406, 2004.
- [29] M. Jackson, M. Antonioletti, N. C. Hong, A. Hume, A. Krause, T. Sugden, and M. Westhead. Performance analysis of the ogsa-dai software. In *Proc. UK e-Science All Hands Meeting (AHM'04)*, pages 15–20, 2004.

- [30] A. Jones, M. Miller, R. Aebersold, R. Apweiler, C. Ball, A. Brazma, J. De Greef, N. Hardy, H. Hermjakob, S. Hubbard, P. Hussey, M. Igra, H. Jenkins, R. Julian Jr., K. Laursen, S. Oliver, N. Paton, S. Sansone, U. Sarkans, C. Stoeckert Jr, C. Taylor, P. Whetzel, J. White, P. Spellman, and A. Pizarro. The Functional Genomics Experiment model (FuGE): an extensible framework for standards in functional genomics. *Nature Biotechnology*, 25(10):1127–1133, 2007.
- [31] P. Jones, R. Côté, L. Martens, A. Quinn, C. Taylor, W. Derache, H. Hermjakob, and R. Apweiler. PRIDE: a public repository of protein and peptide identifications for the proteomics community. *Nucleic Acids Research*, 1(34):659–663, 2006.
- [32] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Survey*, 32(4):422–469, 2000.
- [33] S. Kottha, K. Abhinav, R. Müller-Pfefferkorn, and H. Mix. Accessing bio-databases with OGSA-DAI - a performance analysis. In *Proc. International Workshop on Distributed, High-Performance and Grid Computing in Computational Biology (GCCB'06)*, pages 141–156, 2006.
- [34] A. Langegger, W. Wöß, and M. Blöchl. A Semantic Web middleware for virtual data integration on the Web. In *Proc. European Semantic Web Conference (ESWC'08)*, pages 493–507, 2008.
- [35] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. ACM Symposium on Principles of Database Systems (PODS02)*, pages 233–246, 2002.
- [36] A. Levy, A. Rajamaran, and J. Ordille. Querying heterogeneous information sources using source description. In *Proc. International Conference on Very Large Data Bases (VLDB'96)*, pages 252–262, 1996.
- [37] S. Lynden, A. Mukherjee, A. Hume, A. Fernandes, N. Paton, R. Sakellariou, and P. Watson. The design and implementation of OGSA-DQP: A service-based distributed query processor. *Future Generation Computer Systems*, 25(3):224–236, 2009.
- [38] P. McBrien and A. Poulouvasilis. A uniform approach to inter-model transformations. In *Proc. International Conference on Advanced Information Systems Engineering (CAiSE'99)*, pages 333–348, 1999.

- [39] P. McBrien and A. Poulouvasilis. Data integration by bi-directional schema transformation rules. In *Proc. International Conference on Data Engineering (ICDE'03)*, pages 227–238, 2003.
- [40] P. McBrien and A. Poulouvasilis. Defining Peer-to-Peer Data Integration using Both as View Rules. In *Proc. Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'03 at VLDB'03)*, pages 91–107, 2003.
- [41] P. McBrien and A. Poulouvasilis. P2P query reformulation over Both-as-View data transformation rules. In *Proc. Workshop on Databases, Information Systems and Peer-to-Peer Computing (at VLDB'06)*, pages 310–322, 2006.
- [42] T. McLaughlin, J. Siepen, J. Selley, J. Lynch, K. Lau, H. Yin, S. Gaskell, and S. Hubbard. Pepseeker: a database of proteome peptide identifications for investigating fragmentation patterns. *Nucleic Acids Research*, 34(1), 2006.
- [43] A. Poulouvasilis and C. Small. Algebraic query optimisation for database programming languages. *VLDB Journal*, 5(2):119–132, 1996.
- [44] A. Poulouvasilis and L. Zamboulis. A tutorial on the IQL query language. AutoMed Technical Report 28, March 2007.
- [45] C. Quix. Quality-oriented and metadata-driven integration in information grids. In *Proc. IST Workshop on Metadata Management in Grid and P2P Systems - Models, Services, Architectures (MMGPS'04)*, pages 493–507, 2004.
- [46] N. Rizopoulos. Automatic discovery of semantic relationships between schema elements. In *Proc. International Conference on Enterprise Information Systems (ICEIS'04)*, pages 3–8, 2004.
- [47] J. Saltz, S. Oster, S. Hastings, S. Langella, T. Kurç, W. Sanchez, M. Kher, A. Manisundaram, K. Shanbhag, and P. Covitz.
- [48] B. Seshasayee, K. Schwan, and P. Widener. SOAP-binQ: High-performance SOAP with continuous quality management. In *Proc. International Conference on Distributed Computing Systems (ICDCS'04)*, pages 158–165, 2004.

- [49] J. Siepen, K. Belhajjame, J. Selley, S. Embury, N. Paton, C. Goble, S. Oliver, R. Stevens, L. Zamboulis, N. Martin, A. Poulouvassillis, P. Jones, R. Cote, H. Hermjacob, M. Pentony, D. Jones, C. Orengo, and S. Hubbard. ISPIDER Central: an integrated database web-server for proteomics. *Nucleic Acids Research*, 36(2):485–490, 2008.
- [50] J. Smith, A. Gounaris, P. Watson, N. Paton, A. Fernandes, and R. Sakellariou. Distributed query processing on the Grid. In *Proc. International Workshop on Grid Computing (GRID'02)*, pages 279–290, 2002.
- [51] V. Tannen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proc. International Workshop on Database Programming Languages (DBPL'91)*, pages 9–19, 1991.
- [52] The Gene Ontology Consortium. Gene Ontology: tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.
- [53] G. Trevisol, C. Biancardi, Á. Barbosa, J. Pereira Filho, R. Costa, and E. Cardoso. A distributed query execution engine in a grid environment. In *Proc. IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 418–425, 2007.
- [54] L. Zamboulis, H. Fan, K. Belhajjame, J. Siepen, A. Jones, N. Martin, A. Poulouvassillis, S. Hubbard, S. Embury, and N. Paton. Data access and integration in the ISPIDER proteomics Grid. In *Proc. International Workshop on Data Integration in the Life Sciences (DILS'06)*, pages 3–18, 2006.
- [55] L. Zamboulis, S. Mittal, E. Jasper, H. Fan, and A. Poulouvassillis. Processing IQL queries in the AutoMed toolkit v1.2. AutoMed Technical Report 35, July 2008.
- [56] L. Zamboulis and A. Poulouvassillis. Information sharing for the Semantic Web - a schema transformation approach. In *Proc. International Workshop Data Integration and the Semantic Web (DISWeb at CAiSE'06)*, pages 275–289, 2006.
- [57] L. Zamboulis, A. Poulouvassillis, and G. Roussos. Flexible data integration and ontology-based data access to medical records. In *Proc. IEEE International Conference on Bioinformatics and BioEngineering (BIBE'08)*, page TBC, 2008.

- [58] E. Zdobnov, R. Lopez, R. Apweiler, and T. Etzold. The EBI SRS server — recent developments. *Bioinformatics*, 18(2):368–373, 2002.



Lucas Zamboulis was until recently a researcher at the Department of Computer Science and Information Systems at Birkbeck and the Department of Biochemistry and Molecular Biology at University College London, and he is now at Contextured Ltd. He received his BSc in Informatics from the Aristotle University of Thessaloniki, Greece, and his PhD in Computer Science from Birkbeck. His research interests are in data management, including XML and heterogeneous data transformation and integration, distributed query processing and cloud computing.



Nigel Martin is a Senior Lecturer in the Department of Computer Science and Information Systems, Birkbeck College, University of London. He received his BSc in Mathematics and MSc in Computer Science from University College London and PhD in Computer Science from Birkbeck College. His research interests centre on data management, integration and mining. Current interests include data integration support for data mining, and the management, analysis and mining of genomic and wider biological data.



Alexandra Poulouvasilis is Professor in the Department of Computer Science and Information Systems at Birkbeck. She received an MA in Mathematics from Cambridge University, and an MSc and PhD in Computer Science from Birkbeck. Her research interests centre on information management, integration and personalisation. Since 2003 she has been Co-Director of the London Knowledge Lab, a multi-disciplinary research institution which aims to explore the ways in which digital technologies and new media are shaping the future of knowledge and learning.