Query Processing and Optimisation in Integrated Heterogeneous Grid Resources*

Lucas Zamboulis, Nigel Martin, Alexandra Poulovassilis
School of Computer Science and Information Systems,
Birkbeck, University of London

Abstract

The performance of Grid computing technologies for distributed data access and query processing has been investigated in a number of studies. However, different Grid data sources may have schema conflicts which require fine-grained resolution through the use of data integration technologies that are not supported by the current generation of Grid data access and querying middleware. This is particularly the case with distributed querying and analysis of complex data sources as found in Life Sciences applications. The query performance of architectures that combine Grid data access and query processing capabilities with data integration technologies has not been investigated to date.

In this paper, we investigate architectural, optimisation and performance issues arising from the coupling of Grid data access and distributed querying together with data integration technologies. Specifically, 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. The significance of our results for further development of query processing technology over heterogeneous Grid data sources is discussed.

Our performance analysis had been carried out using a representative integrated data resource developed as part of the ISPIDER (*In Silico Proteome Integrated Data Environment Resource*) project. This project has developed a platform of proteome-related resources using existing Grid middleware, and leveraging standards from the application areas of proteomics and bioinformatics.

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 is often the case in the analysis of Life Sciences data.

*Research funded by the ISPIDER project, BBSRC Grant No. BBSB17220.

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 [1] 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 [22] supports the standardised modelling and exchange of data related to experiments in functional genomics, while the Gene Ontology [39] 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 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 only option if access to the latest versions of the data sources is required. In this paper, we address the virtual integration of heterogeneous Grid-enabled data sources, with particular focus on biological data sources.

Many systems have been developed which create and maintain virtual integrated resources in the Life Sciences domain: examples of significant systems are DiscoveryLink [19], K2/Kleisli [8], Tambis [18], SRS [44]. The aim of these early systems was to provide users with the ability to formulate queries on the integrated resource which would be very complex or very costly if performed directly on the individual data sources, sometimes prohibitively so. 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. More recent work [40; 34; 36; 25; 6] 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. 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 in the virtual global schema.

In our own recent work [41], 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*) project[1], which has developed a platform of proteomics-related resources using existing Grid middleware, and leveraging standards from the application areas of proteomics and bioinformatics. 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 transformation/integration functionality.

In this paper we investigate the architectural, optimisation and performance issues arising from the coupling of Grid distributed querying and data integration technologies. Specifically, we investigate these issues for the OGSA-DAI and OGSA-DQP Grid data access and querying middleware, combined with the AutoMed data transformation/integration system. We present an architectural framework that we have developed for investigating the combination of these technologies and the results of the performance study undertaken. We also discuss the significance of these results for the further development of query processing capabilities over integrated heterogeneous Grid resources.

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 [41], 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 a representative Grid 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 integrated ISPIDER 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)** [15] defines a set of capabilities that address the key concerns of Grid computing systems, and provides a

[1]`http://www.ispider.man.ac.uk`

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 resources 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).

OGSA-DQP (OGSA - Distributed Query Processor)[3] is a service-based distributed query processor over OGSA-DAI enabled resources [1]. 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 **Query Evaluation Service (GQES)**, or **Evaluator**. The Coordinator uses resource metadata and computational resource information to compile, optimise and partition an input query into a distributed query execution plan, and each one of the partitions of this plan is scheduled for execution on one of the Grid's nodes [38; 14]. The Evaluator implements a physical algebra over OGSA-DAI data services and is able to evaluate 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.

The motivation for OGSA-DAI and OGSA-DQP is to develop middleware that interfaces between existing 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/DQP 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.

Performance studies of OGSA-DAI and OGSA-DQP have focussed on the impact of alternative data transfer formats and query processing models. [10] 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 [10], 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.

[24] 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 instant and incremental evaluation of SQL queries. In particular, OGSA-DAI instant evaluation results in a slightly higher data

[2]`http://www.ogsadai.org.uk`

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

throughput compared to incremental evaluation, but JDBC outperforms both.

[2] presents a performance evaluation of OGSA-DQP Version 2, but without investigating 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 is the comparison between instant and incremental query processing, with the latter being outperformed by the former. In our performance evaluation, we use the later OGSA-DQP Version 3.1, due to the lack of support of Version 2 for a number of ISPIDER requirements, such as some relational data types and newer versions of Linux OSs supporting newer hardware. OGSA-DQP Version 3 recommends the use of incremental evaluation, and indeed does not support instant evaluation.

3 AutoMed

AutoMed[4] is a system supporting the transformation and integration of heterogeneous data, offering the capability 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 [28]. For any modelling language, $\mathcal{M}$, specified in this way (via the API of AutoMed's Model Definitions Repository), 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 add to/delete 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.

Instances of modelling constructs within a particular schema are identified by means of their *scheme* enclosed within double chevrons $\langle\langle \ldots \rangle\rangle$ 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 AutoMed's Schemas & Transformations Repository (STR).

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 comprehensions-based functional query language, **IQL**[5]. 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 schema. More 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` and the upper bound may be `Any`, which respectively indicate no known information about the lower or upper bound of the extent of the new construct.

[4]`http://www.doc.ic.ac.uk/automed`

[5]Comprehension-based languages subsume query languages such as SQL-92 and OQL in their expressiveness [4]. The purpose of IQL within the AutoMed system is to provide a common query language that queries written in various high-level query languages (e.g. SQL, XQuery, OQL) can be translated into and out of.

The queries supplied with primitive transformations can be used to translate queries or data along a transformation pathway by means of *query unfolding* — we refer the interested reader to [29; 30; 31] for details of this process. The queries supplied with primitive transformations also provide the necessary information for transformation pathways to be automatically *reversible*, in that each `add`/`extend` transformation is reversed by a `delete`/`contract` transformation with the same arguments, while each `rename` is reversed by a `rename` with the two arguments swapped.

As discussed in [29], this means that AutoMed is a *both-as-view (BAV)* data integration system: the `add`/`extend` steps in a transformation pathway correspond to Global-As-View (GAV) rules [26] as they incrementally define target schema constructs in terms of source schema constructs; while the `delete` and `contract` steps correspond to Local-As-View (LAV) rules [11; 27] since they define source schema constructs in terms of target schema constructs. An in-depth comparison of BAV with other data integration approaches can be found in [29; 30; 31].

3.1 The IQL Query Language

We now give a brief overview of IQL, to the level of detail required for Section 5.

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 ++ appends two lists, and performs bag union and set union on bags and sets, respectively. The operator $\mathsf{flatmap}$ applies a collection-valued function f to each element of a collection and applies ++ to the resulting collections.

The operator $\mathsf{flatmap}$ can be used to specify *comprehensions* over collections. These are of the form $[\mathsf{h} \mid \mathsf{Q_1};\ \ldots;\ \mathsf{Q_n}]$ where h is an expression termed the *head* of the comprehension, and $Q_1, \ldots, Q_n$ are *qualifiers*, with $\mathsf{n} \geq 0$. Each qualifier is either a *filter* or a *generator*. A generator has syntax $\mathsf{p} \leftarrow \mathsf{e}$ 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. Comprehensions are a convenient high-level syntax and add no extra expressiveness to languages such as IQL (because they can be translated into successive applications of $\mathsf{flatmap}$). Comprehension syntax can be used to express *projection*, *selection*, *cartesian product* and *join* operations.

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[6]:

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

[6]Here, $\langle\langle \mathsf{r},\mathsf{a}\rangle\rangle$ and $\langle\langle \mathsf{s},\mathsf{b}\rangle\rangle$ are *schemes* identifying two constructs of an AutoMed relational schema, namely the attribute a of table r and the attribute b of table s. We note that the standard AutoMed encoding of relational schemas decomposes each table $R(a_1, \ldots, a_n)$ into n binary relationships, $R(\bar{k}, a_1), \ldots, R(\bar{k}, a_n)$, where $\bar{k}$ is the subset of the $a_1, \ldots, a_n$ comprising the primary key of R. Thus, the extent of an AutoMed scheme $\langle\langle \mathsf{r},\mathsf{a}\rangle\rangle$ is the projection of table r onto its primary key attributes plus its attribute a. See [29] for details.

and is equivalent to the following query:

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

3.2 Query Optimisation

AutoMed's `QueryOptimiser` component (see Section 4.1) serves as a "policy" class, by coordinating the application of a number of individual optimisers. Below, we briefly discuss three of these optimisers as we will be referring to them again in our performance study later. In general, AutoMed users are free to create their own custom optimisation components and their own own custom optimisers. We refer the interested reader to [21] for details of query optimisation in AutoMed and to [4; 13; 33; 9] for more general discussions of query optimisation in comprehension languages.

The three optimisers presented below operate in tandem, as will become apparent in Sections 5.3 and 5.4. In particular, it is common after the reformulation stage for comprehensions to contain generators that iterate over an expression that appends a number of further comprehensions. Optimiser 2 below can split these into a number of simpler comprehensions. Optimiser 3 can then be applied to prune the number of comprehensions output. Optimiser 1 is necessary since often the output of Opt. 2 contains nested comprehensions, whereas Opt. 3 operates on unnested ones. An example of the optimisation process is given in Section 5.3.

Optimiser 1 This optimiser unnests nested 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 of the rule in the tree:

$$[\mathsf{h}|\mathsf{e}_1; \mathsf{p}_1 \leftarrow [\mathsf{p}_2|\mathsf{Q}_1; \ldots; \mathsf{Q}_n]; \mathsf{e}_2] \Rightarrow [\mathsf{h}|\mathsf{e}_1; \mathsf{Q}'_1; \ldots; \mathsf{Q}'_n; \mathsf{e}_2]$$

There is a proviso to applying this rule, in that the patterns 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.

Optimiser 2 This optimiser rewrites comprehensions containing generators that iterate over an expression appending a number of further 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:

$$[\mathsf{h}|\mathsf{e}; \mathsf{p} \leftarrow \mathsf{e}_1 ++ ... ++ \mathsf{e}_n; \mathsf{e}'] \Rightarrow [\mathsf{h}|\mathsf{e}; \mathsf{p} \leftarrow \mathsf{e}_1; \mathsf{e}'] ++ ... ++ [\mathsf{h}|\mathsf{e}; \mathsf{p} \leftarrow \mathsf{e}_n; \mathsf{e}']$$

(we note that this is the equivalent of distributing selections and projections over the union operation in the relational algebra). The benefit of this optimisation is that some of the resulting simpler comprehensions 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 comprehensions output for a given input comprehension is $\prod_{i=1}^{n} m_i$, where m_i is the number of sub-expressions of the i^{th} generator.

Optimiser 3 This optimiser eliminates comprehensions for which it can infer that they will return empty results because they are undertaking a join over non-overlapping attributes. In particular, this optimisation can applied over attributes that are known to have globally unique values over the data sources being integrated:

$$[\mathsf{h}|\mathsf{e};\mathsf{p}_1 \leftarrow \mathsf{e}_1;\mathsf{e}';\mathsf{p}_2 \leftarrow \mathsf{e}_2;\mathsf{e}''] \Rightarrow []$$

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.

4 Our Architectures

We have developed an architecture, first presented in [41], for the semantic integration of OGSA-DAI enabled data sources, using AutoMed to produce the schema of the virtual integrated resource and to reconcile the semantic heterogeneities between the data source schemas and this integrated schema, and using OGSA-DQP to undertake distributed query processing. 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 comparison between them leads to significant conclusions on the performance impact of OGSA-DQP versus the centralised AutoMed query processor, and of accessing data sources using OGSA-DAI services versus direct access through JDBC.

4.1 Complete Architecture

Figure 1 illustrates our complete 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), using XML request/response documents to automatically retrieve schema metadata from a data source, via the OGSA-DAI service, and create the respective 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 schema matching [35] or schema transformation tools [42] 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.

An OGSA-DQP Evaluator service (QES) is deployed over each OGSA-DAI service. The OGSA-DQP Coordinator service (GDQS) can be deployed on any one of the available Grid nodes, but preferably on the same node as the AutoMed Query Processor to avoid data transfer costs between them.

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

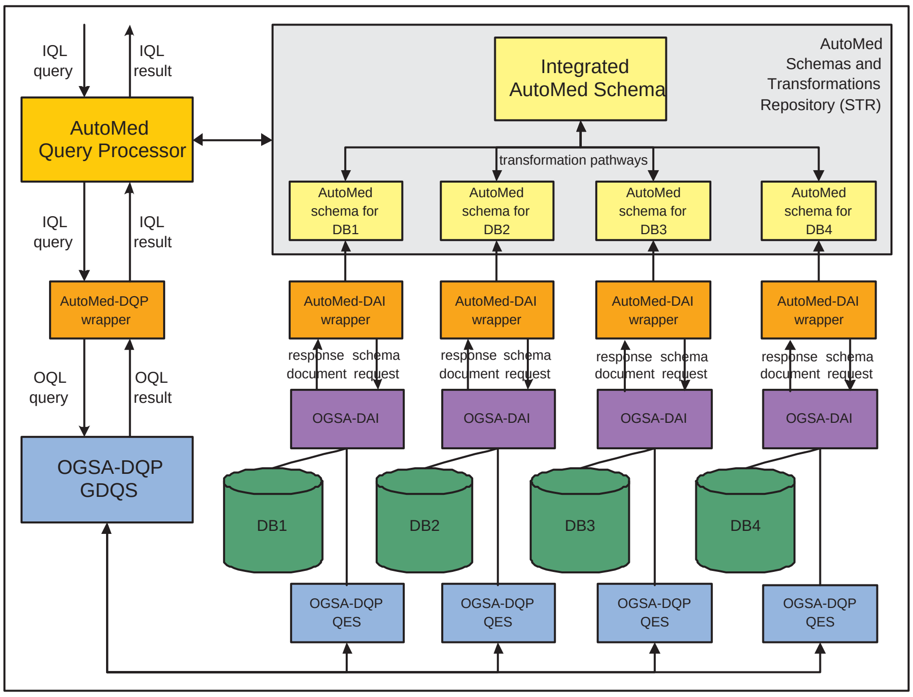


Figure 1: OGSA-DAI/OGSA-DQP/AutoMed Architecture

a high-level query language — SQL and XQuery are currently supported — in which case they are first translated by AutoMed into IQL[7].

An IQL query, Q, on an integrated schema is processed by **AutoMed's Query Processor (AQP)** as follows (as illustrated in Figure 2). 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[8]. The `VariableUnification` component then makes variable equality explicit within Q_{ref}, as discussed in Section 3.1. Next, the `QueryOptimiser` component optimises Q_{ref} by applying a number of query equivalences — as discussed in Section 3.2 — and by aiming to generate the largest possible subqueries that can be submitted to the data source wrappers for evaluation at the data sources. This results in an optimised query Q_{opt}. 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.

We note that in the context of the specific architecture of Figure 1, 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

[7]The SQL-to-IQL translator supports nested Select-Project-Join-Union queries, aggregation functions, and GROUP BY. The XQuery-to-IQL translator supports FLWR queries.

[8]The AQP currently supports GAV, LAV and combined GAV+LAV query reformulation [31]. The ISPIDER integrated schema is defined using GAV rules only, and thus in our performance study below only GAV query reformulation (i.e. query unfolding) is deployed.

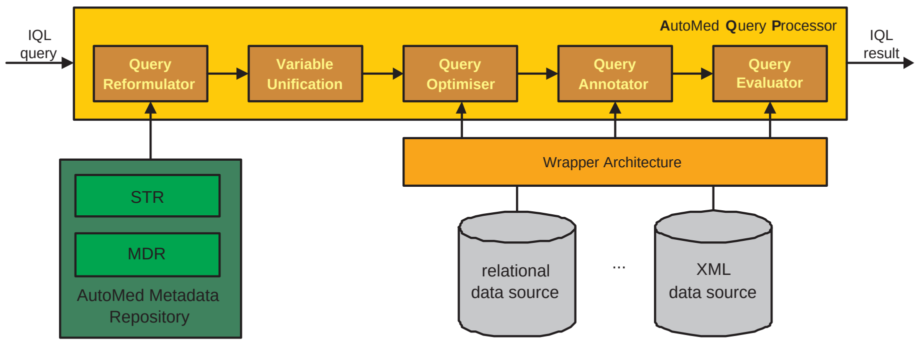


Figure 2: The AutoMed Query Processor Pipeline

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) 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 translated into an IQL result by the AutoMed-DQP wrapper and this is returned to the `QueryEvaluator`. This is responsible for any necessary post-processing of the wrapper results to produce the overall query result returned to the client application. As OGSA-DQP supports a subset of the OQL query language, and IQL is moreover more expressive than OQL, the original query Q submitted to AutoMed's AQP may not be fully translatable into a single OQL query, hence the possible need for such post-processing. In particular, the current release of OGSA-DQP only supports equijoins and one level of nesting, and does not support set operators, functions in the SELECT clause, or self-joins over tables.

We finally note that the AQP undertakes centralised query processing, interacting with the data sources as necessary, whereas OGSA-DQP offers decentralised query processing. Thus, subqueries within a query plan produced by the AQP's `QueryAnnotator` component are evaluated by AutoMed wrapper objects, located within AutoMed, which then return results to the AQP. In contrast, OGSA-DQP evaluates query execution plans produced by the GDQS in a decentralised manner, since each query partition is evaluated by a GQES service and these services are able to interact with each other and with the GDQS.

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 3. Thus, the integration process is the same, but query processing is performed solely by AutoMed. 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

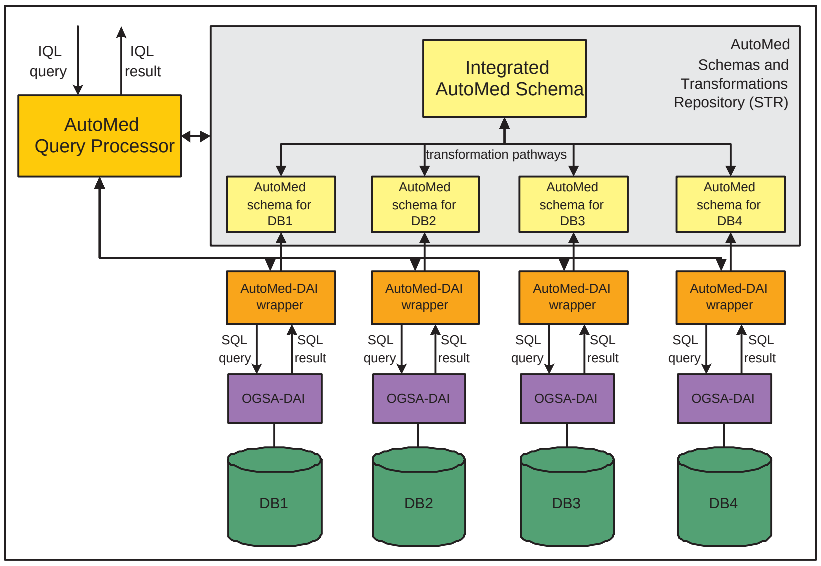


Figure 3: OGSA DAI/AutoMed Architecture

into an SQL query, and submits it to an OGSA-DAI service for evaluation[9]. As before, AutoMed's `QueryEvaluator` component post-processes the results returned by the wrappers and produces the overall query result.

OGSA-DAI services simply 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.

To utilise these greater translation capabilities of the AutoMed-DAI wrappers (compared to OGSA-DQP), AutoMed's `QueryOptimiser` component uses a different optimisation policy in this architecture, and so may produce a different Q_{opt} query for the same reformulated query Q_{ref} compared to the previous architecture. This in turn allows the `QueryAnnotator` to produce a different annotated query Q_{ann}, which may wrap larger subqueries than in the previous architecture.

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 4. 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 struc-

[9]For simplicity, we assume in our discussion here that the data sources are relational databases. This is indeed the case in the ISPIDER project, but need not be the case in general. All three of our architectural variants are able to handle non-relational data sources.

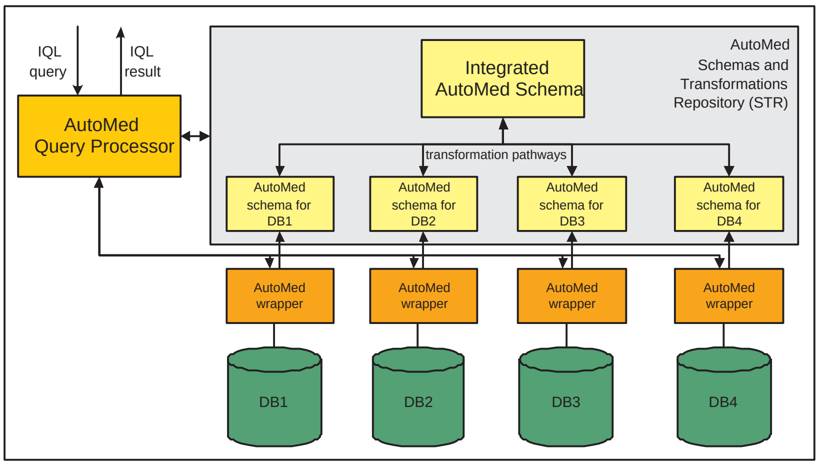


Figure 4: AutoMed-only Architecture

tured 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 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 standards from proteomics, bioinformatics and e-Science [37]. As part of this Grid platform, we have developed an integrated resource of four data sources containing experimental proteomics data. As discussed in [41], 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 [17] 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 [7] 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 [32] captures identification allied to peptide sequence data, coupled with the underlying ion series. ISPIDER used the first version of PepSeeker, containining over 185,000/135,000 peptide/protein hits (over 49,000/47,000 distinct ones) within a total of 4.2Gb of data.

- PRIDE [23] 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 5 illustrates the schema, IS, of the ISPIDER virtual integrated resource. Design of this schema started off with the PEDRo schema as a first version, extending this with constructs from the other three data source schemas as necessary. The ISPIDER project's proteomics experts guided us in deriving the correspondences between the data source schema constructs and the constructs of IS. These correspondences were then used to encode the necessary AutoMed transformation pathways.

A long-standing problem in the Life Sciences is the absence of commonly agreed identifiers for instances of biological entities. The common practice is to use integers which are unique only within the specific resource, and indeed this is the case with the four ISPIDER data sources. To identify entity instances in our virtual integrated schema IS, we have therefore generated life science identifiers, or *LSID*s [5], from the resource-specific identifiers.

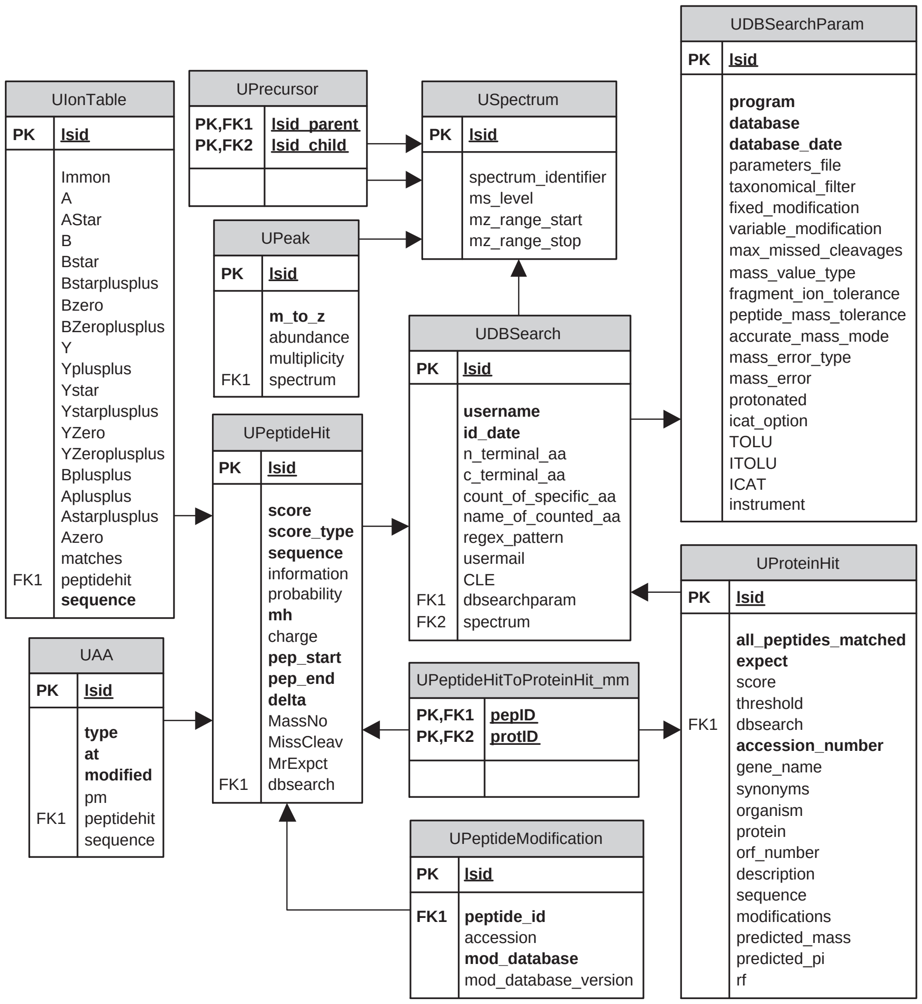


Figure 5: The ISPIDER Integrated Schema

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 *across* data sources. Thus, the conditions for applying Optimiser 3 above 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. 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.

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)

Our performance evaluation has been conducted using three different sets of queries, and investigating one performance factor at a time. For each query, we measure the time taken by: (i) set-up, (ii) optimisation, (iii) evaluation and (iv) the wrappers. The set-up time includes the time spent initialising the AQP, establishing connections with the data sources, and reformulating the query. The optimisation time includes the time spent by the `QueryOptimiser` and `QueryAnnotator` components of the AQP. The evaluation and wrappers times include the time spent by the `QueryEvaluator` and `AutoMedWrapper` components, respectively.

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 the AutoMed STR) and the `AutoMedWrapper` instances (creating connections with the data sources used for connection pooling), but not precomputing or caching any query results. 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 across 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. Finally, Section 5.7 investigates the impact of network speed on query processing in all three architectures.

5.3 User-Provided Queries

Our first set of experiments was with a set of queries provided by our biologist and bioinformatician partners over the ISPIDER integrated resource:

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. Queries Q^2 and Q^3 could have been written more simply using a join rather than checking for membership. However, reformulation of those versions of Q^2 and Q^3 would have resulted in self-joins in some of the data sources and, as discussed in Section 4, the current release of OGSA-DQP does not support self-joins in queries submitted to it.

To illustrate query processing over the integrated ISPIDER resource, consider query Q^1. After reformulation, this becomes query Q^1_{ref} below[10]:

```
[{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']
```

[10]Here, 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.

Table 2: User-provided Queries

Q^1:	[{an, lsid} | {lsid, an} ← ⟨⟨UProteinHit, accession_number⟩⟩; an = 'ENSP00000339074']
Q^2:	[{an, lsid} | {lsid, an} ← ⟨⟨UProteinHit, accession_number⟩⟩; member [lsid | {lsid, d} ← ⟨⟨UProteinHit, description⟩⟩; like d '%Actin%'] lsid]
Q^3:	[{an, lsid} | {lsid, an} ← ⟨⟨UProteinHit, accession_number⟩⟩; member [lsid | {lsid, o} ← ⟨⟨UProteinHit, organism⟩⟩; like o '%sapiens%'] lsid]
Q^4:	[{pr, sc} | {lsid1, pr} ← ⟨⟨UProteinHit, protein⟩⟩; {lsid2, seq} ← ⟨⟨UPeptideHit, sequence⟩⟩; seq = 'ATLTSDK'; {pepID, protID} ← ⟨⟨UPeptideHitToProteinHit_mm⟩⟩; lsid2 = pepID; lsid1 = protID; {lsid2, sc} ← ⟨⟨UPeptideHit, score⟩⟩]

Q^5:	[{an, lsid1, sc} | {lsid2, seq} ← ⟨⟨UPeptideHit, sequence⟩⟩; seq = 'LVNELTEFAK';
	{lsid1, an} ← ⟨⟨UProteinHit, accession_number⟩⟩; an = 'gi—229552';
	{pepID, protID} ← ⟨⟨UPeptideHitToProteinHit_mm⟩⟩;
	lsid2 = pepID; lsid1 = protID;
	{lsid2, sc} ← ⟨⟨UPeptideHit, score⟩⟩]

Q^6 :	[{an, seq, sc, pr, dbs} | {lsid1, an} ← ⟨⟨UProteinHit, accession_number⟩⟩; lsid1 = {'URN:LSID:ispider.man.ac.uk:pedro', 1069}; {pepID, protID} ← ⟨⟨UPeptideHitToProteinHit_mm⟩⟩; lsid1 = protID; {lsid2, seq} ← ⟨⟨UPeptideHit, sequence⟩⟩; lsid2 = pepID; {lsid2, sc} ← ⟨⟨UPeptideHit, score⟩⟩; {lsid2, pr} ← ⟨⟨UPeptideHit, probability⟩⟩; {lsid1, dbs} ← ⟨⟨UProteinHit, dbsearch⟩⟩]

We notice that the reformulated query contains a union of four comprehensions, each of which undertakes a Select-Project-Join query on one of the data sources. This is because, in the case of ISPIDER, the queries within the transformation pathways that populate the (virtual) global schema constructs from source schema constructs are themselves Select-Project-Join queries (there are no instances of grouping or aggregating, even though this is supported by AutoMed and IQL).

After optimisation, query Q^1_{ref} becomes Q^1_{opt} below. The `QueryOptimiser` component has simplified Q^1_{ref} using the optimisers discussed in Section 3.2. The IQL variables starting with a dollar character are system-generated variables generated by the optimisers:

```
[{$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^1_{opt} and identifies the largest subqueries that can be evaluated by `AutoMedWrapper` instances. The annotated query Q^1_{ann} is then submitted to the `QueryEvaluator`. For architectures A_1 and A_2, this annotated query contains four wrapper objects, one for each of the comprehensions above, since each comprehension refers to a different data source. For architecture A_3 the annotated query similarly has four instances of the OGSA-DQP wrapper: even though OGSA-DQP does have access to all four data sources concurrently, the current version does not support `UNION` and thus four separate queries need to be submitted to AutoMed-DAI wrapper instances by AutoMed's `QueryEvaluator`.

The other user-provided queries result in similarly annotated queries, i.e. containing up to four comprehensions being appended. Some queries contain fewer comprehensions because some data sources do not contribute to certain integrated schema constructs specified in the query. For example, Q_2 results in a single comprehension since only PEDRo contributes to construct $\langle\langle \mathsf{UProteinHit}, \mathsf{description} \rangle\rangle$.

Figure 6 shows the running times for the user-provided queries using all three architectures, and splitting the execution times into the four parts discussed earlier. 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.

We finally note that optimisation for query Q^6 takes more than 7 seconds. This query is a join involving half the tables of the integrated schema and containing 6 generators. After reformulation, these generators are sourced from 4, 3, 4, 2, 3 and 2 data sources respectively. As a result, after the application of Opt. 2 of Section 3.2, the query contains $\prod(4 * 3 * 4 * 2 * 3 * 2) = 576$ comprehensions. Opt. 1 and Opt. 3 greatly simplify this output of Opt. 2, but this requires a significant amount of time. Note that if optimisation were not performed on Q^6, then all the source data encompassed by its generators would need to be retrieved, incurring a huge data transfer and evaluation cost (much greater than 7 seconds).

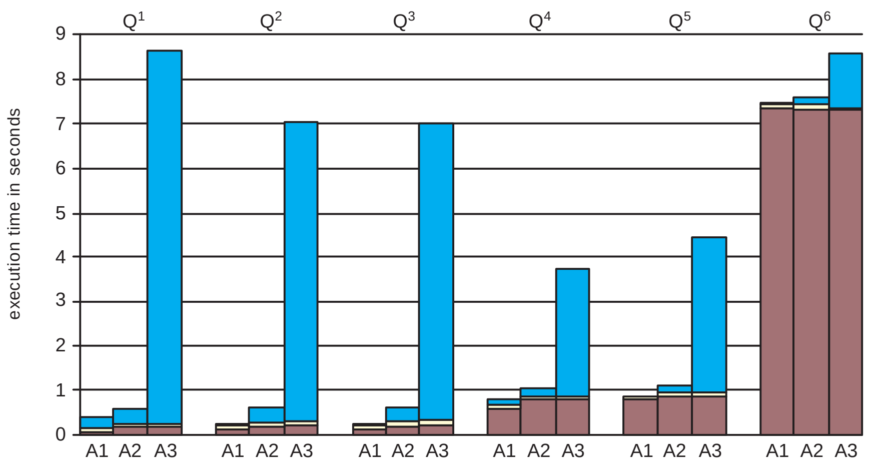
This last finding, combined with the fact that this issue is not uncommon in data integration settings and also with the non-negligible optimisation times for Q^4 and Q^5, points to the need for more work on improving query optimisation performance in AutoMed, e.g. by caching previous costly optimisation results, caching intermediate optimisation results, as well as re-implementing in a parallel fashion those optimisers that repeatedly traverse and rewrite queries.

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, apart from identifying a number of improvements needed, such as support for the `UNION` operator and improvement of the communication between its GDQS and GQES services, 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.

For the purposes of our performance evaluation, we have 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 (however, this second set of queries do not have an actual biological meaning).

Figure 7 illustrates the join pattern between the Q^7, Q^8 and Q^9 subsets of queries, after they have been reformulated and optimised. For example, query Q^{7a} results in a query performing an equijoin over the PRIDE and PepSeeker data sources, whereas query Q^{8a} results in a query performing an equijoin over gpmDB and PEDRo. Query Q^{9a} results in an equijoin over queries Q^{7a} and



	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3
🟦 wrappers	0.2	0.4	8.4	0	0.3	6.7	0	0.3	6.6	0.1	0.2	2.9	0	0.2	3.5	0	0.1	1.2
🟨 evaluation	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
🟫 optimisation	0.1	0.2	0.2	0.1	0.2	0.2	0.1	0.2	0.2	0.6	0.8	0.8	0.8	0.9	0.9	7.3	7.3	7.3
🟪 setup	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 6: Evaluation of User Queries for Architectures A_1 (left), A_2 (middle) and A_3 (right).

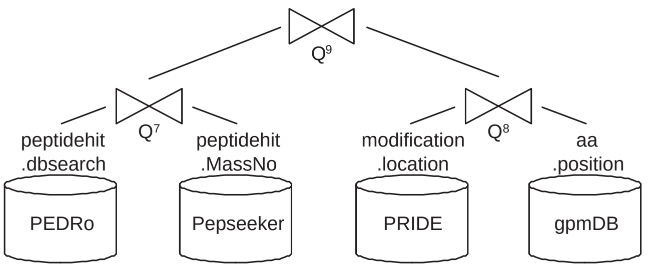


Figure 7: Queries Q^7-Q^9 Perform Joins Across Data Sources

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^{9b}, 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 $\mathsf{spi} = 1645$, and 559 tuples from PepSeeker, after the application of filter $\mathsf{m} > 1600$. The number of tuples input to each join operation is significant because, as discussed below, a bottleneck in processing this set of queries is observed with architectures A_1 and A_2 when the joins are evaluated by AutoMed's AQP (not OGSA-DQP, as in the complete architecture A_3). The AQP only supports the nested loops join algorithm, which is dependent on the size of its input arguments.

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

Table 3: Distributed Join Queries

Q^{7a}:	[{m}|{sp, spi} ← ⟨⟨USpectrum, spectrum_identifier⟩⟩; spi = 1645; {{peph1, peph2}, m} ← ⟨⟨UPeptideHit, MassNo⟩⟩; m > 1600; m = spi]
Q^{7b}:	[{m}|{sp, spi} ← ⟨⟨USpectrum, spectrum_identifier⟩⟩; spi > 1600; spi < 2365 {{peph1, peph2}, m} ← ⟨⟨UPeptideHit, MassNo⟩⟩; m > 1600; m = spi]
Q^{7c}:	[{m}|{sp, spi} ← ⟨⟨USpectrum, spectrum_identifier⟩⟩; spi > 1600; spi < 3740 {{peph1, peph2}, m} ← ⟨⟨UPeptideHit, MassNo⟩⟩; m > 1600; m = spi]
Q^{8a}:	[{at}|{{aa1, aa2}, at} ← ⟨⟨UAA, at⟩⟩; aa2 < 50000; at > 1600; at < 1650; {{peph1, peph2}, {d1, d2}} ← ⟨⟨UPeptideHit, dbsearch⟩⟩; d2 > 1600; d2 < 1700; at = d2; peph2 > 6200; peph2 < 6251]
Q^{8b}:	[{at}|{{aa1, aa2}, at} ← ⟨⟨UAA, at⟩⟩; aa2 < 50000; at > 1600; at < 2100 {{peph1, peph2}, {d1, d2}} ← ⟨⟨UPeptideHit, dbsearch⟩⟩; d2 > 1600; d2 < 1700; at = d2; peph2 > 6200; peph2 < 6376]
Q^{8c}:	[{at}|{{aa1, aa2}, at} ← ⟨⟨UAA, at⟩⟩; aa2 < 50000; at > 1600; {{peph1, peph2}, {d1, d2}} ← ⟨⟨UPeptideHit, dbsearch⟩⟩; d2 > 1600; d2 < 1700; at = d2; peph2 > 6200; peph2 < 6501]

Q^{9a}:	[{k}|{k} ← [{m}|{sp, spi} ← ⟨⟨USpectrum, spectrum_identifier⟩⟩; spi = 1645;
	{{peph1, peph2}, m} ← ⟨⟨UPeptideHit, MassNo⟩⟩; m > 1600; m = spi];
	{k} ← [{at}|{{aa1, aa2}, at} ← ⟨⟨UAA, at⟩⟩; aa2 < 50000; at > 1600; at < 1650;
	{{peph1, peph2}, {d1, d2}} ← ⟨⟨UPeptideHit, dbsearch⟩⟩;
	d2 > 1600; d2 < 1700; at = d2; peph2 > 6200; peph2 < 6251]]

Q^{9b}:	[{k}|{k} ← [{m}|{sp, spi} ← ⟨⟨USpectrum, spectrum_identifier⟩⟩; spi > 1600; spi < 2365 {{peph1, peph2}, m} ← ⟨⟨UPeptideHit, MassNo⟩⟩; m > 1600; m = spi]; {k} ← [{at}|{{aa1, aa2}, at} ← ⟨⟨UAA, at⟩⟩; aa2 < 50000; at > 1600; at < 2100 {{peph1, peph2}, {d1, d2}} ← ⟨⟨UPeptideHit, dbsearch⟩⟩; d2 > 1600; d2 < 1700; at = d2; peph2 > 6200; peph2 < 6376]]
Q^{9c}:	[{k}|{k} ← [{m}|{sp, spi} ← ⟨⟨USpectrum, spectrum_identifier⟩⟩; spi > 1600; spi < 3740 {{peph1, peph2}, m} ← ⟨⟨UPeptideHit, MassNo⟩⟩; m > 1600; m = spi]; {k} ← [{at}|{{aa1, aa2}, at} ← ⟨⟨UAA, at⟩⟩; aa2 < 50000; at > 1600; {{peph1, peph2}, {d1, d2}} ← ⟨⟨UPeptideHit, dbsearch⟩⟩; d2 > 1600; d2 < 1700; at = d2; peph2 > 6200; peph2 < 6501]]

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

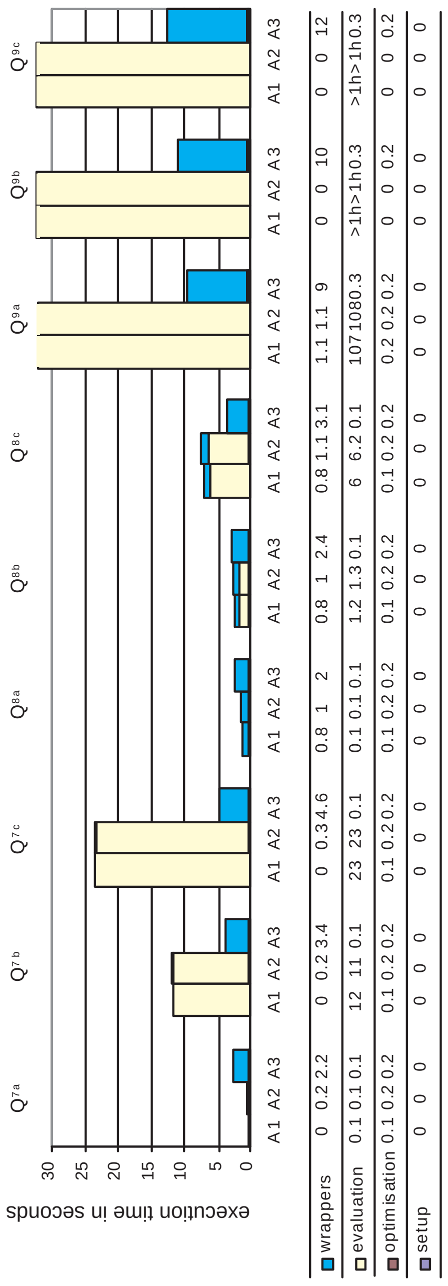
Data Source	Q^{7a}	Q^{7b}	Q^{7c}	Q^{8a}	Q^{8b}	Q^{8c}	Q^{9a}	Q^{9b}	Q^{9c}
PRIDE	4	1,374	2,750				4	1,374	2,750
PepSeeker	559	559	559				559	559	559
gpmDB				37	430	968	37	430	968
PEDRo				50	175	300	50	175	300
Tuples output	2,236	2,236	2,236	28	112	194	15,652	15,652	15,652

$Q^{9a} - Q^{9b}$ — 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 user-provided first set of queries earlier).

The running times for this second set of queries using the three different architectures are shown in Figure 8. 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, and also supports a distributed hash-join implementation.

The results of this and the previous section indicate that, although AutoMed's AQP is able to adequately handle all stages of query processing for SPU queries, plus joins within data sources, its `QueryEvaluator` component cannot efficiently evaluate queries which contain joins across data sources.

This finding validates the design of our overall architecture A_3, which com-



	Q7a A1	Q7a A2	Q7a A3	Q7b A1	Q7b A2	Q7b A3	Q7c A1	Q7c A2	Q7c A3	Q8a A1	Q8a A2	Q8a A3	Q8b A1	Q8b A2	Q8b A3	Q8c A1	Q8c A2	Q8c A3	Q9a A1	Q9a A2	Q9a A3	Q9b A1	Q9b A2	Q9b A3	Q9c A1	Q9c A2	Q9c A3
■ wrappers	0	0.2	2.2	0	0.2	3.4	0	0.3	4.6	0.8	1	2	0.8	1	2.4	0.8	1.1	3.1	1.1	1.1	9	0	0	10	0	0	12
□ evaluation	0.1	0.1	0.1	12	11	0.1	23	23	0.1	0.1	0.1	0.1	1.2	1.3	0.1	6	6.2	0.1	107	108	0.3	>1h	>1h	0.3	>1h	>1h	0.3
■ optimisation	0.1	0.2	0.2	0.1	0.2	0.2	0.1	0.2	0.2	0.1	0.2	0.2	0.1	0.2	0.2	0.1	0.2	0.2	0.2	0.2	0.2	0	0	0.2	0	0	0.2
■ setup	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 8: Evaluation of Queries Q^{7a}-Q^{9b} for Architectures A_1 (left), A_2 (middle) and A_3 (right)

bines the respective advantages of AutoMed for fine-grained data transformation and integration, and OGSA-DQP for Grid-based distributed query processing.

5.5 Incremental Query Evaluation

AutoMed's AQP supports both instant and incremental query evaluation. In particular, the `QueryEvaluator` supports incremental evaluation of IQL operators that take at least one collection-valued argument. With this mode of evaluation, when evaluating a collection-valued expression, a single call to the `QueryEvaluator` will return only n results of the final result set, rather than the full result; and the full result is evaluated in as many successive calls as are necessary of the `QueryEvaluator`, each returning a packet of n results. Both JDBC and OGSA-DAI also support incremental evaluation, while OGSA-DQP V3.1 operates in incremental mode only. Thus, the query processing pipelines of all three architectures are able to support this evaluation mode.

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. On the other hand, 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 evaluation for the AQP, and the results confirmed our expectations: the running times for all queries and architectures are only slightly slower than when using instant 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 instant 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 instant, but not parallel incremental, evaluation. This `ParallelQueryEvaluator` [16] parallelises the evaluation of operators with at least two collection-valued arguments (such as ++).

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:

$$[\{\mathsf{lsid}, \mathsf{id}\} | \{\mathsf{lsid}, \mathsf{id}\} \leftarrow \langle\langle \mathsf{UPeptideHit} \rangle\rangle; \mathsf{id} < 5000]$$

and the corresponding query after reformulation and optimisation is:

$$\begin{array}{l}
[\{\mathsf{\$d1}, \mathsf{'URN{:}LSID{:}ispider.man.ac.uk{:}pepseeker'}\} | \mathsf{\$d1} \leftarrow \mathsf{pepseeker} : \langle\langle \mathsf{peptidehit} \rangle\rangle; \mathsf{\$d1} < 5000] \\
++ [\{\mathsf{\$d2}, \mathsf{'URN{:}LSID{:}ispider.man.ac.uk{:}pedro'}\} | \{\mathsf{\$d2}, \mathsf{\$e}\} \leftarrow \mathsf{pedro} : \langle\langle \mathsf{peptidehit} \rangle\rangle; \mathsf{\$d2} < 5000] \\
++ [\{\mathsf{\$d3}, \mathsf{'URN{:}LSID{:}ispider.man.ac.uk{:}gpmdb'}\} | \mathsf{\$d3} \leftarrow \mathsf{gpmdb} : \langle\langle \mathsf{peptide} \rangle\rangle; \mathsf{\$d3} < 5000] \\
++ [\{\mathsf{\$d4}, \mathsf{'URN{:}LSID{:}ispider.man.ac.uk{:}pride'}\} | \mathsf{\$d4} \leftarrow \mathsf{pride} : \langle\langle \mathsf{pride_peptide} \rangle\rangle; \mathsf{\$d4} < 5000]
\end{array}$$

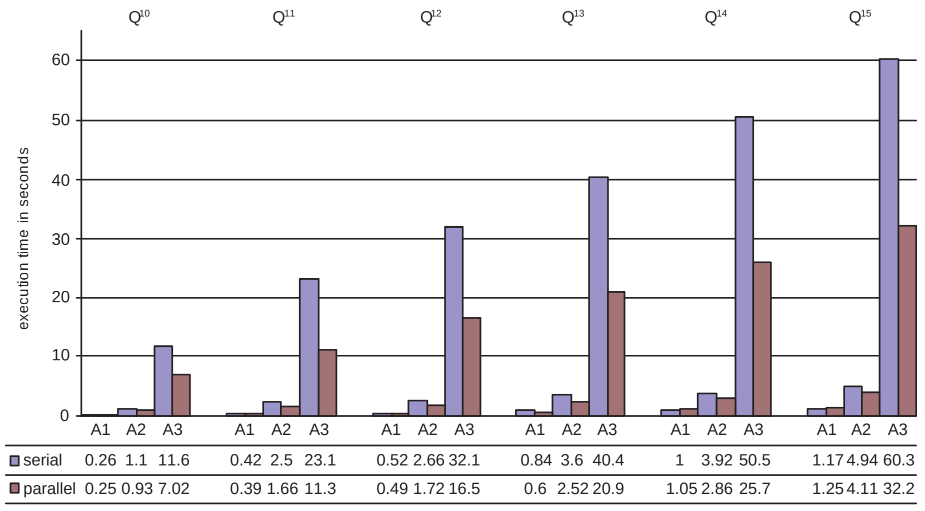
Looking at this optimised query, we see that the annotated query will contain 4 wrappers for all three architectures. The other five queries in this new query set contain a different constant within the filter in the comprehension, using a step of 5,000 from 5,000 up to 30,000, which has the effect of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

Figure 9 shows the running times for these six new queries. The results at first seem contradictory. On the one hand, results for A_1 are the same for serial and parallel evaluation. On the other hand, parallel evaluation results in a small benefit for A_2 and a significant speedup for A_3.

Closer examination offers valuable insights on parallel query processing in a Grid data integration setting. The impressive benefit for A_3 is a result of parallelising interactions with the wrappers, i.e. the time spent evaluating queries at the data sources and transmitting the results back to the AQP. As discussed in Section 5.3 and illustrated in Figure 6, 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. On the other hand, A_1 performs very 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.

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 $\mathsf{Q}_1 ++ \mathsf{Q}_2$, where Q_1 and Q_2 are subqueries sent to AutoMed wrappers for evaluation, would benefit significantly by parallel processing of the ++ operator.

We finally note that the AQP does not support combined parallel and incremental evaluation, even though they both support each feature in isolation. Implementing this additional capability, and evaluating its effect on our three architectures, would be an area of interesting future work.



	Q10 A1	Q10 A2	Q10 A3	Q11 A1	Q11 A2	Q11 A3	Q12 A1	Q12 A2	Q12 A3	Q13 A1	Q13 A2	Q13 A3	Q14 A1	Q14 A2	Q14 A3	Q15 A1	Q15 A2	Q15 A3
▨ serial	0.26	1.1	11.6	0.42	2.5	23.1	0.52	2.66	32.1	0.84	3.6	40.4	1	3.92	50.5	1.17	4.94	60.3
▨ parallel	0.25	0.93	7.02	0.39	1.66	11.3	0.49	1.72	16.5	0.6	2.52	20.9	1.05	2.86	25.7	1.25	4.11	32.2

Figure 9: 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 limiting[11] 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

[11]Rate 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.

options, L_1, L_2, L_3 and L_4. Figure 10 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[12]), and so drawing overall conclusions is not straightforward.

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 10. 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}.

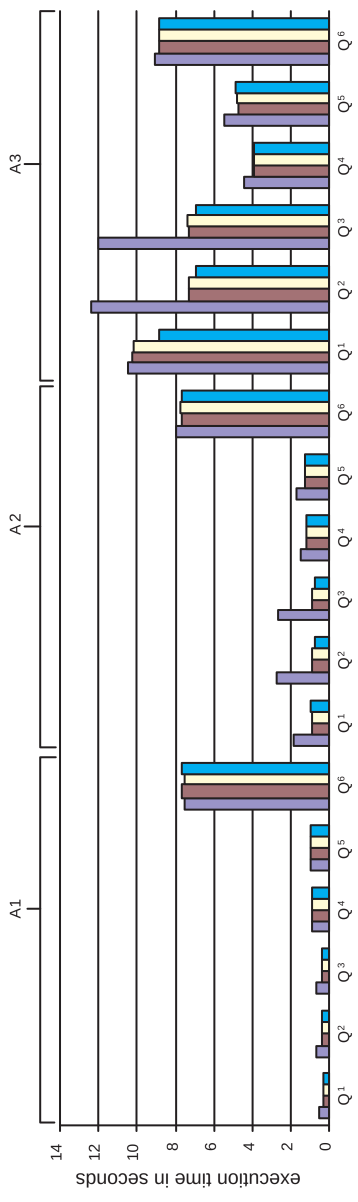
These results highlight the need for any deployed architecture to be able to determine the speed characteristics of the connections between Grid nodes and to support cost optimisers that provide alternative query plans based on these characteristics. A closer examination of this aspect of query optimisation 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.

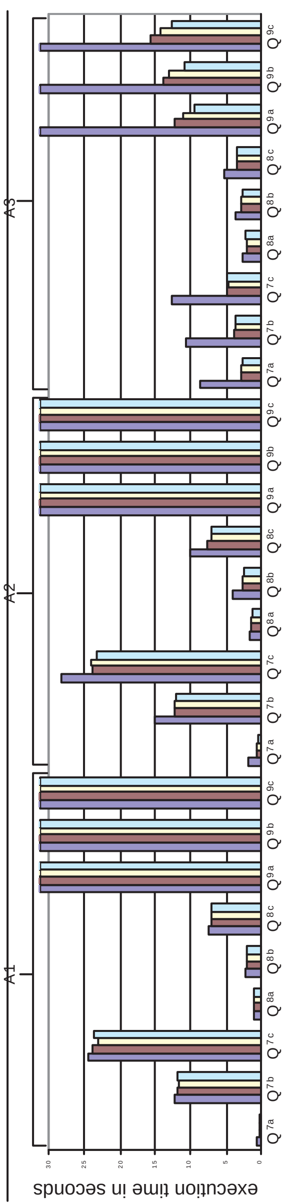
6 Conclusions and Future Work

In earlier work, we described an architecture for the virtual integration of heterogeneous Grid resources that provides complex data transformation and integration capabilities coupled with distributed query processing over heterogeneous data sources. To achieve this, we combined state-of-the-art technologies in data integration, namely the AutoMed system, and in Grid access and query process-

[12]Note that this slowdown is a result of these queries employing a membership sub-query, rather than a self-join, due to OGSA-DQP's inability 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.



	Q1	Q2	Q3	Q4	Q5	Q6	Q1	Q2	Q3	Q4	Q5	Q6	Q1	Q2	Q3	Q4	Q5	Q6
▨ L1	0.525	0.66	0.629	0.913	1.025	7.578	1.851	2.696	2.673	1.499	1.758	7.971	10.47	12.39	11.99	4.433	5.479	9.095
▨ L2	0.272	0.396	0.366	0.898	0.975	7.605	0.927	0.878	0.87	1.172	1.298	7.66	10.25	7.292	7.233	3.936	4.73	8.829
▨ L3	0.278	0.388	0.36	0.892	0.965	7.547	0.916	0.88	0.912	1.2	1.293	7.697	10.18	7.283	7.322	3.894	4.778	8.855
▨ L4	0.253	0.371	0.348	0.88	0.966	7.593	0.98	0.742	0.725	1.152	1.281	7.648	8.853	6.917	6.896	3.93	4.899	8.81



	Q7a	Q7b	Q7c	Q8a	Q8b	Q8c	Q9a	Q9b	Q9c	Q7a	Q7b	Q7c	Q8a	Q8b	Q8c	Q9a	Q9b	Q9c	Q7a	Q7b	Q7c	Q8a	Q8b	Q8c	Q9a	Q9b	Q9c
▨ L1	0.67	12.3	24.4	1.08	2.39	7.42	108	>1h	>1h	1.82	15.0	28.3	1.73	3.8	10	111	>1h	>1h	8.57	10.6	12.7	2.59	3.7	5.36	50.1	53.5	56.9
▨ L2	0.26	11.8	23.7	1.08	2.21	7.00	108	>1h	>1h	0.57	12.2	23.7	1.38	2.63	7.75	110	>1h	>1h	2.77	3.94	4.87	2.21	2.69	3.35	12.1	13.8	15.5
▨ L3	0.26	11.6	23.2	1.08	2.21	7.05	107	>1h	>1h	0.57	12.2	23.9	1.39	2.65	7.45	110	>1h	>1h	2.72	3.63	4.81	2.21	2.71	3.34	11.0	12.9	14.2
▨ L4	0.25	11.7	23.4	1.08	2.20	7.00	107	>1h	>1h	0.49	11.9	23.3	1.33	2.56	7.46	109	>1h	>1h	2.54	3.66	4.93	2.25	2.65	3.35	9.51	10.9	12.6

Figure 10: Left: Instant Evaluation of Queries Q^1-Q^6 for Architectures A_1, A_2 and A_3 and Link Speeds L_1-L_4. Right: Instant Evaluation of Queries Q^{7a}-Q^{9c} for Architectures A_1, A_2 and A_3 and Link Speeds L_1-L_4.

ing, 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 data transformation/integration with distributed query processing, and we believe that our findings will be of benefit to others wishing to combine these capabilities to support Grid applications requiring sophisticated data integration and querying over distributed heterogeneous data resources.

Our investigation has demonstrated that our full architecture is able to efficiently evaluate Select-Project-Join-Union queries over Grid-enabled heterogeneous data sources. Combining data transformation/integration capabilities, as exemplified by AutoMed, with the OGSA's data access and distributed query processing middleware has thus been demonstrated to be feasible from a query performance perspective.

Our investigation has also demonstrated that while AutoMed alone can efficiently evaluate Select-Project-Union queries on the global schema, the full architecture, including OGSA-DQP, is needed in order to efficiently evaluate queries that involve distributed joins across data sources. We have also identified some shortcomings of OGSA-DQP Version 3.1, including the impact on query performance due to lack of support for the UNION operator and the inefficiency of Select-Project queries, and these findings have been communicated to the OGSA-DQP team.

For the future, there are two possible ways forward to further improve query performance in our architecture: either extend OGSA-DQP with these additional capabilities, or extend the AutoMed query processor with more sophisticated query processing functionality, including a physical algebra for IQL, cost-based optimisation and query planning, and distributed query processing. OGSA-DAI itself also requires further extension so as to export additional metadata from data sources (if this is available) in order support more effective cost-based optimisation (e.g. information about dataset sizes, distributions and access paths) — and again, we have communicated this recommendation to the OGSA-DAI team.

Our future work includes: providing JDBC and OGSA-DAI interfaces over the AutoMed Query Processor, in order to promote interoperability between AutoMed and other Grid middleware, such as Taverna [20]; investigating and handling the impact of evolutions of the integrated and data source schemas c.f. [12]; and extending our architecture to leverage the benefits of ontology-based data integration [43]. The ISPIDER integrated resource itself will shortly be made publicly available by deploying AutoMed as a service within the ISPIDER Central website [37].

Acknowledgements

The authors would like to thank the following people: Dimitris Fourkiotis and Jamie Walters for providing the initial implementations of AutoMed's parallel evaluator and SQL-to-IQL translator, respectively; our ISPIDER collaborators from the University of Manchester for providing the initial set of queries; and Steven Lynden, Arijit Mukherjee and Ally Hume for their help with OGSA-DAI and OGSA-DQP.

References

[1] M. N. Alpdemir et al. Service-based distributed querying on the Grid. In *Proc. Int. Conference on Service Oriented Computing*, pages 467–482, 2003.

[2] M. N. Alpdemir et al. Experience on performance evaluation with OGSA-DQP. In *Proc. U.K. e-Science All Hands Meeting (AHM'05)*, 2005.

[3] M. Antonioletti et al. The design and implementation of Grid database services in OGSA-DAI. *Concurrency - Practice and Experience*, 17(2–4):357–376, 2005.

[4] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.

[5] T. Clark, S. Martin, and T. Liefeld. Globally distributed object identification for biological knowledgebases. *Briefings in Bioinformatics*, 5(1):59–70, 2004.

[6] C. Comito and D. Talia. XML data integration in OGSA Grids. In *Proc. Data Management in Grids (DMG at VLDB'05)*, pages 4–15, 2005.

[7] R. Craig, J. P. Cortens, and R. C. Beavis. Open source system for analyzing, validating, and storing protein identification data. *Journal of Proteome Research*, 3(6), 2004.

[8] S. B. Davidson et al. K2/Kleisli and GUS: Experiments in integrated access to genomic data sources. *IBM Systems Journal*, 40(2):512–531, 2001.

[9] S. B. Davidson, C. Overton, V. Tannen, and L. Wong. BioKleisli: A digital library for biomedical researchers. *Int. J. on Digital Libraries*, 1(1):36–53, 1997.

[10] B. Dobrzelecki et al. Profiling OGSA-DAI performance for common use patterns. In *Proc. U.K. e-Science All Hands Meeting (AHM'06)*, 2006.

[11] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *Proc. ACM Symposium on Principles of Database Systems (PODS97)*, pages 109–116, 1997.

[12] H. Fan and A. Poulovassilis. Schema evolution in data warehousing environments — a schema transformation-based approach. In *Proc. International Conference on Conceptual Modeling (ER'04)*, pages 639–653, 2004.

[13] 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.

[14] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000.

[15] I. Foster et al. 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`.

[16] 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.dcs.bbk.ac.uk/ lucas/msc/Fou07.pdf.

[17] K. Garwood et al. Pedro: A database for storing, searching and disseminating experimental proteomics data. *BMC Genomics*, 5(1), 2004.

[18] C. A. Goble et al. Transparent access to multiple bioinformatics information sources. *IBM Systems Journal*, 40(2):532–551, 2001.

[19] L. M. Haas et al. Discoverylink: A system for integrated access to life sciences data sources. *IBM Systems Journal*, 40(2):489–511, 2001.

[20] D. Hull, K. Wolstencroft, R. Stevens, C. A. Goble, M. R. Pocock, P. Li, and T. M. Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(2):729–732, 2006.

[21] E. Jasper, A. Poulovassilis, L. Zamboulis, and Hao Fan. Processing IQL queries and migrating data in the AutoMed toolkit. AutoMed Technical Report 20, July 2006.

[22] A. R. Jones et al. The Functional Genomics Experiment model (FuGE): an extensible framework for standards in functional genomics. *Nature Biotech.*, 25(10):1127–1133, 2007.

[23] P. Jones et al. PRIDE: a public repository of protein and peptide identifications for the proteomics community. *Nucleic Acids Research*, 1(34):659–663, 2006.

[24] S. Kottha, K. Abhinav, R. Müller-Pfefferkorn, and H. Mix. Accessing biodatabases with OGSA-DAI - a performance analysis. In *Proc. Int. Workshop on Distributed, High-Performance and Grid Computing in Computational Biology (GCCB'06)*, pages 141–156, 2006.

[25] 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.

[26] M. Lenzerini. Data integration: A theoretical perspective. In *Proc. ACM Symposium on Principles of Database Systems (PODS02)*, pages 233–246, 2002.

[27] A. Levy, A. Rajamaran, and J. Ordille. Querying heterogeneous information sources using source description. In *Proc. Int. Conf. on Very Large Data Bases (VLDB'96)*, pages 252–262, 1996.

[28] P. J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Proc. International Conference on Advanced Information Systems Engineering (CAiSE'99)*, pages 333–348, 1999.

[29] P. J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. International Conference on Data Engineering (ICDE'03)*, pages 227–238, 2003.

[30] P. J. McBrien and A. Poulovassilis. 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.

[31] P. J. McBrien and A. Poulovassilis. 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.

[32] T. McLaughlin et al. Pepseeker: a database of proteome peptide identifications for investigating fragmentation patterns. *Nucleic Acids Research*, 34(1), 2006.

[33] A. Poulovassilis and C. Small. Algebraic query optimisation for database programming languages. *VLDB J.*, 5(2):119–132, 1996.

[34] 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.

[35] N. Rizopoulos. Automatic discovery of semantic relationships between schema elements. In *Proc. International Conference on Enterprise Information Systems (ICEIS'04)*, pages 3–8, 2004.

[36] J. Saltz et al. caGrid: design and implementation of the core architecture of the cancer biomedical informatics grid. *Bioinformatics*, 22(15):1910–1916, 2006.

[37] J. A. Siepen, K. Belhajjame, J. N. Selley, S. Embury, N. W. Paton, C. A. Goble, S. G. Oliver, R. Stevens, L. Zamboulis, N. J. Martin, A. Poulovassilis, P. Jones, R. Cote, H. Hermjakob, M. Pentony, D. T. Jones, C. Orengo, and S. J. Hubbard. ISPIDER Central: an integrated database web-server for proteomics. *Nucleic Acids Research*, 36(2):485–490, 2008.

[38] J. Smith, A. Gounaris, P. Watson, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou. Distributed query processing on the Grid. In *Proc. Grid Computing*, pages 279–290, 2002.

[39] The Gene Ontology Consortium. Gene Ontology: tool for the unification of biology. *Nature Genet.*, 25:25–29, 2000.

[40] G. G. Trevisol et al. A distributed query execution engine in a grid environment. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 418–425, 2007.

[41] L. Zamboulis, H. Fan, K. Belhajjame, J. A. Siepen, A. Jones, N. J. Martin, A. Poulovassilis, S. Hubbard, S. M. Embury, and N. W. Paton. Data access and integration in the ISPIDER proteomics Grid. In *Proc. Data Integration in the Life Sciences (DILS'06)*, pages 3–18, 2006.

[42] L. Zamboulis and A. Poulovassilis. Information sharing for the Semantic Web - a schema transformation approach. In *Proc. International Workshop Data Integration and the Semantic Web (at CAiSE'06)*, pages 275–289, 2006.

[43] L. Zamboulis, A. Poulovassilis, and J. Wang. Ontology-assisted data transformation and integration. In *Proc. Ontologies-Based Databases and Information Systems (ODBIS at VLDB'08)*, page TBC, 2008.

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