Master's Thesis

Ibidas: to serve and to protect

Thesis Committee:
Prof.dr.ir. M.J.T. Reinders
Prof.dr.ir. A. P. de Vries
Ir. J.J. Bot
Ir. M. Hulsman
Dr. M. Roos

Author	**Patrick van Kouteren**
Email	pvankouteren@gmail.com
Student number	1384953
Thesis supervisor	Prof.dr.ir. M.J.T. Reinders Ir. J.J. Bot
Date	November 9, 2009 - 14:00

Information and Communication Theory Group [I,C]^T

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

Preface

This document is the result of research, preceeded by a literature study, done on the semantic Web, data warehousing and database security in the context of biological entities. The document consists of two parts: a final report and a work document. The final report consists of the introduction, implementation of the two main features and the conclusions of the research. The work document also includes some minor features and preparations regarding the implementation, design choices made, planning and progress.

This research was done in the ICT group at Delft University of Technology, and supervised by prof. dr. ir. M.J.T. Reinders and ir. J.J. Bot. The source code of Ibidas is available upon request.

Acknowledgements

Although a lot of people supported me during my graduation period, I would like to mention a couple of people who were important in making this project possible and keeping me motivated. First of all I would like to thank Marcel, Jan and Marc for giving me the opportunity to work on a subject I enjoy working on, even though it's only partially related to bioinformatics. Working and discussing design and implementation issues with Jan and Marc provided me much insights in design choices as well as a good reflection of my way of programming while learning from their ways of getting things done. When being caught up too much in implementation details, Marcel provided insights from another angle which often transformed a problem into a better solvable one. Also while writing this thesis, Jan and Marcel provided me with useful feedback on my style of writing. As a result of that, the difference between the first draft and this final document is huge. My thanks also go to Floris Sluiter working at SARA for providing the basic security model.

Furthermore I'd like to thank all the students for being present and participating at the student meetings. What goes around, comes around: as I often had something questions and comments about their presentations, they also provided me with useful feedback to improve my way of presenting my work; the students I got to work with on the 11th floor for having discussions varying from the quality of the coffee to specific issues regarding my project. Special thanks goes out to Bas and Tisha, who were by far most present in 'the glass room' and always in the mood for any type of discussion.

Of course I could not have done this without the everlasting support of my family, girlfriend and friends. Good times are easy, but during hard times they were at their best by accepting my situation and doing everything to be of any help. Big things really come in small packages.

Contents

1 Introduction **6**

2 Database security **10**
2.1 Virtual private databases 10
2.2 Postgres login roles and usergroups 12
2.3 Implementation 13
2.4 Query performance analysis 14
2.4.1 Expectations 14
2.4.2 Results 16

3 Web services **19**
3.1 Libraries and set up 19
3.2 Implementation 20
3.2.1 AtomicServices 21
3.2.2 SOAP server 21
3.2.3 SOAP serializers 21

4 Using Ibidas in workflows **24**
4.1 Example Taverna workflow 24
4.2 Example Taverna workflow execution 27

5 Discussion and future work **28**
5.1 Discussion 28
5.1.1 Usergroup resolution 28
5.1.2 Joining order 29
5.2 Future work 29
5.2.1 RESTful transferring 29
5.2.2 Authenticated Web services 30
5.2.3 Ibidas sessions 30
5.2.4 JSON 30

List of Tables

2.1 PostgreSQL explanation of the speedtest queries 16

4.1 Inputs for our Taverna workflow 27

List of Figures

1.1 Overview of the intended setup of an Ibidas instance. 7
1.2 An example scenario to clarify the application of Ibidas 9

2.1 The options to apply filtering of data 11
2.2 Security implementation overview. 15
2.3 Query times per run 18

3.1 Overview of the layers which are used with Web service access. . 20
3.2 Overview of the SOAP serializers present in Ibidas. 23

4.1 Venn diagram of gene set enrichment analysis 25
4.2 Execution of the scenario 26

Abstract

In this thesis we present the database security mechanism and Web services implementation for the Ibidas system. The security mechanism restricts access to data by source per user. Web services provide an interface to call Ibidas methods and share data from and with other applications. We will see the need for the features presented, how they are implemented and the choices which were made during implementation. Furthermore we will show an example of the use of the Web services using the Taverna workbench application to solve a typical situation a researcher could face. We conclude this thesis with a discussion regarding our implementation and some future work indications.

Chapter 1

Introduction

The World Wide Web contains a large amount of information on all kinds of aspects on biological entities like genes and proteins. However, this information is often stored in a flat way (e.g. text files) and for interpreting the data certain knowledge on the origin of the data is needed. This knowledge is not explicitly defined, which means that only people having this knowledge know what the data represents.

Measuring any type of property of proteins results in a set of data containing specific information on those properties (e.g. protein mass, structure characteristics etc.). For using this data, background knowledge (meta-data) on the meaning of the results is required. A transition is currently going on, enriching the Web with additional knowledge in the form of meta-data and ontologies: semantics. These semantics add meaning to data in such a way that relationships between data can be derived from these semantics. This is especially useful when composing a data warehouse as the various types of data can then be integrated by use of these semantics. This means that knowledge is not system-specific any more: Genes which are in a different database can be related to proteins elsewhere in a particular type of relationship (e.g. transcription factor, gene product etc.).

The data which is put in a data warehouse can originate from a variety of sources. The soure of data is actually of importance as not all data may be publicly available. This can be due to commercial interests (e.g. pharmaceutical companies, biotechnology companies etc.) or due to privacy (e.g. patient data). Such data needs a form of protection to only grant authorized users access to that data.

Although data can be related and stored in a data warehouse, tools are needed to be able to work with this data. External applications can be used to solve this. For providing the appropriate data to these applications, web services are becoming more and more popular. These web services are running server-sided and can be queried for data. Already established tools (e.g. workbench applications like Taverna[9, 20] or visualization tools like Cytoscape[1, 17]) can benefit from this. Such interoperability between various well-used systems is of interest for the biologists as it takes away the unnecessary data (pre-)processing and eliminates human errors during such data processing steps.

Ibidas provides a data integration platform enabling integration of data originating from different sources for bioinformatics purposes in a non-persistent

way. The coupled PostgreSQL[13] database allows for persistent storage of the integrated data if desired. Next to plain storage of raw sequences and accessions of genes and proteins, it allows linking these data together. These data types with different namespaces are mapped to one uniform namespace, which allows for easy data conversion. This means the data can be represented in a uniform format which makes working with the data easier. For working with this data, the Ibidas system can be used directly, but it can also be used to execute parts of a workflow like computations or just data provisioning. Figure 1.1 shows a schematic overview of the Ibidas system and indicates the positions of the features.

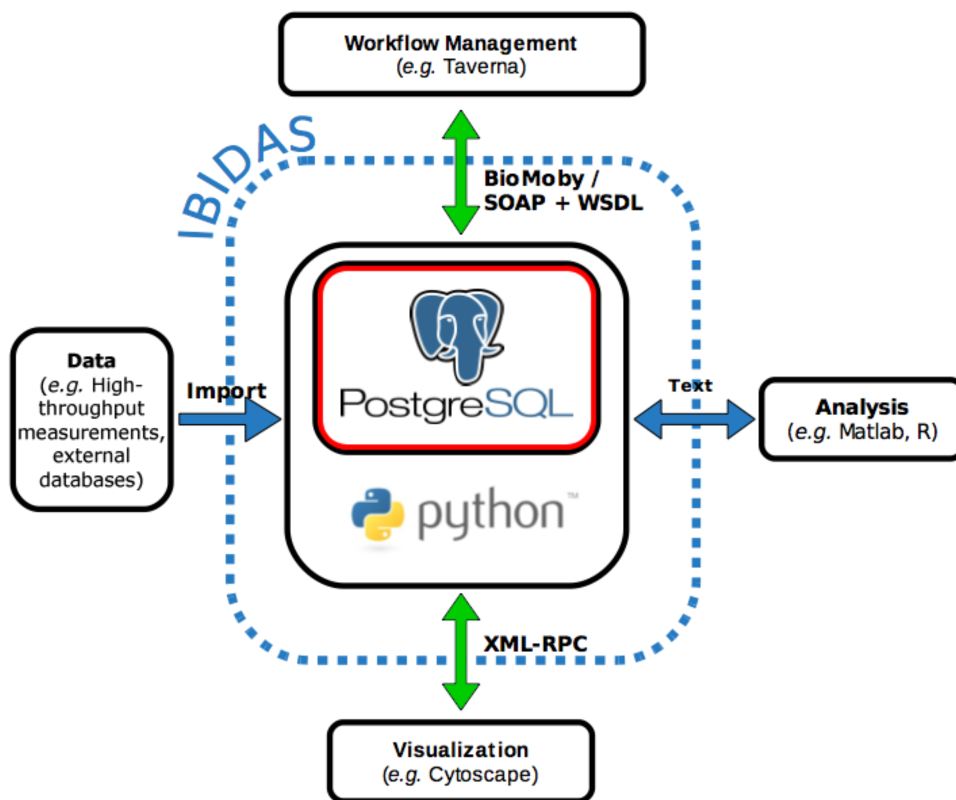


Figure 1.1: Overview of the intended setup of an Ibidas instance. The dashed area is the area Ibidas spans. In this area the layered structure can be seen. Outside the dashed area are some examples of commonly used applications which can interact with the web services layer.

For restricted data, the relationship between the actual data and access rights can only be kept by storing them together. Therefore an authority schema within Ibidas is needed. Such an authority structure should be used to limit user access to publicly available data and data toch which access is granted. For gaining the user's trust, it's important to have a transparent policy regarding critical functionality like the security. Therefore the authorization structure is place visibly outside the program code. In figure 1.1 the authorization structure

is denoted by the red mark located at the PostgreSQL database management system.

Traversing an authorization structure to extract user access rights for data is guaranteed to have implications on the speed with which the requests are executed and the data is returned. To quantify these implications we ran tests before and after application of security and compared the execution times.

Web services are basically small functions which can be executed by a server providing that service. Client applications should have a way to communicate with an Ibidas instance to obtain data and to call server-sided methods. Example Web services for Ibidas are retrieving a particular dataset by some constraints (e.g. all genes on the eleventh chromosome) and calculating a gene set enrichment score. These functions are called as if they are part of the application the user is running. In the background the application connects to the Web server, calls the method there and collects the result. The implementation of such functionalities should work regardless of which protocol is used. The green arrows in figure 1.1 are the Web service communication channels in Ibidas.

One of the most commonly executed operations by researchers is a gene set enrichment analysis based on experimental results. Such an analysis is used to find sets of genes which occur more often than random wihtin a particular other set of genes. This indicates deregulations of gene expression which is interesting to perform further research on. If one's interested in researching the role of transcription factors in this experiment, the transcription factors for these genes need to be obtained. This can be done by requesting the transcription factors for the enriched genes. A popular database consisting many transcription factors is TRANSFAC[21]. To obtain the sequences of the transcription factors of the enriched genes another database call is necessary.

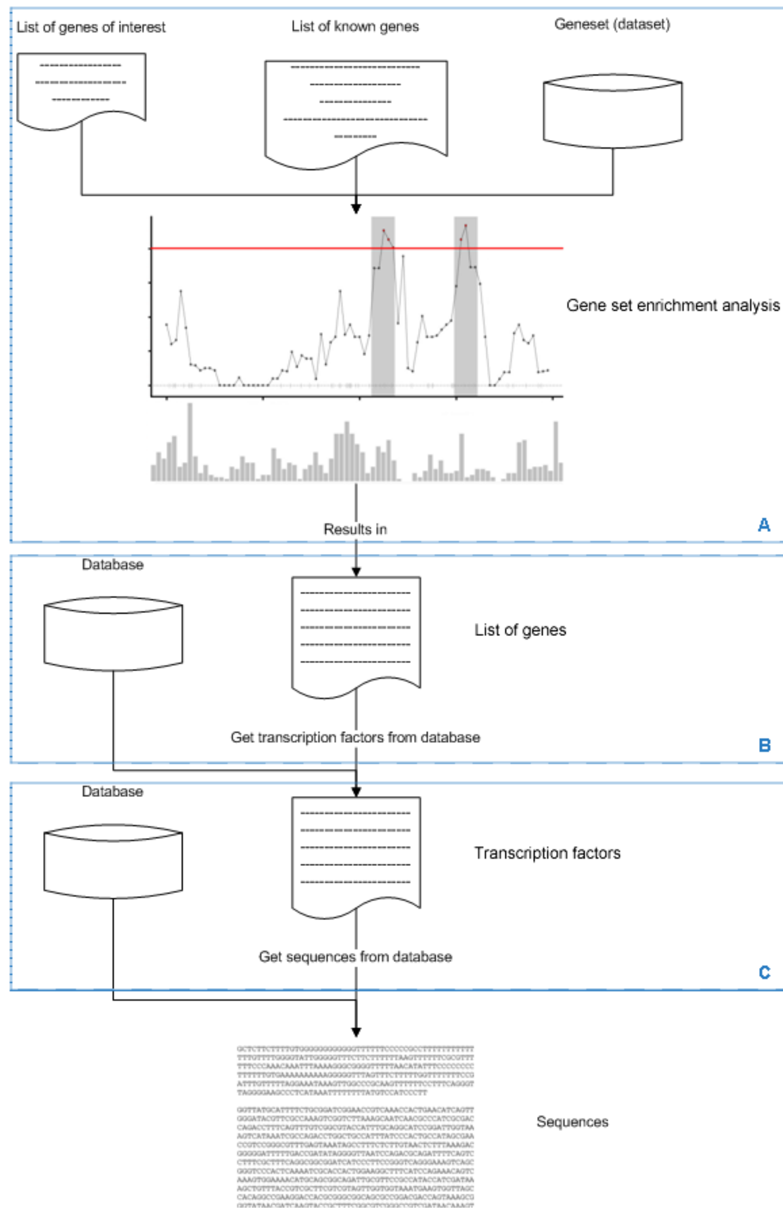


Figure 1.2: An example scenario of a research process. The aim is to retrieve the sequences of the transcription factors which regulate the genes of interest. These genes of interest probably have a similar functionality and / or belong to the same functional pathway. Finding such probable relationships is often done by performing a microarray test which compares expression patterns. For our scenario we take the results of such a test as input. First a gene set enrichment analysis is used to check if the list of genes of interest occurs more often than random within a reference dataset (A). A database is used to obtain the transcription factors of these genes (B). Then a database is used to obtain the sequences of these transcription factors (C).

Chapter 2

Database security

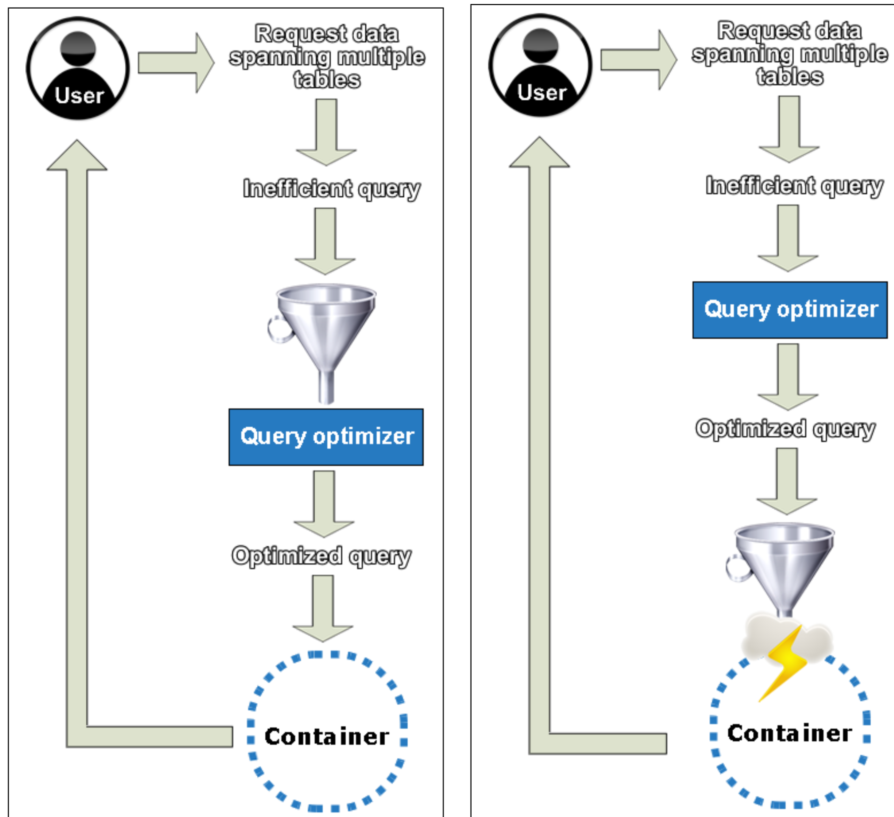
2.1 Virtual private databases

For our personal dataset access feature it is required that there is a way to restrict users in which data they can access. In concrete terms: we need a way of hiding data in database tables from users.

For hiding data which the user is not allowed to see, a filtering mechanism is needed. In this filtering mechanism we apply the access rights on the data: we check the user's access rights on the data requested, filter out the data for which no access is granted and present the resulting data to the user. As requests for data may take a long time to process, it pays off to optimize the call before sending it. By rewriting the call to a form which the system can handle faster, less disk accesses and internal memory are needed, which speeds up the response time. Ibidas has a query optimizer to determine which source should supply which data and based on this, it rewrites the query to an optimal form. Figure 2.1 shows where we can apply a filter. We could apply it before querying the database (figure 2.1(a)), but we can also leave the filtering to the data container (figure 2.1(b)). Filtering data in the code allows us to add our filter before the query optimizer rewrites the call to its final form. However, the filtering mechanism is not visibile to the user. Because our system is new, user distrust is more of an issue than it is with established systems. Users might trust native database functions better than custom code, so we need a way to express strong access controls in a trustworthy way. This could be done either in the database itself or system-wide.

System-wide approaches focus on shared security policy within the OS. This policy holds for all applications running at the server or workstation. SE Postgres[14] is such an approach, using a secure protocol to ensure secure connections to its resources, including Web services. The Secure Document Management System labels documents and assigns access rights to them in a similar way Postgres treats objects. Such a solution is very specific as it's dependent on the OS and the applications. This is in contradiction with the ideas of software developers not to introduce dependencies on OS or specific third-party software configurations.

Performing access control in the database itself means performing security at the level closest to the data itself. This makes the security mechanism straight-



(a) Applying the filter before the query optimizer will rewrite the query is the most optimal situation.

(b) Applying the filter in the data container. The effect of optimized query coming from the query optimizer is partly negated by applying the filter on the optimized query.

Figure 2.1: Possible locations of applying the user access rights (the filtering) to secure the data. The container could be a file, a database, a Web service or any type of combination of such sources.

forward and transparent to the user. Data filtering or hiding can be done on the tables by a form of row-level security or by using views[22].

The Oracle database software[11] provides Fine Grained Access Controls (or also called virtual private databases), a form of row-level security, which we are looking for in PostgreSQL. PGACL[12] is such a plugin for PostgreSQL. Unfortunately row-level access means that access controls need to be defined at every row in every table. In the Ibidas multiple inheritance structure this could not only introduce problems because of inherited access control definitions, but also take longer to evaluate all rows and the amount of tables and rows implies that each row takes extra storage space.

Both SE Postgres and PGACL take care of database security, but they lack functionality and performance required for our system. Securing a whole system seems a good robust approach, but it should fit our aim to make Ibidas available for every operating system. Access control modules derived from Oracle work on the database tables, but are not mature enough (none is a stable first version) and the storage structure has a negative influence on query times, which decreases user-perceived performance. Because both these approaches have too much downsides we chose to use views.

A view is a virtual table built on a stored query. This allows us to define views which incorporate user access rights while storing these rights in a separate table in an efficient way. The security is then transparent to the user while query times are short compared to an approach like PGACL. In the next sections we will see how we defined and implemented these views in Ibidas and PostgreSQL.

2.2 Postgres login roles and usergroups

For storing the user access rights we allow grouping of users (e.g. research departments). Users can be assigned to multiple groups. These groups can be part of a usergroup hierarchy. The ability to view particular data then depends on the usergroup(s) the user belongs to. Postgres itself offers the creation, modification and deletion of users and usergroups. This can be done through the command-line, which Ibidas also uses.

Storing the usergroup in a parent-child relationship between usergroups would be a natural way of storing such data. As different usergroups have different rights, querying such a structure requires a recursive query to traverse the hierarchy to reconstruct all the user's rights, which is time consuming. Therefore we store the hierarchy structure in a 'flat' way: if a usergroup is part of another usergroup, we store a relationship between the user and both usergroups instead of storing the relation between the usergroups and storing a relation between the user and the lowest usergroup. This way we can rerieve all usergroups a user belongs to in one straightforward query.

The advantage of this approach is that querytimes benefit from this as no recursive queries are called. The disadvantages become clear during modification of the memberships. Moving a usergroup to another usergroup (e.g. when restructuring departments) will require all users to be restructured to the new hierarchy. When the hierarchy itself is stored separately, only a couple of rows are affected by the change and the users stay untouched. In the flat storage structure the users need to be reassigned to usergroups. This is much more error prone.

Postgres itself uses a database to store its structure in. We use the tables in this database (system tables) for managing users and usergroups. We use the Postgres system tables `pg_catalog.pg_user` which holds users and `pg_catalog.pg_group` which contains the usergroups. The system table `information_schema.applicable_roles` holds the relation between them and the information about the database rights of the user. An advantage of these native postgres shadow tables is that these users and usergroups are cluster-wide. This means that there's only one copy of `pg_group` and `pg_user` per database cluster, and not per database. This is a useful advantage when more than one database server is required (due to the amount of data, load balancing for performance etc.).

All the data security is taken care of by the database. This means that bypassing the security is limited to attacks on the database. The amount of effort required to succeed completely depends on the system administrator. There are two possibilities for illegaly obtaining the data. The first way is to get into the database and alter the user rights so data which previously wasn't visible through the views will become visible. For doing this root access to the database is required in order to be able to alter the tables.

An attack on the physical Postgres data file is another way to try and get the data. This data is stored in the data directory of PostgreSQL. When installing PostgreSQL, a separate user postgres is created on the OS. This user is given rights to the data folder while no other user can perform any action on this folder. Only if one can switch to this postgres user, the data directory can be read or copied and the security fails. Summarized, data can only be illegaly accessed when either knowing the root password of the Postgres application or the postgres user of the OS.

2.3 Implementation

In the previous section we've seen that storing the usergroup hierarchy in a simple flat structure has less negative effects on the execution speeds of queries than a more complex structure. This section will discuss how the security in Ibidas is implemented by using the capabilities of the PostgreSQL database management system.

When a query for data is done, access control needs to be taken into account. The query will first arrive at the query optimizer which rewrites the query to its most efficient form. When this is done, the optimized query will be appended with the access control part. The basic structure to traverse is shown in figure 2.2(a). This structure is queried as follows: First all sets to which a user has access to are retrieved. From there on the items of these sets are retrieved. In the database this means that users are joined with sets and the result of this is joined with items. The usually large datasets might take a lot of memory space which might be needed for further database operations. By joining sets with users we reduce the data in memory as soon as possible to only applicable data for the user.

In order to execute the authentication on all tables where data is selected from, a view called 'mydatasets_view' returns the datasets accessible by the logged in user. We apply restrictions per source, so there are views for every table having a link to the set table containing these sources. The views restrict

the results such that only results from the mydatasets view are returned. When querying the database, it needs to be told explicitly that data has to be selected from the view, so instead of using the real tables to perform the query on (figure 2.2(a)), we switch the tables for which views exist in the code with the appropriate views. This is situation is depicted in figure 2.2(b). The views or virtual tables already incorporate the access rights and they are only filled with data which the user may access. This is what we defined earlier as the filtering done by the database and it satisfies our design criteria regarding the protection of datasets and doing this in a transparent way as the views can be checked in the database management system.

2.4 Query performance analysis

To analyse the implication the authentication has on the query time we've run test queries to quantize and visualize the difference in query times. We imported the data of the Molecular Signatures Database[19, 8], also known as MSigDB, into Ibidas. Then we constructed queries for reconstructing the data again. For reconstructing the data, in total four table joins are necessary in Ibidas. In total we've created five queries to show the difference in query times over the amount of joins and the data contained in these tables. Both queries selected the same data from the database and had the same results.

2.4.1 Expectations

The PostgreSQL command-line allows for query explanation[4]. A query is explained by means of a query execution plan. The execution plan shows how a query result is obtained, how complex a query is and (possible) bottlenecks can be derived from it.

When using authentication, for every table that is used the authenticating part (the mydatasets_view) needs to be consulted. Because of this extra and repetitive part that is executed, we expect the query time to be larger than when not using authentication.

Every test was run one hundred consecutive times from a cold start to prevent any type of caching having influence on the next series of runs of a query. Table 2.1 gives an overview of the explanations of the queries by Postgres. This gives an indication what to expect of the query times with respect to each other. In this table it can already be seen that although more tables need to be queried, the queries having two joins or more have a lower maximum cost and less rows in case of authentication. This means that Postgres expects the query to return the result faster when views are used (when the authentication is incorporated) than when the tables are directly used. This relates to the rows column which contains the number of rows which need to be considered. It is clear that from two joins on the relationship between authenticated and unauthenticated queries seem inversely proportional in terms of the rows to be considered and the maximum costs related to that. This is because of the PostgreSQL query optimization strategy[10]. Until a particular number of joins, the PostgreSQL query optimizer uses a near-exhaustive search to join tables. For a large number of joins it uses a genetic algorithm, which seems to greatly reduce the amount of rows. This large amount of joins is reached pretty soon by the

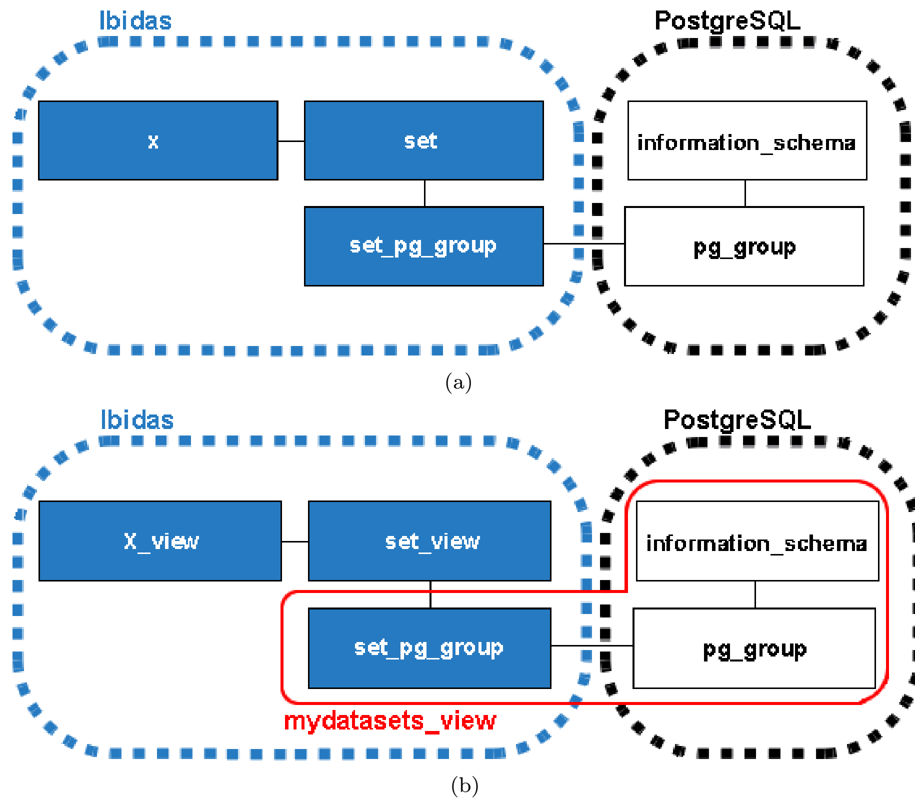


Figure 2.2: The basic structure which plays a role at the security level. In figure (a) every table (denoted by the 'table' X) has a reference to the set table. The sets in this set table are associated with PostgreSQL usergroups (contained in the pg_group table) by the set_pg_group relationship table. The white tables are the native PostgreSQL tables which are used. Note that these are all many to many relations which makes querying such a structure a quickly diverging task which might take a lot of time. However, when authentication is not used, the existing link between the set table and the set_pg_group relationship table is not evaluated. The user is requesting data which is contained in (data)sets. When authentication is used (figure (b)), per set we apply the user access rights. The mydatasets_view is the view responsible for the security. It resolves the datasets a usergroup may access in two joins (of three tables) and it links directly to the set_view. The data is requested from the table_view (depicted as X_view). This view only contains the data originating from datasets which are contained in the mydatasets_view. this view will result in incorporation of the user access rights as the join filters out the rows which are associated with datasets users may access.

Authentication	Joins	Min.cost	Max.cost	Rows
Off	0	0.00	18.39	3
Off	1	18.42	12168.26	410014
Off	2	18.42	363416.46	74062879
Off	3	8117.11	6076762.50	1247398475
Off	4	8117.11	21629360.43	4397936241
On	0	20.77	39.20	3
On	1	60.04	12209.88	410014
On	2	1903.59	52305.12	1110943
On	3	1906.53	18668.71	280663
On	4	2608.21	37761.83	14834

Table 2.1: The results of explaining the queries on the tables (authentication off) and on the views (authentication on) as done by Postgres. Per joined view, an authentication part is included which consists of two joins. The minimal cost is the start-up time before the first row can be returned and the maximum cost is the time it takes to return all rows. The time units are expressed in disk page fetches. The rows column indicates how many table rows are considered during execution of the query.

authenticated query due to the incorporated authentication which brings along two joins per authentication which is done per table.

We run one hundred consecutive runs per query. When looking at the individual query times we expect the Postgres caching mechanism to reduce the query times after the first run. This means that the first run will need more time to return the results. After that, the results will be cached and the query times will then be about the same.

2.4.2 Results

The tests were run on a MacBook Pro with the following specifications:

- Mac OS X 10.5.8 Leopard
- Intel Core2Duo 2,4 Ghz (3 MB L2 cache, FSB 800) CPU
- 4 GB 667 Mhz DDR2 SDRAM memory
- 200 GB SATA harddisk (7200 RPM, 16 MB cache)

Regarding the resulting graphs there are some results confirming our expectations. Caching takes place for all of the authenticated queries and most of the unauthenticated queries (figures 2.3(a), 2.3(d), 2.3(e)). After the caching is done the query times stay about the same. It's interesting to see that the first run for the authenticated query in figure 2.3(d) takes twice as much time as the first run for the authenticated query in figure 2.3(e) while joining three views instead of four (and with that performing an authentication less). As these times are in the order of 45 and 90 seconds, we cannot explain this by environmental factors like background processes waking up. After caching has occurred, the query times in figure 2.3(d) are lower than in figure 2.3(e), like one would expect based on the amount of tables considered during execution.

The results of figure 2.3(b) are interesting as this behavior is not quite expected. Query times for both authenticated queries and unauthenticated queries show a consistent alternating behavior with much resemblance. Although the variance within this alternating behavior is about 0.2 seconds, it is unknown why this actually occurs. We expected the first run to take longer as Postgres will cache the results, but we see for the unauthenticated query that the first run is actually the fastest run. When we ran the queries for figure 2.3(a) and figure 2.3(b) immediately after each other, we observed similar behavior. This was probably due to caching effects from the first query. However, in this case the test was run from a cold start without it being preceeded by earlier tests. This rules out caching effects, but it also rules out the only plausible explanation. A last clear observation from figure 2.3(b) is the large peak around run 73. Regarding the facts that the query time is not doubled, but increased by about 1.2 seconds and the authenticated query time doesn't show such a peak, we assume this peak can be explained by a background process responding to a wake-up call from the operating system while running that particular test. We therefore consider this an outlier. The same motivation holds for run 38 in the unauthenticated query in figure 2.3(c), which we also consider an outlier.

Figure 2.3(c) shows a graph which is completely in line with our expectations. From the figure we can clearly see that the query times for the authenticated query are consistently more than twice as low as for the unauthenticated query and the first run of the authenticated query is not taking longer than the next runs. The explanation by Postgres shown in table 2.1 already predicted that the genetic algorithm would execute the authenticated query in a more efficient way. The explanation however suggested that authenticated queries containing two joins or more would run faster than their unauthenticated version. Based on the Postgres explanation we expected figures 2.3(d) and 2.3(e) to look like figure 2.3(c) instead of looking like 2.3(a). We only observed the predicted efficiency for two joins. After that it looks like the near-exhaustive search is used again while the amount of tables considered in the query increases, but we cannot confirm or deny this.

Overall the tests show a nice performance. Except from the results for the queries containing two joins, we can see from the figures that the average diference in query times is slightly higher when having to perform authentication, but not more than twice as high. Next to that the variance of the query times is very low, which indicates that the response time is very consistent. Although the results show it's faster to not use authentication, the difference between authenticated queries and unauthenticated queries might only be noticeable when performing queries on millions of records spanning all tables in the database.

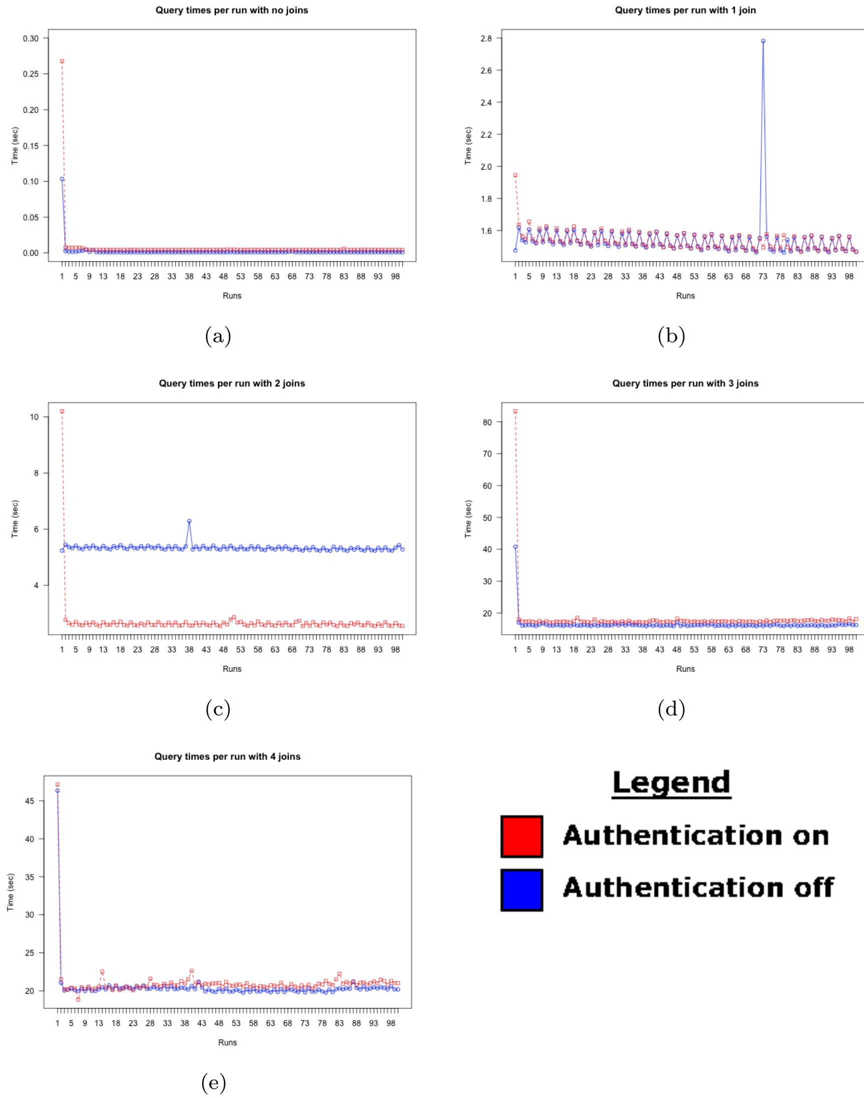


Figure 2.3: The query times of figures (a) and (c) show the expected behavior based on the Postgres explanation. Caching occurs for all queries except the unauthenticated ones in figures (b) and (c). Both the facts that the first run of the unauthenticated query is the fastest instead of the slowest and that the alternating behavior is only present in figure (b) are hard to explain. The large peak on run 73 in figure (b) and run 38 in figure (c) are probably related to a background process in the OS. Figures (d) and (e) resemble much with figure (a) whereas resemblance with figure (c) is expected.

Chapter 3

Web services

Our system is accessible in different ways. Indirect interaction is done through Web services. External applications, like the popular Taverna workbench, can run methods remotely on an Ibidas server as a part of a workflow. For communication with remote services, two transport protocols are commonly used. The first one is XML-RPC, the other one is SOAP. In the next sections we will see how we organized the Web services in Ibidas and how they work.

3.1 Libraries and set up

Basically it doesn't matter if Ibidas is run as a local instance or remotely. Although sharing openly available datasets can be done in a pretty straightforward way, we are also working with possible personalized datasets which require a little bit more attention as we can only share these to authenticated parties. Having said that, an important issue is introduced: remote authentication.

It's not possible (yet) to secure SOAP or XML-RPC communication in such a way that the security of the data is guaranteed after the data has been transferred. Therefore we concentrate on offering unauthenticated and unsecured services, so services which handle openly available data only.

Various applications use various types of communication. The most important applications considered here are Cytoscape and Taverna. The Cytoscape visualization application can communicate through XML-RPC with external data sources to obtain data and visualize it. The Taverna workbench can import WSDL[2] files from remote servers to discover SOAP services. Figure 3.1 shows the Ibidas Web service workings. The AtomicServices library is the class which executes methods on the Ibidas command-line. These methods are called by the server classes. These server classes are protocol-specific and translate the Web service calls to AtomicServices method calls. Because the translation occurs at the server classes the actual executing class, AtomicServices, is protocol-independent. Because of that, adding functionality to this class means that the functionality is added for all supported protocols at the same time.

The SOAP communication is slightly different than XML-RPC. WSDL files are files describing server properties on how a SOAP service should be called, which parameters should be provided, of which type these parameters should be and what you'll get back from the service and what the data type of the result

is.

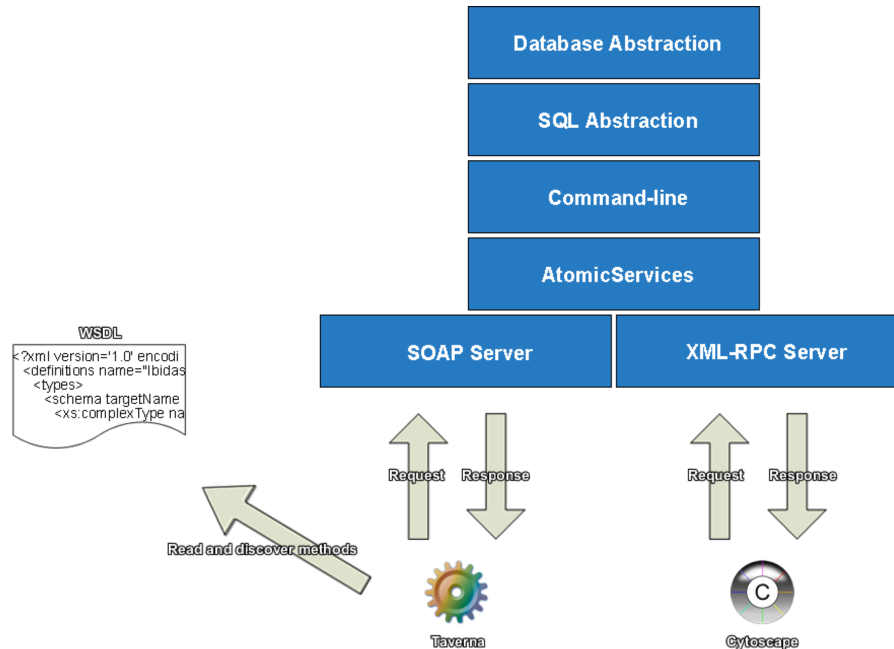


Figure 3.1: Overview of the layers which are used with Web service access. The SOAP and XML-RPC servers provide the interface to the applications and translate (wrap and unwrap) the communication messages from and to the AtomicServices class. Both servers listen on a different particular server port to which the users connect. The AtomicServices class receives the calls from the servers and processes them. It defines the available methods and acts on the command-line interface like other Ibidas classes.

For XML-RPC, there is a built-in module in Python to create a simple server (in just a couple of lines of code) which can listen on a server port for requests and handle them. For SOAP there is no such module built into Python. Therefore we've chosen to use the Soaplib library[18] to handle SOAP messages. Soaplib offers CherryPy as a Web server, so this is installed with Soaplib as well. A nice feature of Soaplib is that it can automatically create a WSDL file based on the services that are offered. In our case it means that a WSDL file will be created based on the methods defined in the AtomicServices.

3.2 Implementation

There are multiple transport protocols to transfer the same data. In a neat object-oriented structure this means that only connecting interfaces for these protocols need to be implemented whereas the functional layers can be the same. Currently Ibidas has an XML-RPC server, IbidasXMLRPCLight, and a SOAP server, IbidasSOAP. These servers are the access points for clients and client applications to connect to Ibidas. This section will treat the implementation of the SOAP server, the SOAP serializers and the executing AtomicServices class.

3.2.1 AtomicServices

The implementation of the AtomicServices class doesn't differ from regular Python classes. It maintains a database connection and it defines some functions which convert Ibidas objects containing data from the database to a representation which can be sent in a SOAP message. This is called serialization. Our SOAP serializers are discussed in the SOAP serializers section.

Data representing objects within Ibidas are called container objects. These container objects contain actual data from the database. The contents of such an object are variable, dependent on which data is requested from the database. This can range from one single item to all contents of the whole database. Container objects are basically a database in object form: these objects can be queried in a similar way a database is queried.

As mentioned earlier Soaplib will generate a WSDL file based on the contents of the AtomicServices class. In order to be able to do this, Soaplib must recognize the methods which users can call as being SOAP methods. This is done by applying decorators[5] to the methods intended for remote calling. Decorating a method doesn't alter the workings of the methods, but rather adds a description for Soaplib to the method of which type the incoming parameters need to be and what the data type of the result is. As this is what WSDL is all about, Soaplib gathers the method names and the corresponding decorators and transforms this information to a WSDL representation. Although these methods are also used by the XML-RPC service, the decorators are only used for the SOAP service and do not affect the XML-RPC service.

3.2.2 SOAP server

For handling requests, a Web Server Gateway Interface is needed. Python supplies such a module, wsgiref, and Soaplib comes with CherryPy. Both WSGI modules are sufficient for our SOAP server. We chose to use wsgiref, just because it comes with the Python distribution and so it's more likely to stay compatible with our Python code.

The XML-RPC server implementation allows for registering instances of classes which handle incoming requests. The WSGI server doesn't allow this and Soaplib requires the SOAP methods to be in the server class to be able to create a WSDL file and execute the defined methods. Therefore the executing class, AtomicServices, is written in such a way that it can be registered as executing class as well as it can provide methods to a server class. By letting the SOAP server inherit both the WSGI application and the AtomicServices class Soaplib can access the SOAP methods while AtomicServices is kept in a separate class so the XML-RPC can still access it.

3.2.3 SOAP serializers

The Web service methods return results which need to be put in a SOAP message for sending it to the client. For this purpose there are serializers. Object serializer classes declare the structure of an object in terms of their attributes with their type. This type can be a lower-order serializer again. Soaplib provides a couple of primitive serializers for basic data types like strings, booleans,

integers and arrays. As more complex structures are required, serializers can be constructed which are composed of other serializers.

As stated explained in the AtomicServices section, Ibidas uses container objects which can contain any kind of information. Such a container can be the result of a Web service method executed by AtomicServices. It should be serialized to pack it into a SOAP message and to send the message back to the client, i.e. a serializer for the container object should be a variable serializer which depends on the data in the container. However, when starting a SOAP server, a WSDL file is published which declares the structure of the available objects. This means that before any transaction is done, the object structure, and with that the object serializers, are finalized. Because of this it's not possible to introduce serializers depending on the actual data in a container object. As a result of that a static serializer is created which has the ability to store all possible data which a container object may contain. This serializer is called the IbidasMessage serializer.

In the worst case, the whole Ibidas database is contained in a container object. The serializer for a container object should thus be a serializer for the whole database. This introduces a dependency: the serializer for a container depends on the database structure (the database schema). A schematic overview of the IbidasMessage serializer is provided in figure 3.2. The database is divided into tables. These tables have fields, which are of a particular primitive type. For these primitive types Soaplib offers serializers already. Based on those serializers we construct a complex serializer for each table (which depends on the table structure). Eventually the IbidasMessage serializer is a complex serializer built of all table serializers. As the database schema may be changed these serializers are not provided with Ibidas, but they are generated upon installing.

The generating script contains a mapping of database fieldtypes to Soaplib primitive serializers. For each table it generates a serializer based on the current fields and their types. In the end it will construct the IbidasMessage based on the serializers it has created previously. Because the serializers are static from that moment on, a change in the database schema means that it's necessary to regenerate the serializers because of the dependencies between the database schema and the serializers.

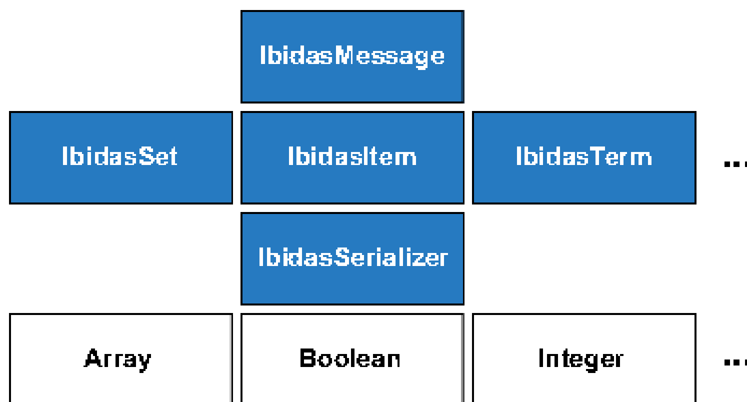


Figure 3.2: Overview of the SOAP serializers which are present in the Ibidas system. The white boxes contain the primitive types which are supplied by Soaplib. The IbidasSerializer is a generic serializer for Ibidas objects. It can contain all kinds of combinations of these primitive serializers. All serializers depicted on top of the IbidasSerializer extend this generic serializer. These serializers are table specific and being generated based on the database schema. Finally the IbidasMessage is the generic serializer which contains all specific IbidasSerializers. The IbidasMessage is a serializer for container objects.

Chapter 4

Using Ibidas in workflows

To show an example application of Ibidas, the scenario of figure 1.2 is executed like a researcher would do by using the Taverna workbench. The scenario starts when the researcher has test results which need to be processed. These test results could, for example, be gene expression data from a microarray test. Such a microarray test results in a list of genes with an expression value which is measured in the amount of mRNA bound to the microarray spot.

The researcher is interested if a set of genes satisfying a particular constraint (e.g. genes with an expression level above a certain threshold) considered in the microarray test occurs more often than random within a specific group of genes (e.g. breast cancer related genes). This type of testing is called gene set enrichment analysis. An overview is provided in figure 4.1. The result of a gene set enrichment test is a p-value. If the p-value is less than a particular (user-defined) threshold, the selected gene set is correlated with the reference gene set and this set is considered enriched.

When the selected set of differentially expressed genes is actually enriched, it indicates that this gene set contains genes that are possibly related. Deregulated expression might have to do with the transcription factors binding to the gene's binding sites located somewhere in the upstream region of the gene. Therefore it's interesting to get all transcription factors for those genes. Afterwards the sequences of the transcription factors can be collected and, for example, compared to see if there are similar specific subsequences which could be indicating of a cause of different behavior of the transcription factors and with that deregulated gene expression.

4.1 Example Taverna workflow

For executing the scenario of figure 1.2 a Taverna workflow was created which uses the Ibidas Web services defined in the AtomicServices class. The complete workflow is given in figure 4.2. Three of the four parameters determine the input of the gene set enrichment analysis, which is executed by the *return_enriched_geneset* service. The category_id refers to an MSigDB category which we use here to select the reference gene set. The other two parameters, 'selected_list' and 'microarray_genes', come from the microarray test and contain the gene set of interest and all the genes present at the microarray respectively.

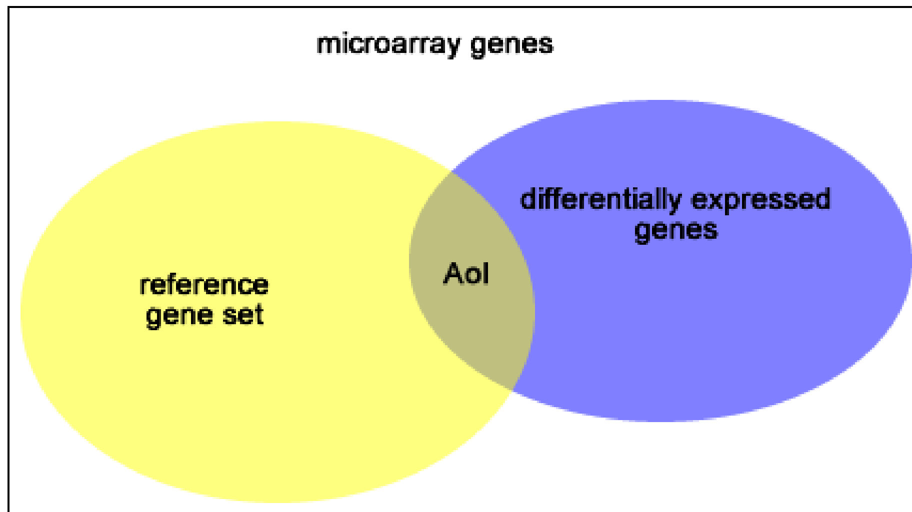


Figure 4.1: Overview of the gene set enrichment analysis. The box spans all microarray probes. The reference gene set is a functional category of genes against which the set of differentially expressed genes is tested. The resulting p-value indicates if the overlap (The Area of Interest, or AoI) between the reference gene set and the differentially expressed genes can occur by pure chance.

When the calculated p-value is below the threshold (which is the fourth parameter), then the gene set is considered enriched and it will be passed on to the next service (*get_accession_names*). If not, the result is an empty set and the workflow result is also empty.

At the *get_transcription_factors_from_genes* service in figure 4.2 TRANSFAC is used to obtain the transcription factors of the genes of the enriched set. The tables in TRANSFAC which are used to do this are *gene*, *factor* and *site*. The accession names are translated to TRANSFAC gene identifiers. After that the binding sites for every gene can be retrieved from the *site* table. The *factor* table is then used to obtain the transcription factors binding to each of these binding sites. The result is again returned to Taverna.

Obtaining the sequence data from TRANSFAC is straightforward. These sequences are stored with the transcription factors in the *factor* table. As the *get_transcription_factors_from_genes* Web service already returned the transcription factors for the genes of interest, this list is used by the *get_TRANSFAC_sequences* service for selecting the sequences from the table and returning them.

In the end, this simple workflow shows how to obtain the sequences of transcription factors of genes in which a researcher is particularly interested. With the Web services provided by Ibidas this can be done in just a couple of steps. The services used in this scenario are just a couple of the services offered by Ibidas. For instance, when not working with sequences, but just identifiers, there is a Web service which can convert these identifiers to their equivalent identifiers for other databases or Web services. All data which is being returned can easily be splitted or converted to an output which is suitable either as input for any other service (e.g. a Web service or an application etc.) or as resulting

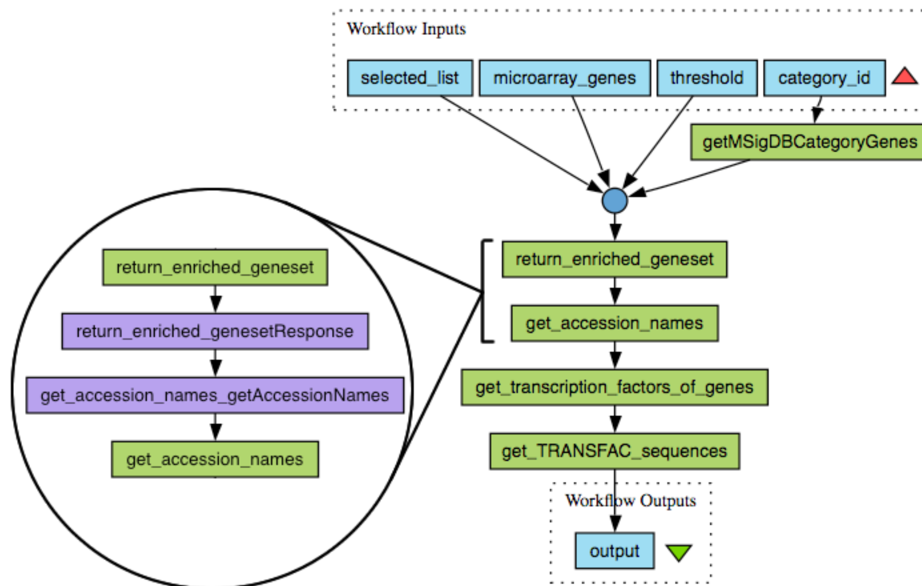


Figure 4.2: The workflow for the scenario as drawn by the Taverna workbench. The green blocks are the actual Web service calls We left out the XML-splitters (as can be seen in the highlight) which are required for input and output conversion between services. The 'selected_list' parameter contains the genes of interest (which satisfy a particular constraint). The 'microarray_genes' parameter contains the world, which are all genes considered in the microarray test. The threshold is used to determine if the selected_list occurs more often than random within the reference dataset. The reference set is, in our case, an MSigDB category defined by the category identifier 'category_id'. Based on these user-defined input paramters (depicted on top of the image) this workflow detects if a set of genes of interest occurs more often than random within another gene set. If so, it collects the transcription factors which regulate transcription of these genes and returns the sequences of these transcription factors. The workflow output 'output' will contain the results after executing the workflow.

Input	Value
selected_list	[CDKN2B, CDK4, CDK6, ZBTB17, CDKN2A, CCND1]
microarray_genes	[EP300, CCNG1, CABP1, BMP10, CDKN2B, CDK4, CDK6, ZBTB17, SAA2, ANG, CDKN2A, CCND1]
threshold	0.05
category_id	c1:200

Table 4.1: These inputs correspond to the workflow inputs in figure 4.2. Lists are placed between square brackets. These input values can be used in our scenario to reconstruct our test results.

output.

4.2 Example Taverna workflow execution

For testing the workflow of figure 4.2 we chose a number of related genes and added some arbitrary chosen genes to be the microarray_genes. We selected a couple of these related genes to be the selected_list input. A commonly used threshold is 5%, so we adapted this and took 0.05 to be our threshold. As our reference dataset, we chose to use category c1:200 of MSigDB. This category contains genes in the cytogenetic band of chr9p21. An overview of the parameters and their contents is given in table 4.1.

CDKN2B, CDKN2A, CDK4, CDK6, ZBTB17, CCND1 and CCNG1 are genes involved with the G1 phase in the cell cycle. This phase is about cell growth and it is characterized by a great amount of protein synthesis and a high metabolic rate in the cell. Because of this rate and the large amount of proteins to be created, it's a phase which is error prone and sensitive to changes in the DNA of the genes encoding for these proteins. We chose six of these genes to be the input of 'selected_list'.

The gene set enrichment test done by *return_enriched_geneset* results in a p-value of 5.81472181842e-05, which is below the threshold of 0.05 we've set. Therefore the accession names are collected by the *get_accession_names* service. This is done because the accessions which were used as input were translated to Ibidas item identifiers for the enrichment test. After conversion the *get_transcription_factors_of_genes* services consults TRANSFAC and obtains the transcription factors for the genes of the selected_list parameter. These transcription factors include the oncogenes CBFA2 and MYC and some other known transcription factors like SP1 which are known to regulate these genes. After that the *get_TRANSFAC_sequences* service uses these transcription factor accessions to obtain their sequences. We obtained 113 FASTA sequences in total of these transcription factors.

The workflow SCUFL file used for this test can be found in the examples folder in the subversion repository of Ibidas, located at
https://gforge.nbic.nl/svn/ibidas.

Chapter 5

Discussion and future work

This chapter will first discuss some issues which arised during the development of the features discussed in this thesis. Afterwards some future work related to these will be proposed to further improve Ibidas.

5.1 Discussion

5.1.1 Usergroup resolution

There are various ways of implementing a hierarchy in the database. As already suggested a parent-child structure is seen often as the most natural way of storing a hierarchy. When running Ibidas for multiple users (in a mode where authentication is required), resolving the usergroups to obtain all the user's rights is an operation which is done with every database request. Changing or inserting data in this usergroup requires many key changes in this table. Adding a usergroup *a* to another usergroup *b* requires getting all user rights from usergroup *b* (which can also have subgroups) and adding them one by one to all users of usergroup *a*. This can be quite a complicated operation, which fortunately doesn't occur often. Therefore resolving user rights should be a one-pass operation to keep the query times low. Although recursive queries are not an option in such structures, we considered alternatives.

One of them was suggested at the MySQL dev site[3] which uses pointers for preorder traversal through a hierarchy. Searching this structure might be fast, but inserting a node which should not be placed at the end of the tree requires shifting of indices to maintain the preorder advantage. This requires too many unnecessary key changes in the database which also brings along unnecessary risks of errors and possibility of data corruption.

The Oracle database allows for travelling through a hierarchy by using a 'connect by' clause which allows for parent-child relations within a table in such a way that the child is a parent in its turn of another row. As a connect by clause is Oracle specific and its not part of the official SQL standard, PostgreSQL will not adapt such a clause. However, the WITH-clause[16] should work more or less in the same way as connect by in Oracle. Its part of SQL:1999 and later and is present in PostgreSQL 8.4. This version was released the 1st of July 2009. By using the WITH-clause, the usergroup hierarchy will be stored in the more natural parent-child structure. Insertion or modification of this structure

currently is an intensive operation which will be made much easier by adopting the with clause. A comparison should be made between the execution speed of user rights resolution performed the way we currently do it and the speed when using a with clause. If the resolution is as fast or faster, the with clause should be adapted. Although this clause is particularly interesting in recursive queries like our usergroup hierarchy, for now we want to also support Postgres databases prior to version 8.4.

5.1.2 Joining order

When performing our filter we chose to first create a list of sets which the user has access to. From there on this view, the mydatasets view, is used for creating the final results. This way the filter is performed at the beginning of the query execution. Another way is to not look at users when executing the query and just retrieve the data. The join with users will be done later when the sets are joined with the items. This way the filter is applied at the end of the query execution. This is likely to impose larger memory usage than the previous way as more records (and larger results) are put into memory buffers. The queries are less complex though, which might save processor load compared to the query where the sets are rst restricted to user. Perhaps the extra memory usage could be minimized by having a smart way of caching of intermediate results. Basically it seems that the query is a trade-off between memory usage and CPU load.

5.2 Future work

This section will introduce some thoughts on future work to improve Ibidas. The ideas range from about reducing computation time to increasing popularity among researchers. They are based on experiences during implementation of the discussed features and trends in the development of the WWW.

5.2.1 RESTful transferring

We have implemented the most popular Web service protocols XML-RPC and SOAP. However, these protocols are 'active' protocols which send data in packages to a client. An increasing popular approach is REST[15] (so-called RESTful approaches). According to REST, everything is considered a resource which has a representation. These resources are identified through a URI. They can be called through a URL e.g. `/data/msigdb/c1` which would show all data contained in the first category (c1) of MSigDB.

The main benefit of REST over SOAP and XML-RPC is the way of returning data. REST uses the HTTP or HTTPS protocol without having any type of vocabulary (like the XML tags) required to transfer data which does not have to be in a pre-defined specific structure (like SOAP). Because of this, the envelope (SOAP) and tags (XML-RPC and SOAP) are not necessary and this means that there's less overhead when transferring the data to the client. The larger the dataset to be transferred the larger the advantage of REST becomes. As Ibidas contains a lot of data we expect that a RESTful approach increases the data transferring speed of our Web services significantly.

5.2.2 Authenticated Web services

Currently Web services are only offered for datasets which do not have limited access. Private datasets cannot yet be transferred to be used in external applications. To realize this, authentication should be possible in Web service sessions. Currently such an initiative is being implemented by the people at SARA[6]. They are developing a way to use login credentials (from their grid) with SOAP messages in Taverna.

For us to be able to utilize this a REST interface should be created as already suggested earlier. Next to that the login credentials for the Sara grid should be incorporated somewhere in Ibidas to use the REST authentication with Taverna.

5.2.3 Ibidas sessions

In our scenario we've seen that when calling an Ibidas Web service, we get back the results of that service. When using this data as input for another Ibidas Web service this is not the optimal way as we are retrieving the data and sending it back again. Therefore when using multiple Ibidas Web services after each other, a session should be created. This allows for keeping the output of a method at the Ibidas instance and using it directly as an input for the next service. Instead of transferring data, a pointer should be generated which identifies either the physical stored data or the query at the Ibidas instance. This pointer should be the parameter which is transferred between client and server. This saves a lot of time and unnecessary data transferring. A session should start when an Ibidas Web service is called and it should stop when the last Ibidas Web service of that series is executed. Only when the session stops, data is actually returned to the client.

Although this saves time for executing a workflow, it increases the amount of data present at the Ibidas instance if data resulting from a query is stored. For computations, extra processor load is implied. These are factors which should be taken into account when creating such a feature.

5.2.4 JSON

JSON[7] stands for Javascript Object Notation. It is an alternative data interchange format for XML supported by practically all programming languages. It has a kind of flat representation which makes it light-weight. Python 2.6 and later includes a JSON encoding and decoding module by default. JSON is gaining more and more popularity on the Web for presenting data in an easy way. From a web page, Web services can also be called. A Web interface can be built which offers all possible offline Ibidas operations through Web services and visualizes the JSON data. This might convince more users to use and support Ibidas and offers them to test it out online before using our Web services in their applications.

Bibliography

[1] Cytoscape Consortium. http://www.cytoscape.org/.

[2] W3C consortium WSDL specification. http://www.w3.org/TR/wsdl.

[3] MySQL dev. http://dev.mysql.com/tech-resources/articles/hierarchical-data.html.

[4] PostgreSQL explain. http://www.postgresql.org/docs/8.3/static/sql-explain.html.

[5] Python Software Foundation. http://www.python.org/dev/peps/pep-0318/.

[6] SARA grid computing. http://www.sara.nl/.

[7] JSON. http://json.org/.

[8] MSigDB. http://www.broadinstitute.org/gsea/msigdb/.

[9] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[10] PostgreSQL Query Optimizer. http://www.postgresql.org/docs/8.3/interactive/geqo.html.

[11] Oracle. http://www.oracle.com.

[12] PGACL. http://code.google.com/p/pgacl.

[13] PostgreSQL. http://www.postgresql.org.

[14] SE PostgreSQL. http://code.google.com/p/sepgsql.

[15] REST. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[16] PostgreSQL RFP. http://archives.postgresql.org/pgsql-hackers/2008-02/msg00642.php.

[17] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: A software environment for integrated models of biomolecular interaction networks. *Genome Research*, 13(11):2498–2504, 2003.

[18] Soaplib. http://trac.optio.webfactional.com/.

[19] Aravind Subramaniana, Pablo Tamayoa, Vamsi K. Moothaa, Sayan Mukherjeed, Benjamin L. Eberta, Michael A. Gillettea, Amanda Paulovichg, Scott L. Pomeroyh, Todd R. Goluba, Eric S. Landera, and Jill P. Mesirova. Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles. *PNAS*, 102(43):15545–15550, 2005.

[20] Taverna. `http://taverna.sourceforge.net/`.

[21] TRANSFAC. `http://www.gene-regulation.com/pub/databases.html`.

[22] PostgreSQL views. `http://www.postgresql.org/docs/current/interactive/tutorial-views.html`.