GMB: An Efficient Query Processor for Biological Data

Kamal Taha[a]* and Ramez Elmasri[b]

[a] Khalifa University of Science, Technology & Research, Box 127788, Abu Dhabi, UAE

[b] The University of Texas at Arlington, Box 19015, Arlington, Texas 76019, USA

Summary

Bioinformatics applications manage complex biological data stored into distributed and often heterogeneous databases and require large computing power. These databases are too big and complicated to be rapidly queried every time a user submits a query, due to the overhead involved in decomposing the queries, sending the decomposed queries to remote databases, and composing the results. There is also considerable communication costs involved. This study addresses the mentioned problems in Grid-based environment for bioinformatics. We propose a Grid middleware called GMB that alleviates these problems by caching the results of Frequently Used Queries (FUQ). Queries are classified based on their *types* and *frequencies*. FUQ are answered from the middleware, which improves their response time. GMB acts as a gateway to TeraGrid Grid: it resides between users' applications and TeraGrid Grid. We evaluate GMB experimentally.

1 Introduction

The recent availability of high throughput technologies has resulted in a large amount of biological data. Currently, there are more than 1000 biological databases publicly available. These databases contain huge amount of heterogeneous information about genes, proteins and genetic diseases resulted from extensive biological-related research projects. These public databases (e.g., GenBank, PIR, SwissProt, PDB, Pathway, Prosite, ENZYME, EPD, REBASE, Pfam, and Blocks) are maintained by different research centers and institutions[1], which causes barriers to the integration of the databases. Moreover, these distributed databases are usually too big and complex. They vary in the type of the stored data, the data format, and access methods. The user must decide which data source to access and in which order, how to retrieve the data and how to combine the results. In addition, the output of one service may have to be manually manipulated in order to be used as input for another service. Also, data from two or more services may have to be semantically integrated by resolving heterogeneities in the output of the various services. These factors make the execution of tasks involving more than one service time consuming, and the task of retrieving data requires a great deal of effort and expertise on the part of the user.

Grid community believes that these problems can be alleviated using Grid's distributed high performance computing and collaboration application [2, 5, 21, 31]. However, the response time of a query is still high. When a large number of sequences are submitted to run through a computationally intensive tool, for example running a BLAST tool for NCBI's non-redundant protein sequences containing over 3 millions sequences, it usually takes a few days to get the results for all the sequences.

* To whom correspondence should be addressed. Email: kamal.taha@kustar.ac.ae

[1] Such as the US National Center for Biotechnology Information (NCBI), the European Bioinformatics Institute (EBI), and the Swiss Institute of Bioinformatics (SIB).

Frequently Used Queries (FUQ) in bioinformatics applications should be given special treatment. This is because there are certain queries submitted by scientists and researchers over and over again. The predicate conditions' values and/or operators of these queries may differ but the queries' multi *computational steps* remain the same. Consider for example the query "*Find all genes in the human genome that have TTGGACAGGATCCGA followed by GCCGCG within 40 symbols in a 4000 symbol stretch upstream of the gene*". A scientist or a researcher may submit the same query multiple times, but changes only the genome sequences and/or the number of symbol stretch. We call a query without predicate values and operators a *Query Type*. For example, if we strip the predicates' values from the query stated above, the result will be the Query Type: "*Find all genes in the human genome that have …. followed by … within … symbols in a …. symbol stretch upstream of the gene*". That is, a Query Type is a query composed only of computational steps (without predicate condition operators and values). A Query Type represents all queries that have the same sequence of computational steps regardless of their predicate conditions' values and operators. Each Query Type is assigned a numeric ID. For example, Q_i denotes Query Type *i*. All queries that have the same type *i* will be assigned the ID Q_i. Fig. 1 shows a sample of nine Query Types.

We propose in this study a query processor called GMB (Grid Middleware for Bioinformatics). GMB can be considered as a type of middleware running at a central site. It acts as a gateway to TeraGrid Grid [29]: it resides between users' applications and TeraGrid Grid. GMB classifies queries per their types, ranks the query types based on their frequencies, and answers the frequently used ones from results cached in the middleware. The results of queries whose types are classified as FUQ types will be cached at the middleware. Thereafter, each FUQ of these types will be answered from the middleware rather than from remote sites, which improves the query's response time. GMB has been implemented to search and access the TeraGrid-ORNL [30] data resources. The following summarizes the advantages of the GMB framework to biologists and researchers: (1) it greatly improves the *response time* of biological queries, (2) it solves the problems caused by the delaying of results due to network traffic and slow servers at remote sites, (3) it enables caching of results at the middleware, which minimizes the problems of heterogeneity, enormity, and complexity of distributed biological databases, (4) it enables access to TeraGrid Grid, which *integrates* high-performance computers, data resources, and high-end experimental facilities around the US (*it integrates the resources of eleven partner sites*), (5) it enables biologists and researchers to query a very large number of *biological-specific* databases, and (6) it provides a framework that can work with *any* biological database.

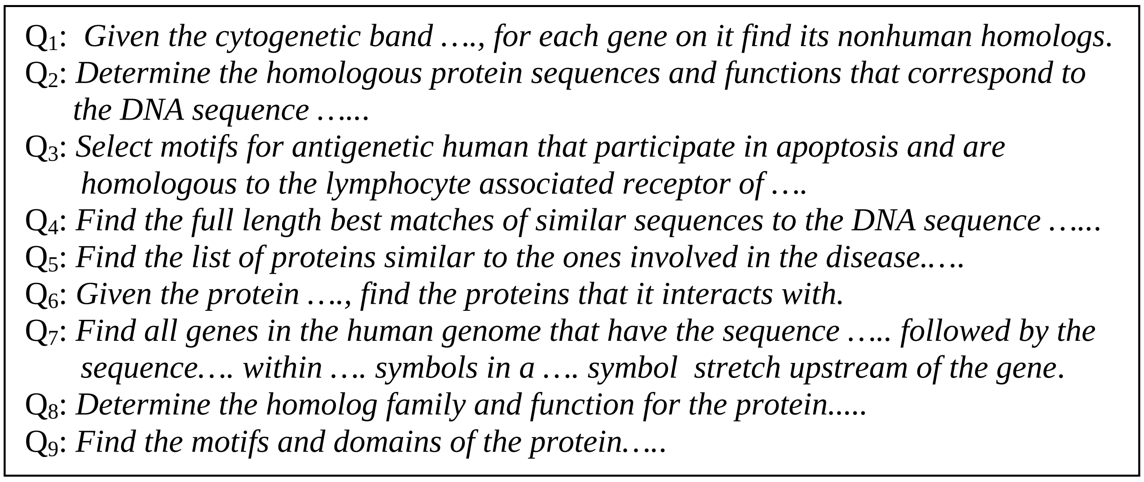

Q_1: *Given the cytogenetic band …., for each gene on it find its nonhuman homologs.*
Q_2: *Determine the homologous protein sequences and functions that correspond to the DNA sequence …...*
Q_3: *Select motifs for antigenetic human that participate in apoptosis and are homologous to the lymphocyte associated receptor of ….*
Q_4: *Find the full length best matches of similar sequences to the DNA sequence …...*
Q_5: *Find the list of proteins similar to the ones involved in the disease.….*
Q_6: *Given the protein …., find the proteins that it interacts with.*
Q_7: *Find all genes in the human genome that have the sequence ….. followed by the sequence…. within …. symbols in a …. symbol stretch upstream of the gene.*
Q_8: *Determine the homolog family and function for the protein.....*
Q_9: *Find the motifs and domains of the protein…..*

Fig. 1: A sample of nine Query Types

The paper proceeds as follows. Section 2 presents related work. In section 3, we describe how Query Types are defined and represented graphically. In section 4, we describe how a query's type is identified. In section 5 we describe how FUQ are answered from cached results. We investigate in section 6 the determination of ideal number of FUQ that guarantee acceptable hit rate and performance. In section 7, we present the system architecture. We present our experimental results in section 8 and conclusions in section 9.

2 Related Work

Science communities [10, 14] provide access to archival data to a distributed community using middleware systems such as SRM (Storage Resource Manager) [13, 15, 4] and Earth System Grid. SRM provides an interface between caches and mass storage systems by queuing requests. There are other techniques for accelerate scientific data accesses by offering dataset caching (e.g., IBP [18], DPSS [6], HPSS-disk caches [26]). The key differences between our proposed techniques and these techniques are that the later support only entire dataset caching and they do not provide facilities that improve the response time of FUQ.

There are three different approaches to database integration: information linkage, data translation, and query translation [28]. Information linkage establishes relationships between data sources using cross-references. This eases the navigation over different data sources. But, the information is not actually integrated, which is the main drawback of this approach. This approach is used in public biological databases such as Prosite and PDB. In the data translation approach, all data from the databases are integrated in a central data repository. Data from different sources are translated to a unified global conceptual schema and stored at the central repository. In the query translation method, the user queries are fragmented into different sub-queries. Then, the sub-queries are executed by mediators. A number of works (e.g., TAMBIS [24], TSIMMIS [19], OntoFusion [27], BACIIS) studied the integration of heterogeneous databases. TAMBIS integrates life science databases that consist of protein and nucleic acid data sources. BACIIS integrates all possible life science data sources. TSIMMIS translates queries and information, extracts data from web sites, combines information from several sources, and allows browsing of data sources over the Web. TSIMMIS is a tool for building integration systems and is not itself an integration system. TSIMMIS utilizes a distributed CORBA protocol for submitting queries and for returning results [16]. This distributed protocol returns partial results to the client and requires server side support.

The abovementioned systems include the Distributed Information Systems Control World [20]. This system is a middleware for integrating distributed applications across the networks [17]. These systems provide facilities to access information from multiple resources. Using these systems, a biologist can search multiple biological databases. The results of each search process are the composed result instances obtained from different remote sites. The key difference between our proposed system and these systems is that the later only integrates heterogeneous databases and do not provide accelerated access to the scientific data.

3 Querying Biological Data and Representing its Query Types

3.1 Querying Biological Data

To query biological data using GMB, a user needs to issue a *loosely structured query*. Loosely structured querying allows combining some structural constraints within a keyword query by specifying the context where a search term should appear (combining keywords and search names). It does not require the user to learn a query language or to be aware of the structure of the underlying data. GMB accepts a loosely structured query that has the form $Q(s_{k1}$ = "k_1",

…, s_{kn} = "k_n", R_1?, …, R_n?), where k_i denotes a keyword of the information to be searched, s_{ki} denotes the name of the search containing the keyword k_i, and R_i denotes the name of a requested (return) result. Consider for example the query "*select motifs for antigenetic human that participate in apoptosis and are homologous to the lymphocyte associated receptor of death (also known as lard)*" (recall Query Type Q_3 in Fig. 1). This query can be represented in a loosely structured form as follows: *Q(function = "human antigen", function = "apoptosis", protein homolog = "lard", motif?).*

3.2 Representing Query Types of Biological Data

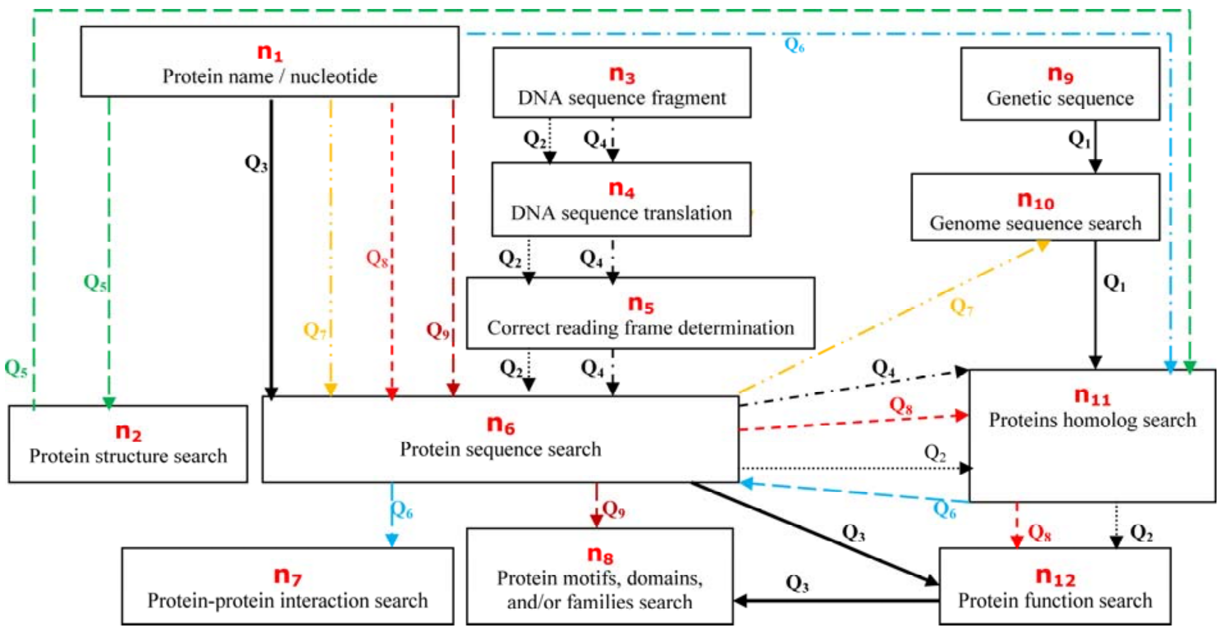
We describe in this section how Query Types are defined and represented graphically. First, we need to identify all Biomolecules' search classifications. Towards this, we represent Biomolecules and their search classifications ontologically. Second, we construct a graph called a *Global Query Graph* (GQG) based on the search classifications described above. Each node n_i in the graph represents one of the computational steps used for processing a query. Each edge represents a transitional sequence from one computational step to another for processing a query. The edges are created by marking the paths between computational steps that Query Types go through for processing. For example, if a Query Type is processed through the computational steps' sequence n_1, n_2, n_3, the edges $n_1 \rightarrow n_2 \rightarrow n_3$ represent this Query Type and its path in the GQG. We now formalize the GQG concept in Definition 1:

> *Definition 1, Global Query Graph (GQG): A GQG is a pair of sets, GQG = (T_S , E). T_S is a finite set of nodes representing computational steps for processing queries. That is, T_S = {T_i | T_i is a computational steps and $1 \leq i \leq |T_S|$). E is the set of edges representing paths between the computational steps for processing the queries. E is a binary relation on T_S, so that $E \subseteq T_S \times T_S$.*

Each edge (n_i, n_j) in the GQG is labelled with the ID of the Query Type processed through the computational steps (nodes) n_i and n_j. That is, an edge labelled Q_j represents one of the paths that a query of type Q_j takes for processing. Fig. 2 shows a GQG constructed by marking the paths of the nine Query Types presented in Fig 1. See Appendix for a description of the nine Query Types in Fig. 2 and the sequence of computational steps required for processing them.

4 Identifying the Type of a Query

We describe in this section how GMB identifies the type of a query by matching it with Query Types in the GQG. Let us first redefine the concept of Query Type based on the GQG concept presented in definition 1. A Query Type Q_i is a set of edges connecting computational steps in the GQG and these edges represent the sequence of paths between the computational steps that a query of type Q_i takes for processing. After a query is submitted to GMB, it matches the path of the query with the paths of Query Types in the GQG to identify the query's type. If there is a match, GMB will assign the query the ID of the matching Query Type. Otherwise, it will mark the path of this query in the GQG as a path of a new Query Type and assign it a new ID. We constructed an algorithm called Classify_Query that identifies the type of a query based on the described process.



n1 Protein name / nucleotide
n2 Protein structure search
n3 DNA sequence fragment
n4 DNA sequence translation
n5 Correct reading frame determination
n6 Protein sequence search
n7 Protein-protein interaction search
n8 Protein motifs, domains, and/or families search
n9 Genetic sequence
n10 Genome sequence search
n11 Proteins homolog search
n12 Protein function search
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
Q9

Fig. 2: A GQG showing the paths of the 9 Query Types in Fig. 1 (n_i denotes node number i)

The algorithm is shown in Fig. 3. First, the computational steps (in terms of their node names in the GQG) that the query goes through for processing are queued in a queue called Q_u. The inputs to the algorithm are: (1) Q_u, (2) the GQG represented as *adjacency lists* (*Adj*[*n*]) for fast navigation, and (3) ID_n (the list of Query Types' IDs that visit each node *n* in the GQG (If Query Type Q_i visits node *n* , $i \in \mathrm{ID}_n$). Fig. 4 shows ID_n of each node *n* in the GQG shown in Fig. 2. Fig. 5 shows the Adjacency list of each node *n* in the GQG shown in Fig. 2. We now formalize the adjacency list concept.

Definition 2, Adjacency list: Let GQG = (V, E), where V is the set of nodes and E is the set of edges in the GQG. The adjacency list of the GQG consists of an array Adj of |V| lists, one for each node in V. For each $u \in V$ *, the adjacency list Adj[u] contains all the nodes v such that there is an edge (u, v)* $\in$ *E.*

Algorithm Classify_Query works as follows. Every time a node *n* is de-queued from queue Q_u (line 4) and marked as visited (line 11) to denote it has been processed. Below are all possible processing scenarios that identify a query's type after the last node is de-queued from Q_u:

1. *There are Query Types left in set cur_qrys:*
   a. *There is only one Query Type left in the set (line 21):*
      - ➢ This indicates that there is only one Query Type takes the exact path that the query took in the GQG. Thus, the ID of the remaining Query Type identifies the query's type (line 22). Example 1 illustrates this scenario.
   b. *There are more than one Query Type left in the set:*
      - ➢ *The last node de-queued from* queue Q_u *is not a leaf node in the GQG (line 23):*
        - *n* is the last node de-queued from Q_u and *m* is the node de-queued prior to it. First, *m* is removed from the adjacency list of *n* (line 25). Then, a set difference operation between the remaining list and set cur_qrys is performed (line 27) and the result is a Query Type, whose ID identifies the query's type (line 28). Example 2 illustrates this scenario.
      - ➢ *The last node de-queued from* queue Q_u *is a leaf node in the GQG (line 30).*
        - First, the last node de-queued from Q_u is removed from the adjacency list of the node de-queued prior to it (line 31). Then, a set difference operation between the remaining list and set cur_qrys is performed (line 33) and the result is a Query Type, whose ID identifies the query's type (line 34).

2. *There is no Query Type left in set cur_qrys (line 17) or the last de-queued node is not in the adjacency list of the node de-queued prior to it (line 19):*

This indicates there is no matching Query Type. The path that this query took in the GQG will be considered the path of a new Query Type. This path will be marked in the GQG and assigned an ID (lines 18 and 19).

The time complexity of algorithm Classify_Query is $O\left(\sum_{i=1}^{|Q_u|} Adj[m_i]\right)$, where m_i is one of the

```
Classify_Query ($Q_u$, $Adj[n]$, $ID_n$) {
1.  $n$ ← de-queue($Q_u$)                                  /*de-queue an element from queue $Q_u$*/
2.  visit($n$) = TRUE                                                  /*mark node n as visited*/
3.  cur_qrys ← $ID_n$                                            /*assign $ID_n$ to set cur_qrys*/
4.  while $Q_u$ is not empty
5.      do {
6.          $m$ ← $n$
7.          $S$ ← Adj[$m$]                  /*assign the adjacency list of node m to set S */
8.          $n$ ← de-queue($Q_u$)
9.          if visit($n$) != TRUE
10.             then {
11.                 visit($n$) = TRUE                               /* mark node n as visited*/
12.                 $K$ ← all distinct nodes in list $ID_n$
13.                 }
14.             else  $K$ ← all duplicate nodes in list $ID_n$
15.         if $n \in S$                                         /*if node n is adjacent to node m*/
16.             then  cur_qrys ← cur_qrys $\cap K$
17.                 if cur_qrys == NIL
18.                     Then the Query Type is new. Assign it an ID, mark its path in the GQG, and exit
19.             else add $n$ to Adj[$m$], assign the Query Type an ID, mark its path in the GQG, and exit
20.     }
21. if | cur_qrys | == 1                                  /*if set cur_qrys contains one ID only*/
22.     then  This ID identifies the query's  type
23.     else{ if Adj[$n$] $\neq$ NIL                          /*if n is not a leaf node in the GQG*/
24.             then {
25.                 $S$ ← Adj[$n$] - $m$
26.                 adj_qrys ← the $ID_n$ of each node in set $S$
27.                 cur_qrys ← cur_qrys \ adj_qrys                           /*set difference*/
28.                 The ID in set cur_qrys identifies the query's type
29.                 }
30.             else {
31.                 $R$ ← Adj[$m$] – $n$                               /*remove n from Adj[m]*/
32.                 pred_ qrys ← the $ID_n$ of each node in set $R$
33.                 cur_qrys ← cur_qrys\ pred_ qrys                          /*set difference*/
34                  The ID in set cur_qrys identifies the query's type
                    }
            }
    }
```

Fig. 3: Algorithm Classify_Query

$ID_{n1} = \{Q_3, Q_5, Q_7, Q_8, Q_9\}$; $ID_{n2} = \{Q_5\}$; $ID_{n3} = \{Q_2, Q_4\}$
$ID_{n4} = \{Q_2, Q_4\}$; $ID_{n5} = \{Q_2, Q_4\}$; $ID_{n6} = \{Q_2, Q_3, Q_4, Q_6, Q_7, Q_8, Q_9\}$
$ID_{n7} = \{Q_6\}$; $ID_{n8} = \{Q_3, Q_9\}$; $ID_{n9} = \{Q_1\}$; $ID_{n10} = \{Q_1, Q_7\}$
$ID_{n11} = \{Q_1, Q_2, Q_4, Q_5, Q_6, Q_8\}$; $ID_{n12} = \{Q_2, Q_3, Q_8\}$

Fig. 4: *ID_n* of each node in the GQG in Fig. 2

$n_1 \rightarrow \{n_2, n_6, n_{11}\}$; $n_2 \rightarrow \{n_{11}\}$; $n_3 \rightarrow \{n_4\}$; $n_4 \rightarrow \{n_5\}$
$n_5 \rightarrow \{n_6\}$; $n_6 \rightarrow \{n_7, n_8, n_{10}, n_{11}, n_{12}\}$
$n_9 \rightarrow \{n_{10}\}$; $n_{10} \rightarrow \{n_{11}\}$; $n_{11} \rightarrow \{n_6, n_{12}\}$; $n_{12} \rightarrow \{n_8\}$

Fig. 5: Adjacency list of each node in the GQG in Fig. 2

nodes in queue Q_u. We now present examples 1 and 2 to illustrate the key lines of the algorithm. The examples illustrate two of the processing scenarios described above that identify the type of a query. The input to the algorithm is the GQG shown in Fig. 2 represented by the ID_n and *Adj*[*n*] shown in figures 4 and 5 respectively. First, the computational steps (in terms of their node names in the GQG in Fig. 2) of each query are queued in queue Q_u.

Example 1: Find the motifs for antigenetic human that participate in apoptosis and are homologous to the lymphocyte associated receptor of death (also known as lard).

First, the query's nodes are queued in queue Q_u: $Q_u = [n_1, n_6, n_{12}, n_8]$.
Line 1: $n \leftarrow n_1$. *Line 3*: cur_qrys = $\{Q_3, Q_5, Q_7, Q_8, Q_9\}$. The following table shows some of the other key lines of the algorithm that process the query.

Line 4	*Line 6*	*Line 7*	*Line 8*	*Line 12*	*Line 15*	*Line 16*
1st while iteration	$m \leftarrow n_1$	S = Adj[n_1] = $\{n_2, n_6, n_{11}\}$	$n \leftarrow n_6$	K = ID_{n6} = $\{Q_2, Q_3, Q_4, Q_6, Q_7, Q_8, Q_9\}$	$n_6 \in$ S	cur_qrys = $\{Q_3, Q_5, Q_7, Q_8, Q_9\} \cap \{Q_2, Q_3, Q_4, Q_6, Q_7, Q_8, Q_9\} = \{Q_3, Q_7, Q_8, Q_9\}$
2nd while iteration	$m \leftarrow n_6$	S = Adj[n_6] = $\{n_7, n_8, n_{10}, n_{11}, n_{12}\}$	$n \leftarrow n_{12}$	K = ID_{n12} = $\{Q_2, Q_3, Q_8\}$	$n_{12} \in$ S	cur_qrys = $\{Q_3, Q_7, Q_8, Q_9\} \cap \{Q_2, Q_3, Q_8\} = \{Q_3, Q_8\}$
3rd while iteration	$m \leftarrow n_{12}$	S = Adj[n_{12}] = $\{n_2, n_3, n_8\}$	$n \leftarrow n_8$	K = ID_{n8} = $\{Q_3, Q_9\}$	$n_8 \in$ S	cur_qrys = $\{Q_3, Q_8\} \cap \{Q_3, Q_9\} = \{Q_3,\}$

Line 22: The query's type is Q_3.

Example 2: Find the motifs and domains of the protein CALU.

First, the query's nodes are queued in queue Q_u: $Q_u = [n_1, n_6, n_8]$.
Line 1: $n \leftarrow n_1$. *Line 3*: cur_qrys= ID_{n1}= $\{Q_3, Q_5, Q_7, Q_8, Q_9\}$. The following table shows some of the other key lines of the algorithm that process the query.

Line 4	*Line 6*	*Line 7*	*Line 8*	*Line 12*	*Line 15*	*Line 16*
1st while iteration	$m \leftarrow n_1$	S = Adj[n_1] = $\{n_2, n_6, n_{11}\}$	$n \leftarrow n_6$	K = ID_{n6} = $\{Q_2, Q_3, Q_4, Q_6, Q_7, Q_8, Q_9\}$	$n_6 \in$ S	cur_qrys = $\{Q_3, Q_5, Q_7, Q_8, Q_9\} \cap \{Q_2, Q_3, Q_4, Q_6, Q_7, Q_8, Q_9\} = \{Q_3, Q_7, Q_8, Q_9\}$
2nd while iteration	$m \leftarrow n_6$	S = Adj[n_6] = $\{n_7, n_8, n_{10}, n_{11}, n_{12}\}$	$n \leftarrow n_8$	K = ID_{n8} = $\{Q_3, Q_9\}$	$n_8 \in$ S	cur_qrys = $\{Q_3, Q_7, Q_8, Q_9\} \cap \{Q_3, Q_9\} = \{Q_3, Q_9\}$

Line 31: $R = \{n_7, n_8, n_{10}, n_{11}, n_{12}\} - n_8 = \{n_7, n_{10}, n_{11}, n_{12}\}$.
Line 32: pred_ qrys = $\{Q_6, Q_1, Q_7, Q_2, Q_4, Q_5, Q_8, Q_3\}$.
Line 33: cur_qrys = $\{Q_3, Q_9\} \setminus \{Q_6, Q_1, Q_7, Q_2, Q_4, Q_5, Q_8, Q_3\} = \{Q_9\}$.
Line 34: The query's type is Q_9.

5 Answering and Updating FUQ

5.1 Answering FUQ from Cached Hash Tables

After GMB computes the list of FUQ types, it constructs a multi-valued hash table for each FUQ type in the list. Each table contains the results of a query type. The tables are populated from biological databases at remote site(s) and cached at the middleware. Each table has the following relation schema: $R(n_{p1}, n_{p2}, \ldots, n_{r1}, n_{r2}, \ldots)$, where:

- R: a Query Type's ID (e.g. Q_i).
- n_{pi}: a predicate condition attribute (computational step), whose values filter result tuples, retaining only those satisfying the condition.
- n_{ri}: a result/return attribute (computational step) whose values are the query's results.

The set of all tuples to be retrieved (for populating columns n_{ri}) can be expressed using the Tuple Relational Calculus as follows: $\{t_1.n_{ri}, t_2.n_{rk}, \ldots, t_n.n_{rm} \mid \text{COND}\ (t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$, where $t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m}$ are tuple variables, each n_{rj} is an attribute of the relation on which t_i ranges, and COND (t_i) is a conditional expression involving t_i. The result of the query is the set of all tuples t_i that satisfy COND (t_i). Table 1 depicts a fragment of hash table containing the results of query type 5 (recall Fig. 1).

Table 1: A fragment of hash table containing the results of Query Type 5 (Q_5)

Disease name	*Proteins involved in the disease*	*Proteins similar to the ones involved in the disease*
HIV	- gp120 - gp41	- Envelope glycoprotein gp160, Gag polyprotein - Protein Nef, - Protein Rev - …
Malaria	- Heme Detoxification Protein (HDP)	- 110 kDa antigen - Circumsporozoite protein - Merozoite surface antigen 2 - 101 kDa malaria antigen - …
Alzheimer	- amyloid precursor protein (APP)	- Amyloid beta A4 protein - Cathepsin D - Presenilin-1 - Alpha-synuclein - …
….	…..	…..

When GMB determines that the type of a submitted query is currently an active FUQ type, it finds a pointer to the FUQ type's cached hash table from a catalog. It then strips from the query the value(s) of its n_{pi} attribute to use it as a key to the hash table to get the results. If the query's type contains more than one predicate condition, GMB will concatenate the values of the query's n_{pi} attributes and use the concatenated value as a key to the hash table.

5.2 Updating the Set of FUQ Types

GMB keeps a set of FUQ types at the middleware and it dynamically updates this set periodically at certain intervals called *Update Points*. We formalize this concept as follows:

Notation 1, Update Points: Update Points are temporal points at which the set of FUQ types is dynamically updated. The interval k between two adjacent Update Points is the number of queries submitted to the middleware between the two points.

FUQ types are determined by sorting Query Types in decreasing order based on their frequency, and then selecting the top x ones as the set of FUQ types to be stored at the middleware. Let S_i and S_{i+1} be sets containing the FUQ types at the adjacent Update Points i and i+1 respectively. Let x denote the number of FUQ types in each set. Let D be a set containing the *set difference* of S_{i+1} and S_i. That is, $D = S_{i+1} \setminus S_i$. Let R be a set containing the *intersection* of S_{i+1} and S_i. That is, $R = S_{i+1} \cap S_i$. Each FUQ type in set D requires a hash table to be created, populated, and cached at the middleware. The FUQ types in set R do not require the creation of new hash tables, since they already have ones. To minimize the overhead of creating and populating new hash tables, we need to maximize set R and minimize set D. Intuitively, set R increases if we decrease the interval k between i and i+1. However, if we decrease the interval k, we would increase the number of Update Points, which leads to expensive computational overhead. On the other hand, increasing the interval k may decrease the chance of some Query Types to be included in the set of FUQ types even though they are actually frequently used. Therefore, selecting a near-optimal value for k is crucial.

We can consider the interval k between Update Points i and i+1 a near-optimal, if the k submitted queries have exactly x distinct Query Types, for the following reason. FUQ tend to have temporal locality property. And, since set S_i contains x query types, there is a high probability that either all or the majority of the Query Types in set S_{i+1} be the same x Query Types in set S_i. Thus, a near-optimal k between Update Points i and i+1could be achieved by setting parameter k when the number of submitted distinct Query Types between the two points reaches x.

6 Determining the Number of FUQ types and Update Points

6.1 Determining an Ideal Number of FUQ Types

We investigate in this subsection the determination of an ideal number of FUQ types that guarantee acceptable hit rate and performance. Intuitively, increasing the number of FUQ types increases hit rate. But, for performance reasons we need to limit the number of these Query Types to reduce the overhead of creating and populating new hash tables. Thus, we need to adopt a criterion for what an ideal number of FUQ types. Usually, hit rate increases significantly as the number of FUQ type increases up to a certain point, then the impact of the increase on the hit rate starts to decline. Thus, it is crucial to determine the stationary point at which the impact of FUQ type on hit rate starts to decline. Let w be a parameter ratio of the number of FUQ types to the overall number of Query Types. We specify below the number of FUQ types x in terms of parameter w. In section 8, we determine experimentally a stationary value for parameter w at which the impact of FUQ type on hit rate starts to decline.

$$\textit{Lemma}: \quad x \approx w \times \left\lfloor \frac{m-5}{2} \right\rfloor \times \frac{m(m-1)(m-2)\ldots\left\lfloor \frac{3m-5}{4} \right\rfloor}{3 \times 4 \times \ldots \times \left\lceil \frac{m+1}{4} \right\rceil}$$

for all values of $m \geq 8$, where m is the number of nodes in the GQG.

Proof: Let n be the number of *all possible* Query Types. $n = m + \binom{m}{2} + \binom{m}{3} + \ldots + \binom{m}{m-1}$.

The value of m is negligible. So, $n = \sum_{i=2}^{m-1} \frac{m!}{(m-i)!\,i!} = \sum_{i=2}^{m-1} \frac{m\,(m-1)\ldots(m-i-1)\,(m-i)!}{(m-i)!\ i!} = \sum_{i=2}^{m-1} \frac{m\,(m-1)\ldots(m-i-1)}{i!}$, which is a summation of $m-3$ terms. The last two terms are always insignificant. After disregarding these two terms, we can observe symmetry between the first and last $\left\lfloor \frac{m-5}{2} \right\rfloor$ terms. If we factor out $\frac{m(m-1)(m-2)(m-3)}{2}$ from all the terms, the term in the middle of the first $\left\lfloor \frac{m-5}{2} \right\rfloor$ terms is computed as follows (after being factored out): $\frac{(m-4)(m-5)\ldots\left\lfloor \frac{3m-5}{4} \right\rfloor}{3\times 4\times\ldots\times\left\lceil \frac{m+1}{4} \right\rceil}$. The summation of the first $\left\lfloor \frac{m-5}{2} \right\rfloor$ terms $\approx$ $\frac{m(m-1)\,(m-2)\,(m-3)}{2}\left\lfloor \frac{m-5}{2} \right\rfloor \times \frac{(m-4)(m-5)\ldots\left\lfloor \frac{3m-5}{4} \right\rfloor}{3\times 4\times\ldots\times\left\lceil \frac{m+1}{4} \right\rceil}$. Multiplying this summation by 2 gives the value of n: $n \approx \left\lfloor \frac{m-5}{2} \right\rfloor \times \frac{m(m-1)(m-2)\ldots\left\lfloor \frac{3m-5}{4} \right\rfloor}{3\times 4\times\ldots\times\left\lceil \frac{m+1}{4} \right\rceil}$.

$$w = \frac{x}{n}.\ \text{Therefore, } x = w * n \approx w \times \left\lfloor \frac{m-5}{2} \right\rfloor \times \frac{m\,(m-1)\,(m-2)\ldots\left\lfloor \frac{3m-5}{4} \right\rfloor}{3\times 4\times\ldots\times\left\lceil \frac{m+1}{4} \right\rceil} \qquad (1)$$

6.2 Determining Acceptable Update Points

We constructed a probability formula that helps in determining an acceptable interval k between each two adjacent Update Points. Let $\Pr(V_{x,k})$ denote the probability of the event that the k queries submitted between Update Points i and i+1 have exactly x distinct types. The formula computes the probability of each (x, k) pair. Let:

- T denotes a set that contains all possible Query Types.
- n denotes the number of these Query Types.
- Q_j denotes that a query of type j is submitted to the middleware between Update Points i and i+1.
- $\overline{Q}_j$ denotes that there is no query of type j is submitted to the middleware between Update Points i and i+1.
- y denotes the number of distinct types of queries that have not been submitted $y = n\text{-}x$.
- V_x denotes the event that the queries submitted between i and i+1 have exactly x distinct types.

T ={Q$_1$, Q$_2$,.., Q$_n$}. The sample space V_x constitutes $\binom{n}{y}$ different set combination. *Hereinafter, the operators $\cap$ and $\cup$ represent the probability notations "and" and "or" respectively.*

$$\Pr(V_x) = \binom{n}{y} \Pr(\overline{Q}_1 \cap \overline{Q}_2 \cap\ldots\cap \overline{Q}_y \cap Q_1 \cap Q_2 \cap\ldots\cap Q_x).$$

Let $G = \bar{Q}_1 \cap \ldots \cap \bar{Q}_y \cap Q_1 \cap \ldots \cap Q_x$. Then $\bar{G} = Q_1 \cup \ldots \cup Q_y \cup \bar{Q}_1 \cup \ldots \cup \bar{Q}_X$.

$\bar{G} = (Q_1 \cup \ldots \cup Q_y) \cup ((\bar{Q}_1 \cap \ldots \cap \bar{Q}_y) \cap (\bar{Q}_1 \cup \bar{Q}_2 \cup \ldots \cup \bar{Q}_x))$.

$\Pr(\bar{G}) = \Pr(Q_1 \cup \ldots \cup Q_y) + \Pr((\bar{Q}_1 \cap \ldots \cap \bar{Q}_y) \cap (\bar{Q}_1 \cup \bar{Q}_2 \cup \ldots \cup \bar{Q}_x)) = \bar{G}\, 1 - \Pr(\bar{Q}_1 \cap \ldots \cap \bar{Q}_y) + \Pr((\bar{Q}_1 \cap \ldots \cap \bar{Q}_y) \cap (\bar{Q}_1 \cup \bar{Q}_2 \cup \ldots \cup \bar{Q}_x))$.

By using the symmetry among $\bar{Q}$, we have:

$\Pr(\bar{G}) = 1 - \Pr(\bar{Q})^y + \Pr(\bar{Q}^y \bar{Q}_1 \cup \bar{Q}^y \bar{Q}_2 \cup \ldots \cup \bar{Q}^y \bar{Q}_x) =$

$1 - \Pr(\bar{Q})^y + \Pr(\bar{Q}^{y+1} \cup \bar{Q}^{y+1} \cup \ldots \cup \bar{Q}^{y+1}) =$

$$x \Pr(\bar{Q})^{y+1} - \binom{x}{2} \Pr(\bar{Q})^{y+2} + \sum_{i=3}^{x-1} \binom{x}{i} \Pr(\bar{Q})^{i+y+1} + \Pr(\bar{Q})^n \approx x \Pr(\bar{Q}^{y+1} - \binom{x}{2} \Pr(\bar{Q})^{y+2}.$$

Therefore, $\Pr(\bar{G}) \approx 1 - \Pr(\bar{Q})^y + x \Pr(\bar{Q})^{y+1} - \binom{x}{2} \Pr(\bar{Q})^{y+2}$.

$$\Pr(G) \approx \Pr(\bar{Q}^y) - x \Pr(\bar{Q})^{y+1} + \binom{x}{2} \Pr(\bar{Q})^{y+2}.$$

$$\Pr(V_x) = \binom{n}{y} \Pr(G) \approx \binom{n}{y} [\Pr(\bar{Q}^y) - x \Pr(\bar{Q})^{y+1} + \binom{x}{2} \Pr(\bar{Q})^{y+2}]$$

Thus, $\Pr(V_{x,k}) \approx \binom{n}{y} [\Pr(\bar{Q})^{yk} - x \Pr(\bar{Q})^{k(y+1)} + \binom{x}{2} \Pr(\bar{Q})^{k(y+2)}]$.

We can adopt a criterion for an *acceptable k*, such as a value for *k* that yields $\Pr(V_{k-x})$ close to 50%. We can try different values for *k* greater than *m*, and keep decreasing *k* until we get $\Pr(V_{k-x})$ close to 50%.

7 System Architecture and Implementation

Fig. 6 shows the system architecture. The arrows in the figure depict the following information. The arrows $\dashrightarrow$ depict the processing path of a Query Type that is currently in the list of FUQ types. The arrows $\cdots\cdots\blacktriangleright$ depict the processing path of a Query Type that is currently not in the list of FUQ types. The arrows $\longrightarrow$ depict the processing actions when the list of FUQ types is updated. Result/return attributes of Query Types are kept in a *catalog*. Table 2 shows the format of a catalog, which associates each attribute with the following:

(1) The location of the remote biological database that contains the attribute.

(2) The ID of the Query Type that contains the attributes.

(3) The Current Status of the Query Type. Since the list of FUQ types is periodically updated dynamically, the catalog keeps track of the current status of each Query Type. Status "A" indicates that the Query Type is currently an active FUQ type. Status "I" indicates that the Query Type is currently not a FUQ type.

(4) A pointer to the cached hash table in the middleware that contains the attribute's values.

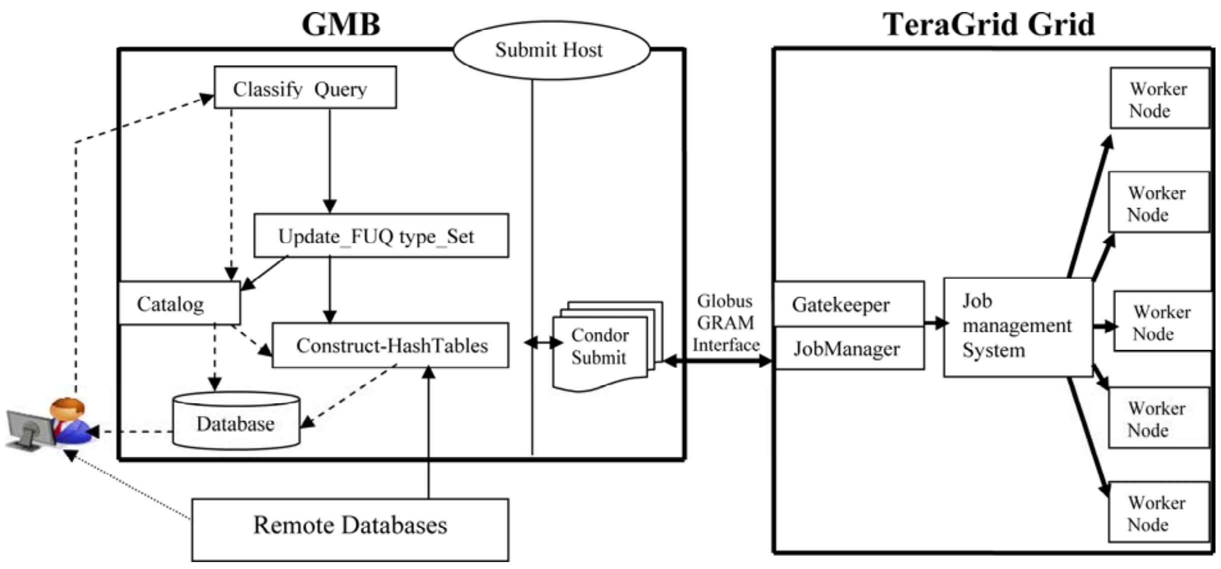


Fig. 6: System architecture

Table 2: Fragment of a catalog

Attribute name	*Remote database location*	*Query Type ID*	*Query Type current status*	*Pointer to cached hash table*
….	….	….	….	….

GMB employs module Classify_Query to validate and classify queries per their types and frequency. The module sends the type of the query to the catalog. If the catalog indicates that the type is currently a FUQ type, the query is answered from the cached hash tables at the middleware. Module Update_FUQtype_Set updates the set of FUQ types at Update Points. For each FUQ type, module Construct-HashTables constructs a multi-valued hash table for each FUQ type, populates it, and caches it.

After each *k* number of queries submitted, module Classify_Query notifies module Update_FUQtype_Set to re-compute/update the list of FUQ types. Module Update_FUQtype_Set performs the following tasks:

- Updates the catalog with the current list of FUQ types.
- For each Query Type that is newly added to the list of FUQ types, the module requests the instance values of the query's result/return attributes from remote databases.
- For each Query Type that is currently not in the list of FUQ types and it was in the list before the update, the module deletes the Query Type's cached hash table.

We installed on the submit machine all the standard Grid middleware components such:

- Condor: Condor is a workload management system. It provides a job queuing mechanism, resource monitoring, resource management, and scheduling policy. When a user submits his/her jobs, Condor puts them into a queue, runs the jobs based upon a scheduling policy, and informs the user when the jobs complete.
- Globus toolkit: The toolkit includes software services and libraries for resource discovery, monitoring and management, in addition to data management.
- GridFTP: GridFTP is an extension of the standard File Transfer Protocol (FTP) for use with Grid computing. It provides high performance file transfer for Grid applications.

We also created a Globus GRAM interface. The following list contains the key software components we installed in the system: Nimbus framework [22]; Clarens [8]; RENCI's Science Desktop [25]; Application Hosting Environment [1]; Custom MDS Java Query client;

Client desktop grid job submission framework [7]. The gatekeeper of the TeraGrid acts as the single point of job submissions to the resources. This gatekeeper provides the Globus GRAM interface and is responsible for submitting the job to the local job management system. The Condor will submit a job to TeraGrid using GRAM interface

8 Experimental Results

We have implemented GMB middleware in Java, run on Intel® Xeon® Processor X7560 Sequence server, with 1Terabyte of memory, 16GB DIMMS, and 2.266 GHz base frequency. We run the client side application on Intel(R) Core(TM)2 Dup CPU processor, with a CPU of 2.1 GHz and 3 GB of RAM, under Windows Vista. We used TeraGrid-ORNL [30] data resources for the experiments. For dataset accesses, we obtained real-world trace archive containing access logs to the National center for Biotechnology Information (NCBI) [23]. From this archive, we randomly selected 200 access records. We converted and formatted these records to queries accepted by GMB. From the 200 records, we considered the 40 most frequent ones as FUQ. These queries fall under the nine Query Types shown in Fig. 1. We cached the results of the nine FUQ types in the middleware.

8.1 Query Execution Time

We first answered the 40 FUQ directly from TeraGrid databases and then from the cached results at the middleware and computed their execution times. Fig. 7 shows the average query execution time for each of the nine Query Types. As the figure shows, the execution time of a FUQ answered from the cache is almost negligible. Let t_i be the time of populating a hash table h_i with the results of Query Type Q_i and then caching it. As Fig. 7 shows, t_i is always less than the average execution time of a query of type Q_i answered directly from TeraGrid databases. Let: m_i be the number of FUQ answered from hash table h_i, $t_{qi}^{overall}$ be the overall execution time of query q_i of type Q_i, and t_{qi}^{cache} be the time that took q_i to be answered from the cache. $t_{qi}^{overall} = t_{qi}^{cache} + t_i/m_i$. As can be inferred from Fig. 7 that the value t_i/m_i is very small. Thus, even if we consider the overhead t_i, $t_{qi}^{overall}$ is still negligible. Note that the set of FUQ types may remain completely or partially unchanged at Update Points, which eliminates or minimizes the overhead of populating new hash tables.

8.2 Hit Rate

We define hit rate as the rate of finding the type of a submitted query in the set of FUQ types. From the types of the 200 queries, we selected the top 5% frequent ones to be the set of FUQ types. We then ran a random query selector program 20 times on the 200 queries and computed the hit rate. We then varied the percentage of FUQ types six times by the increments of 10% and repeated the same process described above. Fig. 8 shows the hit rates. As the figure shows, the hit rate increases significantly as the percentage of FUQ type increases from 5% to about 25%. Then, the impact of the increase of FUQ type on hit rate declines significantly. As stated in section 6, we should limit the number of adopted percentage of FUQ types at the middleware for performance reasons. Thus, we can select a percentage of FUQ types that does not exceed 25%.

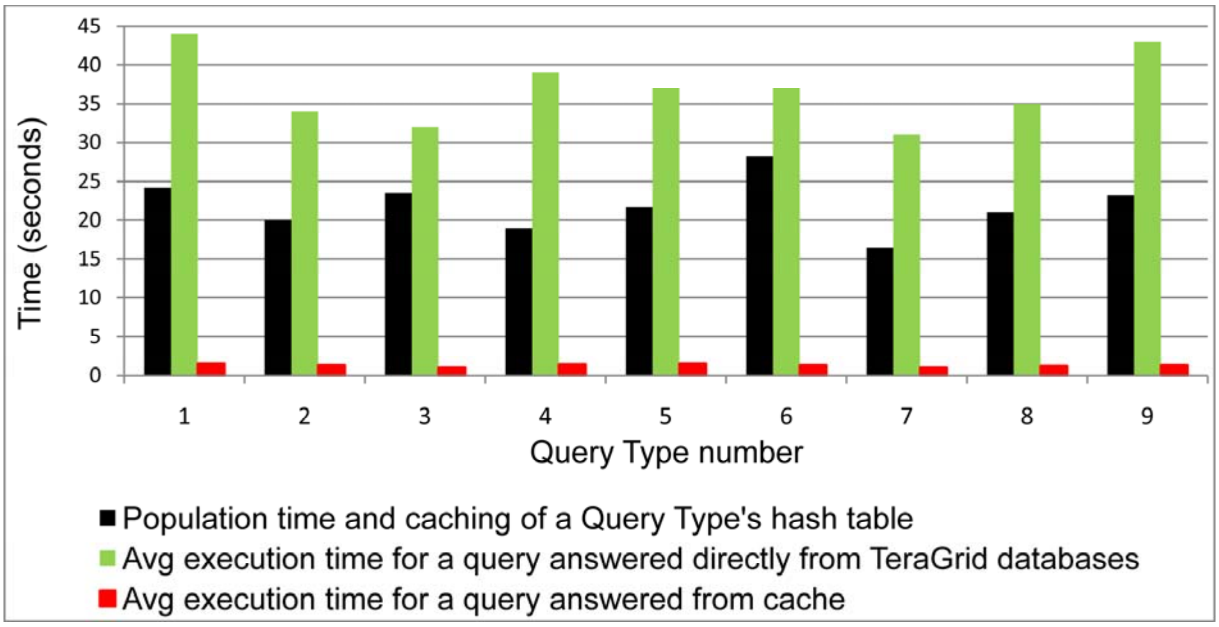


Fig. 7: Average query execution time and hash table population time per Query Type

Based on our experiments, the ideal (smallest) percentage of FUQ types that achieves good hit rate ranges from 20% to 25% of the overall number of Query Types. Thus, to achieve an ideal number of FUQ type we should set parameter *w* in Equation 1 (recall section 6.1) to a value in the range 0.2 – 0.25.

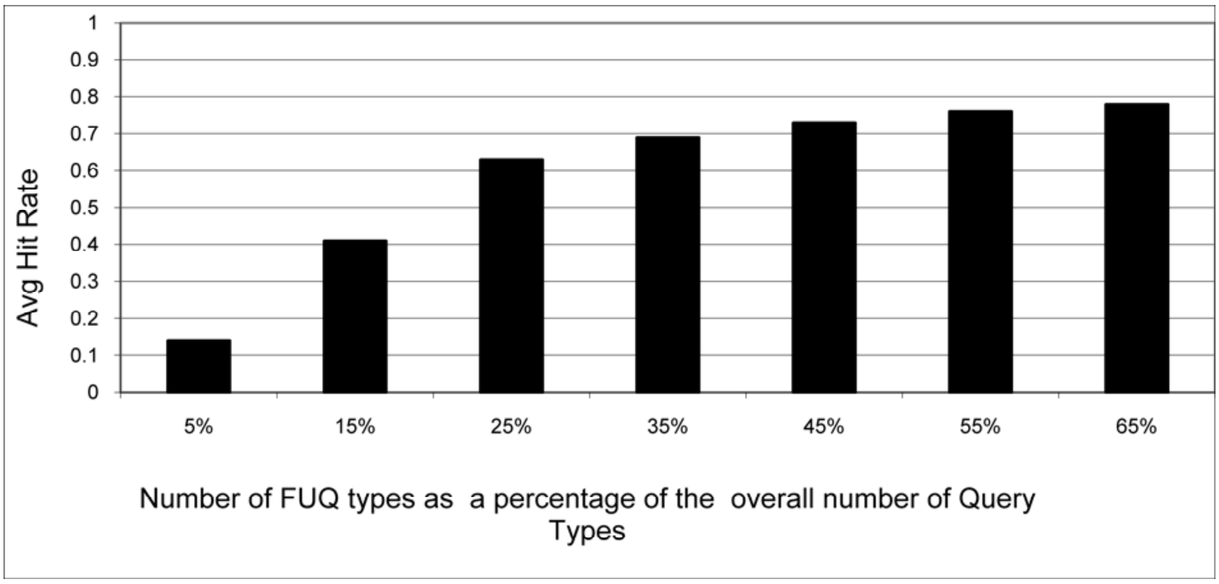


Fig. 8: Hit rate

8.3 Impact of Number of Requests on Hit rate

To study the impact of the combination of number of requests and number of FUQ types on hit rate, we computed hit rate under various number of requests (ranged from 25 to 200 requests) and under various percentages of FUQ types (ranged from 5% to 65%). Fig. 9 shows the results. As the figure shows:

- At smaller number of requests: as the percentage of FUQ increases, hit rate increases in *high rates*.
- At higher number of requests: as the percentage of FUQ increases, hit rate increases in *low rates*.

That is, at smaller number of requests, hit rate increases as the number of FUQ increases in higher rates than at higher number of requests. Averaging the hit rates of the different FUQ percentages at a certain point of number of requests gives a value close to the hit rate corresponding to the 25% FUQ trend at this point. This confirms our finding stated in section 8.2 that the smallest percentage of FUQ types that achieves good hit rate ranges from 20% to 25%.

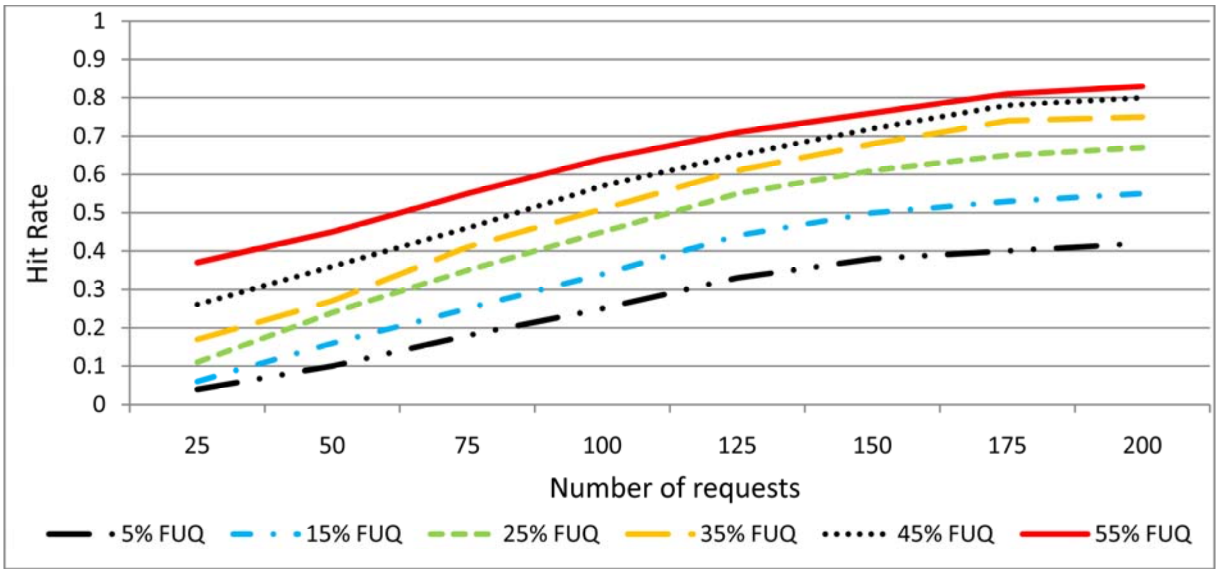


Fig. 9: Hit rate per number of requests

8.4 Cache Size

To study how big cached hash tables are, we computed for each FUQ type, its average cached hash table size as a percentage of the biological data, from which the tables were populated (see Fig. 10). As the figure shows, the average cached hash table size is always less than 14% of the biological data. As can be seen, the size of cached hash tables are not enormous, where virtual memory may not be needed. To study cached hash tables scalability, we computed the percentage of cache size needed to store 5%, 15%, 25%, 35%, 45%, 55%, and 65% of FUQ types (see Fig. 11). As the figure shows, the scalability is acceptable and that only less than 50% of cache is needed to store 65% of FUQ types.

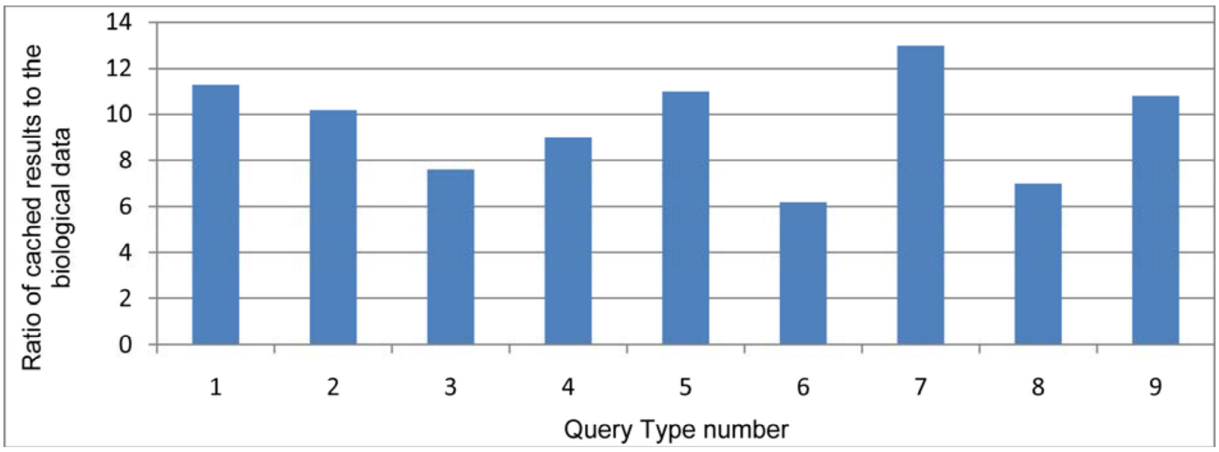


Fig. 10: Average cached hash table size as a percentage of the biological data

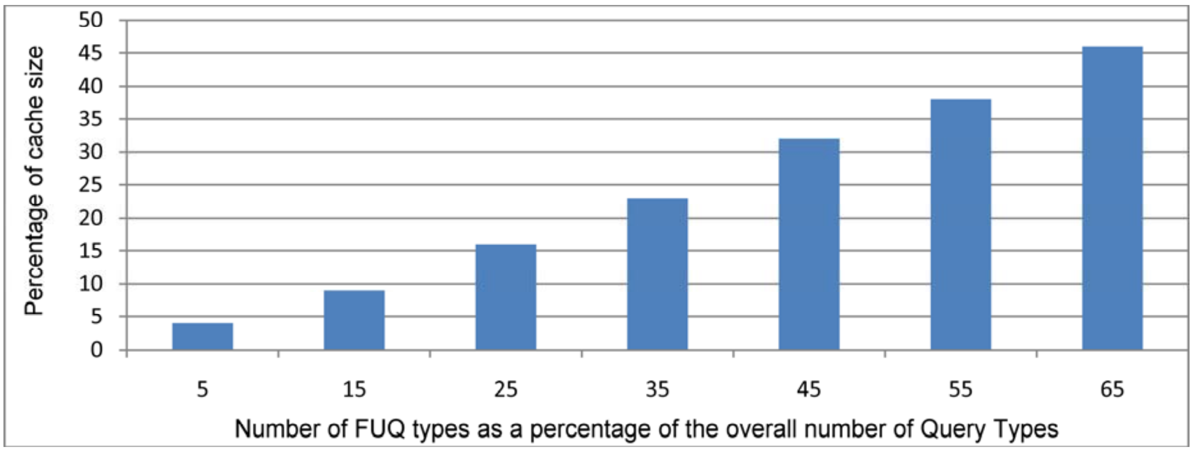


Fig. 11: Percentage of cache size needed to store variable percentage of FUQ types

9 Conclusion

We propose in this study a Grid middleware called GMB that improves the response time of FUQ over Grid environment for bioinformatics. GMB classifies queries per their types and ranks Query Types based on their frequencies. Each FUQ will be answered from the middleware rather than from remote sites, which improves the query's response time. The experimental results showed the following:

(1) The execution time of a FUQ answered from the cache is almost negligible.
(2) Hit rate increases significantly as the percentage of FUQ type increases from 5% to about 25%. Then, the impact of the increase on hit rate declines significantly.
(3) The ideal percentage of FUQ types that achieves good hit rate ranges 20% - 25%.
(4) As the percentage of FUQ increases, hit rate increases in *high rates* when the number of requests is small. And, as the percentage of FUQ increases, hit rate increases in *low rates* when the number of requests is high.
(5) The size of cached hash tables is not enormous Virtual memory may not be needed.
(6) The scalability of cache is acceptable.

References

[1] Application Hosting Environment (AHE): http://www.realitygrid.org/AHE/A
[2] Asia Pacific BioGRID: http://compaq.apbionet.org/grid/.
[3] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Essential components for the grid. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*, 2003.
[4] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Essential components for the grid. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid Resource Management: State of the Art and Future Trends*, 2003.
[5] BioPerl: http://www.bioperl.org/wiki/Main_Page
[6] B. Tierney, J. Lee, M. Holding, J. Hylton, and F. Drake. A network-aware distributed storage cache for data intensive environments. In *Proceedings of the IEEE HPDC-8*, 1999.
[7] Client desktop grid job submission framework: http://projects.arcs.org.au/trac/grisu/
[8] Clarens: http://clarens.sourceforge.net/
[9] Ccsm-community climate system model. http://www.ccsm.ucar.edu, 2010.
[10] Ccsm-community climate system model. http://www.ccsm.ucar.edu, 2006.

[11] D. Pérez-Rey, V. Maojo, M. García-Remesal, R. Alonso-alvo, H. Billhardt, F. Martin-Sánchez, and A. Sousa. ONTOFUSION: Ontology-based integration of genomic and clinical databases. Computers in Biology and Medicine 36:712–730, 2006.

[12] Earth system grid. http://www.earthsystemgrid.org, 2010.

[13] E. J. Otoo, D. Rotem, A. Romosan, and S. Seshadri. File Caching in Data Intensive Scientific Applications on Data-Grids. In *Proceedings of Data Management in Grids (DMG)*, 2005.

[14] Earth system grid. http://www.earthsystemgrid.org, 2006.

[15] E. J. Otoo, D. Rotem, and A. Romosan. Optimal file-bundle caching algorithms for data-grids. In *Proceedings of Supercomputing*, 2004.

[16] H. Garcia-Molina and A. Paepcke, Proposal for I**3 Client Server Protocol, *Technical Report*, 1996.

[17] K. A. Hawick, H. A. James, and P. D. Coddington. A Reconfigurable Component-based Problem Solving Environment. *Proc. HICSS-34*, 2000.

[18] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proc. NetStore '99.*

[19] J. Hammer, M. Breunig, H. Garcia-Molina, S. Nestorov, V. Vassalos, R. Yerneni. Template-based Wrappers in the TSIMMIS System. *In Proc Twenty-Sixth SIGMOD*, 1997.

[20] K. A. Hawick, H. A. James, and J. A. Mathew., Remove Data Access in Distributed Object-Oriented Middleware, *Parallel and Distributed Computing Practices*, 2000.

[21] myGrid: mygrid.man.ac.uk.

[22] Nimbus: http://workspace.globus.org/

[23] NCBI: http://www.ncbi.nlm.nih.gov/

[24] N. W. Paton, R. Stevens, P. G. Baker, C. A. Goble, S. Bechhofer, and A. Brass. Query Processing in the TAMBIS Bioinformatics Source Integration System, *Proc. SSDBM'09.*

[25] RENCI's Science Desktop: https://portal.renci.org/RENCIDesktop/

[26] R. A. Coyne and R. W.Watson. The parallel i/o architecture of the high-performance storage system (hpss). In *Proceedings of the IEEE MSS Symposium*, 1995.

[27] R. Alonso-Calvo, V. Maojo, H. Billhardt, F. Martin-Sanchez, M. García-Remesal, D. Pérez-Rey. An agent- and ontology-based system for integrating public gene, protein, and disease databases. Journal of Biomedical Informatics, 40(1):17-29, 2007.

[28] W. Sujanski. Heterogeneous Database Integration in Biomedicine. Journal of Biomedical Informatics, 34(4):285-298, 2001.

[29] TeraGrid: https://www.teragrid.org/

[30] TeraGrid-ORNL: http://www.ornl.teragrid.org/guides/userguide.html

[31] The Life Science Grid Research Group: www.gridforum.org

Appendix

The following are descriptions of the nine Query Types shown in figures 1 and 2 as well as the sequence of computational steps required for processing the Query Types:

Q_1: Answering this Query Type requires two searches. The first is to find information about each gene on the cytogenetic band from a database such as the Genome Database (GDB). The second search is for finding information about which gene is a homolog of other gene using a database such as Entrez.

Q_2: First, the sequence that corresponds to the unknown DNA fragment is processed through a translator such as Transeq. The subsequent step is the identification of the correct reading frame by searching a composite database such as OWL, and using each of the six possible reading frames as an input until a hit is found. Once the appropriate peptide is identified, the corresponding sequence is used to search a protein database in order to retrieve the entire protein sequence. In the 4th step, the protein family is identified by using a homology search service such as PSI-BLAST. The 5th step consists of obtaining the function of the sequence by searching several databases such as Prosite to locate information about the target protein.

Q_3: First, the proteins with the required protein name are selected from a database. Second, only the entries that have antigen function and apoptosis process are selected. Finally, the resultant sequences are passed to a database for the scanning of their motifs.

Q_4: After the sequence that corresponds to the DNA sequence is processed through a translator and the correct reading frame is identified, the corresponding sequence is used to search a protein database for retrieving the entire protein sequence. Then a database such as PDB-BLAST is searched for retrieving full-length sequence of best matches.

Q_5: Consider for example that a pharmaceutical researcher wants to know the list of proteins similar to the ones involved in HIV. The HIV virus itself is composed of two RNA strands encased in a protein envelope. The viral envelope has 2 proteins, named gp120 and gp41. First, a database such as PDB can be used to find the structure of gp120 and gp41. Then the resulting sequences are submitted to multiple similarity matching sources.

Q_6: In the first step, the homologous protein sequences are determined through BLAST search, and an interested protein is selected. In the second step, a database, such as PIRPSD is queried for the basic information of the selected protein. In the third step, the proteins having interaction with the selected protein are determined by querying a database such as DIPDB.

Q_7: Consider for example the query "find all genes in the human genome that have *TTGGACAGGATCCGA* followed by *GCCGCG* within *40* symbols in a *4000* symbol stretch upstream of the gene". First BLAST is performed to locate all instances of the two query patterns on the human genome. Then, the results of these matches are combined to find all pairs that are within 40 symbols of each other. Next, a gene database is consulted to check if this match is in the region upstream of any known gene.

Q_8: First, BLAST search is performed for determining the protein's sequence. Then, the proteins that are homologous to the given protein are determined. Finally, the function of the given protein is determined.

Q_9: First, BLAST search is performed for determining the protein's sequence. Then, a database such as PROSITE is searched for the protein's motifs and domains.