Processing Scientific Mesh Queries in Graph Databases

Alireza Rezaei Mahdiraji
Jacobs-University
Bremen, Germany
a.rezaeim@jacobs-university.de

Peter Baumann
Jacobs-University
Bremen, Germany
pbaumann@jacobs-university.de

ABSTRACT

In this work-in-progress paper, we model scientific meshes as a multi-graph in Neo4j graph database using the graph property model. We conduct experiments to measure the performance of the graph database solution in processing mesh queries and compare it with GrAL mesh library and PostgreSQL database on synthetic and real mesh datasets. The experiments show that the databases outperform the mesh library. However, each of the databases perform better on specific query type, i.e, the graph database shows the best performance on global path-intensive queries and the relational database on local and field queries. Based on the experiments, we propose a mediator architecture for processing mesh queries by using the three database systems.

Categories and Subject Descriptors

H.2 [**Database Management**]; H.2.1 [**Logical Design**]: Data models; H.2.4 [**Systems**]: Query processing

Keywords

CW-Complex; Data Model; Geometry; Graph Database; Mediator; Query Processing; Scientific Mesh; Topology

1. INTRODUCTION

Scientific meshes (a.k.a unstructured meshes or just meshes) subdivide a domain into simpler geometric elements called *cells* which are glued together by incidence relationships. Such subdivisions allow more accurate computations of complex physical domains by providing approximate physical object representations. Finite element analysis, medicine, seismology, solid modeling, oceanography, climate modeling, and GIS are some of the domains which use meshes. Meshes in these domains have millions of cells with large scale numeric datasets which associate data values to each cell.

Currently, no database vendor offers build-in mesh functionalities and the main mesh manipulation tools are mesh libraries. Mesh libraries need deep procedural programming knowledge, do not scale with dataset size, and applications based on them are costly to maintain.

In our previous work, we proposed Incidence multi-Graph Complex (ImG-Complex) model which represents mesh topology as a multi-graph [10] [11]. In this paper, we show how the ImG model can be implemented in graph databases such as Neo4j. Then, we describe how some of the common mesh queries can be expressed as graph queries. Then, we compare the performance of graph-based solution for meshes with GrAL mesh library and PostgreSQL relational database. We report query response time, memory usage, and disk usage of each solution by executing the sample mesh queries on synthetic and real mesh datasets. The results show that graph databases are good for special types of mesh queries called path-intensives, i.e., the use of global mesh information, but they perform poorly on local mesh queries. Finally, we propose a mediator system architecture for mesh queries which combines the strengths of the three systems.

This paper is organized as follows: Section 2 introduces mathematical concepts, and Section 3 discusses related work. In Section 4 we show how to model ImG model in Neo4j graph database. Section 5 presents experimental results comparing GrAL, Neo4j, and PostgreSQL performance on mesh queries and constraints. Section 6 concludes the paper.

2. MATHEMATICAL CONCEPTS

The mathematical root of meshes is the concept of CW-complexes from algebraic topology. Informally, CW-complexes describe how a set of cells are topologically connected together. In the sequel, we informally and briefly introduce some of basic related concepts. We refer the interested reader to topology references [5].

A mesh of dimension d is the union of *k-cells* ($0 \leq k \leq d$) which are glued together by the *side-of relationship*. 0-cells, 1-cells, 2-cells, and 3-cells are known as vertices, edges, faces, and bodies. Triangles (2-cells), tetrahedrons (3-cells), and other polyhedrons are typical examples of cells which are used in practice. The set of cells of a mesh must satisfy certain conditions e.g., the intersection of two cells is either empty or another cell of the mesh (of lower dimension). The *boundary* of each cell is formed of cells of lower dimension.

The *side-of relationship* determines the topological structure of a mesh. It defines a partial order on the mesh, which in well-behaved meshes is *graded*, i.e., the boundary of a k-cell is made of $(k-1)$-cells. In that case it can be depicted by a Hasse diagram (a.k.a *incidence graph*).

WWW'14 Companion, April 7–11, 2014, Seoul, Korea.
ACM 978-1-4503-2745-9/14/04.
http://dx.doi.org/10.1145/2567948.2580061.

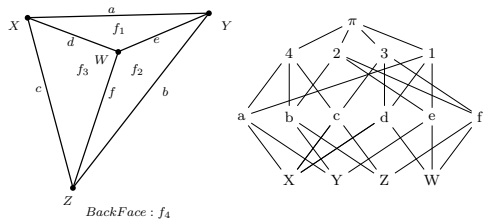


Figure 1: A tetrahedron and its corresponding incidence graph.

Table 1: Examples of side-of relationships of tetrahedron in Figure 1.

$\mathbf{0-cell \prec 1-cell}$	$\mathbf{1-cell \prec 2-cell}$	$\mathbf{2-cell \prec 3-cell}$
$X \prec a$	$a \prec f_1$	$f_1 \prec \pi$
$X \prec d$	$a \prec f_4$	$f_2 \prec \pi$
$X \prec c$	$b \prec f_2$	$f_3 \prec \pi$
$Y \prec a$	$b \prec f_4$	$f_4 \prec \pi$
$Y \prec b$	$c \prec f_3$	
...	...	

In Figure 1 (left), the mesh consists of a single tetrahedron π, four faces f_i, six edges (i,.e., $a, b, \ldots, f$), and four vertices X, Y, Z, W. The intersection of f_3 and f_2 is the edge f and the vertices Z and W, and the boundary of f_3 is formed by the edges f, d, e and the vertices X, Z, W, or, equivalently, we say that f, d, e and X, Z, W are the *sides* of f_3, in a short notation expressed as $e \prec f_3$ (read "e is a side of f_3"). We also say that f_3 and its sides are *incident*, and we call f_3 and f_2 *adjacent* because they share a common edge. In Figure 1 (right), the cells are ordered into layers of equal dimension and the side-of relationship is shown only between cells of dimensions differing by 1.

In contrast to structured meshes (represented as arrays), unstructured meshes need to explicitly store topology because cells can have a different number of neighboring cells. Unstructured meshes have three main components, namely, topology or combinatoric structure (which is determined by side-of relationship), geometry (which describes geometric embeddings of cells and the mesh), and data values or fields (which assign data values for all or some of cells). Many meshes have simple linear geometry which can be specified by listing coordinates of vertices. Table 1 and Table 2 represent part of topology of tetrahedron as side-of expressions and an example field of Figure 1, respectively.

3. RELATED WORK

Mesh libraries such as CGAL [1] and GrAL [7] are dedicated libraries which include mesh algorithms and can be run on mesh representations. The libraries are written in C++ and often use low-level file-based I/O APIs. In comparison to CGAL, GrAL has the more generic approach to meshes and it can express virtually any combinatoric query using domain specific language of iterators [6][1]. GrAL provides many mesh primitives to reduce amount of programming in C++. Processing mesh queries based on the libraries requires deep C++ programming knowledge. Such query implementations rely highly on input file structure and internal mesh representation, i.e., any changes in the structure require changes in the implementation. Thus, the implementations are less reusable and costly to maintain. Furthermore, mesh libraries do not have high-level query language and do not scale with dataset size (i.e., are bounded to memory size).

Table 2: Example pressure field for faces of tetrahedron in Figure 1.

Face	Pressure (kPa)
f_1	0.26
f_2	0.36
f_3	0.29
f_4	0.31

Although the relational model does not have sufficient abstraction form meshes, it can represent mesh domains. In [9], the authors show how to implement a new efficient indexing technique for tetrahedral meshes called Directed Local Search (DLS) in Microsoft SQL Server 2005 using C# within a relational mesh schema specific for tetrahedral meshes [8]. It is worth mentioning that there is no official support in SQL Server for meshes.

Incidence multi-Graph Complex (ImG-Complex) extends the incidence graph model by supporting multi-incidence relationships. Incidence Graph (IG) is a directed graph in which each node is a cell and each edge shows a side-of relationship. Multi-incidence relationships cannot be represented by simple incidence graph . ImG-Complex is a multi-graph data model, i.e., allows multiple links in the incidence graph. It can represent a wide range of real mesh domains. The ImG-Complex model has sets of constraints to limit the model to smaller object classes or geometric representations based on the properties of the object classes such as manifold and pseudo-manifold [11].

4. MODELING MESHES IN NEO4J GRAPH DATABASE

A natural abstraction for the ImG-complex model is the property graph data model used by graph databases. Graph databases are NoSQL databases that use graph structures to represent objects and the relationships between objects. They represent data by nodes, links, and properties. Nodes represent objects and links show the relationship between nodes. A typical graph data model contains two main components: 1) nodes with properties, and 2) named relationships with properties. Some graph models also contain hypergraphs [4].

In comparison to relational databases, data definition and manipulation in graph databases can be done either using declarative graph query languages or APIs. The latter has unlimited expressive power for querying but has a lower level of abstraction for users [3].

A popular open source graph database is Neo4j. It is under AGPLv3 license and has been operational since 2003. Neo4j is schema-less, i.e., it allows modeling of complex and dense graph domains. It supports ACID properties, and can do high performance graph operations [2]. Neo4j's data model is property multigraph, i.e., data is stored in nodes and relationships of a multi-graph with pairs of key-value properties. Neo4j's query language is called Cypher. In Cypher, the user specifies the starting point and the desired outcome using graph pattern matching. Graph traversal is done using patterns. A pattern has a starting point(s), one or more paths; a path being a sequence of nodes and relationships that always start and end in nodes. For instance, path (a)–>(b) is a path starting from node "a", with an outgoing link from it to node "b". We refer the reader to Neo4j documentation for further details on Neo4j and Cypher [2].

A 3D linear mesh domain can be modeled using property multi-graph. There are four types of nodes (i.e., vertex, edge, face, and body) and one type of relationship (i.e., side-of) which connects these nodes. Each node has its own properties and can have n field values. Geometry is modeled as simple data values, i.e., coordinates as properties of nodes. Figure 2 shows a property graph model for 3D linear mesh domain.

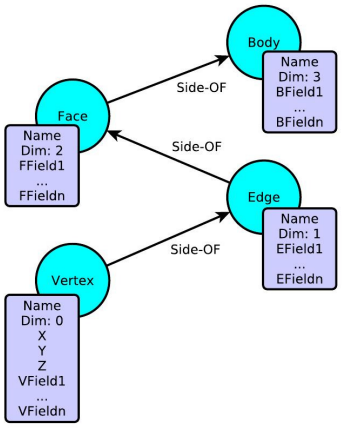


Figure 2: Property graph data model for 3D linear meshes.

In the sequel, we show how to formulate two recurrent mesh queries in Cypher query language.

Q1. Get list of cells which a particular cell is side-of. This query has several variations. One variation is to retrieve the list of cells which a given vertex "v" is side-of. The following Cypher statements express the query:

```
START r=node(0), v=node(12)
MATCH r--f--e--v
RETURN r,f,e
```

The query matches the pattern r–f–e–v for interval (v,r] (r is the root). This pattern specifies all higher dimensional cells for which vertex "v" is side-of. The result set is all intermediate edges, faces, and 3-cells which "v" is side-of. Symbol "--" shows the unidirectional relationship and node(0) refers to root of the graph.

Q2. Iterating over neighbors (adjacent) of a cell. This query also has several variations. The following Cypher statements extracts adjacent edges to edge "a" which share a common vertex:

```
START a = node(20)
MATCH a-[:side-of]->v<-[:side-of]-e
RETURN e;
```

"side-of" is the type of relationship between nodes of the graph. The pattern has two parts. The part a-[:side-of]->v extracts all the vertices of edge "a". The parts v<-[:side-of]-e finds all the edges which the vertices are side-of. The set "e" does not contain edge "e". Note that here we used directed relationships, because the direction matters.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

Experimental Design. We are interested in evaluating the performance of three systems, namely, GrAL, PostgreSQL, and Neo4j graph database on querying meshes. The goal is to *evaluate performance of mesh libraries, graph databases, and relational databases for processing mesh queries.* We also report peak memory usage and disk usage of the systems for each dataset and compare simplicity of query formulation. We selected five queries used frequently in mesh domains (we saw two of them in Section 5) querying different components of meshes, i.e., topology, geometry, and fields.

Table 3: Number of cells and side-of relationships in each synthetic and real dataset.

Dataset	Number of Cells	Number of Side-Of
D1	39602	88605
D2	159202	357205
D3	358802	805805
D4	638402	1434405
D5	998002	2243005
D6	1744506	3231605
Cow	278594	650048
Horse	290895	678748
Stanford Bunny	432143	1008322

Implementation Details. We run the experiments on a system with four cores (3.2 GHz processor) with 4GB of RAM and XUbuntu 12.10 operating system. We use GrAL as of 1.10.2013, Neo4j 2.0.0, and PostgreSQL 9.1. GrAL is compiled using gcc 4.6.3 with setting -ftemplate-depth-200 which controls depth of template instantiation. We run each query ten times, each time under cold systems condition, i.e., we empty all the system caches and stop all system daemons processes. Finally, we restart the database servers before each query execution. The response time and memory usage are averaged over ten trials for each pair of (query,dataset).

The Neo4j database uses the property graph in Figure 2 up to dimension 2 (i.e., faces) to model the 2D synthetic mesh datasets. We create a least upper bound (LUB) m object such that all faces are side-of of m. If considered as the whole mesh object it can make graph pattern formulation easier. Indexes on all properties are built. The relational schema in PostgreSQL contains two tables as follows:

cell(id,name,dimension)
incidence(from,to)
coordinates(vid,x,y)
vfield(vid,pressure)

The incidence table contains vertex-to-edge, edge-to-face, and face-to-body relationships. PostgreSQL uses B-tree indexes on all the columns.

We used Cypher (for Neo4j queries), C++ language (for GrAL queries), and SQL (for PostgreSQL queries) for implementing the queries.

Datasets. We used both synthetic and real datasets, i.e., six synthetic datasets and three real datasets. The synthetic datasets are generated using a synthetic generator for Cartesian meshes. It defines a vertex and a 2-cells $((i,j), (i+1,j), (i+1,j+1), (i,j+1))$ for each (i,j). A simple field function is defined for each vertex: $f_v(v_x, v_y) = v_x + v_y$, i.e., the field of a vertex is the sum of its x and y coordinates.

The real datasets are Stanford Bunny (courtesy of Stanford University), Cow, and Horse are in *tri* file format.

Table 3 shows the number of cells and side-of relationships in each real and synthetic dataset. The datasets are converted to appropriate formats to be imported to GrAL, Neo4j, and PostgreSQL. The bulk load of PostgreSQL works much faster than import mechanism of Neo4j.

5.2 Query Evaluation

In this section, we show how five mesh queries can be implemented in GrAL, Neo4j, and PostgreSQL and we report

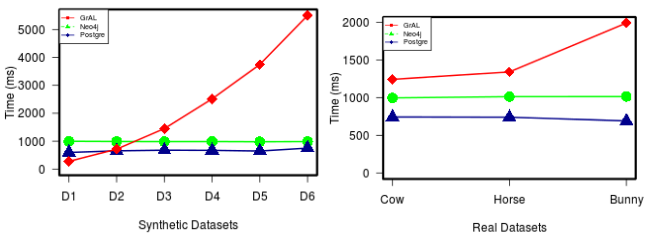
the performance of each system, readability of the query, disk usage, and peak memory usage per query.

Q1. In Section 5, we saw this query in Cypher language. The following listing shows the equivalent SQL query. The SQL query joins three instances of *incidence* relation to build the path between vertices to the root mesh element. It has another join to *cell* to specify "v" as the vertex.

```
SELECT i2.from, i3.from, i3.to
FROM cell c JOIN incidence i1
    ON c.id = i1.from
  JOIN incidence i2 ON i1.to = i2.from
  JOIN incidence i3 ON i2.to = i3.from
WHERE c.name='v0'
```

It can be seen that the graph query is shorter and more intuitive than the SQL query and GrAL C++ API.

Figure 3 presents response time of query **Q1** on the three systems. The x-axis shows the list of datasets: first the six synthetic datasets (ordered based on their sizes) and the three real datasets (also ordered by their size). We observe that response time of PostgreSQL is faster than GrAL and Neo4j. The GrAL performs well on small datasets but its response time linearly increases with dataset size and on bigger datasets it is the slowest system in comparison to the databases. Furthermore, response time of the databases systems is almost constant over the dataset size. The performance of PostgreSQL does not decrease on joining bigger tables. The reason is that our current datasets are not big enough datasets to emphasize this point. The same trend can be observed on both synthetic and real datasets.



(a) (b)

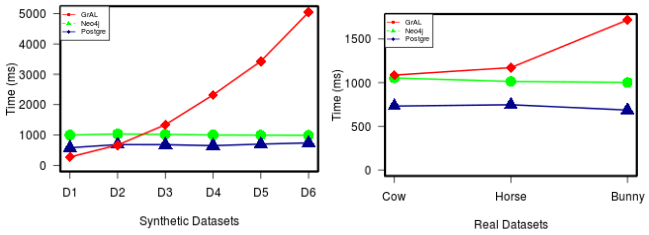
Figure 3: a) Performance of Q1 on GrAL, Neo4j, and PostgreSQL on synthetic datasets and on b) real datasets.

Q2. In Section 5, we saw this query in Cypher language. The SQL query for **Q2** is as follows:

```
SELECT c.name
FROM cell c
WHERE id in (
        SELECT i2.to
        FROM incidence i2, incidence i1, cell c1
        WHERE i2.from = i1.from
                and c1.name='e10' and c1.id=i1.to)
```

The query contains a sub-query which joins two instances of *incidence* with *cell* to find all edges which share one vertex with edge "a". Then, the outer query retrieves the names of those edges from the *cell*. Also in this case, the graph query is simpler and easier to understand. Figure 4 depicts response time of **Q2**. PostgreSQL is the fastest system and GrAL is the slowest.

Q3. The query is known as *graded constraint* which verifies if every maximal chain in the mesh graph has the same length [11]. Because the GrAL data structure is graded by



(a) (b)

Figure 4: Performance of Q2 on GrAL, Neo4j, and PostgreSQL on synthetic datasets and on b) real datasets.

construction, we only compare Neo4j and PostgreSQL for the **Q3**. The SQL query for the constraint is as follows:

```
SELECT count(*)
FROM incidence i3
  LEFT JOIN incidence i2 ON i3.from=i2.to
  LEFT JOIN incidence i1 ON i2.from=i1.to
WHERE i3.to = 0
  AND (i3.to IS NULL OR i3.from IS NULL
  OR i2.from IS NULL OR i1.from IS NULL)
```

Three instances of *incidence* are left joined to capture all paths from the root to vertices. Left join allows the capture of paths with length less than three, e.g., an isolated edge which is not connected to any faces and is directly linked to the root. The WHERE clause counts all rows which contains at least one NULL value, i.e., a path with length less than three. The Neo4J query is:

```
START b = node(0)
MATCH p = b-[:side-of*1..2]->v
WHERE v.dimension = 0
RETURN count(p);
```

The graph query is much simpler than the SQL one: pattern [:side-of*1..2]->v captures all paths of length one or two between root cell and cell "v" where "v" is conditioned to be a vertex. If such path exists it means that the graph is not graded.

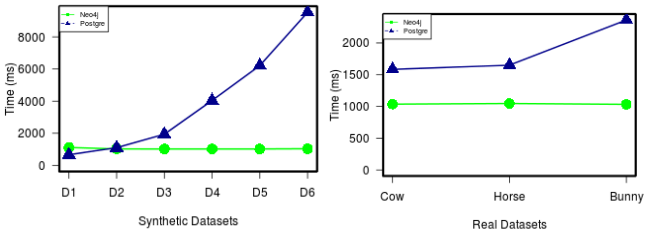
Figure 5 illustrates response time of **Q3**. Neo4j outperforms PostgreSQL in checking graded constraint. The performance of PostgreSQL decreases rapidly with database size. This query needs to check the length of all the paths from the root element (the mesh itself) to all vertices and make sure they are all of the same length. This is where the true power of a graph database such as Neo4j resides, i.e., path-intensive queries. It must be noted that both systems have low variance for response time over the trials which suggests the experiments' environment is stable. Overall, PostgreSQL has lower variance in comparison to Neo4j.

Q4. Given a face f, it retrieves coordinates of all its vertices. The Neo4j query for is listed below:

```
START f=node(53)
MATCH f--e--v
RETURN collect(distinct v.x),
        collect(distinct v.y);
```

The equivalent SQL query is as follows:

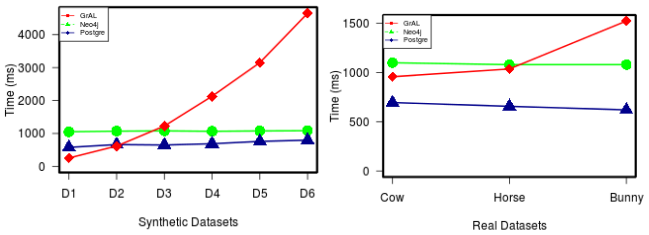
```
SELECT DISTINCT vertex_id,x,y
FROM coordinates AS co, cell AS c,
        incidence AS i1, incidence AS i2
WHERE i1.to = i2.from AND i2.to=c.id
        AND c.id=1
        AND co.vertex_id = i1.from
```



(a) (b)

Figure 5: Performance of Q3 on GrAL, Neo4j, and PostgreSQL on synthetic datasets and on b) real datasets.

Figure 6 illustrates response time of **Q4**. PostgreSQL outperforms GrAL and Neo4j on this simplest geometric query. We can see small deviations in the databases performance when dataset sizes grow. This is due to high variance in execution time over trials. However, the databases have almost constant performance over datasets. We observe that retrieval of numeric data from the graph database is less efficient than the relational database.



(a) (b)

Figure 6: Performance of Q4 on GrAL, Neo4j, and PostgreSQL on synthetic datasets and on b) real datasets.

Q5. It returns the number of vertices in which their field value is greater than a given threshold t. The Neo4j query is as follows:

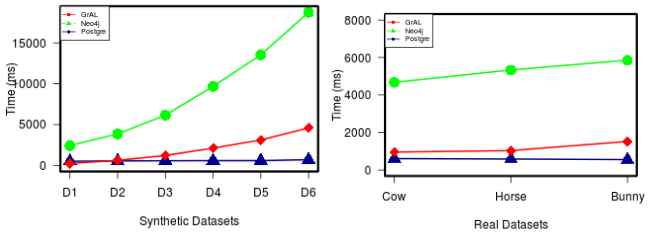
```
START v=node(*)
WHERE v.dim=0 and v.field > t
RETURN count(v);
```

The SQL query is as follows:

```
SELECT count(*)
FROM vfields
WHERE field > t
```

Figure 7 shows response time of **Q5**. Response time of Neo4j drastically increases with dataset size and has the worst performance, GrAL performance as before is linear w.r.t. dataset size. PostgreSQL shows the best performance and its time increases only slightly with dataset size. However, on real dataset the trend is different and the smallest dataset (i.e., cow) has the biggest time. The reason is that the result set of other two datasets for **Q5** is zero while the result set of Cow is 3462. This query shows that the graph database is not efficient in computing field functions.

Disk Usage. Figure 8 illustrates disk size (in megabytes) of each dataset on the systems. On average Neo4j and PostgreSQL disk spaces are 38 and 21 times bigger than GrAL. The disk usage of Neo4j is bigger than other systems, because graph databases store all the related relationships at



(a) (b)

Figure 7: Performance of Q5 on GrAL, Neo4j, and PostgreSQL on synthetic datasets and on b) real datasets.

each node while in RDBMS the structure is defined in table level. GrAL has the smallest sizes because it constructs the required data structure in memory.

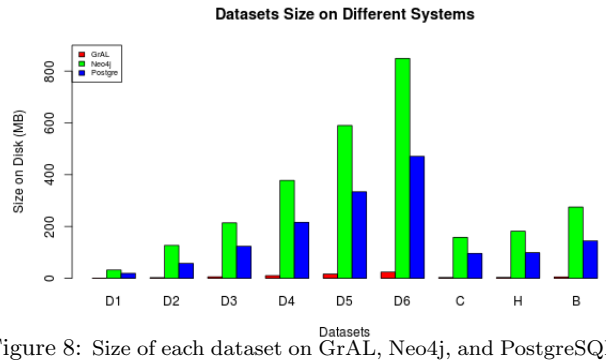
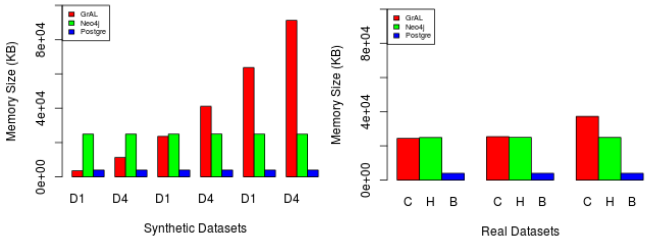


Figure 8: Size of each dataset on GrAL, Neo4j, and PostgreSQL.

Memory Usage. Figure 9 shows maximum resident size of the three systems in the memory during the execution of **Q2**. GrAL uses more memory in comparison to other systems and the memory usage increases with dataset size. The reason is that GrAL loads the datasets into memory and build data structure. The database systems use less memory and the memory usage does not increase with datasets' size. The main reason is that databases load indexes into memory instead of the whole datasets. PostgreSQL has the smallest memory usage. Similar patterns are observed for real datasets and other queries.



(a) (b)

Figure 9: Performance of Q5 on GrAL, Neo4j, and PostgreSQL on synthetic datasets and on b) real datasets.

5.3 Summary and Discussion

Based on the experiments, we conclude that PostgreSQL performs well on local mesh queries (e.g., **Q1** and **Q2**) and field queries (e.g., **Q5**). With the exception of **Q3**, the Post-

greSQL performance does not deteriorate when join operations are on the incidence table mainly because the join operation performance of RDBMS depend on the number of tuples involved in the join which is very small in case of **Q1** and **Q2**.

Neo4j performs better on global path-intensive queries (e.g., **Q3**), but it performs very poorly on local queries (e.g., **Q2** and **Q4**) and global field queries **Q5**. Constant performance on **Q1**, **Q2**, **Q3**, and **Q4** is because graph database performance depends on degree of each node which is very small value in the mesh applications. Neo4j uses the most disk space and its main memory usage is very close to GrAL.

Because of file I/O and in-memory construction, GrAL is very fast on small dataset and does not scale with the dataset size. In case of **Q5**, GrAL outperforms Neo4j. GrAL uses less disk size but its main memory usage is more than the databases.

In almost all the experiments, the graph queries are more concise and readable w.r.t. the semantics of the queries than the SQL queries. GrAL implementations are the most difficult ones.

In brief, the experiments suggest that to achieve high performance, global topological mesh queries must be executed on graph databases and local topological queries and field queries on relational engines. Since neither graph databases nor relational databases provides advanced geometric algorithms, we need a mesh library with geometric algorithms.

We believe to efficiently process all types of the mesh queries we need to combine the strengths of the relational and graph databases together with a mesh geometric library. We can integrate the three systems using a mediation software. A *mediator* resides between user query and different data sources. It receives a user query and finds an efficient execution plan for it. It translates the query and extracts the relevant data from each data source. Finally, it merges the results from sources into a single result set [12].

In the case of a mesh mediator system, the graph database stores only topological mesh structures and the relational database stores all three components of a mesh. We envision the architecture in Figure 10 for a mesh mediator system.

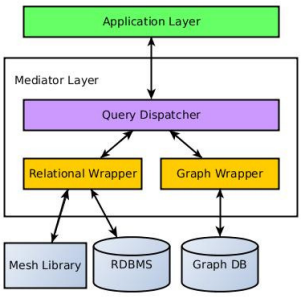


Figure 10: Architecture of a mediator system for processing mesh queries.

The user submits her/his queries using a high-level declarative mesh language. The query language is designed based on mesh data models such as ImG-Complex. The mediator software analyses the user queries and dispatches them based on their types to either graph wrapper or relational wrapper, e.g., **Q3** to graph wrapper and **Q2** to relational wrapper. A *wrapper* translates a query into source specific queries and converts the data returned by the data sources into the global data model, i.e., mesh data model. Each data source has one wrapper. For instance, in the Figure 10, the graph wrapper translates a user query to a graph query. The relational wrapper is more complex and uses both the RDBMS and the mesh library for query translation.

Although such mediator scenario offers better performance and scalability, it stores topological information redundantly and may not be suitable for small mesh datasets.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we modeled scientific meshes using property multigraph model of graph databases. We conducted experiments which show pros and cons of Neo4j, GrAL, and PostgreSQL in querying meshes. The experiments show Neo4j graph database is efficient for path-intensive queries, but not for local, geometric, and field queries .

In future, we want to pursue the idea of the mesh mediator system. First step to this goal is to design a declarative mesh query language. We also want to define a mesh benchmark and experiment with big mesh datasets.

7. ACKNOWLEDGMENTS

This work has been funded by the EU FP7-INFRA project EarthServer. The authors would like to thank Dr. Guntram Berti for his help with GrAL library.

8. REFERENCES

[1] CGAL, Computational Geometry Algorithms Library. Viewed December 2013.

[2] Neo4j - the world's leading graph database, Viewed December 2013.

[3] R. Angles. A comparison of current graph database models. In *ICDE Workshops*, pages 171–177, 2012.

[4] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1:1–1:39, Feb. 2008.

[5] G. Berti. *Generic software components for Scientific Computing*. PhD thesis, BTU Cottbus, 2000.

[6] G. Berti. A generic toolbox for the grid craftsman. In *Proceedings of the 17th GAMM-Seminar Leipzig*, pages 1–28. Citeseer, 2001.

[7] G. Berti. Gral - the grid algorithms library. *Future Generation Computer Systems*, 22, 2006.

[8] G. Heber and J. Gray. Supporting finite element analysis with a relational database backend, part i: There is life beyond files. *CoRR*, abs/cs/0701159, 2007.

[9] G. Heber and J. Gray. Supporting finite element analysis with a relational database backend, part ii: Database design and access. *CoRR*, abs/cs/0701160, 2007.

[10] A. R. Mahdiraji and P. Baumann. Database support for unstructured meshes. *Proc. VLDB Endow.*, 6(12):1404–1409, Aug. 2013.

[11] A. Rezaei Mahdiraji, P. Baumann, and G. Berti. Img-complex: graph data model for topology of unstructured meshes. In *Proceedings of the 22nd ACM international conference on CIKM*, pages 1619–1624. ACM, 2013.

[12] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, 1992.