Charles University

Faculty of Mathematics and Physics

BACHELOR THESIS

Eldar Guliyev

Comparative Analysis of Multi-model Databases

Department of Software Engineering

Supervisor of the bachelor thesis: doc. RNDr. Irena Holubová, Ph.D.

Study programme: Computer Science

Study branch: IOIA

Prague 2022

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In . . . . date . . . .

Author’s signature

Title: Comparative Analysis of Multi-model Databases

Author: Eldar Guliyev

Department: Department of Software Engineering

Supervisor: doc. RNDr. Irena Holubová, Ph.D.

Abstract:

The thesis is devoted to performance analysis of multi-model database management systems. Data models, multi-model DBMS and query languages were studied. Based on comparison of existing database benchmarks and multi-model DBMS functionality, requirements to the benchmarking process were identified. For the performance benchmarking, a cross-platform benchmarking application with graphical user interface was designed and implemented. The benchmarking application has a plugin architecture giving the possibility to create a DLL-plugin and test a DBMS which is not supported in the initial release. ArangoDB, RavenDB and MongoDB were tested with focus on document and graph data models.

Keywords: multi-model database management systems, benchmark, ArangoDB, RavenDB, MongoDB

Content

Introduction 5

1. Multi-Model DBMS 8

1.1 Data models 8

1.2 DBMS and MMDBMS 11

1.3 Review and comparison of MMDBMSs 13

1.4 Detailed review of ArangoDB, RavenDB and MongoDB 17

1.4.1 ArangoDB 18

1.4.2 RavenDB 19

1.4.3 MongoDB 22

Results of Chapter 1 23

2. Design and implementation of the benchmarking application 24

2.1. Introduction to benchmarking 24

2.2 Requirements of the benchmarking application 27

2.3 Benchmarking process 29

2.4 Application architecture 30

2.4.1 Abstract component 31

2.4.2 DBMS plugin 32

2.4.3 UI component 33

2.4.5 Runner component 36

2.5 Plugin development 37

Results of Chapter 2 38

3. User documentation 39

3.1 Installation 39

3.2 Benchmark start 39

4. Experiments 43

4.1 Dataset description 43

4.2 Execution environment 44

4.3 Workload 44

4.3 Results discussion 56

Conclusion 58

References 59

Appendix A. Data scheme 64

Appendix B. Jupyter notebook for data visualization 65

Appendix C. Zip-package 66

Introduction

Data persistence is a key part of software functional requirements. Almost every software application saves its state between executions. At least, user settings are saved, at most, data entries, documents, binary attachments and other digital assets are stored for future editing and viewing. Data of a simple structure as a short list of key-value pairs could be saved in local text configuration files or binary files. Usually data of complex domains contains instances of interrelated entities. Storing and processing complex data requires the management of relations and constraints. If data is modified by more than one user or application instance simultaneously, a transaction mechanism and multi-thread server are needed to be implemented. As more data is accumulated, it is used for analysis and report generation. Aggregations, projections, joins of collections of entity objects, filters are operations, implementation of which must be fast and memory efficient. This explains the demand for reusable and well-designed executable software components for data access and management. Database management systems (DBMS) are components of software systems that expose a standard interface for data storage and access, implement commonly used functionality for data queries and maintain data integrity.

A software project starts from the identification and analysis of requirements which is followed by data model design of the domain area. When the domain model is created, a software architect chooses a suitable DBMS for a software project. A software architect takes into account the nature and the structure of the data, the functional and non-functional requirements and other restrictions: execution environment, programming languages, the experience of software developers with particular DBMS, price, maintenance period, functionality, scalability, performance.

Performance is a key software quality attribute that influences the success of a software project. As a DBMS is used as-is – without code modification, its performance and other quality attributes are determined by an external team of developers. DBMSs have their own development lifecycles: newer versions could be developed and delivered once every two or more years. Security bug fixes are done and distributed quite quickly, while performance enhancements require systematic work on architecture and algorithms. On average, e.g. MariaDB Server has had one stable major release every year [1]. Or, Microsoft releases a new version of SQL Server once in about two years [2]. The overall security level of a DBMS could be evaluated by a review of security reports, but performance requires more detailed research. Firstly, the support of different index types, query optimization and

other features may be studied. Then developers could create software prototypes and test them with different DBMSs. While this approach allows one to get a rough estimate of performance, it seems that formal procedure for DBMSs bench-mark and comparison would give a solid basis for making decisions on software architecture and choice of DBMS.

Recently a multi-model DBMS (MMDBMS) – a new type of DBMS supporting several data models – has become popular among software developers. MMDBMSs may simplify the architecture of software and improve software quality attributes like adaptability and maintainability. MMDBMS can be used in areas where it is not possible to design and implement a concrete data model within one abstract pattern. Examples of such areas are social networks (users' profiles and relations) or user generated content, fraud and data leak detection (sequences of users' and clients' actions and collections of corporate documents), recommendation systems (history of user's actions and website pages). Performance characteristics of popular relational DBMS (Microsoft SQL Server, Oracle , PostgreSQL [3], MySQL/MariaDB, FireBird [4]) and tools for performance tuning and measurement (query plan viewers, performance monitors, missing index scanners, etc.) are published as certification exam preparation materials, books and online guides on official documentation portals. MMDBMSs lack such detailed documentation and tools. It is necessary to compare the functionality and architectures of MMDBMs and determine how the performance of different multi-model DBMSs varies.

This bachelor thesis is devoted to a comparative analysis of MMDBMSs and development of the benchmarking application for MMDBMS. The work splits into the following tasks:

1. Study of abstract data models and comparison of MMDBMSs, choice of DBMSs that are suitable for benchmarking and performance comparison (Chapter 1).
2. Study of benchmarking methods and algorithms, analysis of existing results of DBMS benchmarking. Requirements identification and analysis for benchmarking process. (Chapter 2)
3. Design of benchmarking procedure and its implementation in a software application (Chapters 2, 3, 4).
4. Obtaining data for performance benchmarking (Chapter 5).
5. Experiments: quantitative assessment of performance and comparison of DBMSs (Chapter 5).

1. Multi-Model DBMS

For DBMS analysis, it is necessary to study data models, compare features of DBMSs and describe in detail those which are relevant to multi-model functionality. Special attention will be paid to the features of query languages.

1.1 Data models

Abstract data model is a high-level model describing how data elements and their properties are identified, grouped and related to each other and what operations could be performed on them. Commonly recognised models are relational, graph, document, key-value and object.

Relational data model was described by E.F. Codd [5]. The model consists of relations with fixed schema (set of attributes) and the relational algebra – set of operations most important of which are join (product), union and projection. The schema (structure) of the data is known in advance in the relational model [5]. Relations contain tuples. Each tuple is a vector with a predefined number of components, their order and data type. The SQL language is a standard tool for data definition and manipulation in this model. According to the DBMS popularity rank [6], the relational model is frequently used by developers as the top four DBMSs support this model.

In the **document-oriented** approach data is stored as a collection of documents – uniquely identified self-describing data entries. Every entry is represented as a separate XML or JSON document. Typically, documents in a collection could have a non-fixed schema and contain different sets of attributes. The document data model allows us to implement aggregates [7] – data units that can store dependent elements inside them without references to other data collections (references to tables via foreign keys). In self-contained documents data items (containers and internal parts) are stored as they appear in the domain area. As the domain area evolves, the schema of documents may change, but they would still be stored in the same collection. A collection of business cards could be an example of it. Regardless of a business card format, the number of printed contact phones and emails on it, all cards with internal information (collection of contacts) on them are stored in the same collection. In document oriented DBMS query languages are often based on a mix of JavaScript and JSON syntaxes. As an example, MongoDB [8] or CoachDB [9] use JavaScript to specify operations on the data and JSON to filter, update, insert or delete data items.

Wide-column databases store data entries that contain groups (families) of column values. Each group of column values is accessed by the same key of a row (see Fig .1.1). Values of a column family are stored continuously on a physical disk [10]. Wide-column databases are similar to document-oriented databases in the way that rows may have different column families [11]. Wide-column databases are suitable for workloads like real-time analytics or write-heavy log operation of IoT applications [11].

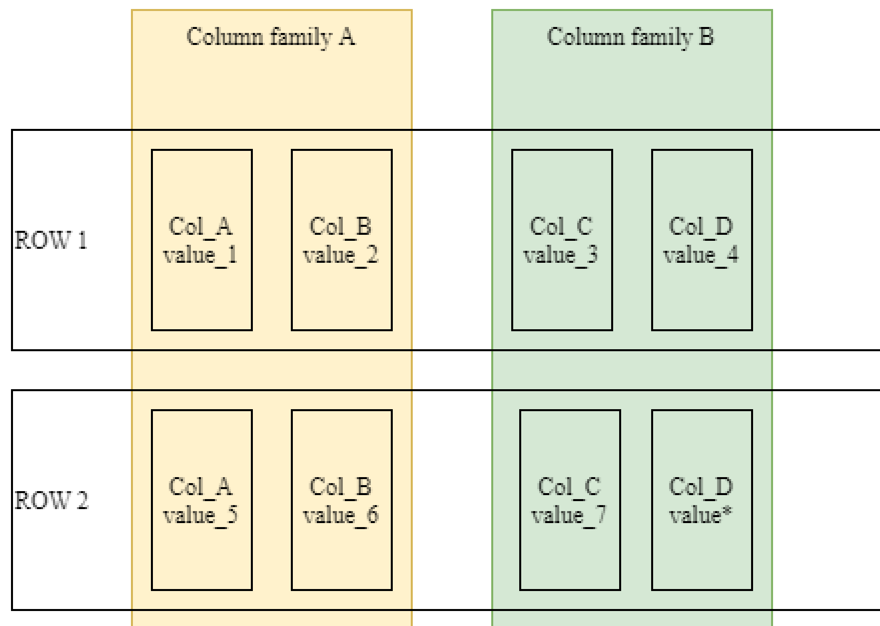

Fig. 1.1 – An example of wide-column data entries [11]

In **object-oriented** databases data entries are stored as class objects which have not only a state (values of fields – variables inside class object), but also a behavior (methods and properties with methods get and set) and may support inheritance. Objects are mutually related * by pointers. The main similarity with the relational model is that classes in object oriented programming languages like relations have a predefined schema. An object-oriented database is very bound to a programming language and execution environment because data needs to be restored into a set of objects in the memory of the software application process. When object data is accessed by one user (thread) only, it could be done by serialization and deserialization to local files [12]. At the time of writing this text, it was found that previously popular OODBMS db4o (134-th place in the DB-Engines Ranking [6]) was lastly updated on 22.09.2019. Another OODBMS objectbox (185-th place in the DB-Engines Ranking [6]) is in active development, but it supports only a limited list of programming languages: Java, Kotlin, C / C++, Python, Go, Dart / Flutter, Swift, EdgeX (very popular PHP and C# languages are not supported). Because of the above limitations, object oriented DBMS are not widely used.

Since about 2000-es a new object-oriented design pattern – **Object-Relational Mapping (ORM)** has been used by developers. ORM solutions are available for popular DBMSs and programming languages, including C# (Entity Framework, NHibernate), Java (Hibernate), Python (SQLALchemy), PHP (Eloquent). The pattern utilizes the fact that only state of objects need to be stored in the database, but behavior is statically coded. The pattern allows to load data from a traditional RDBMS and map it to dynamic lists of class objects. As a result, the state is loaded from a DBMS. The behavior of the objects is coded in methods of classes that represent the entities of the application domain. For data read an ORM library opens a connection to the DBMS, executes a SELECT query and creates an array or a dynamic list of entity class objects. Modified or newly created class objects are processed by an ORM library and transformed to CREATE or UPDATE queries to DBMS. The current demand for OODBMS among developers is almost fully eliminated by ORM libraries.

Key-value databases store data as an instance of an associative array. Each data element is accessed by its key. The value of a data element could be a string, a number or a document with complex structure, which nevertheless doesn't allow any internal documents inside it. Main difference from the document oriented model is that in key-value databases data elements are not grouped into containers like tables or collections. New data is added by inserting a new key into the associative array. Common use cases for key-value databases are caching, message queuing, and session management [13]. Key-value databases could contain pointers or descriptors to larger parts of data. Storing data in RAM, key-value DBMSs like Redis or the Memcached library offer exceptional performance by the price of relatively simple data structure which does not allow grouping entries in collections.

In **graph model** data is organized as graphs and manipulated with graph operations [14]. Graph data model is suitable for domains where entities have a lot of connections which do not fit fixed relation schema. An instance of a data object is represented as a node or a directed or undirected relation. Both nodes and edges could contain key-value properties. Unlike document databases, in graph databases data elements do not store complex documents with inner elements [15].

Regardless of the data model, entries in a database may be stored in **row oriented** or **column oriented** approaches [13]. Column oriented databases may follow the relational model, but have a different approach for storage of data: data is stored in columns where the order of the elements (column values) is respected in every column [13]. Column oriented approach is implemented in traditional row oriented DBMS as special indexes (columnstore indexes in Microsoft SQL Server) or may be used as the main approach for data storage

(ClickHouse, Apache HBase, Apache Cassandra [13]). Column oriented databases have better data compression because columns are compressed independently. Read queries with aggregation and projection operations execute faster making column oriented storage more suitable for analysis (OLAP) tasks. Row oriented storage is more preferable for OLTP tasks: data modification operations execute much slower in column oriented storage, because these operations require rebuilding of every column storage [13].

1.2 DBMS and MMDBMS

DBMS is a central component in corporate software systems. In [16] two types of client-server software architectures are described: client-DBMS (two-tier) and three-tier (DBMS – Application Server – Thin Client) architecture. In three-tier architecture (see Fig. 1.2) an intermediate component Application Server is used. The Application Server is responsible for business logic and interaction with one or more DBMSs.

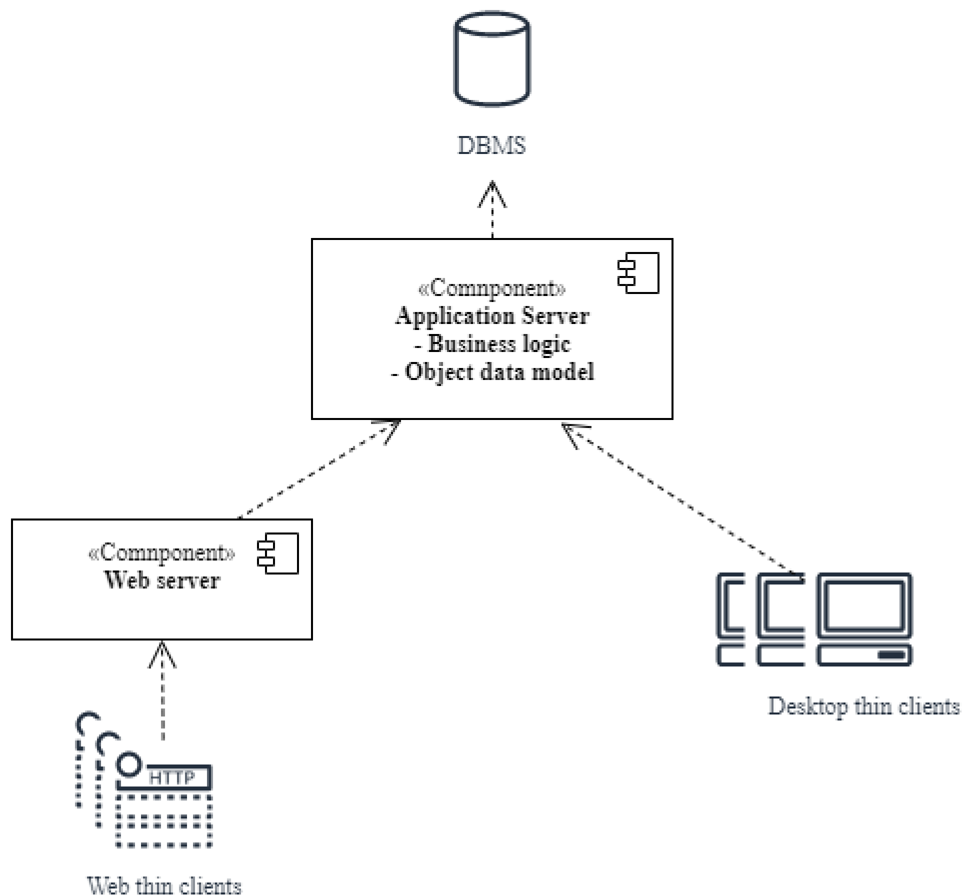

Fig 1.2 – Three-tier architecture [16]

With the growth of social media, digitalisation of the economy and development of machine learning and AI, large amounts of data are generated and processed daily. The data can not fit one particular data model and requires different approaches for storage and processing. Multi-model DBMS, which support more than one data model, could decrease efforts for software development, deployment and maintenance. Traditional software architecture (see Fig. 1.3) has the central component (Application Server) which interacts with different DBMSs, aggregates data and returns it to clients. Developers have to implement an aggregation process from multiple data sources and subsystems for every DBMS. Besides aggregation, implementation of security requirements is complicated by necessity to store users and roles in one DBMS and control access to other data sources on the Application Server level. Data sources are accessed under the application server account which could decrease security. If data sources are accessed with a user's account, all user accounts must be synchronized for every data source.

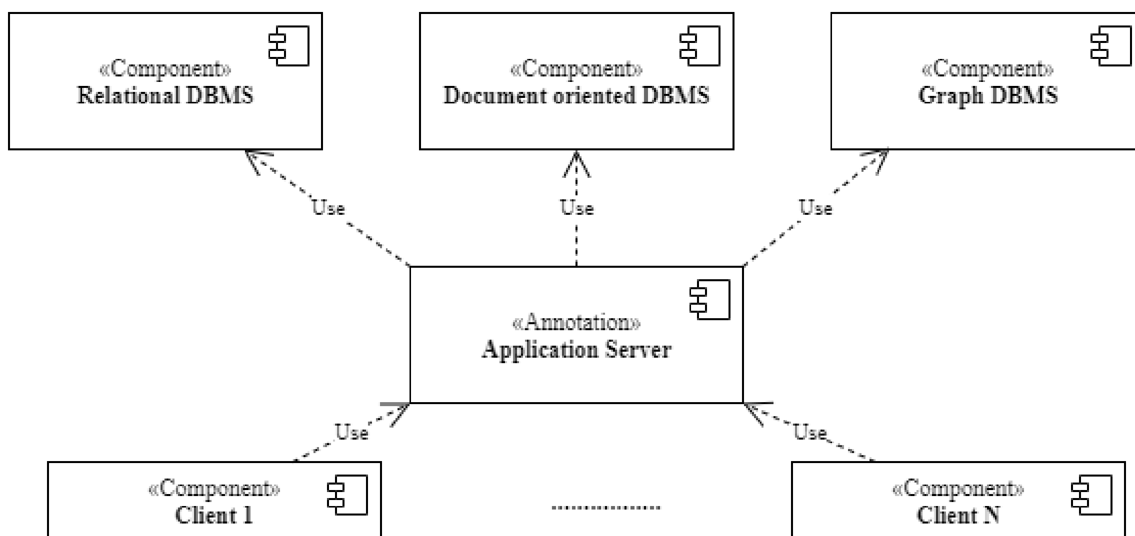

Fig. 1.3 – An application server aggregating data from DBMSs [16]

MMDBMS allows implementation of database-centric architecture (see Fig. 1.4) which could be both two-tier or three-tier. In this architecture a significant part of business logic is implemented in the database as programmable objects: stored procedures and functions. In the database-centric architecture CRUD operations could be accessed via an interface consisting of stored procedures. For corporate applications it could be especially important to control data access on the DBMS level: every connection is opened under a user's account, direct access to tables or collections is forbidden and the data is accessed via

stored procedures. Additionally, deployment and maintenance costs of single MMDBMS are less than for traditional architecture with multiple DBMSs.

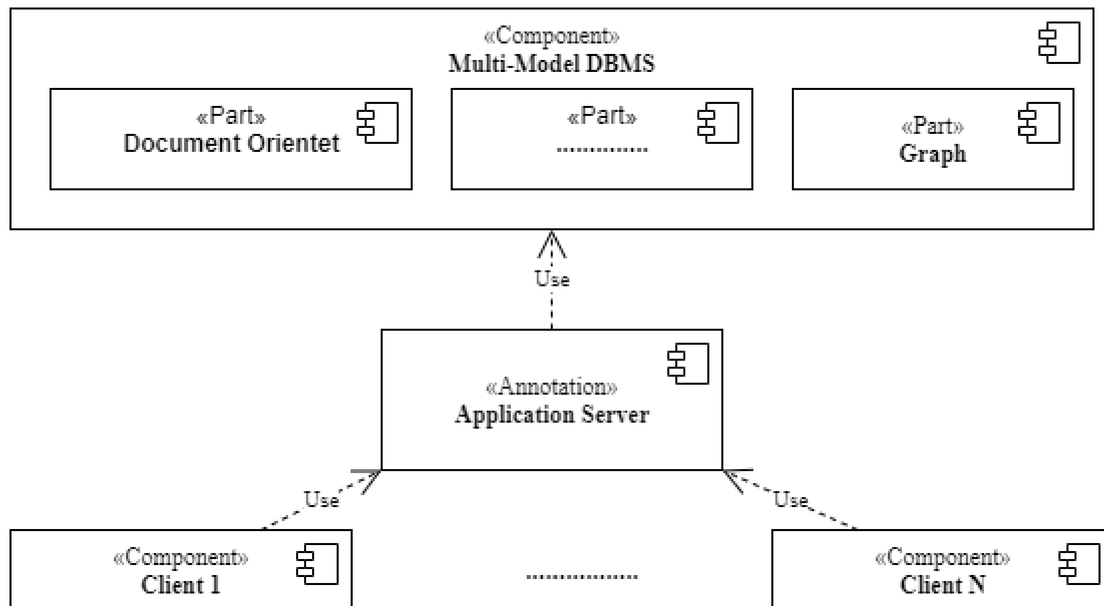

Fig. 1.4 – An application Server accessing data from a MMDMS [16]

1.3 Review and comparison of MMDBMSs

For comparison of MMDBMSs the following characteristics were chosen:

1. Supported data models. A MMDBMS with more models supported is a more general solution and has higher adaptability characteristics. However, more general MMDBMS may have worse performance due to its complexity and increased hardware and execution environment requirements.
2. Architecture of DBMS and implementation of data model. Depending on implementation, a DBMS could have the primary data models and extensions for support of remaining or a top level API on a set of data models.
3. Indexes. Depending on the types of supported indexes, a DBMS could be optimized for different workloads: with indexes queries may be executed without sequential scan.
4. Programmable objects like functions, stored procedures, views, packages. More types of programmable objects and several scripting languages could help implementation of database-centric architecture.

5. Supported APIs. An SQL API could help developers to carry out data migration from a relational DBMS.

Multi-model and document oriented DBMSs became available to developers in 2010-es. Open source DBMSs were chosen for comparison (see Table 1.1).

Table 1.1. Multi-model database management systems.

Name, Release year	**Programming language**	**document model**	**graph model**	**relational model**	**Query language**
ArangoDB, 2011	C++	X	X		AQL
OrientDB, 2010,	Java	X	X	X	Gremlin, SQL with extensions for graphs
RavenDB, 2010	C#	X	X (experimental feature)		RQL (Raven Query Language) for data manipulation, data definition operations only via a driver API
CrateDB, 2016	Java	X		X	SQL
MongoDB, 2009	C++/C/JavaScript	X	X (only limited support for graph queries)		JavaScript/JSON, GraphQL
Couchbase, 2010,	C++				N1QL
Neo4j, 2010	Java	X (limited support)	X		Cypher

For further analysis, DBMSs with support of graph and document models are selected.

OrientDB is a multi-platform MMDBMS that supports graph and document data models over an hierarchical API: document API and graph API on the top of it (see Fig. 1.5).

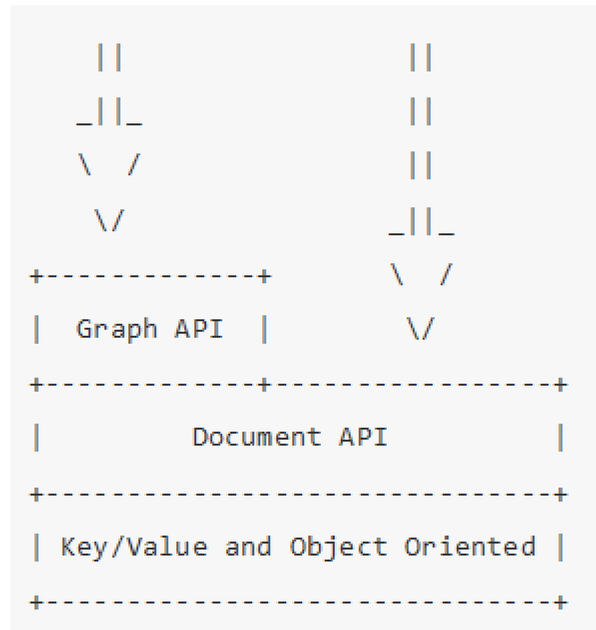

Fig. 1.5 – OrientDB API [17]

In OrientDB documents always belong to classes – special containers which partially play the same role as tables in relational databases and collections in document oriented databases like MongoDB. Unlike tables and collections, OrientDB classes support inheritance and have one of three types: schema-less, schema-full or schema-mixed. Objects of a schema-full class have predefined properties with constraints and optional indexes. Schema-less objects can have no constraints on type and number of attributes. Schema-mixed objects must follow a predefined schema and may contain additional attributes. A document belonging to the class inherits all its properties. Classes are grouped into clusters – units of physical data storage [18].

OrientDB supports four index algorithms [19]: SBTree, Hash, HashV2, Lucene. Additionally, custom indices could be written in Java. Indexes could be used explicitly in queries (“**SELECT FROM INDEX:<index-name> WHERE key = <key>**”).

On the Document API layer documents could be referenced to each other by the LINK mechanism. Links are set between classes and play the same role as foreign keys in relational databases. For unique relations between vertices edges are created [*20*]. Vertices are documents of the base class V. New vertices could belong to the class V or any derived class. Edges are documents of the base class E. As any document, an edge can contain content [*21*].

Unfortunately, OrientDB developers do not maintain the C# data driver [22]. As C# popularity is increasing and Java is stepping down [23], this fact could be considered as a serious disadvantage of OrientDB.

ArangoDB is positioned as a native multi-model database: the same query language and database engine are used for all supported models [24]. The document, graph and key-value models are implemented based on the document data model [24]. In an ArangoDB database documents are grouped in collections. Collections could be one of two types: vertex and edge. A collection could have a schema object bound to it. Schemas have the control level attribute. Depending on the control level, all documents or only newly inserted are validated against the schema [25].

RavenDB is primarily document oriented, but it supports graph querying as an experimental feature [26]. The graph support is implemented as an extension to the RQL query language. RavenDB does not store edges as separate documents. Graphs are built and analyzed based on existing documents (vertices) and references to other documents inside them. Edge name is set by the attribute containing the reference. RavenDB offers two types of indexes:auto and static. Auto indexes are created automatically by the server Query Optimizer component. In the documentation it is stated that RavenDB avoids full scan operations during query execution by use of auto indexes [27]. Static indexes could be Map or Map-Reduce.Map indexes specify by what fields documents could be searched. Map-reduce indexes used for aggregation queries: selection values from fields and with use of a reduce function and calculation of the aggregation result [28].

MongoDB is another example of a document oriented DBMS. It allows storing documents with and without schema. Documents could be referenced by a special document ID – objectID data type. In [29] it is stated that MongoDB could be used for relational, object and graph operations. MongoDB has limited support for graph queries. In official MongoDB documentation [29] it is stated that a dedicated graph DBMS may be necessary for frequent use of graph capabilities. That statement makes us assume that the speed of graph processing in MongoDB may not be not fast enough. For graph processing referenced documents are used. Graph edges could be expressed only as references between documents.

Neo4j is a graph DBMS that is often used with MongoDB. Neo4j does not have full support of the document data model, but allows to store key-value properties which are assigned to nodes and edges. Neo4j may be used with a document oriented DBMS if full support of the document model is required. For example, a background software application

for data synchronization between Neo4j and MongoDB is available as an open source project on Github [30].

To sum up the review of the MMDBMSs, ArangoDB, OrientDB and RavenDB (with insignificant limitations) are the DBMSs that have full support for document and graph storage and processing. Additionally, OrientDB implements the concepts of class and inheritance instead of collections. This allows development of rich domain models. Another major difference of OrientDB is that it utilizes class and inheritance for storage of graph edges. MongoDB could be considered as a single model DBMS with extensions. Use of a system of MongoDB and Neo4J does not meet the purpose of improving software maintainability as it increases complexity of a software system.

1.4 Detailed review of ArangoDB, RavenDB and MongoDB

ArangoDB, RavenDB and MongoDB could be good candidates for further analysis of performance as their query languages share common features (see Table 1.2) . For all of them it is possible to implement a benchmarking application using C# programming language.

Table 1.2 Comparison of query languages

Feature	**ArangoDB (AQL)**	**RavenDB (RQL)**	**MongoDB (JavaScript)**
Functions	+	JavaScript functions	JavaScript functions
Views	+	implemented as JavaScript indexes	SELECT queries with limited number of operators
Stored procedures	-	-	-
Graph edges	Edge collection	Document attributes	
Multi-model query	AQL statements	Index+graph extensions	View + graph extensions or Aggregation pipeline
Shortest path search	+	+	-
JOIN	+	+ partial support with the include operator	+ $lookup operator

For detailed analysis of the DBMSs and their query languages, the demo dataset was used (see Appendix A).

1.4.1 ArangoDB

ArangoDB query language AQL supports document and graph queries. The structure of a simple AQL document query is the following:

```
FOR element IN collection
FILTER element.attribute1 == "value"
SORT element.attribute2
RETURN element
```

An example of document query which selects documents containing more than 10 tags:

```
FOR doc IN people
FILTER count(doc.tags) > 10
SORT doc.name
return doc
```

Graph query structure resembles document query:

```
WITH people, relations
FOR v
IN min_path_length..max_path_length
OUTBOUND vertex_collection edge_collection
```

An example of graph query calculates shortest path between two people:

```
WITH people, relations
FOR v
IN 1..2
OUTBOUND people people_relations
```

An example of multi-model (graph and document) query firstly filters projects and people (document part) and then calculates shortests paths (graph part) between them:

```
WITH people, projects,  relations
FOR p IN people
FILTER count(p.tags) > 10
FOR pr IN projects
FILTER pr._id == projects/25000'
FOR v, e IN OUTBOUND p TO pr people_project_relations
RETURN [v._key, e._key]
```

ArangoDB queries are processed by the query optimizer which builds query plans. The query optimizer could build a query plan automatically or use the index which is

explicitly set by a user. In vertex collections hash indexes for *_from* and *_to* are automatically created.

An example of ArangoDB query plan:

```
FOR doc IN people
FILTER count(doc.tags) > 10 SORT doc.name return doc
Execution plan:
Id   NodeType         Est.   Comment
1    SingletonNode       1    * ROOT
8    IndexNode       16255      - FOR doc IN people   /* skiplist index
scan */    FILTER (COUNT(doc.`tags`) > 10)   /* early pruning */
7    ReturnNode     16255       - RETURN doc
Indexes used:
By   Name                      Type       Collection   Unique   Sparse
Selectivity   Fields       Ranges

8    idx_1718512802621030400   skiplist   people        false     false
99.98 %   [ `name` ]    *
```

1.4.2 RavenDB

RavenDB query language RQL supports document queries and has an extension for graph querying. Besides RQL, RavenDB has native support for LINQ – a query language which is a part of Microsoft .NET.

Every document query begins from the *from* keyword. Then a collection or an index is specified:

```
from Collection_or_index_name as item
where search(attribute, "value1")
select item.attribute1, item.attribute2
```

The significant limitation of standard collection query is that besides built-in functions and comparison operators no operations and arithmetic expressions could be used. As an example, the following query violates the standard RQL syntax:

```
from People as p
group by p.Department
where p.BaseSalary*1.3>1000
order by count() as long desc
select count(), p.Department.
```

For this purpose, a JavaScript or C# index could be created. In RavenDB user-defined indices are elements that not only used for faster scanning of data elements, but partially implement functionality of SQL views and functions. RavenDB offers two types of user defined indexes: map and map-reduce. Map index specifies which fields from document should be indexed and selected [28]:

```
map('People', function (employee){
      return {
            FirstName : employee.FirstName,
            LastName : employee.LastName,
            Salary:CalculateSalary(employee.Year,
employee.BaseSalary),
            Manager: employee.ReportsTo};
})
```

Map-reduce index allows us to aggregate data. A map-reduce index contains two parts: map and reduce. The map part selects data from a collection (projection operation). Then the reduction is applied to the map results [28]. The example below calculates the average salary per department:

```
Maps = new HashSet<string>()
{
     map('People', function (p){
      return {
            FirstName : p.FirstName,
            LastName : p.LastName,
            Salary: CalculateSalary(p.Year, p.BaseSalary),
            Manager: p.ReportsTo,
            Department: p.Department,
            Count:1
            };
      });
};

Reduce = @"groupBy(x => x.Department)
 .aggregate(g => {
      var salary_sum = g.values.reduce((sum,x) => x.Salary + sum,0);
      var number = g.values.reduce((sum,x) => x.Count + sum,0);
      return {
```

```
                    Department: g.key,
                    Avg: salary_sum / number
            }
})";
```

A graph query contains three parts: **(initial nodes)-[edges]->(ending nodes)**. Graph query starts from the *match* keyword, after which initial nodes are defined in brackets. After hyphen an edge document attribute is set. Finally, an ending node collection is set in brackets. A simple query selects friends list using the match operator:

```
match
    (People as person1) -
    [FriendOf as FriendOf]->
    (People as person2)
select
   id(person1) as id1,
   id(person2) as id2,
   person1.name as name1,
   person2.name as name2
```

For graph traversals recursive queries are used. In the example below, paths from movies with the tag to people are searched recursively. The min and max params of the path are set to 2 and 3 respectively:

```
match
    (movies as mv where tags in ("1995 films "))-
    recursive as chainOfCommand(2, 3, shortest)
    {
         [refs as rfs]->
         (People as participants)
    }
select
{
    title : mv.title,
    role : chainOfCommand.map(x => x.participants.role).join(' >> ')
}
```

With the recursive part it is possible to set minimum and maximum numbers of elements in paths.

Multi-model queries can be implemented by combination of indexes and graph queries. The query below selects employees with salary greater than $3000 and their manager:

```
match
    (index 'map_index_1' as worker where Salary>3000)-
    [Manager as mgr]->
    (index 'map_index_1'  as manager)
```

1.4.3 MongoDB

MongoDB utilizes a JavaScript/JSON based query language. Typical READ query uses the following pattern:

```
db.<collection>.find($query);
while (myCursor.hasNext())
{
      print(tojson(myCursor.next()));
}
```

For more complex queries, an aggregation pipeline is used. An aggregation pipeline is a container for sequential data processing stages. It may include filtering (the $match stage), graph processing (the $graphLookup stage) and grouping (the $group stage) [31].

For graph queries the $graphLookup operator is used. It recursively searches a collection(s) with restriction on depth and optional filters [32]:

```
{
$graphLookup:
{
 from: <collection>,
 startWith: <expression>,
 connectFromField: <string>,
 connectToField: <string>, as: <string>,
 maxDepth: <number>,
 depthField: <string>,
 restrictSearchWithMatch: <document>
}
}
```

The example below selects friends of friends with max depth 5:

```
db.people.aggregate(
```

```
        [
                { $graphLookup:
                        { from: "people",
                        startWith: "$FriendOf",
                        connectFromField: "FriendOf",
                        maxDepth: 5,
                        connectToField: "_id",
                        as: "FriendsLine" }
                }
        ]
)
```

A multi-model query with mongoDB could be written as a combination of a view and a graph traversal query or via an aggregation pipeline. After the view is created, it is possible to use it in a graph query. An example of a view returns employees with salary greater $3000:

```
db.createView( "HighIncomeEmployees", "Employees", [ { $gt: { Salary: 3000 } } ] )
```

Results of Chapter 1

Data models and MMDBMSs were studied and compared. The main criteria for choice of DBMSs for benchmarking were support of the data models and availability of C# data access driver. Because the thesis is devoted to MMDBMSs, chosen DBMSs must support graph and document data. Regarding the second criteria, chosen DBMSs must support C# which will allow us to create modules and perform benchmarking in the same conditions. Additionally, benchmarks written in C# could represent interest for an increasing number of Microsoft .Net developers. **ArangoDB**, **RavenDB** and **MongoDB** were chosen for comparison. It would be worth it to find out if use of RavenDB auto indexes ensures a good level of performance. Additionally, it is necessary to find out if MongoDB graph extensions have acceptable performance.

2. Design and implementation of the benchmarking application

Development of the benchmarking application is a software development process which splits into the following stages: requirements analysis and identification, design and implementation. For requirements analysis and identification the benchmarking process is studied and benchmarking systems are compared. Based on the comparison, a list of requirements is built. Then a software application architecture is designed and documented with use of UML diagrams. Finally, the architecture is implemented in the code.

2.1. Introduction to benchmarking

Benchmarking is a process of assessing performance of computer hardware or programs. The results of benchmarking are quantitative characteristics of performance. Benchmarking is carried out by use of special software – a benchmarking application. A benchmarking application takes sample workload, reads user preferences and starts execution of tests. After execution of tests quantitative results are stored in a local file system or shown to the user. Workload is a set of operations with the object of testing that imitate some real daily activity: file copying, video and audio conversion for hardware benchmarking; browsing of web-site pages, creating reports in a Customer Relationship Management (CRM) system or chatting and voice calling in a messenger application for software benchmarking. Examples of popular hardware benchmarks are CPU-Z [33], MemTest86 [34] and Geekbench [35].

Servers are tested for performance to estimate their response time, memory consumption and other params. A typical performance benchmarking tool like Microsoft Wcat may contain the following components: agents and a controller [36]. Agents are installed on separate machines. A controller is an application or script that sends a test scenario and a start command to clients. Then clients execute workload and send reports to the controller. Finally, the controller saves or displays the results. Multi-client deployment of benchmarking applications may be necessary if network and hardware capacity (ability to generate a certain number of queries) of a client does not fit the processing capacity of the server. Multi-client deployment is especially important for stress testing. Unlike web-servers, DBMSs do not send messages like the HTTP 503 messages. For an isolated DBMS (without software systems like web-sites or CRM) it is wise to estimate response time under a given

workload. This will allow us to evaluate algorithms implemented in the DBMS, but not the whole hardware and software system configuration.

For DBMS benchmarking (see Fig 2.1), a benchmark application should contain two main elements as input data: a database schema and a workload model. A workload is represented by a sequence of database read and write queries [37]. The benchmarking process starts from a data load. After data is loaded, a series of multithreaded tests are executed and performance metrics are calculated. Response time, number of operations, usage of memory and disk are used as measured performance characteristics. [38] states that latency is one of core performance characteristics that influence user experience. Latency is directly connected with throughput: if the hardware configuration is constant, latency increases with increase of throughput. Another performance characteristic is scaling which measures performance change with increasing number of nodes in a database cluster. Availability is studied in different conditions of hardware failure: whole node failure, network failure, disk failure. When an element of the database cluster is turned off, benchmarking is carried out.

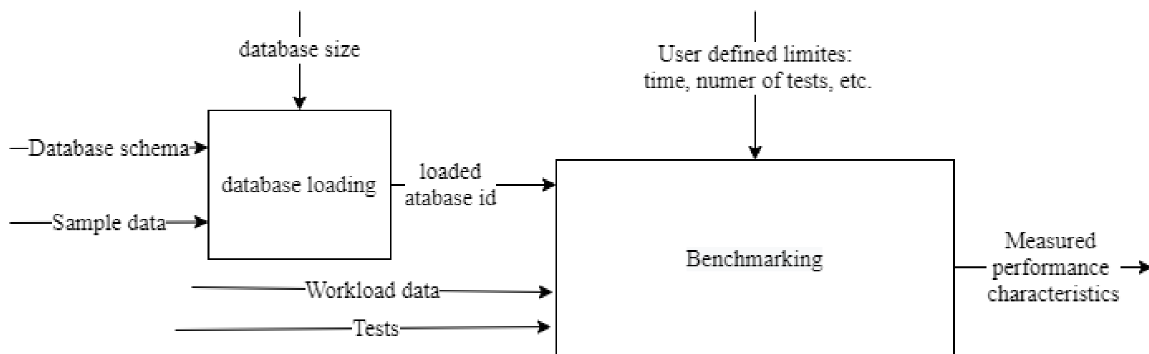

Fig. 2.1 – DBMS benchmarking process

Four MMDBS benchmarks were found and chosen for further analysis (see Table 2.1).

Table 2.1. Benchmark results

Benchmark	**Programming language**	**DBMSs**	**Workload**	**Winner**
YCSB [39]	Java, Python	ArangoDB, Elastic Search, OrientDB	social network data	MongoDB
Arango tests [40]	JavaScript (NodeJS), shell scripts	Neo4j, MongoDB, PostgreSQL, OrientDB, ArangoDB	pokec dataset (social network data) by Stanford University	ArangoDB
Unibench [41]	Java, NodeJS	ArangoDB, OrientDB	E-commerce	ArangoDB in majority of tests
Benchmarking orientdb with neo4j and mongodb [42]	Shell scripts	OrientDB, Neo4j, MongoDB	social network data	MongoDB for document queries, OrientDB for majority of graph queries

[41] has a detailed description of E-commerce multi-model queries, but for the benchmark execution the authors used different DBMS drivers (NodeJS for ArangoDB and Java for OrientDB) which could distort the results of the experiment. [43] has a simple, but not well-designed implementation: workloads are hard-coded as class methods, each workload class in UniBench has a fixed number of workload methods. Besides architecture drawbacks, in [43] there is no implementation of multi-thread executor of database workload. Current code in [43] does not fully relate with the paper [41]: it does not contain any NodeJS code mentioned in the article.

In [39] it was said that not only workload, but database size and deployment configuration influence performance results. In [39] the Yahoo! Cloud Serving Benchmark (YCSB) testing tool was used for performance comparison of MongoDB, OrientDB, ArangoDB and ElasticSearch. The YCSB tool is implemented as a set of Java modules accessing a database cluster (see Fig. 2.2). The YCSB tool is run via controlling sh (Linux) or bat (Windows) scripts.

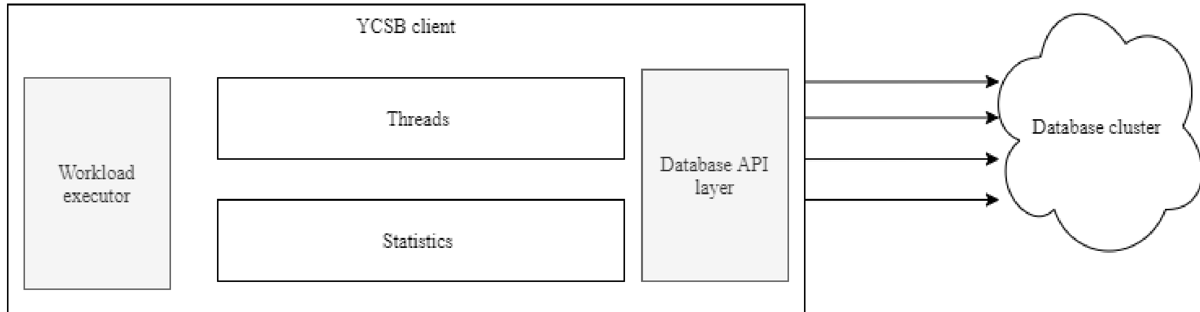

Fig. 2.2 – YCSB performance benchmarking tool architecture [39]

The testing was performed on a single machine. The authors did not clarify if the benchmarking application was run on a separate machine. Based on the benchmark results, MongoDB had significantly better performance [39].

In [40] workload was splitted into the following parts: single document (user profile) read, single document (user profile) write, aggregation (age distribution for all profiles), neighbors search (profiles of direct neighbors and neighbors of neighbors), shortest path (path lengths between nodes in social graph). Additionally, memory consumption was measured. [40] denotes that the MongoDB extensions for graph queries (particularly the graphlookup operator) had an unacceptable level of performance. It is worth mentioning that in [40] it is said about a warm-up phase – a special procedure that executes simple queries that do data scanning. This procedure allows a DBMS to cache a certain number of data entries in RAM depending on a DBMS internal architecture. Though the [40] benchmark has strong points such as good planning, standardized hardware (Amazon Cloud) and ability to run in a local environment and repeat the benchmark (the scripts are open source and are available on Github). The main disadvantage is that it was carried out by not a fully independent organization.

All the above mentioned benchmarking applications ([40], [41] and [39]) have only a command line user interface. Additionally, YCSB requires both Java and Python to be installed. Another drawback of these benchmarks is that they don’t provide a standard template of a DBMS workload executor (module or plugin) and documentation (guide) for adding workload for new DBMS.

2.2 Requirements of the benchmarking application

After analysis of the benchmarking applications a list of functional and nonfunctional requirements was created. From a user perspective (see Fig. 2.3), the application will provide a graphical user interface for starting the benchmark procedure, modifying the settings and views statistics of several benchmarks.

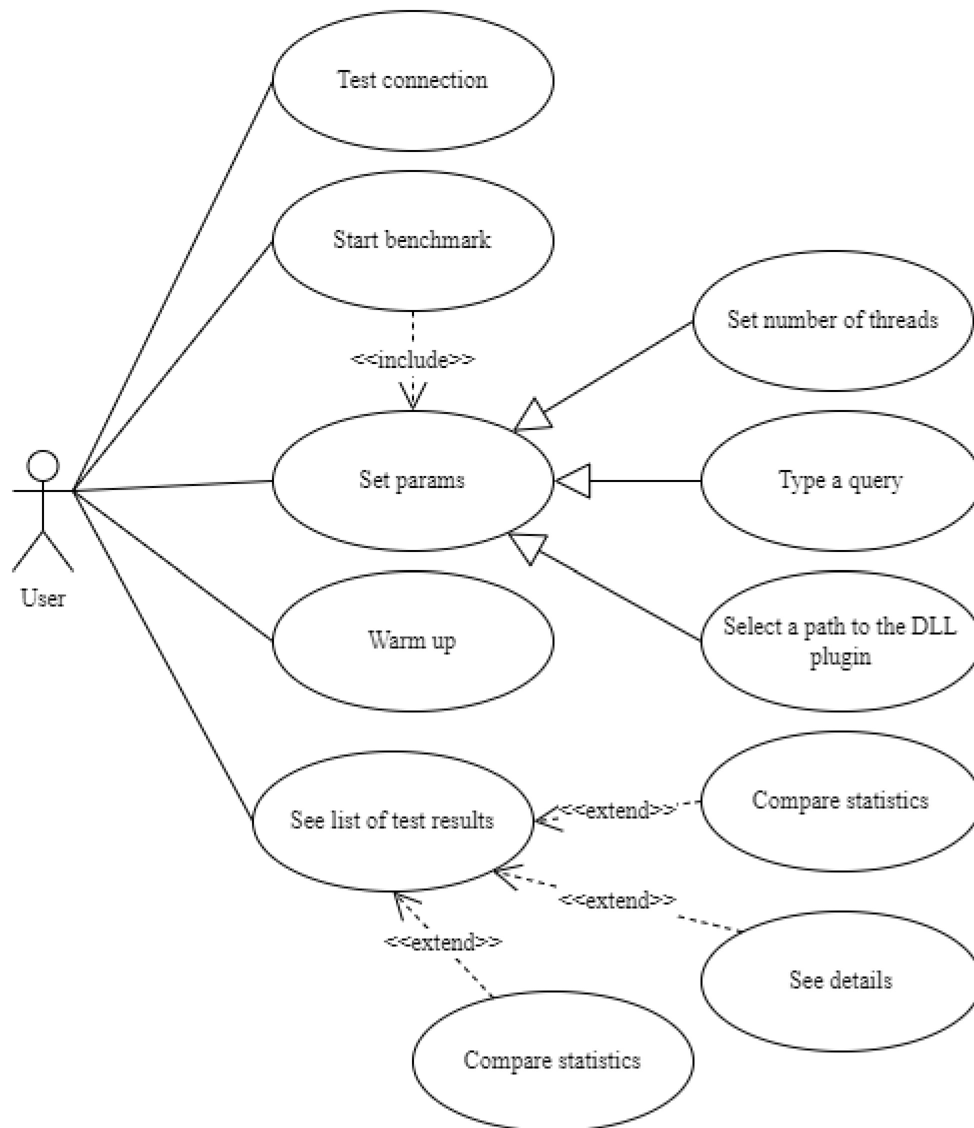

Fig. 2.3 – Functional requirements of the benchmarking application

Workload requirements:

1. The workload should contain indexes.
2. The schema must be multi-model and contain elements of graph and document data models.

3. Data queries must include shortest path calculation.

Requirements to the benchmarking procedure:

1. Warm-up procedure. All tests must be run after the warm-up procedure. The warm-up procedure must execute read queries. The warm-up procedure must load every document one time only.
2. Benchmarking application deployment. The benchmark application and the DBMS server must be deployed on separate machines to minimize influence on the benchmarking results.
3. The benchmarking application must execute tests in multithread mode.
4. The benchmarking application must provide a graphical user interface.
5. Comparison of benchmarking statistics must be displayed.
6. DBMS support. The benchmarking must contain ready-to-use modules for RavenDB, MongoDB and ArangoDB.
7. Measured performance characteristics: latency (maximum and average query execution times).

Non-functional requirements to benchmarking:

1. Extendability. DBMS modules must be created as pluggable components.
2. Maintainability: the Model-View-ViewModel pattern must be used for the UI implementation, the application code must be divided into projects.
3. Portability. It must be possible to run the application on Windows and Linux.

2.3 Benchmarking process

The main use scenario of the benchmarking application (see Fig. 2.4) comprises three steps: setting of the benchmarking process preferences, executing the benchmarking process and display of benchmark results. A user can start benchmarking after choosing a path to a DBMS module. When the benchmarking is finished, its results are saved in a subdirectory inside the application directory. Afterwards it could be used to compare multiple runs of the benchmarking procedure.

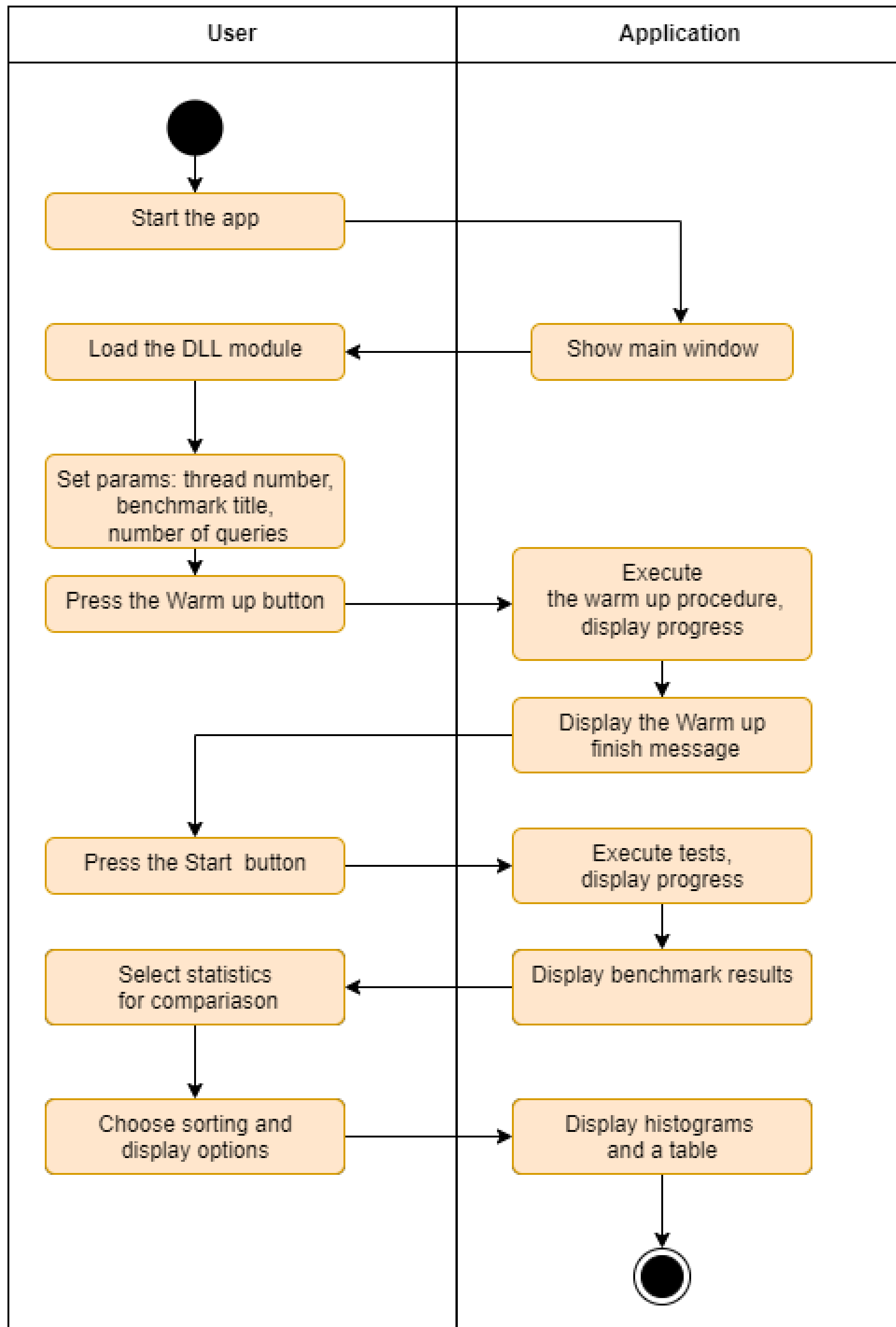

Fig. 2.4 – General workflow

2.4 Application architecture

The benchmarking application consists of three main components: UI, Runner and Abstract (see Fig. 2.5). The UI component provides a form based user interface and interacts with an instance of the Runner component. Runner is responsible for data load, tests

execution and statistics calculation. The UI component displays the state of the benchmarking process, buttons for state changing and viewing statistics in tabular form and histograms. The Abstract component contains data types which are used by the benchmarking application and pluggable DBMS modules.

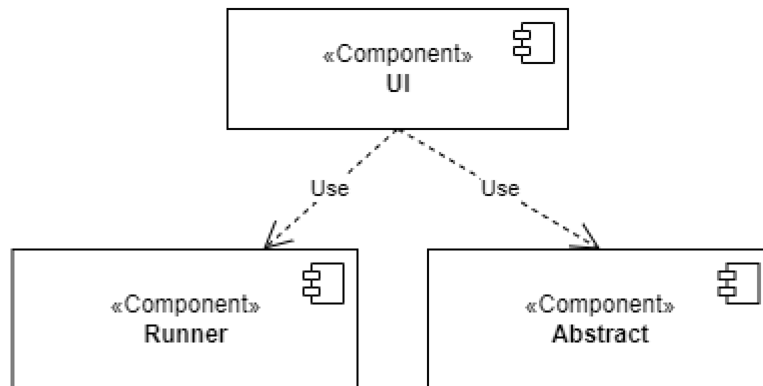

Fig. 2.5 – Component architecture

The application is implemented with the use of Microsoft .NET Standard 5.0 execution environment and SDK. The use of it allows us to run the application on Windows, Linux and MacOS.

2.4.1 Abstract component

The Abstract component contains the WorkloadCreator interface, inside which the following operations are declared:

1. TestConnection – performs a connection test to the DBMS server with a given connection string.
2. WarmUp – performs a warm up procedure.
3. ExecuteQuery – executes a query from the workload.
4. GetTitle – returns the default title for the benchmark.
5. Query – property for setting and getting a query text.
6. Stop – property for stopping warming up or workload execution.

The component is referenced by the benchmarking application and pluggable modules and is compiled into a DLL file.

2.4.2 DBMS plugin

DBMS plugin is a replaceable unit of the benchmarking application (see Fig. 2.6). It is compiled into a DLL-library and contains C# code, which implements connection and queries to particular DBMS and reads settings. Every DBMS module must contain a class implementing the IWorkloadCreator interface, which is defined in the Abstract component. The class must have only a default constructor. Additionally, a module must contain all necessary params of DBMS connection in the Schema.json file. A compiled plugin is placed in a separate folder and comprises the following files:

1. module_name.dll (compiled module).
2. Abstract.dll – a module with the IWorkloadCreator interface. The same version of Abstract.dll must be used in the benchmarking application.
3. Schema.json – a json file with connection string and other params.
4. A set of dll-files of which module_name.dll depends on. They could be database access drivers, libraries for data conversion, etc.

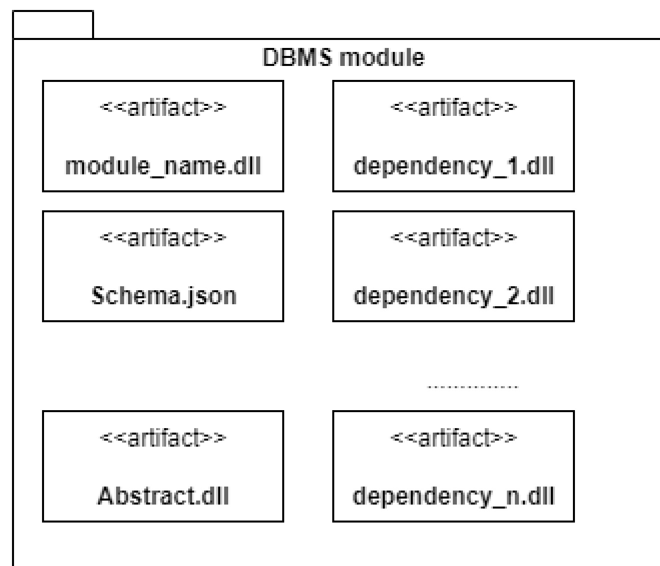

Fig. 2.6 – The internal structure of DBMS Loader module

DBMS modules are the parts of the benchmarking application plugin architecture (see Fig. 2.7). The architecture consists of two key elements: a consumer (UI) and plugins (DBMS modules). The auxiliary element is the Abstract package which contains definitions of data types (the IWorkloadCreator interface) used by both key elements: the consumer (UI) and plugins (DBMS modules).

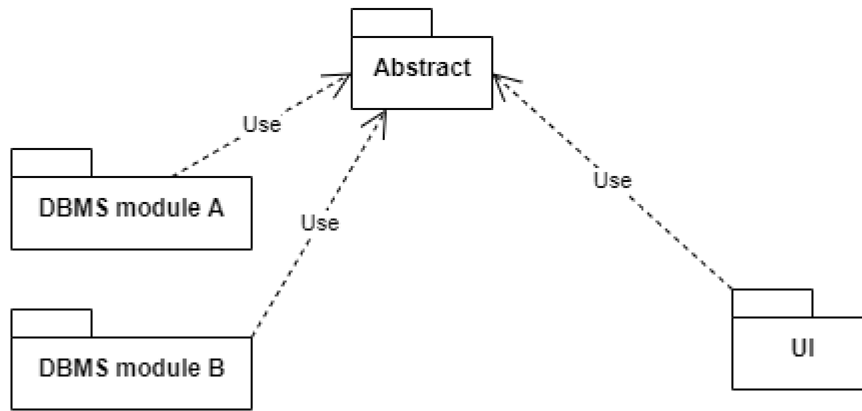

Fig. 2.7 – Plugin architecture

DBMS modules are loaded via the reflection mechanism which is a part of Microsoft .NET. A dll file is scanned for classes implementing the IWorkloadCreator interface. If a class is found, the module could be loaded and executed. The following code contains the implementation of the plugin loading:

```
var asm = Assembly.LoadFrom(modulePath);
      var type = asm.GetTypes().Where(x =>
x.IsAssignableTo(typeof(IWorkloadCreator))).FirstOrDefault();
var moduleClass = Activator.CreateInstance(type) as IWorkloadCreator;
if (moduleClass == null) throw new Exception("broke");
var task = Task.Run(async () => await moduleClass.TestConnection());
```

2.4.3 UI component

The UI (see Fig. 2.8 and Fig 2.9) is implemented as a C# form based application. The component is based on the Avalonia library and the Avalonia project template. The library allows the implementation of a cross platform form based user interface. The Avalonia library resembles the functionality of Microsoft WPF: it uses XAML for user interface markup and has an almost identical set of graphical user controls. The MVVM design pattern is used as a main architectural solution. The MVVM architecture separates application logic and user interface and comprises three main elements: model, viewmodel and view [44]. Model contains classes (BenchmarkResult), which represent data models. The user interface elements, their layout and styles (the View part) are declared in the XAML documents App.axaml and MainWindow.axaml. The application logic code is completely separated from the UI code: it contains no UI element IDs or properties. The viewmodel code (the MainWindowViewModel class) exposes data elements and data transformation methods

which are referenced in the UI code by the binding mechanism. The commands subpackage contains classes (the LambdaCommand and CommandBase classes), which are used for definition of methods that handle UI events. The implementation template for the Command package was taken from [45] and [46].

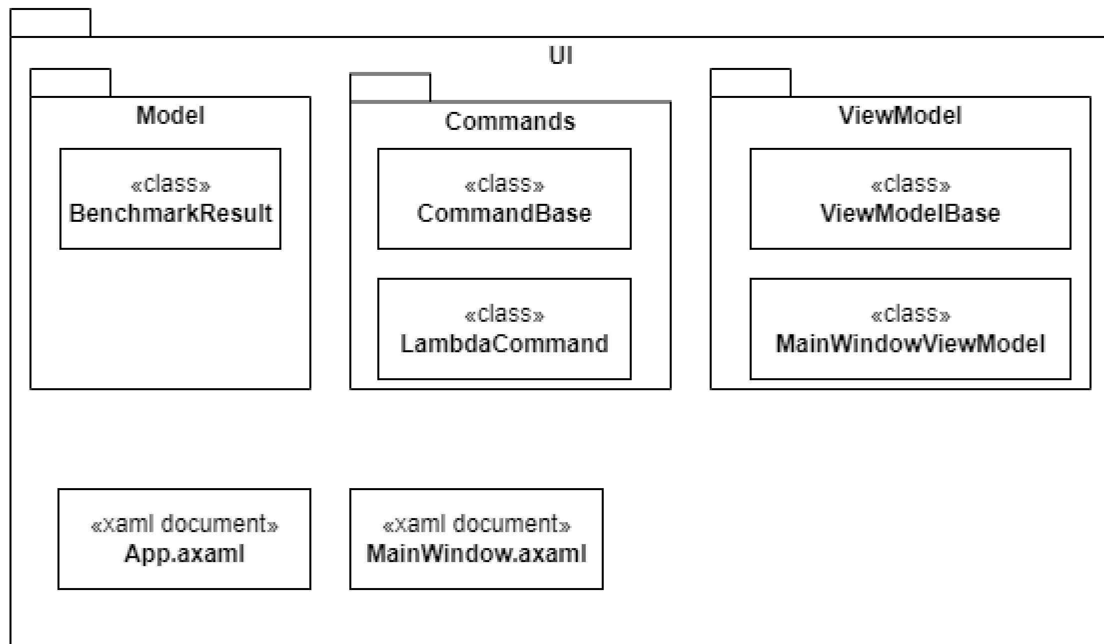

Fig. 2.8 – The structure of the UI component

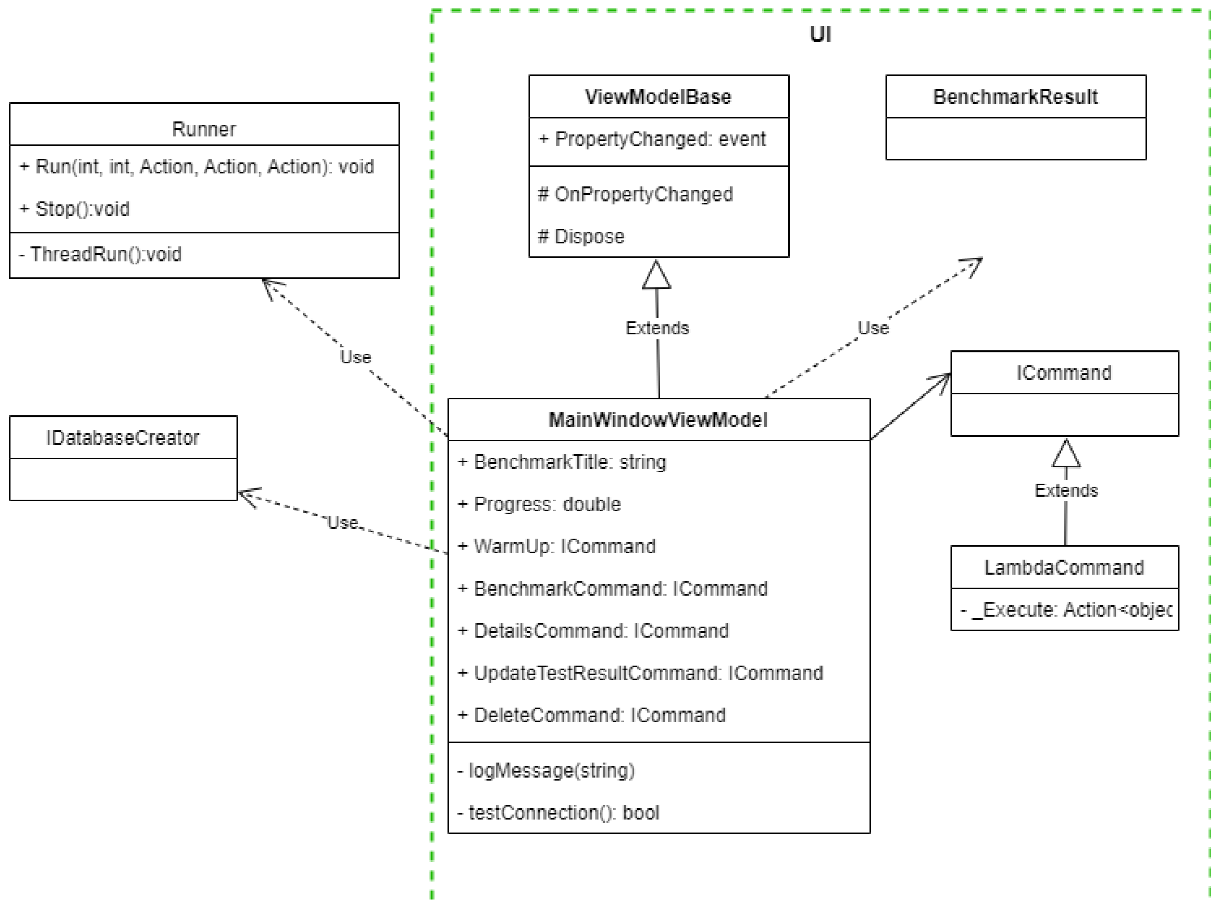

Fig. 2.9 – Class diagram

The UI component contains a form consisting of two tabs: Benchmark and Comparison (see Fig. 2.10). The first contains interface elements for control of the benchmarking process. The latter allows us to compare results of current and previous benchmarks.

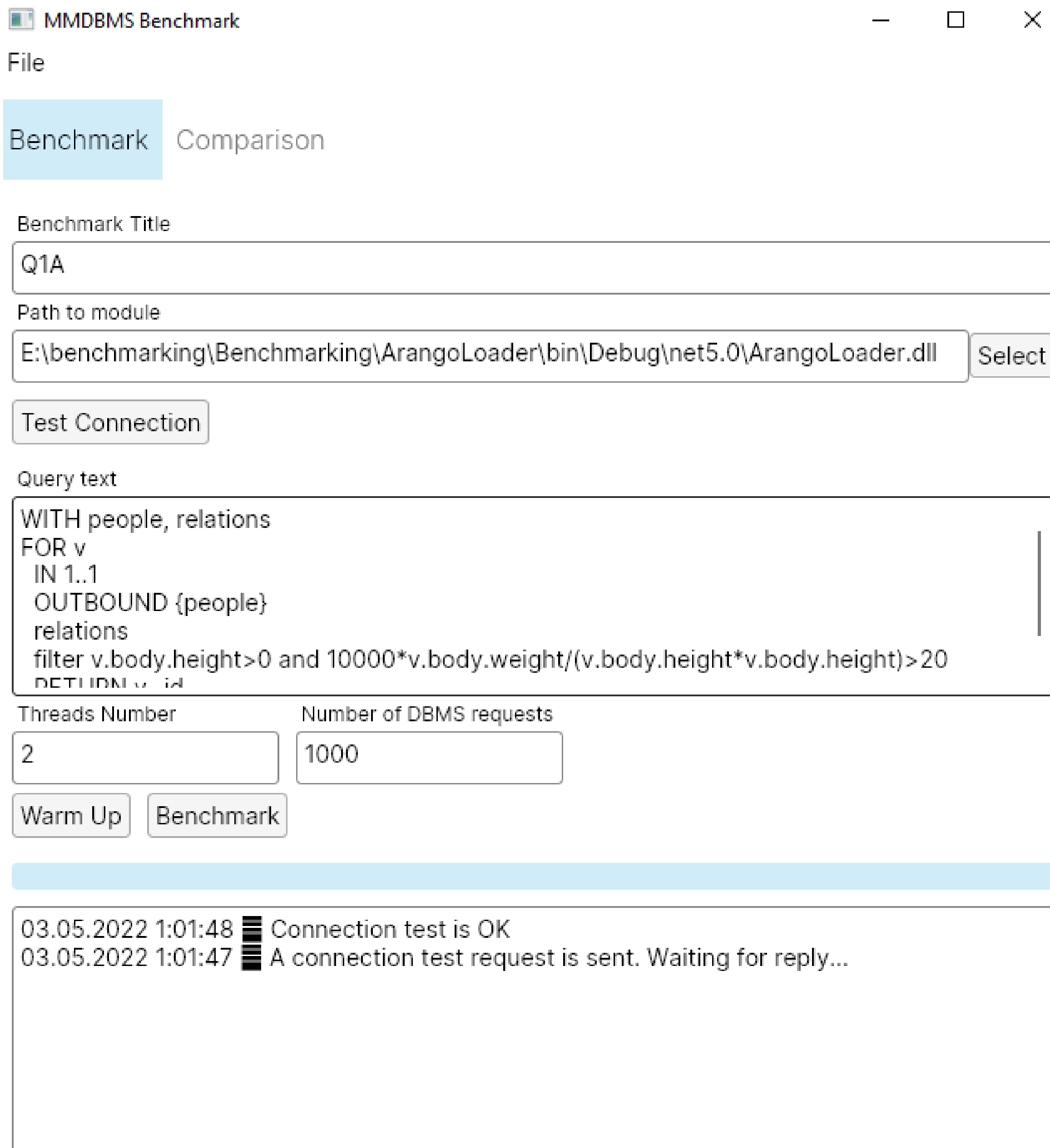

Fig 2.10 – Main window, the Benchmark tab

The UI application uses oxyPlot for displaying a bar chart of benchmarking results. Elements for comparison are chosen by a set of checkboxes. A user can see details of a benchmark in a separate window (for detailed description of the user interface see Chapter 3 Section 3.2).

2.4.5 Runner component

The Runner component contains an implementation of the multi-threaded benchmarking process. The component is implemented in the Runner class and exposes an

interface with two methods: Run and Stop. The Run method accepts the number of threads, the number of executed queries (workload elements) and references to four functions:

1. execWorkLoad is a function executing workload (usually the function calls the ExecuteQuery method of an IWorkloadCreator instance);
2. updateProgress is a function for reporting of progress;
3. logMessage is a function that reports state change of every thread (each thread is identified uniquely by an integer number);
4. reportFinish is a function for reporting benchmark finish.

Each thread executes the execWorkLoad function in a loop. After the function is executed, the number of executed workload elements is updated. The progress is calculated and checked for the finish condition. Depending on the calculation result, updateProgress or reportFinish is called.

2.5 Plugin development guide

To create a new DBMS plugin, the developer need to follow these steps:

1. Create a new .NET core class library project using Visual Studio or .NET command line.
2. Add reference to the Abstract project containing the IWorkloadCreator interface.
3. In the project setting set CopyLocalLockFileAssemblies True.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
  </PropertyGroup>
  ….
</Project>
```

4. Add a Schema.json file with a connection string and other params.
5. Add a class implementing the IWorkloadCreator interface. Implement loading of settings from the Schema.json file.
6. Compile the DLL-library. The library and its dependencies should be automatically copied to the output folder after the compilation finish.

Results of Chapter 2

The features of benchmarking systems were investigated. Differences of DBMS benchmarking from other benchmarking procedures were found. Requirements to the benchmarking procedure were identified and analyzed. The requirements analysis results took into account multi-model workload, previous results of DBMS benchmarking, architecture and functionality of existing benchmarking applications.

Based on the identified requirements, a benchmarking application was developed. The application has cross-platform graphical user interface and plugin architecture.

3. User documentation

3.1 Installation

The zip package (see Appendix C) should be unzipped. Before the start of the application in the app folder, it is necessary to install Microsoft .NET Standard 5.0. The application could be run on Windows, Linux and MacOS with Microsoft .NET Standard 5.0 installed. A DBMS plugin with its dependencies should be located in a separate folder.

3.2 Benchmark start

To start a benchmark, a user should do the following:

1. Prepare a connection to the DBMS by setting up an SSH tunnel or using a local instance of the DBMS. An SSH tunnel could be set in a bash console (Linux or MacOs) or in PowerShell (Windows).

Examples of bash command for establishing a tulles:

MongoDB version 5 Community: `ssh -L 27017:localhost:27017 guliyev@acheron.ms.mff.cuni.cz -p 42027`

ArangoDB communi ty v3.9: `ssh -L 8529:localhost:8529 guliyev@acheron.ms.mff.cuni.cz -p 42027`

RavenDB 5: `ssh -L 8080:localhost:8080 guliyev@acheron.ms.mff.cuni.cz -p 42027`

2. Start the application (the **UI.exe** file).
3. Select the DLL-module (**ArangoLoader.dll** for ArangoDB, **RavenLoader.dll** for RavenDB, **MongoLoader.dll** for MongoDB) for the DBMS (see Fig. 3.1).

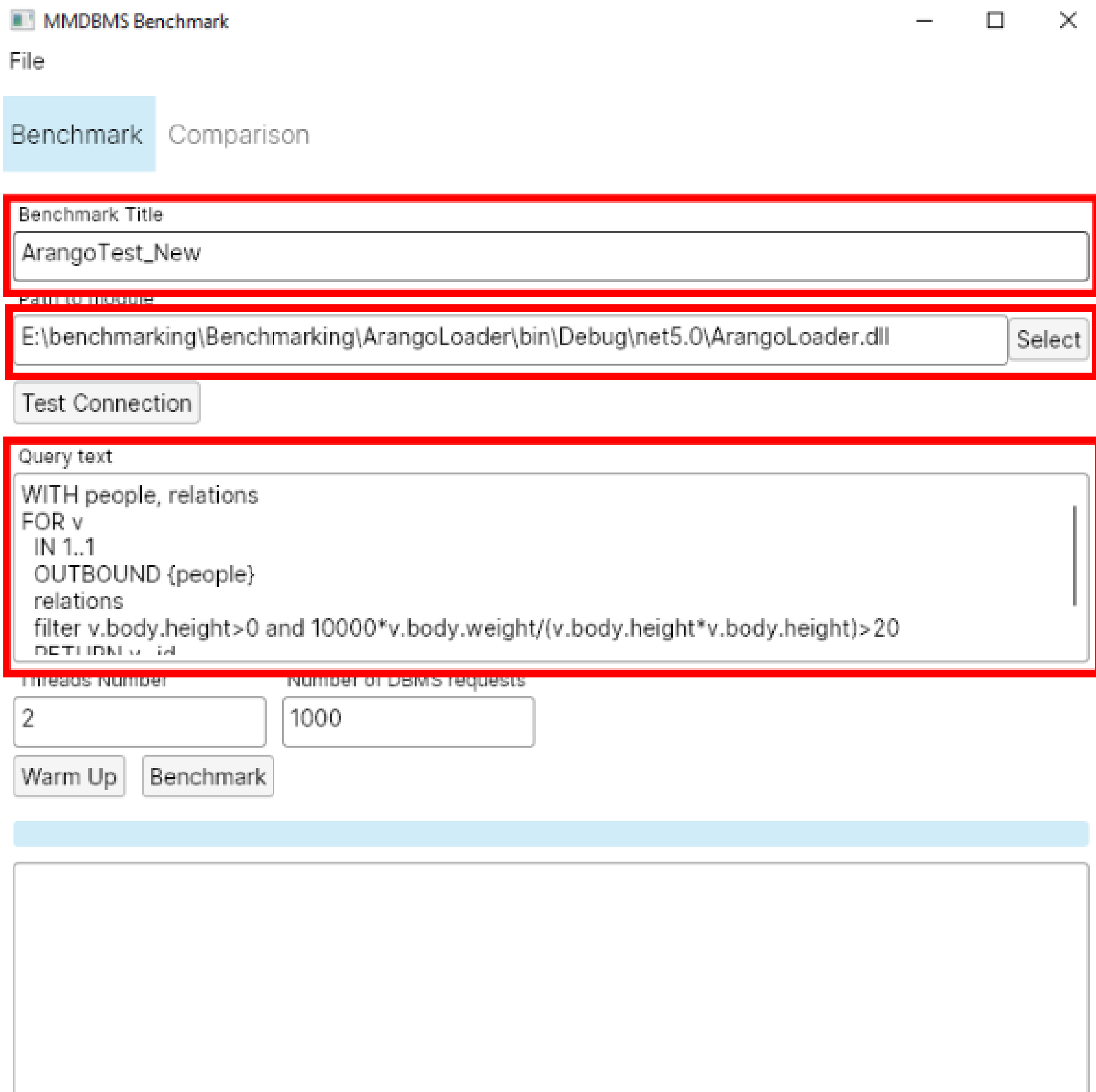

Fig. 3.1– The Benchmark Tab

4. In the query field enter a query replacing record IDs by collection name double curly brackets ({{collection_name}} instead of “collection_name/id”) for ArangoDB or a number (Q1, Q2, Q3) of a pre-built query for RavenDB or MongoDB (see Fig. 3.1).
5. Enter a benchmark title that will help you to identify the benchmark later (see Fig. 3.1).
6. Click the Warm Up button (optional).
7. Click the benchmark button.
8. Wait till the benchmark is finished.
9. Click the Comparison tab (see Fig. 3.3).
10. Click the Refresh button.

11. Select benchmark results to compare (see Fig. 3.3).
12. Click the Details button to see the benchmark details data (see Fig. 3.3 and Fig. 3.4).

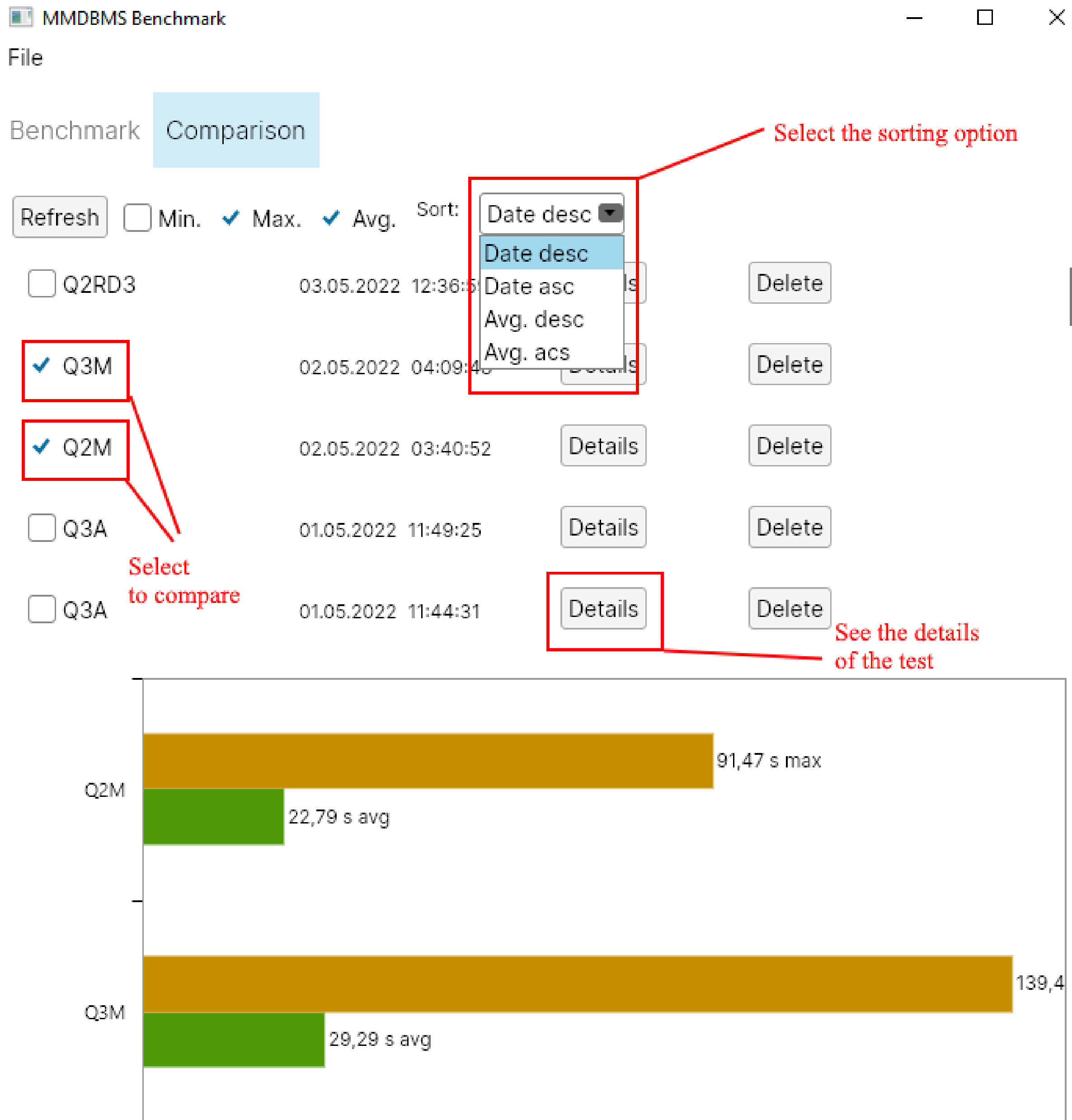

Fig. 3.3 – The Comparison tab

Fig. 3.4 – The benchmark details window

13. After the benchmark is executed, the results are saved in the tests folder. The benchmark results are presented by three files: timestamp.json (benchmark params and calculated values), timestamp.txt (query execution times) and timestamp.query (query text). The files could be processed in an external application if necessary.

4. Experiments

During the experiments stage the benchmarking application is used for evaluation of multi-model performance characteristics.

4.1 Dataset description

For experiments the Pokec dataset [47] was chosen. The Pokec dataset has been already used for the benchmark [40]. From the dataset 400,000 user profiles and 1,000,000 relations (social graph edges) were loaded. Descriptions of Slovakia geoplaces were loaded from [48]. The dataset contains social network user' profiles (the people entity) and friendship relations (the Friends attribute of the people entity) between them. The following user profile attributes were chosen: weight, height, gender, age, region. Additionally, a document collection for region descriptions (the geoplaces entity) was created (see Fig. 4.1). Entries of the people collection are connected to the geoplaces collection via the Geoname and Name attributes respectively.

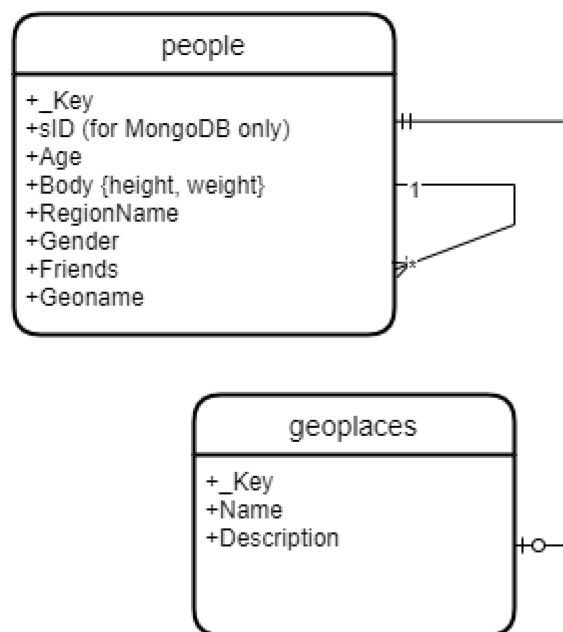

Fig. 4.1 – Dataset schema

In ArangoDB indexes for the attributes _id, _key, _from and _to of the vertex collection are created automatically. In RavenDB manual index and auto indexes are used.

In MongoDB the sId attribute was added to the people collection for unique identification of each record by a string value.

4.2 Execution environment

DBMS servers were deployed on Ubuntu Linux server 20.04 LTS with a 4 core processor and 4GB RAM.

According to the requirements, each query must access data of two models: document (aggregation, filter, join) and graph (shortest path, neighbors search). Queries were executed in four threads. Each workload type contained 1000 queries.

For data visualization Python, Jupyter, seaborn, numpy and matplotlib tools were used (see Appendix B). Current release of the benchmarking application does not display histograms due to limitations of the oxyPlot library and Avalonia. The implementation of a Matplotlib type histogram in Avalonia would require development of a class library performing operations with low level graphic primitives: it may be implemented in future versions of the application. Benchmark results were presented as histograms with query execution time on X-axis and quantity of queries corresponding to the particular execution time on Y-axis. The input data files for the benchmark result visualization was taken from the tests folder which was located in the directory of the benchmarking application executable file. An input data file for the Jupyter notebook is a multi line text file every line of which contains a query execution time as a float value.

4.3 Workload

Workload type 1. Find user's friends (graph part) with body mass index (BMI) above the normal (document part) .

For **RavenDB** an index was created for calculation of BMI:

```
index:
map("people", (person) => {
    if (person.body != null) {
        return {
                                                                    mass:
10000*person.body.weight/(person.body.height*person.body.height)
        };
    }
})
```

The index was the only possible solution of BMI calculation by means of RQL. The graph part does not contain recursive search and looks only for nearest friends:

```
match
    (index 'bmi' as person1 where id()='people/1')-
    [Friends[] as friend2]->
    (index 'bmi'  as endEdge where mass>22 )
```

For **ArangoDB**:

```
WITH people, relations
FOR v
  IN 1..1
  OUTBOUND {{people}}
  relations
    filter v.body.height>0 and 10000*v.body.weight/(v.body.height*v.body.height)>20
RETURN v._id
```

For **MongoDB:**

View (document query:, body mass index calculation and filtering):

```
db.createView(
      "people1",
      "people",
      [
      {
            $match: {
                        $expr: { $function: {
                        body: function(weight, height){ return height>0 &&weight*10000/(height*height)>20;},
                        args: [ "$body.weight", "$body.height" ],
                        lang: "js"
                        }
                  }
            }}
      ], {}
);
```

Graph Query:

```
        db.people_view.aggregate(
        [
                {$match: {"sId":"people/1"}},
                { $graphLookup:
                        {
                        from: "people",
                        startWith: "$Friends",
                        connectFromField: "Friends",
                        connectToField: "sId",
                        as: "Frd",
                        maxDepth: 1
                        }
                }
        ]
);
```

Histograms (see Fig. 4.2, Fig. 4.3 and Table 4.1) show that ArangoDB and RavenDB have comparably the same performance. An index (a view analog) for RavenDB was created before the benchmark execution. Indexes for height and weight do not affect performance because the graph part of the query is executed first. After the index on the sId attribute was added to MongoDB (see Fig. 4.4), it outperformed ArangoDB and RavenDB. The results prove that even in case of search of adjacent vertices the MongoDB has a modest performance level.

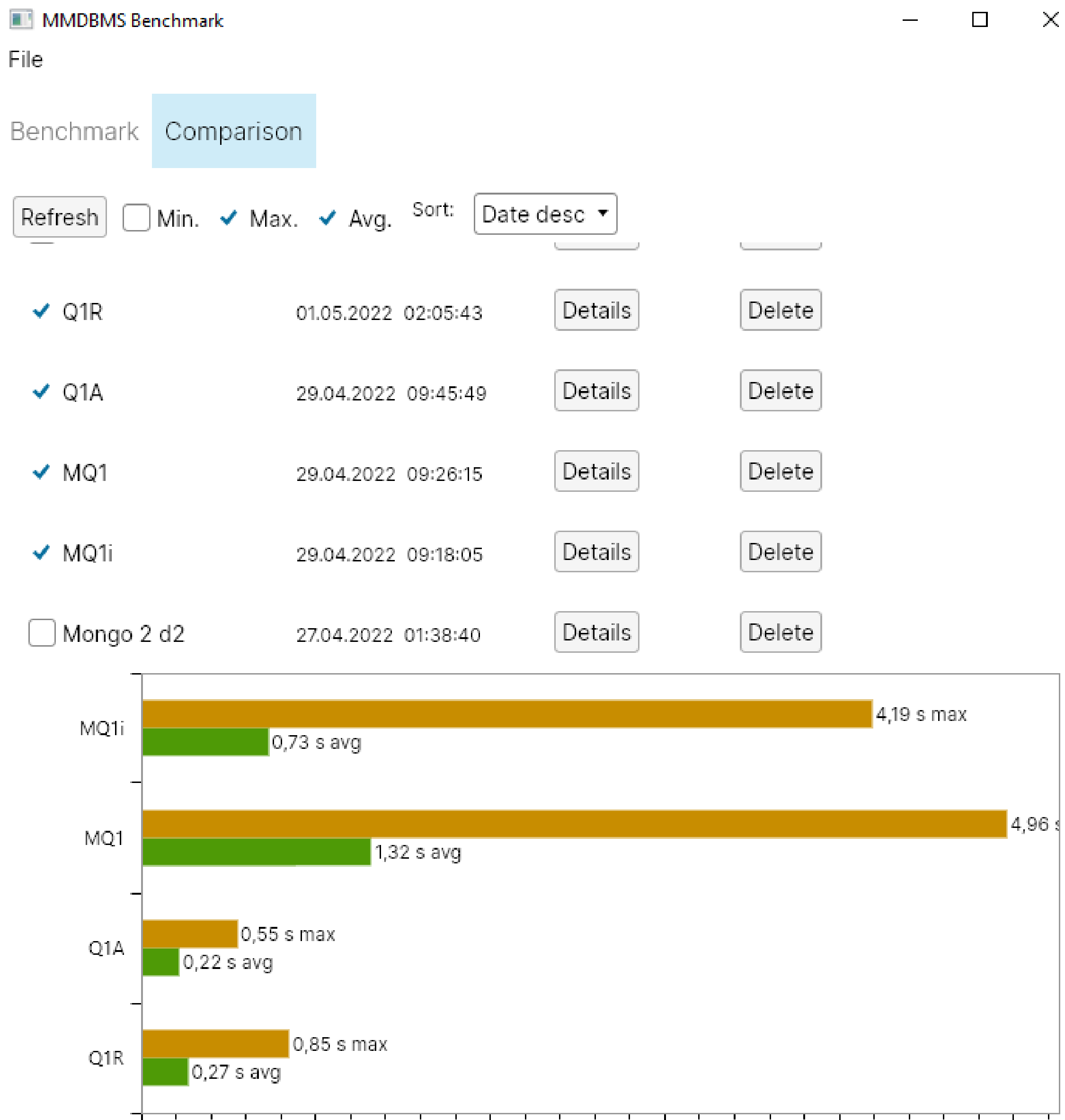

Fig. 4.2 – Type 1 workload results from the benchmarking application (MQ1 – MongoDB, MQ1i – MongoDB with the index, Q1R – RavenDB, M1A – ArangoDB)

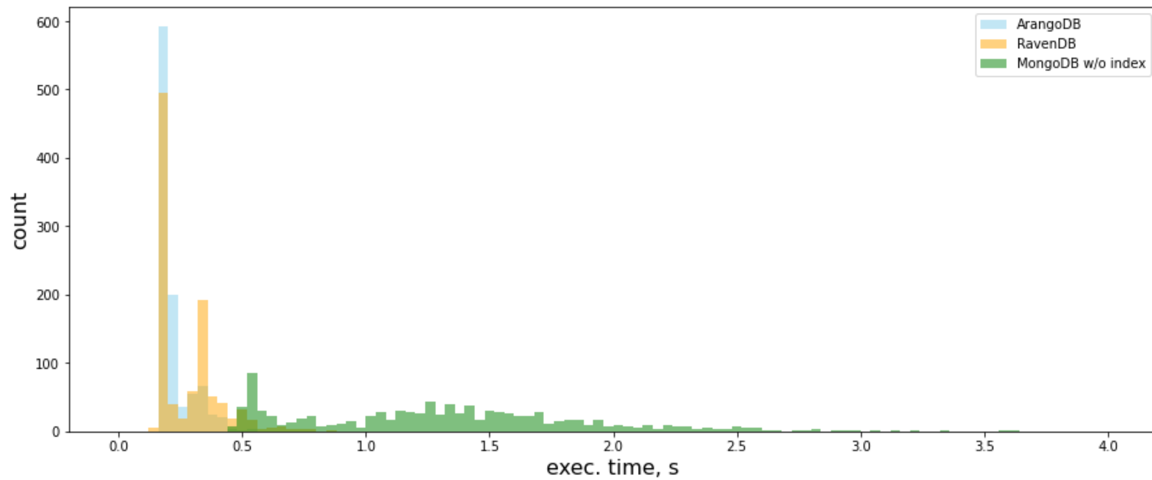

Fig. 4.3 – Distribution of the type 1 workload execution times

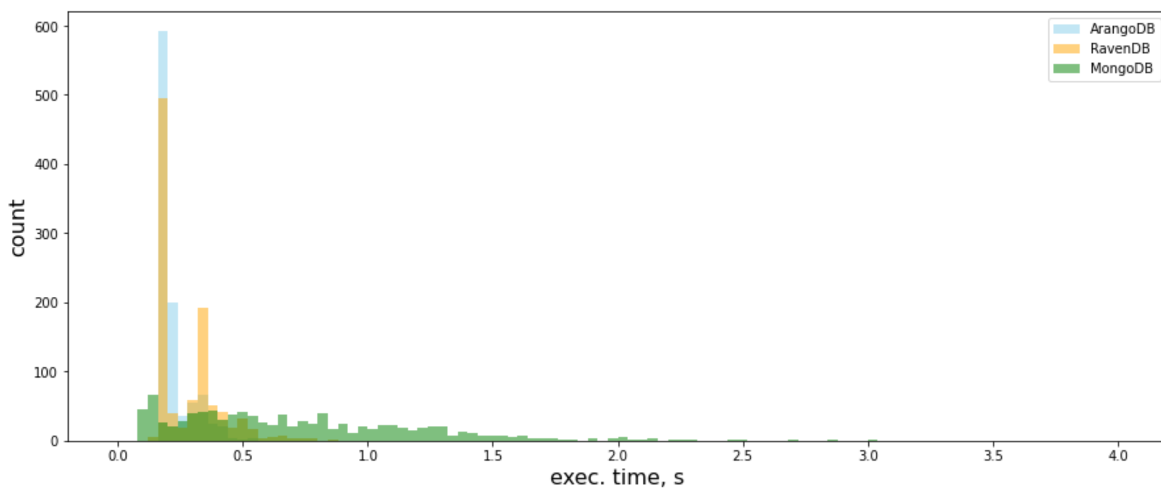

Fig. 4.4 – Distribution of the type 1 workload execution times with MongoDB attribute referenced.

Table 4.1 Workload type 1 benchmark results

DBMS	**95-percentile execution time, s**
ArangoDB	0.36
RavenDB	0.51
MongoDB	2.39 / 1.61 with index

Workload type 2. Find the shortest path between two users (graph part) and get a list of regions with descriptions (document part).

For **RavenDB** (due to the fact that RQL does not support joins, the workload consists of two queries):

Part 1:

```
match
     (people as person1 where id()="people/6126")-
      recursive as way(0, 2, shortest)
     {
     [Friends[] as friend]->
     (people as middle)
     }-
     [Friends[] as friend2]->
     (people as endpoint where id()="people/6127")
select way[].middle.Friends
```

Part 2:

```
from geoplaces as geo
where name in (....)
select geo.name, geo.descriptions
```

For **ArangoDB**:

```
WITH people, relations
FOR v
  IN OUTBOUND SHORTEST_PATH {{people}} to {{people}}
  relations
    LET geo = (FOR x IN geoplaces FILTER x.name==v.region return
x.description)
RETURN {id:v._id, g:geo.description}
```

For **MongoDB**:

For MongoDB the query searches for paths from one person with specified max depth due to the limitation of the graphLookup operator. Then a programmer must check if any path exists between two people and look for the shortest path among those found. The query was written in this way due to the fact that MongoDB does not have an operator for the shortest path search. It is possible to find the shortest path in a path array programmatically on the client side.

document part:

```
db.createView(
   "people2",
   "people",
   [
```

```
        {  $lookup:
          {
            from: "geoplaces",
            localField: "geo",
            foreignField: "name",
            as: "place"
          }
    }
        ], {}
);
```

graph part:

```
db.people.aggregate(
        [
                {$match: {"sId":"people/1"}},
                {$graphLookup:
                        {
                                from: "people2",
                                startWith: "$Friends",
                                connectFromField: "Friends",
                                connectToField: "sId",
                                as: "Frd",
                                maxDepht: 2
                        }
                }
        ]
  )
```

The workload 2 test results (see Fig. 4.5, Fig. 4.6 and Table 4.7) show that unlike MongoDB, both ArangoDB and RavenDB guarantee acceptable execution time for almost all queries.

MMDBMS Benchmark

File

Benchmark Comparison

Refresh Min. Max. Avg. Sort: Date desc

Q2RD3 03.05.2022 12:36:59 Details Delete

Q3M 02.05.2022 04:09:48 Details Delete

Q2M 02.05.2022 03:40:52 Details Delete

Q3A 01.05.2022 11:49:25 Details Delete

Q3A 01.05.2022 11:44:31 Details Delete

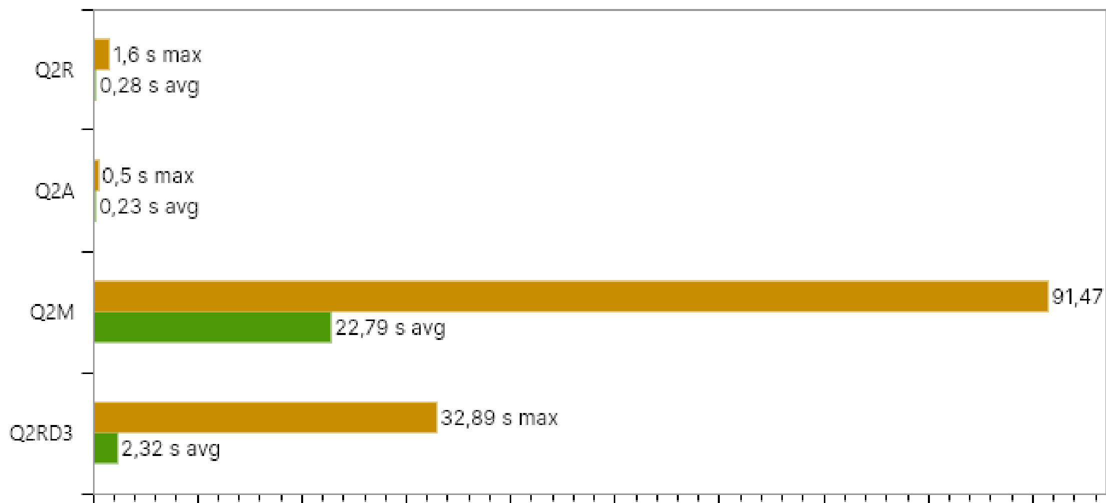

Fig. 4.5 – Type 2 workload results from the benchmarking application (Q2M – MongoDB, Q2R – RavenDB depth=2, Q2RD3 – RavenDB depth=3, Q2A – ArangoDB)

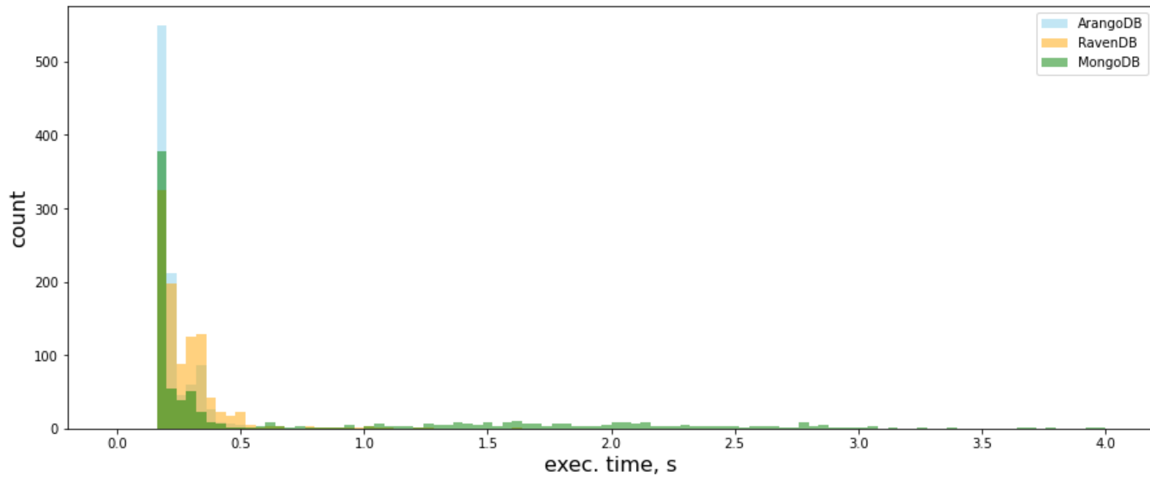

Fig. 4.6 – Distribution of the type 2 workload element execution times

Two variants (with max depth=2 and max depth=3) of RavenDB workload type 2 queries were compared. RavenDB performance decreases significantly with depth increase (see fig. 4.7).

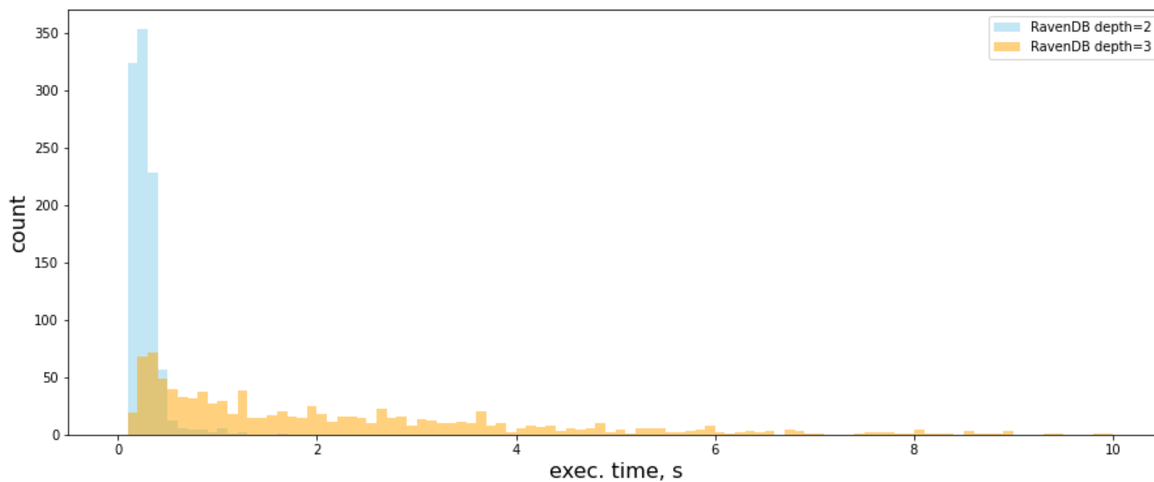

Fig. 4.7 – Distribution of the type 2 workload element execution times

Table 4.2 Workload 2 benchmark results

DBMS	**95-percentile execution time, s**
ArangoDB	0.36
RavenDB (depth=2)	0.49
RavenDB (depth=3)	6.78
MongoDB (maxDepth=2)	17.58

Workload type 3. Select friends and friends of friends (graph part, see Fig. 4.8) and count them by the region (document part – grouping).

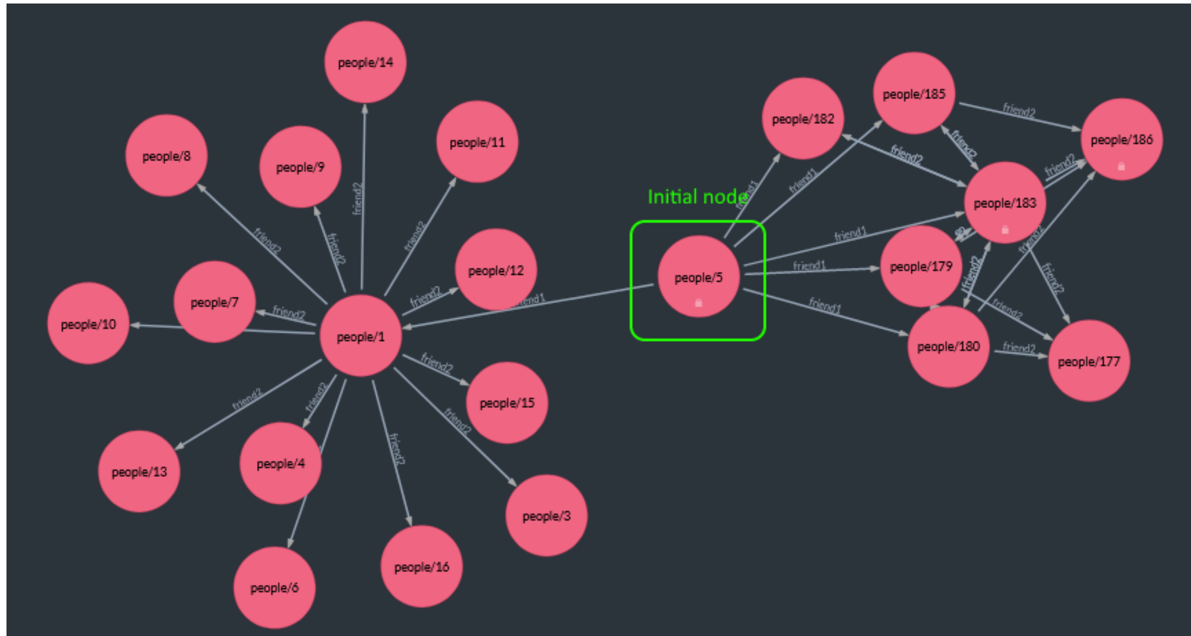

Fig. 4.8 – Graphical representation of the workload type 3 graph part (created in RavenDB Management Studio)

For **RavenDB** (due to the fact that RQL does not support joins, the workload consists of two queries):

Part 1:

```
match
    (people as person1 where id()="people/5")-
    [Friends[] as friend1]->
    (people as middlepoint where id()!="people/5")-
     [Friends[] as friend2]->
    (people as endpoint where id()!="people/5")
select endpoint
```

Part 2:

```
from people as p
where p.id() in ()
group by p.region
select p.region, count(p)
```

For **ArangoDB:**

```
WITH people, relations
```

```
FOR v
  IN 1..2
  OUTBOUND {{people}}
  relations
  RETURN
  MERGE(v, {
    region: (
        FOR t IN geoplaces
          FILTER t.name == v.region
          RETURN t.description
    )
  })
```

MongoDB:

graph part:

MongoDB Community 5 is not able to process views containing graph queries with depth 2 or more (see Fig. 4.9).

```
db.createView("people3", "people", [{$graphLookup: { from: "people", startWith: "$Friends", connectFromField: "Friends", connectToField: "sId", as: "Frd", maxDepth:2,
depthField: "depth" } } ], {})
```

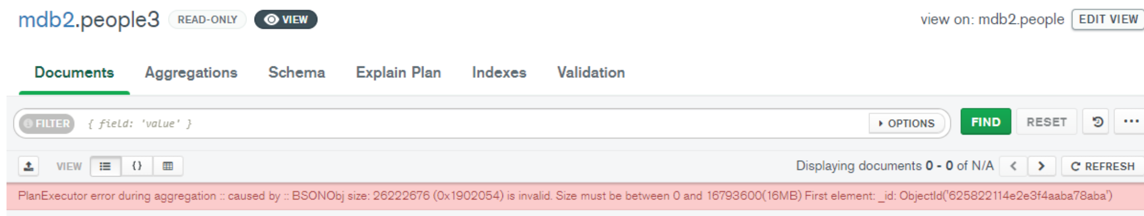

Fig. 4.9 – Result of a query to the view containing graph query

```
db.people.aggregate(
    [ {$match:{"sId":"people/1"}}, {$graphLookup: { from: "people2", startWith: "$Friends", connectFromField: "Friends", connectToField: "sId", as: "Frd", maxDepth:2,
depthField: "depth" } }, { $project: { "Frd.sId": 1} } ]
  )
```

document part:

```
db.people.aggregate(
```

```
[{$match : { "sId": [val1, val2, …, val_n]}},
 {$group : { "_id": "$geo", "count": {$sum: 1}}},
]
)
```

Histogram (see Fig. 4.10, Fig. 4.11 and Table 4.3) show that in complex queries ArangoDB outperforms RavenDB because for the latter it was necessary to decompose the workload element to a set of queries. MongoDB did not show any acceptable result.

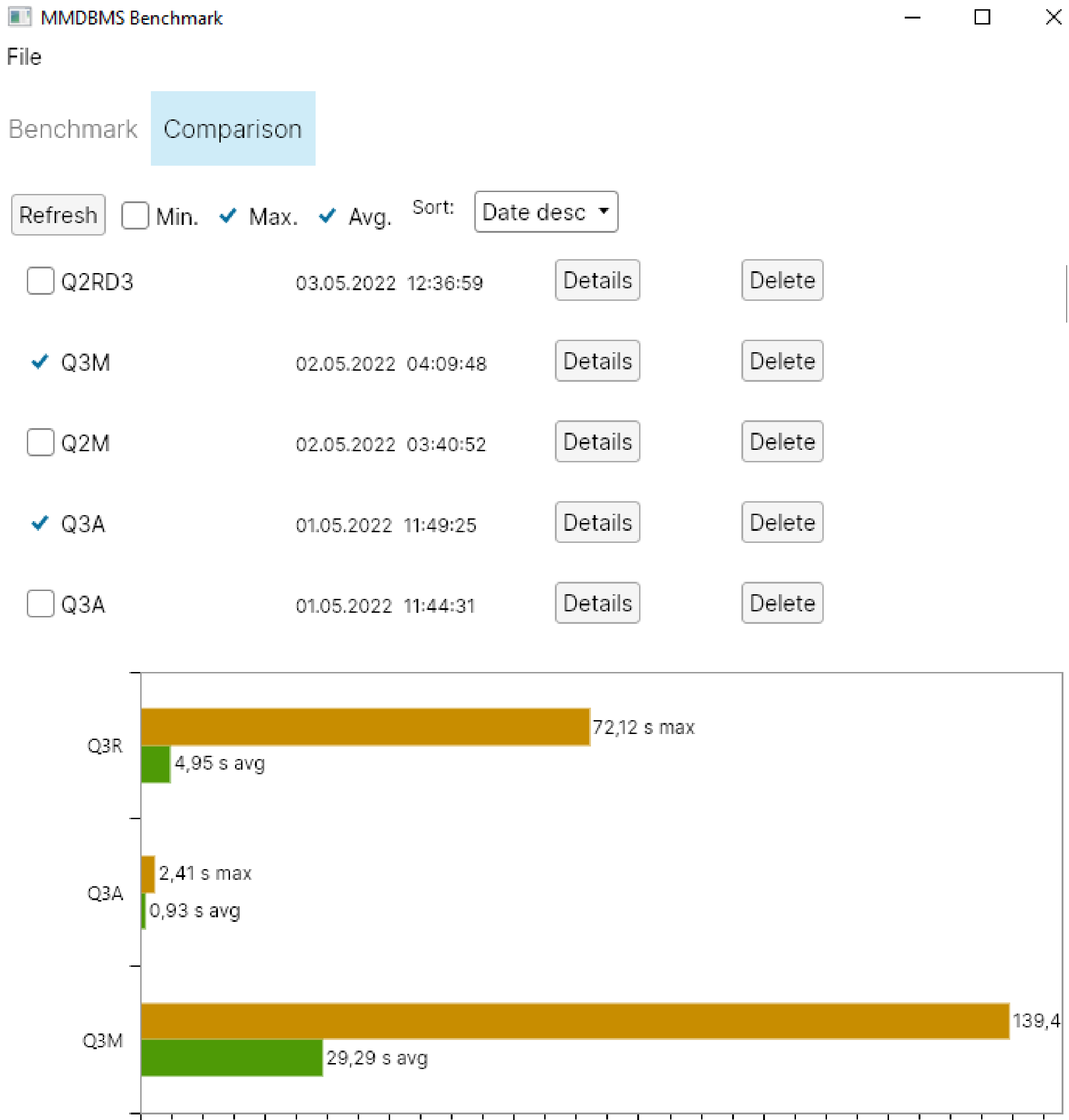

Fig. 4.10 – Type 3 workload results from the benchmarking application (Q3M – MongoDB, Q3R – RavenDB, Q3A – ArangoDB)

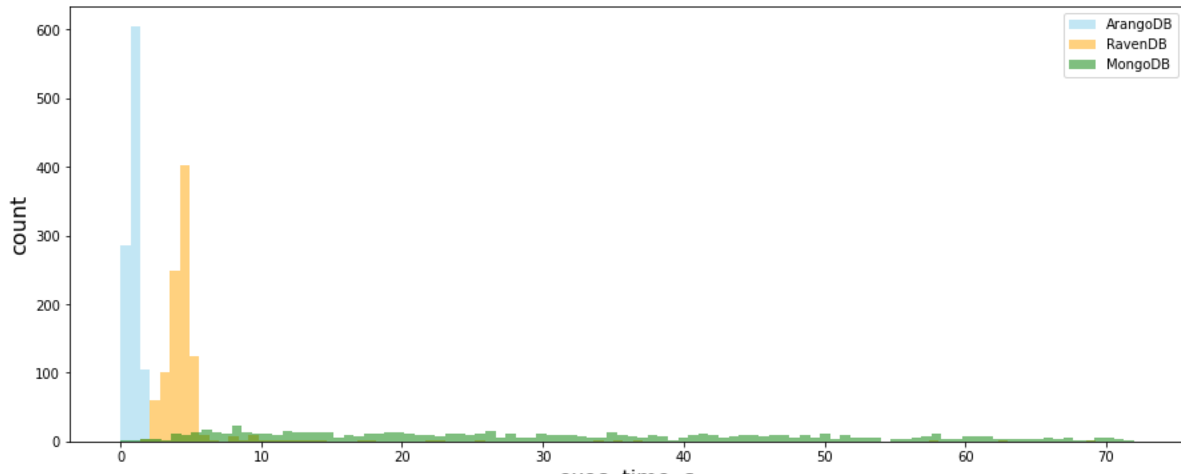

Fig. 4.11 – Distribution of the type 3 workload element execution times

Table 4.3. Workload 3 benchmark results

DBMS	**95-percentile execution time, s**
ArangoDB	1.59
RavenDB	6.00
MongoDB	71.87

4.3 Results discussion

The benchmarking results (see Table 4.4) show that ArangoDB has a significant advantage over RavenDB and MongoDB: it has predictable and high performance in all tests.

Table 4.4. The benchmarking results

Workload type	**DBMS**	**Avg. execution time, s**	**Max. execution time, s**	**Squared deviation, s**	**Total workload execution time**
1	ArangoDB	0.22	0,55	0.06	1 min. 27 s.
	RavenDB	0,27	0,85	0.12	1 min. 39 s.
	MongoDB (w/o index)	1.32	4.96	0,60	5 min. 59 s.
	MongoDB	0.73	4.19	0.51	3 min. 31 s.
2	ArangoDB	0.23	0.50	0.06	1 min. 28 s.
	RavenDB	0.28	1.60	0.14	1 min. 40 s.

	(depth=2)				
	RavenDB (depth=3)	2.32	32.89	2.69	10 min. 11 s.
	MongoDB	31.52	91.47	21.68	111 min. 25 s.
3	ArangoDB	0.93	2.41	0.37	4 min. 22 s.
	RavenDB	4.95	72.12	5.53	21 min. 7 s.
	MongoDB	33.83	139.43	22.72	130 min. 38 s.

ArangoDB seems more preferable choice for C# developers because of the following reasons:

1. Unlike RQL, AQL is a more mature query language and has support for joins, arithmetic operations, subqueries and other essential features. AQL graph query syntax is more concise and allows us to specify additional options like a graph traverse algorithm.
2. ArangoDB outperforms RavenDB in cases when queries for RavenDB have to be decomposed.
3. RavenDB is more optimized for usage with C# and LINQ, but graph queries must be written in RQL and separated from LINQ queries.
4. The MongoDB graph extension (represented by the graphLookup operator) gives a moderate and predictable level of performance in case of adjacent vertex search only. With max depth 2 or more performance decreases significantly.
5. The MongoDB graphLookup operator does not support search between two nodes and shortest path search.

Both ArangoDB and RavenDB could not be used for implementation of database centric architecture as they don’t support the full set of programmable objects available in other DBMSs PostgreSQL or Microsoft SQL Server.

Conclusion

Main results of the work are the cross-platform benchmarking application, detailed analysis of multi-document functionality and performance benchmarks for the ArangoDB, MongoDB and RavenDB DBMSs.

As result of the analysis, it was found that ArangoDB implementation of graph and document data models is more suited to the needs of .NET developers in terms of performance, portability and functionality. Regarding MongoDB and RavenDB, it is possible to conclude that their implementation of graph data models are not mature enough: graph and document queries are difficult to combine.

Unlike existing open-source solutions, the benchmarking application has a multi-platform graphical user interface and a plugin architecture with predefined data types. The extendable application architecture gives an opportunity for developers to create their own DLL-modules for new workloads and DBMSs. Developers' efforts are decreased because they do not need to study the application source code and have only to implement the plugin interface following the developer's guide. The application has good maintainability characteristics as it is structured into components and the UI part is implemented with the MVVM design pattern and asynchronous commands.

Future work may include development of data visualization capabilities, plugin manager and a DLL-plugin for PostgreSQL. Plugin manager will provide an interface for faster plugin installation from a remote repository. A PostgreSQL plugin may increase the popularity of the benchmarking application among a large group of PostgreSQL developers. Additionally, it may be worth it to create a multi-agent version of the benchmarking application. A multi-agent version will allow us to execute heavier workloads sent from a group of computers.

References

[1] Download mariadb server. *MariaDB.org* [online]. 29 September 2021. [Accessed 5 November 2021]. Available from: https://mariadb.org/download/

[2] Determine the version, edition, and update Level - SQL Server. *Determine the version, edition, and update level - SQL Server | Microsoft Docs* [online]. [Accessed 6 May 2022]. Available from: https://docs.microsoft.com/en-us/troubleshoot/sql/general/determine-version-edition-update-level

[3] TRUSKOWSKI, Wojciech, KLEWEK, Rafał and SKUBLEWSKA-PASZKOWSKA, Maria. Comparison of mysql, MSSQL, PostgreSQL, Oracle databases performance, including virtualization. *Journal of Computer Sciences Institute*. 2020. Vol. 16, p. 279–284. DOI 10.35784/jcsi.2026.

[4] WOJARNIK, G.. Selection of working database for the genetic algorithm processing data of exchange quotations. *Information Systems in Management*. 2016. Vol. 5.

[5] CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM*. 1970. Vol. 13, no. 6p. 377–387. DOI 10.1145/362384.362685.

[6] Popularity ranking of database management systems. *DB Engines* [online]. [Accessed 7 May 2022]. Available from: https://db-engines.com/en/ranking

[7] NoSQL databases: An overview. T*houghtworks* [online]. [Accessed 23 April 2022]. Available from: https://www.thoughtworks.com/insights/blog/nosql-databases-overview

[8] What is mongodb? *MongoDB Manual* [online]. [Accessed 23 April 2022]. Available from: https://docs.mongodb.com/manual/

[9] Design documents. *Apache CouchDB® 3.2 Documentation* [online]. [Accessed 23 April 2022]. Available from: https://docs.couchdb.org/en/stable/ddocs/index.html

[10] PORE, Akshay. NoSQL Data Architecture & Data Governance: Everything You Need To Know. *DATAVERSITY* [online]. 16 February 2018. [Accessed 7 May 2022]. Available from: http://www.dataversity.net/nosql-data-architecture-data-governance-everything-need-know/

[11] What is a wide-column database? definition & faqs. ScyllaDB. (n.d.). Retrieved November 11, 2021, from https://www.scylladb.com/glossary/wide-column-database/.

[12] Serialization - .NET. Serialization - .NET. *Microsoft Docs* [online]. [Accessed 24 April 2022]. Available from: https://docs.microsoft.com/en-us/dotnet/standard/serialization/

[13] DRAKE, Mark and OSTEZER. A comparison of nosql database management systems and Models. *DigitalOcean* [online]. 9 August 2019. [Accessed 24 April 2022]. Available from: https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models

[14] ANGLES, Renzo and GUTIERREZ, Claudio. Survey of graph database models. *ACM Computing Surveys*. 2008. Vol. 40, no. 1p. 1–39. DOI 10.1145/1322432.1322433.

[15] Neo4j Doc manager - developer guides. *Neo4j Graph Data Platform* [online]. [Accessed 24 April 2022]. Available from: https://neo4j.com/developer/neo4j-doc-manager/

[16] MICROSOFT PATTERNS & PRACTICES TEAM. *Microsoft® Application Architecture Guide*. 2. Microsoft Press, 2009.

[17] Graph or document API? *Graph or Document API? · OrientDB Manual* [online]. [Accessed 30 April 2022]. Available from: https://orientdb.com/docs/2.2.x/Choosing-between-Graph-or-Document-API.html

[18] Schema · *OrientDB Manual* [online]. [Accessed 6 April 2022]. Available from: https://orientdb.com/docs/last/general/Schema.html

[19] Indexing. *Indexing · OrientDB Manual* [online]. [Accessed 26 April 2022]. Available from: https://orientdb.com/docs/2.2.x/Indexes.html

[20] Basic concepts. *Basic Concepts · OrientDB Manual* [online]. [Accessed 30 April 2022]. Available from: http://orientdb.com/docs/3.0.x/datamodeling/Concepts.html

[21] Classes, schema and constraints. *OrientDB Manual* [online]. [Accessed 30 April 2022]. Available from: https://orientdb.com/docs/last/gettingstarted/Tutorial-Classes.html

[22] ORIENTECHNOLOGIES. Net support broken · issue #9681 · orientechnologies/orientdb. *GitHub* [online]. [Accessed 26 April 2022]. Available from: https://github.com/orientechnologies/orientdb/issues/9681

[23] TIOBE Index for December 2021. *TIOBE* [online]. [Accessed 8 December 2021]. Available from: https://www.tiobe.com/tiobe-index/

[24] Advantages of Native Multi-Model. *ArangoDB* [online]. 9 December 2020. [Accessed 22 November 2021]. Available from: https://www.arangodb.com/community-server/multi-model/

[25] ArangoDB Data Model & Concepts: Arangodb documentation. ArangoDB Data Model & Concepts | ArangoDB Documentation [online]. [Accessed 7 November 2021]. Available from: https://www.arangodb.com/docs/stable/data-modeling-concepts.html

[26] NoSQL database documentation. *RavenDB NoSQL Database* [online]. 7 June 2021. [Accessed 7 November 2021]. Available from: https://ravendb.net/docs/article-page/5.2/csharp/indexes/querying/graph/graph-queries-overview

[27] Welcome to ravendb: Inside ravendb. *RavenDB NoSQL Database* [online]. 12 May 2021. [Accessed 30 April 2022]. Available from: https://ravendb.net/learn/inside-ravendb-book/reader/4.0/9-querying-in-ravendb

[28] NoSQL database documentation. RavenDB NoSQL Database [online]. 24 February 2022. [Accessed 13 April 2022]. Available from: https://ravendb.net/docs/article-page/4.2/csharp/indexes/map-reduce-indexes

[29] Using mongodb as graph database: Use cases. *MongoDB* [online]. [Accessed 26 April 2022]. Available from: https://www.mongodb.com/databases/mongodb-graph-database

[30] NEO4J-contrib/neo4j_doc_manager: Doc manager for neo4j. *GitHub*. Retrieved April 26, 2022, from https://github.com/neo4j-contrib/neo4j_doc_manager

[31] Aggregation pipeline. *Aggregation Pipeline - MongoDB Manual* [online]. [Accessed 29 April 2022]. Available from: https://www.mongodb.com/docs/manual/core/aggregation-pipeline/

[32] $graphLookup (aggregation). $graphLookup (aggregation) - *MongoDB Manual* [online]. [Accessed 7 April 2022]. Available from: https://www.mongodb.com/docs/manual/reference/operator/aggregation/graphLookup/

[33] CPU-Z: Softwares, [no date]. CPUID [online], [Accessed 6 May 2022]. Available from: https://www.cpuid.com/softwares/cpu-z.html

[34] Memtest86 - official site of the x86 memory testing tool. *Official Site of the x86 and ARM Memory Testing Tool* [online]. [Accessed 6 May 2022]. Available from: https://www.memtest86.com/

[35] Introducing geekbench 5. *Geekbench 5 - Cross-Platform Benchmark* [online]. [Accessed 6 May 2022]. Available from: https://www.geekbench.com/

[36] Testing the performance of an IIS application. *Microsoft Docs* [online]. [Accessed 2 May 2022]. Available from: https://docs.microsoft.com/en-us/previous-versions/iis/6.0-sdk/ms524518(v=vs.90)

[37] DARMONT , Jérôme. Database Benchmarks. *arxiv.org* [online]. 27 January 2017. Available from: https://arxiv.org/abs/1701.08052

[38] COOPER, Brian Frank. Home · Brianfrankcooper/YCSB Wiki. *GitHub* [online]. [Accessed 2 May 2022]. Available from: https://github.com/brianfrankcooper/YCSB/wiki

[39] MISHRA, Omji, LODHI, Pooja and MEHTA, Shikha. Document oriented NoSQL databases: An empirical study. *Data Science and Analytics*. 2018. P. 126–136. DOI 10.1007/978-981-10-8527-7_12.

[40] WEINBERGER, Claudius. NoSQL performance benchmark 2018 – mongodb, PostgreSQL, OrientDB, neo4j and arangodb. *ArangoDB* [online]. 10 December 2020. [Accessed 2 May 2022]. Available from: https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/

[41] ZHANG, Chao, LU, Jiaheng, XU, Pengfei and CHEN, Yuxing. Unibench: A benchmark for Multi-model Database Management Systems. *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence*. 2019. P. 7–23. DOI 10.1007/978-3-030-11404-6_2.

[42] MACAK, Martin, STOVCIK, Matus, BUHNOVA, Barbora and MERJAVY, Michal. How well a multi-model database performs against its single-model variants: Benchmarking orientdb with neo4j and mongodb. *Proceedings of the 2020 Federated Conference on Computer Science and Information Systems*. 2020. DOI 10.15439/2020f76.

[43] HY-UDBMS/Unibench: Towards benchmarking multi-model DBMS. *GitHub* [online]. [Accessed 6 May 2022]. Available from: https://github.com/HY-UDBMS/UniBench

[44] SMITH, J. Patterns - WPF apps with the model-view-viewmodel design pattern. *Microsoft Docs* [online]. [Accessed 6 May 2022]. Available from: https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern

[45] Infarh/CV19. *GitHub* [online]. [Accessed 10 May 2022]. Available from: https://github.com/Infarh/CV19

[46] THIRIET, John. MVVM - going async with Async Command. *John Thiriet* [online]. 16 August 2019. [Accessed 10 May 2022]. Available from: https://johnthiriet.com/mvvm-going-async-with-async-command/

[47] TAKAC, L., ZABOVSKY, M. Data Analysis in Public Social Networks, *International Scientific Conference & International Workshop Present Day Trends of Innovations*, May 2012 Lomza, Poland.

[48] Regions of Slovakia. *slovakiacom* [online]. 30 November 2020. [Accessed 28 April 2022]. Available from: https://www.slovakia.com/regions/

Appendix A. Data scheme

The database used in the Chapter 2 for description of query language features consists of two collections. Examples of entries are shown below.

An example of the people collection entry:

```
{
	"Id":"people/1",
	"FirstName":"Alex",
	"LastName": "Springer",
	"Department":"Actors",
	"ReportsTo":"people/25",
	"FriendOf":["people/3", "people/6", "people/8"],
	"Tags":["2021", "marketing"],
	"Year":"1992",
	"BaseSalary":3000
}
```

An example of the projects collection entry:

```
{
	"Id":"projects/25",
	"Title":"Construction site #25",
	"Tags":["2021", "Nature"],
	"Workers":["people/0", "people/24"],
}
```

Appendix B. Jupyter notebook for data visualization

```
#open data files
with open("1.txt") as file:
    lines = file.readlines()
data = [float(i.replace(",", ".")) for i in lines]
with open("1_1.txt") as file:
    lines = file.readlines()
data2  = [float(i.replace(",", ".")) for i in lines]
with open("1_2.txt") as file:
    lines = file.readlines()
data3  = [float(i.replace(",", ".")) for i in lines]

#display graph
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings('ignore')
plt.figure(figsize=(15, 6))
plt.hist(data, range=(0,4), alpha = 0.5, bins=100, color = "skyblue")
plt.hist(data2, range=(0,4),  alpha = 0.5,bins=100, color = "orange")
plt.hist(data3, range=(0,4),  alpha = 0.5,bins=100, color = "green")

plt.legend(['ArangoDB', 'RavenDB', 'MongoDB'])
plt.ylabel("count", fontsize=16)
plt.xlabel("exec. time, s", fontsize=16)
plt.show()

#print percentiles
import numpy as np
print("ar1 : ",
       np.percentile(data, 95))
print("ar2 : ",
       np.percentile(data2, 95))
print("ar3 : ",
       np.percentile(data3, 95))
```

Appendix C. Zip-package

The package contains the following elements:

1. thesis.pdf – digital version of the thesis in PDF
2. app – the folder containing the application executable and dependencies. In subfolders DBMS plugins are placed.
3. source.zip – the archive with Microsoft Visual Studio project.