NDBI040 assignment - static and dynamic analysis of querying in multi-model database systems

1) Choose one multi-model database system preferably from this list:

ArangoDB, Citus, CrateDB, MonetDB, OrientDB, Couchbase, GridDB, RavenDB, Redis, Tarantool, YugabyteDB, PostgreSQL, ArcadeDB, ScyllaDB, TerminusDB, BangDB, MariaDB, MarkLogic, MySQL, Virtuoso

- Do not choose a system chosen by 2 other students.
    - See table with results
- You can choose any other multi-model system if you want.
    - See https://db-engines.com/en/ranking
- You can use the information from presentations from the previous year.
    - See nosql server, folder: /home/NDBI040/PRESENTATIONS
- You can cooperate if you want, but each of you should use your selected dataset and prepare your own presentation in the end.

Choose one of the following data sets:

- Yelp dataset: https://www.kaggle.com/datasets/yelp-dataset/yelp-dataset
- IMDb dataset: https://developer.imdb.com/non-commercial-datasets/

(The one that is not chosen yet for the selected DBMS by someone else.)

Send an e-mail with your choice.

Deadline: **1.4. 2024**

2) Perform a static analysis of the supported data models and query languages. Specifically, for each supported data model, determine the following:

- How is the data model implemented?
    - E.g., in ArangoDB the graph model is added to the document model by introducing a special document collection with graph edges connecting documents.
- How can different models be linked and worked with?
    - E.g., embedding - PostgreSQL allows JSONB, JSON and XML columns to be "embedded" into the relational model, but we cannot embed relational data into JSON or XML data in a similar way. I.e., you could say that this is one-way embedding.
- Are indexes over attributes represented in a particular data model supported?
    - E.g., can PostgreSQL index over attributes of nested JSON and XML document?
- What querying means are supported over the data expressed by the different supported models?
    - For example, create and complete a table like this:

Type of construct	Relational	Document (JSON)	Document (XML)
Projection	SELECT	?	?
Source	FROM	?	?
Selection	WHERE	?	?
Aggregation/aggregation functions	GROUP BY ... HAVING	?	?
Join	JOIN	?	?

Graph Traversal X (JOIN)	JOIN	?	?
Unlimited Traversal	WITH RECURSIVE	?	?
Optional	OUTER JOIN	?	?
Union	UNION	?	?
Intersection	INTERSECTION	?	?
The Difference	EXCEPT	?	?
Sorting (on non/indexed columns)	ORDER BY	?	?
Skipping	OFFSET	?	?
Limitation	LIMIT	?	?
Distinct	DISTINCT	?	?
Aliasing	AS	?	?
Nesting	(SELECT ...)	?	?
MapReduce	(GROUP BY ... HAVING)	?	?

Note: The database systems may have different numbers of supported models. Choose two, ideally, the ones that show the most multi-model features. Leave out the trivial key/value model.

3) Perform deployment of the DBMS (e.g. in Docker) and perform an experimental analysis of the querying:

For example, deploy a database system in Docker, prepare the dataset (IMDb/Yelp) for querying and measure the times needed to execute each query (3.1 - 7) below, if possible. Run each query 20 times, remove the minimum/maximum and then determine the average time (or standard deviation).

Assumptions:

- Queries can be trivial and should be trivial, i.e. don't use aggregation in a projection query, don't add join in an aggregation query, etc.
- The goal is to verify the performance of a particular aspect of querying, e.g. just aggregation (independent of other constructs) or conjunction

3.1) Selection, Projection, Source of data

3.1.1) Filtering on a non-indexed column, exact match, e.g., select actor with the name "Arnold" (attribute name is not indexed!)

```
SELECT * FROM Actors WHERE name = "Arnold";
```

3.1.2) Filtering on a non-indexed column, range query, e.g., select actors with salary between 25000 and 35000 (once again, column "salary" is not indexed!)

```
SELECT * FROM Actors WHERE salary BETWEEN '25000' AND '35000';
```

3.1.3) Filtering on indexed column, exact match, e.g., select actor with age equal to 30 (attribute age is indexed!)

```
SELECT * FROM Actors WHERE age = 30;
```

3.1.4) Filtering on indexed column, range query, e.g., select actors with age between 30 and 45 (attribute age is indexed!)

```
SELECT * FROM Actors WHERE age BETWEEN '30' AND '45';
```

3.2) Aggregation

3.2.1) Use aggregation function count, e.g., count number of actors per age

```
SELECT age, COUNT(*) AS actorsCount FROM Actors GROUP BY age;
```

3.2.2) Similarly, express a query for aggregate function MAX

3.3) Join (or graph traversal)

3.3.1) Joining / traversal where two entities are connected by non-indexed columns, e.g., join movies and actors where movies were filmed in the same year as actor was born

```
SELECT *
FROM Actors AS a INNER JOIN Movies AS m ON a.year_of_birth = m.year;
```

3.3.2) Joining / traversal over indexed column, e.g., find names of all actors in each movie

3.3.3) Complex join involving multiple JOINS, e.g., over 5+ tables or graph traversal over 3 types of node labels and 2 types of edge labels

Complex query with JOINS to retrieve order details

```
SELECT *
FROM Actor a
         JOIN
     Acts am ON a.customerId = aa.customerId
         JOIN
     Movie m ON am.movieId = m.movieId
         JOIN
     Directed dm ON m.movieId = dn.movieId
         JOIN
     Director d ON d.directorId = dm.directorId;
```

```
MATCH (a:Actor)-[am:Acts]->(m:Movie)-[dn:Directed]->(d:Director)
RETURN ...
```

3.3.4) Recursive query, e.g., find all direct and indirect relationships between people

SQL: WITH RECURSIVE query

3.3.5) Optional traversal

SQL: LEFT OUTER JOIN
Cypher: OPTIONAL MATCH

Get a list of all people and their friend count (0 if they have no friends)

```
SELECT P1.personId,
       P1.firstName,
       P1.lastName,
```

```
       COUNT(P2.personId) AS friendCount
FROM Person P1
         LEFT OUTER JOIN Person_Person PP on P1.personId = PP.personId1
         LEFT OUTER JOIN Person P2 on PP.personId2 = P2.personId
GROUP BY P1.personId;
```

3.4) Set operations

3.4.1) Union, e.g., get a list of contacts (email and phone) for both Actors and Directors

3.4.2) Intersection, e.g., get a list of shared contacts between actors and directors

3.4.3) Difference, e.g., find a list of contacts that are exclusive for directors (e.g., no actors has the same contact)

3.5) Sorting

3.5.1) Sorting over non-indexed column, e.g., sort actors by salary

```
SELECT * FROM Actors ORDER BY salary;
```

3.5.2) Sorting over indexed column, e.g., sort actors by age

```
SELECT * FROM Actors ORDER BY age;
```

3.6) Distinct

3.6.1) Apply distinct, e.g., find unique combinations of name,surname in the table of Actors

```
SELECT DISTINCT name, surname FROM Actors;
```

3.7) MapReduce (or equivalent aggregation), e.g., find the number of movies played by actor and only those who have played at least in 1 movie

The result of this phase will be a table/graph with average of measured times, e.g.:

	Relational	**Document (JSON)**	**Document (XML)**
1.1) Filtering on non-Indexed column, exact match	?	?	?
1.2) Filtering on non-Indexed column, range query	?	?	?
1.3) Filtering on indexed column, exact match	?	?	?
1.4) Filtering on indexed column, range query	?	?	?
2.1) Use aggregation function COUNT	?	?	?
2.2) Use aggregation function MAX	?	?	?
3.1) Joining / traversal where two entities are	?	?	?

connected by non-indexed columns			
3.2) Joining / traversal over indexed column (two entities)	?	?	?
3.3) Complex join involving multiple JOINS, e.g., over 5+ tables or graph traversal	?	?	?
3.4) Recursive query	?	?	?
3.5) Optional traversal	?	?	?
4.1) Union	?	?	?
4.2) Intersection	?	?	?
4.3) Difference	?	?	?
5.1) Sorting over non-indexed column	?	?	?
5.2) Sorting over indexed column	?	?	?
6.1) Apply distinct	?	?	?
7) MapReduce	?	?	?

Notes:

- If any of the queries cannot be expressed, provide a reason (with a link to the documentation, for example) why the query cannot be executed.
- You don't have to use the whole data set if your installation can't handle it. For example, 1,000,000 records are enough. In general, use the maximum amount that allows you to get a result in a reasonable time.
- The specific times you measure will not be evaluated, so you don't have to stress if the system is inefficient.

4) Submit:

- A script that installs a database system or deploys a system in Docker
- Transformation scripts that convert the selected dataset so that it can be imported into the database system. Represent the dataset appropriately, using each of the selected data models. For example, if the DBMS supports 3 data models (relational, JSON, and XML) and you select relational model and JSON document model, then represent the data as CSV and JSON.
- Scripts that import the data to the database
- List of executed queries
- A script that executes queries and measures the time it takes to execute them
- Table (excel or csv) containing all measured times
- A final presentation (i.e., 10-20 minute video) summarising deployment, the static properties of the selected DBMS and the results of the experiments (i.e., the static and experimental analysis)

Submit the complete solution by creating a subdirectory

NDBI040_<system name>_<dataset name>

(e.g. NDBI040_OracleDB_IMDb) in your home directory on the nosql server and storing all parts of the solution there.

Deadline: **5.5. 2024**

5) Evaluate the work of your colleagues:

Shortly after the submission deadline you will be assigned with 2 works of your students to evaluate.

Deadline: **19.5. 2024**

Note: If you encounter any technical difficulties (e.g., with installation, data transformation, queries, etc.), you can contact Pavel Koupil (pavel.koupil@matfyz.cuni.cz).