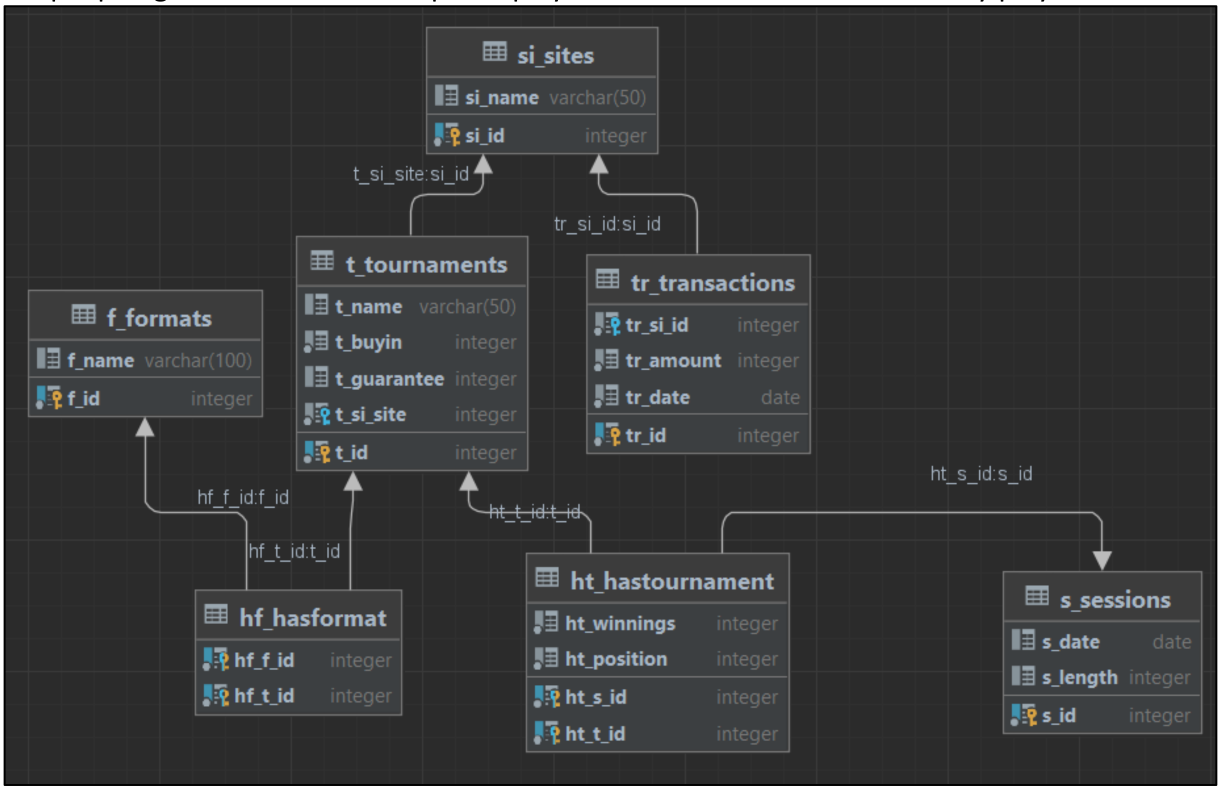
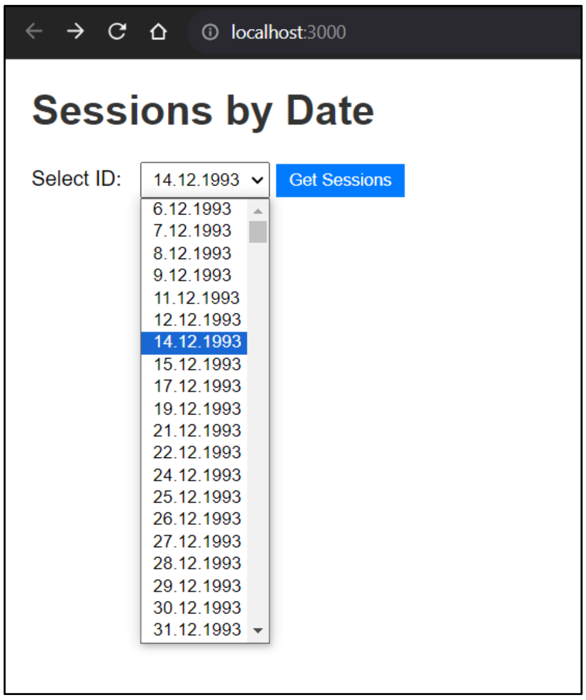
Schritt 1

Simple postgres-db for an online poker player to track the tournaments they play:

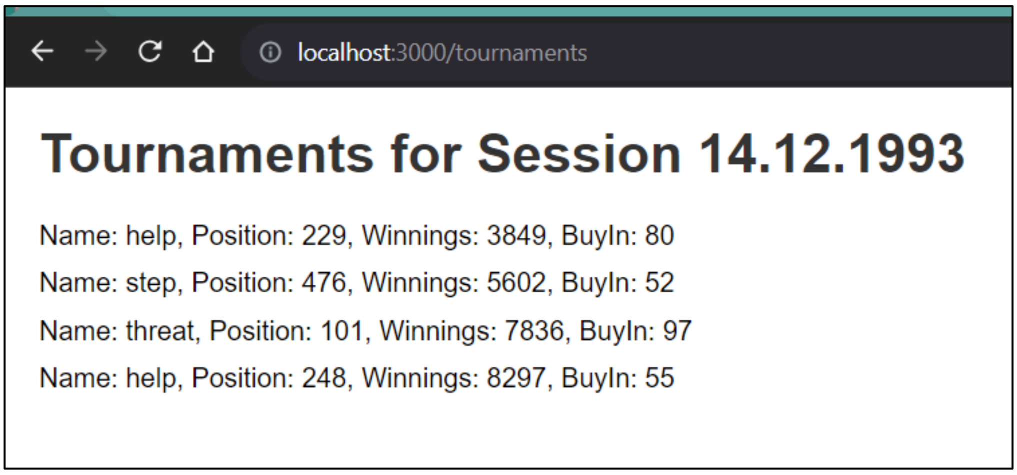


For the frontend we have the functionality that the user can select the date of a session from a drop down to then see the tournaments they played on that day and the results they achieved:



Sessions by Date

Pressing Get Sessions leads to:



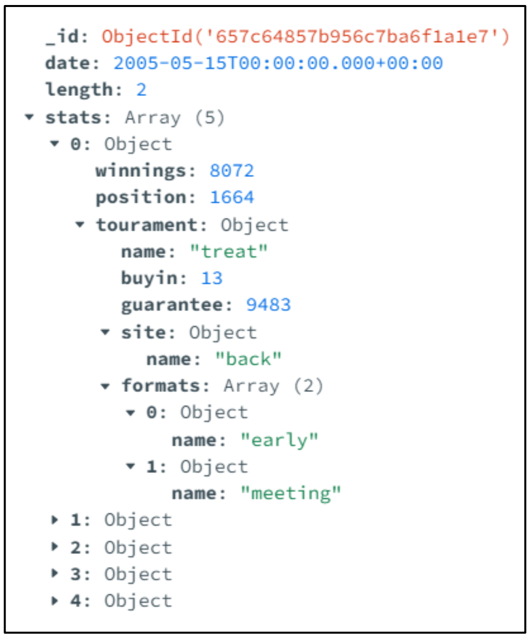
localhost:3000/tournaments

Tournaments for Session 14.12.1993

```
Name: help, Position: 229, Winnings: 3849, BuyIn: 80
Name: step, Position: 476, Winnings: 5602, BuyIn: 52
Name: threat, Position: 101, Winnings: 7836, BuyIn: 97
Name: help, Position: 248, Winnings: 8297, BuyIn: 55
```

Schritt 2

Model can be found here. To fulfill all the criteria, we wrote a runnable implementation like Mr. Schletz's generating data here.
Sample data in DB:



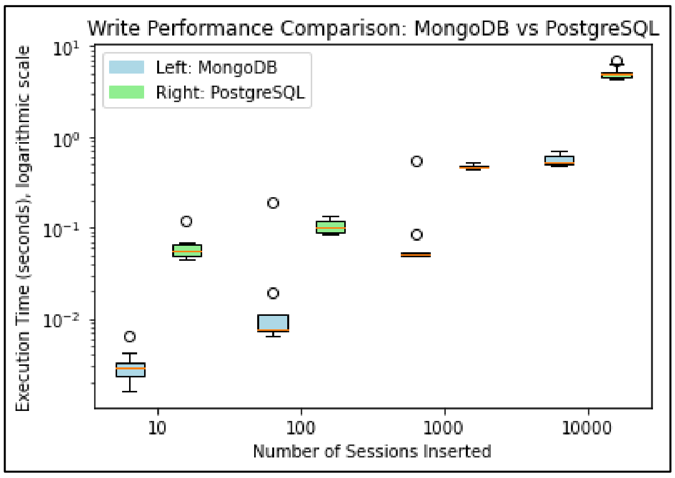
```
  _id: ObjectId('657c64857b956c7ba6f1a1e7')
  date: 2005-05-15T00:00:00.000+00:00
  length: 2
▾ stats: Array (5)
  ▾ 0: Object
      winnings: 8072
      position: 1664
    ▾ tourament: Object
        name: "treat"
        buyin: 13
        guarantee: 9483
      ▾ site: Object
          name: "back"
      ▾ formats: Array (2)
        ▾ 0: Object
            name: "early"
        ▾ 1: Object
            name: "meeting"
  ▸ 1: Object
  ▸ 2: Object
  ▸ 3: Object
  ▸ 4: Object
```

Schritt 3

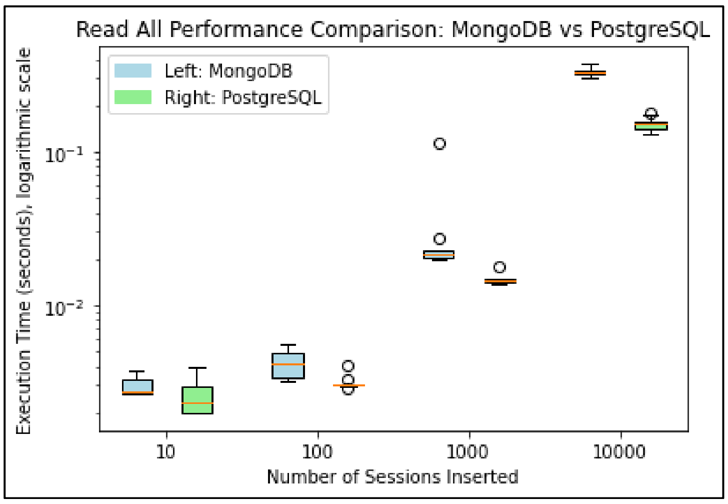
Runnable with the Jupyter Notebook PerformanceComparison.ipynb

Write Test

We insert 10, 100, 1000, 10000 sessions, each with 1-5 tournaments. These inserts are the basis for all future tests.

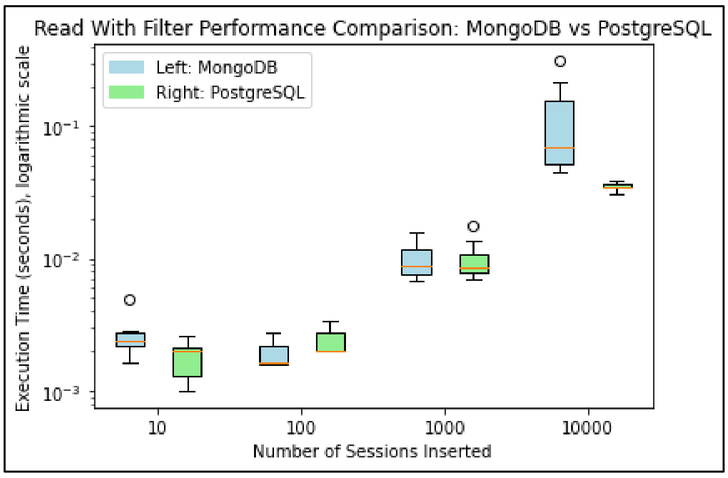


Read All



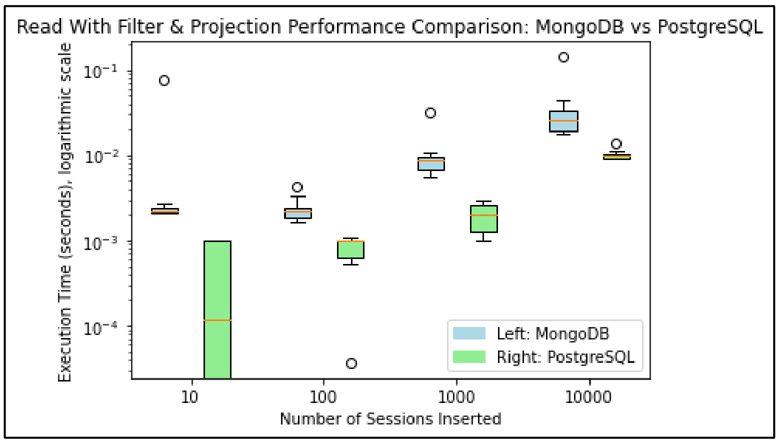
Read With Filter

We Filter by the session length, selecting all sessions with the length of 1, which should be 20%



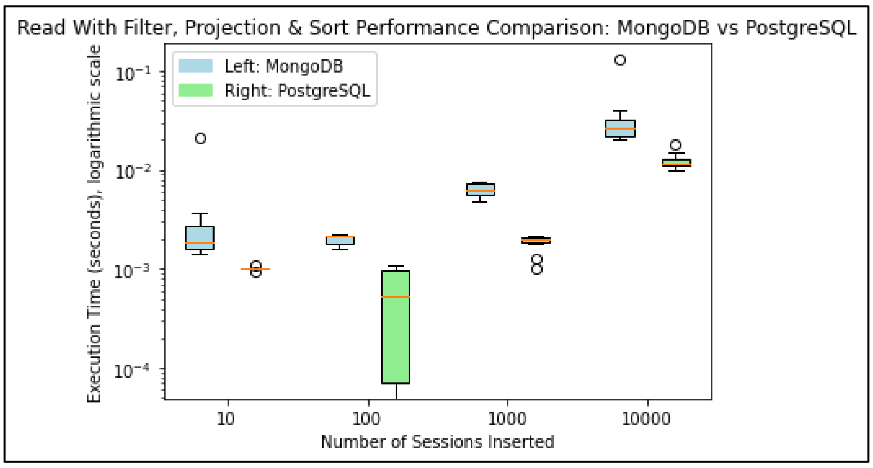
Read With Filter & Projection

We project so that we only select a subset of the data provided in sessions and tournaments. Due to this we also use fewer joins in the sql implementation as we are not getting all the data.



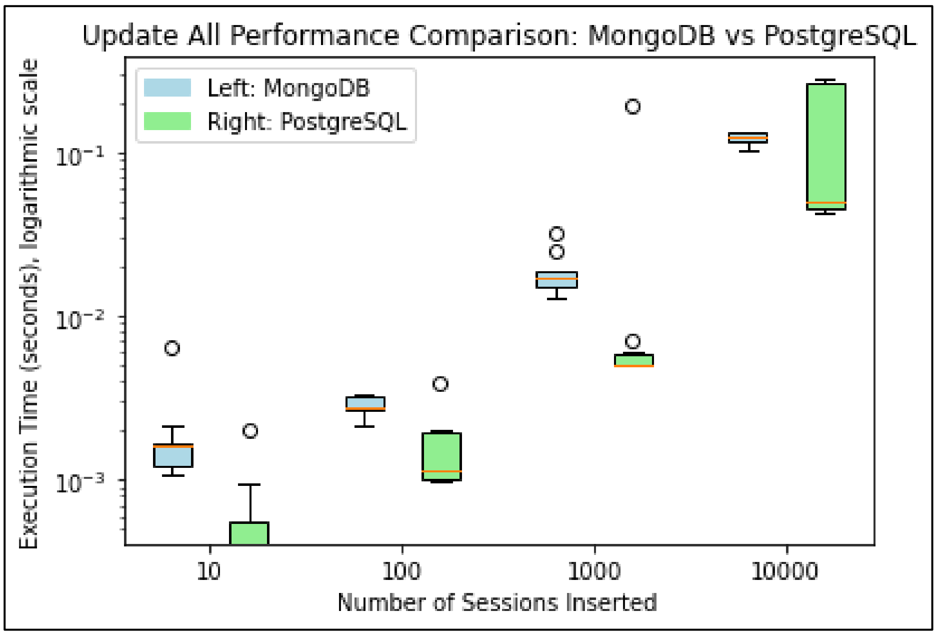
Read With Filter, Projection & Sorting

We add sorting by the date.



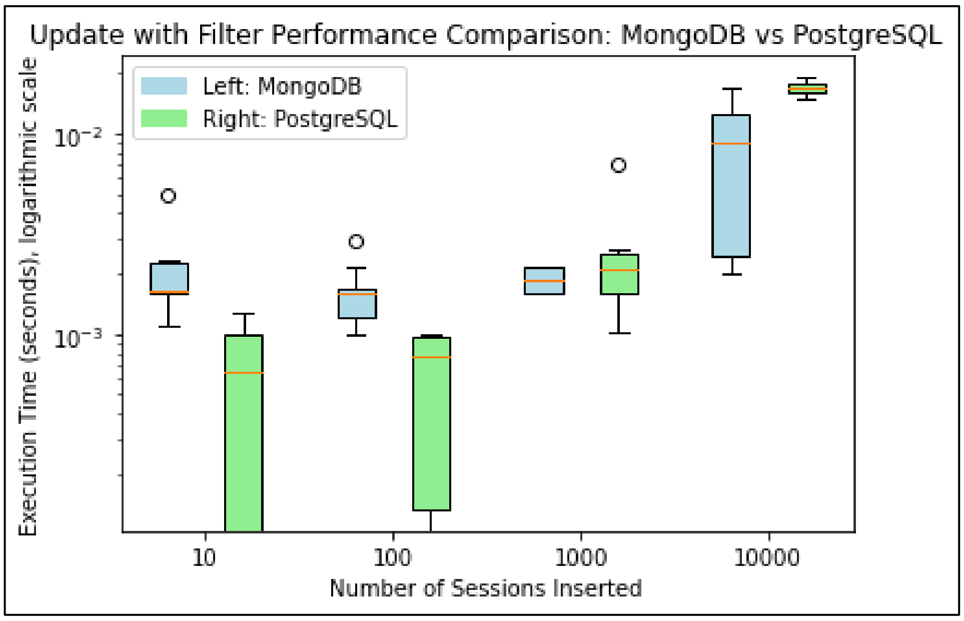
Update All

We increment every length of the session by one.

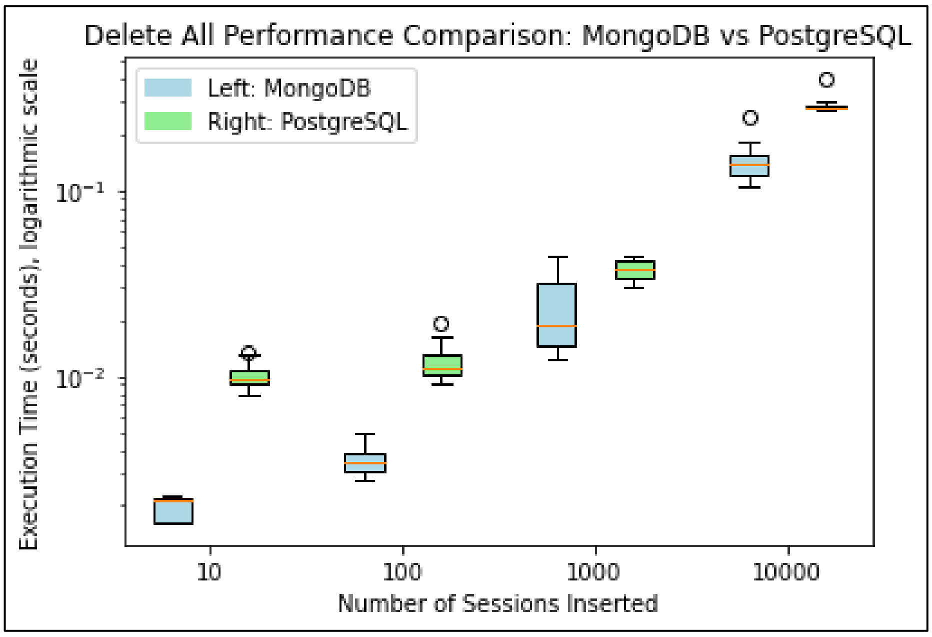


Update With Filter

We only update the sessions with a length of 1, approximately 20%.



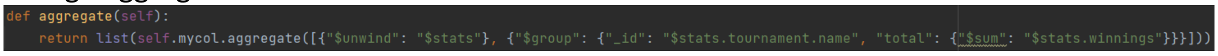
Delete All



Bonus

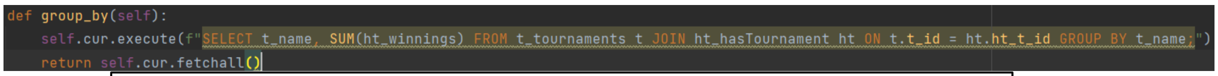
1. Aggregation

Mongo aggregate:

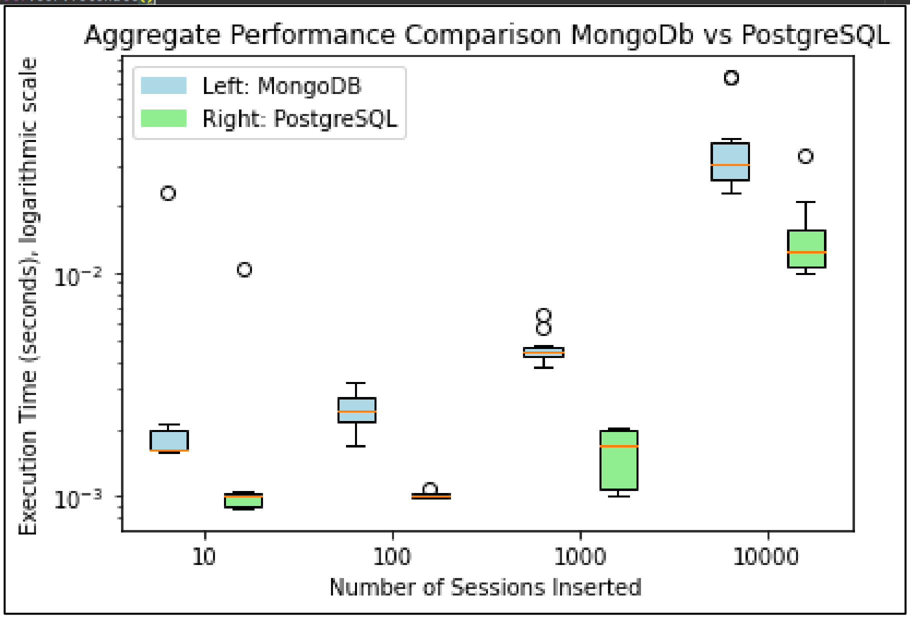


```
def aggregate(self):
    return list(self.mycol.aggregate([{"$unwind": "$stats"}, {"$group": {"_id": "$stats.tournament.name", "total": {"$sum": "$stats.winnings"}}}]))
```

SQL version:



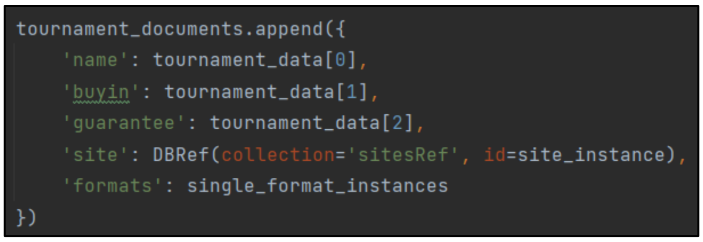
```
def group_by(self):
    self.cur.execute(f"SELECT t_name, SUM(ht_winnings) FROM t_tournaments t JOIN ht_hasTournament ht ON t.t_id = ht.ht_t_id GROUP BY t_name;")
    return self.cur.fetchall()
```



2. Referencing

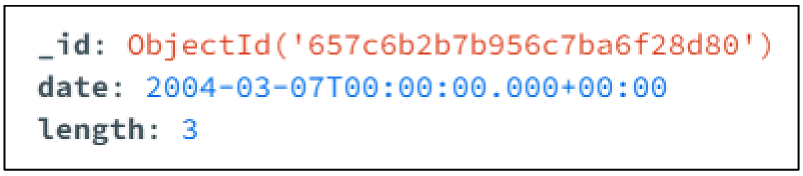
Model

The adapted model can be found here. Compared to the old one there isn't too much change, except that N:M can now be modelled properly. The data faker for referencing can be found here. This uses the bson DBref feature:



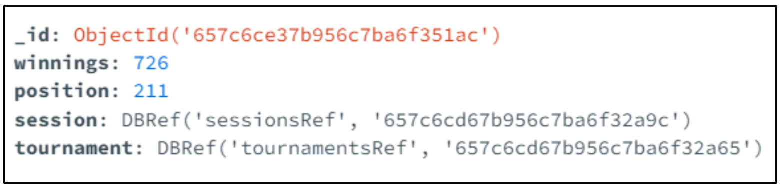
```
tournament_documents.append({
    'name': tournament_data[0],
    'buyin': tournament_data[1],
    'guarantee': tournament_data[2],
    'site': DBRef(collection='sitesRef', id=site_instance),
    'formats': single_format_instances
})
```

In the new model the session looks like this:



```
_id: ObjectId('657c6b2b7b956c7ba6f28d80')
date: 2004-03-07T00:00:00.000+00:00
length: 3
```

While then the session stats collection works with references:

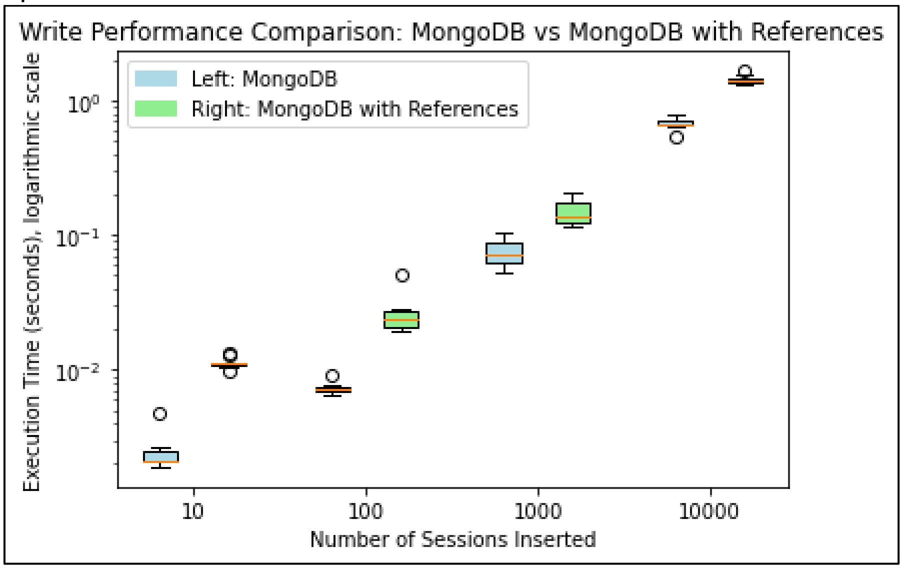


```
_id: ObjectId('657c6ce37b956c7ba6f351ac')
winnings: 726
position: 211
session: DBRef('sessionsRef', '657c6cd67b956c7ba6f32a9c')
tournament: DBRef('tournamentsRef', '657c6cd67b956c7ba6f32a65')
```

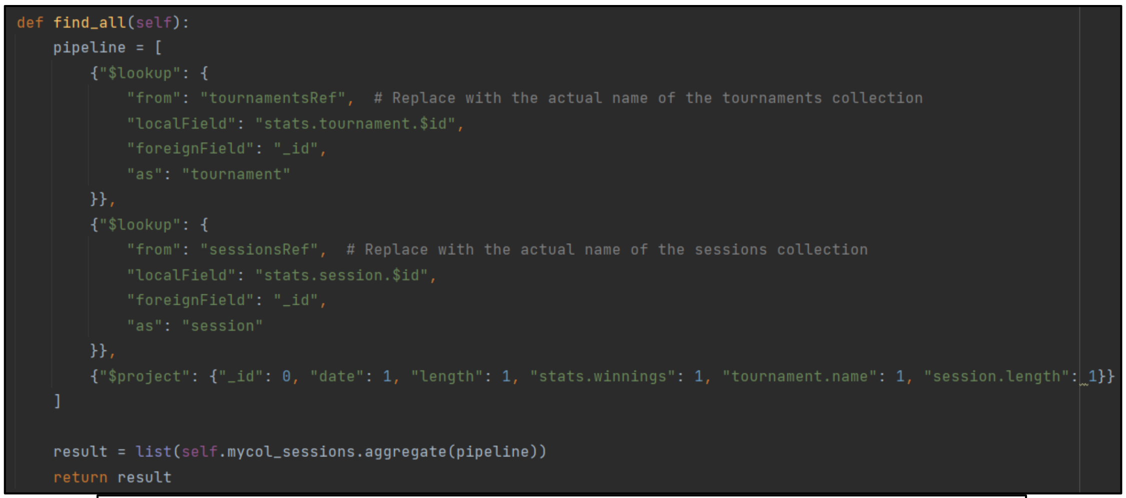
NOTE: We are using **pyMongo** which is not a proper ORM like **mongoengine**. This means that there is no domain model-level support for DBRef. Luckily there is still support for manual db-referencing when making inserts, which we will use.

Performance Comparison

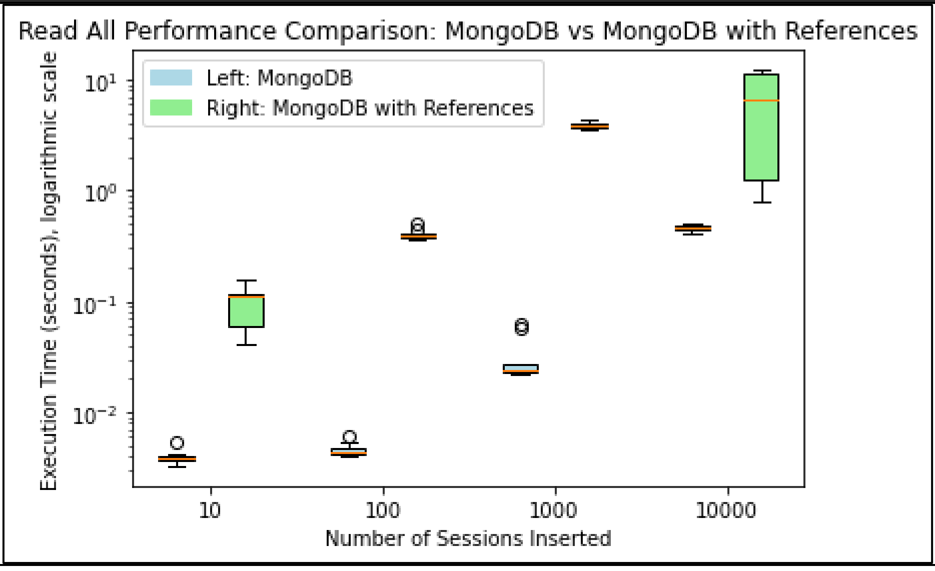
The use of references leads to performance slowdowns across the board. One thing to note is that the inserts are not 100% comparable, as with the base mongo implementation the data is first prepared in document form and only the insertion time is measured. For the referencing, however, this was not feasible and there is likely a bit of overhead in those numbers which is not directly attributed to the database performance.

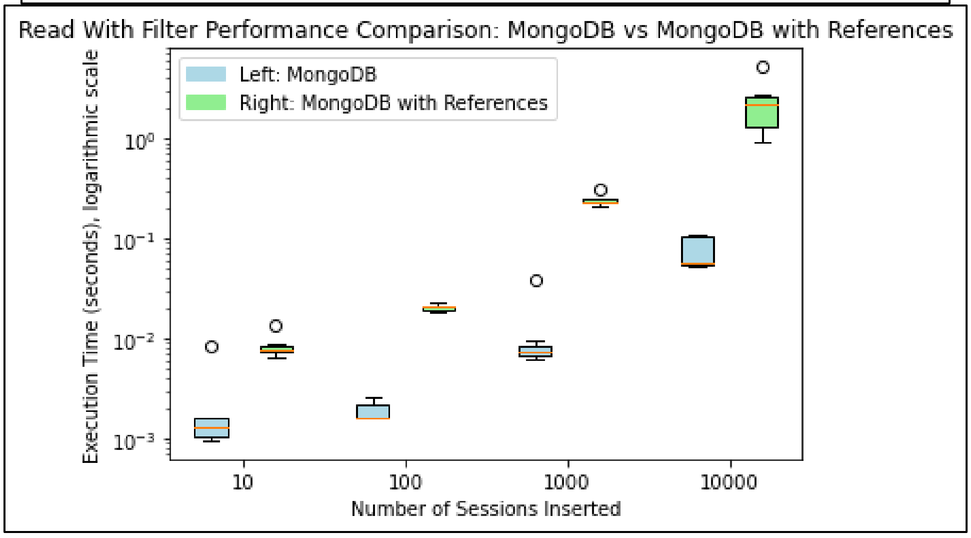


For the queries we used the lookup operation:



```python
def find_all(self):
    pipeline = [
        {"$lookup": {
            "from": "tournamentsRef",  # Replace with the actual name of the tournaments collection
            "localField": "stats.tournament.$id",
            "foreignField": "_id",
            "as": "tournament"
        }},
        {"$lookup": {
            "from": "sessionsRef",  # Replace with the actual name of the sessions collection
            "localField": "stats.session.$id",
            "foreignField": "_id",
            "as": "session"
        }},
        {"$project": {"_id": 0, "date": 1, "length": 1, "stats.winnings": 1, "tournament.name": 1, "session.length": 1}}
    ]

    result = list(self.mycol_sessions.aggregate(pipeline))
    return result
```

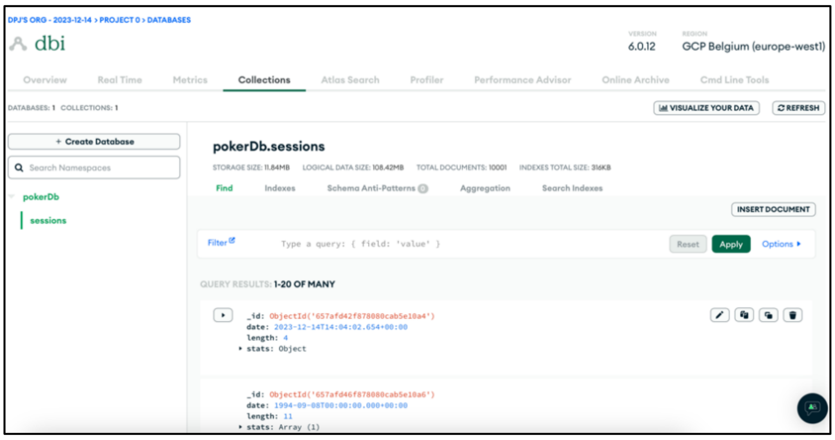




3. Cloud

General

We did give Atlas Cloud a try and it works amazingly:



VERSION 6.0.12 REGION GCP Belgium (europe-west1)

DATABASES: 1 COLLECTIONS: 1

pokerDb.sessions

STORAGE SIZE: 11.84MB LOGICAL DATA SIZE: 108.42MB TOTAL DOCUMENTS: 10001 INDEXES TOTAL SIZE: 316KB

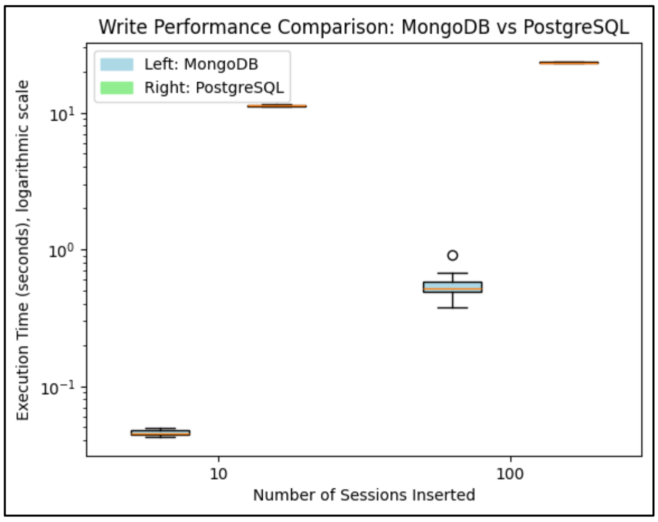
QUERY RESULTS: 1-20 OF MANY

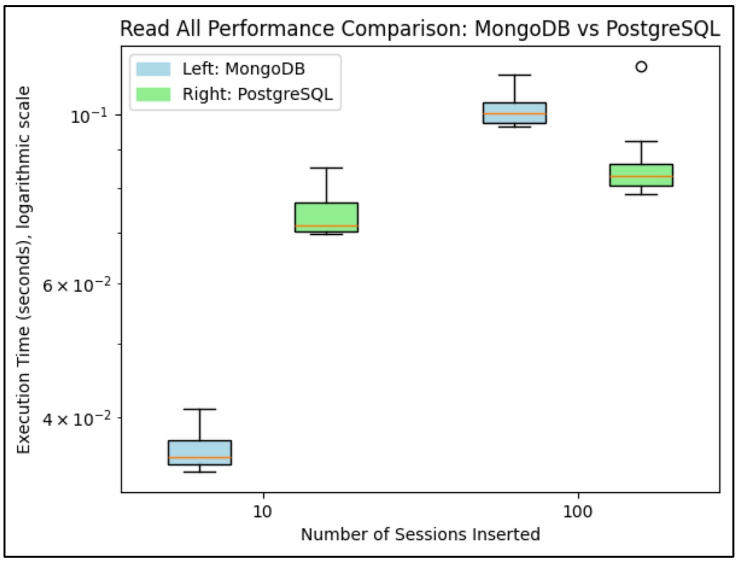
```
_id: ObjectId('657afd42f878080cab5e10a4')
date: 2023-12-14T14:04:02.654+00:00
length: 4
▸ stats: Object

_id: ObjectId('657afd46f878080cab5e10a6')
date: 1994-09-08T00:00:00.000+00:00
length: 11
▸ stats: Array (1)
```

For the relational DB we opted not to use Azure because we didn't have credits left. We therefore rely on Google Cloud. More details on the infrastructure are to be found the Something Cool section of this document.

Benchmark

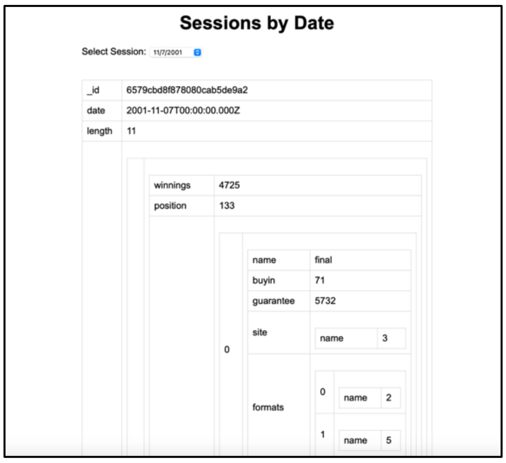
(note that our google cloud instance does have very limited compute resources; especially when compared to a database as a service solution like Atlas. Also, the network forwarding is a lot more complex – all this impacts the results – so take with a big grain of salt):





4. Mongo Frontend

Mongo Frontend can be viewed in here. As requested, we implemented the **filter feature**. As discussed with you, because this is the frontend-optimized DB version the filtering is done on the frontend as it is provided with all the sessions. The drawbacks of this approach are obvious, more data more loading time, but for frontend dev experience this is the simplest, easiest/most optimized.



Sessions by Date

Select Session: 11/7/2001

_id	6579cbd8f878080cab5de9a2
date	2001-11-07T00:00:00.000Z
length	11

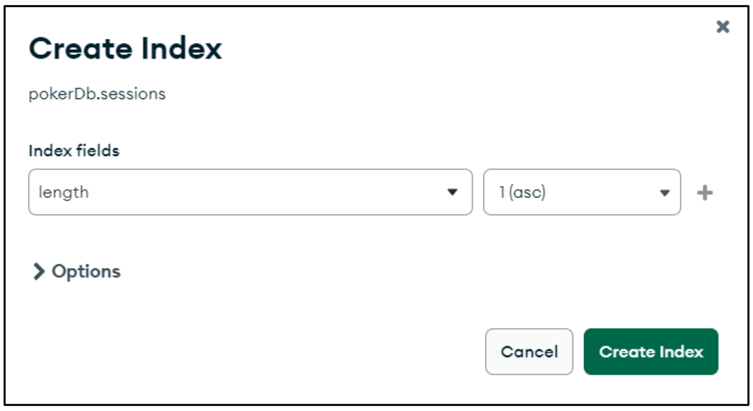
winnings	4725
position	133

0

name	final
buyin	71
guarantee	5732
site	name 3
formats	0 name 2; 1 name 5

5. Index Runtime

Since in our testing we filtered by length we decided to create an index on the length. This was done using the MongoDB compass app:

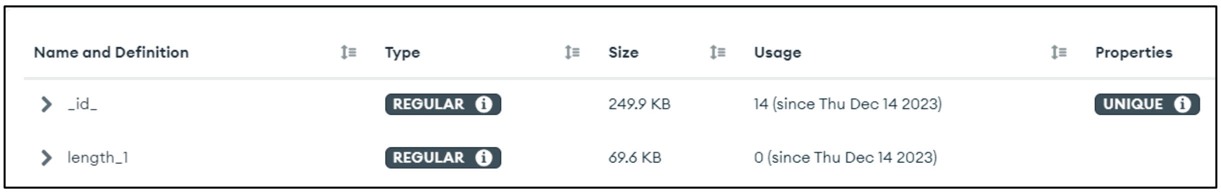


Create Index

pokerDb.sessions

Index fields

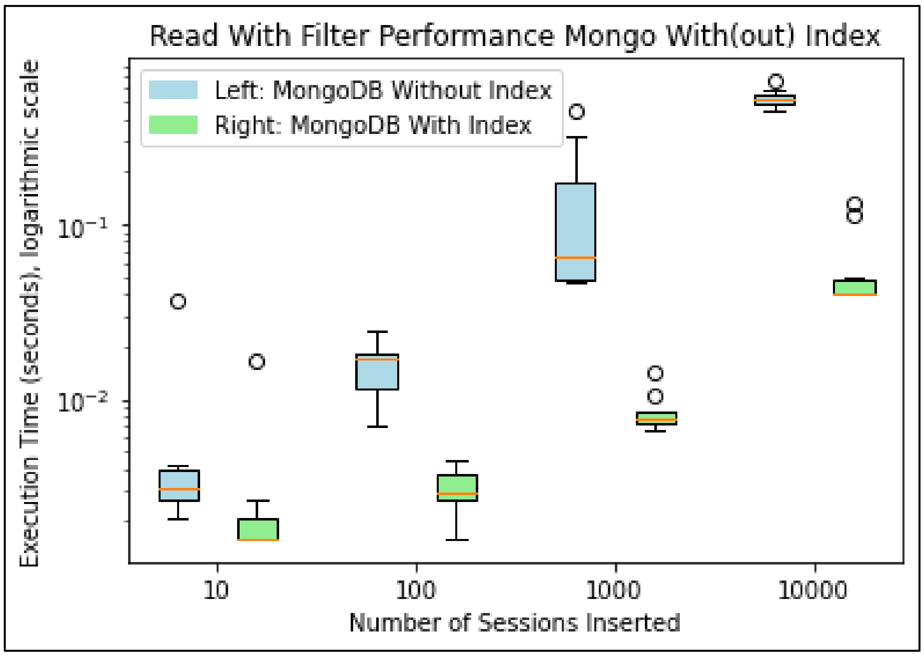
length — 1 (asc)



Name and Definition	Type	Size	Usage	Properties
_id_	REGULAR	249.9 KB	14 (since Thu Dec 14 2023)	UNIQUE
length_1	REGULAR	69.6 KB	0 (since Thu Dec 14 2023)	

Test

Testing was again done in the Jupyter notebook. The results of a run without indexing where saved and compared to those after indexing. An increase in performance can be seen.

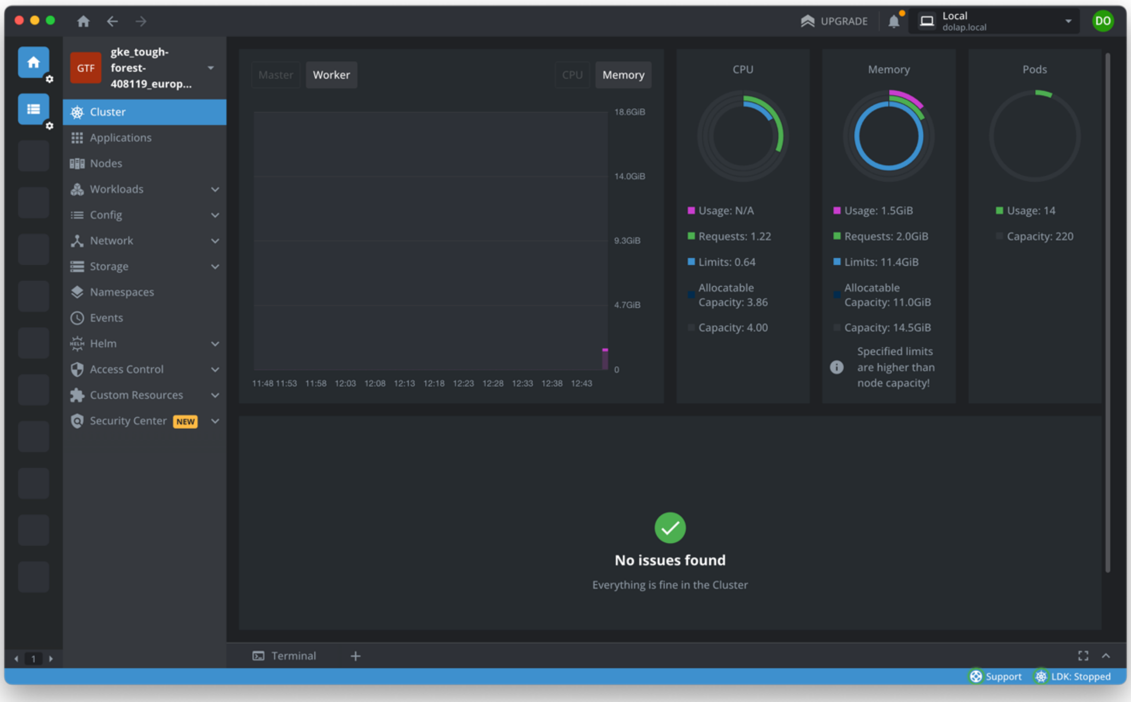


6. Something Cool

We use Google Cloud to run our relational database. To make things **even cooler** we don't simply run the DB as a service but rather in Kubernetes environment as a full deployment! The deployment files can be found here.

Our cluster:

```
NAME        LOCATION        MASTER_VERSION     MASTER_IP      MACHINE_TYPE   NODE_VERSION       NUM_NODES  STATUS
schmangie   europe-west1-b  1.28.3-gke.1203001 35.195.228.76  n1-standard-2  1.28.3-gke.1203001 2          RUNNING
```



CPU

- Usage: N/A
- Requests: 1.22
- Limits: 0.64
- Allocatable Capacity: 3.86
- Capacity: 4.00

Memory

- Usage: 1.5GiB
- Requests: 2.0GiB
- Limits: 11.4GiB
- Allocatable Capacity: 11.0GiB
- Capacity: 14.5GiB

Specified limits are higher than node capacity!

Pods

- Usage: 14
- Capacity: 220

No issues found

Everything is fine in the Cluster