Recenzje gier komputerowych

Skład zespołu

- Karol Hamielec
- Michał Janeczko

Źródło

Steam

Cel projektu

Celem projektu jest porównanie działania różnych rodzai baz danych. W realizowanym projekcie użyliśmy jednej bazy relacyjnej, którą jest PostgreSQL oraz dwóch baz nierelacyjnych, którymi są Mongo oraz Redis. Bazy mają zawsze taką samą ilość identycznych danych, aby osiągnąć jak najbardziej realistyczne wyniki. Testy są napisane w sposób identyczny dla każdej bazy oraz sprawdzają własności baz takie jak: search, save i delete.

Dane

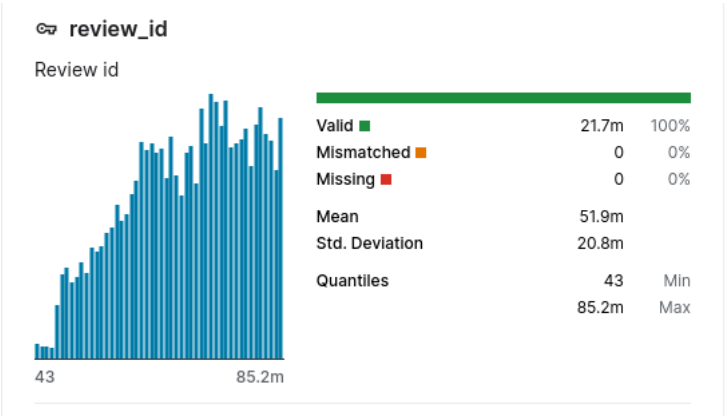
Dane użyte w projekcie są dostępne na stronie https://www.kaggle.com/datasets/najzeko/steam-reviews-2021 w formacie csv. Dane zajmują łącznie 8.17GB i obejmują recenzje gier komputerowych z 2021 dla ponad 300 gier dostępnych na tej platformie w wielu językach. Dane składają się z 23 kolumn:

1. # - Index
2. app_id – numer id gry w platformie
3. app_name – nazwa gry na platformie
4. review_id – numer id recenzji
5. language – język, w którym została napisana recenzja
6. review – tekst recenzji
7. timestamp_created – czas utworzenia recenzji
8. timestamp_updated – ostatnia aktualizacja recenzji
9. recommended – czy recenzja poleca grę
10. votes_helpful – ilość głosów, że recenzja była pomocna
11. votes_funny – ilość głosów, że recenzja była zabawna
12. weighted_vote_score – wynik obliczany na podstawie ilości pomocnych głosów
13. comment_count – ilość komentarzy do recenzji
14. steam_purchase – stwierdzenie, czy autor kupił grę na steamie
15. received_for_free – stwierdzenie, czy autor recenzji otrzymał grę za darmo
16. written_during_early_access – stwierdzenie, czy recenzja była napisana na początku dostępu do gry
17. author.steamid – id autora recenzji
18. author.num_games_owned – ilość gier posiadanych przez autora recenzji
19. author.num_reviews – łączna ilość recenzji autora

20. author.playtime_forever- całkowity czas poświęcony w recenzowanej grze przez autora recenzji
21. author.playtime_last_two_weeks – czas odtwarzania recenzowanej gry przez autora recenzji w ostatnich 2 tygodniach
22. author.playtime_at_review – czas poświęcony na grę przez autora do czasu wystawienia recenzji
23. author.last_played – czas kiedy autor recenzji ostatni raz grał w recenzowaną grę

Kluczami w tej tablicy danych są wartości app_id, review_id i author_id. Dzięki temu można jednoznacznie określić recenzję do której chcemy się odwołać.

Oprócz tego na powyższej stronie możemy porównać statystyki dla każdej kolumny, tzn. czy nie ma uszkodzeń danych, wartości średnie, odchylenie standardowe, kwantyle, unikalne wartości, wartości najczęściej występujące itp.



Wybrane bazy danych

- PostgreSQL – obok MySQL i SQLite, jeden z najpopularniejszych otwartych systemów zarządzania relacyjnymi bazami danych.
- MongoDB – otwarty, nierelacyjny system zarządzania bazą danych. Charakteryzuje się brakiem ściśle zdefiniowanej struktury obsługiwanych baz danych. Zamiast tego dane składowane są jako dokumenty w stylu JSON.
- Redis – otwartoźródłowe oprogramowanie działające jako nierelacyjna baza danych przechowująca dane w strukturze klucz-wartość w pamięci operacyjnej serwera, przeznaczona do działania jako klasyczna baza danych lub rozproszony cache.

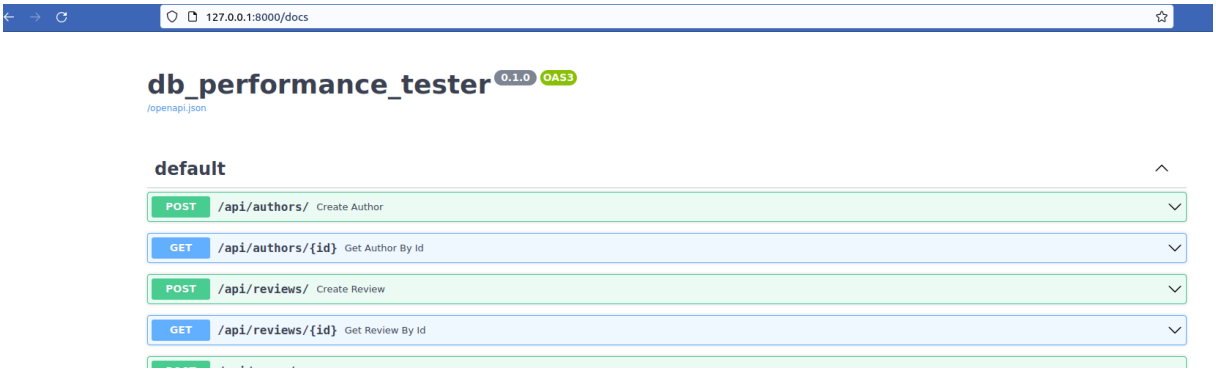
Wykorzystane oprogramowanie

- Git - rozproszony system kontroli wersji
- Docker - otwarte oprogramowanie służące do konteneryzacji aplikacji
- Python – język programowania na którym opierał się projekt
- FastApi – biblioteka do tworzenia aplikacji webowych w języku Python
- Node.js – wieloplatformowe środowisko uruchomieniowe do uruchamiania skryptów w języku JavaScript
- Angular – framework do tworzenia aplikacji frontendowych w języku TypeScript

- SQLAlchemy - biblioteka programistyczna napisana w języku programowania Python, służąca do pracy z bazami danych oraz mapowania obiektowo-relacyjnego. W tym przypadku została użyta do bazy PostgreSQL
- PyMongo - dystrybucja Pythona zawierająca narzędzia do pracy z MongoDB
- Redis-Py – biblioteka służąca do komunikacji z bazą Redis
- Ngx-charts – biblioteka dla Angulara służącą do wizualizacji danych za pomocą wykresów

Konfiguracja środowiska deweloperskiego

1. Instalacja brakujących programów lub aktualizacja istniejących: git, python i docker
2. Sklonowanie kodu z repozytorium
   git clone https://github.com/lewelyn7/ztbd_project.git
3. Przejście do katologu ztpd_project
4. Przy pierwszym uruchomieniu zbudowanie obrazu dockera
   docker compose build
5. Po zbudowaniu obrazu uruchomienie obrazu
   docker compose up
6. W nowym terminalu przechodzimy do katalogu backend
7. Instalujemy potrzebne paczki
   pip install -r requirements.txt
8. Uruchomiamy aplikacje
   uvicorn app.main:app --reload
9. W przeglądarce sprawdzamy, czy w API zostały załadowane dostępne metody i testy
   http://127.0.0.1:8000/docs
   Rezultat powinien wyglądać podobnie do zamieszczonego niżej:



10. Przechodzimy do katalogu frontend/db-perf-tester/
11. Instaluje pakiety do widoku aplikacji
    npm install
12. Włączamy aplikacje
    npx ng serve
13. Sprawdzamy, czy pod wskazanym linkiem (http://localhost:4200/) znajduje się aplikacja z wszystkimi testami wydajnościowymi
14. Uruchomiamy interesujące nas testy

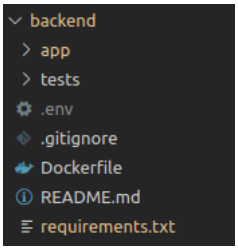
Kod dostępny w projekcie

W repozytorium znajduje się podział na kilka części.

W folderze domyślnym znajdują się pliki do zbudowania obrazu Dockeru, zbudowania angulara oraz stworzenia środowiska

```
FROM node:18-alpine AS angular-build
WORKDIR /usr/src/app
COPY frontend/db-perf-tester/*.json ./
RUN npx npm install
RUN npx npm install @angular/cli
COPY frontend/db-perf-tester/src ./src
RUN ls -al
RUN npx ng build --build-optimizer --base-href="/static/"

FROM tiangolo/uvicorn-gunicorn-fastapi:python3.9
ENV PYTHONPATH "${PYTHONPATH}:/"
ENV PORT=8000
RUN pip install --upgrade pip
COPY ./backend/requirements.txt /app/
RUN pip install -r requirements.txt
COPY ./backend/app /app
COPY ./backend/.env /app/.env
COPY --from=angular-build /usr/src/app/dist/db-perf-tester/* /app/static/
```

W folderze backend znajduje się cała aplikacja od tej strony w folderze app. Oprócz tego znajduje się plik z wymaganymi bibliotekami z pythona (requirements.txt) oraz Dockerfile, który zawiera szereg instrukcji do stworzenia w pełni funkcjonalnego obrazu




```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.9

ENV PYTHONPATH "${PYTHONPATH}:/"
ENV PORT=8000

RUN pip install --upgrade pip

COPY ./requirements.txt /app/

RUN pip install -r requirements.txt

COPY ./app /app
```

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.9

ENV PYTHONPATH "${PYTHONPATH}:/"
ENV PORT=8000

RUN pip install --upgrade pip

COPY ./requirements.txt /app/

RUN pip install -r requirements.txt

COPY ./app /app
```

W folderze app mamy dostępne kilka podfolderów.

W folderze core mamy dostępne pliki config oraz utils, które pozwalają się poprawnie połączyć z bazami danych oraz implementują wyjątki.

```python
    @validator("BACKEND_CORS_ORIGINS", pre=True)
    def assemble_cors_origins(cls, v: Union[str, List[str]]) -> Union[List[str], str]:
        if isinstance(v, str) and not v.startswith("["):
            return [i.strip() for i in v.split(",")]
        elif isinstance(v, (list, str)):
            return v
        raise ValueError(v)

    POSTGRES_SERVER: str
    POSTGRES_USER: str
    POSTGRES_PASSWORD: str
    POSTGRES_DB: str
    DATABASE_URI: str
    MONGODB_URI: str
    REDIS_HOST: str
    REDIS_PORT: int
```

Informacje dotyczące konfiguracji są dostępne w pliku.env

```
backend > .env
1   PROJECT_NAME=db_performance_tester
2   BACKEND_CORS_ORIGINS=["http://localhost:8000", "https://localhost:8000", "http://localhost", "https://localhost"]
3   DATABASE_URI="sqlite:///./sql_app.db"
4
5   POSTGRES_USER=postgres
6   POSTGRES_PASSWORD=postgres
7   POSTGRES_SERVER=database
8   POSTGRES_DB=app
9   REDIS_HOST=localhost
10  REDIS_PORT=6379
11  REDIS_URI="redis://localhost:password@localhost:6379"
12  MONGODB_URI="mongodb://root:example@localhost:27017"
```

W folderze daos mamy zaimplementowane DAO dla autora, recenzji i gry. DAO to komponent dostarczający jednolity interfejs do komunikacji między aplikacją, a bazą danych. Przykładowy DAO dla autora jest zaprezentowany poniżej:

```python
class AuthorDAO(ABC):

    @abstractmethod
    def get_by_id(self, id: str) -> Author:
        pass

    @abstractmethod
    def save(self, author: AuthorCreate) -> Author:
        pass

    @abstractmethod
    def delete(self, id: str):
        pass

def get_dao(db: str, session: Session = Depends(get_session)):
    if db == "postgresql":
        dao = AuthorDAOSql(session)
        yield dao
    elif db == "mongodb":
        dao = AuthorDAOMongo(mongodb)
        yield dao
    elif db == "redis":
        dao = AuthorDAORedis(get_redis(RedisDbs.AUTHORS))
        yield dao
    else:
        raise NotImplementedError()
```

```python
class AuthorDAOMongo(AuthorDAO):

    def __init__(self, db: Database) -> None:
        super().__init__()
        self.db = db
        self.collection = db.get_collection("authors")

    def get_by_id(self, id: str) -> t.Optional[Author]:
        author_bson = self.collection.find_one({'id': id})
        if author_bson:

            author_mongo = AuthorMongo(**author_bson)

            return Author.from_orm(author_mongo)
        else:
            return None

    def save(self, author: AuthorCreate) -> Author:
        author_mongo = AuthorMongo(**author.dict())
        author_json = author_mongo.dict(by_alias=True)
        if self.get_by_id(author.id):
            raise ValueError("exists")
        self.collection.insert_one(author_json).inserted_id
        ret = self.get_by_id(author.id)
        if ret is None:
            print(ret)
            raise ValueError("couldnt get after add")
        else:
            return ret

    def delete(self, id: str):
        self.collection.delete_one({'id': id})
```

W folderze databases znajdują się pliki, które tworzą sesje dla baz danych. Przykładowa sesja dla Redisa jest okazana poniżej:

```python
class RedisDbs(Enum):
    AUTHORS = 0
    GAMES = 0
    REVIEWS = 0


redisInstanceAuthors = redis.Redis(
    settings.REDIS_HOST,
    settings.REDIS_PORT,
    db=RedisDbs.AUTHORS.value,
    retry_on_error=[exceptions.BusyLoadingError],
    retry=Retry(backoff=ConstantBackoff(3), retries=-1),
)
redisInstanceGames = redisInstanceAuthors
redisInstanceReviews = redisInstanceAuthors
schema = (
    TextField("$.author_id", as_name="author_id"),
    TextField("$.game_id", as_name="game_id"),
    TextField("$.content", as_name="content"),
)
try:
    info = redisInstanceReviews.ft("reviews_idx").info()
except exceptions.ResponseError as e:
    print("adding index")
    redisInstanceReviews.ft("reviews_idx").create_index(
        schema,
        definition=IndexDefinition(prefix=["review:"], index_type=IndexType.JSON),
    )


def get_redis(db: RedisDbs) -> redis.Redis:
    if db == RedisDbs.AUTHORS:
        return redisInstanceAuthors
    elif db == RedisDbs.GAMES:
        return redisInstanceGames
    elif db == RedisDbs.REVIEWS:
        return redisInstanceReviews
    else:
        raise ValueError("no such db in redis")
```

W folderze models mamy zaimplementowane zasady opisujące strukturę danych w bazie danych. Przykładowa struktura dla gier:

```python
class GameDB(Base):
    __tablename__ = "games"
    id = Column(String(25), primary_key=True, index=True)
    name = Column(String(40))
    reviews = relationship("models.review.review.ReviewDB", back_populates="game", lazy="dynamic")

class GameMongo(BaseModel):
    mongo_id: ObjectId = Field(default_factory=PyObjectId, alias="_id")
    id: str
    name: str

    class Config:
        allow_population_by_field_name = True
        arbitrary_types_allowed = True
        json_encoders = {ObjectId: str}

class GameBase(BaseModel):
    name: str
    id: str

class GameRedis(GameBase):
    pass

class Game(GameBase):

    class Config:
        orm_mode = True

class GameCreate(GameBase):
    pass
```

W routers znajdują się funkcje możliwe do użycia na każdej bazie danych oraz znajdują się tam testy wydajnościowe, które pozwalają nam sprawdzić szybkość działania bazy danych dla odpowiednich funkcji oraz wybranej ilości iteracji.

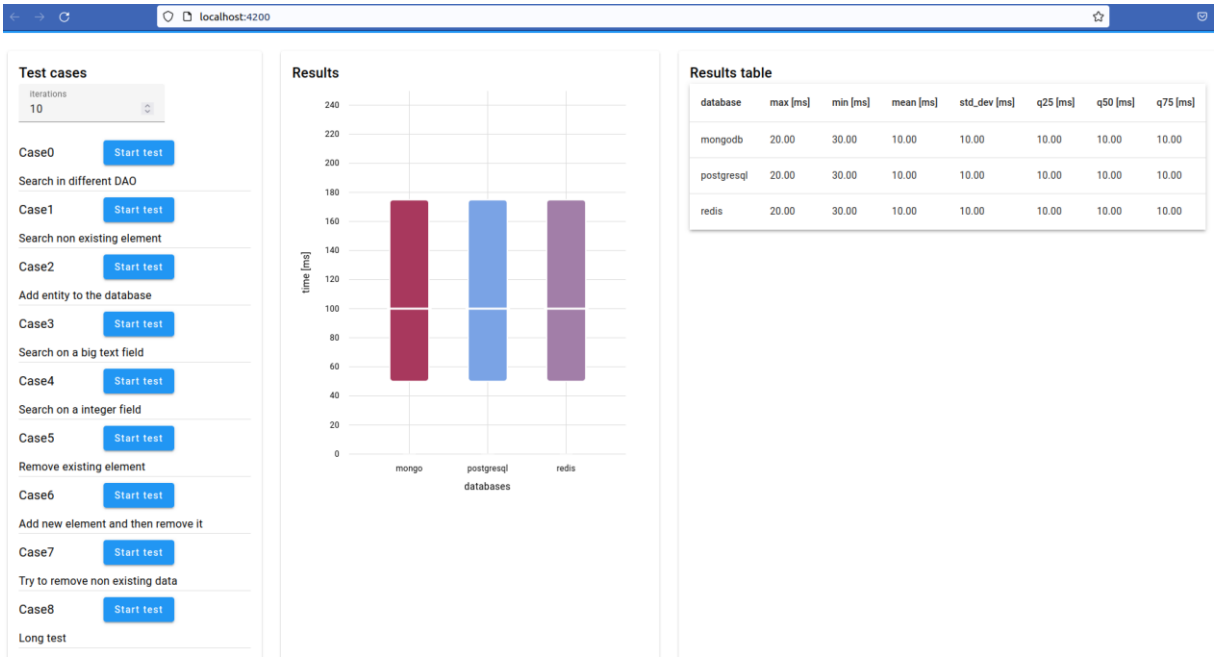
```python
router = APIRouter(prefix="/api/tests")

@dataclass
class CommonSettings:
    reviews_dao: review.ReviewDAO
    games_dao: game.GameDAO
    authors_dao: author.AuthorDAO

def get_common_settings(
    games_dao: game.GameDAO = Depends(game.get_dao),
    reviews_dao: review.ReviewDAO = Depends(review.get_dao),
    authors_dao: author.AuthorDAO = Depends(author.get_dao)
):
    return CommonSettings(
        reviews_dao=reviews_dao,
        games_dao=games_dao,
        authors_dao=authors_dao
    )
```

Jeśli chodzi o folder frontend to mamy tam wygenerowany kod w angularze, który służy do stworzenia aplikacji. Znajdują się tam pliki html, css, które odpowiadają za pojawienie się elementów w aplikacji. Z bardziej interesujących części mamy plik app.component.html gdzie są dostępne parametry wizualne aplikacji.

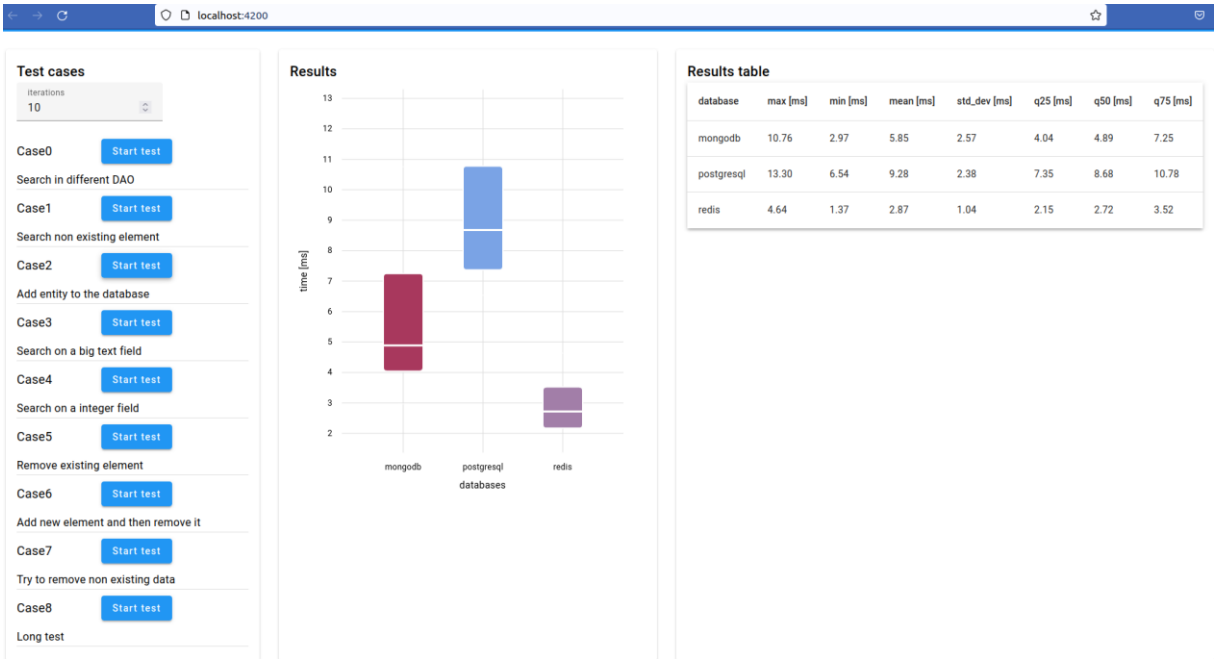
W pliku app.components.ts wyliczamy odpowiednie wartości z przeprowadzonych testów. Takimi wartościami są wartości minimalne, maksymalne, średnia oraz odchylenie standardowe dla każdej bazy osobno. Oprócz tego tam dodajemy listę testów, które możemy włączyć w aplikacji z odpowiednią liczbą iteracji.

Wygląd aplikacji

Po uruchomieniu aplikacji i wejściu na http://localhost:4200/ otrzymujemy widok domyślny:



Po ewentualnym zmienieniu wartości Iterations i kliknięciu Start test ukażą się rezultaty zarówno na wykresie jak i w tabeli.



Testy

Wszystkie testy zostały napisane w języku python. Mają one tę samą strukturę dla każdej bazy danych i wywołują one kolejne bazy danych z tymi samymi parametrami oraz wykonują tą samą czynność na bazie danych. Obliczają one czas na wykonanie danej czynności przez policzeniem różnicy czasu. Następnie otrzymany czas zamieniamy na ms, aby móc wynik przedstawić na wykresie.

Scenariusze testowe:

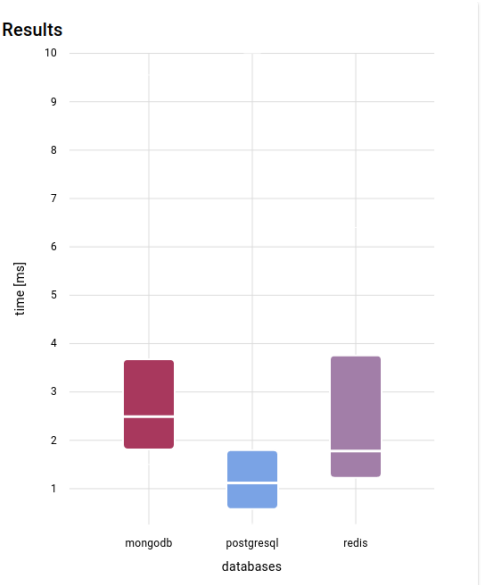
- Case0 – wyszukiwanie elementu przy odwołaniu do innego DAO
- Case1 – wyszukiwanie nieistniejącego elementu w odpowiednim DAO
- Case2 – dodanie nowego elementu do bazy danych
- Case3 – wyszukiwanie istniejącego elementu tekstowego w istniejącej kolumnie
- Case4 – wyszukiwanie istniejącego elementu liczbowego w istniejącej kolumnie
- Case5 – usunięcie istniejącego elementu z bazy danych
- Case6 – dodanie nowego elementu , a następnie usunięcie tego elementu z bazy danych
- Case7 – próba usunięcia nieistniejącego elementu
- Case8 – próba wyszukania nieistniejącego elementu, następnie dodanie nowego elementu do bazy danych, wyszukanie tego elementu, a następnie jego usunięcie

Każdy test jest powtarzany określoną ilość razy, którą ustalamy w aplikacji w polu Iterations. Uzyskujemy w ten sposób bardziej rzeczywiste pomiary, które pozwalają odrzucić rezultaty związane z błędem pomiarowym urządzenia. Wyniki następnie są prezentowane na wykresie gdzie jest podział dla każdej bazy danych, a następnie są obliczane kwantyle. Całkowite wyniki, które zawierają średnią, wartości minimalne i maksymalne, ochylenie standardowe i kwantyle są pokazane w tabeli Results table.

Wyniki

Każdy test został puszczony dla kilku różnych iteracji, aby móc przeanalizować wydajność różnych baz danych i sprawdzić, czy się zmienia wraz ze wzrostem iteracji.

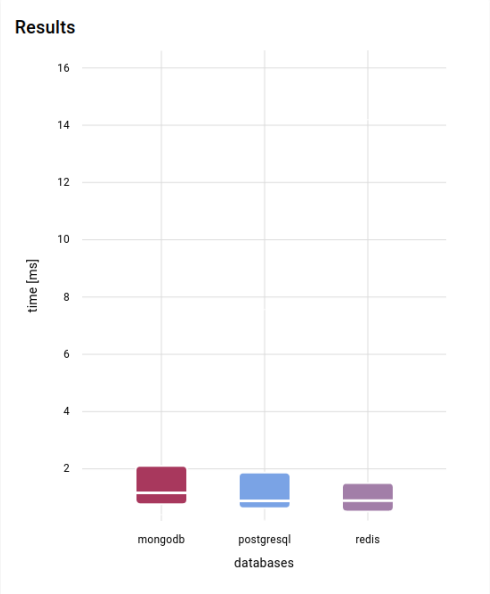
- Case0 – wyszukiwanie elementu przy odwołaniu do innego DAO
  Dla 10:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	9.55	1.50	3.28	2.29	1.80	2.49	3.68
postgresql	10.01	0.28	1.95	2.75	0.57	1.12	1.80
redis	6.40	0.26	2.64	2.04	1.22	1.78	3.76

Dla 100:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	16.61	0.40	2.26	2.95	0.74	1.16	2.11
postgresql	7.56	0.21	1.43	1.40	0.60	0.88	1.87
redis	14.19	0.18	1.64	2.34	0.49	0.89	1.51

Dla 1000:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	30.63	0.31	1.22	1.39	0.59	0.89	1.39
postgresql	14.54	0.16	1.17	1.14	0.56	0.88	1.36
redis	16.43	0.20	1.05	1.25	0.47	0.71	1.17

Dla 10000:

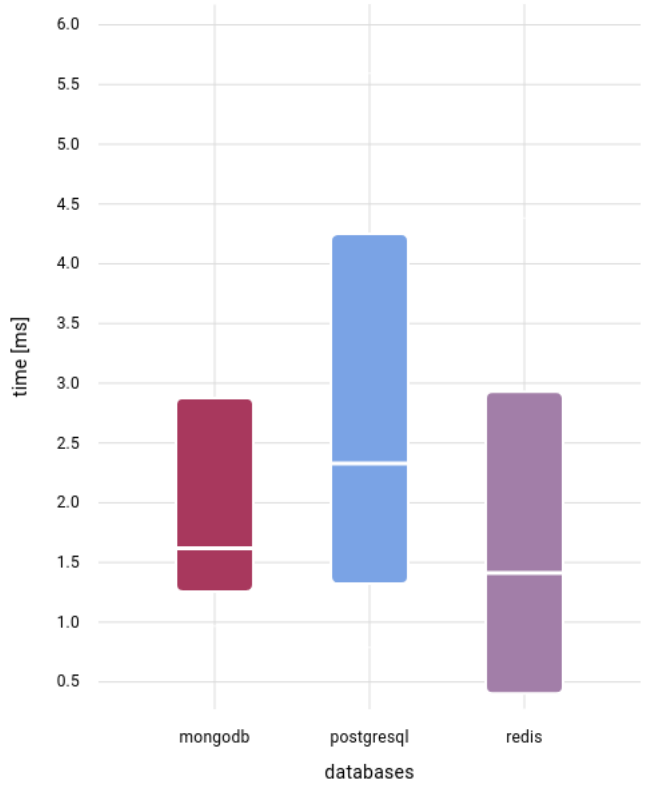
Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	27.58	0.28	1.39	1.45	0.64	1.04	1.63
postgresql	34.95	0.15	1.23	1.30	0.62	0.94	1.44
redis	30.70	0.11	1.15	1.22	0.52	0.86	1.39

Wniosek: W przypadku tego testu dla małych wartości najlepiej wypada relacyjna baza danych (postgresql). Wraz ze wzrostem ilości iteracji przewagę zdobywa Redis, który działa najszybciej. W tym przypadku nie warto stosować mongodb.

- Case1 – wyszukiwanie nieistniejącego elementu w odpowiednim DAO
  Dla 10:

Results

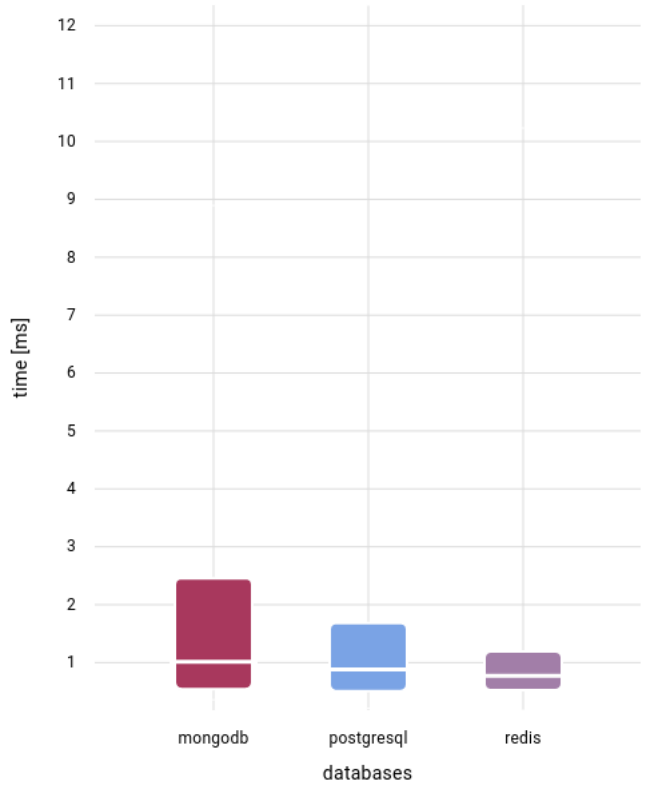


Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	6.17	0.97	2.32	1.62	1.25	1.62	2.88
postgresql	5.59	0.79	2.80	1.67	1.31	2.33	4.25
redis	4.38	0.27	1.80	1.53	0.40	1.41	2.93

Dla 100:

Results



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	8.89	0.36	1.83	1.86	0.52	1.01	2.46
postgresql	12.35	0.21	1.49	1.73	0.50	0.88	1.68
redis	10.23	0.18	1.20	1.38	0.50	0.77	1.19

Dla 1000:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	18.59	0.30	1.27	1.45	0.51	0.94	1.47
postgresql	20.50	0.17	1.14	1.16	0.55	0.89	1.31
redis	18.65	0.12	1.05	1.21	0.51	0.72	1.15

Dla 10000:

Results table

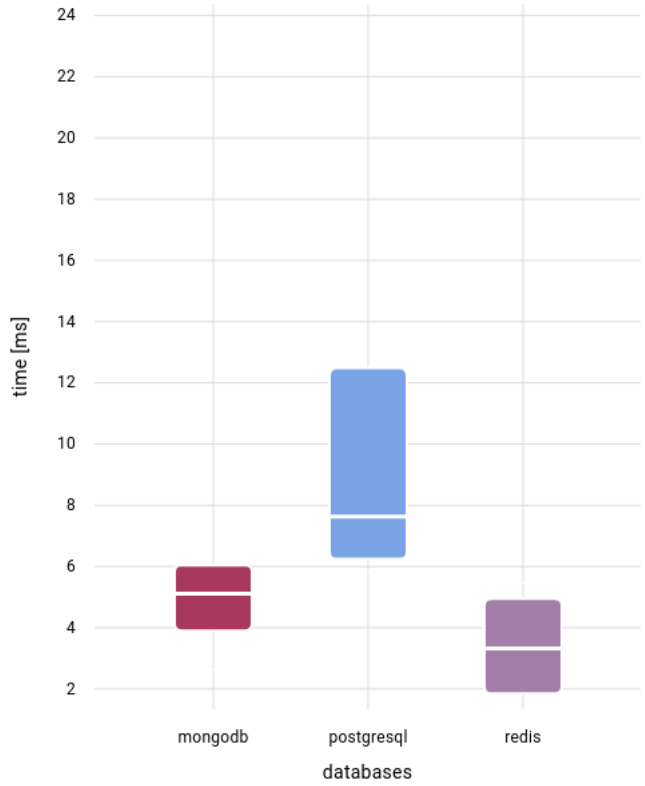
database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	32.07	0.28	1.32	1.39	0.62	1.01	1.55
postgresql	56.42	0.14	1.18	1.32	0.58	0.95	1.40
redis	24.53	0.09	1.07	1.10	0.52	0.81	1.28

W tym przypadku również najszybciej działa Redis niezależnie od ilości danych.

- Case2 – dodanie nowego elementu do bazy danych

Dla 10:

Results

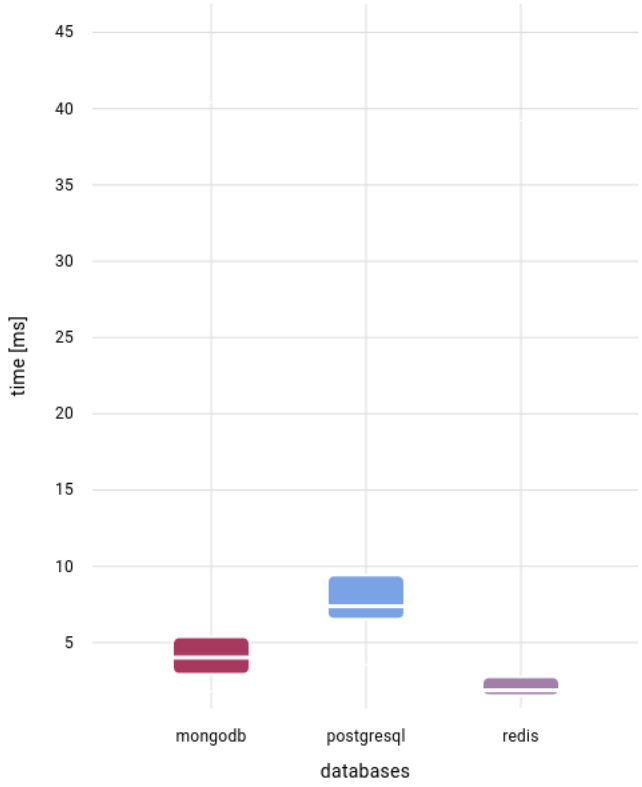


Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	17.75	2.63	6.41	4.36	3.88	5.11	6.04
postgresql	24.37	5.84	10.54	5.91	6.24	7.63	12.48
redis	5.46	1.34	3.34	1.57	1.82	3.32	4.94

Dla 100:

Results



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	40.42	1.81	4.63	4.02	2.86	4.03	5.38
postgresql	46.88	3.57	8.26	4.56	6.48	7.39	9.43
redis	39.24	0.70	2.64	3.86	1.49	1.88	2.76

Dla 1000:

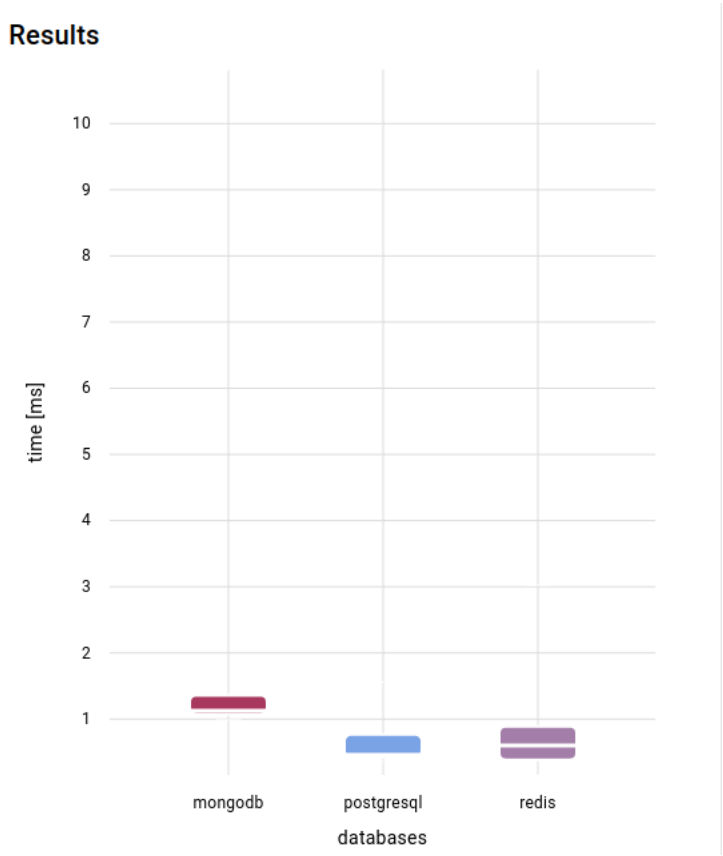
Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	10.33	1.02	2.70	1.35	1.82	2.36	3.15
postgresql	17.79	2.17	4.76	2.52	3.15	3.97	5.37
redis	8.44	0.19	1.31	0.95	0.71	1.06	1.61

Wyniki wskazują, że w tym przypadku bazy nierelacyjne działają o wiele lepiej niż relacyjna. Wśród baz nierelacyjnych lepszą okazuje się być redis.

- Case3 – wyszukiwanie istniejącego elementu tekstowego w istniejącej kolumnie

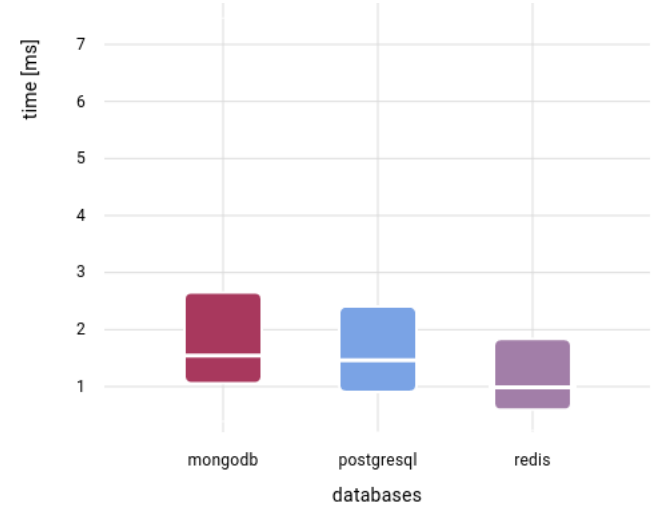
Dla 10:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	10.81	0.99	2.44	2.98	1.06	1.12	1.36
postgresql	1.56	0.36	0.71	0.43	0.41	0.46	0.77
redis	3.02	0.16	0.93	0.89	0.37	0.60	0.89

Dla 100:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	7.46	0.39	2.05	1.49	1.06	1.54	2.65
postgresql	9.20	0.20	1.90	1.63	0.89	1.46	2.41
redis	12.58	0.23	1.59	1.83	0.58	0.99	1.84

Dla 1000:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	17.67	0.27	1.33	1.53	0.53	1.00	1.50
postgresql	15.30	0.15	1.15	1.26	0.39	0.94	1.35
redis	15.74	0.14	0.88	1.05	0.44	0.61	0.92

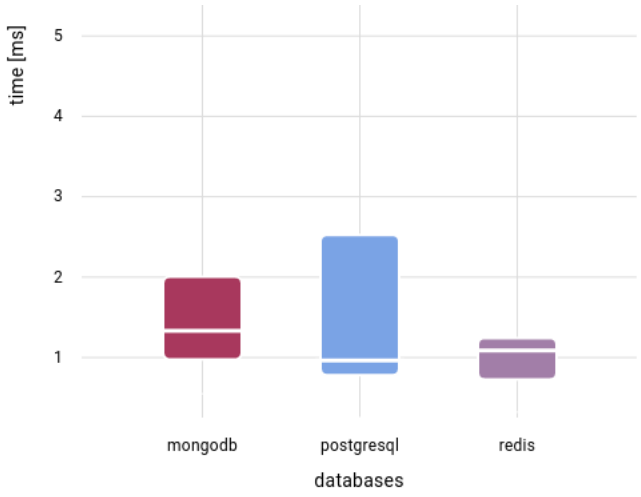
Dla 10000:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	32.54	0.27	1.31	1.35	0.62	1.02	1.52
postgresql	32.31	0.15	1.19	1.27	0.59	0.97	1.44
redis	20.92	0.07	0.96	1.02	0.46	0.70	1.12

Przy większej ilości iteracji przewagę zdobywa redis. Jednak dla małej ilości najszybszy jest postgresql. Mongodb jest zawsze najwolniejsze.

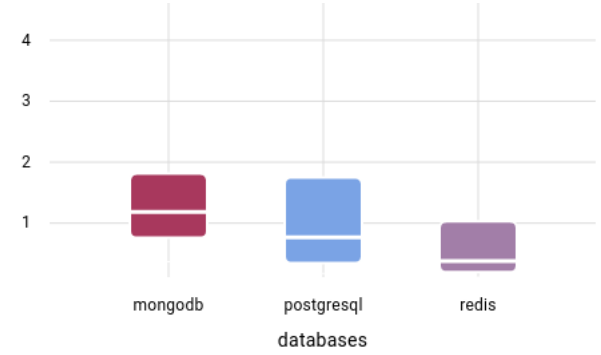
- Case4 – wyszukiwanie istniejącego elementu liczbowego w istniejącej kolumnie
  Dla 10:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	9.02	0.55	2.17	2.36	0.97	1.33	2.00
postgresql	6.32	0.25	1.79	1.75	0.77	0.96	2.52
redis	7.01	0.32	1.62	1.85	0.72	1.09	1.24

Dla 100:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	11.08	0.37	1.63	1.57	0.75	1.18	1.81
postgresql	11.70	0.17	1.45	1.81	0.33	0.77	1.75
redis	7.93	0.11	0.89	1.23	0.18	0.38	1.03

Dla 1000:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	19.23	0.32	1.64	1.61	0.70	1.15	1.91
postgresql	16.41	0.14	1.58	1.79	0.54	0.98	1.98
redis	16.13	0.10	1.28	1.38	0.57	0.88	1.51

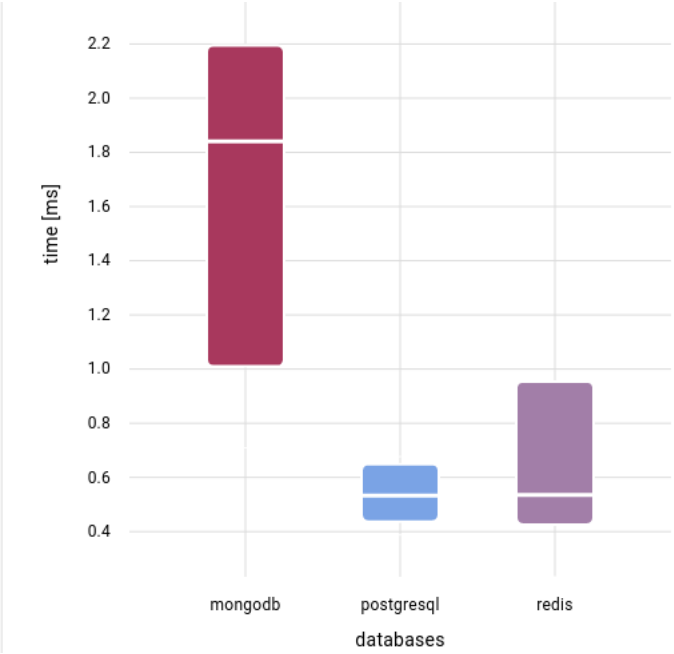
Dla 10000:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	30.70	0.24	1.33	1.42	0.62	0.96	1.54
postgresql	21.32	0.15	1.22	1.18	0.60	0.94	1.45
redis	19.83	0.10	1.06	1.10	0.49	0.77	1.24

Dla każdej ilości iteracji najszybsza była nierelacyjna baza danych redis. Natomiast relacyjna baza postgresql była szybsza od nieralacyjnej bazy mongodb.

- Case5 – usunięcie istniejącego elementu z bazy danych
  Dla 10:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	2.84	0.71	1.69	0.67	1.01	1.84	2.19
postgresql	0.68	0.39	0.54	0.11	0.43	0.53	0.65
redis	2.40	0.23	0.87	0.72	0.42	0.54	0.95

Dla 100:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	5.03	0.40	1.34	0.96	0.80	1.07	1.40
postgresql	5.53	0.29	0.86	0.76	0.44	0.56	1.02
redis	9.28	0.21	1.32	1.25	0.62	0.93	1.50

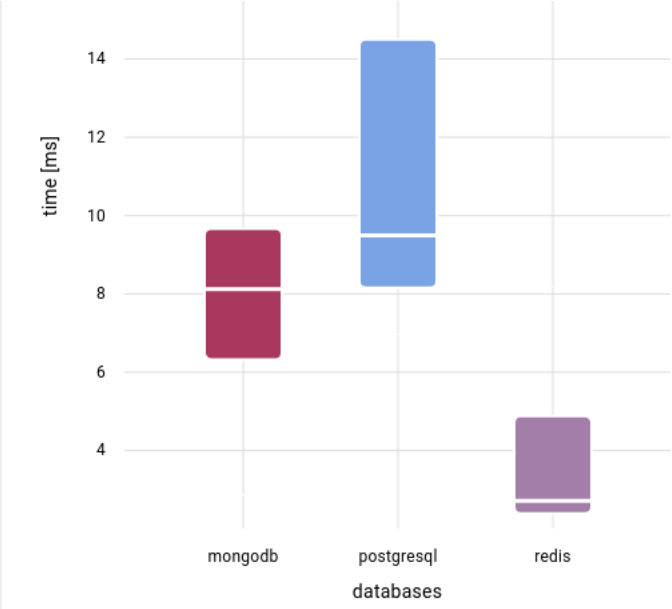
Dla 1000:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	7.74	0.31	0.98	0.75	0.55	0.75	1.13
postgresql	9.19	0.25	0.69	0.67	0.39	0.47	0.72
redis	6.84	0.07	0.77	0.72	0.36	0.57	0.90

W tym przypadku najlepiej sprawdza się baza relacyjna dla małych wartości. Dla większych lepsze jest użycie bazy nierelacyjnej Redis.

- Case6 – dodanie nowego elementu , a następnie usunięcie tego elementu z bazy danych
  Dla 10:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	15.81	2.86	8.12	3.50	6.29	8.12	9.67
postgresql	20.02	6.96	11.42	4.19	8.14	9.49	14.50
redis	8.18	2.01	3.73	1.93	2.38	2.71	4.88

Dla 100:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	41.72	2.11	5.61	4.24	3.86	4.74	6.36
postgresql	51.18	3.57	8.82	6.31	4.98	6.34	11.24
redis	42.31	1.06	3.31	4.30	1.90	2.29	3.15

Dla 1000:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	18.66	1.56	3.99	1.80	2.76	3.49	4.71
postgresql	31.42	2.85	6.82	3.49	4.37	5.62	8.30
redis	12.03	0.38	2.22	1.35	1.30	1.85	2.74

W przypadku tego testu widać, że bazy nieralacyjne działają szybciej niż relacyjne. Wśród baz nierelacyjnych o wiele szybszą jest redis.

- Case7 – próba usunięcia nieistniejącego elementu
  W tym teście pominęliśmy wartości dla postgresql ze względu na implementację usuwania.
  Dla 10:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	3.63	0.42	1.33	1.03	0.62	0.87	1.83
redis	2.81	0.21	0.69	0.73	0.34	0.50	0.66

Dla 100:

Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	9.87	0.44	1.31	1.60	0.66	0.81	1.16
redis	12.61	0.13	1.07	1.90	0.35	0.49	0.72

Dla 1000:

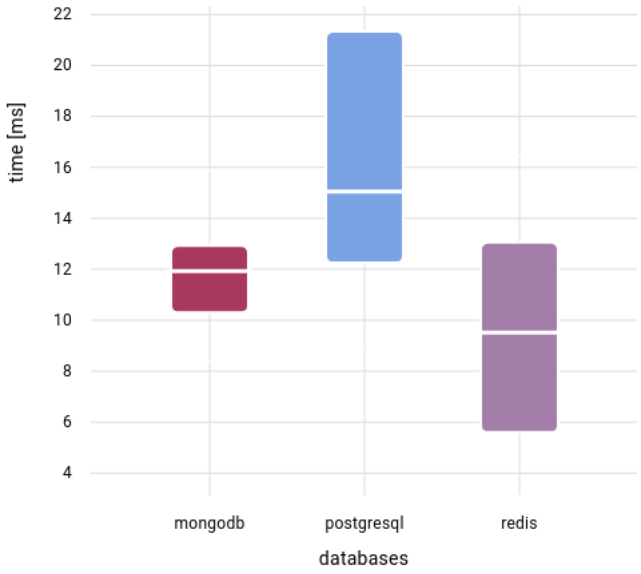
Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	24.81	0.26	0.99	1.98	0.47	0.58	0.75
redis	29.40	0.09	0.60	1.96	0.13	0.21	0.35

Dla każdej wartości szybszą jest redis.

- Case8 – próba wyszukania nieistniejącego elementu, następnie dodanie nowego elementu do bazy danych, wyszukanie tego elementu, a następnie jego usunięcie

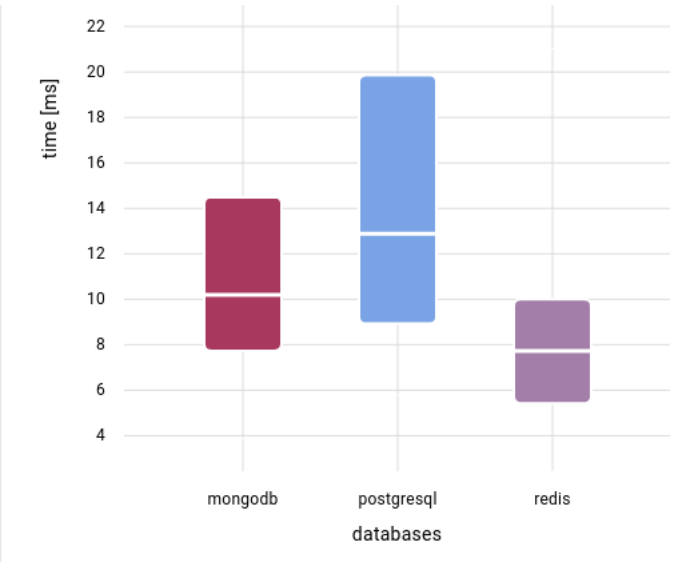
  Dla 10:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	15.89	8.39	11.81	2.08	10.26	11.92	12.93
postgresql	30.84	8.93	17.40	6.68	12.23	15.05	21.34
redis	16.43	3.12	9.50	4.48	5.58	9.52	13.05

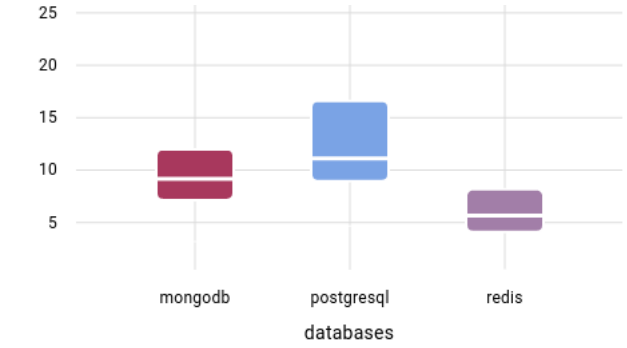
  Dla 100:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	24.76	3.86	11.15	4.57	7.70	10.20	14.49
postgresql	33.47	5.79	14.78	7.04	8.89	12.89	19.86
redis	21.04	2.44	8.08	3.27	5.39	7.73	10.00

Dla 1000:



Results table

database	max [ms]	min [ms]	mean [ms]	std_dev [ms]	q25 [ms]	q50 [ms]	q75 [ms]
mongodb	46.82	3.18	10.06	4.20	7.12	9.16	11.99
postgresql	67.84	4.68	13.82	7.54	8.90	11.13	16.61
redis	30.87	0.50	6.52	3.60	3.99	5.68	8.19

W przypadku użycia wszystkich zaimplementowanych funkcji widać przewagę nierelacyjnych baz danych nad relacyjnymi. Jednak i w tym przypadku jest duża różnica między wartościami dla baz NoSQL, gdyż Redis lepiej się sprawdza niż mongodb.

Podsumowanie

Po zaimplementowaniu kodu oraz przeprowadzeniu testów można stwierdzić, że każda baza działa w inny sposób. Bazy relacyjne są w stanie lepiej działać dla małych ilości powtórzeń jeśli chcemy coś wyszukać w bazie danych. Jednak wraz ze wzrostem ilości coraz bardziej widać przewagę baz NoSQL.

Jeśli chodzi o dodawanie nowych elementów to za każdym razem lepiej się prezentowały bazy nierelacyjne. Jest to związane ze sposobem przechowywania danych w tych bazach oraz brakiem sztywnego modelu danych. Bazy relacyjne podczas dodawania danych dokonują szeregu operacji sprawdzenia poprawności wprowadzanych rekordów. W wyniku tego operacja dodawania danych jest bardziej kosztowna obliczeniowo, w zamian otrzymujemy jednak pewność, że przechowywane dane są spójne. W przypadku usuwania, jeśli wiemy, że dany element istnieje w bazie to lepiej sprawdzi się relacyjna baza danych. Jeśli jednak nie ma takiej pewności bądź jest większa ilość iteracji to warto zastosować rozwiązanie NoSQL.

Na podstawie przeprowadzonych testów można zauważyć, że same bazy nierelacyjne działają w inny sposób. Jest to związane z ich sposobem przechowywania danych. O wiele szybciej działała baza Redis niż mongodb, która w każdym teście osiągała lepsze wyniki wydajnościowe. Jednak jest ona dużo bardziej ograniczona pod względem możliwości operacji i agregacji wykonywanych na bazie. Jest to pewien kompromis wydajności i możliwości funkcjonalnych bazy danych. W zależności od tego co jest naszym priorytetem powinniśmy wybrać odpowiednie rozwiązanie. Jeśli jest to wydajność i szybkość to z pewnością Redis będzie lepszą opcją, ale będziemy musieli liczyć się z faktem, że większość operacji, które mogłaby wykonać za nas baza danych, będziemy musieli zaimplementować samodzielnie w naszej aplikacji. Jeśli natomiast zależy nam na wszechstronności, uniwersalności, możliwych typach danych oraz możliwych funkcjonalnościach to baza MongoDB będzie lepszym wyborem. Prawdopodobnie, dla większej ilości danych, znacząco przekraczającej wielkość dostępnej pamięci RAM, lepszym rozwiązaniem również będzie MongoDB, ponieważ baza Redis projektowana była z założeniem przechowywania prawie wszystkich danych w pamięci operacyjnej serwera.