

AGH

AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Maciej Kubicki

kierunek studiów: **informatyka stosowana**

JSON w bazie danych PostgreSQL

JSON in PostgreSQL database

Opiekun: **dr inż. Grażyna Krupińska**

Kraków, Styczeń 2017

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

.....
(czytelny podpis)

Merytoryczna ocena pracy przez opiekuna:

Celem pracy inżynierskiej Pana Macieja Kubickiego było po pierwsze przetestowanie mechanizmów związanych z przechowywaniem dokumentów w formacie JSON w relacyjnej bazie danych PostgreSQL. Drugim celem było porównanie wydajności mechanizmów obsługi formatu JSON w bazie PostgreSQL z mechanizmami dostępnymi w przykładowych nierelacyjnych bazach danych (Couchbase , MongoDB).

Po zaprezentowaniu w rozdziale pierwszym celów pracy, kolejny rozdział opisuje składnię formatów zapisu dokumentów JSON oraz XML . Dalej znajdujemy prezentację idei relacyjnych baz danych na przykładzie bazy PostgreSQL, oraz nierelacyjnych baz danych (baza typu kluczwartość, bazy grafowe, bazy dokumentowe, bazy kolumnowe). Rozdział piąty poświęcony został operatorom, indeksom, funkcjom przetwarzającym oraz funkcjom konwertującym działającym na danych typu JSON w bazie danych PostgreSQL. Rozdział szósty przedstawia mechanizmy przetwarzania danych typu JSON w bazach dokumentowych. Ostatni rozdział to opis oraz podsumowanie przeprowadzonych na prezentowych bazach testów wydajnościowych.

W trakcie pracy autor wykazał się dużą samodzielnością oraz umiejętnością wykorzystania różnych narzędzi bazodanowych. Drobne błędy w cytowaniach, oraz dosyć pobieżna analiza wyników przeprowadzonych testów nie zmieniają pozytywnej oceny pracy.

Końcowa ocena pracy przez recenzenta: 4.0

Data: 18.01.2017

Podpis:.....

Skala ocen: 5.0 – bardzo dobra, 4.5 – plus dobra, 4.0 – dobra, 3.5 – plus dostateczna, 3.0 – dostateczna, 2.0 – niedostateczna

Merytoryczna ocena pracy przez recenzenta:

Przedstawiona do recenzji praca inżynierska dotyczy możliwości składowania danych w formacie JSON w relacyjnej bazie danych PostgreSQL. Autor opisał w pracy format JSON porównując go z XML-em oraz wskazaną w tytule funkcjonalność, która została zaimplementowana od wersji 9.2 na tle innych baz relacyjnych oraz współcześnie bardzo dynamicznie rozwijających się baz nierelacyjnych. Na uwagę zasługuje próba porównania wydajności przetwarzania danych JSON w bazie PostgreSQL oraz dwóch systemach baz NoSQL (Couchbase i MongoDB). Autor na wstępie przygotował odpowiednio duże zbiory danych, a następnie przeprowadził na nich serię testów odpowiadających typowym sposobom ich przetwarzania (wyszukiwanie, grupowanie, aktualizacja i usuwanie danych). Testy prowadzone były w seriach po 10 prób, tak aby uzyskane wyniki były bardziej reprezentatywne, jednak autor nie pomyślał, aby porównując uzyskane wartości średnie, przypisać im niepewności wynikające ze statystycznego rozrzutu uzyskanych wyników. Stąd przy niewielkich różnicach czasów dla wybranych testów trudno ocenić, czy rzeczywiście są one statystycznie istotne. Ponadto cytując literaturę autor pominął w tekście odwołania do pozycji 4, 6 i 7. Nieprawidłowa jest też kolejność cytowania, a praca zawiera szereg drobnych błędów. Mimo tych uchybień uważam pracę za wartościową, a uzyskane wyniki za ciekawe.

Końcowa ocena pracy przez recenzenta: 4.0

Data: 18.01.2017

Podpis:.....

Skala ocen: 5.0 – bardzo dobra, 4.5 – plus dobra, 4.0 – dobra, 3.5 – plus dostateczna, 3.0 – dostateczna, 2.0 – niedostateczna

Spis treści

1	Wstęp	5
1.1	Wykorzystane oprogramowanie	5
2	Format JSON	6
2.1	Składnia i typy danych JSON-a	6
2.2	Przykładowy JSON	7
2.3	Porównanie z XML-em	8
3	Relacyjne i nierelacyjne bazy danych	9
3.1	Relacyjne bazy danych	9
3.2	Nierelacyjne bazy danych	11
3.2.1	Baza klucz-wartość	11
3.2.2	Baza grafowa	11
3.2.3	Baza dokumentowa	12
3.2.4	Baza kolumnowa	12
4	PostgreSQL	13
4.1	Funkcjonalność	13
4.2	Typy json i jsonb	13
4.3	Dane testowe	14
4.4	Operatory JSON	16
4.5	Funkcje przetwarzające dokument JSON	20
4.6	Funkcje tworzące dokument JSON	23
4.7	Tworzenie indeksów w dokumentach JSON	25
5	Bazy dokumentowe	27
5.1	Couchbase Server	27
5.1.1	N1QL	27
5.2	MongoDB	29
5.2.1	Przeszukiwanie kolekcji	29
5.3	Porównanie baz dokumentowych z PostgreSQL-em	32
5.3.1	Wprowadzenie danych	32
5.3.2	Zapytania nr 1	32
5.3.3	Zapytania nr 2	33
5.3.4	Zapytania nr 3	34
5.3.5	Zapytania nr 4	34
5.3.6	Zapytania nr 5	35
5.3.7	Zapytanie nr 6	36
5.3.8	Rozmiar na dysku	36
5.3.9	Podsumowanie wydajności	37
5.3.10	Spojrzenie z innej strony	37
6	Bibliografia	38

1 Wstęp

Od początku istnienia baz danych to zapotrzebowanie rynku, czy to naukowego, czy też komercyjnego oraz postęp technologii informatycznych kształtowały ich rozwój. Pojęcie bazy danych stało się popularne we wczesnych latach sześćdziesiątych, kiedy to poszukiwano wydajnych metod dostępu do składowanych informacji. Kolejne lata to zaprezentowanie i stopniowy rozwój relacyjnego modelu, który jest stosowany do dziś. Jeszcze w końcowych latach wieku XX pojawiły się takie pojęcia jak rozproszone i obiektowe bazy danych. Początek XXI wieku to gigantyczny wzrost liczby przechowywanych informacji w związku z którym wystąpiła potrzeba szukania nowych modeli przechowywania danych oraz ich przetwarzania. Pojawiły się rozwiązania NoSQL-owe. Pojęcie to pochodzi od zwrotu *non SQL* (ang. *non relational* - nie relacyjne), jednak obecnie wiąże się je ze powiedzeniem angielskim - *not only SQL* (nie tylko relacyjne). Proces rozwoju baz danych nie jest ukierunkowany tylko w jednym kierunku. Współbieżnie następuje postęp w aspekcie takich pojęć jak hurtownie danych (ang. *Data Warehouse*, obecnie utożsamiane z nazwą *Buisness Intelligence*), czy też Big Data.

Jedną podgałęzią wspomnianego wcześniej NoSQL-a jest model dokumentowy, który początkowo był oparty o dokumenty typu XML, jednak typ ten obecnie jest wypierany przez wydajniejsze oraz dużo bardziej wpisujące się w konwencje NoSQL, a mianowicie przez typ JSON. Celem pracy jest przedstawienie możliwości tego formatu zaimplementowanego w relacyjnej bazie danych jaką jest PostgreSQL oraz porównanie tychże narzędzi z innymi rozwiązaniami NoSQL-owymi.

Praca składa się z kilku części. W pierwszej przedstawiłem format JSON, wraz próbą porównania go z jego prekursorem w dziedzinie dokumentowych baz danych XML-em. W drugiej zestawilem ze sobą model relacyjny oraz nierelacyjny baz danych. W trzeciej części prezentuję PostgreSQL w aspekcie możliwości wykorzystania JSON-a. W ostatniej części przedstawiam nierelacyjne bazy danych - Couchbase i MongoDB, a następnie zestawiam je z PostgreSQL w celu zbadania wydajności.

1.1 Wykorzystane oprogramowanie

W pracy inżynierskiej używałem PostgreSQL 9.6.1, MongoDB 3.2.1, Couchbase Server Enterprise 4.5.1 oraz Python 2.7 wraz z odpowiednimi bibliotekami.

2 Format JSON

Pełne rozwinięcie JSON to JavaScript Object Notation. Geneza tego formatu jest związana z pojawieniem się zapotrzebowania na lekki - wydajny, przyjazny dla użytkownika standard asynchronicznej wymiany danych między serwerem, a użytkownikiem z pominięciem dodatkowych rozszerzeń przeglądarki. Początkowo JSON był elementem JavaScript-u, jednak obecnie jest notacją niezależną. Format ten jest popularnie stosowany w implementacji AJAX-a (ang. *asynchronous JavaScript and XML*), obecnie można spotkać określenie AJAX (ang. *asynchronous JavaScript and JSON*). Technologie te odnoszą się do dynamicznego odświeżania fragmentów aplikacji webowych, bez konieczności przeładowywania całej aplikacji. Wstępnie jako formę wymiany danych stosowana XML-a, jednak swoista natywność JSON-a dla JavaScriptu spowodowała to, że wypiera on XML w tym aspekcie. Kolejnymi zaletami nad XML-em są mniejszy rozmiar oraz brak ścisłych reguł odnoszących się do formatowania. Inną grupą technologii, dla które coraz powszechniejszy staje się JSON są web serwisy (ang. *web services*). Domyślnie dokumenty JSON-a zakodowane są przy pomocy UTF-8. Rozszerzenie plików zawierających tę notację to ".json". Format JSON jest wspierany przez wiele języków programowania np. JAVA, Python i C#. [9]

2.1 Składnia i typy danych JSON-a

Typy danych:

- typ string - sekwencja znaków rozpoczęta i zakończona ". Pusty string jest poprawny,
- typ numeryczny,
- typ logiczny,
- null,
- tablica - może zawierać dane różnych typów, a także być pusta,
- obiekt - rozpoczyna się od { i kończy na }, zawiera pary klucz:wartość, gdzie klucz to string, a wartość może być każdym z typów.

2.2 Przykładowy JSON

```
1 {
2     "objectID": 1232214,
3     "book": {
4         "title": "Pan Tadeusz",
5         "pages": 1200,
6         "date": null,
7         "publisher": "PolishPub",
8         "author": {
9             "firstName": "Adam",
10            "lastName": "Mickiewicz"
11        }
12    },
13    "isAvailable": true,
14    "whoHadIt": [{
15        "clientID": 1,
16        "clientName": "John",
17        "clientLastName": "Knox",
18        "phoneNumber": "608123123"
19    }, {
20        "clientID": 12,
21        "clientName": "Ann",
22        "clientLastName": "Snow",
23        "phoneNumber": "612123123"
24    }, {
25        "clientID": 1112,
26        "clientName": "Gary",
27        "clientLastName": "Cena",
28        "phoneNumber": "512123123"
29    }],
30    "whoHasIt": null,
31    "timeOfTakig": "2016-12-19 22:03:49",
32    "timeOfReturning": "2016-12-25 12:03:49"
33    "location": "PolRom3F";
34 }
35 }
```

Listing 1: Dokument JSON

Przykład używa wszystkich powyżej zaprezentowanych typów danych. Obiekt przedstawia dokument, jaki mógłby być użyty w aplikacji obsługującej bibliotekę. Każdy obiekt ma swoje unikalne id, klucz book, zawiera "podklucze" opisujące dany egzemplarz. Klucz isAvailable to typ logiczny mówiący czy egzemplarz jest dostępny. Tablica whoHadIt zawiera obiekty trzech ostatnich osób, które wypożyczyły egzemplarz, w zależności od indeksów tablicy ustalamy kolejność. Klucz whoHasIt przechowuje osobę, która posiada aktualnie egzemplarz, jeśli jest null to egzemplarz jest w bibliotece. Obiekt też ma daty ostatniego wypożyczenia i oddania, a także sygnaturę, na której półce powinien znajdować się egzemplarz.

2.3 Porównanie z XML-em

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2   <object ID="1232214" loc="PolRom3F" isAvailabe="true">
3     <book pages="1200">
4       <title>Pan Tadeusz</title>
5       <date />
6       <publisher>PolishPub</publisher>
7       <author>
8         <firstName>Adam</firstName>
9         <lastName>Mickiewicz</lastName>
10      </author>
11    </book>
12    <whoHadIt queue="1">
13      <clientID>1</clientID>
14      <clientName>John</clientName>
15      <clientLastName>Knox</clientLastName>
16      <phoneNumber>608123123</phoneNumber>
17    </whoHadIt>
18    <whoHadIt queue="2">
19      <clientID>12</clientID>
20      <clientName>Ann</clientName>
21      <clientLastName>Snow</clientLastName>
22      <phoneNumber>612123123</phoneNumber>
23    </whoHadIt>
24    <whoHadIt queue="3">
25      <clientID>1112</clientID>
26      <clientName>Gary</clientName>
27      <clientLastName>Cena</clientLastName>
28      <phoneNumber>512123123</phoneNumber>
29    </whoHadIt>
30    <whoHasIt />
31    <timeOfTakig>2016-12-19 22:03:49</timeOfTakig>
32    <timeOfReturning>2016-12-25 12:03:49</timeOfReturning>
33  </object>
```

Listing 2: Dokument XML

Powyższy XML jest jednym z możliwych odpowiedników przykładowego JSON-a. Czytelność obydwu dokumentów jest zbliżona, jeśli dokument JSON-owy jest odpowiednio sformatowany. Warto zaznaczyć, że format zgodny ze specyfikacją jest warunkiem parsowania XML-a. Z tego wynika, że XML jest zawsze przejrzysty. Jeśli chodzi o JSON-a, to tam nie ma czegoś takiego jak sprawdzanie sposobu formatowania, jednak każda biblioteka obsługująca ten typ dokumentów posiada funkcje formatujące. Warto zauważyć, że dokument JSON ma rozmiar około 30% mniejszy, niż XML. Obydwa formaty mają wsparcie wielu języków programowania, a także wielu dodatkowych narzędzi o podobnym zastosowaniu np. XML Schema i JSON Schema do walidacji struktury, XPath i JsonPath do przeszukiwania. Także wybór między tymi formatami jest trudny i nie da się jednoznacznie stwierdzić, który z nich jest lepszy. W różnych przypadkach lepiej może się sprawdzić albo jeden, albo drugi format.

3 Relacyjne i nierelacyjne bazy danych

3.1 Relacyjne bazy danych

Lata siedemdziesiąte XX wieku to początek rozwoju relacyjnego modelu jaki znamy dziś. Za pioniera tego podejścia w stosunku do danych jest Edgar F. Codd. Kolejną kluczową postacią w aspekcie tego rodzaju baz jest dr. Peter Chen, utożsamiany z modelem zależności encji - E-R (ang. *Entity-Relationship Model*).

W tym modelu relacyjnym najważniejsza jest relacja, którą można utożsamiać z tabelą. Relacja to zbiór dziedzin, z których każda składa się z nagłówka oraz treści. Odnosząc to stwierdzenie do pojęcia tabeli, możemy sprowadzić pojęcie relacji do zbioru nazw kolumn (nagłówków) oraz rekordów (treści). Każdy wiersz tabeli, czyli tak zwana krotka wyróżnia się od pozostałych za sprawą klucza głównego, składającego się z jednego, bądź większej ilości nagłówków. Istotną kwestią jest powiązanie między tabelami, które wynika ze wspomnianego wcześniej modelu E-R. Oto możliwe połączenia:

- 1 do 1 - wiersz w relacji R_1 może posiadać tylko jeden odpowiadający mu wiersz w R_2 i na odwrót, na przykład mamy tabelę Użytkownicy i ZestawUstawień. Każdy użytkownik ma zdefiniowane swoje ustawienia i każde zdefiniowane ustawienia odnoszą się do jednego użytkownika,
- 1 do N - wiersz w relacji R_1 może posiadać jeden lub wiele powiązanych wierszy z relacji R_2 , na przykład tabela Użytkownicy i Zdjęcia. Każdy użytkownik może posiadać wiele zdjęć.
- N do M - jeden lub wiele wierszy z relacji R_1 może być powiązanych z jednym lub wieloma wierszami relacji R_2 i na odwrót. Za przykład posłużyć mogą tabele Ludzie i Adresy.

Model relacyjny jest wsparty przez algebrę relacyjną. Bezpośrednio opartym na tejże algebrze jest język SQL, który służy do manipulacji i zarządzania danymi w relacyjnych bazach danych. Algebra relacyjna określa osiem podstawowych operatorów działających na relacjach. W niektórych rozwiązaniach można spotkać dodatkowe operatory, które wynikają z tych podstawowych:

- selekcja,
- projekcja,
- iloczyn,
- iloraz,
- przecięcie,
- złączenie,
- suma,
- iloraz.

Aspektem, który implikuje to, że relacyjne bazy danych są nierozzerwalnie związane z takim systemami jak banki, czy też sklepy jest transakcja, tak zwana ACID. Pojęcie to można rozumieć jako zbiór poleceń, które mają być wykonane na bazie, zwracający wyniki dla klienta. Zbiór ma być przetworzony kompletnie lub wcale. ACID oznacza cechy transakcji:

- atomowość (ang. *atomicity*) - to znaczy, że albo zostanie przetworzona w całości, albo w ogóle,
- spójność (ang. *consistency*) - baza danych przed i po transakcji pracuje poprawnie,
- izolacja (ang. *isolation*) - transakcje są od siebie odseparowane, ich instrukcje się nie przeplatają,
- trwałość (ang. *durability*) - zatwierdzona transakcja i wynikające z niej modyfikacje w bazie są trwałe.

Oprogramowanie, które służy do zarządzania relacyjnymi bazami danych określa się jako SZRBD - system zarządzania relacyjną bazą danych (ang. *RDBMS - relational database management system*). Oto kilka takich systemów:

- Microsoft SQL Server,
- IBM DB2,
- MySQL,
- PostgreSQL,
- Oracle Database.

3.2 Nierelacyjne bazy danych

Termin nierelacyjnych baz danych określa bardzo szeroki zakres rozwiązań technologicznych, z których każde odbiega od modelu relacyjnego w mniejszym lub większym stopniu. Pojęcie to stało się rozpoznawalne na początku XXI. wieku, jednak niektóre z rozwiązań opisanych tym słowem rozwijało się wcześniej. Postęp zagadnień związanych z NoSQL-em przede wszystkim wynikał z niedoskonałości modelu relacyjnego w pewnych kwestiach. Przykładem takiego aspektu jest sytuacja, gdy aplikacja ma współpracować z dynamicznie rozrastającą się bazą, której dane są nieustrukturyzowane, bądź ich forma się zmienia w czasie. Innym niezmiernie ważnym aspektem jest skalowalność horyzontalna, która w przypadku nierelacyjnego podejścia jest atutem. Sposób strukturyzacji w relacyjnych bazach danych zwykle ograniczał do skalowalności wertykalnej.

Patrząc trochę z innego punktu widzenia w ramach zestawienia ze sobą relacyjnego i nierelacyjnego podejścia warto przytoczyć teorie CAP, wedle której jest niemożliwe pełne spełnienie trzech podstawowych założeń dla rozproszonego systemu bazodanowego, tzn. spójności (ang. *Consistency*), dostępności (ang. *Availability*) i tolerancji na podział (ang. *Partition tolerance*). Dla baz relacyjnych spójność jest cechą najistotniejszą, natomiast dla nierelacyjnych skalowalność, która wiąże się z partycjonowaniem. Podział w relacyjnych bazach danych jest czymś nietrywialnym na przykład w związku z istnieniem operatora złączenia.[5][8]

Wyróżniamy kilka typów baz nierelacyjnych:

- bazy klucz-wartość (ang. *key-value stores*),
- bazy grafowe (ang. *graph stores*),
- bazy dokumentowe (ang. *document oriented database*),
- bazy kolumnowe (ang. *wide-column store*).

3.2.1 Baza klucz-wartość

Jest to bardzo prosty rodzaj bazy, która może przechowywać jedynie klucz i odpowiadającą mu wartość, która może być listą krotek innej bazy, bądź listą obiektów lub prostą wartością. Rozwiązanie to przypomina słownik ze skryptowego języka Python. Istnieje wiele implementacji tego modelu. Niektóre wersje wprowadzają sortowanie kluczy, także widzimy różnice w przechowywaniu danych (RAM, dyski). Przykładem wykorzystanie takiego rodzaju bazy mogą być system obsługujący jakieś zdarzenia - klucz jako identyfikator zdarzenia, a wartość jako zbiór wszystkie informacje na temat zdarzenia. Przykładowe bazy klucz-wartość: Redis, BerkleyDB.

3.2.2 Baza grafowa

Grafy w nauce znajdują wiele zastosowań, okazały się również atrakcyjną strukturą danych. W bazach grafowych dane oraz ich powiązania są reprezentowane w postaci węzłów, krawędzi oraz własności. Węzeł, który można postrzegać jako obiekt jest powiązany z innym obiektem poprzez krawędź. Zarówno węzeł, jak i krawędź może mieć własności, czyli jakieś cechy je opisujące. Struktura grafu daje możliwość o wiele wydajniejszego wyszukiwania danych w wypadku wielokrotnego zagłębienia zapytania, niż w wypadku modelu relacyjnego. W celu wyekstrahowania zagnieżdżonych danych w wypadku relacyjnym konieczne jest użycie złączeń i kluczy obcych, natomiast w podejściu grafowym powiązane dane są połączone krawędziami z odpowiednią własnością i to wpływa na wydajność zapytań. Grafy w informatyce są dość powszechne i dzięki temu istnieje wiele algorytmów

ułatwiających pracę z tym rodzajem bazy danych. Popularnym przykładem wykorzystania tych baz są sieci społecznościowe. Aplikacją działającą z tym rodzajem bazy jest Twitter. Przykładowe bazy grafowe: Neo4j, OrientDB, FlockDB.

3.2.3 Baza dokumentowa

Ten rodzaj baz można by uznać jako podgrupę, bądź rozwinięcie baz klucz-wartość, gdyż dane są tu przechowywane w jakimś pojemniku (np. Bucket, Collection) w formie dokumentów (wartość), z których każdy ma przypisany identyfikator (klucz). Arcyważny dla tych baz jest format przechowywania dokumentów. Istnieje tu kilka możliwości w zależności od implementacji bazy: JSON, XML, BSON, YAML. Przechowywany dokument zwykle nie ma narzuconej struktury. Przede wszystkim ten rodzaj baz jest popularny w aplikacjach webowych, w szczególności korzystających z architektury RESTful. Przykładowe bazy dokumentowe: MongoDB, Couchbase, CouchDB, NoSQL, DocumentDB.

3.2.4 Baza kolumnowa

Bazy kolumnowe wraz z dodawaniem kolejnych rekordów nie rozszerzają się w "dół", lecz w "szerz". Wynika to z faktu, że dane z kolumny są dodawane obok siebie. Taka implementacja pozwala na szybkie wykonywanie poleceń selekcji, agregacji, czy też filtracji. Z drugiej strony inne rodzaje poleceń mogą być wolniejsze od analogicznych w modelu relacyjnym. Ten rodzaj baz korzysta z normalnego odpowiednika SQL. Fakt szybkiego wykonywania poleceń, powoduje, że bazy kolumnowe powszechnie są wykorzystywane w analizie danych. Przykładowe bazy kolumnowe: Cassandra, HBase.

4 PostgreSQL

PostgreSQL jest dynamicznie rozwijającym się systemem zarządzania relacyjną bazą danych (SZRBD), który jest darmowy. Jak sama nazwa wskazuje system jest zgodny ze standardem SQL. Postgres ma również pewne znamiona obiektowej bazy danych. Ważną cechą tego oprogramowania jest dostępność na wiele systemów operacyjnych oraz architektur. Kilkukrotnie w ciągu roku wydawane są nowe wersje rozszerzające funkcjonalność, bądź też udoskonalające już zaimplementowane rozwiązania.

4.1 Funkcjonalność

Dane w Postgresie są przechowywane w zdefiniowanych przez użytkownika tabelach. SZBZ umożliwia tworzenie triggerów, widoków, indeksów, własnych typów danych, czy też własnych funkcji. Do ich tworzenia użytkownik ma w dyspozycji wiele języków programowania.

Ogranicznik	Wartość
Maksymalny rozmiar bazy danych	brak ograniczenia
Maksymalna ilość wierszy tabeli	brak ograniczenia
Maksymalny rozmiar tabeli	32 [TB]
Maksymalny rozmiar wiersza	1.6 [TB]
Maksymalny rozmiar pola	1 [GB]

Tabela 1: Tabela ograniczeń dla PostgreSQL [1]

Do łatwego obsługiwanego serwera istnieje kilka aplikacji, między innymi: PgAdmin, psql, phpPgAdmin.

4.2 Typy json i jsonb

Między kilkudziesięcioma innymi typami od wersji 9.2 pojawił się typ json, a od wersji 9.4 jsonb. Obydwie struktury służą do zapisu dokumentów w formacie JSON. Podstawową różnicą między nimi jest sposób przechowywania. Mianowicie pierwszy z nich jest zapisany jako dokładna kopia parametru wejściowego. Natomiast drugi z typów jest przechowywany w formie binarnej (BSON). BSON jest binarnym odpowiednikiem JSON-a, z minimalnymi rozwinięciami, na przykład BSON wspiera o kilka więcej typów danych - typ data, czy też wyrażenie regularne. Zarówno typ json i jsonb narzuca sprawdzenie zgodności danych z formatem JSON. Zmienne typowe dla JSON-a użyte w dokumencie są konwertowane do odpowiadającym im typów PostgreSQL.[1]

Typ JSON	Typ PostgreSQL
numeryczny	numeric
string	text
logiczny	boolean
null	(none)

Tabela 2: Schemat konwersji typów

Ze sposobu przechowywania tych typów wynika to, że typ json jest szybszy do zapisu, niż jsonb, jednak jest wolniejszy w przetwarzaniu, gdyż jest parsowany przy każdym użyciu. Dodatkowym atutem wersji binarnej jest możliwość tworzenia indeksów na kluczach wewnątrz dokumentu. [1]

4.3 Dane testowe

Do przedstawienia możliwości użycia formatu JSON w PostgreSQL przygotowałem dwa zestawy danych. Pierwszy zestaw został wygenerowany na podstawie AdventureWorks2008 z Microsoft SQL Server. Oto kod, który wygenerował zbiór 18798 dokumentów JSON.

```
1 from DatabasePreparation import *
2 import pyodbc
3 cnxn = pyodbc.connect('DRIVER={SQL Server};SERVER=localhost\
    SQLEXPRESS;DATABASE=AdventureWorks2008;')
4 cursor = cnxn.cursor()
5 cursor.execute("SELECT e.EmailAddress, pp.FirstName, ISNULL(
    pp.MiddleName, '') AS MiddleName, pp.LastName, p.
    BusinessEntityID, [AddressLine1], ISNULL([AddressLine2], '')
    AS AddressLine2, [City], [PostalCode] FROM [
    AdventureWorks2008].[Person].[Address] a, [
    AdventureWorks2008].[Person].[EmailAddress] e, [
    AdventureWorks2008].[Person].[BusinessEntityAddress] p, [
    AdventureWorks2008].[Person].[Person] pp WHERE p.AddressID
    = a.AddressID and pp.BusinessEntityID=p.BusinessEntityID
    and e.BusinessEntityID=pp.BusinessEntityID;")
6 rows = cursor.fetchall()
7
8 with open('person.json', 'w') as pl:
9     for row in rows:
10         if is_json(to_json(row)):
11             pl.write(to_json(row, False) + '\n')
```

Listing 3: Generowanie dokumentów JSON

Funkcja `to_json` na podstawie wyniku wiersza generuje dokument JSON. Struktura wygenerowanych dokumentów nie jest jednakowa. Mianowicie nie każdy człowiek (odpowiadający mu dokument) posiada klucz `MiddleName` oraz `address_line2`. Dodatkowo liczba adresów e-mail przypisanych do osoby może wynosić 0,1,2.

Oto przykładowy dokument:

```
1 {
2     "person": {
3         "personID": 12,
4         "FirstName": "Thierry",
5         "MiddleName": "B",
6         "LastName": "DHers"
7     },
8     "adres": {
9         "city": "Bothell",
10        "adres_line1": "1970 Napa Ct.",
11        "postal_code": "98011"
12    },
13    "email": ["thierry0@adventure-works.com", "DHers12
14              @mail.com"]
15 }
```

Listing 4: Wygenerowany dokument JSON

Tworzę tablice dla typu json i jsonb, a następnie wypełniam je:

```
1 CREATE TABLE PersonJSONB (
2     "ObjectID" SERIAL NOT NULL ,
3     data JSONB NOT NULL,
4     PRIMARY KEY("ObjectID"));
5 CREATE TABLE PersonJSON(
6     "ObjectID" serial NOT NULL,
7     data JSON NOT NULL,
8     PRIMARY KEY ("ObjectID"));
```

Listing 5: Tworzenie tabeli w PostgreSQL z typami jsonb i json

Drugi zestaw obejmuje prostą tabelkę:

```
1 CREATE TYPE Gender AS ENUM( 'M' , 'F' );
2 CREATE TABLE Users (
3     idUsers SERIAL NOT NULL ,
4     username VARCHAR(20) NOT NULL ,
5     password VARCHAR(255) NOT NULL ,
6     mail VARCHAR(255) NOT NULL ,
7     Name VARCHAR NOT NULL ,
8     Surname VARCHAR NOT NULL ,
9     town VARCHAR(20) NOT NULL ,
10    gender Gender NOT NULL ,
11    country VARCHAR(255) NOT NULL,
12    PRIMARY KEY(idUsers));
```

Listing 6: Tworzenie tabeli z drugim zestawem danych

4.4 Operatory JSON

PostgreSQL udostępnia kilkanaście operatorów dla typu jsonb, tylko część z nich działa na json. Pierwsza grupa operatorów działa na obu typach i ich działanie skoncentrowane jest na przeszukiwaniu dokumentu JSON.

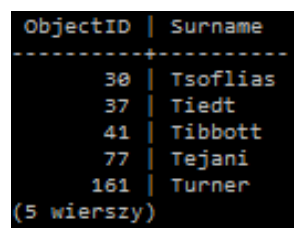
- json / jsonb -> 'klucz' / indeks
- json / jsonb ->> 'klucz' / indeks
- json / jsonb #> 'ścieżka'
- json / jsonb #>> 'ścieżka'

Operatory z podwójnym > zwracają wynik jako text, natomiast z pojedynczym > w tym typie co lewa strona operatora (jsonb/json).

Przykład 1.

```
1 SELECT "ObjectID", data->'person'->>'LastName' AS Surname FROM
   personjsonb WHERE data->'person'->>'LastName' LIKE 'T%' LIMIT 5;
```

Listing 7: Przykład 1.



ObjectID	Surname
30	Tsoflias
37	Tiedt
41	Tibbott
77	Tejani
161	Turner

(5 wierszy)

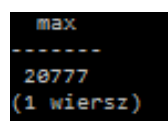
Rysunek 1: Wynik zapytania z przykładu 1.

Na przykładzie widzimy użycie i operatora -> i ->>, w miejscu gdzie zwracamy Surname moglibyśmy użyć ->, jednak zwrócone nazwiska były by w cudzysłowach. Natomiast przy operatorze LIKE musimy użyć ->>, by zwrócić text, ponieważ operator LIKE działa na wartości tekstowej.

Przykład 2.

```
1 SELECT max(cast(data->'person'->>'personID' AS int)) FROM personjsonb;
```

Listing 8: Przykład 2.



max
20777

(1 wiersz)

Rysunek 2: Wynik zapytania z przykładu 2.

Na przykładzie widzimy użycie i operatora -> i ->>, dodatkowo dodaję rzutowanie i funkcję agregującą max. Funkcjonalność typowa dla SQL współgra z typami do obsługi JSON.

Przykład 3.

```
1 SELECT data #> '{person, LastName}' AS NAME, COUNT(*) FROM public .  
   personjsonb GROUP BY NAME;
```

Listing 9: Przykład 3.

name	count
"Roy"	2
"Rivera"	93
"Northup"	1
"Petculescu"	1
"Heidepriem"	1
"Morgan"	93
"Morris"	90
"Su"	1

Rysunek 3: Fragment wyniku zapytania z przykładu 3.

W tym przykładzie widzimy przeszukiwanie przy pomocy #> i dodane grupowanie po nazwisku.

Przykład 4.

```
1 SELECT data->'email'->0 AS FirstEmail FROM public.personjsonb WHERE  
   jsonb_array_length(data->'email')=1 LIMIT 3;
```

Listing 10: Przykład 4.

firstemail
"mikaelo@adventure-works.com"
"belinda@adventure-works.com"
"calvin2@adventure-works.com"
(3 wiersze)

Rysunek 4: Wynik zapytania z przykładu 4.

Tutaj jest jeszcze przeszukiwanie dokumentu z odwołaniem się do indeksu tablicy.

Przykład 5.

```
1 SELECT DISTINCT data->'address'->'city' AS City FROM personjsonb ORDER BY  
   City ASC;
```

Listing 11: Przykład 5.

city
"Ballard"
"Baltimore"
"Barstow"
"Basingstoke Hants"

Rysunek 5: Fragment wyniku zapytania z przykładu 5.

Tu mamy wyszukanie unikalnych nazw miast z użyciem sortowania.

Druga grupa operatorów działa już tylko na jsonb. Ich działanie polega na sprawdzeniu zawierania się, łączeniu obiektów, a także uszczuplaniu obiektów. Wewnątrz obiektu JSON poprawnym typem jest inny obiekt JSON, a więc możliwe jest przypisanie go do klucza lub dodanie go do tablicy - zagnieżdżanie dokumentów. Na przykładzie dokumentów JSON znajdujących się w pierwszym zestawie testowym stwierdzenie zerowy poziom zagłębienia odnosi się do kluczy "person", "address", ("email").

- jsonb @> (<@) jsonb - sprawdzenie czy lewy (prawy) jsonb zawiera się w prawym (lewym) w zerowym poziomie zagłębienia dokumentu,
- jsonb ? 'klucz' - czy istnieje klucz jsonb w zerowym poziomie zagłębienia dokumentu,
- jsonb ?|(?&) 'klucze[]' - czy któryś z kluczy (wszystkie klucze) jest (są) zawarty (zawarte) w jsonb w zerowym poziomie zagłębienia dokumentu,
- jsonb - indeks • jsonb - 'klucz' • jsonb #- 'ścieżka' - uszczuplenie obiektu o element o danym indeksie, kluczu, lub o zadanej ścieżce

Przykład 1.

```

1 SELECT count(data) FROM public.personjsonb WHERE data ? 'email';
2 SELECT count(data) FROM public.personjsonb WHERE data ?| array['person', '
  address', 'email'];
3 SELECT count(data) FROM public.personjsonb WHERE data ?& array['person', '
  address', 'email'];

```

Listing 12: Przykład 1.

Trzy zapytania pokazują przykładowe wykorzystanie operatorów zawierania. W pierwszym wypadku zliczam wszystkie dokumenty gdzie jest klucz 'email' w zerowym poziomie zagłębienia. W kolejnym zliczam te, które zawierają co najmniej jeden z kluczy z tablicy w zerowym poziomie zagłębienia, a w ostatnim te, które posiadają wszystkie klucze z tablicy na tym poziomie. Dane stworzyłem w taki sposób, że wszystkie dokumenty mają 'person' i 'address', a więc pierwsze i ostatnie zapytanie powinno dać taki sam wynik, a środkowe liczbę wszystkich dokumentów. Wyniki powyższych zapytań to: 12516, 18798, 12516.

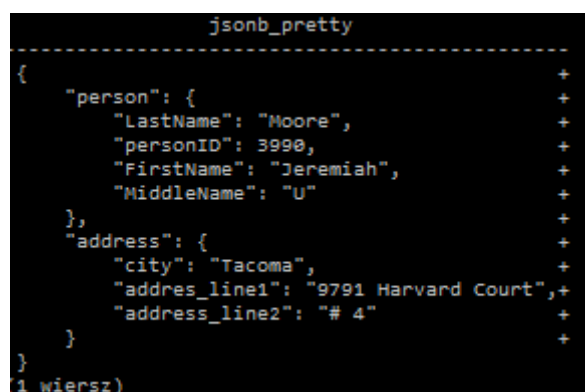
Przykład 2.

```

1 SELECT jsonb_pretty(data #- '{"address","postal_code"}') FROM public.
  personjsonb WHERE "ObjectID"=1921;

```

Listing 13: Przykład 2.



```

      jsonb_pretty
-----
{
  "person": {
    "lastName": "Moore",
    "personID": 3990,
    "firstName": "Jeremiah",
    "middleName": "U"
  },
  "address": {
    "city": "Tacoma",
    "address_line1": "9791 Harvard Court",
    "address_line2": "# 4"
  }
}
1 wiersz)

```

Rysunek 6: Wynik zapytania z przykładu 2.

Powyższy przykład pokazuje użycie operatora #-, aby pozbyć się niechcianego pola w dokumencie.

Przykład 3.

```
1 BEGIN TRANSACTION;
2 SELECT jsonb_pretty(data) FROM public.personjsonb WHERE "ObjectID"=1922;
3 UPDATE public.personjsonb SET data = (SELECT data #- '{"email"}' || '{"
  email": ["newmail@hotmail.com"]}'::jsonb FROM public.personjsonb
  WHERE "ObjectID"=1922) WHERE "ObjectID"=1922;
4 SELECT jsonb_pretty(data) FROM public.personjsonb WHERE "ObjectID"=1922;
5 ROLLBACK;
```

Listing 14: Przykład 3.



```
----- jsonb_pretty -----
{
  "email": [
    "jeremiah2@adventure-works.com",
    "Taylor3991@mail.com"
  ],
  "person": {
    "LastName": "Taylor",
    "personID": 3991,
    "FirstName": "Jeremiah",
    "MiddleName": "M"
  },
  "address": {
    "city": "Royal Oak",
    "postal_code": "V8X",
    "address_line1": "5906 Walnut Place"
  }
}
```

Rysunek 7: Przed modyfikacją



```
----- jsonb_pretty -----
{
  "email": [
    "newmail@hotmail.com"
  ],
  "person": {
    "LastName": "Taylor",
    "personID": 3991,
    "FirstName": "Jeremiah",
    "MiddleName": "M"
  },
  "address": {
    "city": "Royal Oak",
    "postal_code": "V8X",
    "address_line1": "5906 Walnut Place"
  }
}
```

Rysunek 8: Po modyfikacji

Ten przykład przedstawia przeprowadzenie modyfikacji dokumentu JSON. Kasuję wszystkie adresy mailowe z wybranego dokumentu i dodaję nowy. Używam tu operatorów #- do skasowania i || do dodania nowego adresu (w praktyce do połączenia dwóch obiektów jsonb).

4.5 Funkcje przetwarzające dokument JSON

Obecnie mamy dostępne kilkadziesiąt funkcji działających na typie json i jsonb. Pierwsza grupa to tak zwane funkcje przetwarzające. Jest ich kilkanaście dla jsonb i o kilka mniej dla json. Funkcje dla typu binarnego zaczynają się od jsonb_, a dla drugiego typu od json_. Na przykład: jsonb_array_length i json_array_length. Wszystkie funkcje dla dokumentów przechowywanych jako json mają odpowiedniki swoich funkcji dla jsonb. Do zadań funkcji przetwarzających należy wyekstrahowanie pewnych informacji, zmiana sposobu prezentacji dokumentu, a także zmiany w strukturze tego dokumentu - dodawanie elementów, czy też zmiana aktualnych. Oto prezentacja kilku przykładów:

Przykład 1.

```
1 SELECT jsonb_pretty(data) FROM public.personjsonb WHERE
   jsonb_array_length(data->'email') = 2 LIMIT 1;
```

Listing 15: Przykład 1.



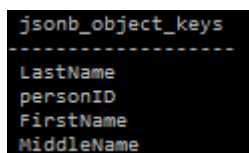
Rysunek 9: Wynik zapytania z przykładu 1.

Powyżej widzimy użycie dwóch funkcji, które już wcześniej się pojawiły - jsonb_pretty (dostępna tylko dla jsonb) i jsonb_array_length. Pierwsza służy do formatowania dokumentu, aby były czytelne, a druga zwraca długość tablicy.

Przykład 2.

```
1 SELECT jsonb_object_keys(data->'person') FROM personjsonb WHERE "ObjectID"
   = 1921;
```

Listing 16: Przykład 2.



Rysunek 10: Wynik zapytania z przykładu 2.

Tu użyta funkcja jsonb_object_keys służy do wyekstrahowania kluczy z najbardziej zewnętrznego poziomu (zerowego poziomu zagnieżdżenia) dokumentu przekazanego.

Przykład 3.

```
1 CREATE TYPE temp_person AS ( "LastName" varchar(25), "personID" int, "  
    FirstName" varchar(25), "MiddleName" varchar(25));  
2 SELECT (jsonb_populate_record(null::temp_person, data->'person')).* from  
    public.personjsonb LIMIT 5;
```

Listing 17: Przykład 3.

LastName	personID	FirstName	MiddleName
DHers	12	Thierry	B
Kuppa	123	Vamsi	N
Abbas	285	Syed	E
Sandberg	251	Mikael	Q
Ralls	124	Kim	T

Rysunek 11: Wynik zapytania z przykładu 3.

Ten przykład pokazuje jak można przenieść dane z dokumentów do struktury tablicowej. W PostgreSQL możemy znaleźć kilka sposobów do takiego działania.

Przykład 4.

```
1 SELECT jsonb_array_elements(data->'email') from personjsonb;
```

Listing 18: Przykład 4.

```
jsonb_array_elements  
-----  
"thierry@adventure-works.com"  
"DHers12@mail.com"  
"syed@adventure-works.com"  
"Abbas285@mail.com"  
"mikael@adventure-works.com"  
"kim@adventure-works.com"  
"Ralls124@mail.com"  
"belinda@adventure-works.com"  
"calvin2@adventure-works.com"  
"karl3@adventure-works.com"
```

Rysunek 12: Fragment wyniku zapytania z przykładu 4.

Powyższy przykład pokazuje jak wyekstrahować wszystkie adresy e-mail ze wszystkich dokumentów. Warto zaznaczyć, że nie we wszystkich dokumentach znajdują się tablica "email", oraz długość tej tablicy w każdym dokumencie może wynieść 0, 1, 2. To oczywiście jest fragment wyników.

Przykład 5.

```
1 SELECT jsonb_typeof(data) AS Document, jsonb_typeof(data->'email') AS  
    Email_Array, jsonb_typeof(data->'person'->'personID') AS PersonID,  
    jsonb_typeof(data->'email'->0) AS Email, jsonb_typeof('true') AS Bool,  
    jsonb_typeof('null') AS Null FROM personjsonb LIMIT 1;
```

Listing 19: Przykład 5.

document	email_array	personid	email	bool	null
object	array	number	string	boolean	null

Rysunek 13: Wynik zapytania z przykładu 5.

PostgreSQL daje nam możliwość sprawdzenia typów obiektów z dokumentu - `jsonb_typeof`. Powyżej widzimy wszystkie możliwe typy danych w formacie JSON.

Przykład 6.

```
1 BEGIN TRANSACTION;
2 SELECT jsonb_pretty(data) FROM public.personjsonb WHERE "ObjectID"=1922;
3 UPDATE public.personjsonb SET data = (SELECT jsonb_set(data, '{email,1}',
    "newmail@hotmail.com",false) FROM public.personjsonb WHERE "ObjectID"
    =1922) WHERE "ObjectID"=1922;
4 SELECT jsonb_pretty(data) FROM public.personjsonb WHERE "ObjectID"=1922;
5 ROLLBACK;
```

Listing 20: Przykład 6.



```
jsonb_pretty
-----
{
  "email": [
    "jeremiah2@adventure-works.com",
    "Taylor3991@gmail.com"
  ],
  "person": {
    "LastName": "Taylor",
    "personID": 3991,
    "FirstName": "Jeremiah",
    "MiddleName": "M"
  },
  "address": {
    "city": "Royal Oak",
    "postal_code": "V8X",
    "address_line1": "5906 Walnut Place"
  }
}
(1 wiersz)
```

Rysunek 14: Przed modyfikacją



```
jsonb_pretty
-----
{
  "email": [
    "jeremiah2@adventure-works.com",
    "newmail@hotmail.com"
  ],
  "person": {
    "LastName": "Taylor",
    "personID": 3991,
    "FirstName": "Jeremiah",
    "MiddleName": "M"
  },
  "address": {
    "city": "Royal Oak",
    "postal_code": "V8X",
    "address_line1": "5906 Walnut Place"
  }
}
(1 wiersz)
```

Rysunek 15: Po modyfikacji

Powyżej widzimy chyba najważniejszą z funkcji przetwarzających - `jsonb_set`. Równie ważna jest funkcja `jsonb_insert`. Obie są dostępne tylko dla typu `jsonb`. Pierwsza z nich:

```
jsonb_set(dokument, 'ścieżka', 'nowy element', dodać_nowy = true)
```

Jej działanie polega na zmianie elementu znalezionego w przekazanej ścieżce na 'nowy element'. Jeśli dokument nie zawiera takiej ścieżki i `dodać_nowy` jest `true` to dodana zostanie nowa para "klucz":"wartość".

```
jsonb_insert(dokument, 'ścieżka', 'nowy element', wstaw_za = false)
```

Ta metoda ma za zadanie dodawania wartości do tablicy. W wypadku wcześniejszej funkcji istnieje możliwość jedynie dodania elementu na koniec tablicy, tu możemy wstawić go wszędzie w tablicy. O miejscu czy wstawiamy go przed, czy po wskazanej ścieżce decyduje parametr `wstaw_za`.

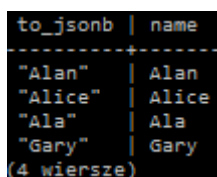
4.6 Funkcje tworzące dokument JSON

Oprócz funkcji przetwarzających PostgreSQL dysponuje drugą grupą funkcji służącą do konwersji różnego rodzaju danych do typów jsonb i json. Funkcjonalność ta jest przydatna wtedy, gdy posiadamy bazę relacyjną, a w aplikacji np. webowej wymieniamy dane przy pomocy technologii AJAX (AJAJ). Postgres daje nam do dyspozycji kilka różnych algorytmów przemiany danych w format dokumentu JSON. Oto kilka przykładów:

Przykład 1.

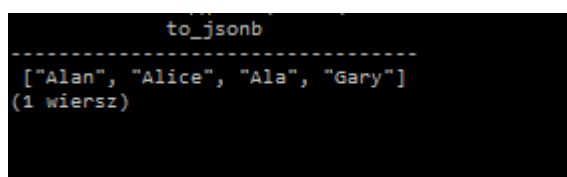
```
1 SELECT to_jsonb(Name),Name FROM Users LIMIT 4;  
2 SELECT to_jsonb(ARRAY(SELECT Name From Users LIMIT 4));
```

Listing 21: Przykład 1.



to_jsonb	name
"Alan"	Alan
"Alice"	Alice
"Ala"	Ala
"Gary"	Gary

Rysunek 16: Zapytanie nr. 1



to_jsonb
["Alan", "Alice", "Ala", "Gary"]

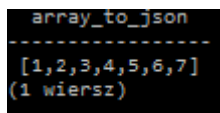
Rysunek 17: Zapytanie nr. 2

Funkcje to_jsonb() oraz odpowiadająca jej to_json() konwertują przekazany parametr do odpowiadającego mu typu w formacie JSON. Powyżej widzimy konwersje stringu i tablicy.

Przykład 2.

```
1 SELECT array_to_json(ARRAY(SELECT idUsers From Users));
```

Listing 22: Przykład 2.



array_to_json
[1,2,3,4,5,6,7]

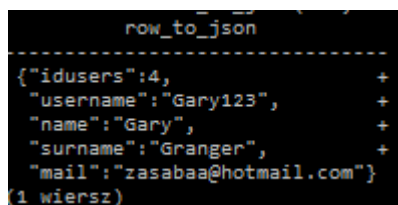
Rysunek 18: Wynik zapytania z przykładu 2.

Funkcja w tym przykładzie jest przeznaczona tylko dla tablic. Ten sam efekt można by otrzymać przy pomocy to_json().

Przykład 3.

```
1 SELECT row_to_json(row,true) FROM (SELECT idUsers ,username , Name, Surname  
    , mail FROM Users WHERE idUsers=4) row;
```

Listing 23: Przykład 3.



row_to_json
{"idusers":4, "username":"Gary123", "name":"Gary", "surname":"Granger", "mail":"zasabaa@hotmail.com"}

Rysunek 19: Wynik zapytania z przykładu 3.

Powyżej widzimy już trochę bardziej skomplikowane przekształcenie. Konwersja całej krotki do obiektu JSON może być całkiem użyteczna. Funkcja jako klucze do wartości z wiersza tablicy przypisuje nazwy atrybutów. Wartość logiczna przekazywana do funkcji `row_to_json` decyduje o sposobie formatowania, tzw. `pretty_bool`.

Przykład 4.

```
1 SELECT json_build_array (ARRAY(SELECT Name From Users) , ARRAY(SELECT
   idUsers From Users));
```

Listing 24: Przykład 4.

```
----- json_build_array -----
[["Alan","Alice","Ala","Gary","Oliver","Jessica","Monica"], [1,2,3,4,5,6,7]]
(1 wiersz)
```

Rysunek 20: Wynik zapytania z przykładu 4.

Funkcja `json_build_array` służy do budowania trochę bardziej wyrafinowanych tablic.

Przykład 5.

```
1 SELECT jsonb_pretty (jsonb_build_object ( 'FirstName' ,Name, 'LastName' ,
   Surname, 'City' ,Town, 'ID' , idUsers)) FROM Users WHERE idUsers=5;
```

Listing 25: Przykład 5.

```
----- jsonb_pretty -----
{
  "ID": 5,
  "City": "Madrid",
  "LastName": "Walker",
  "FirstName": "Oliver"
}
(1 wiersz)
```

Rysunek 21: Wynik zapytania z przykładu 5.

Funkcja `jsonb_build_object` wydaje się najbardziej pożądana, służy do tworzenia obiektów typu `jsonb` (jest również jej odpowiednik dla typu `json`), w których sami możemy nadawać nazwy kluczy.

4.7 Tworzenie indeksów w dokumentach JSON

Wspomnianym już wcześniej atutem typu jsonb jest możliwość utworzenia indeksów na kluczach dokumentu JSON. PostgreSQL daje nam do dyspozycji kilka metod indeksowania między innymi: B-tree, Hash, GIN. W przypadkach, gdy użytkownik nie podaje metody użyta zostaje B-tree (tak jak w poniższym przykładzie). Wszystkie metody działają na kluczach dokumentu JSON.

Przykład

```
1 SELECT "ObjectID", data->'person'->>'LastName'::text AS Surname FROM
   personjsonb WHERE data->'person'->>'LastName'::text LIKE 'T%';
2 CREATE INDEX surname on personjsonb (((data->'person'->>'LastName')::text
   ));
3 SELECT "ObjectID", data->'person'->>'LastName'::text AS Surname FROM
   personjsonb WHERE data->'person'->>'LastName'::text LIKE 'T%';
```

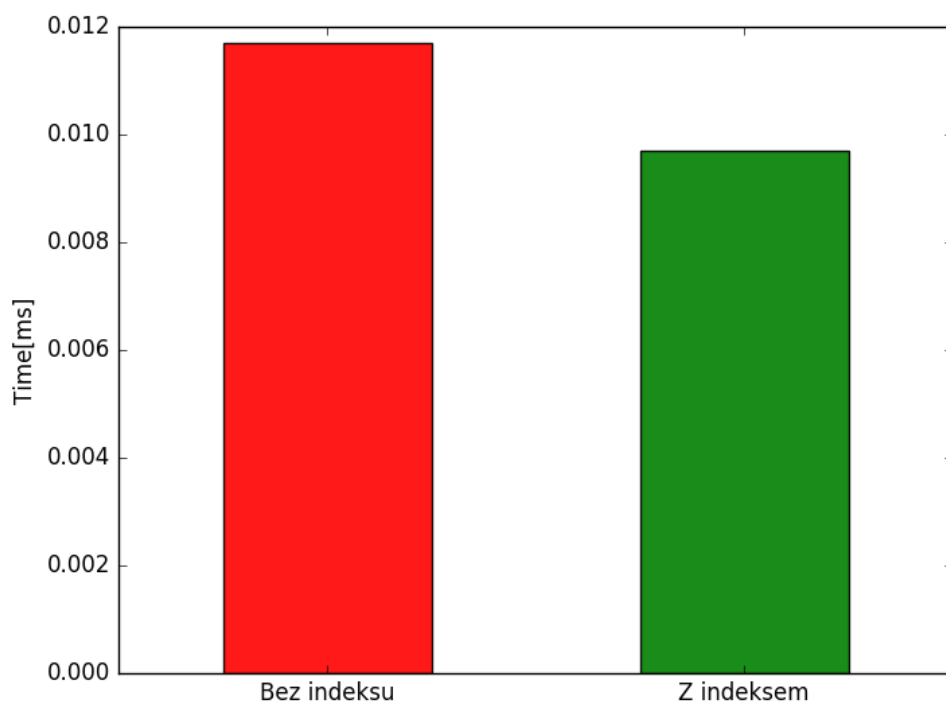
Listing 26: Zapytanie testowe przed i po utworzeniu indeksu

W przykładzie sprawdzam czas wykonania zapytania przed i po utworzeniu indeksu. Zapytanie wykonuje dziesięciokrotnie i wyliczam średnią arytmetyczną. Realizuje to przy pomocy skryptu Python:

```
1 def test2():
2     t = []
3     query = "SELECT \"ObjectID\", data->'person'->>'LastName'::text
4             AS Surname FROM personjsonb WHERE data->'person'->>'LastName'
5             '::text LIKE 'T%';"
6     query_c = "CREATE INDEX surname on personjsonb (((data->'person'
7               ->>'LastName')::text));"
8     query_d = "DROP INDEX surname"
9     conn1 = psycopg2.connect(database="Inz", user="postgres",
10                              password="Password", host="127.0.0.1", port="5433")
11     cur1 = conn1.cursor()
12     t0 = time.time()
13     for i in range(10):
14         cur1.execute(query)
15     t1 = time.time()
16     t.append((t1-t0)/10)
17     cur1.execute(query_c)
18     conn1.commit()
19     t0 = time.time()
20     for i in range(10):
21         cur1.execute(query)
22     t1 = time.time()
23     t.append((t1-t0)/10)
24     f=open('test2.dat', "w")
25     f.write(str(t)+"\n")
26     f.close()
27     cur1.execute(query_d)
28     conn1.commit()
```

Listing 27: Funkcja testująca indeksy w Pythonie

Oto otrzymany wynik:



Rysunek 22: Czas wykonania przykładowego zapytania z użyciem i bez użycia indeksu
Na podstawie wykresu widać minimalny wzrost wydajności w przypadku użycia indeksów.

5 Bazy dokumentowe

5.1 Couchbase Server

Couchbase Server jest wieloplatformowym narzędziem służącym do zarządzania nierelacyjną, dokumentową bazą danych. System pojawił się na rynku w 2010 roku. Wtedy był znany jako Membase Server.[3] Wśród twórców tego oprogramowania znajdują się osoby współtworzące technologię memcached, której założeniem jest przechowywanie danych w pamięci RAM, co w znacznym stopniu zwiększa wydajność pracy. Takie rozwiązanie jest wspierane również przez omawiane narzędzie, chociaż Couchbase ma możliwość przechowywania danych na rzeczywistych dyskach.

Do zarządzania naszym serwerem mamy do dyspozycji przeglądarkowy interfejs, a także wersję konsolową cbq. W ramach serwera możemy tworzyć kilka węzłów, na których tworzy się pojemniki na dane (ang. *Data Buckets*), do których możemy przypisać pamięć RAM, która ma zarządzać pojemnikiem. Couchbase przechowuje dane w postaci dokumentów JSON.

5.1.1 N1QL

Bezapelacyjną zaletą Couchbase Server jest język N1QL, który służy do zarządzania dokumentami w bazie. Jego składnia jest bliźniacza do języka SQL. Utworzyłem pojemnik 'Person_Bucket' z dokumentami JSON (zestaw pierwszy przy prezentacji PostgreSQL). W Couchbase Server mamy możliwość wyboru dwóch rodzajów pojemników - Couchbase Bucket i Memcached Bucket. Pierwszy rodzaj zapisuje dane na dyskach, a drugi używa tylko pamięci RAM, jednak tylko ten pierwszy wspiera zapytania N1QL, dlatego go wybrałem. Oto przykładowe polecenia w N1QL:

Przykład 1.

```
1 CREATE PRIMARY INDEX 'person_index' ON 'Person_Bucket' USING VIEW;
```

Listing 28: Przykład 1.

W celu umożliwienia przeszukiwania pojemnika, należy utworzyć na nim indeks główny.

Przykład 2.

```
1 SELECT count(person) FROM 'Person_Bucket';
```

Listing 29: Przykład 2.

Proste zliczenie dokumentów w pojemniku.

Przykład 3.

```
1 SELECT person.LastName, count(person.LastName) FROM 'Person_Bucket' GROUP BY person.LastName;
```

Listing 30: Przykład 3.

Zgrupowanie przy pomocy wartości klucza "LastName" znajdującego się w dokumencie zagnieżdżonym "person" i zliczenie liczby ich wystąpienia.

Przykład 4.

```
1 SELECT DISTINCT address.city AS City FROM 'Person_Bucket' ORDER BY City ASC;
```

Listing 31: Przykład 4.

Wyselekcjonowanie unikalnych wartości klucza "city" i posortowanie ich.

Przykład 5.

```
1 SELECT person.LastName FROM 'Person_Bucket' WHERE person.LastName LIKE 'T%' AND ARRAY_COUNT(email) = 2;
```

Listing 32: Przykład 5.



Rysunek 23: Fragment wyniku zapytania z przykładu 5.

Selekcja wartości klucza "LastName" w wypadku, gdy zaczyna się od 'T' i rozmiar tablicy "email" w danym dokumencie wynosi 2.

Przykład 6.

```
1 SELECT person, address FROM 'Person_Bucket' WHERE email IS MISSING;
```

Listing 33: Przykład 6.

Selekcja dokumentów zagnieżdżonych "person" i "address" w wypadku, gdy w danym dokumencie nie istnieje klucz "email".

Przykład 7.

```
1 UPDATE 'Person_Bucket' i SET i.email = [i.person.LastName || '@new.pl'] WHERE email IS MISSING LIMIT 3 RETURNING i;
```

Listing 34: Przykład 7.

Dodanie tablicy o kluczu "email" do dokumentów, gdzie ten klucz nie występuje. W tablicy znajduje się adres e-mail wygenerowany na podstawie klucza "LastName". Dodatkowo występuje klauzula LIMIT 3 - aktualizacja w maksymalnie trzech dokumentach.

Przykład 8.

```
1 DELETE FROM 'Person_Bucket' WHERE person.personID >= 18000;
```

Listing 35: Przykład 8.

Usuwanie dokumentów z pojemnika, których wartość klucza "personID" jest większa, bądź równa 18000.

W powyższych zapytaniach widzimy, że do zagłębiania w głąb dokumentów JSON używa się kropki. Można tam dostrzec również przykładowe funkcje oraz operatory do przetwarzania tychże dokumentów (ARRAY_COUNT(), IS MISSING). Istotną różnicą pomiędzy SQL, a N1QL jest format zwróconych danych. W przypadku SQL dane zwracane są jako tabela, a w drugim wypadku jako dokument, lub zbiór dokumentów JSON.

5.2 MongoDB

MongoDB to system zarządzania dokumentową bazą danych. Podobnie jak omawiany wcześniej Couchbase Server jest wieloplatformowym i darmowym rozwiązaniem No SQL-owym. Dane przechowywane są w kolekcjach, do których dodaje się dokumenty w formacie JSON. Te dokumenty przechowywane są w postaci BSON (ang. *Binary JSON*), czyli odpowiedniku jsonb z PostgreSQL. Pierwsza wersja oprogramowania została wydana w 2009 i od tego momentu deweloperzy poczynili duży postęp nad wydajnością i możliwościami systemu.[2] Obecnie MongoDB jest jedną z popularniejszych baz dokumentowych na rynku. Tak samo jak omawiane wcześniej rozwiązanie MongoDB wspiera replikację danych oraz pozwala na horyzontalne skalowanie.

Do obsługi serwera mamy do dyspozycji kilka możliwości, między innymi konsole mongo, a także graficzne interfejsy takie jak Robomongo, czy adminMongo.

5.2.1 Przeszukiwanie kolekcji

Do przeszukiwania i przetwarzania danych MongoDB dysponuje własnym podejściem, które jest odrębne od prezentowanego w przypadku PostgreSQL SQL-a. Po utworzeniu kolekcji i zaimportowaniu do niej dokumentów mamy w zanadru kilkanaście funkcji, które wykonujemy bezpośrednio na kolekcji. Oto kilka z nich: find(), group(), insert(), aggregate(), update(), remove(). Kolekcja person_col zawiera te same dokumenty co Person_Bucket z Couchbase Server. Poniżej znajdują się przykładowe zapytania, realizujące to co przykłady prezentowane dla CouchbaseServer:

Przykład 1.

```
1 db.person_col.find().count();
```

Listing 36: Przykład 1.

Zliczenie dokumentów w kolekcji.

Przykład 2.

```
1 db.person_col.aggregate([{"$group" : { "_id" : "$person.LastName" , "count" : { "$sum" : 1 } } }])
```

Listing 37: Przykład 2.

Grupowanie po wartościach klucza "LastName" i zliczenie ich wystąpienia.

Przykład 3.

```
1 db.person_col.distinct("address.city").sort();
```

Listing 38: Przykład 3.

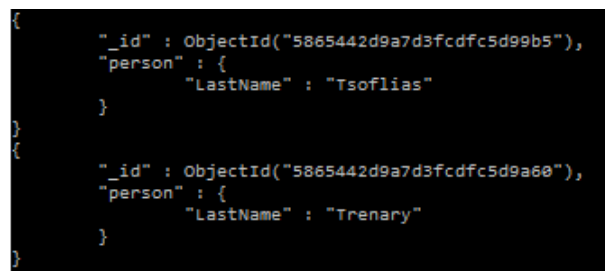
Wyekstrahowanie posortowanej listy unikalnych wartości klucza "city".

Przykład 4.

```
1 db.person_col.find({ "person.LastName": /T.*/, "email": { $size: 2 } }, { "person.LastName": 1 }).pretty();
```

Listing 39: Przykład 4.

Wyszukanie wartości klucza "LastName", gdy zaczynają się od "T" i liczba adresów e-mail przypisanych do danego dokumentu wynosi 2.



```
{
  "_id" : ObjectId("5865442d9a7d3fcd9b5"),
  "person" : {
    "LastName" : "Tsoflias"
  }
}
{
  "_id" : ObjectId("5865442d9a7d3fcd9b6"),
  "person" : {
    "LastName" : "Trenary"
  }
}
```

Rysunek 24: Fragment wyniku zapytania z przykładu 4.

Przykład 5.

```
1 db.person_col.find({ "email": { $exists: null, $ne: true } }, { person: 1, address: 1 });
```

Listing 40: Przykład 5.

Wyszukanie wartości dokumentów zagnieżdżonych "person" i "address", gdy w dokumencie głównym nie ma klucza "email".

Przykład 6.

```
1 db.person_col.find({ "email": { $exists: null, $ne: true } }).limit(3).
  snapshot().forEach(
2     function (elem) {
3         db.person_col.update(
4             { _id: elem._id },
5             { $set:
6                 { 'email': [ elem.LastName + '@new.com' ] }
7             });
8     });
```

Listing 41: Przykład 6.

Dodanie do maksymalnie trzech dokumentów klucza "email" z przypisaną tablicą z adresem e-mail wygenerowanym na podstawie wartości klucza "LastName", gdy w danym dokumencie nie znajdował się wcześniej klucz "email".

Przykład 7.

```
1 db.person_col.remove({$where: "this.person.personID >= 18000"});
```

Listing 42: Przykład 7.

Usuwanie dokumentów z kolekcji, gdy wartość klucza "personID" jest większa, bądź równa 18000.

Zapytania w MongoDB na samym początku mogą wydawać się bardzo skomplikowane, jednak w istocie często są łatwiejsze do napisania, niż te w SQL, czy N1QL. Na pewno w przypadku tego rozwiązania mamy w dyspozycji najwięcej różnego rodzaju operacji w stosunku do dokumentów JSON, niż w PostgreSQL i Couchbase. Ciekawą możliwością jest używanie kodu JavaScriptu wewnątrz zapytań (Przykład 6.).

5.3 Porównanie baz dokumentowych z PostgreSQL-em

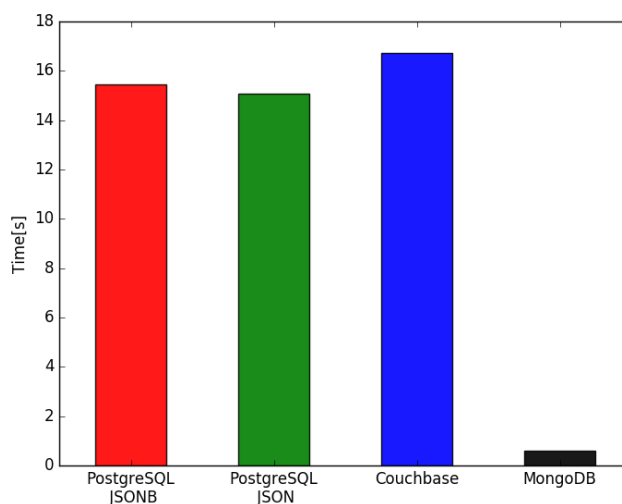
W pierwszej części tego zestawienia skupiłem na zestawieniu aspektów wydajnościowych. Jako zestaw testowy używam tego zbioru dokumentów JSON, który używałem przy prezentacji typów json i jsonb w PostgreSQL-u - tabele personjson, personjsonb z Postgresa, pojemnik Person_Bucket z Couchbase Server i kolekcja person_col z MongoDB. Przygotowałem kilka przypadków testowych i zmierzyłem czas ich wykonywania w każdej z baz. Pierwszym przypadkiem jest wprowadzenie danych do każdej z baz. Kilka kolejnych to wydobywanie danych oraz ich przetworzenia. Czasy wykonywania tych zapytań mierzyłem z poziomu skryptu napisanego w Pythonie z wyjątkiem zapytań w MongoDB, których wykonanie nadzorowałem z poziomu bazy. Ostatni przypadek testowy to sprawdzenie jak dużo miejsca na dysku zajmują przechowywane dane. Przy przypadkach testowych prezentuję zapytanie jakie jest testowane, krótki komentarz i wykres przedstawiający czas wykonanie lub rozmiar zajętego miejsca na dysku.

5.3.1 Wprowadzenie danych

```
1 \i 'person_insert_jsonb.sql'
2 \i 'person_insert_json.sql'
3 mongoimport --db test --collection person_col --drop < person.json
4 cbdocloader -n localhost:8091 -u Administrator -p password -b
   Person_Bucket Person_Bucket.zip
```

Listing 43: Komendy wprowadzające dane

Długość wykonania dwóch pierwszych poleceń mierze bezpośrednio w bazie, natomiast dwa kolejne zwracają czas ich wykonania.



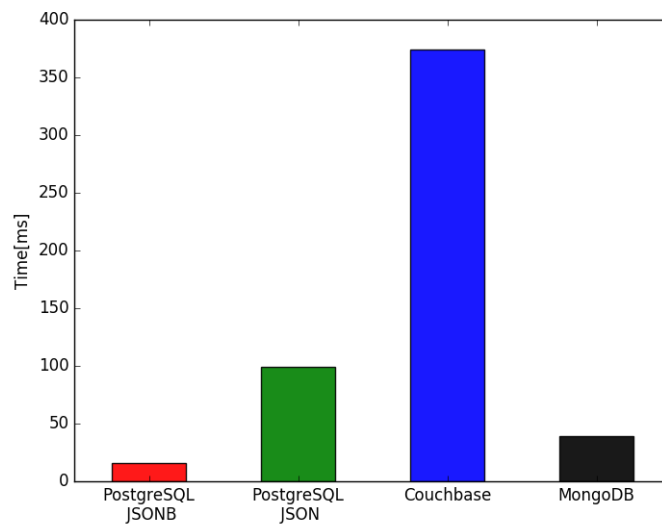
Rysunek 25: Czas wprowadzania danych do poszczególnych baz

5.3.2 Zapytania nr 1

```
1 SELECT max(cast(data->'person'->>'personID' AS int)) FROM personjsonb;
2 SELECT max(cast(data->'person'->>'personID' AS int)) FROM personjson;
3 SELECT max(person.personID) FROM 'Person_Bucket';
4 db.person_col.find({},{ 'person.personID':1}).sort({'person.personID':
  -1}).limit(1)
```

Listing 44: Pierwsza grupa zapytań

Cel zapytań: uzyskanie maksymalnej wartości klucza "personID" w dokumentach.



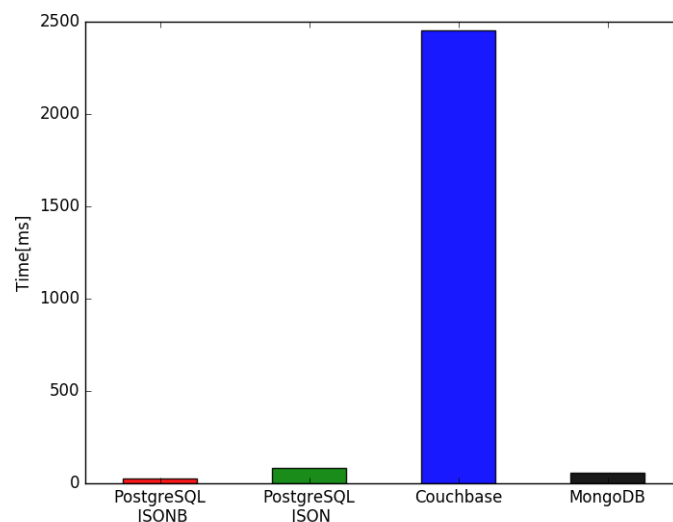
Rysunek 26: Czas wykonania zapytań nr 1 w poszczególnych bazach

5.3.3 Zapytania nr 2

```
1 SELECT data #> '{person, LastName}' AS NAME, COUNT(*) FROM public .
   personjsonb GROUP BY NAME;
2 SELECT data -> 'person' ->> 'LastName' AS NAME, COUNT(*) FROM public .
   personjson GROUP BY NAME;
3 SELECT person.LastName, count(person.LastName) FROM 'Person_Bucket' GROUP
   BY person.LastName;
4 db.person_col.aggregate([{"$group" : { _id: "$person.LastName", count: { $sum
   : 1 }}}]);
```

Listing 45: Druga grupa zapytań

Cel zapytań: zgrupowanie przy użyciu wartości klucza "LastName" wraz z zliczeniem wystąpień wartości tego klucza.



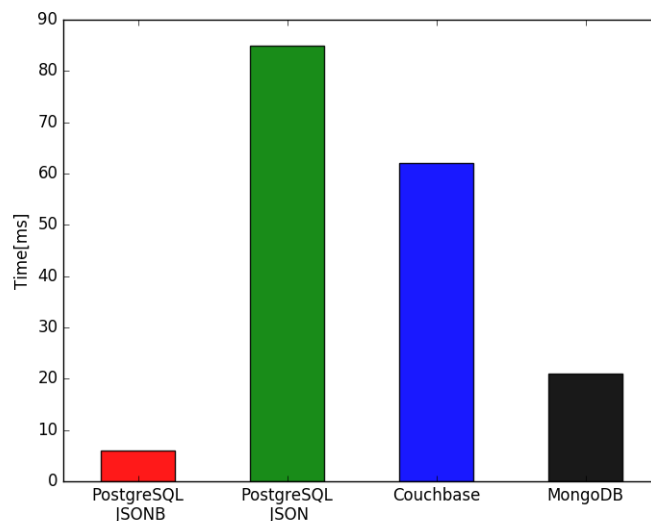
Rysunek 27: Czas wykonania zapytań nr 2 w poszczególnych bazach

5.3.4 Zapytania nr 3

```
1 SELECT data FROM public.personjsonb WHERE data ? 'email';
2 SELECT data FROM public.personjson WHERE data::jsonb ? 'email';
3 SELECT data FROM 'Person_Bucket' WHERE email IS NOT MISSING;
4 db.person_col.find({ "email": { $exists: true, $ne: null } });
```

Listing 46: Trzecia grupa zapytań

Cel zapytania: wyszukanie dokumentów, w których występuje klucz "email".



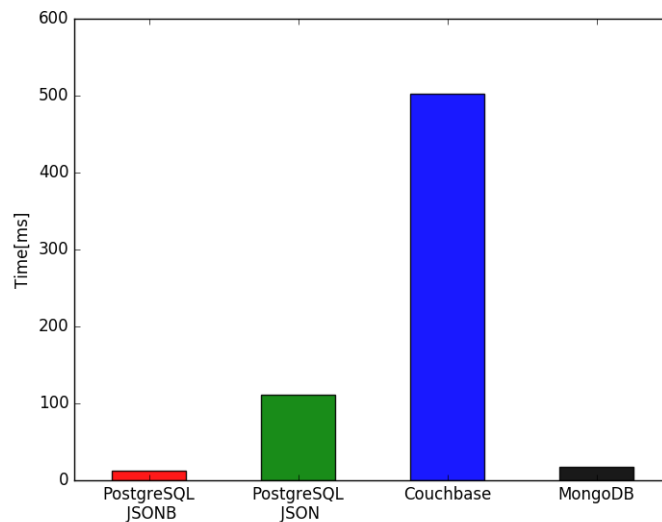
Rysunek 28: Czas wykonania zapytań nr 3 w poszczególnych bazach

5.3.5 Zapytania nr 4

```
1 SELECT "ObjectID", data->'person'->>'LastName' AS Surname FROM
   personjsonb WHERE data->'person'->>'LastName' LIKE 'T%' AND
   jsonb_array_length(data->'email')=2;
2 SELECT "ObjectID", data->'person'->>'LastName' AS Surname FROM personjson
   WHERE data->'person'->>'LastName' LIKE 'T%' AND json_array_length(
   data->'email')=2;
3 SELECT person.LastName FROM 'Person_Bucket' WHERE person.LastName LIKE 'T
   %' AND ARRAY_COUNT(email)=2;
4 db.person_col.find({ "person.LastName": /T.*/, "email": { $size: 2 } }, { "
   person.LastName": 1 }).pretty();
```

Listing 47: Czwarta grupa zapytań

Cel zapytań: wyszukanie danych z dokumentów, gdzie wartość klucza "LastName" zaczyna się od "T" i długość tablicy "email" wynosi 2.



Rysunek 29: Czas wykonania zapytań nr 4 w poszczególnych bazach

5.3.6 Zapytania nr 5

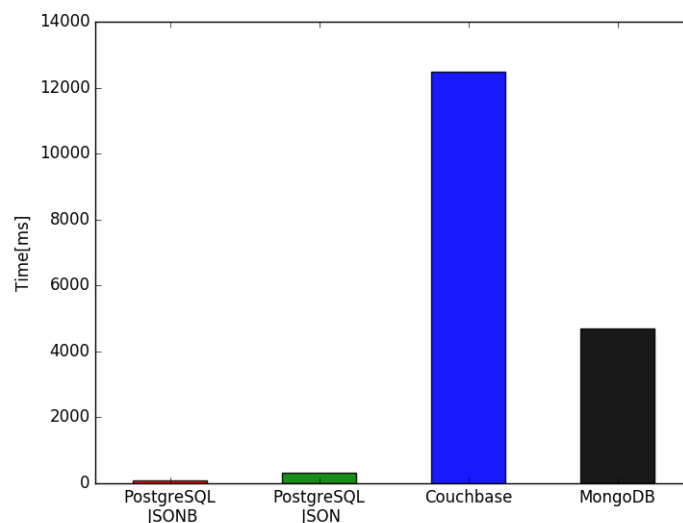
```

1 UPDATE public.personjsonb d SET data = (SELECT data || ('{"email":[' ||
  quote_ident(dd.data->'person'->'LastName' || '@hotmail.com') || ']]')::
  jsonb FROM public.personjsonb dd WHERE d."ObjectID"=dd."ObjectID")
  WHERE not d.data ? 'email';
2 UPDATE public.personjson d SET data = (SELECT data::jsonb || ('{"email":['
  ' || quote_ident(dd.data->'person'->'LastName' || '@hotmail.com') || ']]'
  )::jsonb FROM public.personjson dd WHERE d."ObjectID"=dd."ObjectID")
  ::json WHERE not d.data::jsonb ? 'email';
3 UPDATE 'Person_Bucket' i SET i.email = [i.person.LastName || '@new.pl']
  WHERE email IS MISSING;
4 db.person_col.find({'email': { $exists: null, $ne: true }}).snapshot().
  forEach( function (elem) {db.person_col.update({_id: elem._id },{ $set
    :{'email': [ elem.LastName + '@new.com' ]}}) });

```

Listing 48: Piąta grupa zapytań

Cel zapytań: aktualizacja dokumentów, w których nie występuje klucz "email".



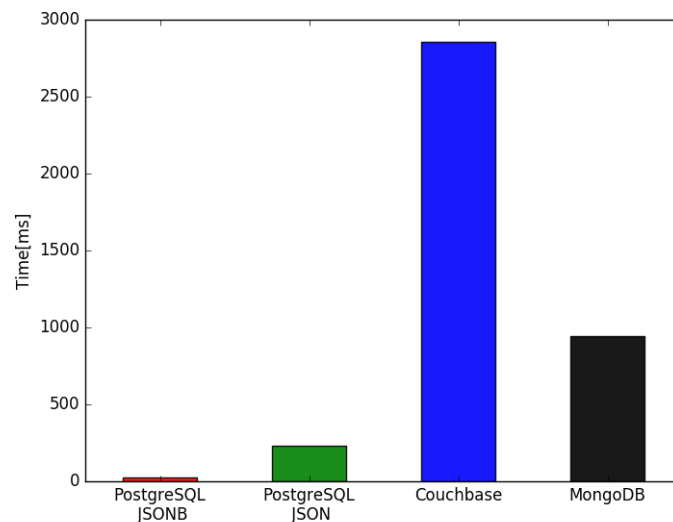
Rysunek 30: Czas wykonania zapytań nr 5 w poszczególnych bazach

5.3.7 Zapytanie nr 6

```
1 DELETE FROM personjsonb WHERE (cast(data->'person'->>'personID' AS int))  
   >=18000;  
2 DELETE FROM personjson WHERE (cast(data->'person'->>'personID' AS int))  
   >=18000;  
3 DELETE FROM 'Person_Bucket' WHERE person.personID >= 18000;  
4 db.person_col.remove({$where: "this.person.personID >= 18000"});
```

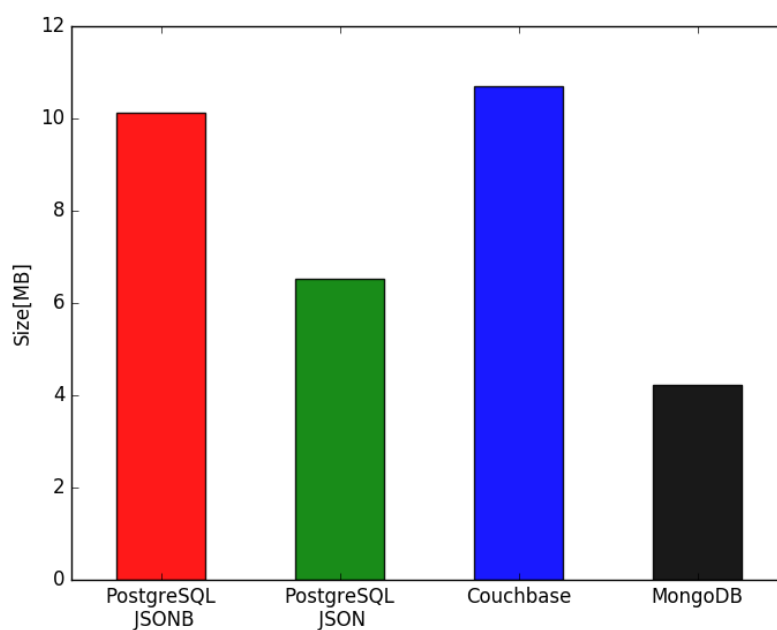
Listing 49: Szósta grupa zapytań

Cel zapytań: skasowanie dokumentów z bazy.



Rysunek 31: Czas wykonania zapytań nr 6 w poszczególnych bazach

5.3.8 Rozmiar na dysku



Rysunek 32: Zajęty rozmiar na dysku w poszczególnych rozwiązaniach

5.3.9 Podsumowanie wydajności

Przedstawione wykresy dają nam dobry obraz działania przedstawionych baz. W przypadku wprowadzania danych zdecydowanym liderem okazała się baza MongoDB, która wyprzedza pozostałe rozwiązania. Potwierdziła się również przedstawiona wcześniej teza zgodnie, z którą json (PostgreSQL) jest szybszy we wprowadzaniu danych od jsonb (minimalnie), a wolniejszy w ich przetwarzaniu. Prawie we wszystkich testach najwolniej działał Couchbase, nawet pomimo stworzenia dodatkowych indeksów, jednak prawie wszystkie testy zakładały użycie N1QL, a więc zmuszony byłem do korzystania z Couchbase Bucket (alternatywą jest Memcached Bucket, który nie wspiera N1QL). Typ jsonb z PostgreSQL i baza MongoDB działają ze zbliżoną prędkością jeśli chodzi o przeszukiwanie danych i ich grupowanie, a json jest trochę wolniejszy. W aspekcie aktualizacji i kasowania danych najwydajniej pracuje Postgres. Ostatnim przypadkiem testowym był pomiar rozmiaru zajętego na dysku. Najkorzystniej wypadła baza MongoDB.

Podsumowując, MongoDB oraz jsonb z PostgreSQL działają porównywalnie. W niektórych przypadkach korzyść widać po jednej ze stron, a w innych po drugiej. Typ json z Postgresa jest wyraźniej wolniejszy od jsonb, więc w praktyce jest nieopłacalny.

5.3.10 Spojrzenie z innej strony

Patrząc z punktu widzenia możliwości przetwarzania dokumentów JSON w przedstawionych bazach to tu widzę przewagę baz dokumentowych. Z pośród prezentowanych systemów największą gamą rozwiązań w tej kwestii dysponuje MongoDB. Inną pozytywną cechą wszystkich porównywanych wyżej baz jest możliwość tworzenia dodatkowych indeksów nawet na polach dokumentów, co zwiększałoby szybkość przeszukiwań.

Pomijając aspekty wydajnościowe, przy decyzji o postawieniu na konkretny rodzaj bazy, mając w zanadru czysto dokumentową i relacyjną z typem obsługującym dokumenty należy rozważyć konsekwencje takiego wyboru. Z jednej strony obsługa transakcji może być niezbędna, albo pomysł, aby wkomponować dokumenty w model relacyjny może być kuszący, jednak należy pamiętać, że funkcjonalność odnosząca się do tychże dokumentów, którą oferują bazy czysto dokumentowe jest znacznie większa, niż w modelu relacyjnym. Dodatkowo nie można zapominać o głównym atucie podejścia nierelacyjnego - skalowalności horyzontalnej.

6 Bibliografia

- [1] PostgreSQL Documentaion: <https://www.postgresql.org/docs/>
- [2] MongoDB Documentation: <https://docs.mongodb.com/>
- [3] Couchbase Server Documentation: <https://developer.couchbase.com/>
- [4] NoSQL: <http://nosql-database.org/>
- [5] Piotr Zieliński: *Wprowadzenie do NoSQL*, <https://msdn.microsoft.com/pl-pl/dn912483.aspx> (22.12.2016)
- [6] A Timeline of Database History : <http://www.quickbase.com/articles/timeline-of-database-history> (28.12.2016)
- [7] Dan Sullivan: *NoSQL For Mere Mortals*, Helion 2015, ISBN: 978-83-283-2488-6
- [8] <https://www.quora.com/What-is-the-relation-between-SQL-NoSQL-the-CAP-theorem-and-ACID> (22.12.2016)
- [9] <http://www.json.org/json-pl.html> (18.12.2016)