

Übersicht über Datenbank-Index-Typen in bekannten Datenbanksystemen mit PostgreSQL Benchmark

Vladimir, Brazhnik

Präsentation im MSE-DB-Seminar (online), Frühlingssemester 2021

Mo. 12. Juli 2021, 17:00 - 18:00 CEST



Inhaltsüberblick

1. Tabellarischer Vergleich aktueller Indextypen
2. Benchmark zwischen einer relationalen und mit JSONB-Objekten verschachtelten Tabelle

Tabellarischer Vergleich aktueller Indextypen

Ausführliche Version in der Seminar Arbeit:

- Link: <https://drive.switch.ch/index.php/s/VBvrbY2J39yrDOr>
- Name: „Übersicht über Datenbank Index-Typen in bekannten Datenbanksystemen - Paper und Dokumentation.pdf“

Zusammengefasste Version als Blogpost (vom Stefan Keller und Simon Wild):

- Link: <https://md.coredump.ch/s/73PZG-btU#>
- Name: „Indizes in PostgreSQL - Ein Überblick“

PostgreSQL Indexe und Kriterien

PostgreSQL Indexe:

- B-Tree
- GiST
- SP-GiST
- Hash
- GIN
- RUM
- BRIN
- Bloom

Kriterien:

- Anwendungsfeld
- Erstellzeit
(sehr langsam, langsam, mässig, schnell)
- Festplattengröße
(klein, mittel, groß)
- Unterstützte Operatoren

| Indexes Criteria | B-TREE | GIST | SP-GIST | Hash | GIN | RUM | BRIN | Bloom |
|--|---|---|---|--|--|---|---|--|
| Kategorie | (balancierter) Baumindex | (balancierter) Baumindex | (nicht-balancierter, platz-partitionierter) Baumindex | Hash Index | Inverted Index | Inverted Index | Block-Range-Index | Hash Index |
| Anwendungsfeld | Geeignet für allgemeine Equality, Comparison und Pattern Matching Operatoren bei häufigen Datentypen (numeric und timestamp/date), welche sortiert werden können | Geeignet für modernere Datentypen wie beispielsweise Geometrie Daten, Textdokumente oder Bilder, bei denen die Vergleichs-Operatoren nicht genügen und zusätzliche, erweiterbare Daten und Queries benötigt werden. Somit auch geeignet für full-text-search, nearest neighbor search sowie das Arbeiten mit Geodaten. | Ist eine Erweiterung des GIST Indexes, welche im Gegensatz zum GIST Index speziell für Datentypen geeignet ist, bei welchen die Suchräume rekursiv in Partitionen unterteilt wurden. Außerdem geeignet für sehr große Datenmengen, welche nicht-balanciert / ungleichmäßig verteilt sind und sich überlappen. | Geeignet und sehr performant bei Daten, welche nur Equality-Operatoren für Point Queries benötigen. | Geeignet und sehr performant für Abfragen, welche nach Elementwerten innerhalb zusammengesetzter Datentypen wie hstore, array, jsonb, tsvector, oder range types. Somit vor allem geeignet für die Text-Suche. | Ist eine Optimierung des GIN Indexes, welche dessen Implementierung referenziert und für die Text-Suche und für zusammengesetzte Queries optimiert wurde. Baut auf dem GIN Index auf und ist somit geeignet für die selben Anwendungen wie der GIN Index. | Geeignet und platzsparend für große (>1M Tupel), natürlich-sequenziellen Daten, bei denen hauptsächlich INSERTs und Queries mit einer großen Ergebnismenge ausgeführt werden. | Geeignet und performant für das Überprüfen, ob ein Wert existiert oder Mitglied einer Menge ist. Performanter für Tabellen mit vielen Attribute auf denen Abfragen nach der Existenz in beliebigen Kombinationen gestellt werden können. |
| Erstellzeit [sehr langsam, langsam, mäßig, schnell] | Mäßig: Balancierte Baumstruktur muss erstellt werden, wobei Zeiger auf die Nachfolger-Knoten in den einzelnen Knoten erstellt werden. Zusätzlich werden die Daten der indierten Werte als Schlüssel und Referenzen auf die Daten als Values sortiert in den Baum eingefügt, sodass dieser balanciert bleibt. | Mäßig-Langsam: Ähnlich wie B-Tree eine balancierte Baumstruktur, welche aber zu anderen balancierten Baumstrukturen erweitert werden kann. Somit variiert die Herstellzeit abhängig von gewählten Baumstruktur und den angepassten Datentypen. | Mäßig-Langsam: Ähnlich wie beim GIST variiert die Herstellzeit abhängig von gewählten nicht balancierten Baumstruktur (beispielsweise Quad_Tree, K-D-Tree, Radix-Tree) und den angepassten Datentypen. | Schnell bis mäßig: Hash Tabelle wird mit den berechneten Hashwerten der indierten Werte als Keys und den Referenzen auf die Daten als Values befüllt. | Langsam: Langsame Herstellzeit, da jeder individuelle Wert innerhalb eines zusammengesetzten Datentyps indiziert werden muss. Herstellzeit ist kompakter für die Indexierung von Daten, bei denen die Schlüssel mehrfach vorkommen. | Langsam bis sehr langsam: Berechnet zusätzlich zu den referenzierten Informationen vom GIN Index noch die zusätzlichen Informationen wie die Zeitstempel oder Lexempositionen. | Sehr schnell: Speicherung nur vom Mindest- und Maximalwert eines Blockes als Referenz zu den gewünschten Bereichen | Schnell: Erstellung einer flachen, auf dem Bloom-Filter aufbauenden Struktur, mit Metadaten und einer Bit-Signatur ob der Werte in der Tabelle existieren könnte. |
| Festplattengröße [klein, mittel, groß] | Mittel: Speichert jeden indierten Wert als Schlüssel, Referenzen zu den jeweiligen Daten als Value, sowie zusätzliche Zeiger auf Nachfolger-Knoten. | Mittel bis groß: Größe variiert abhängig von der gewählten balancierten Baumstruktur. Außerdem besitzt der GIST Index im Gegensatz zum GIN Index mehr Größenbeschränkungen und hat in Vergleich zum GIN Index einen geringeren Festplatten-Footprint. | Mittel bis groß: Ähnlich wie beim GIST variiert die Größe abhängig von der gewählten nicht-balancierten Baumstruktur. | Klein bis mittel: Speichert einen aus den indierten Datenwerten berechneten Hashwert als Key und die Referenzen auf die Daten als Value. Da der Hashwert unabhängig von der Größe der Datentypen ist, benötigt der Hash Index für große Datentypen weniger Speicherplatz. | Groß: Speichert jeden individuellen Wert innerhalb eines zusammengesetzten Datentypen und dessen posting list mit den jeweiligen Zeilen IDs in welchen der Werte vorkommt. Ist wesentlich größer als andere Indizes wie beispielsweise der B-Tree, da jeder individuelle Wert hinterlegt werden muss. Ist jedoch kompakter für die Speicherung von Daten, bei denen die Schlüssel mehrfach vorkommen. | Groß bis sehr groß: Referenziert die gespeicherten Informationen vom GIN Index und speichert noch zusätzliche Informationen wie die Lexempositionen / Zeitstempel und die generischen WAL Aufzeichnungen, weshalb noch zusätzlicher Speicherplatz benötigt wird. | Sehr klein: Speichert nur den Mindest- und Maximalwert eines Blockes als Referenz zu dem Bereich, in welchem der indizierte Wert liegt. | Klein: Speicherung eines Arrays mit n Bits und Hash-Signaturen als Verweise auf die indierten Werte |
| Unterstützte Abfragen | Operatoren: Equality, Vergleich- sowie Pattern-Matching Operatoren Abfragen: Point Queries, Multipoint Queries, Range Queries und Compound Queries | Operatoren: Ermöglicht Entwicklung eigener Datentypen sowie deren Zugriffsmethoden sowie die Erweiterung von Datentypen und Queries mittels spezieller Operatoren für geometrische Datentypen wie box, circle, point, polygon (box_ops, circle_ops, point_ops, poly_ops), Netzwerk Adressen Datentypen wie inet und cidr (inet_ops), jegliche Bereichs Datentypen (range_ops) und Text Suche Datentypen wie tsvector und tsquery (ts_vector_ops & ts_query_ops). Abfragen: Unterstützte Abfragen hängen von den gewählten Operatoren ab. | Operatoren: Ermöglicht Entwicklung eigener Datentypen sowie deren Zugriffsmethoden sowie die Erweiterung von Datentypen und Queries mittels spezieller Operatoren für geometrische Datentypen wie box, point und polygon (box_ops, kd_point_ops, quad_point_ops, poly_ops), Netzwerk Adressen Datentypen wie inet und cidr (inet_ops), jegliche Bereichs Datentypen (range_ops) und Text (text_ops). Abfragen: Unterstützte Abfragen hängen von den gewählten Operatoren ab. | Operatoren: NUR Equality-Operator Abfragen: NUR Point Queries | Operatoren: Spezielle Operatoren für Arrays (arrays_ops), JSONB (jsonb_ops & json_path_ops) sowie tsvector (ts_vector_ops) sowie viele weitere Operatoren für nicht zusammengesetzte Datentypen. Ermöglicht aber auch die Entwicklung und Nutzung von eigenen Datentypen und Zugriffsmethoden. Abfragen: Unterstützte Abfragen hängen von den gewählten Operatoren ab. | Operatoren: Spezielle Operatoren ergänzend zu den Operatoren von GIN Index für häufig verwendete, einfachere Datentypen (rum_type_ops), Arrays (rum_anarray_ops, rum_anarray_addon_ops), sowie tsvector (rum_tsvector_ops, rum_tsquery_ops, rum_ts_vector_addon_ops, rum_tsvector_hash_ops, rum_tsvector_hash_addon_ops). Abfragen: Unterstützte Abfragen hängen von den gewählten Operatoren ab. Jedoch wird die Phrasensuche auf Indexlevel sowie das Ordnen von Daten mittels Zeitstempel ohne zusätzliche CPU-Verarbeitung besser unterstützt, als beim GIN Index. | Operatoren: Hauptsächlich minmax und Inclusion-Operatoren Abfragen: Vorallem Range Queries oder Compound Queries, welche eine große Ergebnismenge zurückgeben | Operatoren: NUR Equality-Operator und Operatoren Klassen für int4 und text Datentypen Abfragen: NUR Point Queries |

B-Tree

Kategorie

- (balancierter) Baumindex

Anwendungsfeld

- Abdeckung der meisten häufigen Datentypen, welche sortiert werden können (numerisch, time/date)
- Equality, Range und Pattern Matching Queries

B-Tree

Erstellzeit

- Mässig
- Balancierte Baumstruktur
- Erstellen von Zeigern auf Nachfolger-Knoten, sowie sortiertes Einfügen von der Daten und Referenzen

Diskplatz

- Mittel
- Speicherung jedes indexierten Wertes, die Referenz zu den Daten und die Zeiger auf die Nachfolger-Knoten

Operatoren (Op.)

- Equality Op.
- Vergleich Op.
- Pattern-Matching Op.

GiST

Kategorie

- (balancierter) Baumindex

Anwendungsfeld

- Neue Datentypen, welche spezielle Operatoren benötigen (Geo-Daten, Textdokumente, Bilder)
- Full-Text-Search, Nearest-Neighbor-Search, Operationen mit Geo-Daten

GiST

Erstellzeit

- Mässig - Langsam
- Erweiterbare balancierte Baumstruktur
- Variiert von der gewählten Baumstruktur sowie dem Datentyp

Diskplatz

- Mittel - Groß
- Erweiterbare Balancierte Baumstruktur
- Variiert von der gewählten Baumstruktur sowie dem Datentyp

Operatoren (Op.)

- Operatoren für verschiedene Datentypen
- Unterstützt Entwicklung eigener Datentypen sowie deren Zugriffsmethoden

SP-GiST

Kategorie

- (nicht-balancierter, platz-partitionierter) Baumindex

Anwendungsfeld

- Datentypen, bei welchen die Suchräume rekursiv in Partitionen unterteilt wurden
- Sehr große, nicht balancierte, sowie sich überlappende Datenmengen
- Full-Text-Search, Nearest-Neighbor-Search, Operationen mit Geo-Daten

SP-GiST

Erstellzeit

- Mässig - Langsam
- Erweiterbare nicht-balancierte Baumstruktur
- Variiert von der gewählten Baumstruktur sowie dem Datentyp

Diskplatz

- Mittel - Groß
- Erweiterbare nicht-balancierte Baumstruktur
- Variiert von der gewählten Baumstruktur sowie dem Datentyp

Operatoren (Op.)

- Operatoren für verschiedene Datentypen
- Unterstützt Entwicklung eigener Datentypen sowie deren Zugriffsmethoden

Hash

Kategorie

- Hash Index

Anwendungsfeld

- Daten welche nur Equality Op. für Point Queries benötigen

Hash

Erstellzeit

- Schnell - Mässig
- Hash-Tabelle mit berechneten Hashwerten und Referenzen auf die Daten

Diskplatz

- Klein - Mittel
- Speicherung von Hash-Wert und der Referenz
- Größe des Hash-Wertes ist unabhängig von der Größe des gewählten Datentypen

Operatoren (Op.)

- Nur Equality Op.

GIN

Kategorie

- Inverted Index

Anwendungsfeld

- Für Elementwerten innerhalb zusammengesetzter Datentypen (z.B. array, jsonb)
- Geeignet für Text-Suche

GIN

Erstellzeit

- Langsam
- Indexierung jedes individuellen Wertes innerhalb des zusammengesetzten Datentypen

Diskplatz

- Groß
- Speicherung jedes individuellen Wertes innerhalb eines zusammengesetzten Datentypen und dessen posting list mit den jeweiligen IDs

Operatoren (Op.)

- Spezielle Operatoren für arrays, jsonb sowie tsvector und anderen zusammengesetzten Datentypen

RUM

Kategorie

- Inverted Index

Anwendungsfeld

- Optimierung vom GIN Index für Text-Suche und für zusammengesetzte Queries
- Geeignet für die selben Anwendungen wie der GIN Index

RUM

Erstellzeit

- Langsam - Sehr Langsam
- Indexierung jedes individuellen Wertes innerhalb des zusammengesetzten Datentypen
- Zusätzliche Berechnung der Zeitstempel und/oder Lexempositionen

Diskplatz

- Groß - Sehr Groß
- Speicherung der Referenzen des GIN Indexes (individuelle Werte innerhalb eines zusammengesetzten Datentypen und dessen posting list mit den jeweiligen IDs) und noch zusätzlichen Informationen wie Zeitstempel und Lexempositionen

Operatoren (Op.)

- Spezielle Operatoren für häufig verwendete, einfachere Datentypen, Arrays sowie tsvector/tsquery, ergänzend zu dem Operatoren vom GIN Index

BRIN

Kategorie

- Block-Range-Index

Anwendungsfeld

- Platzsparend bei grossen (>1M Tupel), natürlich-sequenziellen Daten
- Performanter für INSERTs und Queries mit grossen Ergebnismengen

BRIN

Erstellzeit

- Sehr Schnell
- Berechnung nur vom Mindest- und Maximalwert eines Blockes als Referenz zu dem gewünscht Bereich, in welchem der indexierte Wert liegt

Diskplatz

- Sehr klein
- Speicherung nur vom Mindest- und Maximalwert eines Blockes als Referenz zu dem gewünscht Bereich, in welchem der indexierte Wert liegt

Operatoren (Op.)

- Hauptsächlich Inclusion Op. Und Minmax Op.

Bloom

Kategorie

- Hash Index

Anwendungsfeld

- Platzsparend und performant für das Überprüfen, ob ein Wert existiert oder Mitglied einer Menge ist
- Geeignet für Tabellen mit vielen Attributen, auf denen Abfragen nach der Existenz in beliebigen Kombination gestellt werden können

Bloom

Erstellzeit

- Schnell
- Erstellung einer flachen, auf dem Bloom-Filter aufbauenden Struktur, mit Hash Signaturen als Verweis auf die indexierten Werte und Bit-Signaturen, zur Überprüfung ob der Werte in der Tabelle existiert

Diskplatz

- Klein
- Speicherung eines Arrays mit n Bits und Hash Signaturen

Operatoren (Op.)

- Nur Equality Op. Und Operatoren Klassen für int4 und text Datentypen

Benchmark zwischen einer relationalen und mit JSONB-Objekten verschachtelten Tabelle

Ausführliche Dokumentation in der Seminararbeit:

- Link: <https://drive.switch.ch/index.php/s/VBvrbY2J39yrDOr>
- Name: „Übersicht über Datenbank Index-Typen in bekannten Datenbanksystemen - Paper und Dokumentation.pdf“

Benchmark Repository mit README

- Link: <https://github.com/4realDev/psql-relational-vs-jsonb-benchmark>
- Name: „pgql-relational-vs-jsonb-benchmark“

Ziel und Vorgehensweise

Ermitteln der Grenzen von mit JSONB Objekten verschachtelten Datenbanken

- Benchmark zwischen einer generisch generierten relationalen Tabelle mit normalen Datentypen und einer daraus erstellten Tabelle mit geschachtelten JSONB Objekt
- Untersuchung von beiden Tabellen für jeweils fünf Datensätzen (50K, 100K, 500K, 1M, 2M)
- Vergleich der Tabellen hinsichtlich der Performance der Queries mit und ohne Index, der INSERT, UPDATE und DELETE Operationen mit Index, sowie der Index Erstellzeit und Index Größe

Schema: Relationale Tabelle

CUSTOMERS Tabelle der «Dell DVD Store Database Test Suite»

CUSTOMERID: [PK] integer
FIRSTNAME: varchar(50)
LASTNAME: varchar(50)
AGE: smallint
INCOME: integer
GENDER: varchar(1)
EMAIL: varchar(50)
PHONE: varchar(50)
ADDRESS1: varchar(50)
ADDRESS2: varchar(50)
CITY: varchar(50)
STATE: varchar(50)
ZIP: varchar(9)
COUNTRY: varchar(50)
REGION: smallint
CREDITCARDTYPE: integer
CREDITCARD: varchar(50)
CREDITEXPIRATION: varchar(50)
USERNAME: varchar(50)
PASSWORD: varchar(50)

Schema: JSONB Tabelle

CUSTOMERS Tabelle der «Dell DVD Store Database Test Suite»

| | |
|-------------------------------|------------|
| CUSTOMERID: [PK] integer | customerid |
| FIRSTNAME: varchar(50) | name |
| LASTNAME: varchar(50) | |
| AGE: smallint | personal |
| INCOME: integer | |
| GENDER: varchar(1) | |
| EMAIL: varchar(50) | contact |
| PHONE: varchar(50) | |
| ADDRESS1: varchar(50) | location |
| ADDRESS2: varchar(50) | |
| CITY: varchar(50) | |
| STATE: varchar(50) | |
| ZIP: varchar(9) | |
| COUNTRY: varchar(50) | |
| REGION: smallint | credit |
| CREDITCARDTYPE: integer | |
| CREDITCARD: varchar(50) | |
| CREDITEXPIRATION: varchar(50) | login |
| USERNAME: varchar(50) | |
| PASSWORD: varchar(50) | |

CUSTOMERS_JSONB Tabelle

```
CREATE table CUSTOMERS_JSONB as(
  SELECT
    customerid,
    jsonb_build_object(
      jsonb_build_object(
        jsonb_build_object(firstname, lastname)
        as "name",
        gender,
        age,
        income,
        jsonb_build_object(email, phone)
        as "contact"
      )
    ) as "personal",
    ...
  FROM CUSTOMERS
);
```

Queries

1. Point Query
2. Multipoint Query
3. Range Query
4. Conjunction Query (AND)
5. Disjunction Query (OR)

1. Point Query

Für CUSTOMERS Tabelle:

SELECT

firstname,
lastname,
gender,
age

FROM

customers

WHERE

firstname = 'VKUUXF'
AND lastname = 'ITHOMQJNYX';

Für CUSTOMERS_JSONB Tabelle:

SELECT

personal #> '{personal, name}' ->> 'firstname' AS "firstname",
personal #> '{personal, name}' ->> 'lastname' AS "lastname",
personal -> 'personal' ->> 'gender' AS "gender",
personal -> 'personal' ->> 'age' AS "age"

FROM

customers_jsonb

WHERE

personal@> '{"personal": {"name": {"firstname": "VKUUXF"}}}'
AND personal@> '{"personal": {"name": {"lastname": "ITHOMQJNYX"}}}';

2. Multipoint Query

Für CUSTOMERS Tabelle:

SELECT

firstname,
lastname,
gender,
age

FROM

customers

WHERE

state = 'SD';

Für CUSTOMERS_JSONB Tabelle:

SELECT

personal #> '{personal, name}' ->> 'firstname' AS "firstname",
personal #> '{personal, name}' ->> 'lastname' AS "lastname",
personal -> 'personal' ->> 'gender' AS "gender",
personal -> 'personal' ->> 'age' AS "age"

FROM

customers_jsonb

WHERE

location@> '{"location": {"state": "SD"}}';

3. Range Query

Für CUSTOMERS Tabelle:

SELECT

firstname,
lastname,
gender,
age

FROM

customers

WHERE

age >= 26;

Für CUSTOMERS_JSONB Tabelle:

SELECT

personal #> '{personal, name}' ->> 'firstname' AS "firstname",
personal #> '{personal, name}' ->> 'lastname' AS "lastname",
personal -> 'personal' ->> 'gender' AS "gender",
personal -> 'personal' ->> 'age' AS "age"

FROM

customers_jsonb

WHERE

personal -> 'personal' ->> 'age' >= '26';

4. Conjunction Query

Für CUSTOMERS Tabelle:

SELECT

firstname,
lastname,
gender,
age

FROM

customers

WHERE

age = 26
AND gender = 'F'

Für CUSTOMERS_JSONB Tabelle:

SELECT

personal #> '{personal, name}' ->> 'firstname' AS "firstname",
personal #> '{personal, name}' ->> 'lastname' AS "lastname",
personal -> 'personal' ->> 'gender' AS "gender",
personal -> 'personal' ->> 'age' AS "age"

FROM

customers_jsonb

WHERE

personal@> '{"personal": {"age": 26}}'
AND personal@> '{"personal": {"gender": "F"}}';

5. Disjunction Query

Für CUSTOMERS Tabelle:

SELECT

firstname,
lastname,
gender,
age

FROM

customers

WHERE

age = 26
OR gender = 'F';

Für CUSTOMERS_JSONB Tabelle:

SELECT

personal #> '{personal, name}' ->> 'firstname' AS "firstname",
personal #> '{personal, name}' ->> 'lastname' AS "lastname",
personal -> 'personal' ->> 'gender' AS "gender",
personal -> 'personal' ->> 'age' AS "age"

FROM

customers_jsonb

WHERE

personal@> '{"personal": {"age": 26}}'
OR personal@> '{"personal": {"gender": "F"}}';

Indexe: relationale Datenbank

- Fünf B-Tree Indexe angepasst für jede Query

```
CREATE INDEX firstname_lastname_idx ON customers USING btree (firstname, lastname);
```

```
CREATE INDEX state_idx ON customers USING btree(state);
```

```
CREATE INDEX age_above_twenty_six_idx ON customers USING btree(age) WHERE age >= 26;
```

```
CREATE INDEX age_above_twenty_six_and_gender_female_idx ON customers USING btree(age, gender)  
WHERE age = 26 AND gender='F';
```

```
CREATE INDEX age_above_twenty_six_or_gender_female_idx ON customers USING btree(age, gender)  
WHERE age = 26 OR gender='F';
```


Indexe: JSONB Datenbank

- GIN Indexe für beide JSONB Objekte “personal” und “location”
- “jsonb_path_ops” als verwendeter Operator
 - Weniger Operatoren als Standard-Operator “jsonb_ops”
 - Aber bessere Performance für diese Operatoren als “jsonb_ops”

```
CREATE INDEX personal_gin_idx ON customers_jsonb USING GIN (personal jsonb_path_ops);
```

```
CREATE INDEX location_gin_idx ON customers_jsonb USING GIN (location jsonb_path_ops);
```

Ergebnisse

| Databases (2M Records) Criteria | Relational DB without B-TREE | JSONB DB without GIN | Relational DB with B-TREE | JSONB DB with GIN |
|------------------------------------|---------------------------------|-------------------------|------------------------------|---------------------------|
| Index build time | - | - | 3462,072 | 19055,729 |
| Index size | - | - | 13 MB | 62 MB |
| Point Query | 314.41 | 1197.65 | 0.37 | 0.36 |
| Multipoint Query | 332.69 | 932.96 | 30.75 | 329.18 |
| Range Query | 1986.94 | 5616.58 | 1914.32 | 5418.65 |
| Conjunction Query (AND) | 394.51 | 966.65 | 36.01 | 53.36 |
| Disjunction Query (OR) | 1623.48 | 1886.18 | 1480.39 | 4221.31 |
| Insert | - | - | 343076,445 (05:43,076) | 719036,651 (11:59,037) |
| Delete | - | - | 26928,618 (00:26,929) | 81473,774 (01:21,474) |
| Update | - | - | 13056,619 | 55807,895 |

Ergebnisse

| Databases (2M Records) Criteria | Relational DB without B-TREE | JSONB DB without GIN | Relational DB with B-TREE | JSONB DB with GIN |
|------------------------------------|---------------------------------|-------------------------|------------------------------|---------------------------|
| Index build time | - | - | 3462,072 | 19055,729 |
| Index size | - | - | 13 MB | 62 MB |
| Point Query | 314.41 | 1197.65 | 0.37 | 0.36 |
| Multipoint Query | 332.69 | 932.96 | 30.75 | 329.18 |
| Range Query | 1986.94 | 5616.58 | 1914.32 | 5418.65 |
| Conjunction Query (AND) | 394.51 | 966.65 | 36.01 | 53.36 |
| Disjunction Query (OR) | 1623.48 | 1886.18 | 1480.39 | 4221.31 |
| Insert | - | - | 343076,445 (05:43,076) | 719036,651 (11:59,037) |
| Delete | - | - | 26928,618 (00:26,929) | 81473,774 (01:21,474) |
| Update | - | - | 13056,619 | 55807,895 |

Ergebnisse

| Databases (2M Records) Criteria | Relational DB without B-TREE | JSONB DB without GIN | Relational DB with B-TREE | JSONB DB with GIN |
|------------------------------------|---------------------------------|-------------------------|------------------------------|---------------------------|
| Index build time | - | - | 3462,072 | 19055,729 |
| Index size | - | - | 13 MB | 62 MB |
| Point Query | 314.41 | 1197.65 | 0.37 | 0.36 |
| Multipoint Query | 332.69 | 932.96 | 30.75 | 329.18 |
| Range Query | 1986.94 | 5616.58 | 1914.32 | 5418.65 |
| Conjunction Query (AND) | 394.51 | 966.65 | 36.01 | 53.36 |
| Disjunction Query (OR) | 1623.48 | 1886.18 | 1480.39 | 4221.31 |
| Insert | - | - | 343076,445 (05:43,076) | 719036,651 (11:59,037) |
| Delete | - | - | 26928,618 (00:26,929) | 81473,774 (01:21,474) |
| Update | - | - | 13056,619 | 55807,895 |

Ergebnisse

| Databases (2M Records) Criteria | Relational DB without B-TREE | JSONB DB without GIN | Relational DB with B-TREE | JSONB DB with GIN |
|------------------------------------|---------------------------------|-------------------------|------------------------------|---------------------------|
| Index build time | - | - | 3462,072 | 19055,729 |
| Index size | - | - | 13 MB | 62 MB |
| Point Query | 314.41 | 1197.65 | 0.37 | 0.36 |
| Multipoint Query | 332.69 | 932.96 | 30.75 | 329.18 |
| Range Query | 1986.94 | 5616.58 | 1914.32 | 5418.65 |
| Conjunction Query (AND) | 394.51 | 966.65 | 36.01 | 53.36 |
| Disjunction Query (OR) | 1623.48 | 1886.18 | 1480.39 | 4221.31 |
| Insert | - | - | 343076,445 (05:43,076) | 719036,651 (11:59,037) |
| Delete | - | - | 26928,618 (00:26,929) | 81473,774 (01:21,474) |
| Update | - | - | 13056,619 | 55807,895 |

Ergebnisse

| Databases (2M Records) Criteria | Relational DB without B-TREE | JSONB DB without GIN | Relational DB with B-TREE | JSONB DB with GIN |
|------------------------------------|---------------------------------|-------------------------|------------------------------|---------------------------|
| Index build time | - | - | 3462,072 | 19055,729 |
| Index size | - | - | 13 MB | 62 MB |
| Point Query | 314.41 | 1197.65 | 0.37 | 0.36 |
| Multipoint Query | 332.69 | 932.96 | 30.75 | 329.18 |
| Range Query | 1986.94 | 5616.58 | 1914.32 | 5418.65 |
| Conjunction Query (AND) | 394.51 | 966.65 | 36.01 | 53.36 |
| Disjunction Query (OR) | 1623.48 | 1886.18 | 1480.39 | 4221.31 |
| Insert | - | - | 343076,445 (05:43,076) | 719036,651 (11:59,037) |
| Delete | - | - | 26928,618 (00:26,929) | 81473,774 (01:21,474) |
| Update | - | - | 13056,619 | 55807,895 |

Ergebnisse

| Databases (2M Records) | Relational DB | JSONB DB without | Relational DB | JSONB DB |
|-------------------------|---|------------------|----------------------|-----------------------|
| Criteria | without B-TREE | GIN | with B-TREE | with GIN |
| Index | Seq Scan on customers_jsonb (cost=0.00..196685.34 rows=666667 width=128) (actual time=0.054..0.3957.507 rows=1781374 loops=1) Filter: (((personal -> 'personal'::text) ->> 'age'::text) >= '26'::text) Rows Removed by Filter: 218626 Planning Time: 0.075 ms Execution Time: 4007.599 ms | | 462,072 | 19055,729 |
| Index | | | 4 MB | 62 MB |
| Point | | | 37 | 0.36 |
| Multipoint Query | 332.69 | 932.96 | 30.75 | 329.18 |
| Range Query | 1986.94 | 5616.58 | 1914.32 | 5418.65 |
| Conjunction Query (AND) | 394.51 | 966.65 | 36.01 | 53.36 |
| Disjunction Query | Seq Scan on customers (cost=0.00..69978.00 rows=1778667 width=16) (actual time=0.014..0.756.841 rows=1781374 loops=1) Filter: (age >= 26) Rows Removed by Filter: 218626 Planning Time: 0.750 ms Execution Time: 801.116 ms | | 480.39 | 4221.31 |
| Insert | | | 43076,445 | 719036,651 |
| | | | 5:43,076) | (11:59,037) |
| Delete | | | 5928,618 (00:26,929) | 81473,774 (01:21,474) |
| Update | - | - | 13056,619 | 55807,895 |

Ergebnisse

| Databases (2M Records) Criteria | Relational DB without B-TREE | JSONB DB without GIN | Relational DB with B-TREE | JSONB DB with GIN |
|------------------------------------|---------------------------------|-------------------------|------------------------------|---------------------------|
| Index build time | - | - | 3462,072 | 19055,729 |
| Index size | - | - | 13 MB | 62 MB |
| Point Query | 314.41 | 1197.65 | 0.37 | 0.36 |
| Multipoint Query | 332.69 | 932.96 | 30.75 | 329.18 |
| Range Query | 1986.94 | 5616.58 | 1914.32 | 5418.65 |
| Conjunction Query (AND) | 394.51 | 966.65 | 36.01 | 53.36 |
| Disjunction Query (OR) | 1623.48 | 1886.18 | 1480.39 | 4221.31 |
| Insert | - | - | 343076,445 (05:43,076) | 719036,651 (11:59,037) |
| Delete | - | - | 26928,618 (00:26,929) | 81473,774 (01:21,474) |
| Update | - | - | 13056,619 | 55807,895 |

Ergebnisse

| Databases (2M Records) Criteria | Relational DB without B-TREE | JSONB DB without GIN | Relational DB with B-TREE | JSONB DB with GIN |
|------------------------------------|---------------------------------|-------------------------|------------------------------|---------------------------|
| Index build time | - | - | 3462,072 | 19055,729 |
| Index size | - | - | 13 MB | 62 MB |
| Point Query | 314.41 | 1197.65 | 0.37 | 0.36 |
| Multipoint Query | 332.69 | 932.96 | 30.75 | 329.18 |
| Range Query | 1986.94 | 5616.58 | 1914.32 | 5418.65 |
| Conjunction Query (AND) | 394.51 | 966.65 | 36.01 | 53.36 |
| Disjunction Query (OR) | 1623.48 | 1886.18 | 1480.39 | 4221.31 |
| Insert | - | - | 343076,445 (05:43,076) | 719036,651 (11:59,037) |
| Delete | - | - | 26928,618 (00:26,929) | 81473,774 (01:21,474) |
| Update | - | - | 13056,619 | 55807,895 |

Ergebnisse

Bitmap Heap Scan on customers_jsonb (cost=415.90..91261.93 rows=39800 width=128)
 (actual time=118.173..3223.771 rows=1013460 loops=1)
Recheck Cond: ((personal @> '{"personal": {"age": 26}}'::jsonb) OR (personal @> '{"personal": {"gender": "F"}}'::jsonb))
Rows Removed by Index Recheck: 662295
Heap Blocks: exact=49139 lossy=99199
 -> **BitmapOr** (cost=415.90..415.90 rows=40000 width=0)
 (actual time=110.176..110.176 rows=0 loops=1)
 -> **Bitmap Index Scan on personal_gin_idx** (cost=0.00..198.00 rows=20000 width=0)
 (actual time=4.498..4.498 rows=27294 loops=1)
 Index Cond: (personal @> '{"personal": {"age": 26}}'::jsonb)
 -> **Bitmap Index Scan on personal_gin_idx** (cost=0.00..198.00 rows=20000 width=0)
 (actual time=105.675..105.675 rows=999735 loops=1)
 Index Cond: (personal @> '{"personal": {"gender": "F"}}'::jsonb)
 Planning Time: 0.104 ms
 Execution Time: 3253.896 ms

Disjunction Query (OR)

1623.48

1886.18

Bitmap Heap Scan on customers (cost=8778.50..69042.88 rows=1019092 width=16)
 (actual time=64.091..626.528 rows=1013460 loops=1)
Recheck Cond: ((age = 26) OR ((gender)::text = 'F'::text))
Heap Blocks: exact=44978
 -> **Bitmap Index Scan on age_above_twenty_six_or_gender_female_idx**
 (cost=0.00..8523.72 rows=1019092 width=0)
 (actual time=57.790..57.790 rows=1013460 loops=1)
 Planning Time: 0.157 ms
 Execution Time: 650.859 ms

| | Relational DB with B-TREE | JSONB DB with GIN |
|--|------------------------------|---------------------------|
| | 3462,072 | 19055,729 |
| | 13 MB | 62 MB |
| | 0.37 | 0.36 |
| | | 329.18 |
| | | 5418.65 |
| | 36.01 | 53.36 |
| | 1480.39 | 4221.31 |
| | 343076,445 (05:43,076) | 719036,651 (11:59,037) |
| | 26928,618 (00:26,929) | 81473,774 (01:21,474) |
| | 13056,619 | 55807,895 |

Fazit

- JSONB Datenstruktur wirkt sich, unabhängig von der Tabellengröße, negative auf die Laufzeit in allen Queries aus.
- GIN Index benötigt hat deutlich längere Erstellzeit und benötigt deutlich mehr Speicherplatz
- INSERT, UPDATE und DELETE Operationen, unter der Verwendung vom GIN Index, benötigen deutlich Länger
- Sowohl B-Tree als auch GIN-Index wurden nicht verwendet beim Range Query
- Ab 500K Tupels verschlechtert der GIN-Index die Laufzeit beim Disjunction Query an der JSONB Tabelle