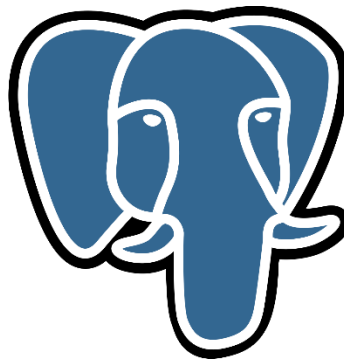


Übersicht über Datenbank-Index-Typen in bekannten Datenbanksystemen mit PostgreSQL-Benchmark



Seminar Datenbanksysteme

Master of Science in Engineering

Profil Computer Science

von

Vladimir Brazhnik

Supervisor: Prof. Stefan Keller

Frühlingssemester 2021

OST Campus Rapperswil

IMPRESSUM

Kontakt: vladimir.brazhnik@ost.ch

Dieses Dokument steht unter der Lizenz CC BY-SA 3.0 CH.

Es wurde erstellt mit Word und den Schriften Calibri, Century Gothic und Consolas

INHALTVERZEICHNIS

1	Einleitung.....	5
1.1	Überblick.....	5
1.2	Indexe.....	6
1.3	Eigenschaften.....	6
1.4	Query-Arten	7
1.4.1	Point Query.....	7
1.4.2	Multipoint Query.....	7
1.4.3	Range Query	7
1.4.4	Compound Query	7
1.5	Index Anwendungen	8
1.6	Index-Kategorien.....	9
1.6.1	Bitmap	9
1.6.2	Hash	9
1.6.3	ISAM	9
1.6.4	Search Tree	9
1.6.5	Inverted Index	9
1.6.6	Block Range Index.....	10
1.6.7	Learned Index	10
1.6.8	Main Memory Index	10
2	Index Implementationen in PostgreSQL	11
2.1	Vorinstallierte Indexe.....	11
2.1.1	B-Tree Index	11
2.1.2	BRIN Index	13
2.1.3	Hash Index	17
2.1.4	GiST Index.....	18
2.1.5	SP-GiST Index.....	20
2.1.6	GIN Index	21
2.2	Zusätzlich beigetragene Indexe.....	23
2.2.1	Bloom Index.....	23
2.3	Externe / experimentelle Indexe.....	25
2.3.1	RUM Index	25
2.3.2	Hippo Index	26
2.3.3	VODKA Index	27
3	Tabellarischer Vergleich aktueller Indextypen	28

4 Benchmark zwischen relationalen und mit JSONB-Objekten verschachtelten Datenbanken.. 30

4.1	Ziel.....	30
4.2	Vorgehensweise.....	30
4.3	Diskussion und Ausblick.....	31
4.4	Nachtrag.....	33

5 Literatur..... Fehler! Textmarke nicht definiert.

APPENDIX Benchmark 37

REQUIREMENTS.....	37
INSTALLATION	37
CONFIGURATION AND USAGE.....	37
DIRECTORIES AND FILES	39
DOCUMENTATION.....	40
Explaining of the important Files and Folders	40
QUERIES.....	41
CREATION OF THE JSONB TABLE CUSTOMERS_JSONB.....	48
INDEX CREATION.....	50
INSERT, DELETE AND UPDATE OPERATIONS	51
RESULTS OF THE BENCHMARK	53
50.000 RECORDS.....	53
100.000 RECORDS.....	54
500.000 RECORDS.....	55
1.000.000 RECORDS.....	56
2.000.000 RECORDS.....	57
INDEX BUILD TIME AND INDEX SIZE	58

1 EINLEITUNG

1.1 Überblick

Dieses Paper verfolgt das Ziel, den Lesern und Leserinnen einen Überblick über die verschiedenen Indextypen in Datenbanken und speziell in PostgreSQL zu verschaffen. Das hierbei geschaffene Verständnis für die unterschiedlichen Indexe und deren Anwendungsfälle soll die Auswahl eines passenden Indexes für die entsprechenden Szenarios erleichtern. Hierzu wird zunächst in dem ersten Kapitel „Einleitung“ ein allgemeiner Einstieg zu den Indexen gegeben, indem die wichtigsten Eigenschaften, Query-Arten, Anwendungen und Kategorien der Indexe vorgestellt werden. Danach werden im zweiten Kapitel „Index Implementationen in PostgreSQL“ die relevantesten Index Implementationen in PostgreSQL mit ihren Vor- und Nachteilen, sowie deren relevantesten Eigenschaften beschrieben. Hierbei werden bekannte PostgreSQL Indexe wie der B-Tree, Hash, GIN, GiST, SP-GiST, BRIN und Bloom Index beschrieben, sowie auch experimentelle Indexe wie der Hippo, RUM und VODKA Index erläutert. Anschliessen werden die zuvor vorgestellten, bekannten Indexe in einer Tabelle im dritten Kapitel „Tabellarischer Vergleich aktueller Indextypen“ gegenübergestellt und hinsichtlich ihrer Anwendungsfelder, Erstellungzeit, Festplattengrösse und der unterstützen Abfragen mit relativen Aussagen verglichen. Das letzte Hauptkapitel des Papers zeigt einen Benchmark zwischen relationalen und mit JSONB-Objekten erstellten Datenbanken. Hierbei soll geklärt werden, ob es Nachteile bei der Schachtelung von Attributen in einem JSON Objekt gibt und wie diese sich auf die Transaktionen und Abfragen auswirken. Dieses Kapitel wurde in Englisch verfasst, da der original Text auch als README.md in dem GitHub Repository hinterlegt ist.

1.2 Indexe

Indexe sind Datenstrukturen, welche auf eine bestehende Tabelle aufgebaut und dieser zugewiesen werden. Folglich benötigen Indexe immer einen zusätzlichen Overhead bei der Erstellungspflege und bei der Speicherung. Indem die Indexe die Tabellen scannen und analysieren, können sie „Shortcuts“ zu den gewollten Werten erstellen, wodurch beispielsweise ein direkter Zugriff auf den Ort möglich ist, an dem der oder die gewünschten Werte gespeichert sind.

Dadurch entfällt die Notwendigkeit, die gesamte Tabelle sequenziell durchsuchen zu müssen und es kann eine deutlich effizientere und somit schnellere Suche ermöglichen werden, bei der nur wenige oder gar keine zusätzlichen Daten von der Disk gelesen werden müssen. Eine Analogie aus der physischen Welt, wäre das Index-Verzeichnis eines grossen Notizbuches, welches an der Seite mit dem Alphabet beschriftet ist. Würde beispielsweise die Seite mit dem „Zoom Links“ gesucht werden, so müsste nicht das gesamte Buch von A bis Z durchsucht werden, sondern es könnte direkt beim Buchstabenindex Z angefangen werden [1, S. 601-602, Chapter 17 Indexing Structures for Files and Physical Database Design].

1.3 Eigenschaften

Vorteile:

- Indexe ermöglichen der Datenbank weniger Daten von der Festplatte lesen zu müssen
- Bestimmte Indexe ermöglichen das Abfragen und effiziente Lesen von Datenblöcken mit naheliegenden Daten in einem bestimmten Bereich
- Bestimmte Indexe sind bereits sortiert, sodass die Suche für sortierte Daten beschleunigt werden kann
- Der Query Planner von PostgreSQL vergleicht automatisch die Suche mit und ohne den Index und wählt dann die effizienteste Methode aus. Ob die Suche mit einem Index effizienter ist, hängt oft von der Grösse der indizierten Tabelle ab. Ist beispielsweise die Tabelle klein genug, wird der Index nicht verwendet, da zunächst die Tabelle des Indexes gelesen werden müsste und erst danach die indizierte Tabelle durchsucht werden kann. Hierbei würde die Verwendung der Sucht mit dem Index den Zeitaufwand erhöhen, weshalb der Query Planner den Index nicht verwendet.

Nachteile:

- Indexe sind eine zusätzliche Datenstruktur, welche zusätzlichen Festplattenspeicher benötigt
- Indexe müssen synchron zu den Daten gehalten werden, daher werden Schreiboperationen wie INSERT, UPDATE und DELETE häufig verlangsamt

Quelle: [2]

1.4 Query-Arten

Im Folgenden werden die Query-Arten beschrieben, welche für den Vergleich der Indexe in der Tabelle verwendet werden. Weitere wichtige Query-Arten, welche hier nicht beschrieben werden, sind: Prefix Match Queries, Extremal Queries, Ordering Queries, Grouping Queries und Join Queries.

1.4.1 Point Query

Der Point Query verwendet den Equal-Operator „=“ und kann keine oder nur eine einzige Zeile zurückgeben. Ein Beispiel hierfür wäre:

```
SELECT name FROM Employee WHERE ID = 8478
```

1.4.2 Multipoint Query

Der Multipoint Query verwendet genauso wie der Point Query den Equal-Operator „=“, kann aber im Gegensatz zum Point Query keine, eine oder auch mehrere Zeilen zurückgeben, abhängig von der Equality Bedingung. Ein Beispiel hierfür wäre:

```
SELECT name FROM Employee WHERE department = 'IT'
```

1.4.3 Range Query

Der Range Query arbeitet mit Operatoren wie >, >=, <, <=, != und BETWEEN und ermöglicht eine Bereichsabfrage, in welcher typischerweise mehrere Zeilen zurückgegeben werden, welche sich in dem angegebenen Interval befinden. Ein Beispiel hierfür wäre:

```
SELECT name FROM Employee WHERE salary >= 155000
```

1.4.4 Compound Query

Der Compound Query nimmt mehrere Queries gleichzeitig entgegen, wobei als Resultat entweder die Konjunktion oder die Disjunktion der Ergebnisse aus der Ergebnismenge genommen wird.

Beim Conjunction Query (AND) enthält die Abfrage mehrere Child Queries und gibt als Resultate nur die Zeilen und Werte zurück, welche alle Child Queries erfüllen. Ein Beispiel hierfür wäre:

```
SELECT name FROM Employee WHERE department = 'IT' AND gender = 'male'
```

Beim Disjunction Query (OR) enthält die Abfrage auch mehrere Child Queries, bei denen jedoch die Resultate eine konfigurierbare Mindestanzahl der Child Queries erfüllen muss. Standardgemäss liegt dieses Minimum bei eins. Ein Beispiel hierfür wäre:

```
SELECT name FROM Employee WHERE department = 'IT' OR gender = 'male'
```

Quelle für Compound Queries: [3]

Quelle für die restlichen Queries: [4]

1.5 Index Anwendungen

Primary Index:

Primary Indexe verweisen auf ein Attribut, nach welchem die Tupel einer Tabelle sortiert gespeichert werden. Sie besitzen einen Primärschlüssel und sind immer „unique“ [1, S. 603, Chapter 17.1.1 Primary Indexes].

Clustered Index:

Der Clustered Index ermöglicht es, die Daten einer Tabelle nach einem Daten-Attribute oder einem Index zu sortiert und physisch sequenziell zu speichern [1, S. 606, Chapter 17.1.2 Clustering Indexes].

Secondary Index:

Secondary Index ist ein zweiter Index, welcher ein Attribut einer Tabelle indiziert, für welches bereits ein Index definiert worden ist [1, S. 609, Chapter 17.1.3 Secondary Indexes].

Index on Expressions:

Diese Index Anwendung basiert auf Funktionen oder Ausdrücken und wird intern zur Laufzeit aus einer oder mehreren Spalten der Tabelle berechnet und aufgebaut. Dadurch kann ein schnellerer Zugriff auf Tabellen ermöglicht werden, welche auf Berechnungen von Ergebnissen basieren [5].

Partial Index:

Partial Index ermöglicht mit Hilfe einer Conditional-Expression die Erstellung eines Indexes für nur eine Teilmenge eines Datensatzes. Werden Partial Indexe für Tabellen verwendet, bei denen die Indizierung von gleichen Werten verhindert werden soll, so sind Partial Indexen kleiner und performanter als gewöhnliche Indexen, welche für den gesamten Datensatz erstellt wurden [6].

Concurrently created Indexes:

Indexe müssen sich neben der Verarbeitung von Abfragen, auch mit Transaktionen und Schreibzugriffen befassen. Beispielsweise sperren normalerweise die Indexe in PostgreSQL, während des gesamten Indexaufbaus, die zu indizierenden Tabellen gegen Schreibzugriffe, sodass der Indexaufbau ungestört in einem einzigen Scan durchgeführt werden kann. Andere Transaktionen können während dieser Zeit die Daten lesen (read), jedoch keine Zeilen in die indizierte Tabelle einzufügen (insert), verändern (update) oder aus der Tabelle zu löschen (delete). Folglich kann die Erstellung von Indexen den regulären Betrieb einer für die Dauer des Erstellens stören. Die concurrently created Indexe in PostgreSQL bieten eine Lösung für dieses Problem, indem Sie die Tabelle zwei Mal scannen und zusätzlich auf die Beendigung von Transaktionen, welche den Index potenziell verändern könnten, warten. Dadurch können concurrently created Indexe deutlich besser in eine Produktionsumgebung eingefügt werden, benötigen aber für deren Aufbau deutlich mehr Aufwand (CPU-Leistung) und Zeit [7] & [1, S. 805-806, 21.6 Using Locks for Concurrency Control in Indexes].

1.6 Index-Kategorien

Im Folgenden werden die hauptsächlich verwendeten, abstrakten Datenstrukturen aufgelistet.

1.6.1 *Bitmap*

Die Kategorie der Bitmap Indexe ist eine populäre Datenstruktur, welche in PostgreSQL nicht als eine on-disk Version mit Bitmap-Index Zugriffsmethoden angeboten wird, sondern nur zu internen Zwecken automatisch verwendet wird und somit ad-hoc bei beispielsweise JOINS verbaut wird. Jedoch wurde ein experimenteller on-disk Bitmap-Index entwickelt, welcher es erlaubt, den Index in seiner Performance zu testen und ihn mit anderen zu vergleichen [8]. Der Bitmap-Index erstellt einen Index für eine oder mehrere Spalten und jeder der individuellen Werte oder Wertebereiche in dieser Spalte [1, S. 634-635, Chapter 17.5.2 Bitmap Indexes].

1.6.2 *Hash*

Die Hash Kategorie bietet Zugriffsstrukturen, welche den von Indexen ähneln, aber auf Hashing, Hash Mapping und den Hash Tables basieren. In PostgreSQL gehört der Hash Index dieser Kategorie an [1, S. 633-634, Chapter 17.5.1 Hash Indexes].

1.6.3 *ISAM*

Die ISAM Kategorie steht für sequential access method und ist eine beliebte Zugriffsmethode auf Datensätze einer geordneten/sortierten Datei, welche sowohl sequenziellen als auch wahlfreien index-basierten Zugriff zulässt. Dieses Indexierungsschema basiert auf der Idee der multilevel Indexe und wird beispielsweise in MySQL bei dem auf ISAM aufbauendem Datenbanksystem MyISAM verwendet [1, S. 601, Chapter 17 Indexing Structures for Files and Physical Database Design] & [9].

1.6.4 *Search Tree*

Die Indexe in der Search Tree Kategorie entsprechen einem speziellen Baumtyp, welcher nur einen bestimmten Wert aus dem Feld des gesuchten Wertes benötigt, um die Suche nach dem Wert auszuführen. Dabei leiten die Indexfeldwerte die Suche weiter zum nächsten Knoten des bestimmten Subtrees, bis der Datenblock erreicht wird, indem der gesuchte Wert liegt. In PostgreSQL gehören der B-Tree Index, der PG GIST Index sowie der SP-GIST dieser Kategorie an [1, S. 618, Chapter 17.3.1 Search Trees and B-Trees].

1.6.5 *Inverted Index*

Die Kategorie der Inverted Indexes zeichnet sich dadurch aus, dass sie eine Abbildung von Wörtern auf deren Positionen in einer Menge von Dokumenten erstellt. Diese Dokumente sind die posting list, welche für jeden Wert eines composite values ein Indexeintrag speichern. In PostgreSQL gehört der PG GIN Index zu dieser Kategorie [10].

1.6.6 Block Range Index

Die Kategorie der Block Range Indexe verfolgt das Ziel, die Performance bei sehr grossen Tabellen ohne Partition zu verbessern. In PostgreSQL gehört der PG BRIN Index zu dieser Kategorie [11].

1.6.7 Learned Index

Die Learned Index Kategorie umfasst alle Indexe, welche mit Hilfe von Machinelearning-Techniken entwickelt werden und ein explizites Model der Daten erstellen, um eine effektive Indizierung zu ermöglichen. Dem Paper „Benchmarking Learned Indexes“ zu folge, benötigen Learned Indexes häufig in der Erstellung deutlich mehr Zeit als insert-optimierte traditionelle Indexe, wie der B-Tree, jedoch sind sie aber in den meisten Fällen performanter und platzsparender als die traditionellen Indexe [12].

1.6.8 Main Memory Index

Der Main Memory Index wird häufig als eine Baumstruktur implementiert (z.B. T-Tree) und speichert die Indexstruktur im Hauptspeicher, sodass keine Festplattenzugriffe mehr notwendig sind. Im Gegensatz zu den Festplatten orientierten Indexen, verfolgt der Main Memory Index somit nicht das Ziel, die Plattenzugriffe und den Plattenspeicherplatz zu minimieren, sondern versucht mit Hilfe des Hauptspeichers die gesamte Berechnungszeit bei gleichzeitiger Verwendung von möglichst wenig Speicher zu reduzieren. Dadurch dass Relationen speicherresident sind, speichert der Main-Memory Index anstatt der Attributwerte der Tabelle, nur die Zeiger, welche auf die Stelle, an der die Tupel gespeichert wurden, verweisen und diese bei Bedarf extrahieren können. Somit hängt der Speicherplatzbedarf vom verwendeten Speicherplatz der Zeiger, der Darstellung von Schlüsseln in Indexknoten und der durchschnittlichen Belegung der Knoten ab. Eine beispielhafte Implementierung eines Main-Memory Indexes ist der T-Tree, welcher eine ordnungserhaltende Baumstruktur ist, die speziell für die Verwendung im Hauptspeicher entwickelt wurde [13].

2 INDEX IMPLEMENTATIONEN IN POSTGRESQL

Im folgenden werden die relevantesten Index Implementationen in PostgreSQL mit ihren Vor- und Nachteilen, sowie deren relevantesten Eigenschaften beschrieben. Hierzu werden die Indexe gegliedert in:

1. Vorinstallierte Indexe
2. Zusätzlich beigetragene Indexe
3. Externe / experimentelle Indexe

2.1 Vorinstallierte Indexe

2.1.1 B-Tree Index

Der Balanced-Tree Index gehört zu der Kategorie der Baumindexe und ist in vielen Datenbanksystemen, wie beispielsweise PostgreSQL, Oracle und MySQL, ein bereits vorinstallierter Standard-Index, da dieser sich für die häufigsten, allgemeinen Situationen und Basis-Datentypen eignet. Dadurch dass der B-Tree Index eine selbst-balancierten Baumstruktur besitzt und sortierte Daten verwaltet, können search, insert und delete Operationen für alle Daten im Baum in logarithmischer Zeit ausgeführt werden. Das ist darin begründet, dass jeder Knoten zwei oder mehr Kindesnoten und mehrere Schlüssel besitzen kann. Dadurch haben alle Blätter die gleiche Distanz zur Wurzel und der Baum bleibt balanciert. Somit können bei der Suche nach einem Schlüsselwert, alle Einträge, angefangen bei dem Wurzelknoten, verglichen werden, bis ein Eintrag gefunden wird, welcher grösser oder gleich dem gesuchten Wert ist. Wird ein Eintrag gefunden, der diese Bedingung erfüllt, so wird dessen Verweis zu dem entsprechenden Zweigknoten gefolgt. Im Zweigknoten wird dann der Prozess wiederholt, bis der gesuchte Wert gefunden oder ein Blattknoten erreicht wird. In dem Blattknoten sind die Daten der indexierten Werte als Schlüssel und die Referenzen auf die jeweiligen Daten als Values hinterlegt. Durch diese Datenstruktur bietet der B-Tree Equality, Comparison und Pattern Matching Operatoren wie <, <=, =, >=, BETWEEN, IN, IS NULL, IS NOT NULL, LIKE, ILIKE für Point-, Multipoint-, Range- und Compound Queries an Basis-Datentypen, welche diese unterstützen [14].

Vorteile:

- Geeignet für die häufigsten, allgemeinen Situationen, da B-Tree über alles gesehen im schlimmsten Fall immer noch über die besten Eigenschaften verfügt
- Führt insert, search und delete Operationen in logarithmische Zeit aus ($O(\log(n))$)
- Daten können bereits sortiert ausgegeben werden, ohne diese zuvor sortieren zu müssen
- Laufzeit bleibt im besten und im schlechtesten Fall gleich
- Unterstützt eine Vielzahl an unterschiedlichen Queries und Operatoren für Basis-Datentypen

Nachteile:

- Modifizierung von Daten mit der Laufzeit $O(\log(n))$ kann von anderen Indexen wie beispielsweise den Hash Index mit konstanter Laufzeit übertroffen werden
- Kann für komplexere Datentypen (z.B. JSONB, Text-Suche, Geometrie Daten) von anderen Indexen übertroffen werden

Quellen: [14] & [1, S. 618, Chapter 17.3.1 Search Trees and B-Trees]

2.1.2 BRIN Index

Der Block Range Index ist eine eigene Kategorie und ist am effizientesten, wenn die Datenmenge sequenziell aneinandergereiht und sehr gross ist (über eine Millionen Einträge). Entscheidend ist, dass die Daten geordnet sein müssen, da sonst der BRIN Index nicht richtig funktionieren kann. Bei geordneten Daten aber, kann der BRIN Index den Speicherplatzverbrauch im Vergleich zum B-Tree signifikant optimieren. Zusätzlich kann der BRIN Index unter der Berücksichtigung einiger Faktoren auch die Performance im Vergleich zum B-Tree optimieren. Ein nützlicher Use-Case für den Einsatz vom BRIN Index wäre beispielsweise das Abfragen von mehreren Verkaufsbestellungen eines Geschäftes, mittels deren Zeitstempel. Der BRIN Index arbeitet mit Blöcken von Daten, wobei von diesen nur eine Referenz auf einem bestimmten Bereich als Index gespeichert wird. Somit kann von jeder Block Tabelle, welche standardgemäss die Block Range 128 hat und eine bestimmte, definierbare Anzahl an Einträgen enthalten kann, der Mindest- und Maximalwert als BRIN Index hinterlegt werden. Sind in einem Block beispielsweise 128 Einträge enthalten, so zeigt ein BRIN Index auf 16.384 Einträge ($128 * 128$).

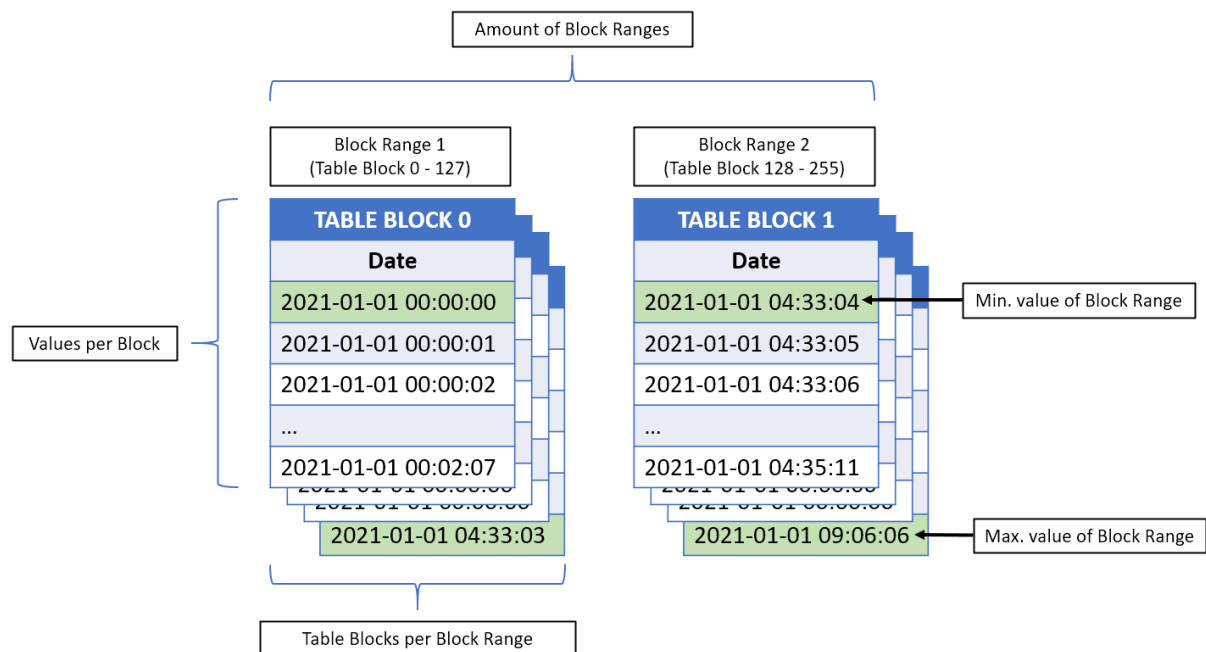


Abbildung 1: Beispiel Struktur einer BRIN Indexes

Eigene Abbildung

Dabei muss beachtet werden, dass der BRIN Index im Gegensatz zum B-Tree nicht direkt auf einen Eintrag verweist, sondern nur auf den Bereich, in dem dieser Eintrag liegt. Somit müssen zusätzlich noch alle Block Table gelesen werden, um die gesuchten Einträge zu finden. Aus diesem Grund ist der BRIN Index deutlich performanter, je mehr Einträge die SELECT Abfrage zurückgibt und je grösser die Tabelle der Einträge ist. Ausserdem ist der BRIN Index ein verlustbehafteter (lossy) Index, da er falsche Treffer produziert und daher mehr Werte zurückgibt, welche zusätzlich herausgefiltert werden müssen. Diese zusätzliche Filterung, genauso wie das Durchlesen aller Block Table Einträge wirkt sich negativ auf die Performance aus, sodass die Hauptstärke des BRIN Indexes die Einsparung des Speicherplatzes ist.

Die Abbildung 2 zeigt die Relation von der Tabellengröße, der Ergebnismenge der SELECT-Abfragen und der resultierenden Performance. Hierbei wurden als Daten jede Sekunde zufällige Texte mit Zeitstempel erstellt. Die Tabellengröße variiert zwischen 1.000 bis zu 1.000.000.000 Einträgen und die Ergebnismenge der SELECT Abfrage variiert von einem single column query mit nur einem Ergebnis, bis hin zu den gesammelten Zeitstempeln während einer Stunde (3600 Ergebnis), einem Tag (86400 Ergebnis) und einem Monat (2678400 Ergebnis). Zu beobachten ist, dass der Query Planner den BRIN Index erst ab 100.000 Einträgen berücksichtigt und der BRIN Index erst ab 1.000.000 Einträge und einer Ergebnismenge von 86400 performanter ist. Ausserdem ist die Performance bei single column queries im Vergleich zum B-Tree deutlich schlechter.

Table Records	1 record		1 Hour		1 Day		1 Month	
	1		3600		86400		2678400	
	BTREE	BRIN	BTREE	BRIN	BTREE	BRIN	BTREE	BRIN
1,000	.7 ms	-	-	-	-	-	-	-
10,000	.7 ms	-	7 ms	-	-	-	-	-
100,000	.7 ms	7 ms	7 ms	9 ms	40ms	-	-	-
1,000,000	.7 ms	8 ms	7 ms	12 ms	42ms	39 ms	-	-
10,000,000	7ms	8 ms	7 ms	12 ms	42ms	40 ms	980ms	825 ms
100,000,000	1.6 ms	10 ms	7 ms	14 ms	43ms	41 ms	990ms	850 ms
1,000,000,000	2.5 ms	25 ms	14ms	25 ms	43ms	55 ms	1000ms	833 ms

Abbildung 2: B-Tree vs. BRIN select speeds at various table size

Quelle: <https://blog.crunchydata.com/blog/avoiding-the-pitfalls-of-brin-indexes-in-postgres>

Zusätzlich wurde die Performance von BRIN Index untersucht, nachdem 20 % der Einträge gelöscht, wiederhergestellt und erneut sortiert wurden. Es stellte sich heraus, dass sich die Performance danach deutlich verschlechterte.

Records	1 record		1 Hour		1 Day		1 Month	
	Before	After	Before	After	Before	After	Before	After
1,000,000	8 ms	35 ms	12 ms	70 ms	39 ms	95 ms	-	-
10,000,000	8 ms	680 ms	12 ms	720 ms	40 ms	750 ms	825 ms	1.2 s
100,000,000	10 ms	3 - 8 s	14 ms	3 - 8 s	41 ms	3 - 8 s	850 ms	3- 8 s

Abbildung 3: BRIN select speed after ~20% delete and ~20% insert

Quelle: <https://blog.crunchydata.com/blog/avoiding-the-pitfalls-of-brin-indexes-in-postgres>

Der Speicherbedarf wird hingegen im Vergleich zum B-Tree mit zunehmender Tabellengröße signifikant geringer. Bereits bei 1.000 Einträgen beträgt der Speicherbedarf nur noch 10% von dem Speicherbedarf, welchen der B-Tree benötigen würde. Bei 1.000.000.000 Einträgen sind es nur noch 0.008%.

Records	Table Size	BTREE Size	BRIN Size
1,000	88kB	40kB	4kB
10,000	544kB	240kB	48kB
100,000	5.1MB	2.2MB	48kB
1,000,000	50MB	21MB	48kB
10,000,000	498MB	214MB	56kB
100,000,000	4.9GB	2.1GB	208kB
1,000,000,000	49GB	21GB	1.7MB

Abbildung 4: B-Tree vs. BRIN index at various table sizes

Quelle: <https://blog.crunchydata.com/blog/avoiding-the-pitfalls-of-brin-indexes-in-postgres>

Somit eignet sich der BRIN Index am besten für Tabellen, die eine sehr grosse, sequenziell aufsteigende und natürlich geordnete Datenmenge besitzen, bei der hauptsächlich INSERT und SELECT Queries ausgeführt werden, wobei die SELECT Queries nach einer grossen Ergebnismenge (eine grossen Range) von Einträgen suchen.

Entscheidend sind somit folgende Faktoren:

- Tabellengröße
- Benötigte Ergebnismenge bei SELECT Queries
- Verwendete Query Operationen (am besten INSERT-only)
- Gruppierung / Ordnung der Daten

Der BRIN Index ist ein von PostgreSQL vorinstallierter Index und dessen build-in Operator Klassen sind zu finden unter der PostgreSQL Dokumentation im Kapitel [67.2 Built-in Operator Classes](#).

Vorteile:

- Benötigt deutlich weniger Speicherplatz als B-Tree Index und ist somit auch deutlich schneller in der Erstellung und einfacher in der Wartung für grössere Datenmengen
- Hat unter Berücksichtigung der oben erwähnten Faktoren eine bessere Performance als B-Tree
- Ermöglicht die Verwendung von Indexen auf sehr grosse Tabellen ohne die Notwendigkeit von horizontaler Partitionierung

Nachteile:

- Erfordert physisch geordnete, sequenziell aneinandergereihte Daten
- Ist nur sinnvoll bei sehr grossen Datenmengen und SELECT Queries, welche eine grossen Ergebnismenge zurückgeben
- Verliert an Performance bei häufiger Anwendung von DELETE und UPDATE Queries
- Ist ein verlustbehafteter Index

Quelle: [15]

2.1.3 Hash Index

Der Hash Index gehört zu der Kategorie der Hash Indexe und ist ein bereits vorinstallierter Index in Postgres, welcher sich durch dessen Key-Value Lookup Tabelle und der Hashfunktion auszeichnet. In dieser Tabelle sind als Keys die Hashes der indexierten Werte und als Values die Referenzen zu den gesuchten Datenwerten hinterlegt. Dadurch, dass aus dem indexierten Wert ein Hash gewonnen wird, welcher dann zu einem Index für das Hash Array umgewandelt wird, kann im besten Fall direkt auf den gesuchten Wert mit einer Laufzeit von $O(1)$ zugegriffen werden. Ist die Hash-Funktion hingegen schlecht und sorgt für viele Kollisionen, so beträgt die Laufzeit im schlechtesten Fall $O(n)$. Hierbei wären alle indizierten Werte in einer verketteten Liste hinterlegt. Da dieser Fall aber sehr unwahrscheinlich ist und die Laufzeit vom Hash Index häufig gegen $O(1)$ verläuft, kann der Hash Index bei grossen Datensammlungen (> 200000) den B-Tree Index hinsichtlich der Performance knapp übertreffen [16].

Ausserdem wird die Grösse des Hash Indexes im Gegensatz zum B-Tree nicht durch die Grösse des indizierten Wertes beeinflusst. Somit kann der Hash Index bei bestimmten, grösseren Datentypen weniger Speicher benötigen als der B-Tree Index. Jedoch hat der Hash Index im Vergleich zum B-Tree deutlich mehr Limitationen. Die wichtigste ist, dass der Hash Index nur über den Equality-Operator verfügt, wodurch die Query Abfragen deutlich eingeschränkter sind. Ausserdem kann der Hash Index nicht verwendet werden für range lookups, Zur Erzwingung von unique constraints, zur Erstellung von Indexe auf mehrere Spalten oder sortierte Indexe, zum Clustern einer Tabelle oder zur Erfüllung von ORDER BY Queries [17].

Vorteile:

- Für Equality Operationen bei grossen Datenmengen sehr performant und sogar performanter als B-Tree, da im besten Fall ein direkter Zugriff ($O(1)$) auf den gesuchten Wert möglich ist
- Grösse vom Hash Index wird nicht beeinflusst durch die Datentypengrösse, wodurch Hash Index bei grossen Datentypen wenig Speicher benötigt und im Vergleich zum B-Tree speicherplatzsparender ist

Nachteile:

- Wenig Flexibilität durch viele Limitationen (siehe oben)
- Beschränkung auf Equality-Operator
- Supportet keine Multicolumn Indexe

Weitere Quelle: [1, S. 633, Chapter 17.5.1 Hash Indexes]

2.1.4 GiST Index

Der Generalized Search Tree Index gehört zu der Kategorie der Baumindexe und ist wie der B-Tree eine ausgeglichene Baumstruktur, welche `<key, pointer>` Paare besitzt. Im Gegensatz zu dem B-Tree verfügt der GiST Index über den entscheidenden Vorteil, dass dieser sämtliche Funktionalitäten aller balancierter Baumstrukturen anbietet und gleichzeitig die Erweiterbarkeit der Daten und zusätzlich noch die Erweiterbarkeit der Queries ermöglicht. Somit ist der GiST Index eine Infrastruktur, in welcher mehrere unterschiedliche Indexstrategien implementiert werden können. Unterstützte Strategien zum beispielsweise populären Baumstrukturen, wie B+-Trees, R-Trees, SR-trees, RD-trees, hB-Trees, TV-Trees, Ch-Trees, partial sum trees, ranked B+-trees. Daher ist der GiST Index beispielsweise im Gegensatz zum B-Tree Index nicht beschränkt auf die Comparison Operatoren, welche bei moderneren Datentypen wie Geometrie-Daten, Textdokumenten oder Bildern häufig nicht genügen. Durch die mögliche Operatoren Vielfalt und die erweiterbare Baumstruktur ist der GiST Index somit genauso wie der GIN Index für Full-Text-Search, Nearest Neighbor Search, sowie das Arbeiten mit raumbezogenen Daten gut geeignet und kann diese deutlich verschnellern. Entsprechend der Generalisierung können die Schlüssel der `<key, pointer>` Paare vom GiST Indexes den indexierten Datenelementen angepasst werden.

Um diese Anpassung zu ermöglichen, müssen sowohl die Daten, welche in den Schlüsseln dargestellt werden, als auch die Methoden der Schlüssel zum Einfügen, Löschen und Suchen definiert werden. Zusätzlich können noch weitere Methoden definiert werden, welche die Performance verbessern können.

Vor allem Domänen Experten können vom hohen Anpassungsgrad der Datenbanksysteme an die Indizierung der Inhalte profitieren. Beispielsweise könnten in einer Datenbank von Bildern domänenspezifische Queries, wie die Suche nach Bildern mit Pferden oder Überbelichtung, definiert werden [18].

Dennoch wird von den Entwicklern hervorgehoben, dass auch wenn jeder Workload indiziert werden kann, nicht jeder Workload automatisch effizient indiziert wird. Hierfür haben die Entwickler die Theory of Indexability entwickelt, in der die Grenzen der Indizierungseffizienz abgebildet und die Indizierungsprobleme beschrieben werden [18].

Vergleicht man den GiST Index mit dem GIN Index, so muss beachtet werden, dass der GiST Index ein verlustbehafteter Index ist, welcher falsche Treffer produziert und somit mehr Werte zurückgibt, welche dann in Nachhinein vom Query Planner ausgefiltert und entfernt werden müssen. Durch diesen zusätzlichen Schritt beim Auslesen von den Daten, ist der GiST Index im Vergleich zu dem GIN Index weniger performant. Ausserdem kann beim GiST Index im Gegensatz zum GIN Index die Performance nicht mittels der Erhöhung des Wartungsarbeitsspeicher (`maintenance_work_memory`) verbessert werden. Der wesentliche Vorteil von GiST Index zum GIN Index ist, dass der GiST Index mehr Grössenbeschränkungen besitzt, als der GIN Index und somit im Vergleich zum GIN Index geringeren Festplatten-Footprint hat [19].

Der GiST Index ist ein PostgreSQL vorinstallierter Index und dessen build-in Operator Klassen sind zu finden unter der PostgreSQL Dokumentation im Kapitel [59.2. Built-in Operator Classes](#).

Die Implementierung sowie eine Erklärung der Datenstruktur sind zu finden unter der PostgreSQL Dokumentation im Kapitel [64.4. Implementation](#) und im Blog Post [Indexes in PostgreSQL – 5 \(GiST\)](#).

Vorteile:

- Bietet sämtliche Funktionalitäten aller balancierten Baumstrukturen an
- Ermöglicht die Erweiterbarkeit von Datentypen und Queries und die Entwicklung eigener Datentypen sowie deren Zugriffsmethoden
- Ist gut geeignet für Full-Text-Search, Nearest-Neighbor-Search und für das Arbeit mit Geodata
- Ermöglicht die Definierung von zusätzlichen Methoden, um die Performance zu steigern
- Hat im Vergleich zum GIN-Index einen geringeren Festplatten-Footprint

Nachteile:

- Ist ein verlustbehafteter Index, weshalb er im Vergleich zum GIN Index weniger performant ist
- Performance lässt sich nicht durch die Erhöhung des Wartungsarbeitsspeichers verbessern
- Nicht jede Indizierung ist automatisch effizient. Es müssen die Grenzen der Indizierungseffizienz und die Indizierungsprobleme in der Theory of Indexability beachtet werden

Quellen: [18, 19]

2.1.5 SP-GiST Index

Der platz-partitionierte GiST Index gehört genauso wie der B-Tree und der GiST Index zu der Kategorie der Baumindexe. Wie der GiST Index, verfolgt auch der SP-GiST Index das Ziel, mehrere populäre Baumstrukturen sowie die Entwicklung eigener Datentypen und deren eigenen Zugriffsmethoden zu ermöglichen. Der Unterschied hierbei ist, dass der SP-GiST ein partitionierter Index ist, welcher auf nicht-balancierte, plattenbasierte und partitionierte Suchbäume spezialisiert ist, bei welchen die Indexe die Suchräume rekursive in Partitionen unterteilen. Somit werden beispielsweise Datenstrukturen wie Quad-Trees, K-D-Trees und Radix-Trees unterstützt. Entsprechend können Suchvorgänge, welche auf Partitionierungsregel angepasst sind, mit Hilfe des SP-GiST Indexes schnell ausgeführt werden. Zusätzlich bietet der SP-GiST Index Performancevorteile bei vielen, sich überlappenden Objekten, wie es bei Geometrie-Daten häufig der Fall ist. Das Ziel von SP-GiST ist, die Knoten eines Suchbaums so auf den Plattenseiten abzubilden, dass eine Suche nur auf wenige Plattenseiten zugreifen muss, auch wenn diese viele Knoten durchläuft. Somit stehen die effiziente Plattenzuordnung und das Durchsuchen der Baumstruktur bei vor allem nicht-balancierten, sich überlappenden, grossen Daten im Vordergrund.

Der SP-GiST Index ist ein PostgreSQL vorinstallierter Index und dessen build-in Operator Klassen sind zu finden unter der PostgreSQL Dokumentation im Kapitel [65.2. Built-in Operator Classes](#).

Die Implementierung sowie eine Erklärung der Datenstruktur sind zu finden unter der PostgreSQL Dokumentation im Kapitel [64.4. Implementation](#) und im Blog Post [Indexes in PostgreSQL – 5 \(GiST\)](#).

Vorteile:

- Bietet sämtliche Funktionalitäten aller nicht-balancierten, plattenbasierten Baumstrukturen an
- Ermöglicht die Erweiterbarkeit von Datentypen und Queries
- Ist gut geeignet für die Indizierung von Geometrie-Daten wie GIS und zur Beschleunigung der Suche von vielen Datentypen, wie beispielsweise Telefon Routing, IP-Routing und String-Matching-Algorithmen, bei welchen die Suchräumen rekursiv in Partitionen unterteilt wurden
- Ist optimiert für auf Partitionsregeln angepasste Suchvorgänge, welche bei grossen, nicht-balancierten Datenmengen durchgeführt werden
- Bietet Performancevorteile bei sich überlappenden Daten

Nachteile:

- Verlustbehafteter Index, weshalb SP-GiST im Vergleich zum GIN Index weniger performant ist

Quellen: [20, 21]

2.1.6 GIN Index

Der generalized inverted Index gehört zu Kategorie der Inverted Indexes und bietet sich an, wenn die vom Index zu verarbeitenden Abfragen nach Elementwerten suchen, welche innerhalb zusammengesetzten Werten (sogenannten „composite values“) sind. Diese zusammengesetzten Werte zeichnen sich dadurch aus, dass diese in einer Spalte mehrere Werte speichern. Beispielhafte Datentypen hierfür wären: `hstore`, `array`, `jsonb`, `tsvector`, `range types`.

Ein geeigneter Use-Case für den Einsatz eines GIN Indexes wäre zum Beispiel eine Textsuche in einem Dokument, bei dem nach einem speziellen Wort gesucht wird [22].

Der GIN-Index speichert ein Set von Schlüssel des Indexes und posting list Paaren. Eine posting list ist ein Set von Zeilen mit IDs, in welchen die jeweiligen Schlüssel der composite values hinterlegt sind. Hierbei kann der gleiche Schlüssel eines composite values in mehreren posting lists vorkommen, da ein Item eines composite values mehrere Schlüssel haben kann [23]. Vereinfacht formuliert, ist der GIN-Index eine Art B-Tree, welcher Indexeinträge für jeden Wert des composite values enthält und somit ein wesentlich grösserer ist [22].

INDEX	
the	[1,3,4]
quick	[1]
brown	[1]
fox	[1]
jumps	[2]
over	[2]
lazy	[3,4]
dog	[3,4]
is	[5]
sleeping	[5]

TABLE	
1	the quick brown fox
2	jumps over
3	the lazy dog
4	because the lazy dog
5	is sleeping

Abbildung 5: Beispiel einer möglichen Struktur eines GIN-Indexes

Quelle: <https://towardsdatascience.com/how-gin-indices-can-make-your-postgres-queries-15x-faster-af7a195a3fc5>

Ein Vorteil vom GIN-Index ist, dass jeder Schlüssel von Index (z.B. „the“) nur einmal gespeichert werden muss. Dadurch ist der GIN Index sehr kompakt und am nützlichsten für Daten, bei denen die Schlüssel mehrfach vorkommen. Ausserdem ist der GIN Index verallgemeinert, da der GIN Index die spezifischen Operationen, welcher er beschleunigt, nicht kennen muss. Stattdessen können für bestimmte Datentypen eigene Strategien definiert und verwendet werden. Beispielsweise kann die `pg_trgm` Extension mit der Option `gin_trgm_ops` für den GIN-Index verwendet werden, sodass PostgreSQL über die ausgewählten Spalten mittels Trigram Indexe indizieren kann. Mittels solcher eigenen Strategie kann definiert werden, wie Schlüssel aus indizierten Elementen und Abfragebedingungen extrahiert werden und wie festgestellt werden soll, ob eine Zeile, die einige der Schlüsselwerte in einer Abfrage enthält, die Abfrage erfüllt. Dadurch ermöglicht der GIN-Index auch die Entwicklung von eigenen Datentypen und deren entsprechenden Zugriffsmethoden [23].

Der GIN Index ist ein vorinstallierter Index in PostgreSQL und dessen build-in Operator Klassen sind zu finden unter der PostgreSQL Dokumentation im Kapitel [66.2. Build-in Operator Classes](#).

Die Implementierung sowie eine Erklärung der Datenstruktur sind zu finden unter der PostgreSQL Dokumentation im Kapitel [66.4. Implementation](#)

Vorteile:

- Sehr schnell für Abfragen, welche nach Elementwerten innerhalb composite values suchen
- Optimal geeignet für Text-Suche (vor allem in Kombination mit der pg_trgm Extension)
- Kompaktere Speicherung von Daten, bei denen Schlüssel mehrfach vorkommen
- Ermöglicht Entwicklung und Nutzung von eigenen Datentypen und Zugriffsmethoden

Nachteile:

- Benötigt mehr Zeit und Speicherplatz zum Erstellen der Indexe als beispielweise B-Tree, da jeder individuelle Wert innerhalb des composite values induziert werden muss

Quellen: [22] & [23]

2.2 Zusätzlich beigetragene Indexe und Erweiterungen

2.2.1 Bloom Index

Der Bloom Index gehört zu Kategorie der Hash Indexe, da er mit Hash-Funktionen und Hashes arbeitet. Basierend auf dem Bloom-Filter ermöglicht der Bloom-Index spezielle Indexzugriffsmethoden, mit welchen effizient und platzsparend geprüft werden, ob ein Element in der Datenbank existiert oder ein Mitglied einer Menge ist. Hierbei wird auf die Datenstruktur vom Bloom-Filter zurückgegriffen, welche im Gegensatz zu den Baumstrukturen vom GiST Index oder vom B-Tree eine flache Struktur besitzt. Diese besteht aus einer Metaseite, gefolgt von regulären Seiten mit Indexzeilen. Jede dieser Indexzeilen enthält eine Signatur (0-Bit oder 1-Bit), welche angibt, ob der Wert in der Tabelle existieren könnte oder nicht existiert. Zusätzlich zu der Signatur ist ein Hashwert als Schlüssel hinterlegt, welcher auf die Tabellenzeile verweist, an welcher der gesuchte Wert hinterlegt ist. Der Filter selbst ist ein Array aus n Bits, welcher anfangs mit 0-Bits gefüllt ist. Jedes Element der Menge wird mittels k verschiedenen Hash-Funktionen auf die k Bits der Signatur abgebildet [24].

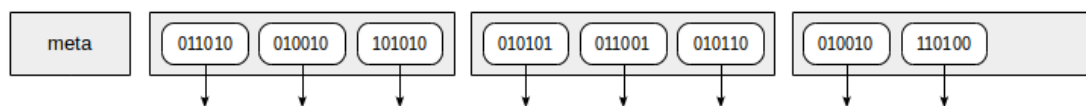


Abbildung 6: Beispiel der Struktur eines Bloom-Indexes

Quelle: <https://postgrespro.com/blog/pgsql/5967832>

Wenn das 0-Bit bei einem Wert gesetzt ist, existiert der gesuchte Wert nicht in der Datenbank und die Suche kann direkt abgebrochen werden. Dadurch müssen weitere zusätzliche Queries und zeitaufwendige Suchvorgänge nicht weiter fortgeführt werden. Wenn das 1-Bit gesetzt ist, könnte der gesuchte Wert in der Datenbank existieren, muss es aber nicht. Der Grund hierfür sind die möglichen Kollisionen der gehashten Indexe. Durch diese ist die Repräsentation verlustbehaftet, wodurch falsch-positive Ergebnisse zurückgegeben werden können. Um sicher zu stellen, dass das Element wirklich in der Menge ist, müssen zusätzlich die Ergebnisse aus der Indexsuche mit den tatsächlichen Attributwerten aus dem Heap-Entry überprüft werden. Die Wahrscheinlichkeit für Kollisionen und falsch-positive Ergebnisse kann reduziert werden, indem die Tabelle des Bloom-Filters vergrößert wird, dadurch werden aber auch die Indexe größer und die Suchen langsamer. Am sinnvollsten ist der Bloom-Index, wenn die indizierte Tabelle viele Attribute hat und die Abfragen beliebige Kombinationen von diesen Attributen auf ihre Existenz in der Tabelle testen. Dieser Use-Case könnte auch performanter mit der Hilfe des B-Trees gelöst werden. Jedoch wären mehrere B-Tree Indexe notwendig, um alle möglichen Abfragen zu unterstützen. Hingegen würde ein einziger Bloom-Index genügen, um alle möglichen Abfragen bezüglich der Existenz der gesuchten Werte in der Tabelle zu überprüfen. Der Nachteil hierbei ist, dass der Bloom-Index ähnlich wie der Hash Index viele Limitationen hat und nur über den Equality Operator verfügt, weshalb dieser nicht so flexibel eingesetzt werden kann, wie beispielsweise der B-Tree. Weitere Limitationen sind, dass der Bloom

Index nur Operatorklassen für die Datentypen `int4` und `text` im Modul enthält und das er keine UNIQUE Indexe und keine Suche nach NULL-Werten supportet [24].

Der Bloom Index ist ein beigetragenes Module in PostgreSQL und ist bereits bei der Postgres Installation als zusätzlich mitgeliefertes Module enthalten [25].

Vorteile:

- Effiziente und platzsparende Prüfung, ob ein Element in der Datenbank existiert oder ein Mitglied einer Menge ist
- Am besten geeignet für Tabellen mit vielen Attribute auf denen Abfragen nach der Existenz der Attribute mit beliebigen Kombinationen der Attribute gestellt werden können
- Nur ein Bloom-Index notwendig, um alle möglichen Abfragen nach der Existenz der Attribute zu unterstützen

Nachteile:

- Ist beschränkt auf den Equality Operator
- Besitzt im Modul nur Operatoren Klassen für die Datentypen `int4` und `text`
- Ist ein verlustbehafteter Index
- Supportet keine UNIQUE Indexe
- Supportet keine Suche nach NULL-Werten

Quelle: [24]

2.2.2 *btree_gin und btree_gist*

Neben dem B-Tree Index existieren noch zwei weitere, als Module bei der PostgreSQL Installation mitgelieferte Index Operatorenklassen („`btree_gin`“ [26] und „`btree_gist`“ [27]), welche zusätzliche beim GIN und beim GiST Index verwendet werden können, um ein B-Tree äquivalentes Verhalten für mehrere unterschiedliche Basis-Datentypen wie `int`, `float`, `numeric`, `timestamp`, `text`, `varchar` zu ermöglichen. Diese Operatorenklassen sind den standard B-Baum-Index Methoden nicht überlegen, da ihnen wichtige Eigenschaften, wie beispielsweise die Erzwingung von Eindeutigkeiten fehlen, jedoch bieten diese einige andere Funktionen, die bei einem standard B-Baum-Index nicht vorhanden sind. Beispielsweise implementieren diese Operatoren zusätzliche, spezifische Operatorenklassen für GIN und GiST Index, welche die jeweiligen Indexe unterstützen. Desweiteren ermöglichen die Operatorenklassen die mehrspaltige Nutzung eines GIN oder GiST-Indexes bei Spalten, die aus Basis-Datentypen sowie aus komplexeren, nur mit dem jeweiligen Index indizierbaren Datentypen, bestehen. Dies kann effizienter sein, als zwei separate Index auf die komplexeren Datentypen und die Basis-Datentypen zu erstellen [26, 27].

2.3 Externe / experimentelle Indexe

2.3.1 RUM Index

Der RUM Index basiert auf den Zugriffsmethoden Code vom GIN Index und referenziert dessen Implementierung. Somit gehört auch der RUM Index auch zu der Kategorie der Inverted Indexes und eignet sich für dieselben Use-Cases wie auch der GIN Index. Jedoch ist der RUM Index eine Optimierung von GIN Index, welche dessen Nachteile beim Full-Text-Search und bei zusammengesetzten Queries (tsvector + timestamp) verbessern soll. Beispielsweise werden die Zugriffsmethoden vom GIN Index verbessert, indem beim RUM Index direkt zu den Indexen die Lexempositionen oder die Zeitstempel hinterlegt werden. Diese werden benötigt, um index-level ranking zu supporten. Da der GIN Index die Lexempositionen aber nicht hinterlegt, müssen diese mittels der CPU nach dem Index-Scan durch einen zusätzlichen Heap-Scan ermittelt werden. Dadurch ist der RUM Index im Gegensatz zum GIN Index deutlich schneller im Ranking und in der Phrasensuche auf Indexlevel sowie im Ordnen der Daten mittels Zeitstempel. Indem der RUM Index die zusätzlichen Informationen nutzt, ermöglicht dieser auch double-field Indexe, wie zum Beispiel tsvector + timestamp, ohne zusätzliche CPU-Verarbeitung zu benötigen [28].

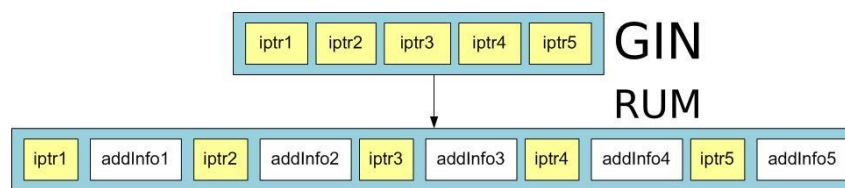


Abbildung 7: Beispiel einer möglichen Struktur eines RUM Indexes

Quelle: <https://github.com/postgrespro/rum>

Der Nachteil von RUM im Vergleich zu GIN ist, dass der RUM Index mehr Zeit zur Erstellung und zum Einfügen von Daten benötigt. Der Grund hierfür ist, dass der RUM Index zusätzliche Informationen neben den Schlüssel speichert und generische Write-Ahead Log Aufzeichnungen verwendet. Dadurch ist der RUM Index größer als der GIN Index und benötigt mehr Speicherplatz.

Der RUM Index ist ein externes Module für PostgreSQL und muss zusätzlich in PostgreSQL installiert werden. Die zusätzlichen Operatoren, Funktionen und die Operator Klassen können im README.md von der GitHub Dokumentation des RUM Indexes gefunden werden [28].

Vorteile:

- Bietet zusätzliche Optimierungen des GIN Indexes für die Text-Suche und für zusammengesetzte Queries
- Supportet index-level ranking, phrase search auf Indexlevel, double-field Indexe und das Ordnen von Daten mittels Zeitstempel ohne zusätzliche CPU-Verarbeitung oder Heap-Scan und ist somit für solche Anwendungen performanter als der GIN Index

Nachteile:

- Benötigt mehr Zeit und Speicherplatz zum Erstellen der Indexe als GIN Index, da neben den vom GIN Index gespeicherten Informationen, noch zusätzliche Informationen wie die Lemexpositionen / Zeitstempel und die generischen WAL Aufzeichnungen hinterlegt werden müssen.

Quelle: [28]

2.3.2 Hippo Index

Der Hippo Index ist ein spärlicher bereichsbasierter Index, der im Gegensatz zu bestehenden Baumindexstrukturen die Speicherung eines Zeigers für jedes Tupel in der indizierten Tabelle vermeidet. Stattdessen speichert der Hippo Index die Festplatten Seitenbereiche, welche die indizierte Datenbanktabelle repräsentieren und verwaltet histogrammbasierte Zusammenfassungen von den Seitenbereichen. Diese Zusammenfassungen sind kurze Histogramme, welche die Datenverteilung einer oder mehrerer Seiten darstellen. Dadurch ist der Hippo Index ein schneller und dennoch skalierbarer Ansatz zur Indizierung von spärlichen Datenbanken und ermöglicht eine Reduzierung des vom Index belegten Speicherplatzes. Jedoch unterstützt der Hippo Index bislang nur den Integer Datentyp und simple Vergleichsoperatoren.

Der Hippo Index ist ein externes Module für PostgreSQL und muss zusätzlich in PostgreSQL installiert werden.

Vorteile:

- Geringer Indizierungs-Overhead
- Konkurrenzfähige Query Performance
- Schnelle Index Maintenance

Nachteile:

- Beschränkung auf nur Integer Datentypen
- Beschränkt auf simple Vergleichoperatoren (<, <=, =, >=, >)

Quelle: [29]

2.3.3 VODKA Index

Der VODKA Index ist ein experimentaler Index, welcher wie der GIN Indexe zu der Kategorie der Inverted Indexe gehört und einen neuen Prototyp von Zugriffsmethoden einführen soll. Diese Zugriffsmethoden basieren stark auf den GIN Index und wurden für eine effizientere Indizierung von geschachtelten Strukturen, wie hstore und jsonb oder für die Freitextsuche optimiert. Durch die Einführung von diesen Datentypen in PostgreSQL, entstand auch die Notwendigkeit, hierarchische keys effizient indizieren zu können. Ein Problem hierbei ist, dass die doppelten keys bei den hierarchischen keys bei unabhängiger Speicherung der key-value Paare für einen unnötig grossen Speicherplatzverbrauch sorgen. Um dieses Problem zu lösen, wollen die Entwickler im VODKA Index die B-Tree Datenstruktur, welche im GIN Index verwendet wird, um die keys zu indizieren, durch eine digitale Baumstruktur ersetzen. Hierfür experimentieren die Entwickler mit hybriden Zugriffsmethoden, welche an die von SP-GiST und GIN angelehnt sind. Somit soll der GIN Index flexibler für den Support von einer grösseren Anzahl an Queries gemacht werden. Dadurch sollte es möglich sein, nicht nur die Schlüssel, sondern auch die Posting Lists zu indizieren.

Der VODKA Index ist ein externes Module für PostgreSQL und muss zusätzlich in PostgreSQL installiert werden.

Vorteile:

- Optimiert für eine effizientere Indizierung von geschachtelten Strukturen, wie hstore und jsonb
- Flexibler für den Support von einer grösseren Anzahl an Queries, sodass nicht nur Schlüssel, sondern auch Posting Lists indiziert werden können

Nachteile:

Wurden nicht thematisiert / noch nicht ermittelt

Quelle: [30]

3 TABELLARISCHER VERGLEICH AKTUELLER INDEXTYPEN

Wie aus dem vorherigen Kapitel ersichtlich wird, unterscheiden sich Indexe vor allem in ihrer Datenstruktur und in den Algorithmen und Techniken, mit welchen sie die Daten abrufen. Dadurch entstehen individuelle Vor- und Nachteile, welche entsprechend genutzt werden können, um mit unterschiedlichen Abfragen und Anwendungsfällen umzugehen. Welcher Index verwendet werden sollte, lässt sich somit nicht pauschalisieren. Jedoch ist beispielsweise bekannt, dass der B-Tree Index, welcher auch als Standard-Index von PostgreSQL verwendet wird, in den meisten Fällen und für die häufigsten Abfragen am besten passt. Werden aber spezielle Fälle wie beispielsweise eine Voll-Text-Suche, das Arbeiten mit GIS-Daten oder mit composite-values betrachtet, so kommen noch weitere Indexe in Frage [31]. Somit hängt die Wahl des Indexes stark von dem Datentyp, der Art der Abfragen und der zugrunde liegenden Daten innerhalb der Tabelle ab.

Um die Indexe sinnvoll miteinander vergleichen zu können, werden die relativen Aussagen zu den Indexen unter der Voraussetzung ermittelt, dass die Indexe mit deren idealen Datentyp implementiert worden sind und dass für diesen die passenden Operatoren existieren. Im Folgenden werden die Zuordnungen der idealen Datentypen zu den Indexen in absteigender Relevanz zu geteilt.

Datentypen	Indexe <i>[absteigend von links nach rechts sortiert nach Relevanz]</i>
numerische Datentypen [32] Date/Time Datentypen [33]	B-Tree, Hash, BRIN
(Grosse) Character Datentypen [34] JSON / JSONB Datentypen [35]	GIN, GIST, Bloom, BRIN, Hash, B-Tree
Arrays Datentypen [36] Composite Datentypen [37]	GIN, B-Tree
Text-Suchdatentypen [38]	GIN, RUM
Geometrische / Raumbezogene Datentypen [39]	GIST, SP-GIST

Entsprechend der Zuordnung von den Datentypen zu den Indexen, werden die aktuellen PostgreSQL Index Implementationen B-Tree, Hash, GIN, GiST, SP-GiST, BRIN und Bloom hinsichtlich ihrer Anwendungsfelder, Erstellzeit, Festplattengröße und der unterstützten Abfragen tabellarisch mit relativen Aussagen verglichen und gegenübergestellt.

Tabellarischer Vergleich aktueller Indextypen

<i>Indexes</i> <i>Criteria</i>	<i>B-TREE</i>	<i>GIST</i>	<i>SP-GIST</i>	<i>Hash</i>	<i>GIN</i>	<i>RUM</i>	<i>BRIN</i>	<i>Bloom</i>
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 Basis-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-seach, nearest neighbor search sowie das Arbeiten mit Geometrie-Daten (insbesondere 2D-Vektorgeometrie-Daten).	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 Attributen 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 indexierten 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 Erstzeit abhängig von gewählten Baumstruktur und den angepassten Datentypen.	Mässig-Langsam: Ähnlich wie beim GIST variiert die Erstzeit 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 indexierten Werte als Keys und den Referenzen auf die Daten als Values befüllt.	Langsam: Langsame Erstzeit, da jeder individuelle Wert innerhalb eines zusammengesetzten Datentyps indexiert werden muss. Erstzeit 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 indexierten 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 Gröszenbeschränkungen und hat in Vergleich zum GIN Index einen geringeren Festplatten-Footerprint.	Mittel bis gross: Ähnlich wie beim GIST variiert die Grösse abhängig von der gewählten nicht-balancierten Baumstruktur.	Klein bis mittel: Speichert einen aus den indexierten 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 Indextypen 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 indexierte Wert liegt.	Klein: Speicherung eines Arrays mit n Bits und der indexierten Werte
Unterstütze 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 Basis-Datentypen (rum_TYPE_ops), Arrays (rum_anynarray_ops, rum_anynarray_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

4 BENCHMARK ZWISCHEN RELATIONALEN UND MIT JSONB- OBJEKTEN VERSCHACHTELTEN DATENBANKEN

4.1 Ziel

Die Datenmodellierung beeinflusst die Optimierungsmöglichkeiten. So ist ein JSON-Dokument eine "Black Box", in der Abfragen nicht wirklich optimiert werden können. Mit PG JSONB hingegen wird versucht, JSONB-Dokumente zu analysieren und damit auch zu optimieren. Allerdings hat auch dies seine Grenzen. Diese Grenzen sollen im folgenden Benchmark ermittelt und aufgezeigt werden.

4.2 Vorgehensweise

Der Benchmark betrachtet zwei Datenbanken; eine relationale Datenbank mit gewöhnlichen Typen und eine Datenbank mit geschachtelten JSONB Objekten. Um aussagekräftige Werte für unterschiedlich grosse Datensätze zu ermitteln, wurden beide Datenbanken für jeweils 50K, 100K, 500K, 1M und 2M Records untersucht. Dabei wurden einmal die Datenbanken ohne Index gebenchmarkt, und einmal mit dem B-Tree Index, welcher die Records der relationalen Datenbank referenziert und einmal mit dem GIN-Index, welcher die Records der JSONB Datenbank referenziert.

Der Benchmark soll dadurch ermitteln, ob es Nachteile hinsichtlich der Performance bei einer Datenbank mit geschachtelten Attributen als JSONB Objekten gibt. Hierzu wurde die Performance beider Datenbanken mittels verschiedenen Queries und Operationen getestet, analysiert und verglichen.

Dazu wurden die folgenden Abfragen und Operationen getestet:

- I Point Query
- II Multipoint Query
- III Range Query
- IV Conjunction Query (AND)
- V Disjunction Query (OR)
- INSERT
- UPDATE
- DELETE

Zum Testen dieser Abfragen und Operationen verwendet der Benchmark die "Dell DVD Store Database Test Suite", um verschiedene Grössen einer Beispieldatenbank zu erzeugen (<https://github.com/dvdstore/ds21/blob/master/ds2>). In dieser Beispieldatenbank wird die Tabelle CUSTOMERS für den relationalen Datenbank-Benchmark und die Tabelle CUSTOMERS_JSONB für den jsonb-Datenbank-Benchmark verwendet. Somit wird die CUSTOMERS Tabelle mit Hilfe der „Dell DVD Store Database Test Suite“ generiert, während die CUSTOMERS_JSONB Tabelle innerhalb des Benchmarks dynamisch aus den Daten der CUSTOMERS Tabelle erzeugt wird.

4.3 Diskussion und Ausblick

Unabhängig von der Tabellengrösse zeigt der Benchmark, dass die JSONB-Tabelle bei allen Abfragen sowie bei der Indexerstellungszeit und Indexgrösse deutlich mehr Zeit benötigt als die relationale Tabelle. Jedoch muss auch beachtet werden, dass bei der relationalen Tabelle der B-Tree Index als partieller Index mit einem Bedingungsausdruck für die RANGE QUERY-, CONJUNCTION QUERY und DISJUNCTION QUERY erstellt worden ist. Somit benötigt dieser Beispielsweise bei dem Disjunction Query nur einen Index-Scan, während der GIN Index hierfür zwei Index-Scans für beide genannten Bedingungen durchführen muss. Dadurch hat der partielle B-Tree Index im Vergleich zum nicht-partiellen GIN Index einen Vorteil in der Performance. Im Nachtrag wird kurz erläutert, was gemacht werden könnte, um diesen Vorteil zu verhindern.

Ausserdem konnte in der Analyse des Query Planners beobachtet werden, dass die Indizes in den Range-Queries sowohl für die JSONB-Tabelle als auch für die relationale Tabelle nicht angewendet wurden. Dies könnte daran liegen, dass die zurückgegebene Ergebnis Menge so gross war, dass der Index sich nicht rentiert hätte.

Query Planer Ergebnisse aus dem Benchmark für die relationale Tabelle mit B-Tree Index und 2M Customers:

```
Seq Scan on customers (cost=0.00..69978.00 rows=1778667 width=16)
    (actual time=0.014..756.841 rows=1781374 loops=1)
    Filter: (age >= 26)
    Rows Removed by Filter: 218626
    Planning Time: 0.750 ms
    Execution Time: 801.116 ms
```

Query Planer Ergebnisse aus dem Benchmark für die JSONB Tabelle mit GIN Index und 2M Customers:

```
Seq Scan on customers_jsonb (cost=0.00..196685.34 rows=666667 width=128)
    (actual time=0.054..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
```

Ein weiteres auffälliges Ergebnis des Benchmarks ist, dass der GIN-Index bei den Disjunction Query in der JSONB-Tabelle ab 500K Tupeln die Suchzeit deutlich verschlechtert, obwohl der Query Planer den Index verwendet. Jedoch scheint es, als würde der Query Planer den `personal_gin_idx` Index zweifach für einen Bitmap Index Scan verwenden, da dieser im Gegensatz zum B-Tree Index, wie zuvor erwähnt, nicht als partitioneller Index erstellt wurde.

```
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
```

Zusammenfassend lässt sich sagen, dass die JSONB-Tabelle in allen Vergleichspunkten schlechter abschnitt als die relationale Tabelle. Basierend auf den Ergebnissen aus dem Benchmark sollte daher eine JSONB-Tabelle in Kombination mit einem Standard-GIN-Index nur dann verwendet werden, wenn der Datentyp / die Daten es nicht anders zulassen.

Jedoch muss hervorgehoben werden, dass der GIN-Index im Benchmark nur mit dem Standard-Operator verwendet worden ist. Möglicherweise könnte die Performance verbessert werden, unter der Verwendung von anderen Operatoren, entsprechend dem Einsatzbereich.

Auch muss erwähnt werden, dass der GIN-Index vermutlich effizienter gewesen wäre, wäre der Einsatzbereich beispielsweise eine Textsuche in einem Dokument, bei der mittels der `pg_trgm` Extension und dem `gin_trgm_ops` Operator, die Textsuche deutlich verschnellert werden könnte.

4.4 Nachtrag

Um den Vorteil des partitionellen B-Tree Indexes gegenüber des nicht partitionellen GIN Indexes zu verhindern, würde es genügen, die Bedingungsaustrücke der partionierten Indexe zu entfernen und nur die jeweiligen Attribute age und gender als Spalten zu indezieren.

Vorher:

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

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

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

Nachher:

```
CREATE INDEX age_idx ON customers USING btree(age);
```

```
CREATE INDEX gender_idx ON customers USING btree (gender);
```

Somit könnte sichergestellt werden, dass sowohl der B-Tree als auch der GIN Index beide unter den selben Bedingungen ohne weitere Optimierungen gebenchmarkt werden.

5 LITERATUR

- [1] R. Elmasri und S. Navathe, *Fundamentals of database systems*. Boston und 24 andere: Pearson, 2016. [Online]. Verfügbar unter: <https://www.auhd.site/upfiles/elibrary/Azal2020-01-22-12-28-11-76901.pdf>
- [2] Malcolm McLean, *Postgres Conference - A talk about indexes*. [Online]. Verfügbar unter: <https://postgresconf.org/conferences/SouthAfrica2018/program/proposals/a-talk-about-indexes> (Zugriff am: 11. Juni 2021).
- [3] Couchbase Docs, *Query Types | Introduction to Query Types | Compound Queries*. [Online]. Verfügbar unter: <https://docs.couchbase.com/server/current/fts/fts-query-types.html> (Zugriff am: 11. Juni 2021).
- [4] Nikolaus Augsten, „Database Tuning | Query Types“. [Online]. Verfügbar unter: https://dbresearch.uni-salzburg.at/teaching/2015ws/dbt/dbt_03-handout-2x2.pdf
- [5] PostgreSQL Documentation, *11.7. Indexes on Expressions*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/indexes-expressional.html> (Zugriff am: 11. Juni 2021).
- [6] PostgreSQL Documentation, *11.8. Partial Indexes*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/indexes-partial.html> (Zugriff am: 11. Juni 2021).
- [7] PostgreSQL Documentation, *CREATE INDEX*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/sql-createindex.html> (Zugriff am: 11. Juni 2021).
- [8] GitHub, *jiayuas/bitmap-postgresql*. [Online]. Verfügbar unter: <https://github.com/jiayuas/bitmap-postgresql> (Zugriff am: 11. Juni 2021).
- [9] Wikipedia, *Index Sequential Access Method*. [Online]. Verfügbar unter: https://de.wikipedia.org/w/index.php?title=Index_Sequential_Access_Method&oldid=199614970 (Zugriff am: 11. Juni 2021).
- [10] Wikipedia, *Inverted index*. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=Inverted_index&oldid=1024439323 (Zugriff am: 11. Juni 2021).
- [11] Wikipedia, *Block Range Index*. [Online]. Verfügbar unter: https://en.wikipedia.org/w/index.php?title=Block_Range_Index&oldid=1017705533 (Zugriff am: 11. Juni 2021).
- [12] R. Marcus *et al.*, „Benchmarking learned indexes“, *Proc. VLDB Endow.*, Jg. 14, Nr. 1, S. 1–13, 2020, doi: 10.14778/3421424.3421425.
- [13] Tobin J. Lehman, Michael J. Carey, „A Study of Index Structures for Main Memory Database Management Systems“. [Online]. Verfügbar unter: <http://www.vldb.org/conf/1986/P294.PDF>
- [14] PostgreSQL Documentation, *11.2. Index Types*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/indexes-types.html> (Zugriff am: 11. Juni 2021).
- [15] John Porvaznik, *Avoiding the Pitfalls of BRIN Indexes in Postgres*. [Online]. Verfügbar unter: <https://blog.crunchydata.com/blog/avoiding-the-pitfalls-of-brin-indexes-in-postgres> (Zugriff am: 11. Juni 2021).

- [16] EDB, *Are Hash Indexes Faster than Btree Indexes in Postgres?* [Online]. Verfügbar unter: <https://www.enterprisedb.com/postgres-tutorials/are-hash-indexes-faster-btree-indexes-postgres> (Zugriff am: 11. Juni 2021).
- [17] H. Benita, „Re-Introducing Hash Indexes in PostgreSQL“, *Haki Benita*, 11. Jan. 2021, 2021. [Online]. Verfügbar unter: <https://hakibenita.com/postgresql-hash-index>. Zugriff am: 11. Juni 2021.
- [18] gist.cs.berkeley.edu, *The Gist of the GiST*. [Online]. Verfügbar unter: <http://gist.cs.berkeley.edu/gist1.html> (Zugriff am: 11. Juni 2021).
- [19] PostgreSQL Documentation, *12.9. GIN and GiST Index Types*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/textsearch-indexes.html> (Zugriff am: 11. Juni 2021).
- [20] Mohamed Y. Eltabakh, Ramy Eltarras und Walid G. Aref, „Space-partitioning Trees in PostgreSQL: Realization and Performance“. [Online]. Verfügbar unter: <https://www.cs.purdue.edu/spgist/papers/icde06.pdf>
- [21] Felix Kunde, *PostGIS Indexing*. [Online]. Verfügbar unter: <https://slides.com/fxku/postgis-indexing#/5/1> (Zugriff am: 11. Juni 2021).
- [22] Malcolm McLean, *A talk about indexes*. [Online]. Verfügbar unter: https://www.youtube.com/watch?v=HAn1xu6_SW0 (Zugriff am: 11. Juni 2021).
- [23] PostgreSQL Documentation, *66.1. GIN Indexes / Introduction*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/12/gin-intro.html> (Zugriff am: 11. Juni 2021).
- [24] PostgreSQL Documentation, *F.5. bloom*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/bloom.html> (Zugriff am: 11. Juni 2021).
- [25] PostgreSQL Documentation, *Appendix F. Additional Supplied Modules*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/contrib.html> (Zugriff am: 12. Juli 2021).
- [26] PostgreSQL Documentation, *F.6. btree_gin*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/btree-gin.html> (Zugriff am: 13. Juli 2021).
- [27] PostgreSQL Documentation, *F.7. btree_gist*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/btree-gist.html> (Zugriff am: 13. Juli 2021).
- [28] GitHub, *postgrespro/rum*. [Online]. Verfügbar unter: <https://github.com/postgrespro/rum> (Zugriff am: 11. Juni 2021).
- [29] GitHub, *DataSystemsLab/hippo-postgresql*. [Online]. Verfügbar unter: <https://github.com/DataSystemsLab/hippo-postgresql> (Zugriff am: 11. Juni 2021).
- [30] Alexander Korotkov, Oleg Bartunov und Teodor Sigaev, *PGCon2014: CREATE INDEX USING VODKA*. [Online]. Verfügbar unter: <https://www.pgcon.org/2014/schedule/events/696.en.html> (Zugriff am: 11. Juni 2021).
- [31] Craig Kerstiens, *A tour of Postgres Index Types*. [Online]. Verfügbar unter: <https://www.citusdata.com/blog/2017/10/17/tour-of-postgres-index-types/> (Zugriff am: 11. Juni 2021).
- [32] PostgreSQL Documentation, *8.1. Numeric Types*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/datatype-numeric.html> (Zugriff am: 11. Juni 2021).
- [33] PostgreSQL Documentation, *8.5. Date/Time Types*. [Online]. Verfügbar unter: <https://www.postgresql.org/docs/current/datatype-datetime.html> (Zugriff am: 11. Juni 2021).

- [34] PostgreSQL Documentation, *8.3. Character Types*. [Online]. Verfügbar unter:
<https://www.postgresql.org/docs/current/datatype-character.html> (Zugriff am: 11. Juni 2021).
- [35] PostgreSQL Documentation, *8.14. JSON Types*. [Online]. Verfügbar unter:
<https://www.postgresql.org/docs/current/datatype-json.html> (Zugriff am: 11. Juni 2021).
- [36] PostgreSQL Documentation, *8.15. Arrays*. [Online]. Verfügbar unter:
<https://www.postgresql.org/docs/current/arrays.html> (Zugriff am: 11. Juni 2021).
- [37] PostgreSQL Documentation, *8.16. Composite Types*. [Online]. Verfügbar unter:
<https://www.postgresql.org/docs/current/rowtypes.html> (Zugriff am: 11. Juni 2021).
- [38] PostgreSQL Documentation, *8.11. Text Search Types*. [Online]. Verfügbar unter:
<https://www.postgresql.org/docs/current/datatype-textsearch.html> (Zugriff am: 11. Juni 2021).
- [39] PostgreSQL Documentation, *8.8. Geometric Types*. [Online]. Verfügbar unter:
<https://www.postgresql.org/docs/current/datatype-geometric.html> (Zugriff am: 11. Juni 2021).

APPENDIX BENCHMARK

REQUIREMENTS

- Windows OS to run the .bat files
- Installation of the “Dell DVD Store Database Test Suite” (<https://linux.dell.com/dvdstore/>) for the generic database generation → see chapter CONFIGURATION

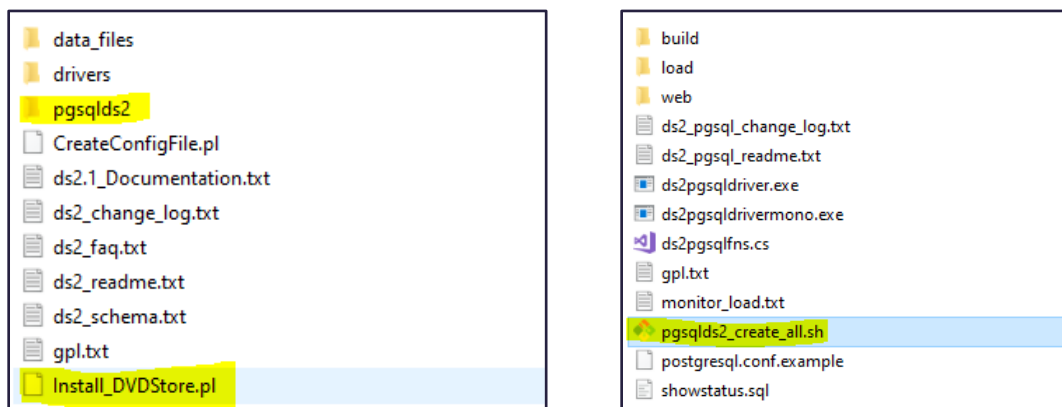
INSTALLATION

- Navigate into your C:\ drive:
> cd C:\
- Clone repository:
> git clone https://github.com/4realDev/psql-relational-vs-jsonb-benchmark.git

CONFIGURATION AND USAGE

- For the Benchmark the “Dell DVD Store Database Test Suite” is used. This is an open source simulation of an online ecommerce site with implementations in Microsoft SQL Server, Oracle, MySQL and PostgreSQL along with driver programs and web applications. This Suite is used to generate five different databases with similar data and data distribution.
- For an installing guide please refer to the readme.txt on the official GitHub repository (https://github.com/dvdstore/ds21/blob/master/ds2/ds2.1_Documentation.txt).

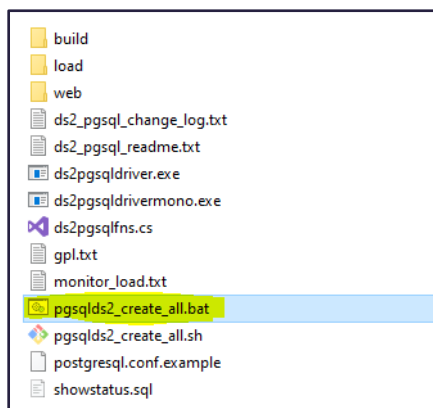
The result inside the ds2 folder should look like this:



- Use the Install_DVDStore.pl script in the root directory to generate the CSV files for the five ds2 benchmark databases with the size of:

Database size	Records inside Customers table
25 MB	50 K
50 MB	100 K
250 MB	500 K
500 MB	1M
1 GB	2M

- To start the generic database generation with the ds2 tool, we need execute the `pgsqlds2_create_all.sh` shell script in the `ds2\pgsqlds2` directory of the ds21 repository. But since the benchmark is on WINDOWS, and the repository only provides a shell script for LINUX, this would not work. Therefore, small adjustments were made on the script, to convert it into a batch (.bat) script. This batch script can be found in the root directory of this repository under the name **adjusted_pgsqlds2_create_all.bat**. Copy paste this script into the same directory where the `pgsqlds2_create_all.sh` script is and execute the **adjusted_pgsqlds2_create_all.bat** script instead of the `pgsqlds2_create_all.sh` script.



- Before execution make sure that no one uses the databases and all the connections to the databases are closed.

The adjustments made to the `pgsqlds2_create_all.sh` to convert it into the

adjusted_pgsqlds2_create_all.bat are the following:

1. Adjust the shell syntax to the batch script to make it work on WINDOWS
2. Commenting out the `pgsqlds2_create_ind.sql` file, which is responsible for the index creation in the generated benchmark database benchmark, as the benchmark first tests the generated database without any indexes, and only then creates its OWN indexes (B-Tree and GIN), with which the database is tested again.
3. Enter the path, where you cloned this repository and directly run the benchmark (execute the `run-benchmark.sql`) on the generated database, after it has generated.

Through this adjustments is it possible to only call one run script to execute the database generation, as well as the benchmark.

DIRECTORIES AND FILES

Within the download you will find the following directories and files, logically grouping common .sql and .bat scripts. You will see something like this:

```
C:
| adjusted_pgsqllds2_create_all.bat
| cache-fill.sql
| convert-relational-db-into-jsonb-db.sql
| create-cache-fill-table.sql
| ignore.txt
| LICENSE
| README.md
| run-benchmark.sql
|
+---jsonb-db
|   jsonb-db__delete.sql
|   jsonb-db__gin-index-creation.sql
|   jsonb-db__I-point-query.sql
|   jsonb-db__II-multipoint-query.sql
|   jsonb-db__III-range-query.sql
|   jsonb-db__insert.sql
|   jsonb-db__IV-conjunction-query.sql
|   jsonb-db__update.sql
|   jsonb-db__V-disjunction-query.sql
|
|   \---results
|       jsonb-db__I-point-query-with-gin-index.txt
|       jsonb-db__I-point-query-without-index.txt
|       jsonb-db__II-multipoint-query-with-gin-index.txt
|       jsonb-db__II-multipoint-query-without-index.txt
|       jsonb-db__III-range-query-with-gin-index.txt
|       jsonb-db__III-range-query-without-index.txt
|       jsonb-db__IV-conjunction-query-with-gin-index.txt
|       jsonb-db__IV-conjunction-query-without-index.txt
|       jsonb-db__V-disjunction-query-with-gin-index.txt
|       jsonb-db__V-disjunction-query-without-index.txt
|
|   \---relational-db
|       relational-db__btree-index-creation.sql
|       relational-db__delete.sql
|       relational-db__I-point-query.sql
|       relational-db__II-multipoint-query.sql
|       relational-db__III-range-query.sql
|       relational-db__insert.sql
|       relational-db__IV-conjunction-query.sql
|       relational-db__update.sql
|       relational-db__V-disjunction-query.sql
|
|   \---results
|       relational-db__I-point-query-with-btree-index.txt
|       relational-db__I-point-query-without-index.txt
|       relational-db__II-multipoint-query-with-btree-index.txt
|       relational-db__II-multipoint-query-without-index.txt
|       relational-db__III-range-query-query-with-btree-index.txt
|       relational-db__III-range-query-query-without-index.txt
|       relational-db__IV-conjunction-query-with-btree-index.txt
|       relational-db__IV-conjunction-query-without-index.txt
|       relational-db__V-disjunction-query-with-btree-index.txt
|       relational-db__V-disjunction-query-without-index.txt
```

DOCUMENTATION

Explaining of the important Files and Folders

adjusted_pgsqls2_create_all.bat

Starts the generic database generation directly with the benchmark (run-benchmark.sql) in the ds2 tool on WINDOWS.

Needs to be copied and pasted into the same directory where the pgsqls2_create_all.sh script from the ds2 GitHub Repo is.

run-benchmark.sql

Start script for the benchmark.

Creates the CUSTOMERS_JSONB table by converting the CUSTOMERS table with the convert-relational-db-into-jsonb-db script. Creates the cache-fill-table with the create-cache-fill-table.sql script. Fills the cache before every query and operation with the cache-fill.sql script. Calls all the queries, insert, delete, and update operations and fills the cache with cache-fill.sql before every query or operation script.

convert-relational-db-into-jsonb-db.sql

Converts the CUSTOMERS relational data table into the CUSTOMERS_JSONB jsonb data table.

create-cache-fill-table.sql

Generates a random table with the same amount of records as chosen for the CUSTOMERS table.

cache-fill.sql

Selects all elements from the created cache-fill-table to override the cache before any query I-V, insert, delete or update operation. Ensures no caching of previous results.

relational-db folder

Includes all .sql scripts for index creation of the B-Tree-Index, the different queries (I-V) and the insert, delete and update operation regarding the relational database.

jsonb-db folder

Includes all .sql scripts for index creation of the GIN-Index, the different queries (I-V) and the insert, delete and update operation regarding the jsonb-db database.

Results folder in jsonb-db and relational-db





















Stores the EXPLAIN ANALYZE output of the query planner with the results from the queries (I-V) and the insert, delete and update operation

QUERIES

The EXPLAIN ANALYZE results of the Query Planner for the queries with and without an index for both the tables, json and relational, will be generated dynamically every time the benchmark is executed.

These results are saved inside .txt files in the folders:

- `psql-relational-vs-jsonb-benchmark\relational-db\results` for the Queries on the relational CUSTOMERS table
- `psql-relational-vs-jsonb-benchmark\jsonb-db\results` for the Queries on the jsonb CUSTOMERS_JSONB table.

 <code>relational-db_III-range-query-query-with-btree-index.txt</code>	 <code>jsonb-db_III-range-query-with-gin-index.txt</code>
 <code>relational-db_III-range-query-query-without-index.txt</code>	 <code>jsonb-db_III-range-query-without-index.txt</code>
 <code>relational-db_II-multipoint-query-with-btree-index.txt</code>	 <code>jsonb-db_II-multipoint-query-with-gin-index.txt</code>
 <code>relational-db_II-multipoint-query-without-index.txt</code>	 <code>jsonb-db_II-multipoint-query-without-index.txt</code>
 <code>relational-db_I-point-query-with-btree-index.txt</code>	 <code>jsonb-db_I-point-query-with-gin-index.txt</code>
 <code>relational-db_I-point-query-without-index.txt</code>	 <code>jsonb-db_I-point-query-without-index.txt</code>
 <code>relational-db_IV-conjunction-query-with-btree-index.txt</code>	 <code>jsonb-db_IV-conjunction-query-with-gin-index.txt</code>
 <code>relational-db_IV-conjunction-query-without-index.txt</code>	 <code>jsonb-db_IV-conjunction-query-without-index.txt</code>
 <code>relational-db_V-disjunction-query-with-btree-index.txt</code>	 <code>jsonb-db_V-disjunction-query-with-gin-index.txt</code>
 <code>relational-db_V-disjunction-query-without-index.txt</code>	 <code>jsonb-db_V-disjunction-query-without-index.txt</code>

Attention: These results will be overwritten every time the benchmark is executed.

The queries and their query plan are shown below. To simplify matters, only the query plans of the queries that refer to the CUSTOMERS / CUSTOMERS_JSONB table with 1,000,000 records were used.

POINT QUERY

relational-db/ relational-db__l-point-query.sql:

```
SELECT
    firstname,
    lastname,
    gender,
    age
FROM
    customers
WHERE
    firstname = 'VKUUXF'
    AND lastname = 'ITHOMQJNYX';
```

Query Plan without Index:

```
Gather (cost=1000.00..29740.10 rows=1 width=16) (actual time=184.855..193.285 rows=0 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Seq Scan on customers (cost=0.00..28740.00 rows=1 width=16)
      (actual time=125.313..125.313 rows=0 loops=3)
      Filter: (((firstname)::text = 'VKUUXF'::text) AND
              ((lastname)::text = 'ITHOMQJNYX'::text))
      Rows Removed by Filter: 333333
  Planning Time: 0.444 ms
  Execution Time: 193.316 ms
```

Query Plan with B-Tree Index:

```
Index Scan using firstname_lastname_idx on customers (cost=0.42..6.19 rows=1 width=16)
(actual time=0.075..0.075 rows=0 loops=1)
  Index Cond: (((firstname)::text = 'VKUUXF'::text) AND
              ((lastname)::text = 'ITHOMQJNYX'::text))
  Planning Time: 2.075 ms
  Execution Time: 0.092 ms
```

jsonb-db/jsonb-db__l-point-query.sql:

```
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"}}}';
```

Query Plan without Index:

```
Gather (cost=1000.00..98860.56 rows=378 width=128) (actual time=940.580..1081.836 rows=0 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Seq Scan on customers_jsonb (cost=0.00..97822.76 rows=158 width=128)
      (actual time=885.136..885.136 rows=0 loops=3)
      Filter: ((personal @> '{"personal": {"name": {"firstname": "VKUUXF"}}}'::jsonb)
              AND (personal @> '{"personal": {"name": {"lastname": "ITHOMQJNYX"}}}'::jsonb))
      Rows Removed by Filter: 333333
  Planning Time: 0.384 ms
  Execution Time: 1081.858 ms
```

Query Plan with GIN-Index:

```
Bitmap Heap Scan on customers_jsonb (cost=37.02..429.51 rows=100 width=128) (actual
time=0.485..0.486 rows=0 loops=1)
  Recheck Cond: ((personal @> '{"personal": {"name": {"firstname": "VKUUXF"}}}'::jsonb)
  AND (personal @> '{"personal": {"name": {"lastname": "ITHOMQJNYX"}}}'::jsonb))
  -> Bitmap Index Scan on personal_gin_idx (cost=0.00..37.00 rows=100 width=0)
      (actual time=0.483..0.483 rows=0 loops=1)
      Index Cond: ((personal @> '{"personal": {"name": {"firstname": "VKUUXF"}}}'::jsonb)
      AND (personal @> '{"personal": {"name": {"lastname": "ITHOMQJNYX"}}}'::jsonb))
Planning Time: 1.187 ms
Execution Time: 0.526 ms
```

MULTI POINT QUERY**relational-db__II-multipoint-query.sql:**

```
SELECT
    firstname,
    lastname,
    gender,
    age
FROM
    customers
WHERE
    state = 'SD';
```

Query Plan without Index:

```
Gather (cost=1000.00..29571.63 rows=8733 width=16) (actual time=0.389..215.115 rows=9720
loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Seq Scan on customers (cost=0.00..27698.33 rows=3639 width=16) (actual
time=0.221..147.784 rows=3240 loops=3)
    Filter: ((state)::text = 'SD'::text)
    Rows Removed by Filter: 330093
Planning Time: 0.084 ms
Execution Time: 215.678 ms
```

Query Plan with B-Tree Index:

```
Bitmap Heap Scan on customers (cost=100.11..16952.34 rows=8733 width=16)
(actual time=2.125..84.251 rows=9720 loops=1)
  Recheck Cond: ((state)::text = 'SD'::text)
  Heap Blocks: exact=6526
  -> Bitmap Index Scan on state_idx (cost=0.00..97.92 rows=8733 width=0)
      (actual time=1.404..1.405 rows=9720 loops=1)
      Index Cond: ((state)::text = 'SD'::text)
Planning Time: 0.125 ms
Execution Time: 85.179 ms
```

jsonb-db__II-multipoint-query.sql:

```
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"}}';
```

Query Plan without Index:

```
Gather (cost=1000.00..98977.24 rows=37830 width=128)
(actual time=1.113..525.799 rows=9720 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Seq Scan on customers_jsonb (cost=0.00..94194.24 rows=15762 width=128)
      (actual time=0.537..457.062 rows=3240 loops=3)
      Filter: (location @> '{"location": {"state": "SD"}}'::jsonb)
      Rows Removed by Filter: 330093
Planning Time: 0.070 ms
Execution Time: 526.301 ms
```

Query Plan with GIN-Index:

```
Bitmap Heap Scan on customers_jsonb (cost=109.50..27921.33 rows=10000 width=128) (actual
time=3.132..474.290 rows=9720 loops=1)
  Recheck Cond: (location @> '{"location": {"state": "SD"}}'::jsonb)
  Heap Blocks: exact=8586
  -> Bitmap Index Scan on location_gin_idx (cost=0.00..107.00 rows=10000 width=0)
      (actual time=2.072..2.073 rows=9720 loops=1)
      Index Cond: (location @> '{"location": {"state": "SD"}}'::jsonb)
Planning Time: 0.131 ms
Execution Time: 475.451 ms
```

RANGE QUERY**relational-db__III-range-query.sql:**

```
SELECT
    firstname,
    lastname,
    gender,
    age
FROM
    customers
WHERE
    age >= 26;
```

Query Plan without Index:

```
Seq Scan on customers (cost=0.00..34990.00 rows=891200 width=16) (actual
time=0.037..472.670 rows=890480 loops=1)
  Filter: (age >= 26)
  Rows Removed by Filter: 109520
Planning Time: 0.072 ms
Execution Time: 493.882 ms
```

Query Plan with B-Tree Index:

```
Seq Scan on customers (cost=0.00..34990.00 rows=891200 width=16) (actual
time=0.016..427.989 rows=890480 loops=1)
  Filter: (age >= 26)
  Rows Removed by Filter: 109520
Planning Time: 0.574 ms
Execution Time: 449.410 ms
```

jsonb-db__III-range-query.sql:

```
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';
```

Query Plan without Index:

```
Seq Scan on customers_jsonb (cost=0.00..165597.92 rows=1260992 width=128) (actual
time=0.058..2281.391 rows=890480 loops=1)
  Filter: (((personal -> 'personal'::text) ->> 'age'::text) >= '26'::text)
  Rows Removed by Filter: 109520
  Planning Time: 0.057 ms
  Execution Time: 2306.498 ms
```

Query Plan with GIN-Index:

```
Seq Scan on customers_jsonb (cost=0.00..98342.66 rows=333333 width=128) (actual
time=0.032..2250.314 rows=890480 loops=1)
  Filter: (((personal -> 'personal'::text) ->> 'age'::text) >= '26'::text)
  Rows Removed by Filter: 109520
  Planning Time: 0.078 ms
  Execution Time: 2274.686 ms
```

CONJUNCTION QUERY**relational-db__IV-conjunction-query.sql:**

```
SELECT
    firstname,
    lastname,
    gender,
    age
FROM
    customers
WHERE
    age = 26
    AND gender = 'F'
```

Query Plan without Index:

```
Gather (cost=1000.00..30444.60 rows=7046 width=16) (actual time=0.385..251.849 rows=6920
loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Seq Scan on customers (cost=0.00..28740.00 rows=2936 width=16) (actual
time=0.398..178.343 rows=2307 loops=3)
    Filter: ((age = 26) AND ((gender)::text = 'F'::text))
    Rows Removed by Filter: 331027
    Planning Time: 0.105 ms
    Execution Time: 252.271 ms
```

Query Plan with B-Tree Index:

```
Bitmap Heap Scan on customers (cost=68.64..15030.45 rows=7046 width=16) (actual
time=1.774..55.900 rows=6920 loops=1)
  Recheck Cond: ((age = 26) AND ((gender)::text = 'F'::text))
  Heap Blocks: exact=5981
  -> Bitmap Index Scan on age_above_twenty_six_gender_female_idx (cost=0.00..66.88
rows=7046 width=0) (actual time=1.149..1.150 rows=6920 loops=1)
    Planning Time: 0.146 ms
    Execution Time: 56.311 ms
```

jsonb-db__IV-conjunction-query.sql:

```
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"}}';
```

Query Plan without Index:

```
Gather (cost=1000.00..98860.56 rows=378 width=128) (actual time=0.626..552.927 rows=6920 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Seq Scan on customers_jsonb (cost=0.00..97822.76 rows=158 width=128)
      (actual time=0.402..480.538 rows=2307 loops=3)
      Filter: ((personal @> '{"personal": {"age": 26}}'::jsonb)
        AND (personal @> '{"personal": {"gender": "F"}}'::jsonb))
      Rows Removed by Filter: 331027
Planning Time: 0.350 ms
Execution Time: 553.394 ms
```

Query Plan with GIN-Index:

```
Bitmap Heap Scan on customers_jsonb (cost=37.02..429.51 rows=100 width=128) (actual time=8.380..616.400 rows=6920 loops=1)
  Recheck Cond: ((personal @> '{"personal": {"age": 26}}'::jsonb)
    AND (personal @> '{"personal": {"gender": "F"}}'::jsonb))
  Heap Blocks: exact=6642
  -> Bitmap Index Scan on personal_gin_idx (cost=0.00..37.00 rows=100 width=0)
      (actual time=7.496..7.496 rows=6920 loops=1)
      Index Cond: ((personal @> '{"personal": {"age": 26}}'::jsonb)
        AND (personal @> '{"personal": {"gender": "F"}}'::jsonb))
Planning Time: 0.411 ms
Execution Time: 617.044 ms
```

DISJUNCTION QUERY**relational-db__V-disjunction-query.sql:**

```
SELECT
    firstname,
    lastname,
    gender,
    age
FROM
    customers
WHERE
    age = 26
    OR gender = 'F';
```

Query Plan without Index:

```
Seq Scan on customers (cost=0.00..37490.00 rows=505621 width=16) (actual time=0.045..508.814 rows=506618 loops=1)
  Filter: ((age = 26) OR ((gender)::text = 'F'::text))
  Rows Removed by Filter: 493382
Planning Time: 0.077 ms
Execution Time: 521.733 ms
```

Query Plan with B-Tree Index:

```
Bitmap Heap Scan on customers (cost=4404.81..34532.86 rows=509203 width=16) (actual
time=30.487..271.310 rows=506618 loops=1)
  Recheck Cond: ((age = 26) OR ((gender)::text = 'F'::text))
  Heap Blocks: exact=22490
  -> Bitmap Index Scan on age_above_twenty_six_or_gender_female_idx
      (cost=0.00..4277.51 rows=509203 width=0)
      (actual time=27.613..27.614 rows=506618 loops=1)
Planning Time: 0.149 ms
Execution Time: 283.255 ms
```

jsonb-db__V-disjunction-query.sql:

```
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"}}';
```

Query Plan without Index:

```
Gather (cost=1000.00..106975.04 rows=75281 width=128) (actual time=0.425..878.729
rows=506618 loops=1)
  Workers Planned: 2
  Workers Launched: 2
  -> Parallel Seq Scan on customers_jsonb (cost=0.00..98446.94 rows=31367 width=128)
      (actual time=0.233..738.660 rows=168873 loops=3)
      Filter: ((personal @> '{"personal": {"age": 26}}'::jsonb)
      OR (personal @> '{"personal": {"gender": "F"}}'::jsonb))
      Rows Removed by Filter: 164461
Planning Time: 0.066 ms
Execution Time: 894.373 ms
```

Query Plan with GIN-Index:

```
Bitmap Heap Scan on customers_jsonb (cost=223.83..45625.54 rows=19885 width=128) (actual
time=60.209..2058.600 rows=506618 loops=1)
  Recheck Cond: ((personal @> '{"personal": {"age": 26}}'::jsonb)
  OR (personal @> '{"personal": {"gender": "F"}}'::jsonb))
  Rows Removed by Index Recheck: 220644
  Heap Blocks: exact=41081 lossy=33091
  -> BitmapOr (cost=223.83..223.83 rows=19985 width=0)
      (actual time=54.096..54.097 rows=0 loops=1)
      -> Bitmap Index Scan on personal_gin_idx (cost=0.00..106.94 rows=9992 width=0)
          (actual time=2.661..2.661 rows=13709 loops=1)
          Index Cond: (personal @> '{"personal": {"age": 26}}'::jsonb)
      -> Bitmap Index Scan on personal_gin_idx (cost=0.00..106.94 rows=9992 width=0)
          (actual time=51.434..51.434 rows=499829 loops=1)
          Index Cond: (personal @> '{"personal": {"gender": "F"}}'::jsonb)
Planning Time: 0.105 ms
Execution Time: 2074.577 ms
```

CREATION OF THE JSONB TABLE CUSTOMERS_JSONB**QUERY**

```
CREATE table CUSTOMERS_JSONB as(
  SELECT
    customerid,
    jsonb_build_object('personal',
      jsonb_build_object(
        'name', jsonb_build_object('firstname', firstname, 'lastname', lastname),
        'gender', gender,
        'age', age,
        'income', income,
        'contact', jsonb_build_object('email', email, 'phone', phone)
      )
    ) as "personal",
    jsonb_build_object('location',
      jsonb_build_object(
        'address1', address1,
        'address2', address2,
        'city', city,
        'state', state,
        'zip', zip,
        'country', country,
        'region', region
      )
    ) as "location",
    jsonb_build_object(
      'creditcardtype', creditcardtype,
      'creditcard', creditcard,
      'creditcardexpiration', creditcardexpiration
    ) as "credit",
    jsonb_build_object(
      'username', username,
      'password', password
    ) as "login"
  FROM CUSTOMERS
);
```


RELATIONAL DATABASE STRUCTURE

```
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)
login: {
  USERNAME,
  PASSWORD
}
```

JSONB OBJECT STRUCTURE

```
CUSTOMERID,
personal: {
  name: {
    FIRSTNAME,
    LASTNAME
  },
  GENDER,
  AGE,
  INCOME,
  contact: {
    EMAIL,
    PHONE
  }
},
location: {
  ADDRESS1,
  ADDRESS2,
  CITY,
  STATE,
  ZIP,
  COUNTRY,
  REGION
},
credit: {
  CREDITCARDTYPE,
  CREDITCARD,
  CREDITCARDEXPIRATION
},
```

INDEX CREATION

CREATE B-TREE INDEX FOR RELATIONAL DATABASE

```
CREATE INDEX firstname_lastname_idx ON customers USING btree (firstname, lastname);  
  
CREATE INDEX state_idx ON customers USING btree(state); // Index used for comparing  
  
CREATE INDEX age_above_twenty_six_idx ON customers USING btree(age) WHERE age >= 26;  
  
CREATE INDEX age_above_twenty_six_or_gender_female_idx ON customers USING btree(age,gender)  
    WHERE age = 26 OR gender='F';  
  
CREATE INDEX age_above_twenty_six_gender_female_idx ON customers USING btree(age, gender)  
    WHERE age = 26 AND gender='F';
```

CREATE GIN INDEX FOR JSONB DATABASE

```
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);
```

INSERT, DELETE AND UPDATE OPERATIONS**INSERT RECORDS INTO THE RELATIONAL DB WITH B-TREE INDEX**

```
INSERT INTO customers(  
    customerid,  
    firstname,  
    lastname,  
    address1,  
    address2,  
    city,  
    state,  
    zip,  
    country,  
    region,  
    email,  
    phone,  
    creditcardtype,  
    creditcard,  
    creditcardexpiration,  
    username,  
    password,  
    age,  
    income,  
    gender)  
WITH tmp as (SELECT MAX(customerid) as max FROM customers)  
SELECT  
    g + tmp.max as customerid,  
  
    substr(md5(random()::text), 1, 50),  
    substr(md5(random()::text), 1, 50),  
  
    substr(md5(random()::text), 1, 50),  
    NULL,  
    substr(md5(random()::text), 1, 50),  
    substr(md5(random()::text), 1, 50),  
    random()::integer * 99999,  
    substr(md5(random()::text), 1, 50),  
    random()::integer * 2,  
  
    substr(md5(random()::text), 1, 50),  
    substr(md5(random()::text), 1, 50),  
  
    random()::integer * 5,  
    substr(md5(random()::text), 1, 50),  
    substr(md5(random()::text), 1, 50),  
  
    substr(md5(random()::text), 1, 50),  
    'password',  
  
    random()::integer * 100,  
    random()::integer * 100000,  
    substr(md5(random()::text), 1, 1)  
FROM tmp, generate_series(1,max) as g;  
VACUUM ANALYZE
```

INSERT RECORDS INTO THE RELATIONAL DB WITH B-TREE INDEX

```
INSERT INTO customers_jsonb(customerid, personal, location, credit, login)
WITH tmp as (SELECT MAX(customerid) as max FROM customers_jsonb)
SELECT
  g + tmp.max as customerid,
  jsonb_build_object(
    'age', random()::integer * 100,
    'name', jsonb_build_object(
      'lastname', substr(md5(random()::text), 1, 50),
      'firstname', substr(md5(random()::text), 1, 50)
    ),
    'gender', substr(md5(random()::text), 1, 1),
    'income', random()::integer * 100000,
    'contact', jsonb_build_object(
      'email', substr(md5(random()::text), 1, 50),
      'phone', substr(md5(random()::text), 1, 50)
    )
  ),
  jsonb_build_object(
    'zip', random()::integer * 99999,
    'city', substr(md5(random()::text), 1, 50),
    'state', substr(md5(random()::text), 1, 50),
    'region', random()::integer * 2,
    'country', substr(md5(random()::text), 1, 50),
    'address1', substr(md5(random()::text), 1, 50),
    'address2', NULL
  ),
  jsonb_build_object(
    'creditcard', substr(md5(random()::text), 1, 50),
    'creditcardtype', random()::integer * 5,
    'creditcardexpiration', substr(md5(random()::text), 1, 50)
  ),
  jsonb_build_object(
    'username', substr(md5(random()::text), 1, 50),
    'password', 'password'
  )
FROM tmp, generate_series(1,max) as g;
VACUUM ANALYZE
```

UPDATE RECORDS FROM THE RELATIONAL DB WITH B-TREE INDEX

```
UPDATE customers SET region = 3, creditcardtype = 6
WHERE country = 'China';
VACUUM ANALYZE
```

UPDATE RECORDS FROM THE JSONB DB WITH GIN INDEX

```
UPDATE
  customers_jsonb SET location = jsonb_set(location, '{location}',
  location->'location' || '{"region": "3"}'),
  credit = jsonb_set(credit, '{creditcardtype}', '6')
WHERE location@> '{"location": {"country": "China"}}';
VACUUM ANALYZE
```

DELETE RECORDS FROM THE RELATIONAL DB WITH B-TREE INDEX

```
DELETE FROM CUSTOMERS
WHERE customerid > (SELECT ((COUNT(*)/2)) FROM CUSTOMERS);
```

DELETE RECORDS FROM THE JSONB DB WITH GIN INDEX

```
DELETE FROM CUSTOMERS_JSONB
WHERE customerid > (SELECT ((COUNT(*)/2)) FROM CUSTOMERS_JSONB);
```

RESULTS OF THE BENCHMARK

Disclaimer on the Index build time and Index size:

For the index build time and size I took the index for the multipoint query as reference.

As a reminder, the multipoint query asks for all the records, where the state = "SD".

For the relation CUSTOMERS table, the B-Tree index could reference the state column directly

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

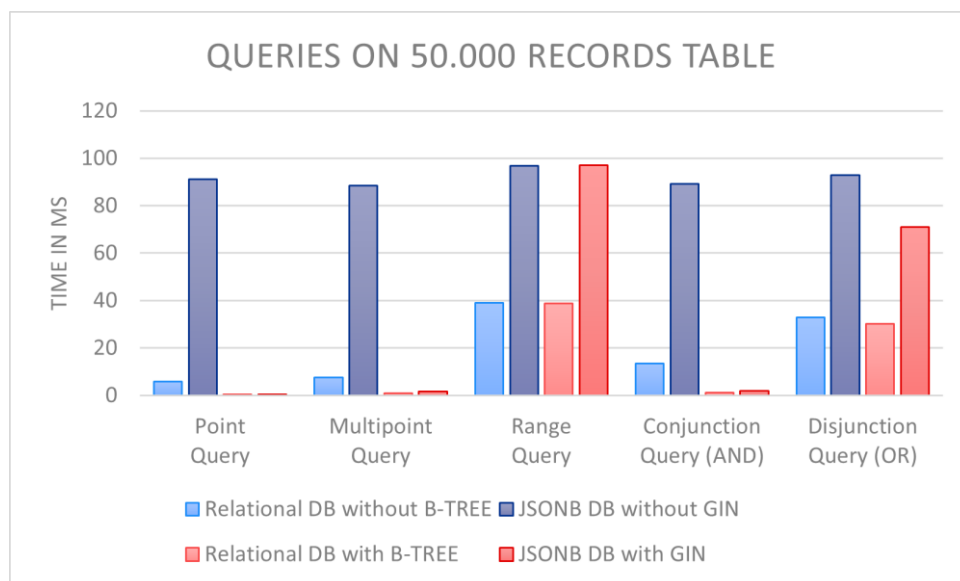
For the jsonb CUSTOMERS_JSONB table, the GIN index had to reference "location", which is an nested JSONB object, storing the state as well as multiple other attributes (address1, address2, city, state, zip, country, region)

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

Therefore, the GIN index needs more build time and has a larger index size.

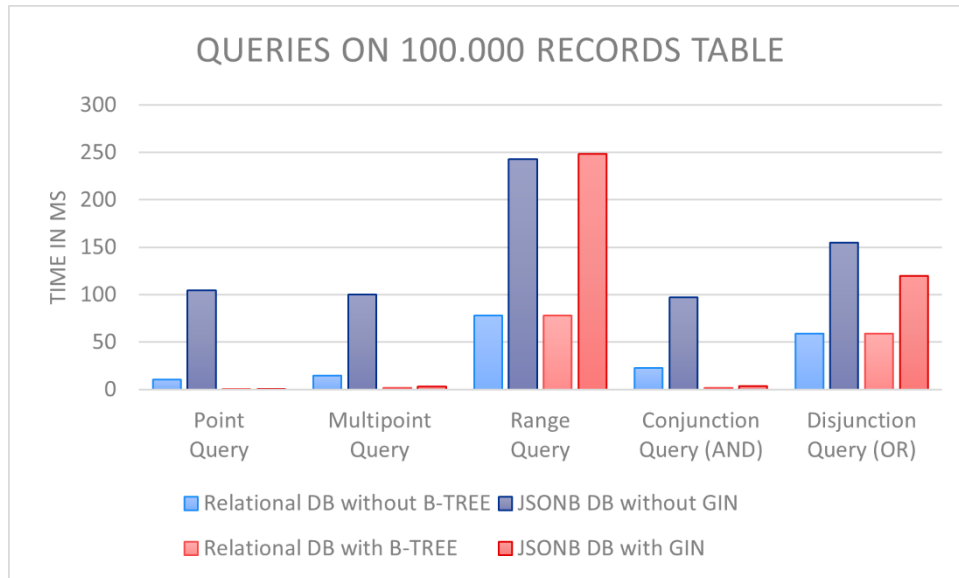
50.000 RECORDS

<i>Databases</i> <i>Criteria</i>	<i>Relational DB without B-TREE</i>	<i>JSONB DB without GIN</i>	<i>Relational DB with B-TREE</i>	<i>JSONB DB with GIN</i>
Index build time	-	-	136.652	526.053
Index size	-	-	360 kB	5912 kB
Point Query	5.83	91.19	0.35	0.36
Multipoint Query	7.47	88.53	1	1.59
Range Query	38.98	96.85	38.88	97.02
Conjunction Query (AND)	13.46	89.09	1.15	1.77
Disjunction Query (OR)	32.98	92.96	30.09	71.05
Insert	-	-	3045	4812.655
Delete	-	-	69.074	77.850
Update	-	-	84.290	99.086



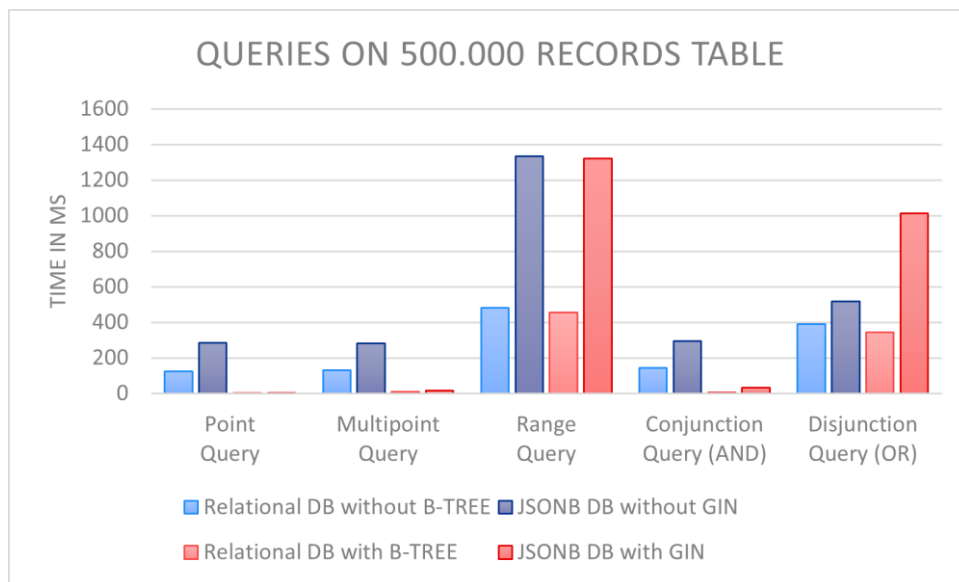
100.000 RECORDS

<i>Databases</i> <i>Criteria</i>	<i>Relational DB</i> <i>without B-TREE</i>	<i>JSONB DB</i> <i>without GIN</i>	<i>Relational DB</i> <i>with B-TREE</i>	<i>JSONB DB</i> <i>with GIN</i>
Index build time	-	-	237.271	1173.697
Index size	-	-	712 kB	10232 kB
Point Query	10.56	104.49	0.28	0.37
Multipoint Query	14.7	100.15	1.72	2.88
Range Query	78.2	242.86	77.94	248.26
Conjunction Query (AND)	22.96	97.21	1.46	3.49
Disjunction Query (OR)	59.17	155.13	58.74	119.57
Insert	-	-	6545.223	10875.262
Delete	-	-	173.489	422.251
Update	-	-	172.176	197.837



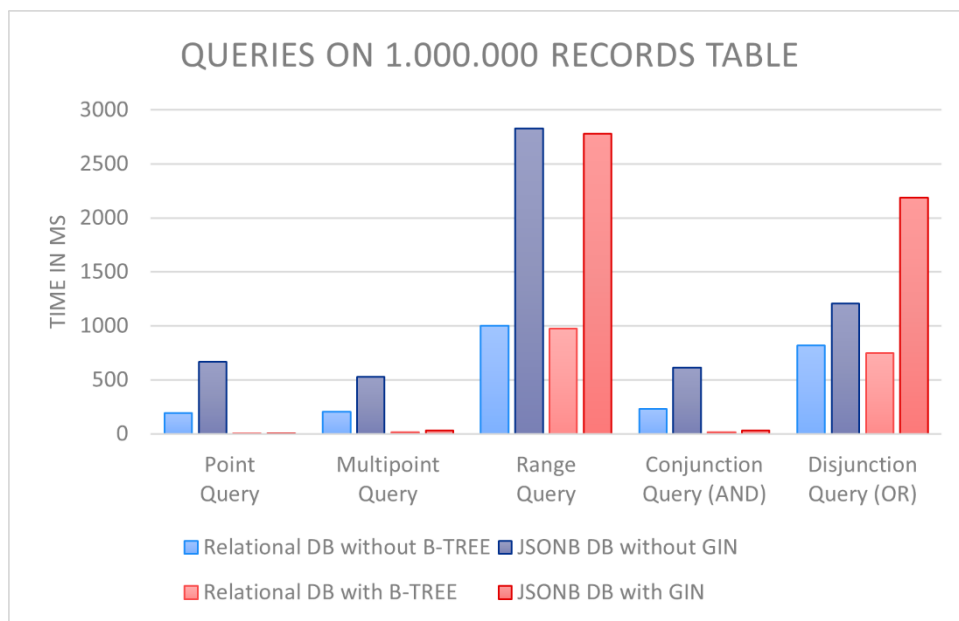
500.000 RECORDS

<i>Databases</i> <i>Criteria</i>	<i>Relational DB without B-TREE</i>	<i>JSONB DB without GIN</i>	<i>Relational DB with B-TREE</i>	<i>JSONB DB with GIN</i>
Index build time	-	-	1211,361	5053,779
Index size	-	-	3432 kB	32 MB
Point Query	124.04	284.6	0.37	0.36
Multipoint Query	130.02	279.21	7.88	14.49
Range Query	481.31	1333.39	455.28	1319.35
Conjunction Query (AND)	142.89	293.31	6.19	31.96
Disjunction Query (OR)	389.43	516.05	342.08	1011.76
Insert	-	-	35493.885 (00:35,494)	87954.862 (01:27,955)
Delete	-	-	2167.978	13837.663
Update	-	-	2639.710	4891.530



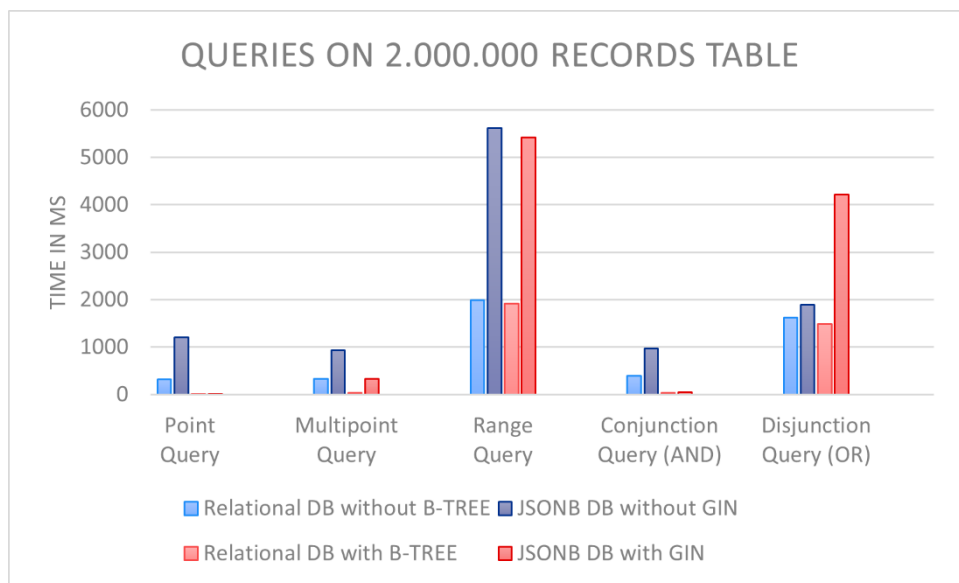
1.000.000 RECORDS

<i>Databases</i> Criteria	<i>Relational DB</i> <i>without B-TREE</i>	<i>JSONB DB</i> <i>without GIN</i>	<i>Relational DB</i> <i>with B-TREE</i>	<i>JSONB DB</i> <i>with GIN</i>
Index build time	-	-	1833,489	9348,099
Index size	-	-	6816 kB	62 MB
Point Query	193.23	666.14	0.3	0.36
Multipoint Query	205.67	528.18	15.57	29.8
Range Query	1001.86	2827.11	976.37	2780.63
Conjunction Query (AND)	232.8	610.25	12.12	32.64
Disjunction Query (OR)	819.49	1207.8	747.22	2185.06
Insert	-	-	79852,687 (01:19,853)	494033,035 (08:14,033)
Delete	-	-	11390,526	27597,459
Update	-	-	4863,120	24153,605



2.000.000 RECORDS

<i>Databases</i> <i>Criteria</i>	<i>Relational DB without B-TREE</i>	<i>JSONB DB without GIN</i>	<i>Relational DB with B-TREE</i>	<i>JSONB DB with GIN</i>
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



INDEX BUILD TIME AND INDEX SIZE

The GIN index takes about 5 times longer to create than the B-tree and is about 10 times larger.

However, as already mentioned in the disclaimer, this was obvious because the GIN index references the entire location attributes instead only the state like the B-Tree Index.

