YugabyteDB YSQL GIN indexes

Jason Kim

April 9, 2021

Contents

1 Intro

1.1 Background

```
CREATE TABLE book (page int PRIMARY KEY, word text, position int);
```

creates a DocDB table with key `page HASH`. I can easily ask for pages 5-12 using this primary key:

```
SELECT * FROM book WHERE page >= 5 and page <= 12;
```

```
CREATE INDEX ON book (word);
```

creates a secondary DocDB table with key `word HASH, page ASC`. This is like the index at the back of a book. I can easily ask what pages have the word `foo` using this index:

```
SELECT page FROM book WHERE word = 'foo';
```

What if the table were structured instead like

```
CREATE TABLE book (page int PRIMARY KEY, words text[]);
```

Now, looking for the specific word `foo` is time-consuming:

```
SELECT * FROM book WHERE words && ARRAY['foo'];
```

Creating a regular index won't help since you still need to search `words` for `foo`.

1.2 Overview

GIN indexes map values *inside a column* rather than the whole column.

```
CREATE INDEX ON book USING gin (words);
```

should create a secondary DocDB table with key `word HASH, pages ASC`, where `word` is a word in `words`. If I insert a page with 300 unique words and 500 total words, I add 300 records to the index, all referencing the same page.

1.3 Terms

- **indexed table**: indexes are on *indexed tables*
- **pending list**: in postgres, to avoid having each GIN index insert hit disk, write to a linear list of tuples (*pending list*) first, and flush it later in bulk
- **posting list**: in postgres, a GIN index tuple maps key to *posting list*, a list of ctids corresponding to the indexed table rows

1.4 Limitations

- GIN indexes can only be used on column types `tsvector`, `anyarray`, `jsonb`
- GIN indexes cannot be unique
- GIN indexes can be on more than one column, but all columns must be GINable

2 Components

First, it helps to know the following:

- access method:[1] storage interface (e.g. `btree`, `lsm`, `gin`)
- operator:[2] (e.g. `int4 >= int8` $\rightarrow$ `bool`, `- int8` $\rightarrow$ `int8`)
- operator family:[3] collection of operators + access method
- operator class:[4] operator family + in type + key type

We only need to concern ourselves with things related to the GIN access method. The components to cover are

1. operators
2. operator classes
3. primitives
4. translations of operators to primitives

2.1 Operators

You may want to familiarize yourself with the operators used by GIN. See the appendix.

2.2 Opclasses

Here are the opclasses that can be used with GIN:[5]

opclass	in type	key type	supported operators
`tsvector_ops`	`tsvector`	`text`	`@@`, `@@@`
`array_ops`	`anyarray`	`anyelement`	`&&`, `<@`, `=`, `@>`
`jsonb_ops`	`jsonb`	`text`	`?`, `?&`, `?|`, `@>`, `@?`, `@@`
`jsonb_path_ops`	`jsonb`	`int4`	`@>`, `@?`, `@@`
`jsonb_full_ops`	`jsonb`	`bytea`	`?`, `?&`, `?|`, `@>`, `@?`, `@@`

Notice that `jsonb` has three opclasses. The first two are in upstream postgres; the third is inspired by CockroachDB's inverted index RFC.[6]

Opclasses are per-column, so you can have multiple on a single index:

See the appendix for examples of translating columns (in type) to keys (key type).

```
CREATE INDEX ON bar USING gin (jsonb_col jsonb_ops, jsonb_col jsonb_path_ops);
```

2.3 Primitives

All the operators boil down to a combination of these primitives:

- $\mathsf{get}_I(f)$: given index I and predicate f, return set of tuples

$$T = \{(i, p) \in I : f(i)\}$$

[1] https://www.postgresql.org/docs/13/catalog-pg-am.html
[2] https://www.postgresql.org/docs/13/catalog-pg-operator.html
[3] https://www.postgresql.org/docs/13/catalog-pg-opfamily.html
[4] https://www.postgresql.org/docs/13/catalog-pg-opclass.html
[5] https://www.postgresql.org/docs/13/gin-builtin-opclasses.html
[6] https://github.com/cockroachdb/cockroach/blob/master/docs/RFCS/20171020_inverted_indexes.md

f should be a simple predicate like

- $i < c$
- $i = c$
- $i > c$
- $i \geq c$
- $i \leq c$
- i starts with c

- $\mathsf{and}(T_1, T_2)$: given sets of tuples T_1, T_2, return

$$T = \{(i, p) \in T_1 \cup T_2 : p \in K\}$$

where

$$K = T_1[\texttt{pk}] \cap T_2[\texttt{pk}]$$

- $\mathsf{or}(T_1, T_2)$: given sets of tuples T_1, T_2, return

$$T = T_1 \cup T_2$$

- recheck: a catchall for additional operations

2.4 Translations

See translations of all operators to primitives in the appendix.

For example, the `=` operator for arrays makes sure that the indexed array column equals the query array. This can be done by the following steps:

1. For each distinct element e in the query array A_q, get a set of tuples T_e where the tuple's row's array contains the element e:

$$\forall e \in A_q, T_e = \mathsf{get}_I(f)$$

where

$$f(i) = (i \overset{?}{=} e)$$

2. Intersect the tuples T_e to filter out any tuples whose row's array doesn't contain **all** of the query array's elements:

$$T = \mathsf{and}_{e \in A_q}(T_e)$$

3. Recheck each row to make sure the `=` operator is satisfied.

3 Read and write path

3.1 Write path

For postgres, `INSERT INTO table_with_gin_index (to_tsvector('simple', 'the quick brown'))` does

1. prepare index keys: `the`, `quick`, `brown`
2. for each index key, append the indexed table ctid to the key's posting list

For Yugabyte, we should

1. prepare index keys: `the`, `quick`, `brown`
2. **for each index key, write (key, ctid) pair to DocDB**

For `DELETE`s and `UPDATE`s, we need to figure out what GIN index records to delete.

3.2 Read path

For postgres, `SELECT * FROM table_with_gin_index WHERE tscol @@ to_tsquery('simple', 'the')` does

1. create scan key: `the`
2. get tuples matching scan key
3. recheck each tuple if needed

For Yugabyte, we should

1. create scan key: `the`
2. **fetch tuples from DocDB matching scan key**
3. recheck each tuple if needed

4 DocDB encoding

In general, GIN index records should be encoded like

```
[<gin_index_key>, <primary_keys>]
```

If there's more than one column in the GIN index, it should also be part of the record:

```
[<gin_column_attnum>, <gin_index_key>, <primary_keys>]
```

If the columns of the GIN index are of different key type, this will likely pose a problem for DocDB since DocDB tables have fixed schemas. Here are several solutions:

- *Create a DocDB table for each column.* Problem is that, in postgres, the index will show up once, but it somehow maps to multiple indexes in DocDB. It will probably take a lot of work to reconcile.
- *Have each GIN index column appear in the record, but make sure only one is active while the rest are null.* Problem is that it's a waste of space, and the code needs to be careful not to violate the constraint.
- *Force all GIN index columns to map to text.* Problem is that you're still forced to do all ASC or all DESC. Also, some extra processing needs to be done to translate between types.
- *Relax the requirement for columns to have a fixed schema.* This may also take work.

Since the attnum DocDB column is internal, allowing the user to specify hash or range on the hidden attnum column will require changes up to the syntax. For now, it can be forced to be range partitioned.

4.1 tsvector

The GIN key is of type `text`. We can encode them to DocDB `kString`. For example, `"the"` is encoded to `"Sthe\x00\x00"` if ascending and `"a\x8b\x97\x9a\xff\xff"` if descending.[7] Since UTF-8 strings are guaranteed to not have bytes `"\x00"`, `"\xfe"`, or `"\xff"`, it should be simple to create bounds for prefix search. In fact, this is already happening for queries like `col LIKE 'foo%'`, which turn into a `QL_OP_BETWEEN` of `"foo"` and `"fop"`.

Range partition is needed for efficient prefix queries. Otherwise, hash partition is fine.

Example:

```sql
CREATE TABLE tsvtab (i int, ts1 tsvector, ts2 tsvector, PRIMARY KEY (i ASC));
CREATE INDEX ON tsvtab USING gin (ts1 ASC, ts2 DESC);
INSERT INTO tsvtab VALUES (4, 'abc abc', 'def ghi');
```

Assuming schemas don't need to be fixed, the index should contain

- `H\x80\x00\x00\x02`, `Sabc\x00\x00`, `SH\x80\x00\x01\x00\x01\x04!\x00\x00`, `!`, `J\x80`, `#...`
- `H\x80\x00\x00\x03`, `a\x9b\x9a\x99\xff\xff`, `SH\x80\x00\x01\x00\x01\x04!\x00\x00`, `!`, `J\x80`, `#...`
- `H\x80\x00\x00\x03`, `a\x98\x97\x96\xff\xff`, `SH\x80\x00\x01\x00\x01\x04!\x00\x00`, `!`, `J\x80`, `#...`

This assumes the column attnum will be encoded using `kInt32`. This can likely be optimized because it's constrained as a 2-byte int.

[7] see `AppendEncodedStrToKey`

```
SELECT * FROM tsvtab WHERE ts1 @@ 'ab:*';
```

should look for

- ≥ `H\x80\x00\x00\x02Sab\x00\x00`
- < `H\x80\x00\x00\x02Sac\x00\x00`

```
SELECT * FROM tsvtab WHERE ts2 @@ 'de:*';
```

should look for

- > `H\x80\x00\x00\x03a\x9b\x99\xff\xff`
- ≤ `H\x80\x00\x00\x03a\x9b\x9a\xff\xff`

TODO: look into `tsvector` **weights**

4.2 anyarray

The GIN key is the type of the array element. We can encode them correspondingly to DocDB. There are no additional requirements since the operators don't need range partitioning.

4.3 jsonb

For `jsonb_ops`, the GIN key is of type `text`. Like tsvector, we can encode to DocDB `kString`. For example, `"\001abc"` can be encoded to `"S\x01abc\x00\x00"` if ascending. There may be prefix operations on strings using the `starts with` jsonpath operator, but since `jsonb_ops` isn't geared towards solving those queries, it should stay largely hash partitioned.

For `jsonb_path_ops`, the GIN key is of type `int4`. Internally, it seems to be unsigned 4-byte int, so let's go with that: we can encode to DocDB `kUInt32`. For example, `2147483648` can be encoded to `"O\x00\x00\x00\x80"` if ascending and `"g\xff\xff\xff\x4f"` if descending.[8] There is no advantage of using range partitioning since the ints are hashes.

For `jsonb_full_ops`, the GIN key is of type `text`. Like before, encode as DocDB `kString`. For this, DocDB will need to interpret the path to push down queries that operate on certain parts of the path, like `'$.a.b.c.ceiling() == 3'`. An alternative is to encode each path part as a separate DocKey component, but then DocDB will need to support flexible schemas.

[8]see `AppendUInt32ToKey`

Appendices

A Key format for normal index

Here is a step-by-step guide to see how normal indexes are represented in DocDB.

```
./bin/yb-ctl create \
  --master_flags "ysql_disable_index_backfill=true" \
  --tserver_flags "TEST_docdb_log_write_batches=true,ysql_disable_index_backfill=true,y⌋
  ↪  sql_num_shards_per_tserver=1"
tail -F ~/yugabyte-data/node-1/disk-1/yb-data/tserver/logs/yb-tserver.INFO
```

```
CREATE TABLE t (p bool PRIMARY KEY, c char, i int);
INSERT INTO t VALUES (true, 'b', 2);
INSERT INTO t VALUES (false, null, null);
```

```
CREATE INDEX ON t (c);
```

Observe logs for *regular* DocDB writes

```
I0216 18:08:50.375550 31976 tablet.cc:1235] T dfc95b4d53b44afebc4827b29bcc6769 P
↪  5fb87e8c88ea477ab7ebb4b9a3bb4bdc: Wrote 2 key/value pairs to kRegular RocksDB:
Frontiers: { smallest: { op_id: 1.3 hybrid_time: { physical: 1613527730374840 }
↪  history_cutoff: <invalid> hybrid_time_filter: <invalid> } largest: { op_id: 1.3
↪  hybrid_time: { physical: 1613527730374840 } history_cutoff: <invalid>
↪  hybrid_time_filter: <invalid> } }
1. PutCF(SubDocKey(DocKey(0xebd4, ["b"], ["G\x8f\xf7T!!"]), [SystemColumnId(0); HT{
↪  physical: 1613527730370761 }]), '#\x80\x01\x98\xbfC\xf5\xd5\xab\x80J$'
↪  (23800198BF43F5D5AB804A24))
2. PutCF(SubDocKey(DocKey(0x4d44, [null], ["G\xdc@F!!"]), [SystemColumnId(0); HT{
↪  physical: 1613527730370761 w: 1 }]), '#\x80\x01\x98\xbfC\xf5\xd5\xab\x80?\xab$'
↪  (23800198BF43F5D5AB803FAB24))
```

In simpler terms,

1. `["b", true]`
2. `[null, false]`

`IndexScan` on `c = 'b'` can look in this index for `["b"]` prefix, get the next key component `true`, then look up `true` in the indexed table.

```
CREATE UNIQUE INDEX ON t (i);
```

Observe logs for *regular* DocDB writes

```
I0216 18:10:06.280701 31753 tablet.cc:1235] T bb21a24b24eb421b8b7a84fb03422271 P
↪  5fb87e8c88ea477ab7ebb4b9a3bb4bdc: Wrote 4 key/value pairs to kRegular RocksDB:
Frontiers: { smallest: { op_id: 1.3 hybrid_time: { physical: 1613527806280035 }
↪  history_cutoff: <invalid> hybrid_time_filter: <invalid> } largest: { op_id: 1.3
↪  hybrid_time: { physical: 1613527806280035 } history_cutoff: <invalid>
↪  hybrid_time_filter: <invalid> } }
1. PutCF(SubDocKey(DocKey(0xc0c4, [2], [null]), [SystemColumnId(0); HT{ physical:
↪  1613527806278239 }]), '#\x80\x01\x98\xbf?o\x8e\x93\x80J$'
↪  (23800198BF3F6F8E93804A24))
2. PutCF(SubDocKey(DocKey(0xc0c4, [2], [null]), [ColumnId(12); HT{ physical:
↪  1613527806278239 w: 1 }]), '#\x80\x01\x98\xbf?o\x8e\x93\x80?\xabSG\x8f\xf7T!!'
↪  (23800198BF3F6F8E93803FAB53478FF7542121))
```

```
3. PutCF(SubDocKey(DocKey(0x4d44, [null], ["G\xdc@F!!"]), [SystemColumnId(0); HT{
↪  physical: 1613527806278239 w: 2 }]), '#\x80\x01\x98\xbf?o\x8e\x93\x80?\x8b$'
↪  (23800198BF3F6F8E93803F8B24))
4. PutCF(SubDocKey(DocKey(0x4d44, [null], ["G\xdc@F!!"]), [ColumnId(12); HT{ physical:
↪  1613527806278239 w: 3 }]), '#\x80\x01\x98\xbf?o\x8e\x93\x80?kSG\xdc@F!!'
↪  (23800198BF3F6F8E93803F6B5347DC40462121))
```

In simpler terms,

1. `[2, null]` ↦ `true`
2. `[null, false]` ↦ `false`

`IndexScan` on `i = 2` can look in this index for `[2]` prefix, get the value `true`, then look up `true` in the indexed table.

B Tables

B.1 Operator definitions

The following PostgreSQL docs cover most of the operators.

- tsvector operators
- anyarray operators
- jsonb operators

For convenience, they are also organized below.

`tsvector` operators:

signature	description	example
`tsvector @@ tsquery`	match query	(see below)

`tsquery` expression operators:

operator	description	example
`expr | expr`	boolean OR	`to_tsvector('foo qux') @@ to_tsquery('foo | bar')`
`expr & expr`	boolean AND	`to_tsvector('bar baz foo') @@ to_tsquery('foo & bar')`
`! expr`	boolean NOT	`to_tsvector('bar baz qux') @@ to_tsquery('! foo')`
`string:*`	prefix match	`to_tsvector('bar foo baz') @@ to_tsquery('fo:*')`

`anyarray` operators:

signature	description	example
`anyarray && anyarray`	overlap	`ARRAY[1, 4, 3] && ARRAY[2, 1]`
`anyarray <@ anyarray`	is contained by	`ARRAY[2, 2, 7] <@ ARRAY[1, 7, 4, 2, 6]`
`anyarray = anyarray`	equal	`ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]`
`anyarray @> anyarray`	contains	`ARRAY[1,4,3] @> ARRAY[3,1,3]`

`jsonb` operators:

signature	description	example
`jsonb ? text`	Is string `text` a top-level key in `jsonb`?	`'{"a":1, "b":2}'::jsonb ? 'b'`
`jsonb ?& text[]`	Are all strings in `text[]` a top-level key in `jsonb`?	`'{"a":1, "b":2, "c":3}'::jsonb ?& ARRAY['b', 'c']`
`jsonb ?| text[]`	Are any strings in `text[]` a top-level key in `jsonb`?	`'{"a":1, "b":2}'::jsonb ?| ARRAY['b', 'c']`
`jsonb @> jsonb`	Does the left `jsonb` contain the right `jsonb` at the top level?	`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb`
`jsonb @? jsonpath`	Does `jsonpath` for `jsonb` return any item? `WITH w AS (SELECT jsonb_path_query( <jsonb>, <jsonpath>)) SELECT count(*) > 0 FROM w;`	`'{"a":[1,2,3,4]}'::jsonb @? '$.a[*] ? (@ > 2)'`
`jsonb @@ jsonpath`	Is the first item of `jsonpath` for `jsonb` true? `SELECT jsonb_path_match( <jsonb>, <jsonpath>, '{}', true);`	`'{"a":[1,2,3,4]}'::jsonb @@ '$.a[*] > 2'`

`jsonpath` expression operators:

Given `j jsonb := '{"a":[2,4,"b",2,"c",0.5], "d":["ef","gh"]}'`,

operator	description	example: `jsonb_path_query(j, · )`	result
`value == value`	equality	`'$.a[*] ? (@ == 2)'`	`2, 2`
`value > value`	greater-than	`'$.a[*] ? (@ > 1)'`	`2, 4, 2`
`value != value`	non-equality	`'$.a[*] ? (@ != 4)'`	`2, 2, 0.5`
		`'$.a[*] ? (@ != 3)'`	`2, 4, 2, 0.5`
		`'$.a[*] ? (@ != "b")'`	`"c"`
		`'$.e ? (@ != 3)'`	(none)
		`'$.e != 3'`	`false`
`! boolean`	boolean NOT	`'$.a[*] ? (!(@ == 4))'`	`2, 2, 0.5`
		`'$.a[*] ? (!(@ == 3))'`	`2, 4, 2, 0.5`
		`'$.a[*] ? (!(@ == "b"))'`	`"c"`
		`'!($.e == 3)'`	`true`
`string like_regex string`	regex match	`'$.d[*] ? (@ like_regex "[hi]$")'`	`"gh"`
`string starts with string`	prefix match	`'$.d[*] ? (@ starts with "e")'`	`"ef"`
`number . ceiling()`	math ceiling	`'$.a[*] ? (@.ceiling())'`	`2, 4, 2, 1`
`value . type()`	JSON type	`'$.a[*] ? (@.type() == "string")'`	`"b", "c"`

B.2 Columns to keys

For each GIN opclass, I give examples of converting the column I'm indexing to the index key(s) I'm storing.

opclass	column	keys
`tsvector_ops`	`to_tsvector('simple', 'fo qu ba fo ba')`	`"fo"`, `"qu"`, `"ba"`
`array_ops`	`ARRAY[1, 2, 3, 2]`	1, 2, 3
`jsonb_ops`	`'{"a":"b", "c":{"d":[-1,[5.2]], "c":"b"}}'`	`"\001a"`, `"\005b"`, `"\001c"`, `"\001d"`, `"\004-1"`, `"\0045.2"`
	`'{"a":{}, "b":[]}'`	`"\001a"`, `"\001b"`
	`'[20, 20.0, 20.000]'`	`"\00420"`
`jsonb_path_ops`	`'{"a":"b", "c":{"d":[-1,[5.2]], "c":"b"}}'`	2076393154, 3631049813, 3671652104, 3705026877
	`'{"a":{}, "b":[]}'`	(none)
	`'[20, 20.0, 20.000]'`	805562689
`jsonb_full_ops`	`'{"a":"b", "c":{"d":[-1,[5.2]], "c":"b"}}'`	`"\Ka\Sb"`, `"\Kc\Kd\A\N-1"`, `"\Kc\Kd\A\A\N5.2"`, `"\Kc\Kc\Sb"`
	`'{"a":{}, "b":[]}'`	`"\Ka\K"`, `"\Kb\A"`
	`'[20, 20.0, 20.000]'`	`"\A\N20"`

To interpret `"\001"` characters, see gin flags in the constants section. `"\A"` characters are similar, but they map to bytes not found in any valid UTF-8 string so that there are no ambiguities. For example, if `"\K"` mapped to `"\001"`, it would be ambiguous whether, upon seeing `"\001"`, a key ends or continues with a literal `"\001"` character. Since numbers are always values, there are no ambiguity issues with them, thankfully. `"\A"` represents an array nest level; `"\K"`, a key; `"\N"`, a number; `"\S"`, a string.

B.3 Operators to primitives

You can represent each GIN operator using the primitives.

tsvector operators to primitives:

signature	translation	notes
`tsvector @@ tsquery`		
$\hookrightarrow$ `tsquery: |`	Given `col @@ 'foo | bar'`, $T_l = \mathsf{get}_I(i \overset{?}{=} \texttt{foo})$ $T_r = \mathsf{get}_I(i \overset{?}{=} \texttt{bar})$ $T = \mathsf{or}(T_l, T_r)$	
$\hookrightarrow$ `tsquery: &`	Given `col @@ 'foo & bar'`, $T_l = \mathsf{get}_I(i \overset{?}{=} \texttt{foo})$ $T_r = \mathsf{get}_I(i \overset{?}{=} \texttt{bar})$ $T = \mathsf{and}(T_l, T_r)$	
$\hookrightarrow$ `tsquery: !`	Given `col @@ '!foo'`, $T = \mathsf{get}_I(i \overset{?}{=} \texttt{foo})$ seqscan tuples where pkey $p \notin T[\texttt{pk}]$	expensive; mem $\propto$ prevalence of RHS
$\hookrightarrow$ `tsquery: :*`	Given `col @@ 'foo:*'`, $T = \mathsf{get}_I(i \overset{?}{\text{starts with}}\ \texttt{foo})$	need range-partition

anyarray operators to primitives:

signature	translation	notes
`anyarray && anyarray`	Given `col && ARRAY[3, 7]`, $T_3 = \mathsf{get}_I(i \overset{?}{=} \mathsf{3})$ $T_7 = \mathsf{get}_I(i \overset{?}{=} \mathsf{7})$ $T = \mathsf{or}(T_3, T_7)$	
`anyarray <@ anyarray`	Given `col <@ ARRAY[3, 7]`, $T_3 = \mathsf{get}_I(i \overset{?}{=} \mathsf{3})$ $T_7 = \mathsf{get}_I(i \overset{?}{=} \mathsf{7})$ $T = \mathsf{or}(T_3, T_7)$ seqscan tuples where pkey $p \in T[\texttt{pk}]$ and recheck the operator	likely cheap: cost $\propto$ size of RHS
`anyarray = anyarray`	Given `col = ARRAY[3, 7]`, $T_3 = \mathsf{get}_I(i \overset{?}{=} \mathsf{3})$ $T_7 = \mathsf{get}_I(i \overset{?}{=} \mathsf{7})$ $T = \mathsf{and}(T_3, T_7)$ seqscan tuples where pkey $p \in T[\texttt{pk}]$ and recheck the operator	likely cheap; mem $\propto$ size of RHS
`anyarray @> anyarray`	Given `col @> ARRAY[3, 7]`, $T_3 = \mathsf{get}_I(i \overset{?}{=} \mathsf{3})$ $T_7 = \mathsf{get}_I(i \overset{?}{=} \mathsf{7})$ $T = \mathsf{and}(T_3, T_7)$	

jsonb operators to primitives (using `jsonb_full_ops` opclass):

signature	translation	notes
`jsonb ? text`	Given `col ? 'foo'`, $T = \mathsf{get}_I(i \overset{?}{\text{starts with}} \texttt{Kfoo})$	
`jsonb ?& text[]`	Given `col ?& ARRAY['foo', 'bar']`, $T_l = \mathsf{get}_I(i \overset{?}{\text{starts with}} \texttt{Kfoo})$ $T_r = \mathsf{get}_I(i \overset{?}{\text{starts with}} \texttt{Kbar})$ $T = \mathsf{and}(T_l, T_r)$	
`jsonb ?| text[]`	Given `col ?| ARRAY['foo', 'bar']`, $T_l = \mathsf{get}_I(i \overset{?}{\text{starts with}} \texttt{Kfoo})$ $T_r = \mathsf{get}_I(i \overset{?}{\text{starts with}} \texttt{Kbar})$ $T = \mathsf{or}(T_l, T_r)$	
`jsonb @> jsonb`	Given `col @> '{"a":3, "b":{"c":["d"]}}'`, $T_1 = \mathsf{get}_I(i \overset{?}{=} \texttt{KaN3})$ $T_2 = \mathsf{get}_I(i \overset{?}{=} \texttt{KbKcAKd})$ $T = \mathsf{or}(T_1, T_2)$	
`jsonb @? jsonpath`		
`jsonb @@ jsonpath`		
$\hookrightarrow$ `jsonpath: ==`	Given `col @@ '$.foo == 7'`, $T = \mathsf{get}_I(i \overset{?}{=} \texttt{KfooN7})$	
$\hookrightarrow$ `jsonpath: >`	Given `col @@ '$.foo > 7'`, $T = \mathsf{get}_I((i \overset{?}{\text{starts with}} \texttt{KfooN}) \wedge (\text{val}(i) \overset{?}{>} 7))$	need simple pushdown; need range-partition
$\hookrightarrow$ `jsonpath: !=`	Given `col @@ '$.foo != 7'`, $T = \mathsf{get}_I(i \overset{?}{\text{starts with}} \texttt{KfooN})$ seqscan tuples where pkey $p \in T[\texttt{pk}]$ and recheck the operator	need range-partition
$\hookrightarrow$ `jsonpath: !`	Given `col @@ '! ($.foo == 7)'`, $T = \mathsf{get}_I(i \overset{?}{=} \texttt{KfooN7})$ seqscan tuples where pkey $p \notin T[\texttt{pk}]$	expensive; mem $\propto$ prevalence of RHS
$\hookrightarrow$ `jsonpath: like_regex`	Given `col @@ '$.foo like_regex "bar$"'`, $T = \mathsf{get}_I(i \overset{?}{\text{starts with}} \texttt{KfooS})$ seqscan tuples where pkey $p \in T[\texttt{pk}]$ and recheck the operator	need range-partition
$\hookrightarrow$ `jsonpath: starts with`	Given `col @@ '$.foo starts with "b"'`, $T = \mathsf{get}_I(i \overset{?}{\text{starts with}} \texttt{KfooSb})$	need range-partition
$\hookrightarrow$ `jsonpath: ceiling`	Given `col @@ '$.foo.ceiling() == 5'`, $T = \mathsf{get}_I((i \overset{?}{\text{starts with}} \texttt{KfooN}) \wedge (\lceil \text{val}(i) \rceil \overset{?}{=} 5))$	need simple pushdown; need range-partition
$\hookrightarrow$ `jsonpath: type`	Given `col @@ '$.foo.type() == "number"'`, $T = \mathsf{get}_I(i \overset{?}{\text{starts with}} \texttt{KfooN})$	need range-partition

Notes:

- For lax jsonpath, more scans may be needed to handle array nest levels
- I make GIN flags ASCII readable: A for array, K for key, N for number, S for string

C Examples

C.1 Example: tsvector

Here is an example of using a tsvector GIN index. It is inspired by a habr blog. Run on upstream postgres.

```sql
CREATE TABLE docs (
    doc text,
    ts tsvector GENERATED ALWAYS AS (to_tsvector('simple', doc)) STORED);
INSERT INTO docs (doc) VALUES
  ('Can a sheet slitter slit sheets?'),
  ('How many sheets could a sheet slitter slit?'),
  ('I slit a sheet, a sheet I slit.'),
  ('Upon a slitted sheet I sit.'),
  ('Whoever slit the sheets is a good sheet slitter.'),
  ('I am a sheet slitter.'),
  ('I slit sheets.'),
  ('I am the sleekest sheet slitter that ever slit sheets.'),
  ('She slits the sheet she sits on.');
SELECT * FROM docs; -- what tsvector looks like
```

```sql
CREATE INDEX ON docs USING GIN (ts);
SET enable_seqscan = OFF;
EXPLAIN SELECT * FROM docs
    WHERE ts @@ to_tsquery('simple', 'many'); -- this is index scan
```

Example of what can be done, all using the index:

```sql
SELECT doc FROM docs WHERE ts @@ to_tsquery('simple', 'many & slitter');
SELECT doc FROM docs WHERE ts @@ to_tsquery('simple', 'many | slitter');
SELECT doc FROM docs WHERE ts @@ to_tsquery('simple', 'slit:* & !slit');
SELECT ts_rank(ts, to_tsquery('simple', 'i & sheet:* & slit:*')) as rank, doc
    FROM docs
    WHERE ts @@ to_tsquery('simple', 'i & sheet:* & slit:*')
    ORDER BY rank DESC;
```

C.2 Example: jsonb

Here is an example of using a jsonb GIN index.

```sql
CREATE TABLE records (p SERIAL PRIMARY KEY, j jsonb);
INSERT INTO records (j) VALUES
  ('{"a": 1}'),
  ('{"b": {"c": "d"}}'),
  ('{"b": {"c": "e"}}'),
  ('{"b": [1, [2, 3], 4], "c": "f"}');
CREATE INDEX ON records USING GIN (j jsonb_ops);
SET enable_seqscan = OFF;
EXPLAIN SELECT * FROM records WHERE j @> '{"b": {}}'; -- this is index scan
```

Example of what can be done, all using the index:

```sql
SELECT * FROM records WHERE j @> '{"b": {}}';
SELECT * FROM records WHERE j ? 'c';
SELECT * FROM records WHERE j ?| ARRAY['a', 'c'];
SELECT * FROM records WHERE j ?& ARRAY['c', 'b'];
SELECT * FROM records WHERE j @? '$.b[*] ? (@ > 3)';
SELECT * FROM records WHERE j @@ '$.b.c == "e"';
```

You can think of

```
SELECT * FROM records WHERE j @? '$.b[*] ? (@ == 3)';
```

to be like

```
SELECT * FROM records WHERE (
    j @@ '$.b[0] == 3' or
    j @@ '$.b[1] == 3' or
    j @@ '$.b[2] == 3');
```

D Advanced material

These are some more involved details that can be helpful to developers.

D.1 Read and write path extended

D.1.1 Write path extended

`INSERT INTO table_with_gin_index (to_tsvector('simple', 'the quick brown'))` does

1. insert to main table (`ExecInsert`, `table_tuple_insert`)
2. insert to gin index (`ExecInsert`, `ExecInsertIndexTuples`)
   a. if fast update is enabled, write index tuples to pending list (`gininsert`, `ginHeapTupleFastInsert`)
   b. otherwise, write index tuples to disk (`gininsert`, `ginHeapTupleInsert`)
      i. extract deduped index keys: `the`, `quick`, `brown` (`ginHeapTupleInsert`, `ginExtractEntries`)
      ii. if tuple with key already exists, append the indexed table ctid to the posting list (`ginEntryInsert`, `addItemPointersToLeafTuple`)
      iii. otherwise, create a posting list containing just the indexed table ctid (`ginEntryInsert`, `buildFreshLeafTuple`)

D.1.2 Read path extended

`SELECT * FROM table_with_gin_index WHERE tscol @@ to_tsquery('simple', 'the | fox')` does

1. extract scan keys: `the` (`gingetbitmap`, `ginNewScanKey`)
2. get blocks from pending list (`gingetbitmap`, `scanPendingInsert`)
3. get blocks from disk (`gingetbitmap`, `startScan`)
4. get tuples from blocks (`BitmapHeapNext`, `table_scan_bitmap_next_tuple`)
5. recheck tuple if needed (`BitmapHeapNext`, `ExecQualAndReset`)

D.2 GDB tips

I walk through how I explore the GIN code using GDB.

D.2.1 Setup for GDB

First, clone the upstream postgres repository `git://git.postgresql.org/git/postgresql.git`. I checkout the `REL_13_2` tag.

Build the code.

```
./configure --enable-debug --prefix=/path/to/postgres/build
make
make install
```

Start a postgres cluster.

```
cd /path/to/postgres/build
bin/pg_ctl -D data -l logfile initdb
bin/pg_ctl -D data -l logfile start
bin/psql -d postgres
```

Connect GDB to the postgres backend.

```
ps -fC postgres # look for line like "postgres: username postgres [local] idle"
gdb -p <pid>
```

To avoid cost optimizations choosing sequential scan over index scan, turn off sequential scan:

```
SET enable_seqscan = off;
```

D.2.2 Viewing scan entries

Scan entries are formed in `ginNewScanKey` called by `gingetbitmap`. Put a breakpoint after that line:

```
b ginget.c:1930
```

Run a select that exercises the index.

```
CREATE TABLE tstab (tsv tsvector);
CREATE INDEX ON tstab USING gin (tsv);
SELECT * FROM tstab WHERE tsv @@ to_tsquery('abc');
```

When the breakpoint hits, you'll have `IndexScanDesc scan` loaded with the scan entries. Three key things to observe are

```
p ((GinScanOpaque)scan->opaque)->nkeys
p ((GinScanOpaque)scan->opaque)->keys[0].nuserentries
p *(text*)((GinScanOpaque)scan->opaque)->keys[i].scanEntry[j]->queryKey
```

where `i` and `j` should vary accordingly. In this case, there should be one key and one entry whose key is `abc`.

The same can be done for other opclasses:

```
CREATE TABLE jbtab (jb jsonb);
CREATE INDEX ON jbtab USING gin (jb);
SELECT * FROM jbtab WHERE jb @> '{"def":"ghi"}';
```

In this case, there should be one key and two entries whose keys are `\001def` and `\005ghi`. The first byte contains flags that, in this case, tell that the type is key and string, respectively.

If we change the opclass, the scan entries should change.

```
DROP INDEX jbtab_jb_idx;
CREATE INDEX ON jbtab USING gin (jb jsonb_path_ops);
SELECT * FROM jbtab WHERE jb @> '{"def":"ghi"}';
```

In this case, there should be one key and one entry whose key is formatted as an unsigned 32-bit integer. Therefore, don't cast as `text*` this time:

```
p ((GinScanOpaque)scan->opaque)->keys[0].scanEntry[0]->queryKey
```

You should get 903080546. This is can be derived as follows:

```
p $def = hash_bytes("def", 3)
p $def_rotate = ($def << 1 ) | ($def >> 31)
p $ghi = hash_bytes("ghi", 3)
p $defghi = $def_rotate ^ $ghi
```

See `JsonbHashScalarValue` for how this derivation is done.

D.2.3 Viewing JsonPathGinNode

When running queries that have jsonpath in the condition, a `JsonPathGinNode` is internally formed. You can see it by setting a breakpoint after extracting it:

```
b jsonb_gin.c:775
```

Run some query with jsonpath.

```
CREATE TABLE jpdemo (jb jsonb);
CREATE INDEX ON jpdemo USING gin (jb);
SELECT * FROM jpdemo WHERE jb @@ 'strict $.abc == "foo"';
```

Generally, you want to look at

```
p node->type
p node->val.nargs
p *((text*)node->args[i]->val.entryDatum)
```

where `i` should vary accordingly. In this case, there's a top node of type `JSP_GIN_AND` with two arguments, both of type `JSP_GIN_ENTRY`. The first has value `\001abc`, and the second has value `\005foo`.

When doing a lax query, notice a change in the structure. The second argument becomes a `JSP_GIN_OR` with two arguments `\001foo` and `\005foo`. This is to handle arrays, illustrated as follows:

```
INSERT INTO jpdemo VALUES ('{"abc": ["bar", "foo"]}');
SELECT * FROM jpdemo WHERE jb @@ 'lax $.abc == "foo"';
```

D.3 Execution trees

D.3.1 Read

```
exec_simple_query
  PortalStart
    ExecutorStart
      standard_ExecutorStart
        InitPlan
          ExecInitNode
            ExecInitBitmapHeapScan
              ExecInitNode
                ExecInitBitmapIndexScan
                  index_beginscan_bitmap
                    index_beginscan_internal
                      ambeginscan
  PortalRun
    PortalRunSelect
      ExecutorRun
        standard_ExecutorRun
          ExecutePlan
            ... ExecBitmapHeapScan
              ExecScanFetch
                BitmapHeapNext
                  MultiExecProcNode
                    MultiExecBitmapIndexScan
                      index_getbitmap
                        gingetbitmap
                    MultiExecBitmapAnd
                    MultiExecBitmapOr
```

Entry point for using text search functions:

```
(gdb) bt
#0  TS_execute (curitem=0x1749d90, arg=arg@entry=0x7ffdf8c8ea50, flags=flags@entry=2,
↪   chkcond=chkcond@entry=0x842ad0 <checkcondition_gin>) at tsvector_op.c:1848
#1  0x00000000008430c3 in gin_tsquery_triconsistent (fcinfo=<optimized out>) at
↪   tsginidx.c:287
#2  0x0000000000881fdd in FunctionCall7Coll (flinfo=0x17ffe98, collation=<optimized
↪   out>, arg1=<optimized out>, arg2=<optimized out>, arg3=<optimized out>,
↪   arg4=<optimized out>, arg5=25151648, arg6=25151592, arg7=25151768) at fmgr.c:1311
#3  0x000000000049dc10 in directTriConsistentFn (key=<optimized out>) at ginlogic.c:97
#4  0x000000000049c39f in startScanKey (ginstate=0x17fe580, so=0x17fe578, so=0x17fe578,
↪   key=0x17fc648) at ginget.c:566
#5  startScan (scan=0x17e73f0, scan=0x17e73f0) at ginget.c:642
#6  gingetbitmap (scan=0x17e73f0, tbm=0x18053d8) at ginget.c:1951
#7  0x00000000004d3d9a in index_getbitmap (scan=scan@entry=0x17e73f0,
↪   bitmap=bitmap@entry=0x18053d8) at indexam.c:671
#8  0x0000000000632882 in MultiExecBitmapIndexScan (node=0x17e7100) at
↪   nodeBitmapIndexscan.c:105
#9  0x00000000006220e1 in MultiExecProcNode (node=<optimized out>) at execProcnode.c:510
#10 0x0000000000631f50 in BitmapHeapNext (node=node@entry=0x17e6e10) at
↪   nodeBitmapHeapscan.c:113
#11 0x00000000006245fa in ExecScanFetch (recheckMtd=0x6321c0 <BitmapHeapRecheck>,
↪   accessMtd=0x631820 <BitmapHeapNext>, node=0x17e6e10) at execScan.c:133
#12 ExecScan (node=0x17e6e10, accessMtd=0x631820 <BitmapHeapNext>, recheckMtd=0x6321c0
↪   <BitmapHeapRecheck>) at execScan.c:199
#13 0x000000000061af52 in ExecProcNode (node=0x17e6e10) at
↪   ../../../src/include/executor/executor.h:248
#14 ExecutePlan (execute_once=<optimized out>, dest=0x17f5ba8, direction=<optimized
↪   out>, numberTuples=0, sendTuples=true, operation=CMD_SELECT,
↪   use_parallel_mode=<optimized out>, planstate=0x17e6e10, estate=0x17e6be8) at
↪   execMain.c:1646
#15 standard_ExecutorRun (queryDesc=0x17f9738, direction=<optimized out>, count=0,
↪   execute_once=<optimized out>) at execMain.c:364
#16 0x0000000000770afb in PortalRunSelect (portal=portal@entry=0x178b008,
↪   forward=forward@entry=true, count=0, count@entry=9223372036854775807,
↪   dest=dest@entry=0x17f5ba8) at pquery.c:912
#17 0x0000000000771d68 in PortalRun (portal=portal@entry=0x178b008,
↪   count=count@entry=9223372036854775807, isTopLevel=isTopLevel@entry=true,
↪   run_once=run_once@entry=true, dest=dest@entry=0x17f5ba8,
↪   altdest=altdest@entry=0x17f5ba8, qc=qc@entry=0x7ffdf8c8
efe0) at pquery.c:756
#18 0x000000000076dabe in exec_simple_query (query_string=0x1724c68 "SELECT doc FROM
↪   docs WHERE ts @@ to_tsquery('simple', 'many & slitter');") at postgres.c:1239
#19 0x000000000076ee37 in PostgresMain (argc=<optimized out>,
↪   argv=argv@entry=0x174f0b8, dbname=0x174f000 "testupdatejoin", username=<optimized
↪   out>) at postgres.c:4315
#20 0x0000000000481e23 in BackendRun (port=<optimized out>, port=<optimized out>) at
↪   postmaster.c:4536
#21 BackendStartup (port=0x1748270) at postmaster.c:4220
#22 ServerLoop () at postmaster.c:1739
```

```
#23 0x00000000006fc793 in PostmasterMain (argc=argc@entry=3, argv=argv@entry=0x171f980)
↪  at postmaster.c:1412
#24 0x0000000000482a6e in main (argc=3, argv=0x171f980) at main.c:210
```

D.4 Constants

For `jsonb_ops` GIN opclass, the one-byte flag at the beginning of each GIN key:

```
#define JGINFLAG_KEY    0x01    /* key (or string array element) */
#define JGINFLAG_NULL   0x02    /* null value */
#define JGINFLAG_BOOL   0x03    /* boolean value */
#define JGINFLAG_NUM    0x04    /* numeric value */
#define JGINFLAG_STR    0x05    /* string value (if not an array element) */
#define JGINFLAG_HASHED 0x10    /* OR'd into flag if value was hashed */
#define JGIN_MAXLENGTH  125     /* max length of text part before hashing */
```

`GinScanKeyData.strategy`:

```
#define JsonbContainsStrategyNumber   7
#define JsonbExistsStrategyNumber   9
#define JsonbExistsAnyStrategyNumber  10
#define JsonbExistsAllStrategyNumber  11
#define JsonbJsonpathExistsStrategyNumber   15
#define JsonbJsonpathPredicateStrategyNumber  16
```

`GinScanKeyData.searchMode`:

```
#define GIN_SEARCH_MODE_DEFAULT         0
#define GIN_SEARCH_MODE_INCLUDE_EMPTY   1
#define GIN_SEARCH_MODE_ALL             2
#define GIN_SEARCH_MODE_EVERYTHING      3   /* for internal use only */
```