

GIN, BTREE_GIN, GIST, BTREE_GIST, HASH & BTREE indexes on JSONB data

Josef Machytka <josef.machytka@netapp.com>
NetApp (Instaclustr-credativ part)
2024-06-27 - Swiss PGDay 2024



Josef Machytka

- Professional Service Consultant - PostgreSQL specialist at NetApp (Instaclustr-credativ part).
- *Credativ* - support for open source: PostgreSQL, Patroni, pgBackRest, Kubernetes pg ops, monitoring.
- *Instaclustr* - PostgreSQL/Cassandra/Kafka cloud solutions.
- *NetApp* - enterprise-grade cloud storage with protection against ransomware attacks, with a focus on AI.

- 30+ years of experience with different databases.
- PostgreSQL (11y), BigQuery (7y), Oracle (15y), MySQL (12y), Elasticsearch (5y), MS SQL (5y).
- DB admin/developer, Data ingestion platforms, Data analysis, Business intelligence, Monitoring.
- Originally from Czechia, currently living in Berlin for 11 years.
- LinkedIn: www.linkedin.com/in/josef-machytka.

Table of contents

- Problems with implementation
- What was tested

- GIN indexes
- BTREE_GIN extension
- GIST & BTREE_GIST indexes
- HASH indexes
- BTREE indexes

- Decomposition of JSON data
- Statistics

Problems with implementation of JSON data



JSON - light and dark side of the force

- Frontend and backend developers love the flexibility of JSON.
- JSON minimizes the need for app changes due to schema changes.
- IoT devices use JSON - W3C Web of Things Working Group standardized JSON for IoT.

- Data quality checks - absolute freedom might be a big challenge.
- Problems with data cleansing and transformation.
- Business intelligence, ML, and reporting need structured and standardized data.

- But the full decomposition of JSON can be a complex and painful task.
- Databases must handle JSON data, there is no escape.

Clients struggle with implementing JSONB

- Articles are often too shallow, repeating documentation.
- Very trivial examples - create a table, insert 3 rows, try explain, celebrate.
- Even ChatGPT-4o is not helpful with deeper and more complicated topics.

- Clients develop with small inadequate datasets.
- Tests are often too simple, just guessing production use cases.
- PostgreSQL dev instance has inadequate configuration.

- Confusion about TOAST tables, compression, and storage.
- Doubts about design - partitions vs 1 big table.
- Developers are obsessed with forcing indexes.

What was tested



What was tested

- Different types of indexes for different use cases.
- Different compression and storage methods.
- One big table vs partitioned tables.

- Influence of PostgreSQL settings - memory, costs.
- Performance under different loads - multiple simultaneous sessions.
- Influence of parallelism.
- Influence of data distribution and selectivity.

- User defined statistics on JSONB columns.
- Full decomposition vs one big JSONB column.
- Deep dive into internals of indexes. Analysis of code.

Dataset for tests

- GitHub Archive events - www.gharchive.org
- Separate .gz files for each hour - YYYY-MM-DD-HH24.json.gz
- One big JSONB column with all the data

```
CREATE TABLE github_events (  
    id SERIAL PRIMARY KEY NOT NULL,  
    jsonb_data JSONB);
```

GitHub events - JSON record

```
{  "id": "26167585827",
  "repo": {    "id": 581592468,
              "url": "https://api.github.com/repos/tiwabs/tiwabs_audio_door_tool",
              "name": "tiwabs/tiwabs_audio_door_tool" },
  "type": "PushEvent",
  "actor": {  "id": 48737497,
              "url": "https://api.github.com/users/tiwabs",
              "login": "tiwabs",
              "avatar_url": "https://avatars.githubusercontent.com/u/48737497?",
              "gravatar_id": "",
              "display_login": "tiwabs" },
  "public": true,
  "payload": {"ref": "refs/heads/master",
              "head": "3ca247941f269bcedeb17e5b12e9b3b74b1c4da2",
              "size": 1,
              "before": "0dd5471667b12084b8fc88b1bca299780382d50a",
              "commits":
                [
                  {  "sha": "3ca247941f269bcedeb17e5b12e9b3b74b1c4da2",
                    "url": "https://api.github.com/repos/tiwabs/...12e9b3b74b1c4da2",
                    "author": { "name": "Tiwabs", "email": "mrskielz@gmail.com" },
                    "message": "fix(export): export nametable if export succed",
                    "distinct": true }
                ],
              "push_id": 12149772587,
              "distinct_size": 1 },
  "created_at": "2023-01-01T13:39:55Z" }
```

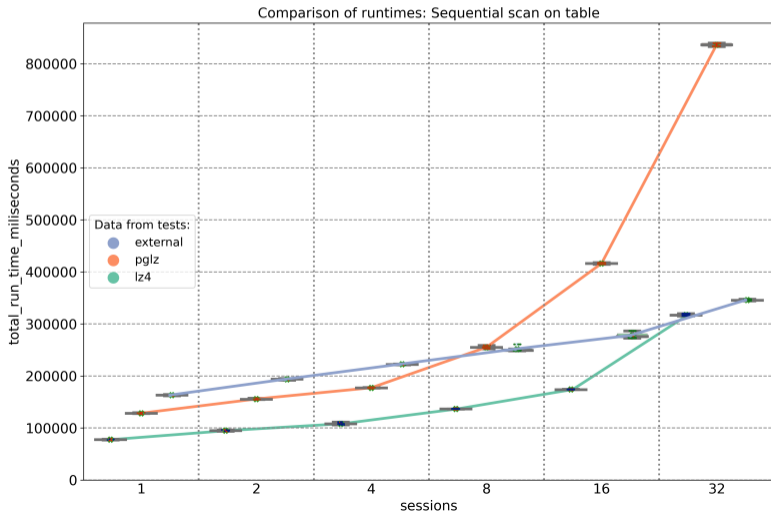
GitHub events - testing details

- Tested in PostgreSQL 15 and 16.
- Python scripts for downloading, importing, analyzing, and testing.
- Multiple local and AWS RDS testing environments.
- Different CPUs, all with 8 cores and 32 GB RAM.
- Used 1 week of data from January 2023.
- In total 17,474,101 rows.
- 3 tables, different compression methods:
 - pglz: 41 GB
 - lz4: 38 GB
 - external storage with no compression: 98 GB



Performance testing - sequential scan on the table

- Aggregation query over all records using sequential scan on the table, without parallelism.
- The old compression method, pglz, was already slower than no compression with 8 sessions on 8 cores.
- With 16, 32, and 64 sessions on 8 cores, pglz became a serious performance bottleneck.



Disk intensive operations - throughput matters

- The same problem occurs on ALL clouds; we just tested it on AWS.
- On AWS RDS SSD 300GB with 3,000 IOPS, the throughput of 125 MiBps was a real disaster.
- All disk-intensive operations were 4x to 5x slower than on the local PC.
- With SSD 500GB and 12,000 IOPS, and a throughput of 500 MiBps, we finally got reasonable results.
- But auto-scaling of the disk can also slow down your actions by 5x or more.
- Never try to save money on a cloud instance by using a slow, small disk.
- Do not rely only on the auto-scaling of the disk.

GIN indexes



GIN indexes

- GIN indexes generally showed very stable performance even under high load.
- But for their usage proper settings are crucial.
- Set `Shared_buffers` to 25% of RAM and `effective_cache_size` to around 50% of RAM.

- GIN indexes do not support parallelism, neither for creation nor for usage.
- Parallelism can be a significant factor in using or not using GIN indexes.

- If parallel workers are available, the planner can choose parallel sequential scan on the table.
- If all parallel workers are in use, the planner uses GIN indexes for new queries.
- Set `Max_parallel_workers_per_gather` = 0 at least for the query.

GIN indexes - parameters tuning

- SSD: `random_page_cost = 1.1`, `effective_io_concurrency = 200`.
- Set `random_page_costs <= seq_page_cost (=1)` if the database is fully cached in memory.
- Different values of `work_mem` had minimal impact if the query used GIN index scan.

- PostgreSQL code: `src/backend/optimizer/path/costsize.c`
- `cpu_tuple_cost`, `cpu_index_tuple_cost`, `cpu_operator_cost` ???
- `parallel_setup_cost`, `parallel_tuple_cost` ???

- The code says "measured on an arbitrary scale".
- Especially `cpu_tuple_cost` is used incredibly often in the code.
- Its value influences the planner's decisions significantly.

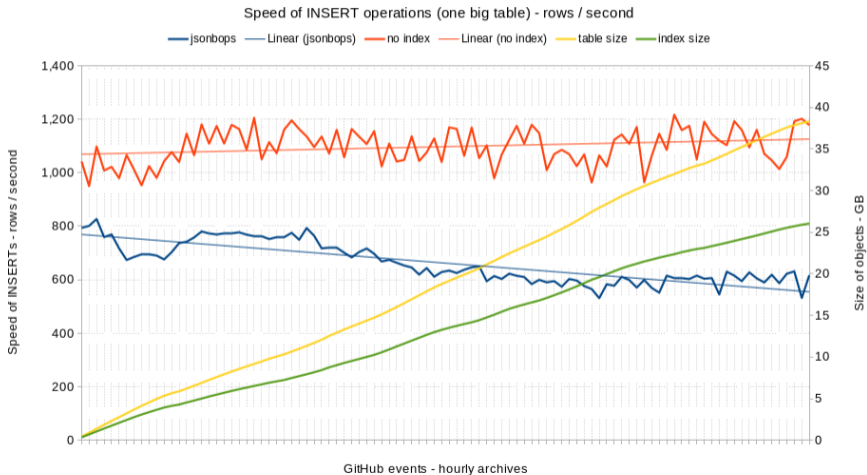
GIN indexes - updates of the index

- It can take hours to create a new GIN index on the whole column with existing data.
- Setting *maintenance_work_mem* has a rather small impact on the speed of creating a GIN index.
- Disk IO is the main factor affecting the speed of creating a GIN index.

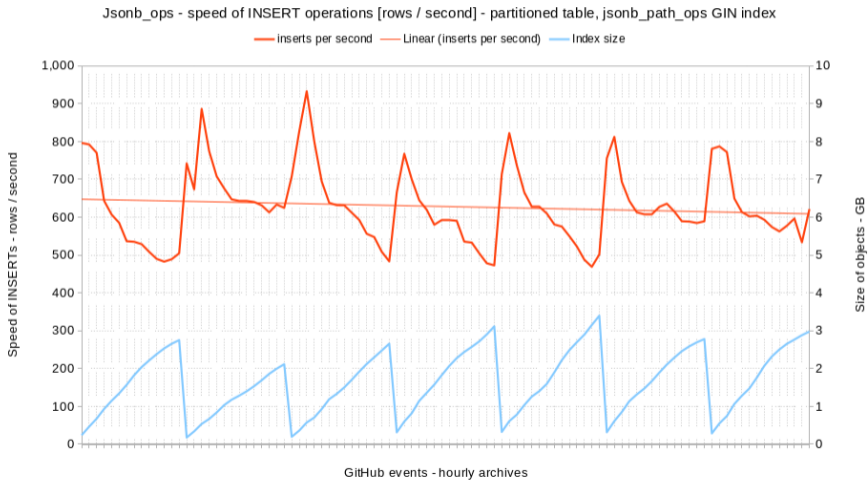
- Updates of GIN indexes become significantly slower as the table size grows.
- The index is rebuilt when the *gin_pending_list_limit* is reached or during vacuuming.
- Default value of *gin_pending_list_limit* is 4MB = 512 data pages.

- The size of the table matters.
- The speed of inserting rows per second can decrease by up to 50%.
- Partitioning can help significantly. However, disk IO is again the main factor.

GIN indexes - speed of inserts - one big table

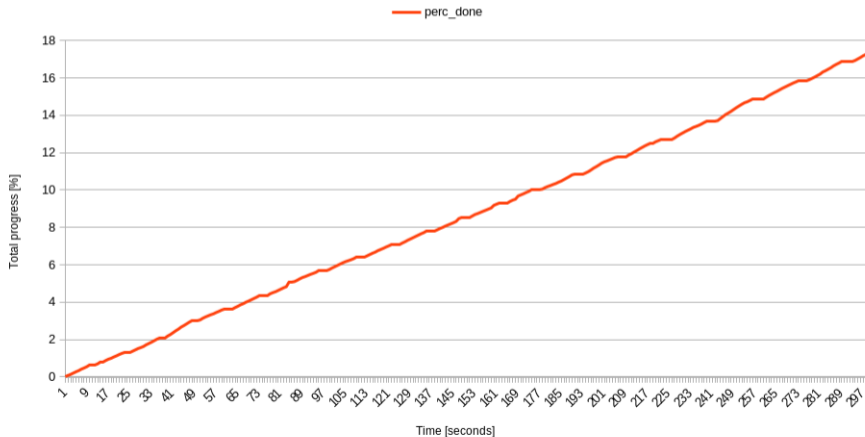


GIN indexes - speed of inserts - partitions



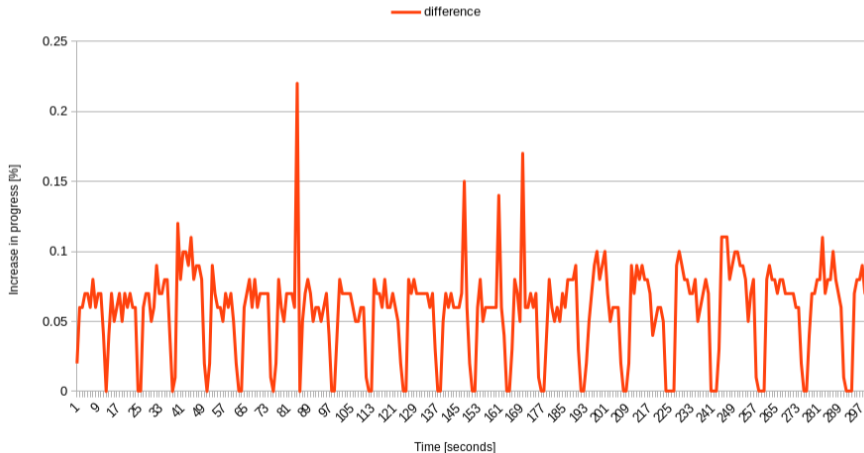
GIN indexes - creation - total progress

Creation of jsonb_ops GIN index - total progress



GIN indexes - creation - difference in progress

Creation of jsonb_ops GIN index - increase in progress



Gin Indexes inspection - pending pages and tuples

- We can use extensions to get some deeper information about GIN indexes
- pgstattuple:
 - pgstatginindex()
- pageinspect:
 - gin_page_opaque_info() - basic info about page
 - gin_metapage_info() - details for metapage
 - gin_leafpage_items() - details for leaf page

```
SELECT * FROM pgstatginindex('index_name');
```

version	pending_pages	pending_tuples
2	414	1853

```
SELECT *  
FROM gin_metapage_info(  
    get_raw_page('index_name', 0))\gx
```

```
pending_head      : 292675  
pending_tail      : 339992  
tail_free_size    : 220  
n_pending_pages   : 414  
n_pending_tuples  : 1853  
n_total_pages     : 339200  
n_entry_pages     : 312283  
n_data_pages      : 24533  
n_entries         : 52572205  
version           : 2
```

--> but before VACUUM these values are only estimates!

Gin Indexes inspection - deeper dive into pages stats

-- How to get proper count of pages?

pg_class: 339986, metapage: 339200 - both are estimates, just taken differently

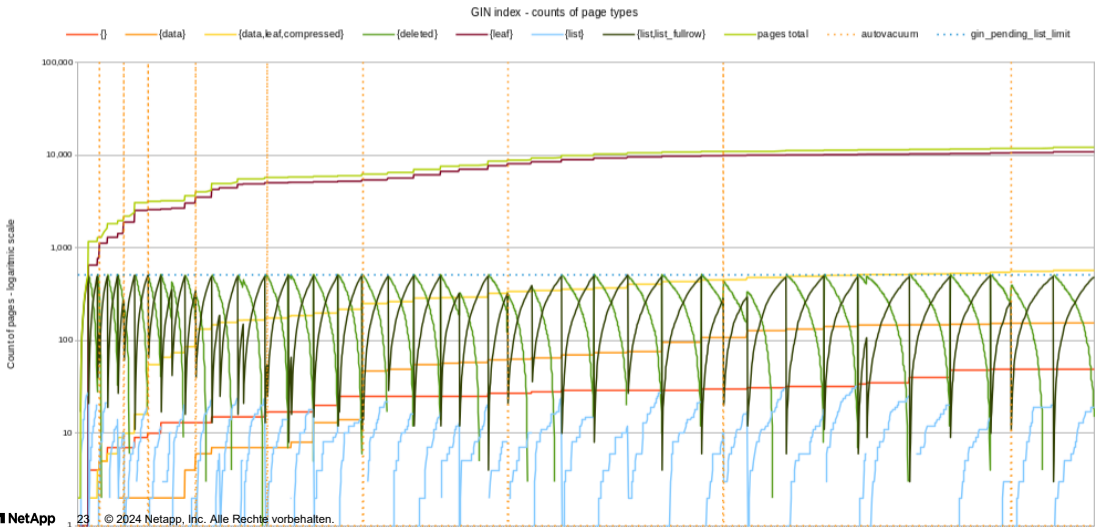
-- Let's calculate the proper count of pages from the size of data files

```
SELECT pg_relation_size('index_name') / 8192;  
-> 357105 pages
```

-- Now we can get statistics about GIN index pages

```
WITH pages AS (  
    SELECT *  
    FROM generate_series(0,  
        (SELECT pg_relation_size('index_name') / 8192) -1) as pagenum)  
SELECT  
    (SELECT flags  
     FROM gin_page_opaque_info(  
         get_raw_page('index_name', pagenum))) as flags,  
    count(*) as pages  
FROM pages GROUP BY flags ORDER BY flags;
```

GIN indexes - rebuild of index during insertion of data



GIN indexes - equality of value - operators @? and @@

- *GIN index with jsonb_ops* operator class is the most versatile but also the biggest.
- It allows searching for *equality* of values on multiple *unknown* levels of keys.
- The @? and @@ operators can be used with * and ** wildcards.
- Example: `WHERE jsonb_data @@ '$.** == "python3" '`
- The size of the jsonb_ops GIN index on the whole column can reach 80% of the table size.

- The operator class *jsonb_path_ops* works only with fully known jsonpath.
- It allows searching for *equality* of values on multiple *known* levels of keys.
- The @? and @@ operators cannot use wildcards, the jsonpath must be known.
- Example: `WHERE jsonb_data @@ '$.payload.pull_request.head.repo.topics[*] == "python3"'`
- The GIN index with jsonb_path_ops on the whole column can reach 30% of the table size.

GIN indexes - operator @>

- If the second object is contained in the first one - an exact match of the key(s) and value(s).
- Works with both operator classes.
- Works for nested objects and arrays.
- Allows searching for *equality of multiple values* in one condition.

- Searching for values from lists of values - events from specific users, a specific repository.
- Run times are in dozens or hundreds of milliseconds.
- Very stable performance even with multiple sessions running in parallel.

- Limitation - the path must be known.
- This will find data: `WHERE jsonb_data @> '{"payload":{"commits":{"author":{"name": "Jane Joy"}}}}'`
- This will not find: `WHERE jsonb_data @> '{"commits":{"author":{"name": "Jane Joy"}}}'`

GIN indexes - other operators

- Operators `?`, `?|`, and `?&`.
- They are used to look for the existence of key(s) on the *top level*.
- These operators only work with the *jsonb_ops* operator class.

- The usage of the GIN index depends on statistics.
- If a key is present in the majority of records, the GIN index is not used.
- If the table is very small, the GIN index is not used.

- The GIN index is only used for keys that are not present in the majority of records.
- Useful for a very dynamic schema or a table that stores many different JSON datasets.

GIN indexes - SQL\JSON operators and methods

- SQL\JSON contains multiple amazing methods, but GIN index does not work for them.
- `like_regex` - tests if the string value returned by jsonpath matches a regular expression:
`WHERE jsonb_data @? '$.description ? (@ like_regex ".*Michigan.*")'`
- `starts with` - tests if the string value returned by jsonpath starts with a specific string:
`WHERE jsonb_data @? '$.laureates[*].firstname ? (@ starts with "Jo")'`
- `exists` - tests if a key exists in the JSONB schema at a given level:
`WHERE jsonb_data @? '$.laureates[*].firstname ? (exists (@))'`
- The PostgreSQL community should consider creating indexes for these operators.

GIN indexes - full text search

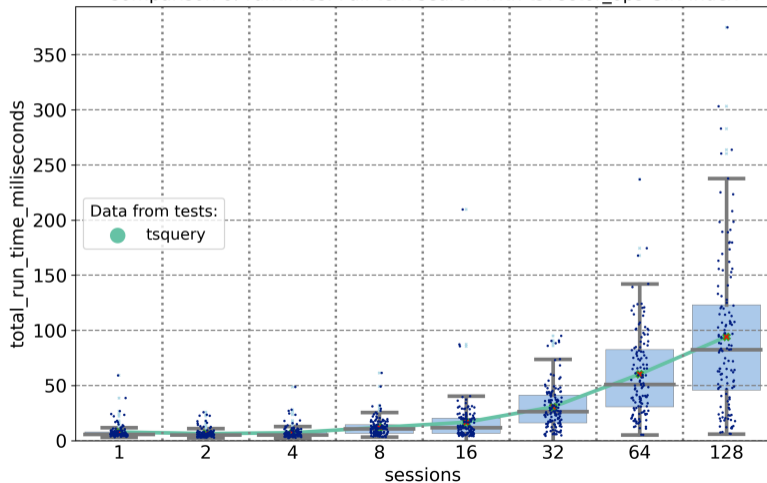
- GIN index with `tsvector_ops` operator class allows full text search.
- The function `jsonb_to_tsvector` converts JSONB data into `tsvector`.
- Example: `WHERE jsonb_to_tsvector('english', jsonb_data, 'string') @@ to_tsquery('search_string')`

- Full text search works for *equality* of words/synonyms.
- You can combine words using AND/OR.

- The `tsvector_ops` index on the whole column can be larger than the table.
- It only makes sense to create an index on free text columns.
- It speeds up search by at least 100 times.
- Performance is very stable under high load.

GIN indexes - full text search - commit messages

Comparison of runtimes: Full text search with tsvector_ops GIN index



GIN indexes - LIKE search

- *Gin_trgm_ops* operator class allows string search using LIKE.
- The index over the whole column does not distinguish keys and values.
- It still performs an *equality* search behind the scenes - *equality of trigrams*.
- Creating an index on free text columns is the only scenario where it makes sense.

- The size of the *gin_trgm_ops* GIN index on the whole column can reach 50% of the table size.
- It significantly speeds up search, even up to 1000x.
- The performance is very stable under high load.

GIN indexes - partitions

- Partitioned tables showed multiple advantages over one big table.
- Query run times using GIN indexes are faster on partitioned tables, approximately 5 times faster.
- Loading data into partitioned tables is faster.
- Updates of GIN indexes on partitions are faster.
- Partitioning also makes GIN indexes more efficient in case of very uneven data distribution.
- Bloat of partial GIN indexes due to the values with very high frequency is much lower.

BTREE_GIN indexes



BTREE_GIN indexes

- The BTREE_GIN extension combines the BTREE and GIN indexes.
- It adds GIN operator classes with BTREE behavior.
- The BTREE text_pattern_ops does not work with BTREE_GIN indexes.
- You can use any GIN operator class with the BTREE_GIN index.

- The BTREE_GIN index can have multiple columns.
- It will optimize the search for any combination of these columns.
- The order of columns does not seem to be important.

- The runtime with the BTREE_GIN index was better than with the GIN index + filter search.
- The run times of the tested use cases were in the range of hundreds of milliseconds.
- The performance was stable even with many parallel sessions.

GIST indexes



GIST indexes

- For indexing geo data, you need GIST indexes.
- Most commonly in GeoJSON format.
- Usually - Type (Point), coordinates [longitude (+/- 0-180), latitude (+/- 0-90)].

```
-- NASA meteorites dataset
```

```
{ "id": "1",  
  "fall": "Fell",  
  "mass": "21",  
  "name": "Aachen",  
  "year": "1880-01-01T00:00:00.000",  
  "reclat": "50.775000",  
  "reclong": "6.083330",  
  "nametype": "Valid",  
  "recclass": "L5",  
  "geolocation": {  
    "type": "Point",  
    "coordinates": [ 6.08333, 50.775 ] } }
```

GIST indexes - PostGIS example

- Let's create a GIST index based on GEOMETRY(point, 4326) PostGIS data type.
- EPSG code 4326 is for WGS 84 spacial reference system.

-- you can create a GIST index on a GEOMETRY column manually:

```
CREATE INDEX ON nasa_meteorits USING GIST(  
  ST_SETSRID(ST_MakePoint(  
    cast(jsonb_data->'geolocation'->'coordinates'->>0 as float),  
    cast(jsonb_data->'geolocation'->'coordinates'->>1 as float) ), 4326) );
```

-- or use PostGIS extension function st_geomfromgeojson

-- expects a GeoJSON object as input, recognizes content automatically:

-- meteorites: { "type": "Point", "coordinates": [6.08333, 50.775] }

-- earthquakes: { "geometry": { "type": "Point", "coordinates": [-104.024, 31.646, 6.8514] } }

```
CREATE INDEX ON nasa_meteorits USING GIST(  
  ST_GeomFromGeoJSON(jsonb_data->'geolocation') );
```

GIST indexes - BTREE_GIST extension

- The BTREE_GIST extension allows you to combine GIST and BTREE indexes.
- You cannot create a GIST index on a whole JSONB column.
- However, you can combine multiple columns into a BTREE_GIST index using different operator classes.

- The intarray extension implements the gist__int_ops and gist__intbig_ops operator classes for arrays.
- There is the gist_trgm_ops operator class for performing LIKE search over strings.
- And the tsvector_ops operator class for creating a GIST index for full-text search.

GIST indexes - BTREE_GIST extension

- Earthquakes dataset - United States Geological Survey (earthquake.usgs.gov).
- GIST index on JSONB column combining multiple extracted values.
- Geolocation, magnitude as a number, place as a trigram, and magnitude type as a list of values.
- Optimizes all variants of queries using these columns.
- Quick to create - 1 minute on a 1 GB dataset. Size is 20% of the table size.

```
CREATE INDEX ON jsonimport USING gist (  
  ST_GeomFromGeoJSON(jsonb_data->'geolocation'),  
  ((jsonb_data->'properties'->>'mag')::numeric),  
  (jsonb_data->'properties'->>'place') gist_trgm_ops,  
  (jsonb_data->'properties'->>'magType') );
```

HASH indexes



HASH indexes

- HASH indexes allow only equality search, not range search.
- Support only single-column indexes and do not allow uniqueness checking.
- Can be smaller and quicker than BTREE indexes, but only on some data.

- In my tests, the performance of HASH and BTREE indexes on unique data was almost the same.
- On non-unique data, HASH indexes were still very quick, but at least 5x slower than BTREE indexes.
- However, HASH indexes have been bigger in size than BTREE indexes - 2x to 3x.

- HASH indexes are extremely sensitive to the data distribution.
- *Require unique or nearly unique data* with a low number of rows per hash bucket.
- Many repeated values lead to a big number of overflow bucket pages and bad performance.

HASH indexes

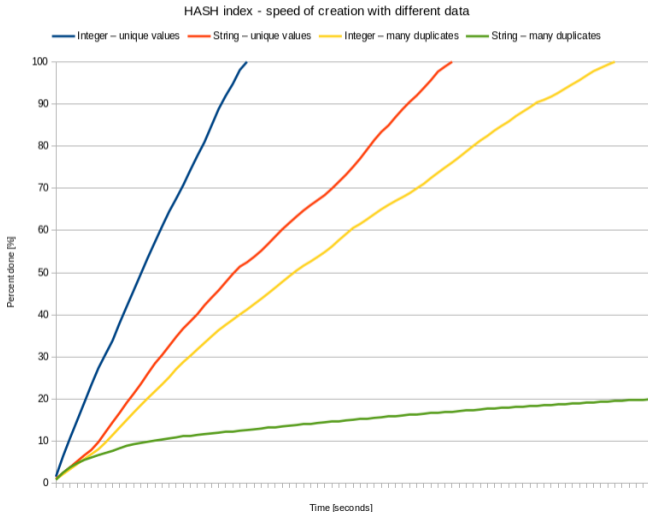
- Always check PostgreSQL statistics on the data before deciding to use a HASH index.
- For JSONB data, create user defined statistics for the keys you want to index.

- In `pg_stats`, a value of `n_distinct = -1` indicates fully unique values.
- If `n_distinct != -1`, check the values in `most_common_vals` and `most_common_freqs`.
- If the values in `most_common_freqs` are 0.1 or higher, a HASH index is not a good choice.
- You can avoid NULLs and empty strings by using a partial index.

HASH index - creation / reindex

- Numbers are processed quicker than strings.
- Unique data is processed very quickly.
- Non-unique strings with many duplicates are processed extremely slowly.

- Unique integers: 30 seconds, highly duplicate integers: 1.5 minute.
- Unique strings: 1 minute, highly duplicate strings: 55 minutes.



HASH indexes - internal structure

- Extension *pageinspect* contains functions for inspecting the index.
- Function *hash_page_type* returns the type of the page.

- If all values are unique, index contains: metapage (0), bucket pages, bitmap page (last).
- If there are duplicates, overflow pages are added after the bitmap page.

- The bitmap pages maintain the map of free and in-use overflow pages.
- Free overflow pages, not attached to any bucket, are kept and reused.

- There are other functions to inspect content of pages.
- *Hash_metapage_info*, *hash_bitmap_info*, *hash_page_items*, *hash_page_stats*.

HASH indexes - maintenance

- HASH indexes are not automatically maintained.
- Empty overflow pages are never removed from the index.
- Shrinking requires REINDEX or VACUUM FULL.

- HASH indexes can be useful only in very specific cases.
- BTREE indexes seem to be a much better choice.

BTREE indexes



BTREE indexes

- BTREE indexes are very small and quick, making them an ideal first choice.
- They allow parallel index build and scan.
- They can be created in minutes, even on large tables.

- BTREE indexes support equality and range queries using operators such as $<$, $<=$, $=$, $>=$, and $>$.
- When combined with *text_pattern_ops* (for each column), they can be used for prefix-LIKE queries.

- Some transformations must be encapsulated into immutable functions.
- Conditions in queries must contain the exact indexed expression.

- Partial BTREE indexes can be very useful for dynamic schemas.
- Whenever possible, use LIMIT to improve the delivery of results significantly.

Sizes of indexes



Summary of results - sizes of indexes

Table - lz4 TOAST compression, 17.5 M rows	38 GB	
GIN index - jsonb_ops - whole JSONB column	25 GB	66 %
GIN index - jsonb_path_ops - whole JSONB column	16 GB	42 %
GIN index - gin_trgm_ops - whole JSONB column	16 GB	42 %
GIN index - tsvector_ops - jsonb_to_tsvector, "string" values	34 GB	90 %
GIN index - tsvector_ops - just commit messages	0.5 GB	1.5 %
GIN index - gin_trgm_ops - just commit messages	1 GB	3 %
BTREE_GIN index - 'payload' jsonb_ops + created_at	23 GB	60 %
BTREE_GIN index - 'payload' jsonb_path_ops + created_at	15 GB	40 %
BTREE index on "created_at" timestamp	120 MB	0.2 %
HASH index on "created_at" timestamp	0.5 GB	1.5 %

JSON decomposition



JSON object decomposition - test with Github events

- Github events schema has 936 unique jsonpaths.
- Some keys contain different data types in different records.
- Schema contains 12 embedded arrays with another JSONB objects -> additional 12 tables.

- Main table would have 807 columns. (Jsonpaths without array elements)
- In total: text 604, number 105, boolean 70, datetime 28 columns.
- I tried to create and fill the main table: *"row is too big: size 9088, maximum size 8160"*

- I would have to put some big nested JSONB objects into separate tables too.
- Decomposition is huge task on its own prone to multiple additional errors.

JSON object decomposition

- A table with hundreds of columns is hard to use.
- The theoretical limit is 1600 columns in the tuple.
- However, the tuple must fit into one data page (8KB).

- A full jsonpath as a column name can easily exceed 63 characters.
- A table with many columns requires careful design due to data types padding.
- Columns must fit into 8-byte blocks - a 64-bit CPU reads a block of 8 bytes.

JSON object decomposition

- Nested composed data types can make the solution even more complex.
- They use extended storage, i.e. TOAST.
- This way you just convert one binary object into another.
- Queries require encapsulation of top-level keys into parentheses.
- Only after really trying it you will realize how big challenge it can be.

JSON object decomposition - use **GENERATED** columns

- Manual decomposition is not worth the trouble.
- You can use *GENERATED columns* for some frequently used jsonpaths.
- This way you avoid a lot of manual work.

```
ALTER TABLE github_events
ADD COLUMN actor_login text
GENERATED ALWAYS AS ((jsonb_data->'actor'->>'login')) STORED;
```

Statistics



Statistics for the whole JSONB object

- The planner seems to be able to deduce statistics for top-level keys.
- For specific jsonpaths use *CREATE STATISTICS* command.
- This way you will have always up-to-date statistics for your use cases.

- Command *CREATE STATISTICS* only prepares statistics object.
- Statistics are gathered by first *ANALYZE* command and later by system as usual.

Statistics for the whole JSONB object

```
CREATE STATISTICS github_actor_login
ON ( ((jsonb_data -> 'actor'::text) ->> 'login'::text) )
FROM github_events;

CREATE STATISTICS github_created_at_ts
ON ( json_datetime_to_timestamp((jsonb_data ->> 'created_at'::text)) )
FROM github_events;
```

- View `pg_stats_ext_exprs` shows statistics.
- Columns `n_distinct`, `most_common_vals`, and `most_common_freqs`, correlation as in `pg_stats`.

Understand your data and use cases

- Indexes are not the "silver bullet" for everything.
- Don't be obsessed with forcing PostgreSQL to use indexes.

- The usage of indexes depends on frequency, selectivity, and correlation.
- In some use cases, a parallel sequential scan can be better than an index scan.

- The runtime of queries depends on data distribution - sorting in memory vs on disk.
- On multi-tenant systems, things are even more complicated.

- Understand your data and use cases to use the right tools!
- Make sure that your PostgreSQL has proper settings.

- Thank you for your attention!
- Questions?

- www.netapp.com
- www.instaclustr.com
- www.credativ.de

- www.netapp.com/blog
- www.instaclustr.com/blog
- www.credativ.de/blog

