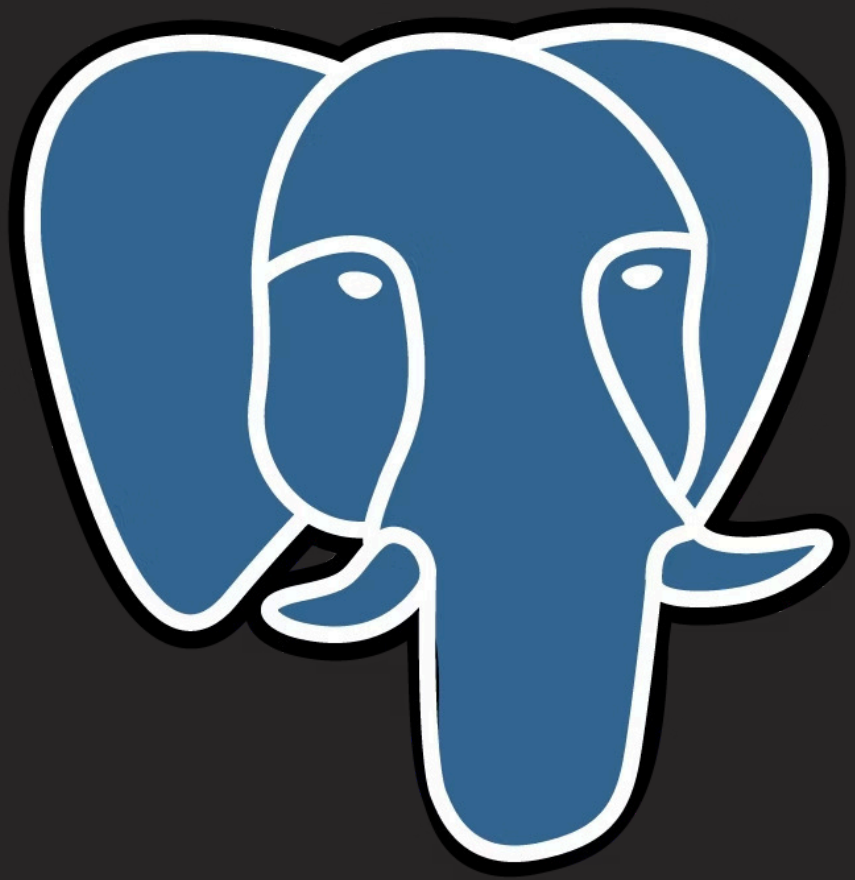
Эффективное использование SQL/JSON в PostgreSQL для обработки неструктурированных данных.



HSTORE — первый шаг к неструктурированным данным

PostgreSQL стал первой реляционной базой данных с поддержкой слабо структурированных данных, введя тип **HSTORE** в **2003** году. Этот тип представляет собой хранилище пар ключ => значение.

`k => v`

`"1-a" => "anything at all"`

HSTORE стал прообразом будущих неструктурированных типов данных.

```
postgres=# SELECT
postgres-# attr
postgres-# FROM
postgres-# books;
                                                                attr
-------------------------------------------------------------------------------------------------------
-----------------------------------------------------
 "weight"=>"13.2 ounces", "ISBN-13"=>"978-1449370000", "language"=>"English", "paperback"=>"2403",
"publisher"=>"Bantam Spectra/US & Voyager Books/UK"
 "weight"=>"14.2 ounces", "ISBN-13"=>"978-1449370001", "language"=>"English", "paperback"=>"2553",
"publisher"=>"Bantam Spectra/US & Voyager Books/UK"
 "weight"=>"15.7 ounces", "ISBN-13"=>"978-1449370002", "language"=>"English", "paperback"=>"2683",
"publisher"=>"Bantam Spectra/US & Voyager Books/UK"
(3 rows)
```

JSON в PostgreSQL

В **2012** году PostgreSQL добавил поддержку типа JSON, который хранит данные в виде текстовой строки с валидацией по стандарту RFC 4627.

Примитивные типы JSON соответствуют типам PostgreSQL, что обеспечивает удобство работы с данными в формате JSON.

Примитивный тип JSON	Тип PostgreSQL	Примечания
`string`	`text`	`\u0000` не допускается, как не ASCII символ, если кодировка базы данных не UTF8
`number`	`numeric`	Значения `NaN` и `infinity` не допускаются
`boolean`	`boolean`	Допускаются только варианты `true` и `false` (в нижнем регистре)
`null`	(нет)	`NULL` в SQL имеет другой смысл

Примеры выражений с типом JSON

Числа, строки, true, false или null

```
SELECT '5'::json;
```

Массивы с элементами разных типов

```
SELECT '[1, 2, "foo", null]'::json;
```

Объекты

```
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;
```

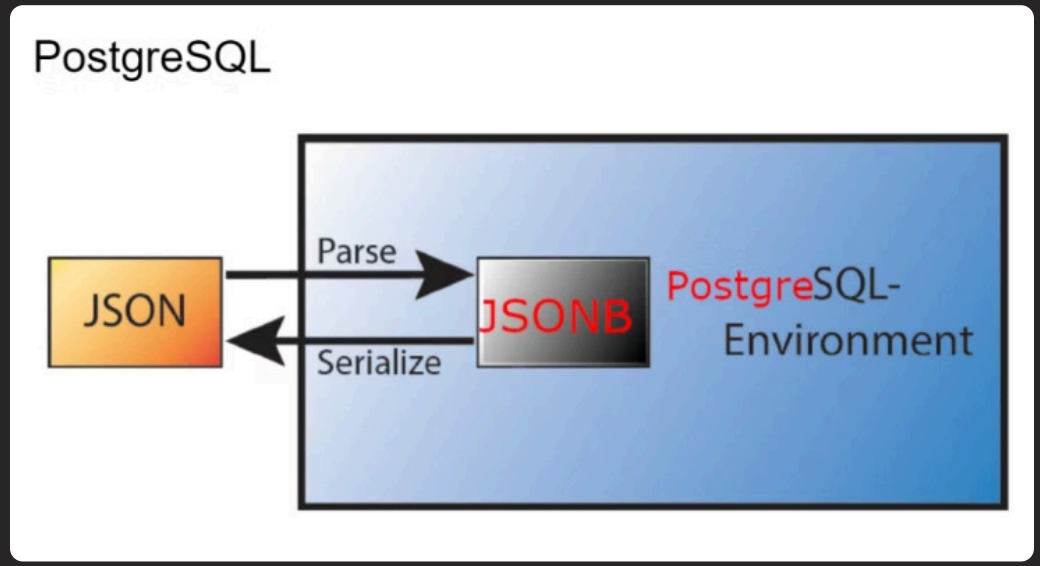
Вложенные структуры

```
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

JSONB — бинарное хранение JSON

В **2014** году появился тип JSONB, который хранит данные в бинарном формате, поддерживает вложенные объекты и массивы, а также индексацию для ускорения запросов.

JSONB парсится один раз и оптимизирован для работы с большими объемами данных, в отличие от JSON, который хранит текст и парсится многократно.



Почему JSONB лучше JSON?

Хранение и парсинг

JSON хранит точную копию текста и парсится многократно, JSONB — бинарный формат с однократным парсингом.

Оптимизация данных

JSONB не хранит отступы и дубликаты ключей, ключи отсортированы по длине.

Функциональность и индексация

JSONB поддерживает больше функций и операторов, а также индексную поддержку для ускорения запросов.

```
SELECT j::json AS json, j::jsonb AS jsonb FROM

(SELECT '{"cc":0, "aa": 2, "aa":1,"b":1}' AS j) AS foo;

json | jsonb

-----------------------------+---------------------------

{"cc":0, "aa": 2, "aa":1,"b":1} | {"b": 1, "aa": 1, "cc": 0}
```

Индексация JSONB в PostgreSQL

Индексы GIN эффективно используются для поиска ключей или пар ключ/значение в больших документах JSONB. Тип JSONB поддерживает индексы btree и hash. Они полезны, только если требуется проверять равенство JSON-документов в целом.

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

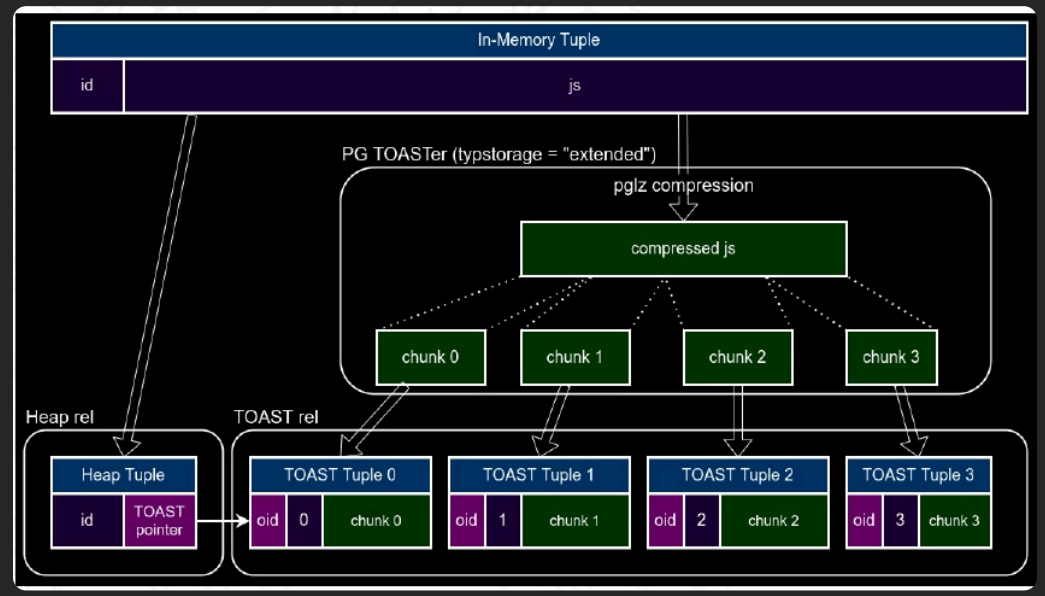
```
CREATE INDEX idxginp ON api USING GIN (jdoc
jsonb_path_ops);
```

Оператор	Тип правого операнда	Описание	Пример
`@>`	`jsonb`	Левое значение JSON содержит на верхнем уровне путь/значение JSON справа?	`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb`
`<@`	`jsonb`	Путь/значение JSON слева содержится на верхнем уровне в правом значении JSON?	`'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb`
`?`	`text`	Присутствует ли ***строка*** в качестве ключа верхнего уровня в значении JSON?	`'{"a":1, "b":2}'::jsonb ? 'b'`
`?|`	`text[]`	Какие-либо ***строки*** массива присутствуют в качестве ключей верхнего уровня?	`'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']`
`?&`	`text[]`	Все ***строки*** массива присутствуют в качестве ключей верхнего уровня?	`'["a", "b"]'::jsonb ?& array['a', 'b']`

Механизм TOAST

TOAST (The Oversized-Attribute Storage Technique) в PostgreSQL автоматически хранит большие данные, превышающие 2 кБ, разделяя их на части и сохраняя в отдельном хранилище.

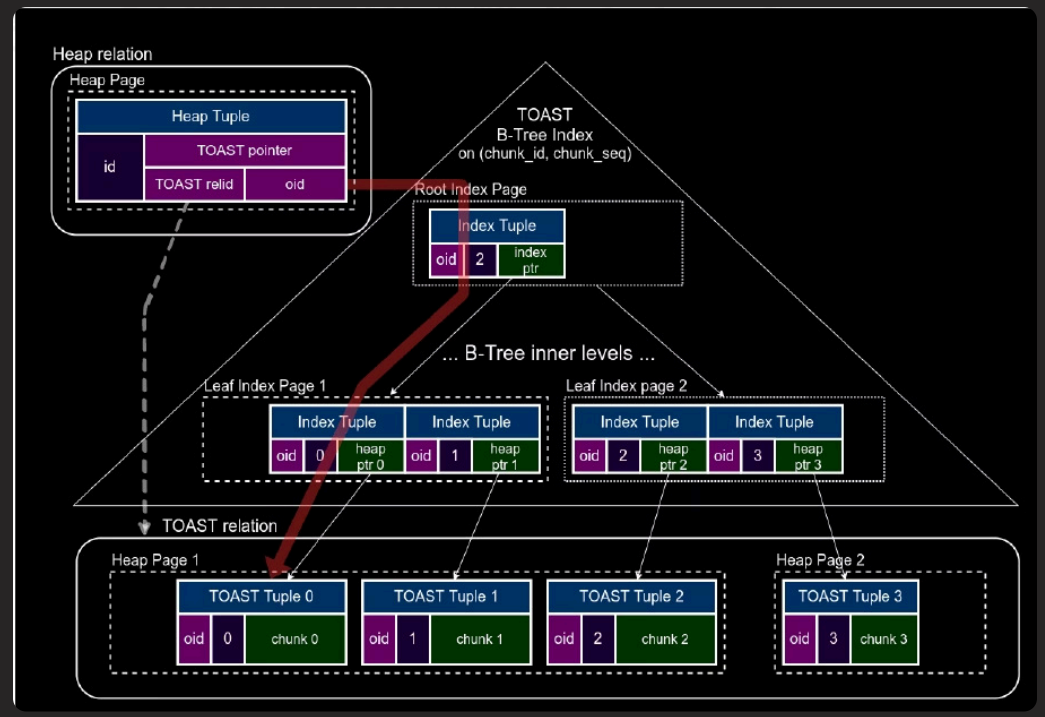
JSON данные сжимаются и разбиваются на чанки, а в основной таблице хранится указатель TOAST с информацией о частях.

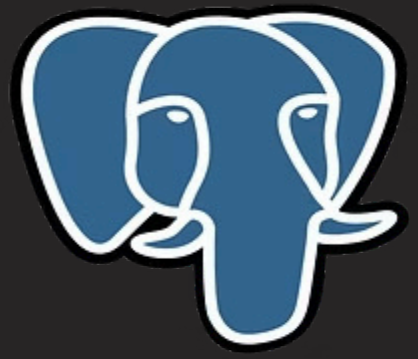


Указатели TOAST и индексная нагрузка

Указатели TOAST не ссылаются напрямую на части данных, а содержат chunk_id, который нужно искать через B-tree индекс по (chunk_id, chunk_seq).

Это вызывает дополнительную нагрузку при чтении небольших фрагментов данных из первого чанка, требуя 3-5 дополнительных индексных блоков.





PostgreSQL

Вопрос для экзамена

Расскажите о ключевых различиях между форматами данных JSON и JSONB в PostgreSQL.