JSON-B, Does it solve everything?

What are the benefits and drawbacks of using JSONB

ENVISAGENICS

Who Am I

- **Anson Abraham**
- Data Architect and Cloud Architect with **Envisagenics**.

- Envisagenics is an AI based Biotech doing Drug discovery in the cloud analyzing RNA data from Cancer Patients.

- We're a Postgres shop. Leveraging Azure Databases for Postgres (which in this case is PostgreSQL 10.1).

Not going to compare to other Doc Stores

1. MongoDB
2. Amazon DynamoDB
3. Amazon DocumentDB
4. Azure CosmosDB
5. RethinkDB
6. CouchBase
7. ArrangoDB
8. ElasticSearch/Solr

WHAT IS JSON(B)

- A new data type introduced in PostgreSQL 9.4 as a way to store a valid JSON elements in a table.
- JSON is stored in plain text
- JSONB is stored in binary representation
- Example of a JSON element:
  - { "ID":"001","name": "Ven", "Country": "Australia", "city": "Sydney", "Job Title":"Database Consultant"}
- Here are some valid JSONB expressions:

```
-- Simple scalar/primitive value
-- Primitive values can be numbers, quoted strings, true, false, or null
SELECT '5'::json;

-- Array of zero or more elements (elements need not be of same type)
SELECT '[1, 2, "foo", null]'::json;

-- Object containing pairs of keys and values
-- Note that object keys must always be quoted strings
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Arrays and objects can be nested arbitrarily
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

Difference Between JSON and JSONB

JSON

- Pretty much like a TEXT data type which stores only valid JSON document.
- Stores the JSON documents as-is including white spaces.
- Does not support FULL-TEXT-SEARCH or GIN Indexing
- Does not support wide range of JSON functions and operators

JSONB

- Stores the JSON documents in Binary format.
- Trims off white spaces and stores in a format conducive for faster and efficient searches
- Supports FULL-TEXT-SEARCH and GIN Indexing.
- Supports all the JSON functions and operators

Very quickly, JSONB supports some operators that JSON doesn’t

- @> or <@
- || : Concatenate two jsonb values into a new jsonb value

```
anson=> \d test_json
                     Table "public.test_json"
 Column |  Type   | Collation | Nullable |                Default
--------+---------+-----------+----------+---------------------------------------
 id     | integer |           | not null | nextval('test_json_id_seq'::regclass)
 data   | json    |           |          |
```

JSON

```
anson=> select * from test_json where data @> '{"lastname":"Abraham"}';
ERROR:  operator does not exist: json @> unknown
LINE 1: select * from test_json where data @> '{"lastname":"Abraham"...
                                           ^
HINT:  No operator matches the given name and argument type(s). You
might need to add explicit type casts.
```

JSONB

```
anson=> select * from test_jsonb where data @> '{"lastname":"Abraham"}'
id |                                                                          data
----+----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 1 | {"age": "", "email": "anson.abraham@gmail.com", "title": "Your Master", "social": {"twitter": "@ansonism", "facebook": "n/a", "linkedin": "anson.abraham", "instagram": "n/a"}, "lastname": "Abraham", "firstname": "Anson"}
(1 row)
```

What are the benefits of using JSONB

- For datasets with many optional values, it is often impractical or impossible to include each one as a table column. In cases like these, JSONB can be a great fit, both for simplicity and performance.
- It also supports nested documents
- JSONB supports integrity constraints
- You can easily update the value to the key in JSONB or add a key or nested doc in the existing key
- JSONB Supports Indexes on Key’s inside document or nested document
- You can put GIN INDEXes on JSONB objects!
- standard SQL and JSONB can be used in the same transactional context, with a tight integration that allows converting JSONB to records (and vice-versa).
- Because it’s postgres it still supports ACID.
- Good for RAD, when doing POC’s.

GIN INDEXING on JSONB

- GIN stands for Generalized Inverted Index. GIN is designed for handling cases where the items to be indexed are composite values, and the queries to be handled by the index need to search for element values that appear within the composite items.
- GIN Indexes the entire document
- Queries could be searches for documents containing specific words.

- ONE Drawback to GIN indexes:
  - takes up more added space than a normal b-tree index.
  - Will be added overhead when running a DML statement.

Example of JSONB table

```
                 Table "example.customer"
     Column     |             Type             | Modifiers
----------------+------------------------------+-----------
 id             | character varying(40)        | not null
 name           | jsonb                        |
 customer_since | timestamp without time zone  |

Indexes:
    "customer_pkey" PRIMARY KEY, btree (id)

Check constraints:
    "validate_customer_name" CHECK ((name ->> 'first name'::text)
IS NOT NULL AND (name ->> 'last name'::text) IS NOT NULL)

Referenced by:
    TABLE "customer_order" CONSTRAINT
"customer_order_customer_id_fkey" FOREIGN KEY (customer_id)
REFERENCES customer(id)
```

```
jsonb=# select * from customer where id = 'cust000000001';
-[ RECORD 1 ]--+-----------------------------------------------
-------------------------------------------------
id             | cust000000001
name           | {"id": "cust000000001", "title": "Dr.",
"initial": "Q", "last name": "Kordon", "first name": "Bob"}
customer_since |


Time: 0.569 ms
```

Indexing on JSONB

WithOut Index

```
jsonb=# select * from customer where name->>'id' =
'cust000000001';

-[ RECORD 1 ]--+--------------------------------------------------
---------------------------------------------------

id             | cust000000001

name           | {"id": "cust000000001", "title": "Dr.",
"initial": "Q", "last name": "Kordon", "first name": "Bob"}

customer_since |


Time: 430.838 ms
```

With Index

```
jsonb=# CREATE INDEX customer_jsonbid_idx ON customer ((name-
>>'id'));


jsonb=# select * from customer where name->>'id' =
'cust000000001';

-[ RECORD 1 ]--+--------------------------------------------------
---------------------------------------------------

id             | cust000000001

name           | {"id": "cust000000001", "title": "Dr.",
"initial": "Q", "last name": "Kordon", "first name": "Bob"}

customer_since |


Time: 0.640 ms
```

INDEX on JSONB Azure PG

```
Table has 1 million rows
anson=> \d measurements
                      Table "public.measurements"
 Column |  Type   | Collation | Nullable |                 Default
--------+---------+-----------+----------+------------------------------------------
 id     | integer |           | not null | nextval('measurements_id_seq'::regclass)
 record | jsonb   |           |          |
```

```
anson=> select * from
measurements where record -
>> 'id' = '3000';
 id | record
----+--------
(0 rows)

Time: 242.691 ms
```

```
anson=> CREATE INDEX ix_measurements_jsonbid on
measurements((record->>'id'));

anson=> select * from measurements where record ->> 'id'
= '3000';
 id | record
----+--------
(0 rows)

Time: 3.337 ms
```

Integrity Constraints on JSONB

```
Check constraints:

    "validate_customer_name" CHECK ((name ->> 'first name'::text)
IS NOT NULL AND (name ->> 'last name'::text) IS NOT NULL)
```

```
INSERT INTO customer VALUES (

       'test2',

       '{"id": "test2","first name": "Hans"}')
```

```
ERROR:  new row for relation "customer" violates check constraint
"validate_customer_name"

DETAIL:  Failing row contains (test2, {"id": "test2", "first
name": "Hans"}, null, null).
```

```
INSERT INTO customer VALUES (
       'test2',
       '{"id": "test2","first name": "Hans", "last name":
"Prince"}')
```

```
jsonb=# SELECT id, jsonb_pretty(name)from customer where id =
'test2';
  id   |         jsonb_pretty
-------+---------------------------
 test2 | {                        +
       |     "id": "test2",       +
       |     "last name": "Prince",+
       |     "first name": "Hans"  +
       | }
```

```
  UPDATE customer
         SET name = name || '{"title": "Hon"}'
         WHERE id = 'test2';
```

```
jsonb=# SELECT id, jsonb_pretty(name)from customer where id =
'test2';
  id   |         jsonb_pretty
-------+---------------------------
 test2 | {                        +
       |     "id": "test2",       +
       |     "title": "Hon",      +
       |     "last name": "Prince",+
       |     "first name": "Hans"  +
       | }
```

```
UPDATE customer

      SET name = name || '{"title": "His Excellence"}'

      WHERE id = 'test2';
```

```
jsonb=# SELECT id, jsonb_pretty(name)from customer where id =
'test2';
  id   |          jsonb_pretty
-------+--------------------------------
 test2 | {                             +
       |     "id": "test2",            +
       |     "title": "His Excellence",+
       |     "last name": "Prince",    +
       |     "first name": "Hans"      +
       | }
```

```
UPDATE customer

      SET name = name ||

        '{"contact": {"fax": "617-123-5678", "cell": "617-123-
        4567"}}'

      WHERE id = 'test2';
```

```
jsonb=# SELECT id, jsonb_pretty(name)from customer where id =
'test2';
  id   |          jsonb_pretty
-------+--------------------------------
 test2 | {                             +
       |     "id": "test2",            +
       |     "title": "His Excellence",+
       |     "contact": {              +
       |         "fax": "617-123-5678",+
       |         "cell": "617-123-4567"+
       |     },                        +
       |     "last name": "Prince",    +
       |     "first name": "Hans"      +
       | }
```

```
UPDATE customer
       SET name = name #- '{contact, fax}'
              WHERE id = 'test2';


jsonb=# SELECT id, jsonb_pretty(name)from customer where id =
'test2';
  id   |          jsonb_pretty
-------+--------------------------------
 test2 | {                             +
       |     "id": "test2",            +
       |     "title": "His Excellence",+
       |     "contact": {              +
       |         "cell": "617-123-0101"+
       |     },                        +
       |     "last name": "Prince",    +
       |     "first name": "Hans"      +
       | }
(1 row)
```

The Caveats to JSONB

1. Slow Queries Due To Lack Of Statistics
2. Comparison operators will not work if using GIN indexes (unless you use jsonb_path_ops)
3. Larger Table Footprint
    1. Does not preserve white space
4. It's stricter
5. Does not preserve the order of object keys
6. Size limitation

Lack of Statistics

For traditional data types, PostgreSQL stores statistics about
the distribution of values in each column of each table, such as:

- the number of distinct values seen
- the most common values
- the fraction of entries that are NULL
- for ordered types, a histogram sketch of the distribution of values in the column

```
CREATE TABLE measurements (
  tick BIGSERIAL PRIMARY KEY,
  value_1 INTEGER,
  value_2 INTEGER,
  value_3 INTEGER,
  scientist_id BIGINT
);
CREATE TABLE scientist_labs (scientist_id BIGSERIAL
PRIMARY KEY, lab_name TEXT);
ANALYZE;
```

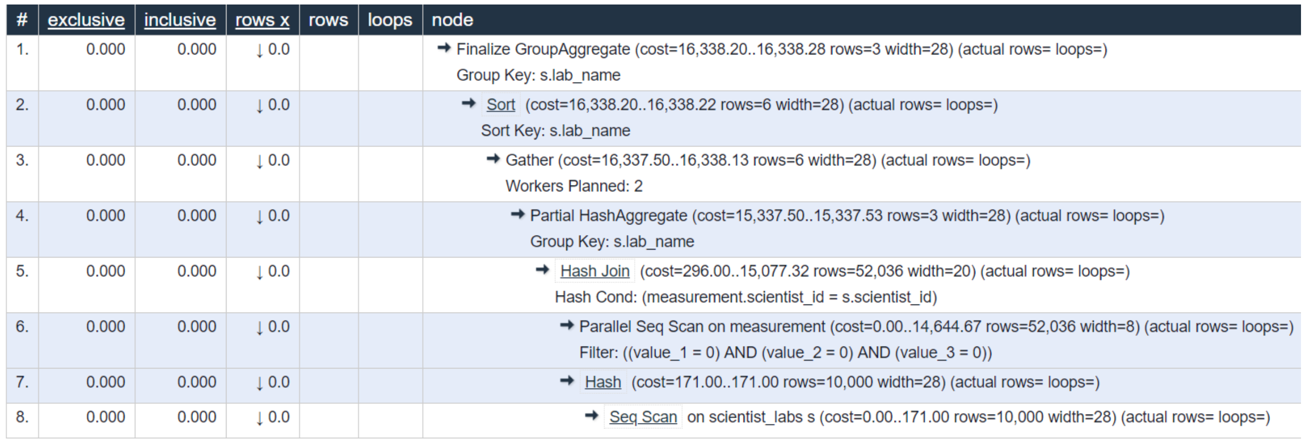
```
SELECT lab_name, COUNT(*)
FROM (
  SELECT scientist_id
  FROM measurements
  WHERE
    value_1 = 0 AND
    value_2 = 0 AND
    value_3 = 0
) m
JOIN scientist_labs AS s
  ON (m.scientist_id = s.scientist_id)
GROUP BY lab_name;
```

```
CREATE TABLE measurement2 (tick BIGSERIAL PRIMARY KEY,
record JSONB);
INSERT INTO measurement2 (record)
  SELECT (
    '{ "value_1":' || trunc(2 * random()) ||
    ', "value_2":' || trunc(2 * random()) ||
    ', "value_3":' || trunc(2 * random()) ||
    ', "scientist_id":' || trunc(10000 * random() + 1) || ' }')::JSONB
  FROM generate_series(0, 999999) i
```

```
SELECT lab_name, COUNT(*)
 FROM (
   SELECT (record ->> 'scientist_id')::BIGINT AS scientist_id
   FROM measurement2
   WHERE
     (record ->> 'value_1')::INTEGER = 0 AND
     (record ->> 'value_2')::INTEGER = 0 AND
     (record ->> 'value_3')::INTEGER = 0
 ) m
 JOIN scientist_labs AS s
   ON (m.scientist_id = s.scientist_id)
 GROUP BY lab_name;
```

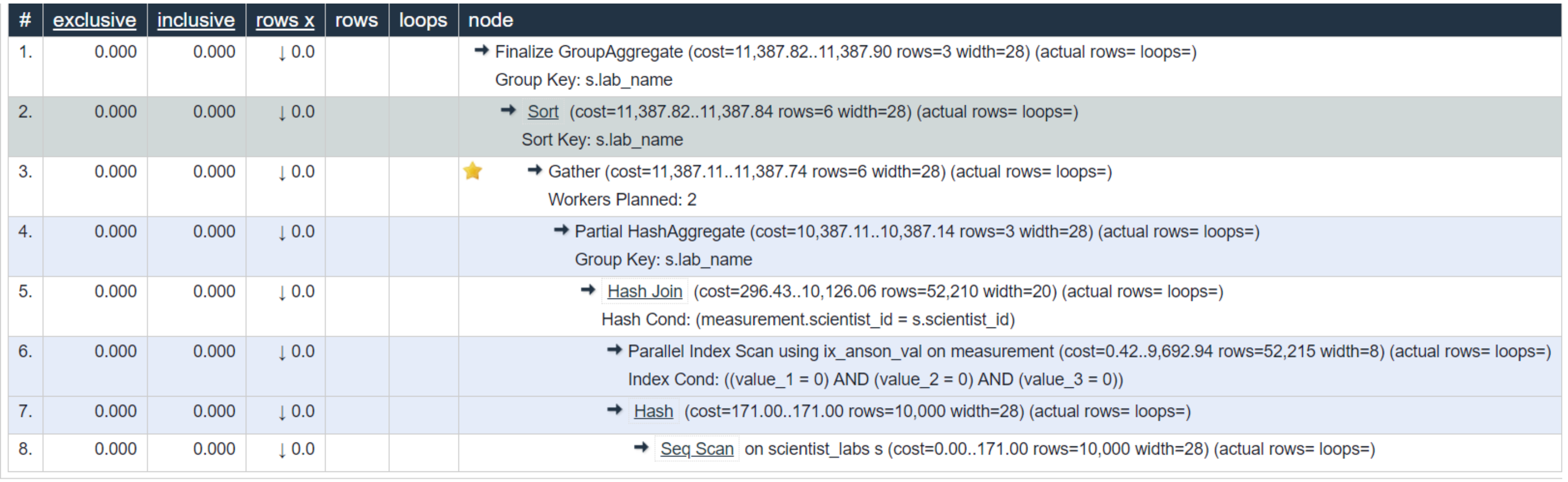
On Azure PG Non-JSON

No Indexes Measurements table on value_n and scientist ID: Time: 10.170 ms



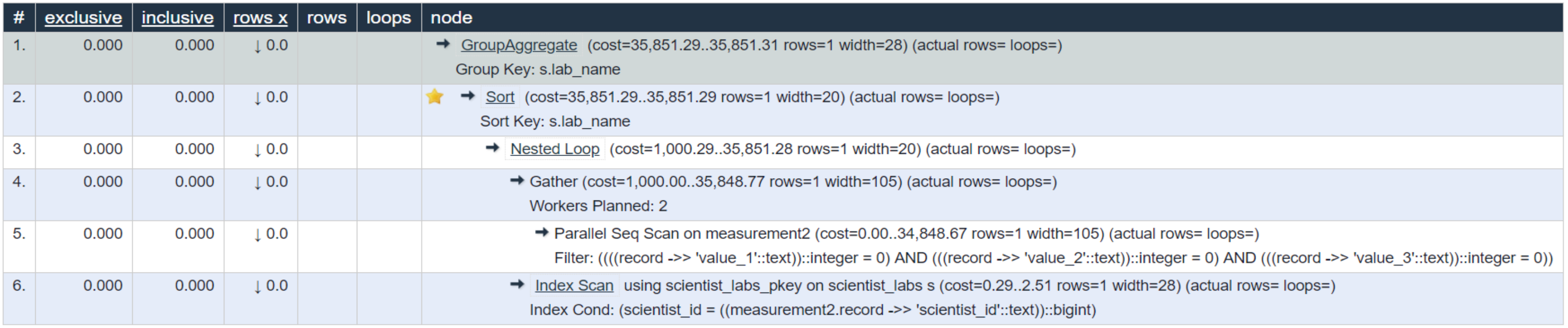
#	exclusive	inclusive	rows x	rows	loops	node
1.	0.000	0.000	↓ 0.0			→ Finalize GroupAggregate (cost=16,338.20..16,338.28 rows=3 width=28) (actual rows= loops=) Group Key: s.lab_name
2.	0.000	0.000	↓ 0.0			→ Sort (cost=16,338.20..16,338.22 rows=6 width=28) (actual rows= loops=) Sort Key: s.lab_name
3.	0.000	0.000	↓ 0.0			→ Gather (cost=16,337.50..16,338.13 rows=6 width=28) (actual rows= loops=) Workers Planned: 2
4.	0.000	0.000	↓ 0.0			→ Partial HashAggregate (cost=15,337.50..15,337.53 rows=3 width=28) (actual rows= loops=) Group Key: s.lab_name
5.	0.000	0.000	↓ 0.0			→ Hash Join (cost=296.00..15,077.32 rows=52,036 width=20) (actual rows= loops=) Hash Cond: (measurement.scientist_id = s.scientist_id)
6.	0.000	0.000	↓ 0.0			→ Parallel Seq Scan on measurement (cost=0.00..14,644.67 rows=52,036 width=8) (actual rows= loops=) Filter: ((value_1 = 0) AND (value_2 = 0) AND (value_3 = 0))
7.	0.000	0.000	↓ 0.0			→ Hash (cost=171.00..171.00 rows=10,000 width=28) (actual rows= loops=)
8.	0.000	0.000	↓ 0.0			→ Seq Scan on scientist_labs s (cost=0.00..171.00 rows=10,000 width=28) (actual rows= loops=)

Added Indexes on Measurements table on value_n and scientist ID: Time: 4.336 ms



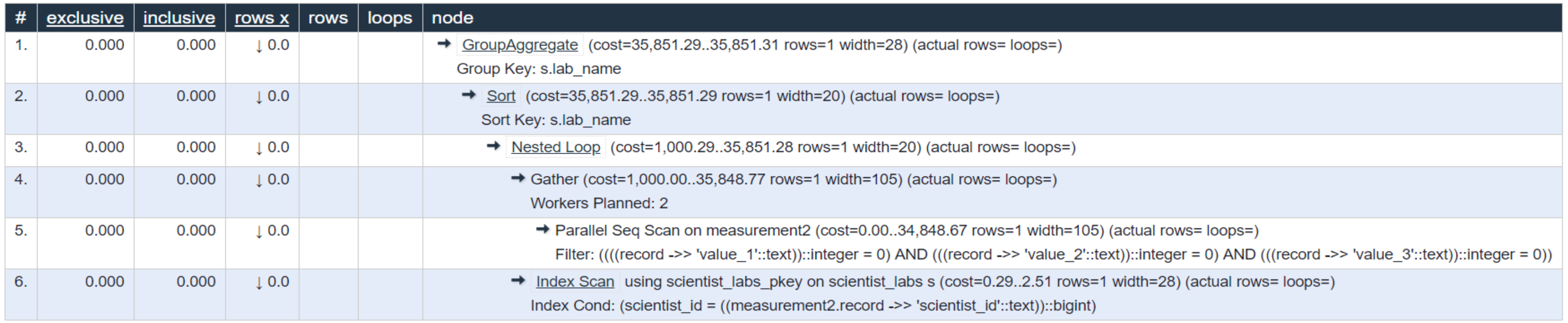
#	exclusive	inclusive	rows x	rows	loops	node
1.	0.000	0.000	↓ 0.0			→ Finalize GroupAggregate (cost=11,387.82..11,387.90 rows=3 width=28) (actual rows= loops=) Group Key: s.lab_name
2.	0.000	0.000	↓ 0.0			→ Sort (cost=11,387.82..11,387.84 rows=6 width=28) (actual rows= loops=) Sort Key: s.lab_name
3.	0.000	0.000	↓ 0.0			⭐ → Gather (cost=11,387.11..11,387.74 rows=6 width=28) (actual rows= loops=) Workers Planned: 2
4.	0.000	0.000	↓ 0.0			→ Partial HashAggregate (cost=10,387.11..10,387.14 rows=3 width=28) (actual rows= loops=) Group Key: s.lab_name
5.	0.000	0.000	↓ 0.0			→ Hash Join (cost=296.43..10,126.06 rows=52,210 width=20) (actual rows= loops=) Hash Cond: (measurement.scientist_id = s.scientist_id)
6.	0.000	0.000	↓ 0.0			→ Parallel Index Scan using ix_anson_val on measurement (cost=0.42..9,692.94 rows=52,215 width=8) (actual rows= loops=) Index Cond: ((value_1 = 0) AND (value_2 = 0) AND (value_3 = 0))
7.	0.000	0.000	↓ 0.0			→ Hash (cost=171.00..171.00 rows=10,000 width=28) (actual rows= loops=)
8.	0.000	0.000	↓ 0.0			→ Seq Scan on scientist_labs s (cost=0.00..171.00 rows=10,000 width=28) (actual rows= loops=)

Azure PG JSON



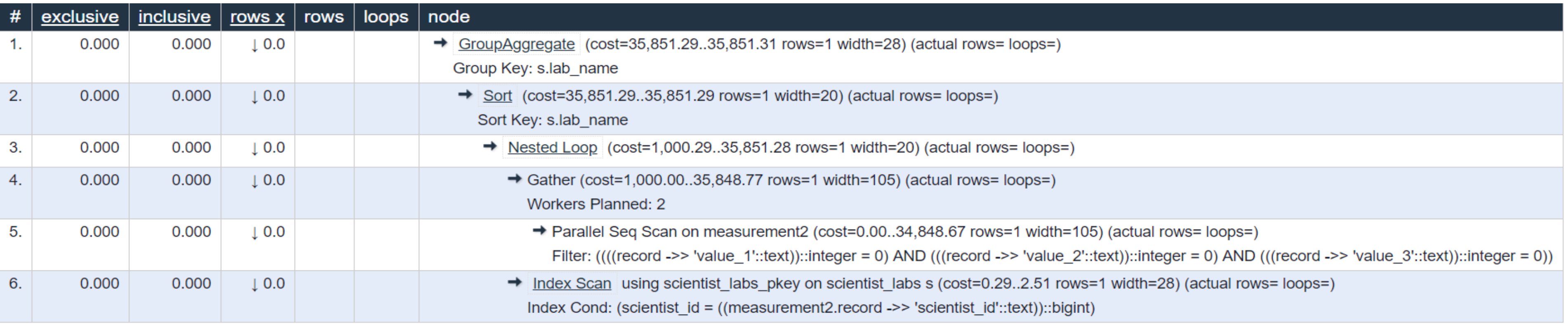
#	exclusive	inclusive	rows x	rows	loops	node
1.	0.000	0.000	↓ 0.0			→ GroupAggregate (cost=35,851.29..35,851.31 rows=1 width=28) (actual rows= loops=) Group Key: s.lab_name
2.	0.000	0.000	↓ 0.0			⭐ → Sort (cost=35,851.29..35,851.29 rows=1 width=20) (actual rows= loops=) Sort Key: s.lab_name
3.	0.000	0.000	↓ 0.0			→ Nested Loop (cost=1,000.29..35,851.28 rows=1 width=20) (actual rows= loops=)
4.	0.000	0.000	↓ 0.0			→ Gather (cost=1,000.00..35,848.77 rows=1 width=105) (actual rows= loops=) Workers Planned: 2
5.	0.000	0.000	↓ 0.0			→ Parallel Seq Scan on measurement2 (cost=0.00..34,848.67 rows=1 width=105) (actual rows= loops=) Filter: ((((record ->> 'value_1'::text))::integer = 0) AND (((record ->> 'value_2'::text))::integer = 0) AND (((record ->> 'value_3'::text))::integer = 0))
6.	0.000	0.000	↓ 0.0			→ Index Scan using scientist_labs_pkey on scientist_labs s (cost=0.29..2.51 rows=1 width=28) (actual rows= loops=) Index Cond: (scientist_id = ((measurement2.record ->> 'scientist_id'::text))::bigint)

No INDEX: Time: 951.118 ms



#	exclusive	inclusive	rows x	rows	loops	node
1.	0.000	0.000	↓ 0.0			→ GroupAggregate (cost=35,851.29..35,851.31 rows=1 width=28) (actual rows= loops=) Group Key: s.lab_name
2.	0.000	0.000	↓ 0.0			→ Sort (cost=35,851.29..35,851.29 rows=1 width=20) (actual rows= loops=) Sort Key: s.lab_name
3.	0.000	0.000	↓ 0.0			→ Nested Loop (cost=1,000.29..35,851.28 rows=1 width=20) (actual rows= loops=)
4.	0.000	0.000	↓ 0.0			→ Gather (cost=1,000.00..35,848.77 rows=1 width=105) (actual rows= loops=) Workers Planned: 2
5.	0.000	0.000	↓ 0.0			→ Parallel Seq Scan on measurement2 (cost=0.00..34,848.67 rows=1 width=105) (actual rows= loops=) Filter: ((((record ->> 'value_1'::text))::integer = 0) AND (((record ->> 'value_2'::text))::integer = 0) AND (((record ->> 'value_3'::text))::integer = 0))
6.	0.000	0.000	↓ 0.0			→ Index Scan using scientist_labs_pkey on scientist_labs s (cost=0.29..2.51 rows=1 width=28) (actual rows= loops=) Index Cond: (scientist_id = ((measurement2.record ->> 'scientist_id'::text))::bigint)

B-Tree index: Time: 886.455 ms



#	exclusive	inclusive	rows x	rows	loops	node
1.	0.000	0.000	↓ 0.0			→ GroupAggregate (cost=35,851.29..35,851.31 rows=1 width=28) (actual rows= loops=) Group Key: s.lab_name
2.	0.000	0.000	↓ 0.0			→ Sort (cost=35,851.29..35,851.29 rows=1 width=20) (actual rows= loops=) Sort Key: s.lab_name
3.	0.000	0.000	↓ 0.0			→ Nested Loop (cost=1,000.29..35,851.28 rows=1 width=20) (actual rows= loops=)
4.	0.000	0.000	↓ 0.0			→ Gather (cost=1,000.00..35,848.77 rows=1 width=105) (actual rows= loops=) Workers Planned: 2
5.	0.000	0.000	↓ 0.0			→ Parallel Seq Scan on measurement2 (cost=0.00..34,848.67 rows=1 width=105) (actual rows= loops=) Filter: ((((record ->> 'value_1'::text))::integer = 0) AND (((record ->> 'value_2'::text))::integer = 0) AND (((record ->> 'value_3'::text))::integer = 0))
6.	0.000	0.000	↓ 0.0			→ Index Scan using scientist_labs_pkey on scientist_labs s (cost=0.29..2.51 rows=1 width=28) (actual rows= loops=) Index Cond: (scientist_id = ((measurement2.record ->> 'scientist_id'::text))::bigint)

GIN INDEX Time: 865.447 ms

Larger table footprint

```
CREATE TABLE measurements (tick BIGSERIAL PRIMARY KEY, record JSONB);

CREATE TABLE measurements2 (tick BIGSERIAL PRIMARY KEY, value_1 INT, value_2 INT,
value_3 INT);

INSERT INTO measurements (record)
  SELECT (
    '{ "value_1":' || trunc(2 * random()) ||
    ', "value_2":' || trunc(2 * random()) ||
    ', "value_3":' || trunc(2 * random()) ||
    ', "scientist_id":' || trunc(10000 * random() + 1) || ' }')::JSONB
  FROM generate_series(0, 999999);
```

- the initial non-JSONB version of our table (measurements2) takes up 79 mb of disk space
- the JSONB variant (measurements) takes 164 mb
  - OVER 50% more data space used!

Cons of JSONB (cont)

- jsonb is stricter, and as such, it disallows Unicode escapes for non-ASCII characters (those above U+007F) unless the database encoding is UTF8. It also rejects the NULL character (\u0000), which cannot be represented in PostgreSQL's text type.

- It does not preserve white space, and it will strip your JSON strings of leading/lagging white space as well as white space within the JSON string, all of which will just untidy your code (which might not be a bad thing for you after all.)

- It does not preserve the order of object keys, treating keys in pretty much the same way as they are treated in Python dictionaries -- unsorted. You'll need to find a way around this if you rely on the order of your JSON keys.

What if Document exceeds size for JSONB?

Horizontal partitioning?

- You can create multiple tables and then have a join in a view per the key in table.
- Drawback:
    - you're maintaining multiple tables for one record
    - Postgres does not do this automatically with sharding.

External DocStore DB (ex: MongoDB)

- MongoDB has size limitations as well (100MB).
- mongoDB does do automatic sharding.
- Need to create a FDW in postgres to return results from mongo or write code to do an "app join" between postgres and mongo.

Foreign Data Wrappers.

- Foreign Table : this is about how to access external data sources and present them as relational tables.
- Datalink : this extends the functionality of database systems to include control over external files without the need to store their contents directly in the database, such as LOBs. A column of a table could directly refer a file.

What about The Hybrid Approach?

- Data which has columns that will always be constant have that as “static” column in table and attaching new events to data can be in jsonb.
- CREATE TABLE measurements (tick bigint, val1 int, val2 int, val3 int, record jsonb).
  - If you’re doing a lookup in the jsonb, create either an index on key or gin index on whole thing.
    - CREATE GIN INDEX ix_measurements_gin on measurements (record);
    - CREATE INDEX ix_measurements_record_key1 on measurements (record - - >> ‘key’);
- Make sure document doesn’t exceed over 268435455 bytes!!!

Some common examples of using JSONB

- Event tracking data, where you may want to include the payload in the event which might vary
- Gaming data is especially common, especially where you have single player games and have a changing schema based on the state of the user
- Tools that integrate multiple data sources, an example here may be a tool that integrates customers databases to Salesforce to Zendesk to something else. The mix of schemas makes doing this in a multitenant fashion more painful than it has to be.
- Maybe TimeSeries data(?)
- Blockchain data
- Genomic Data

When NOT to use JSON-B

- Using it as a key-value store. That's what redis and aerospike are for.
  - If you do, gonna need some really expensive ultra ssd disks.
- Storing Spatial Data. → Err … POSTGIS?
- Store an entire PDF like doc. First, going to have size issues, when parsing. Two Use Elastic Search
- If you have reccurring properties (columns) … use a table. That's what RDBMS' were built for.
- Highly Transactional traffic. If writes are required to be nano-seconds fast. Suggest to use Cassandra.

Some Alternative DocStores out there

- MongoDB: Most popular one out there. Good luck with that.
- RethinkDB: They did close shop, though community is supporting it
- ArrangoDB
- AWS DocumentDB
- Azure CosmoDB.

Acknowledgement / Footnote

- Craig Kerstein: Citus Data (http://www.craigkerstiens.com)
- Marc Linster: EnterpriseDB
- Dan Robinson: Heap.IO (https://heap.io/blog/author/drob)
- https://explain.depesz.com

In conclusion

In most cases JSONB is good when looking for a NoSQL, schema-less, datatype. JSONB isn't always a fit in every data model. However, it's always better to normalize as we saw earlier. If you do have a schema that has a large number of optional columns (such as with event data) or the schema differs based on tenant id then JSONB can be a great fit. In general you want:

- JSONB - In most cases
- JSON - If you're just processing logs, don't often need to query, and use as more of an audit trail.