ADVANCED DATATYPES

Array, JSON and GIN indices

ARRAYS

ARRAYS

```
create table flights(
   airport_from text,
   airport_to   text,
   services     text[]
);


insert into flights values('BTS', 'VIE',
'{lunch, 1st class}');

insert into flights values('BTS', 'FRA',
'{lunch, wifi}');
```

ARRAYS

```
select *
from flights f
where 'wifi' = any(f.services);

select *
from flights f
where f.services @> '{wifi}';
```

USAGE?

- Denormalized schema (not necessarily bad)
- Convenience (avoids many-to-many)
- Keeping track of state in recursive queries (to avoid cycles)

INDEXING ARRAY ACCESS

```
create index idx_services on
flights(services); -- b-tree

set enable_seqscan = 'off';
-- we have only 2 rows
```

INDEXING ARRAY ACCESS

```
explain
select *
from flights f
where 'wifi' = any(f.services);
```

```
Seq Scan on flights f  (cost=0.00..1.07
rows=1 width=96)
   Filter: ('wifi'::text = ANY (services))
```

INDEXING ARRAY ACCESS

```
explain
select *
from flights f
where f.services = '{lunch, wifi}';
```

```
Index Scan using idx_services on flights f
(cost=0.13..8.15 rows=1 width=96)
   Index Cond: (services =
'{lunch,wifi}'::text[])
```

INDEXING ARRAY ACCESS

```
drop index idx_services;
create index idx_services on flights using
gin(services); -- generalized inverted index

set enable_seqscan = 'off';
-- we still have only 2 rows
```

INDEXING ARRAY ACCESS

```
explain
select *
from flights f
where f.services @> '{wifi}';

Bitmap Heap Scan on flights f
(cost=8.00..12.01 rows=1 width=96)
   Recheck Cond: (services @>
'{wifi}'::text[])
   ->  Bitmap Index Scan on idx_services
(cost=0.00..8.00 rows=1 width=0)
         Index Cond: (services @>
'{wifi}'::text[])
```

INVERTED INDEX

TABLE

TID	SERVICES
1	['LUNCH', '1ST CLASS']
2	['LUNCH', 'WIFI']

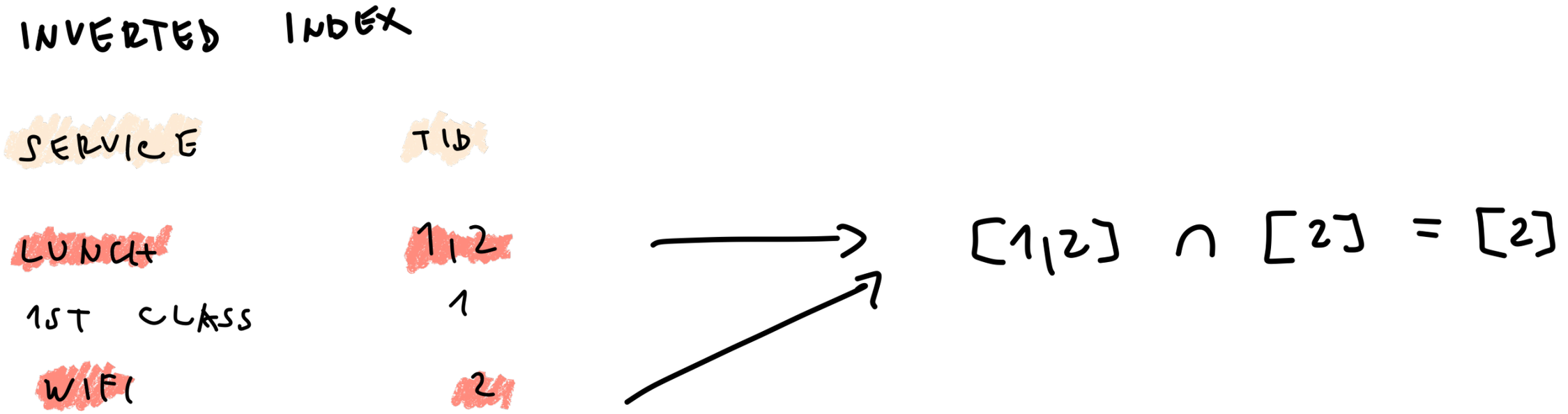
INVERTED INDEX

SERVICE	TID
LUNCH	1, 2
1ST CLASS	1
WIFI	2

INDEXING ARRAY ACCESS

```
explain
select *
from flights f
where f.services @> '{wifi, lunch}';
```

INVERTED INDEX



TRANSFORM ARRAY BACK TO ROWS (E.G. TO JOIN)

```
select airport_from, airport_to,
unnest(services)
from flights f
where 'wifi' = any(f.services);
```

```
 airport_from | airport_to | unnest
--------------+------------+--------
 BTS          | VIE        | lunch
 BTS          | VIE        | wifi
```

JSON

JSON

- JavaScript Object Notation
- Very popular
  - Simple, convenient, human readable, universal support
- Basic data types

JSON EXAMPLE

```
{
  "name": "Advanced datatypes",
  "length": 120,
  "topics": ["arrays", "gin", "json"],
  "teacher": {
     "name": "Tomas",
     "surname": "Kramar"
  }
}
```

USAGE?

- Data schema not known in advance
- Impractical to keep adding columns
- Can introduce new keys (even nested) anytime
- Checks JSON validity

```
create table lectures_json(doc json);

insert into lectures_json values('{name:
"Advanced datatypes", "length": 120}');
```

```
ERROR:  invalid input syntax for type json
LINE 1: insert into lectures_json
values('{name: "Advanced datatypes...
DETAIL:  Token "name" is invalid.
CONTEXT:  JSON data, line 1: {name...
```

```
select doc->name from lectures_jsonb;
ERROR:  column "name" does not exist

select doc->'name' from lectures_json;

          ?column?
----------------------------
 "Advanced datatypes"
```

2 JSON DATATYPES IN POSTGRES

- JSON
  - text representation
  - preserves formatting and key ordering
  - must be re-parsed on each read
- JSONB
  - binary representation
  - loses formatting and key ordering
  - slower to write (encoding), faster to read (no re-parsing)
  - supports indexing

```
create table lectures_json(doc json);
create table lectures_jsonb(doc jsonb);

insert into lectures_json values('{"name":
"Advanced datatypes",  "length": 120}');

insert into lectures_jsonb
values('{"name": "Advanced datatypes",
"length": 120}');
```

```
select * from lectures_json;
{"name": "Advanced datatypes",   "length":
120}

select * from lectures_jsonb;
{"name": "Advanced datatypes", "length":
120}
```

INDEXING JSONB

```
create index idx_doc on lectures_jsonb using
gin(doc);

-- Allows "containment" queries

explain select * from lectures_jsonb
where doc @> '{"name": "Advanced datatypes"}';
```

```
Bitmap Heap Scan on lectures_jsonb
(cost=12.00..16.01 rows=1 width=32)
   Recheck Cond: (doc @> '{"name": "Advanced
datatypes"}'::jsonb)
   ->  Bitmap Index Scan on idx_doc
(cost=0.00..12.00 rows=1 width=0)
         Index Cond: (doc @> '{"name": "Advanced
datatypes"}'::jsonb)
```

GIN INDEX OVER JSONB

TID	JSONB
1	{ "NAME" : "ADVANCED DATA ..", "LENGTH" : 120 }
2	{ "NAME" : "GIS", "LENGTH" : 120 }

KEY / VALUE	TID
NAME / ADVANCED DATA ...	1
LENGTH / 120	1,2
NAME / GIS	2

HANDLING NESTING BY FLATTENING

```
{
    "TEACHER" : {
        "NAME" : "TOMAS"
    }
}
```

TEACHER.NAME / TOMAS

INDEXING JSONB OVER A SINGLE "PROPERTY"

```
create index idx_lectures_on_name on
lectures_jsonb using gin((doc->'name'));

explain select * from lectures_jsonb
 where doc->'name' ? 'Advanced datatypes';
```

```
Bitmap Heap Scan on lectures_jsonb
(cost=8.00..12.02 rows=1 width=32)
   Recheck Cond: ((doc -> 'name'::text) ?
'Advanced datatypes'::text)
   ->  Bitmap Index Scan on idx_lectures_on_name
(cost=0.00..8.00 rows=1 width=0)
         Index Cond: ((doc -> 'name'::text) ?
'Advanced datatypes'::text)
```

JSON FUNCTIONS

- row_to_json
  - select row_to_json(t) from (select name, length from lectures) t
- array_agg
- array_to_json
- ::json cast

WHAT YOU SHOULD KNOW

- Understand principles of inverted index and how it's different from b-tree
- Understand how inverted index answers compound conditions (`@> {a, b}`)
- Understand how b-tree index works for arrays and its limitations compared to inverted index