Inside the machine room of a world map: PostgreSQL and OpenStreetMap

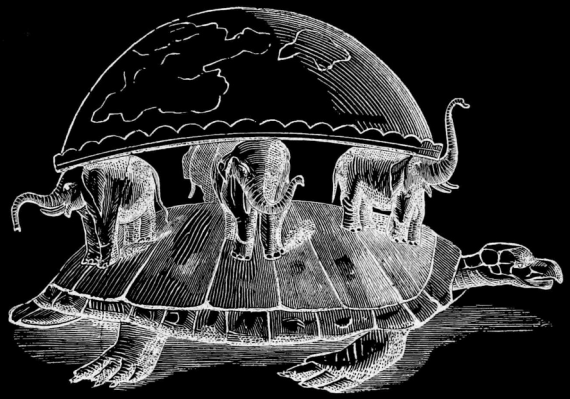
Sarah Hoffmann

About OpenStreetMap

- crowd-sourced spatial database of Earth
- founded in 2004 by hobbyists in UK
- grown to
  - > 50 million km roads
  - > 500 million buildings
  - > 160 million addresses
- used by Meta, Apple, Microsoft, …
- and on your phone

About OpenStreetMap

- PostgreSQL user since 2006
  - main DB
  - rendering
  - search
  - ~~routing~~



About me

- active in OpenStreetMap since 2008
- PostgreSQL user since making the first map in 2009
- maintainer for osm2pgsql, Nominatim, Photon, pyosmium, waymarkedtrails

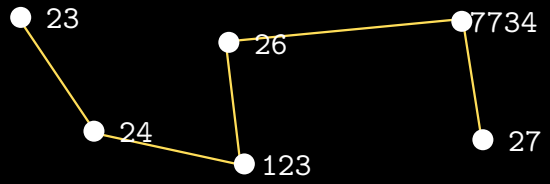
Introduction to OpenStreetMap

(for database engineers)

Base elements: Node

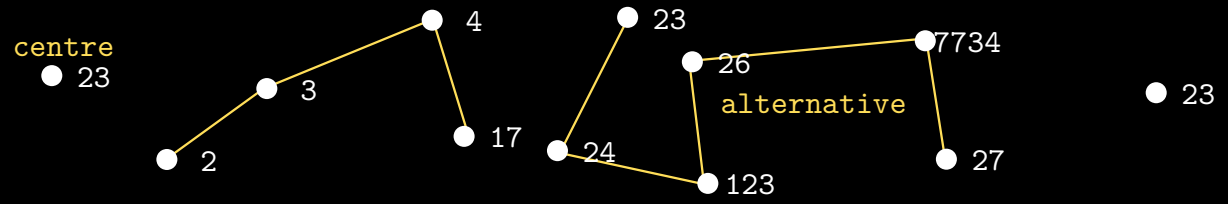
- spatial coordinate on Earth
- in WGS 84

Base elements: Way



- ordered list of node IDs
- length between 2 and 2000

Base elements: Relation



- ordered list of OSM objects (including relations)
- list elements are called members
- optional role: string attribute for member

OSM data elements

Node

WGS84 coordinate

- `id`
- `(lat, lon)`

Way

ordered list of nodes

- `id`
- `nodes = [Id₁, Id₂, ...]`

Relation

ordered list of elements with role

- `id`
- `members = [([n|w|r], Id, role), ...]`

+ Common Attributes

`tags: hstore`	*defines the type and properties*
`version: sequence`	*change tracking*
`visible: bool`	*deletion*
`user, user ID, changeset ID, timestamp`	*housekeeping*

Takeaways

- topology-based data model
  - node: sole element with geometry
  - way: needs node information for assembly
  - relation: needs node and way information for assembly

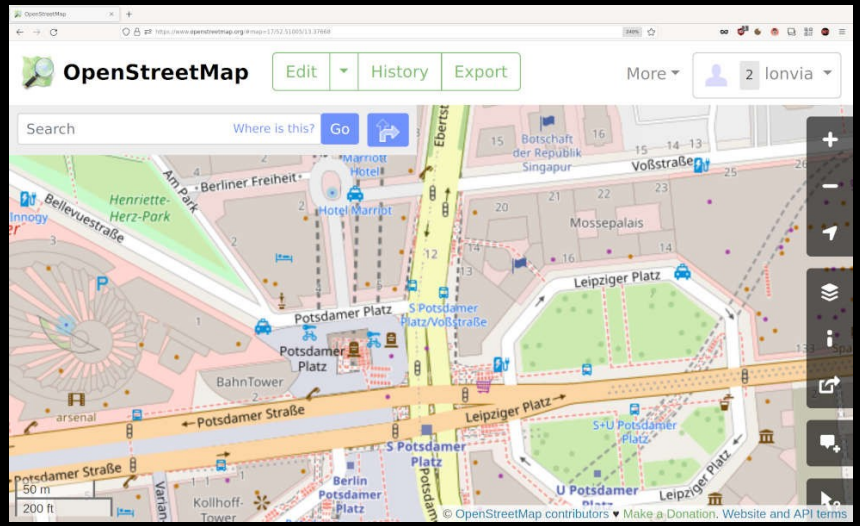
The OSM main DB

https://github.com/openstreetmap/openstreetmap-website/

https://github.com/zerebubuth/openstreetmap-cgimap

About

- Rails application started in 2006 using MySQL
- switched to PostgreSQL 8.3 in 2009 because transactions
- additional C++-based frontend (cgimap) for most data API calls



Hardware and Database

- 1 primary server + 5 read-only backup server
- database: 14 TB (+ 2 TB last year)
  - 12 billion nodes (97% without tags)
  - 1.7 billion ways
  - 44 million relations
- currently still running PostgreSQL 9.5
  (we'll come to that)

Data API

- upload API (avg. 20/s)
  - create/modify/delete single elements
  - bulk-upload elements (transactional)
- download API (for editors only, avg. 30/s)
  - map call (all current data within a bounding box)
  - single elements (optionally + dependent data)

- weekly data dumps + minutely diffs

Database Schema

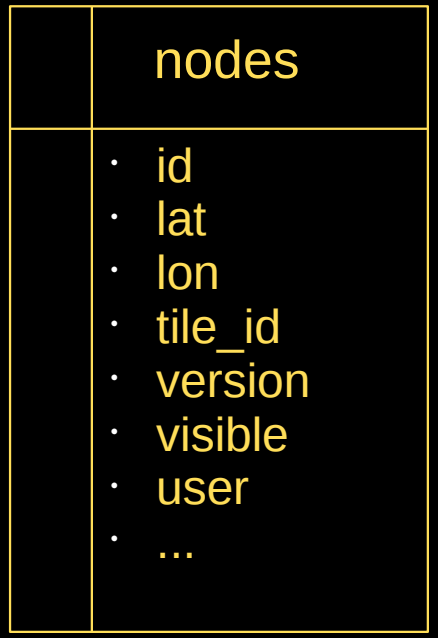
- pure relational model
  - base tables for metadata
  - tag tables with `(id, key, value)`
  - way_nodes with `(id, sequence_id, node_id)`
  - relation_members with `(id, sequence_id, role, member_type, member_id)`
- mirror tables for latest visible version

Database Schema

- pure relational model
  - base tables for metadata
  - tag tables with `(id, key, value)`
  - way_nodes with `(id, sequence_id, node_id)`
  - relation_members with
    `(id, sequence_id, role, member_type, member_id)`
- mirror tables for latest visible version

The Big one.
Table size: 1.5TB
Primary index size: 1.5TB

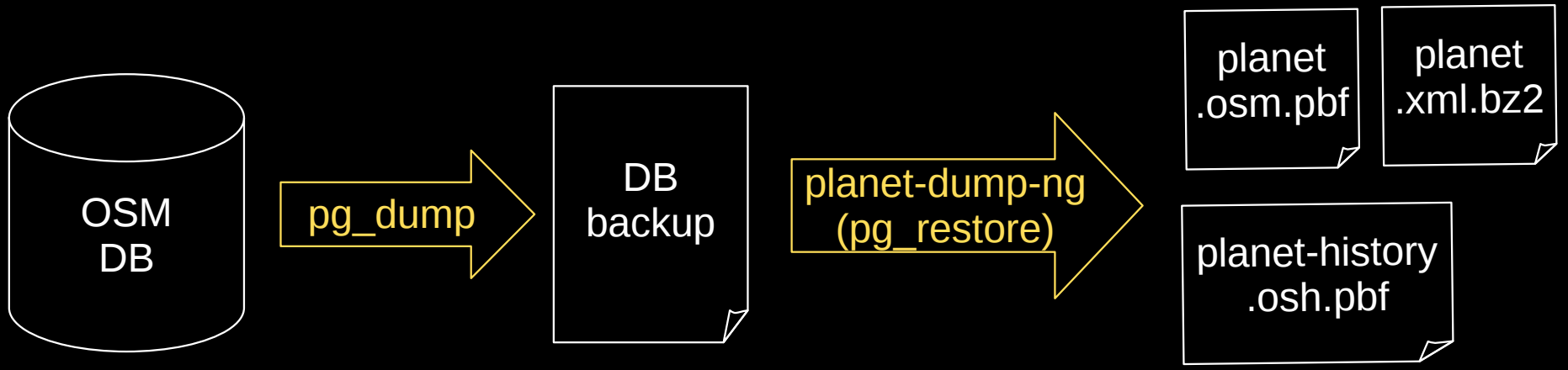
The `nodes` table



size: 1TB

- not a PostGIS table
- only spatial operation: map call
  - tile ID: bit-interleaved lat/lon
  - custom bbox lookup of nodes

Planet dumps



- avoids long-running transactions
- decoupling from main DB server
- published once per week

Minutely diffs: 1st try (until 2009)

- use timestamp of the OSM object
  - 5 minutes behind database
  - get all objects within the minute of interest

Problem:

data loss with long-running transactions

Minutely diffs: 2nd try (until 2021)

- use transaction id to track new data
  - query xMax transaction ID and open transactions
  - get all data from closed transactions since last diff
  - run minutely without delay

Problem:

needs index on xid (disallowed since Postgresql 9.6)

Minutely diffs

- use logical replication
  - simple logical decoding plugin tracking closed transactions
  - external tool writes out logs of committed data
  - second tool turns logs into change files

osm2pgsql

https://github.com/openstreetmap/osm2pgsql/

About osm2pgsql

- loads OSM data into PostgreSQL/PostGIS
- updates from minutely diffs
- created in 2006 for map rendering
- today: flexible database layout with Lua configuration

OSM to Simple Feature Model

Node

Way

Relation

WGS84 coordinate

POINT(...)

OSM to Simple Feature Model

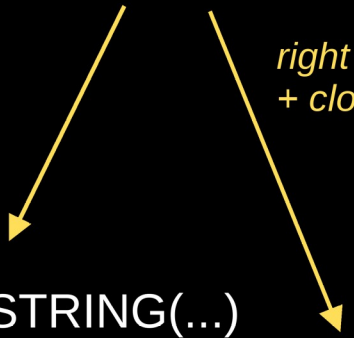
Node

Way

Relation

WGS84 coordinate

ordered list of nodes



right tags
+ closed

POINT(...)

LINESTRING(...)

POLYGON(…)
(without inner rings)

OSM to Simple Feature Model

Node

WGS84 coordinate

POINT(...)

Way

ordered list of nodes

right tags + closed

LINESTRING(...)

POLYGON(…)
(without inner rings)

Relation

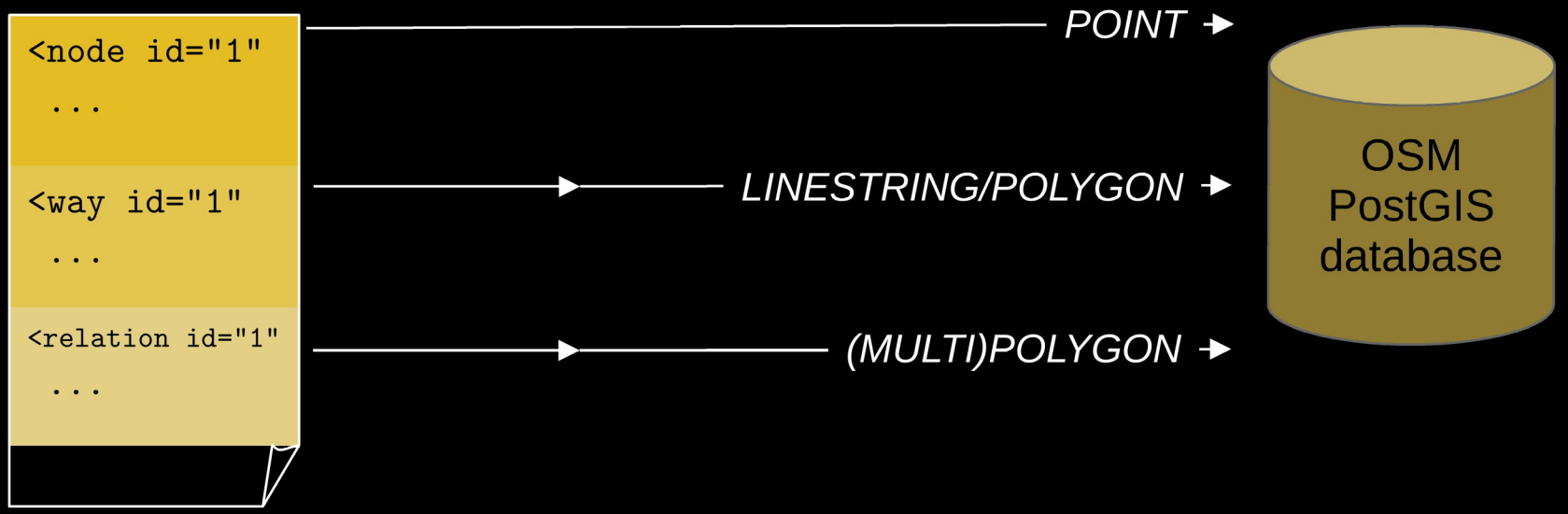
ordered list of OSM elements with role

"type = multipolygon"

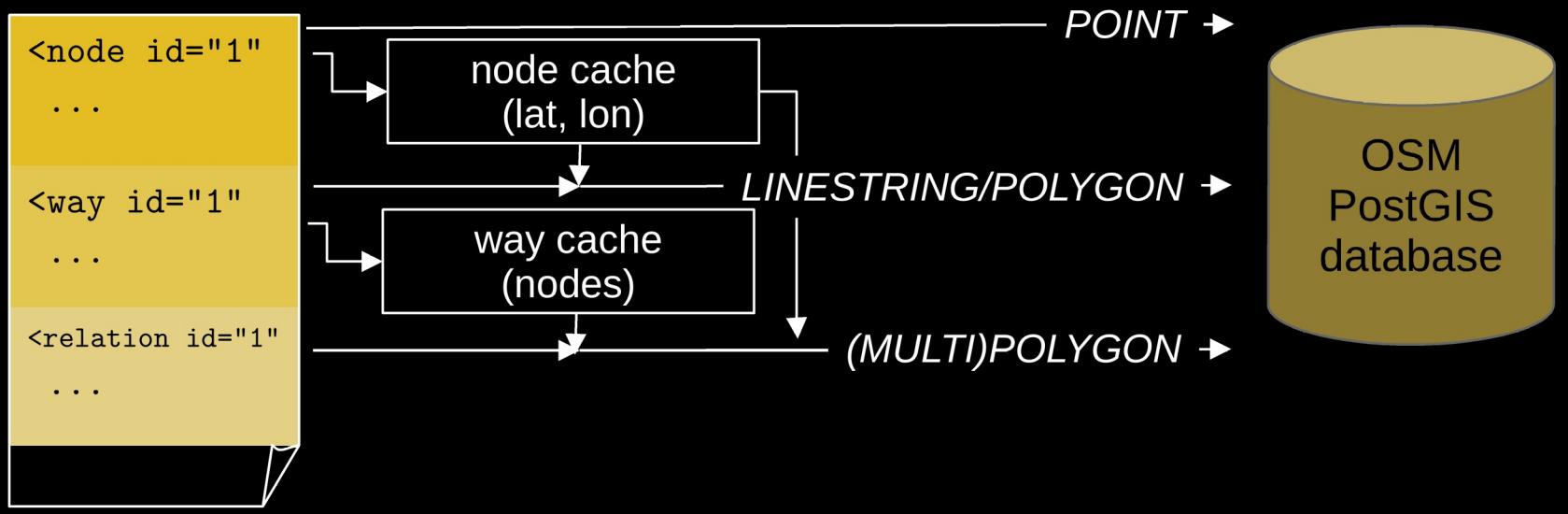
POLYGON(…)
MULTIPOLYGON(...)

?

Pitfalls of OSM data conversion

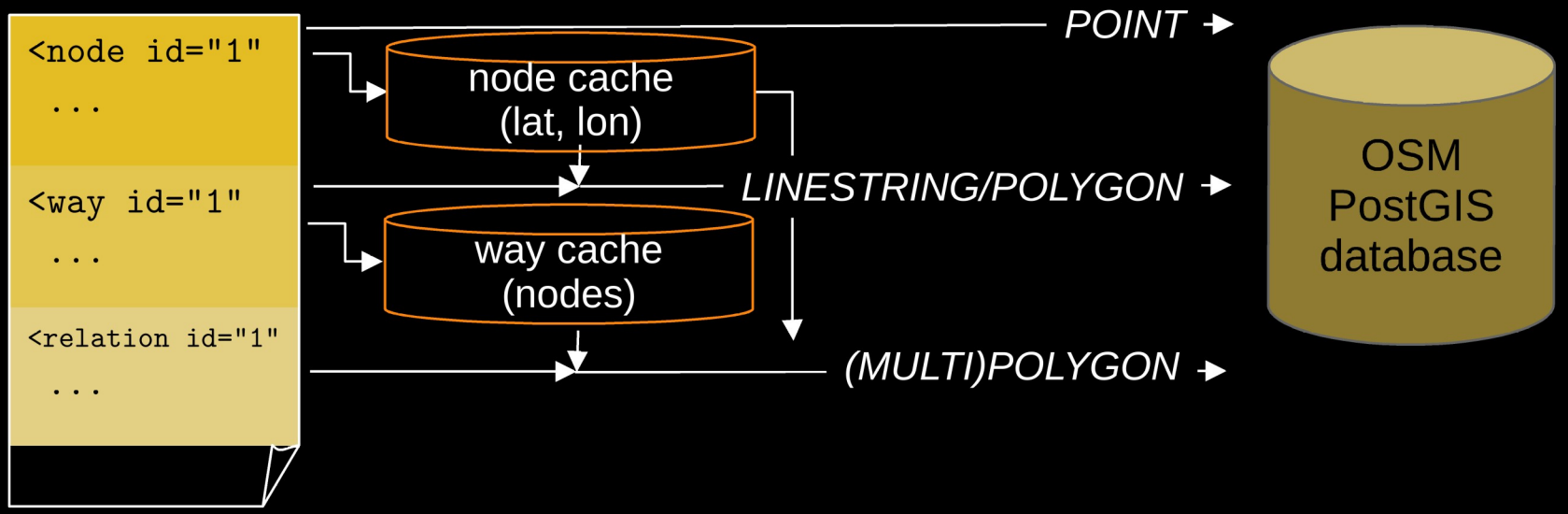


Pitfalls of OSM data conversion



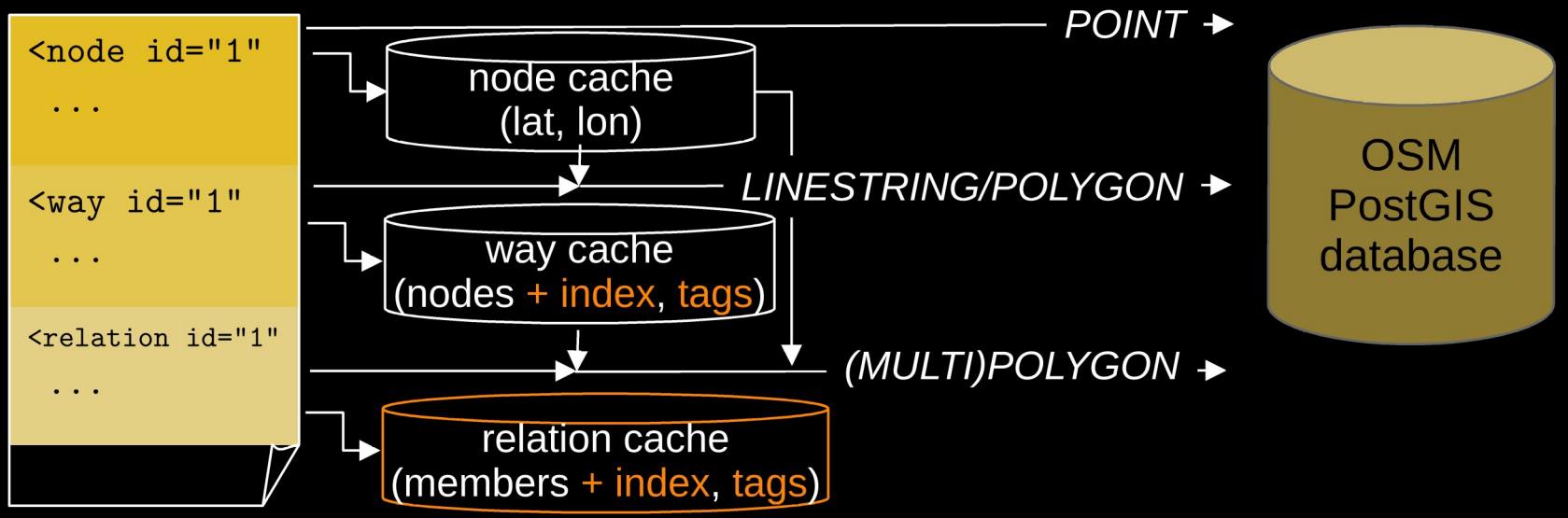
1. Objects cannot be processed independently.

Pitfalls of OSM data conversion



2. Updates do not contain referenced objects.

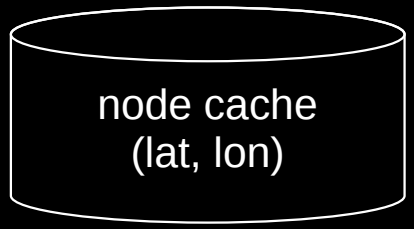
Pitfalls of OSM data conversion



3. Geometries may change when a referenced object changes.

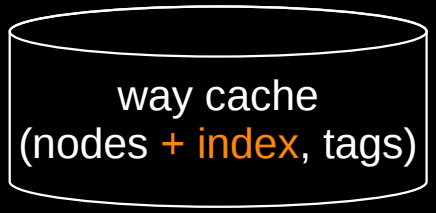
Storing node locations

- as table with `(id, lat, lon)`
  - planet: 490GB + 165GB index
  - writing & read access too slow
- alternative: use external storage
  - simple flat array file



Forwarding node changes to ways

```
SELECT id FROM planet_osm_ways
WHERE ARRAY[34]::bigint[] && nodes;
```



- GIN index over `nodes` bigint array
  - huge index (planet: 280GB)
  - very slow to create (planet: 7.5h) and use

The 'bucket' index

- fold nodes array into node id blocks:

```
CREATE FUNCTION index_bucket(int8[]) RETURNS int8[] AS $$
     SELECT ARRAY(SELECT DISTINCT unnest($1)/32)
$$ LANGUAGE SQL IMMUTABLE;
```

- create index over folded array:

```
CREATE INDEX ON planet_osm_ways USING GIN (index_bucket(nodes))
```

- size: 17GB
  creation time: 2.5h
  average number of ways retrieved: 10

Using GIN indexes

```
nominatim=# EXPLAIN ANALYSE SELECT * FROM planet_osm_ways WHERE ARRAY[34]::bigint[] && nodes;

                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on planet_osm_ways  (cost=34184.30..11583638.00 rows=4209974 width=194)
                                      (actual time=0.029..0.031 rows=0 loops=1)
   Recheck Cond: ('{34}'::bigint[] && nodes)
   ->  Bitmap Index Scan on planet_osm_ways_nodes_idx  (cost=0.00..33131.80 rows=4209974 width=0)
                                                       (actualtime=0.025..0.027 rows=0 loops=1)
         Index Cond: (nodes && '{34}'::bigint[])
 Planning Time: 0.158 ms
 Execution Time: 0.076 ms
(6 rows)
```

Using GIN indexes with JIT

```
osm2pgsql=# EXPLAIN ANALYSE SELECT * FROM planet_osm_ways WHERE ARRAY[34]::bigint[] && nodes;
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on planet_osm_ways  (cost=34184.30..11583638.00 rows=4209974 width=194)
                                      (actual time=6.630..6.632 rows=0 loops=1)
   Recheck Cond: ('{34}'::bigint[] && nodes)
   ->  Bitmap Index Scan on planet_osm_ways_nodes_idx  (cost=0.00..33131.80 rows=4209974 width=0)
                                                       (actual time=6.604..6.606 rows=0 loops=1)
         Index Cond: (nodes && '{34}'::bigint[])
 Planning Time: 0.159 ms
 JIT:
   Functions: 2
   Options: Inlining true, Optimization true, Expressions true, Deforming true
   Timing: Generation 0.768 ms, Inlining 0.000 ms, Optimization 0.000 ms, Emission 0.000 ms, Total 0.768 ms
 Execution Time: 7.640 ms
```

Access patterns

Import:

- dump large amounts of data (exclusive access)
- reads only on finished tables
- durability is not important

Rendering access:

- many readers accessing usually via geometry index (many short-running queries)
- (optional) one writer for updates (minutely)
- durability and consistency matters

Nominatim

https://github.com/osm-search/Nominatim/

About

- started in 2009 as proof-of-concept for geocoding based on PostgreSQL
- main geocoder for osm.org since 2012
- osm2pgsql as importer + heavy PL/pgSQL scripting + PHP for queries
- main goals:
  - 100% OSM planet-wide
  - language-independent (including exotic scripts)
  - minutely updateable
  - fast (around 1k request/s on 3 servers)

Database layout

	Place table
	• object type • names • centroid • geometry • ...

Database layout

	Place table
	• object type • names • centroid • geometry • ...

	Search table

	• search term array over
	  • name terms
	  • address terms
	• plus filter columns (geometry, country, ...)

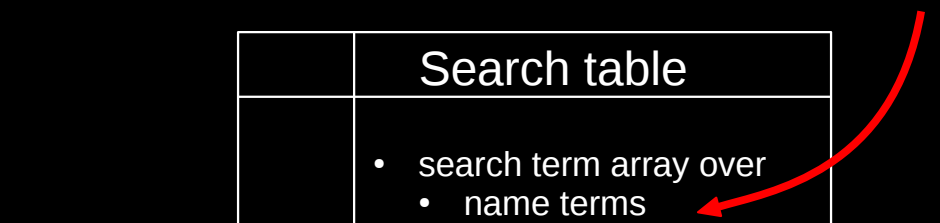
Database layout

	Place table
	• object type • names • centroid • geometry • ...

	Search table

	• search term array over
	  • name terms
	  • address terms
	• plus filter columns (geometry, country, ...)

GIN index
≃ inverted index



Database layout

Place table

- object type
- names
- centroid
- geometry
- ...

Word table

- searchable terms → ID
- token types (word, housenumber, postcode, ...)

Search table

- search term array over
  - name terms
  - address terms
- plus filter columns (geometry, country, ...)

GIN index
≃ inverted index

Address terms

	Address table
	• place object • address object • hierarchy

Address = names of nearby places to help locate object

- computed from object in place table itself
- extensive spatial operations by PostGIS

Place geometries

	Place table
	• object type • names • centroid • geometry • ...

```
Table: 180GB (ca. 300M rows)
GIST(geometry) index: 30GB
```

- spatial search *really* slow
  - mixing country-sized polygons with address points
- no obvious partitioning dimension
  - import: needs partitioning by country
  - search: needs partitioning by type

Partial Geometry Indexes

```
CREATE INDEX ON placex USING gist (geometry)
 WHERE rank_address between 1 and 25 AND ST_GeometryType(geometry) in ('ST_Polygon','ST_MultiPolygon');
CREATE INDEX ON placex USING gist (geometry)
  WHERE address is not null AND rank_search = 30 AND ST_GeometryType(geometry) in ('ST_Polygon','ST_MultiPolygon');
CREATE INDEX ON placex USING gist (geometry)
  WHERE osm_type = 'N' AND rank_search < 26 AND class = 'place' AND type != 'postcode';
CREATE INDEX ON placex USING gist (geometry)
  WHERE osm_type = 'W' AND rank_search >= 26;
CREATE INDEX ON placex USING gist (geometry)
  WHERE St_GeometryType(geometry) in ('ST_Polygon', 'ST_MultiPolygon')
    AND rank_address between 4 and 25 AND type != 'postcode'
    AND name is not null AND indexed_status = 0 AND linked_place_id is null;
```

- highly specialised for given query
- too easy to forget to update the index when changing queries
- views to the rescue?

Using helper tables

- helper tables for finding address objects
  - only relevant OSM objects
  - split geometries
  - per-country partitioning
- access through handcrafted PL/pgSQL

```
CREATE OR REPLACE FUNCTION getNearestNamedPlacePlaceId(in_partiti
  AS $$
DECLARE
  parent BIGINT;
BEGIN
  IF not token_has_addr_place(token_info) THEN
    RETURN NULL;
  END IF;

{% for partition in db.partitions %}
  IF in_partition = {{ partition }} THEN
    SELECT place_id INTO parent FROM search_name_{{ partition }}
      WHERE token_matches_place(token_info, name_vector)
            AND centroid && ST_Expand(point, 0.04)
            AND address_rank between 16 and 25
      ORDER BY ST_Distance(centroid, point) ASC limit 1;
    RETURN parent;
  END IF;
{% endfor %}

  RAISE EXCEPTION 'Unknown partition %', in_partition;
END
$$
LANGUAGE plpgsql STABLE;
```

Database import

❶ *Batch-load place table*

	Place table
	• object type • names • centroid • geometry • indexed_status = 2 • ...

	Word table
	• searchable terms → ID • token types (word, housenumber, postcode, ...)

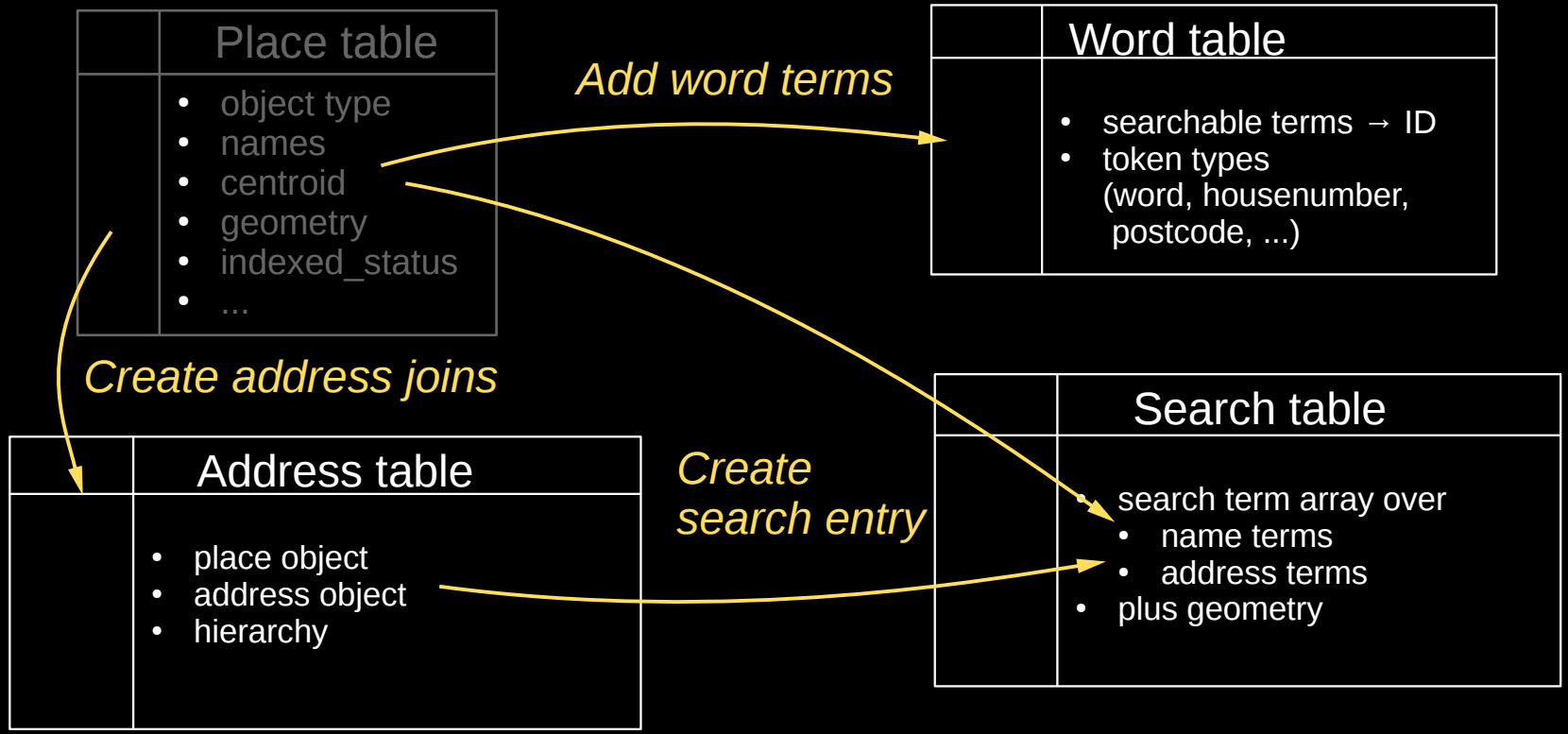
	Address table
	• place object • address object • hierarchy

	Search table

	• search term array over
	  • name terms
	  • address terms
	• plus geometry

Database import

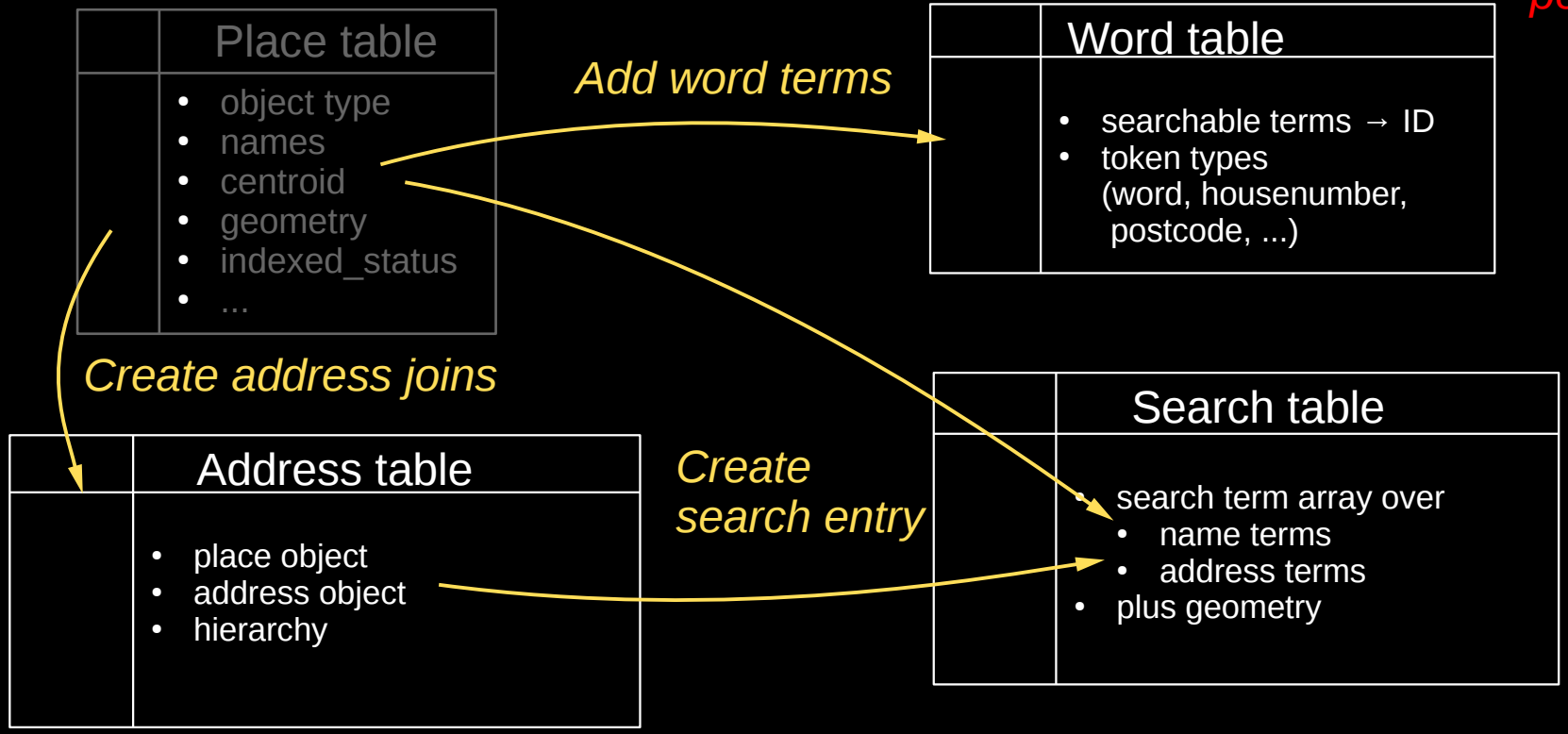
❷ *TRIGGER:* `UPDATE place SET indexed_status = 0`



Database import

❷ *TRIGGER:* `UPDATE place SET indexed_status = 0`

one UPDATE per row



Comparing to Lucene-based DBs

- Geocoding is not full-text search
  - proper names with "interesting" spelling/pronunciation rules
  - mixed languages
  - special address grammar + standard search queries
  - many abbreviations

Quality depends on clever query parsing

Comparing to Lucene-based DBs

Limiting factor: GIN index over int array

- fast enough for queries
- not fast enough for search-as-you-type (until disproven)

The "inverted" index

	Search table

	• search term array over
	  • name terms `int[]`
	  • address terms `int[]`
	• plus filter columns (geometry, country, ...)

```
Table: 65GB
GIN(name) index: 5GB
GIN(address) index: 20GB
```

- needs per-term estimates for correct index use
- manual term counts in Word table
- disable index use through SQL:

```
SELECT * FROM search_name
WHERE name_vector @> ARRAY[2]
AND array_cat(address_vector, ARRAY[]::int) @> ARRAY[45, 78, 45]
```

Thank you, PostgreSQL!

Get involved with OpenStreetMap: `https://osm.org/fixthemap`

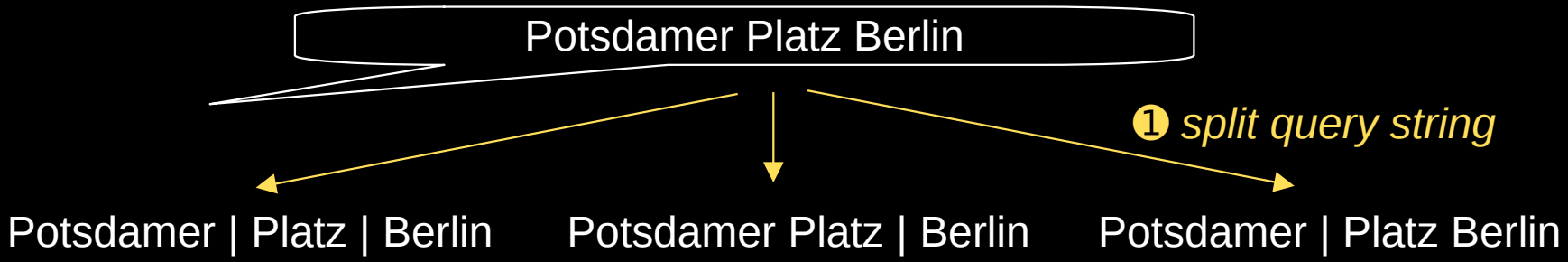
Website: `https://github.com/openstreetmap/openstreetmap-website/`
osm2pgsql: `https://osm2pgsql.org`
Nominatim: `https://nominatim.org`

Sarah Hoffmann lonvia@denofr.de

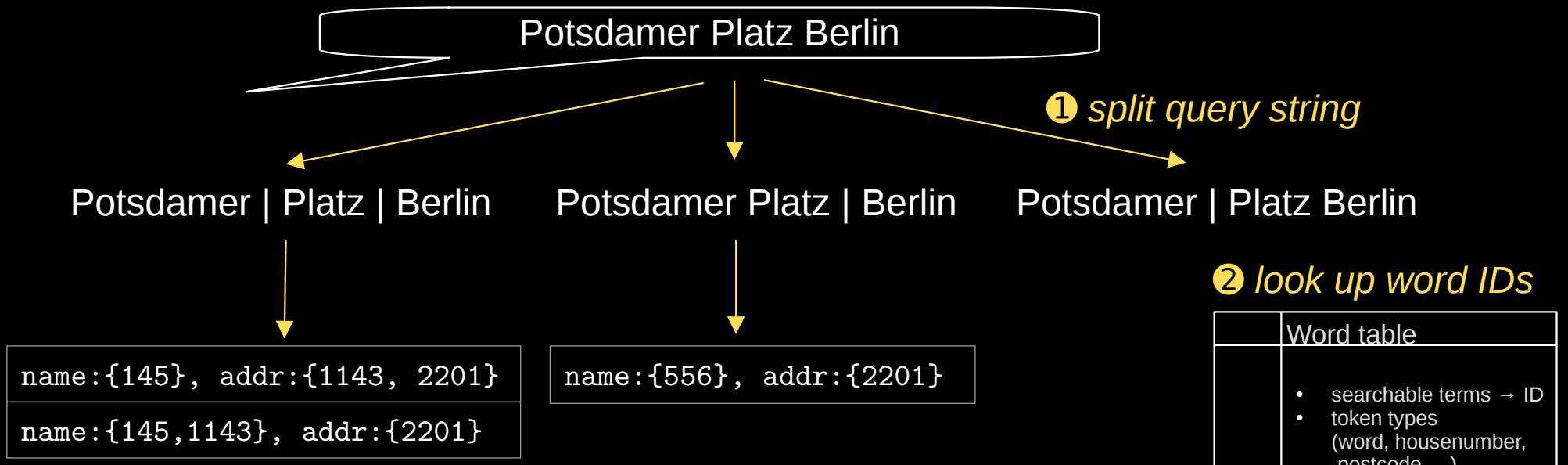
Searching

Potsdamer Platz Berlin

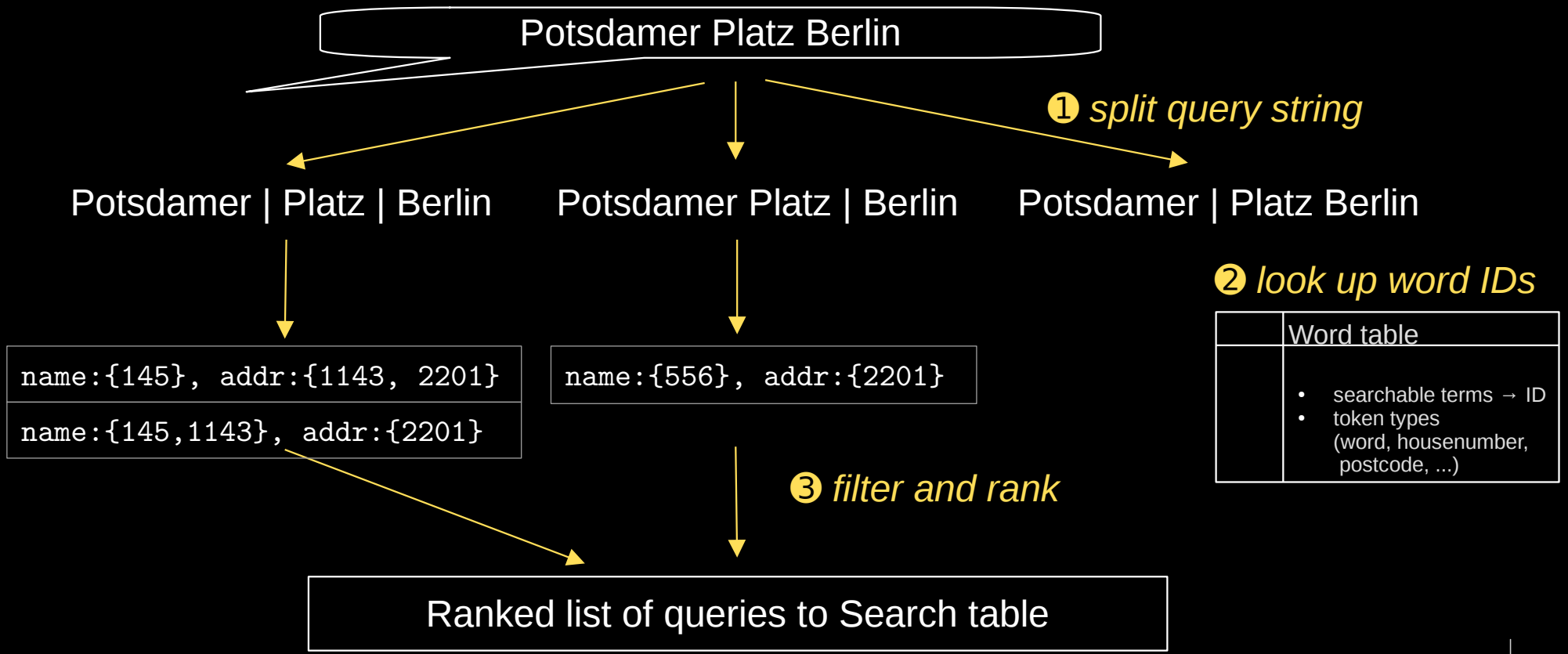
Searching


Potsdamer Platz Berlin
❶ split query string
Potsdamer | Platz | Berlin
Potsdamer Platz | Berlin
Potsdamer | Platz Berlin

Searching


Potsdamer Platz Berlin
❶ split query string
Potsdamer | Platz | Berlin
Potsdamer Platz | Berlin
Potsdamer | Platz Berlin
name:{145}, addr:{1143, 2201}
name:{145,1143}, addr:{2201}
name:{556}, addr:{2201}
❷ look up word IDs
Word table
searchable terms → ID
token types (word, housenumber, postcode, ...)

Searching


Potsdamer Platz Berlin
❶ split query string
Potsdamer | Platz | Berlin
Potsdamer Platz | Berlin
Potsdamer | Platz Berlin
❷ look up word IDs
Word table
• searchable terms → ID
• token types (word, housenumber, postcode, ...)
name:{145}, addr:{1143, 2201}
name:{145,1143}, addr:{2201}
name:{556}, addr:{2201}
❸ filter and rank
Ranked list of queries to Search table

Search Queries – Getting Results