An Elasticsearch Crash Course

Found

Elasticsearch is Everywhere

Why?

Elasticsearch

Jared and Corin @ flickr http://bit.ly/1qHHMPu

Some Use Cases

- Searching pieces of pure text (books, legal documents, blog posts…)
- Searching text + structured data (products, user profiles, application logs)
- Pure aggregated data (statistics, metrics, etc.)
- Geo Search
- Distributed JSON Document DB (Anything)

At a High Level

- Is a database, like any other
- Document Oriented
- Clusters
- Built on Lucene
- Built on an IR foundation
- Can perform fancy tricks with inverted indexes and automata

The Basics of the ES API

Getting Data Into ES

Storing a Document

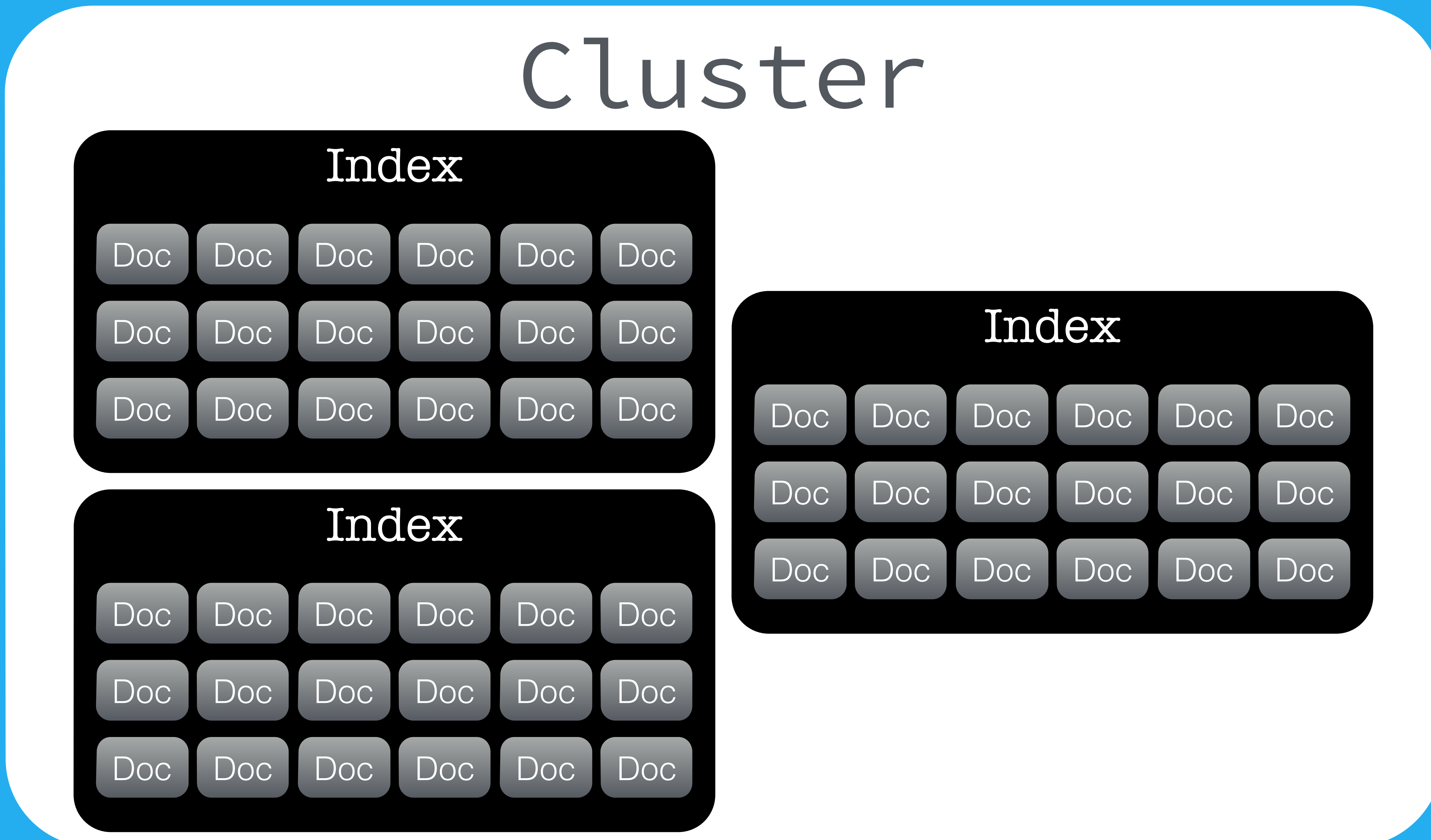
Verb — Index — Type — DocID

```
curl -XPUT http://localhost:9200/literature/quote/one -d'
 {
   "person": "Jack Handy",
   "said": "The face of a child can say it all, especially the mouth part of the face"
 }'
```

Document

Where does the document go?

Indexes live in the cluster
Documents live in indexes



Key Nouns

Documents

- A single Arbitrary JSON object
- Stored as a text blob + indexes on fields
- All fields get an inverted index(es)

```
{
  "person": "Sam",
  "foods": ["Green eggs", "ham"]
  "likeswith": {
     "place": "house",
     "companion": “mouse”,
     "age": 10
   }
}
```

Found

Types

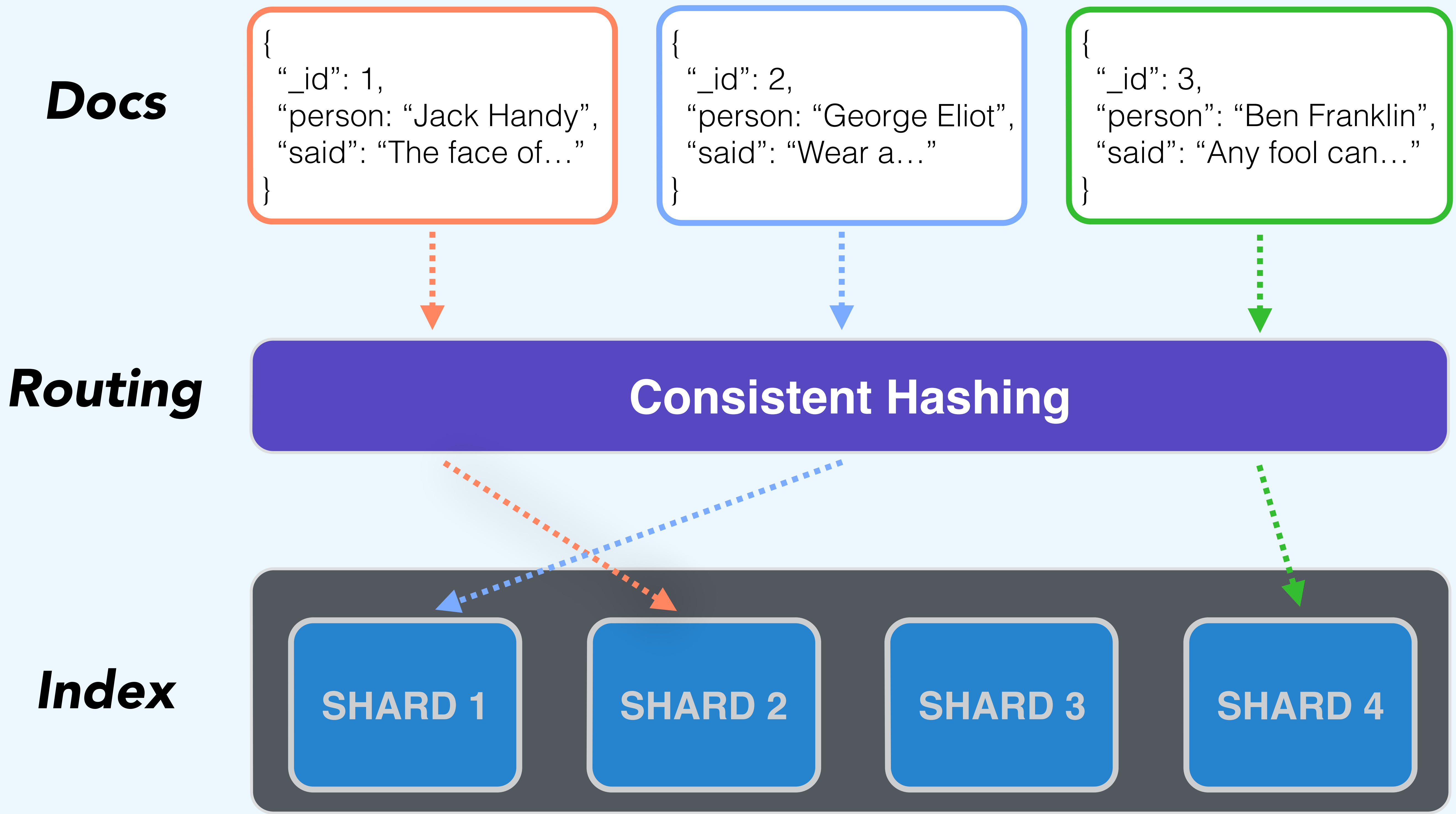
- Defines the schema for documents
- Defines indexing rules as well

```
{
  "human" : {
    "properties" : {
      "person" : {"type" : "string"},
      "age" :   {"type" : "integer"}}}}
```

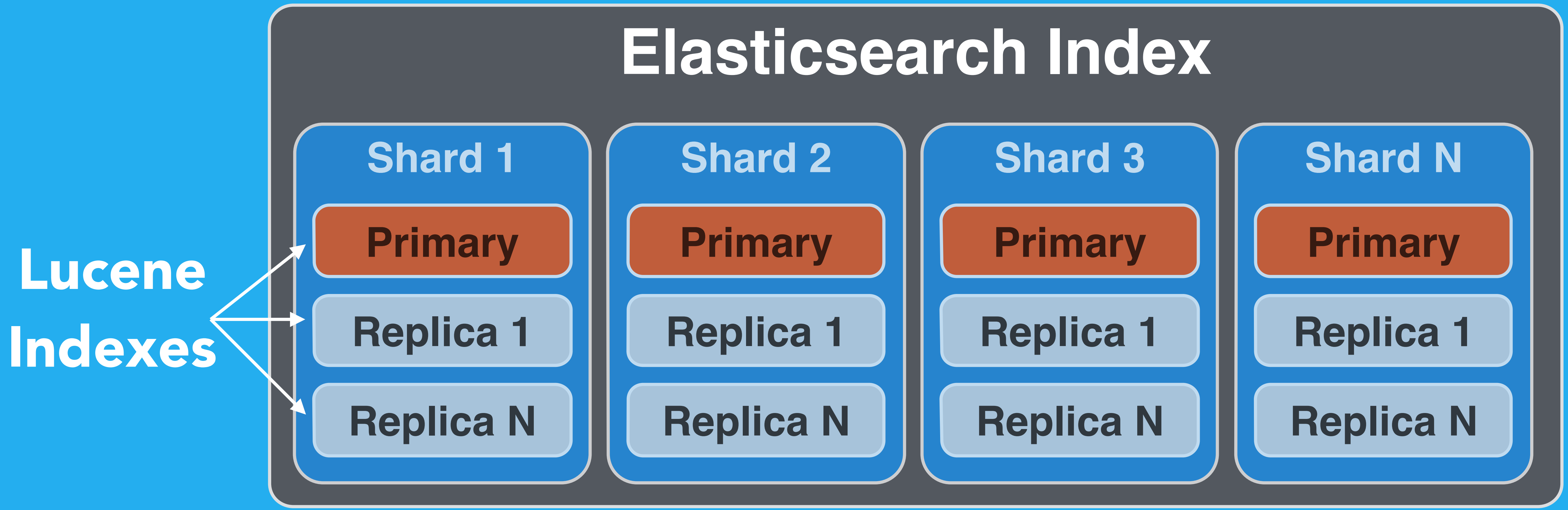
Indexes

- Largest building block in ES
- Container for documents / types
- Composable

Document Storage



Inside an Elasticsearch Index



Each primary or replica shard is a Lucene index

Querying

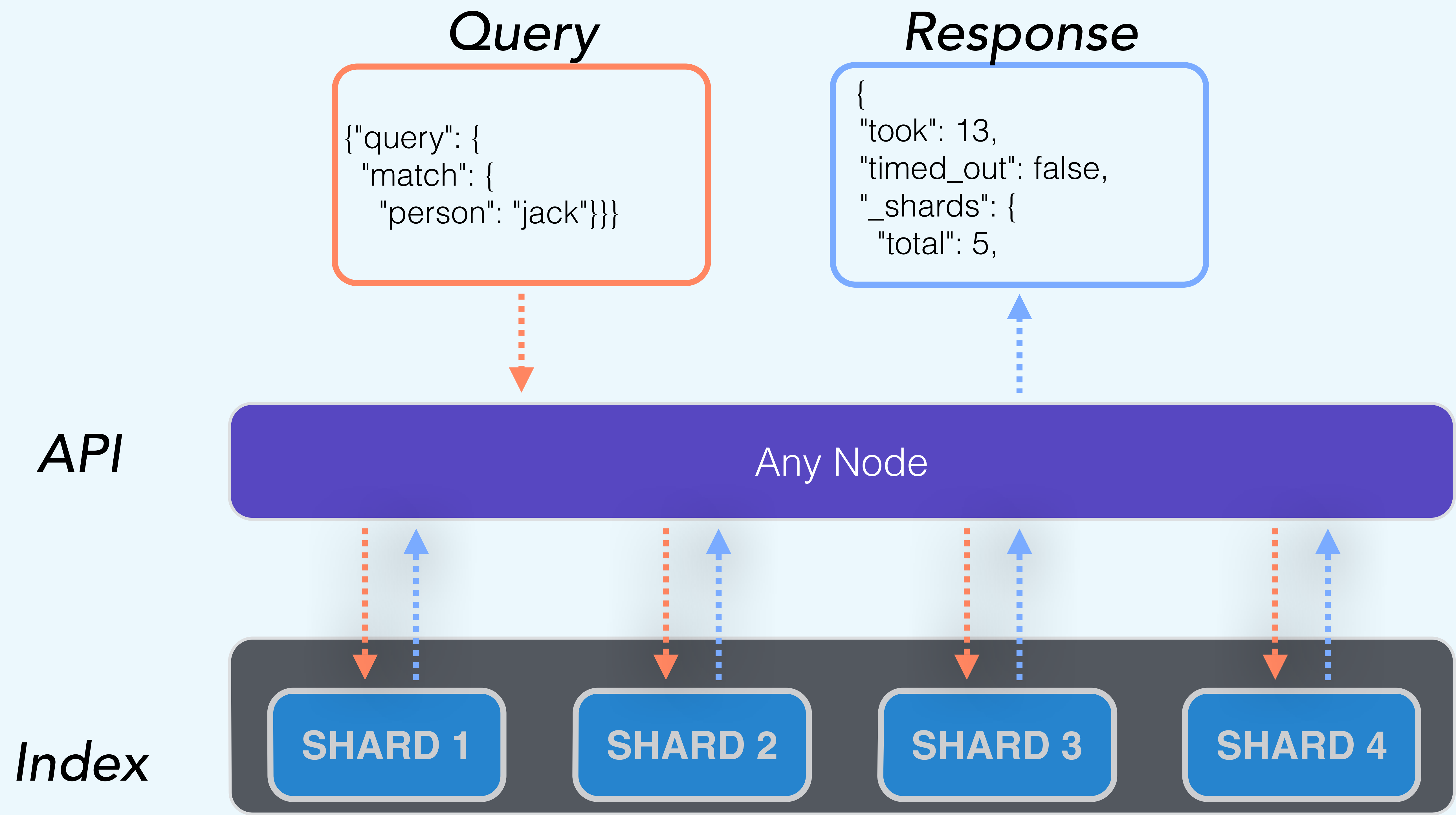
A Simple Query

Verb Index Type Action

```
curl -XPOST http://localhost:9200/literature/quote/_search -d'
 {
   "query": {
     "match": {
       "person": "jack"}}}'
```

Search Body

The Search API in Action



Natural Language Search

Everything should run in sub linear time, usually O(log n)

Martin Fisch @ flickr http://bit.ly/1l4sII3

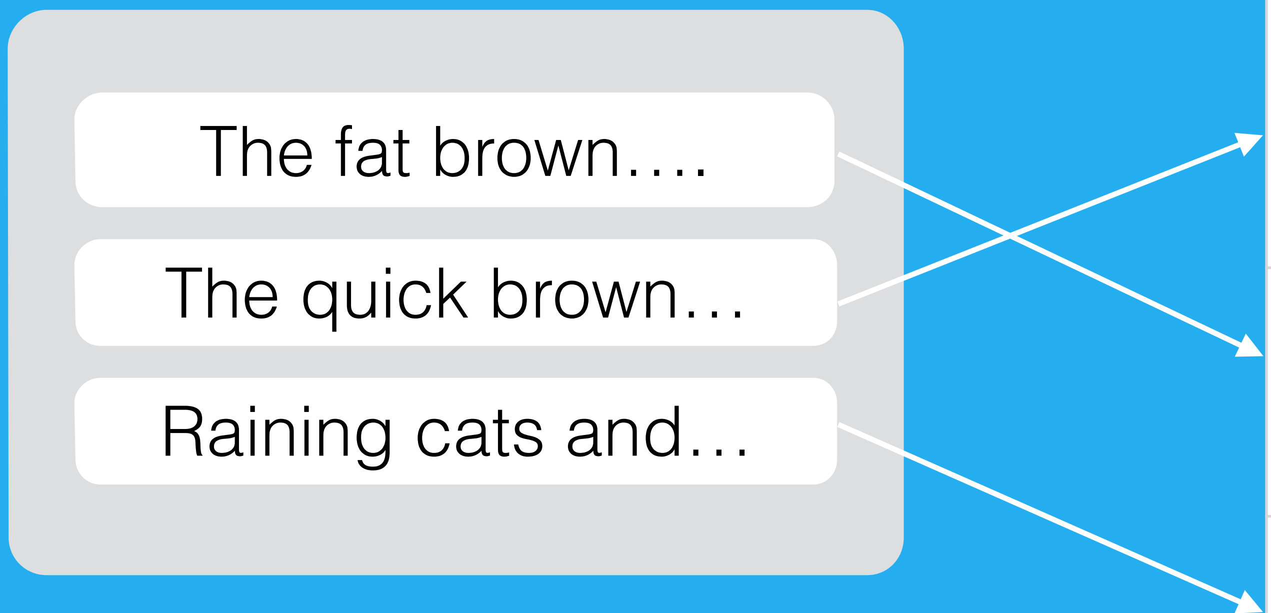
Found

Think of Your Indexes as Trees

Martin Fisch @ flickr http://bit.ly/1l4sII3

Working with Data in SQL

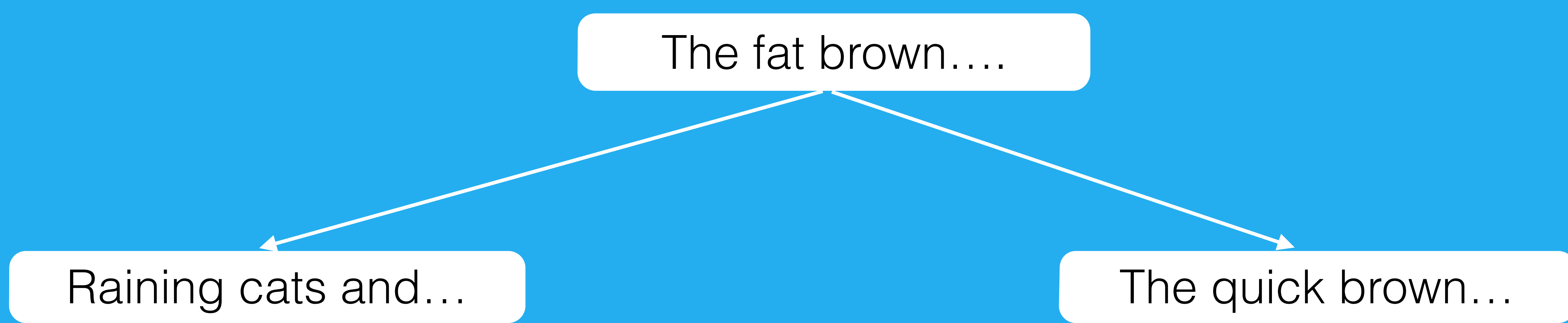
Index on "phrase"



"phrases" table

id	phrase
1	The quick brown fox jumped over the lazy dog
2	The fat brown dog
3	Raining cats and dogs

SQL Index as a B-Tree



Fast Prefix Search

SELECT * FROM phrases WHERE phrase LIKE 'The%'

Found

Standard BTree-based indexes are fast at:

- Exact matches
- Prefix matches

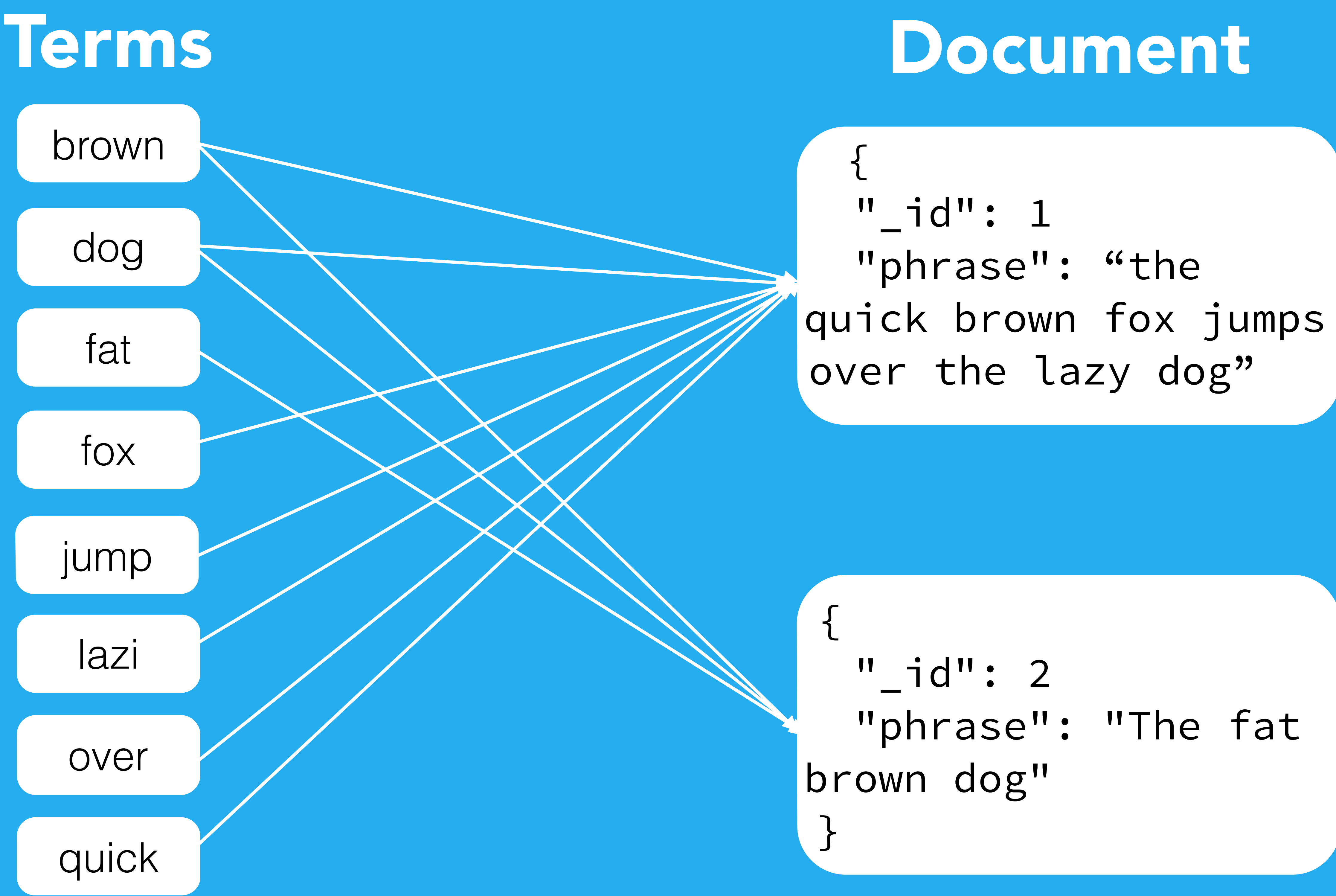
How well does the previous example work given a search for "dog"?

Slow Scan Search

```
SELECT * FROM phrases WHERE phrase LIKE '%dog%'
```

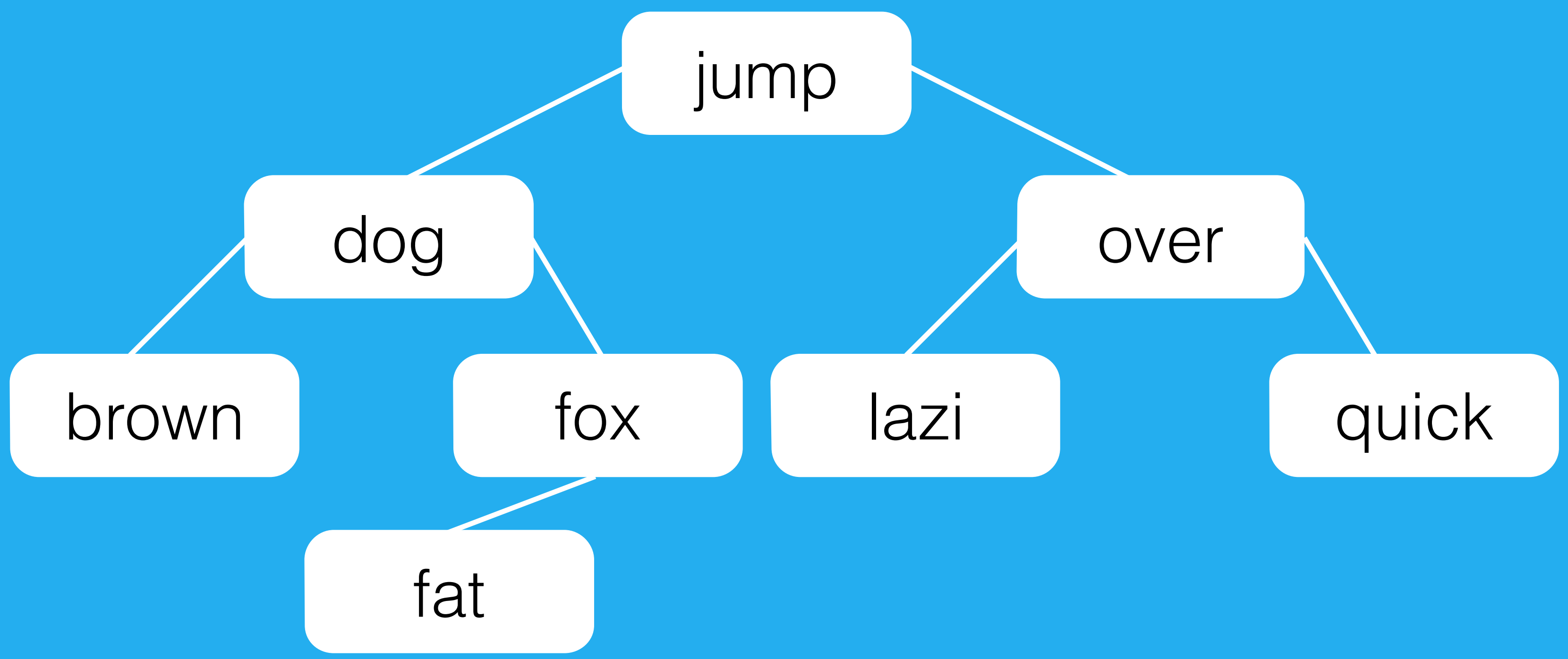
Found

An Inverted Index



An Inverted Index as a Tree

Terms



Sequential Scan City

SELECT * FROM phrases WHERE phrase ILIKE ‘dog’

Uses an index!

SELECT * FROM phrases WHERE LOWER(phrase) =LOWER('dog')

Found

Making the index

```
CREATE INDEX
lcase_phrase_idx ON
phrases (LOWER(phrase));
```

Found

Text In, Terms Out

"Some kind of Text"

↓

ANALYZER

↓

["text", "of", "kind", "some"]

Analysis

"The quick brown fox jumps over the lazy dog"

↓

Snowball Analyzer

↓

["quick"^{2}, "brown"^{3}, "fox"^{4}, "jump"^{5}, "over"^{6}, "lazi"^{7}, "dog"^{8}]

Stemming and Stopwords

"I jump while she jumps and laughs"

↓

Snowball Analyzer

↓

["i"^{1} "jump"^{2}, "while"^{3}, "she"^{4}, "jump"^{5}, "laugh"^{7}]

NGrams

"news"

↓

NGram Analyzer

↓

["n", "e", "w", "s", "ne", "ew", "ws"]

An NGram Search

Query

["n", "e", "w", "ne", "ew"]

Good Match

["n", "e", "w", "s", "ne", "ew", "ws"]

Poor Match

["s", "t", "e", "w", "s", "st", "te", "ew", "ws"]

Path Hierarchy

"/var/lib/racoons"

↓

Path Hierarchy Analyzer

↓

["/var", "/var/lib", "/var/lib/racoons"]

Inverted Index Highlights

- M Terms map to N documents
- Still uses trees, but by breaking up text, performance is gained!
- String broken up into linguistic terms (usually words)
- Postgres users can do this (in a simple form)

List of ES Analysis Tools

Analyzers

- standard analyzer
- simple analyzer
- whitespace analyzer
- stop analyzer
- keyword analyzer
- pattern analyzer
- language analyzers
- snowball analyzer
- custom analyzer

+ Plugins!

Tokenizers

- standard tokenizer
- edge ngram tokenizer
- keyword tokenizer
- letter tokenizer
- lowercase tokenizer
- ngram tokenizer
- whitespace tokenizer
- pattern tokenizer
- uax email url tokenizer
- path hierarchy tokenizer
- classic tokenizer
- thai tokenizer

Token Filters

- standard token filter
- ascii folding token filter
- length token filter
- lowercase token filter
- uppercase token filter
- ngram token filter
- edge ngram token filter
- porter stem token filter
- shingle token filter
- stop token filter
- word delimiter token filter
- stemmer token filter
- stemmer override token filter
- keyword marker token filter
- keyword repeat token filter
- kstem token filter
- snowball token filter
- phonetic token filter
- synonym token filter
- compound word token filter
- reverse token filter
- elision token filter
- truncate token filter
- unique token filter
- pattern capture token filter
- pattern replace token filter
- trim token filter
- limit token count token filter
- hunspell token filter
- common grams token filter
- normalization token filter
- cjk width token filter
- cjk bigram token filter
- delimited payload token filter
- keep words token filter
- classic token filter
- apostrophe token filter

Scoring

=

Relevance

Search Methodology

- Find all the docs using a boolean query
- Score all the docs using a similarity algorithm (TF/IDF)

TF/IDF Boosts When…

- The matched term is ‘rare’ in the corpus
- The term appears frequently in the document

Document Scoring

- Results are ordered based on score (relevance)
- Score based on either TF/IDF or other algorithm
- Custom scoring functions can be sent with query or registered on the server

Document Scoring

- Results are ordered based on score (relevance)
- Score based on either TF/IDF or other algorithm
- Custom scoring functions can be sent with query or registered on the server

Query Types

Phrase Queries

Geo Queries

Numeric Range Queries

More Like This Queries

Autocomplete Queries

Query Types

1. match query
2. multi match query
3. bool query
4. boosting query
5. common terms query
6. custom filters score query
7. custom score query
8. custom boost factor query
9. constant score query
10. dis max query
11. field query
12. filtered query
13. fuzzy like this query
14. fuzzy like this field query
15. function score query
16. fuzzy query
17. geoshape query
18. has child query
19. has parent query
20. ids query
21. indices query
22. match all query
23. more like this query
24. more like this field query
25. nested query
26. prefix query
27. query string query
28. simple query string query
29. range query
30. regexp query
31. span first query
32. span multi term query
33. span near query
34. span not query
35. span or query
36. span term query
37. term query
38. terms query
39. top children query
40. wildcard query
41. text query
42. minimum should match
43. multi term query rewrite

Compose Queries with Boolean / DisMax Queries

Efficient Aggregate Queries: An RDBMS vs Elasticsearch

Elasticsearch is an Information Retrieval (IR) System

An RDBMS is oriented around organizing data

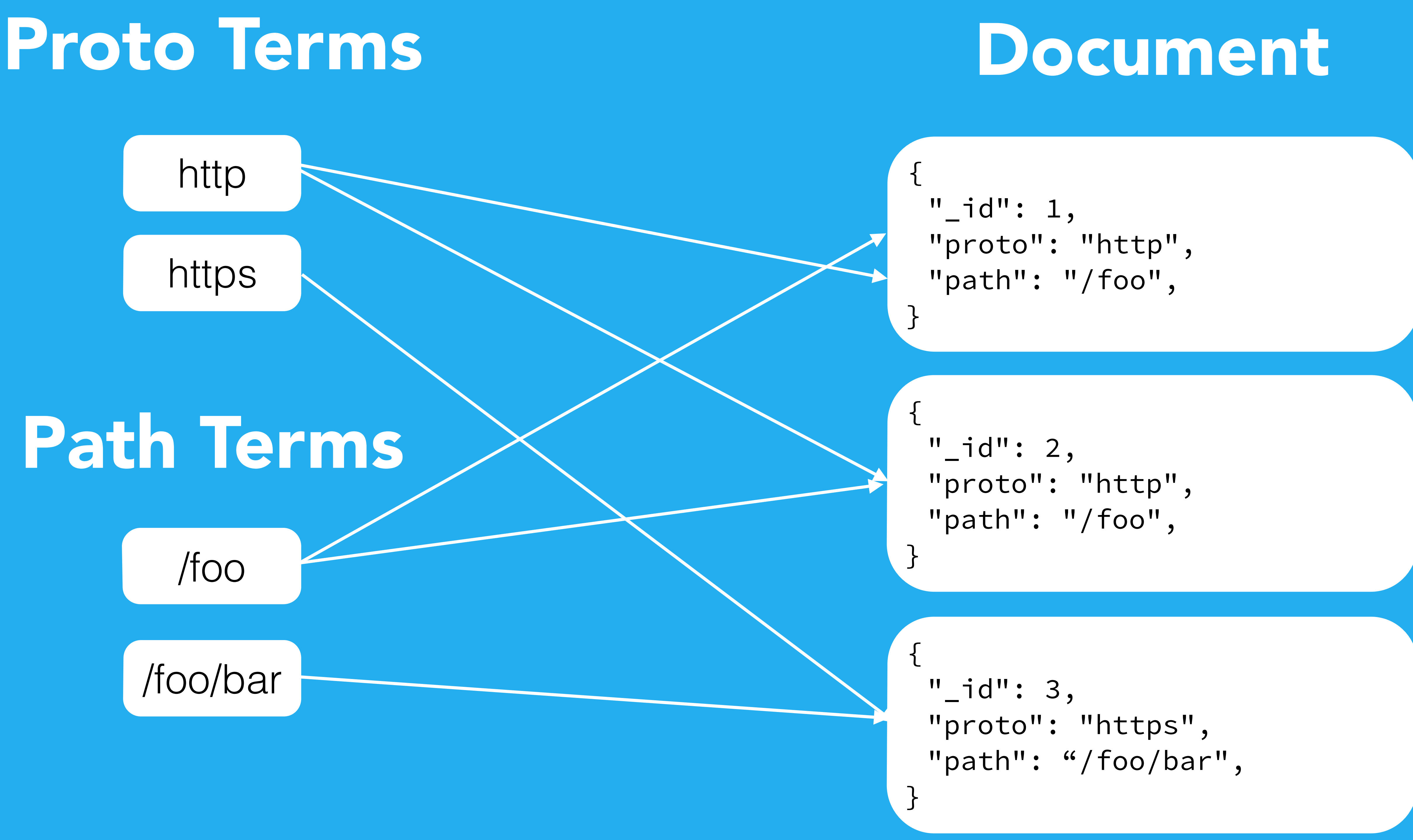
An IR system is oriented around efficient searches

In an RDBMS you create data, then index it

In an IR system you create indexes linked to data

Inverted indexes are fantastically efficient for denormalization!

Inverted Indexes for HTTP Logs



Question:
How many reqs did we get under for each path?

How We Answer It

SQL

```
SELECT
stat,COUNT(*)
FROM logs
WHERE stat IN
('proto','path')
GROUP BY stat
```

ES

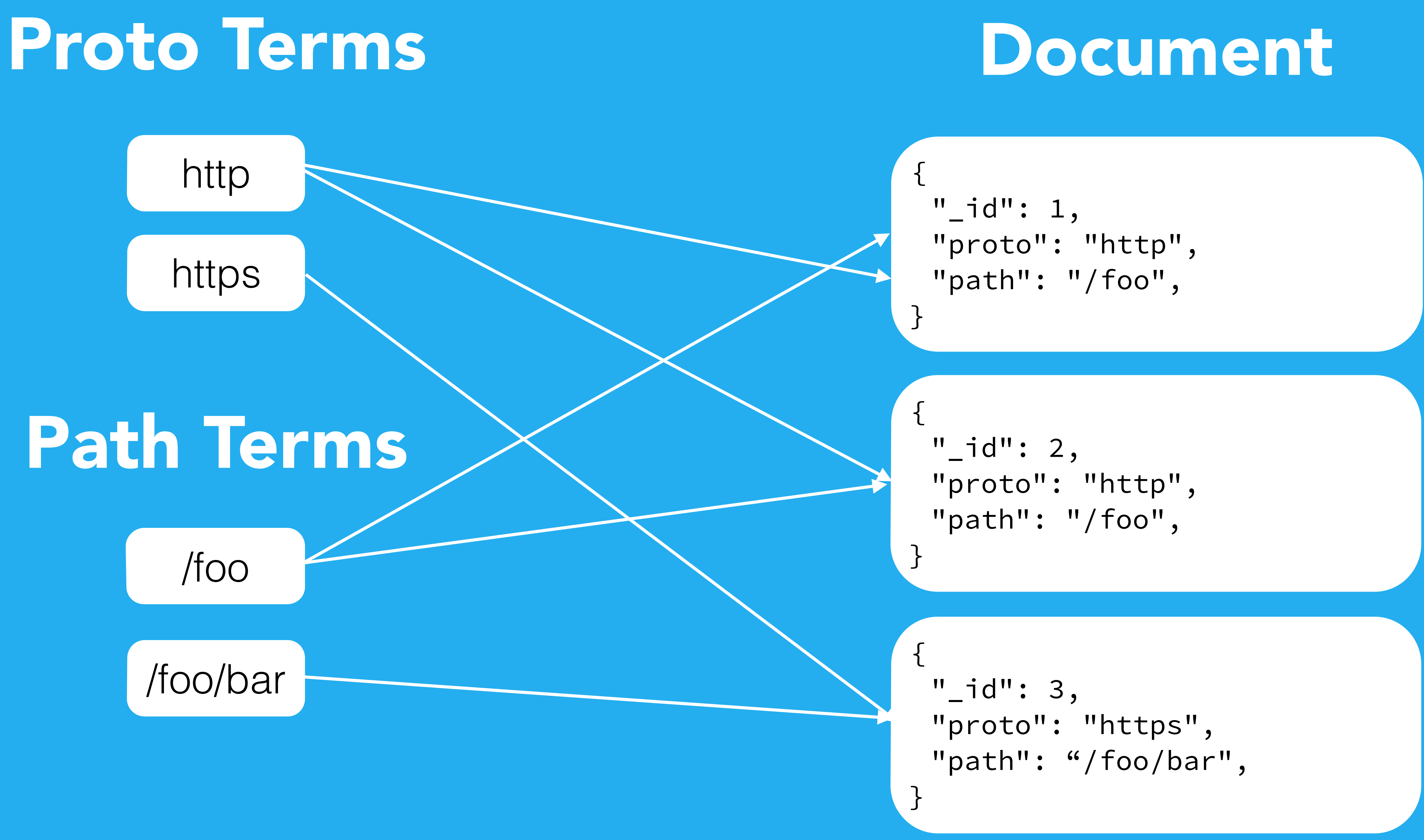
```
{
  "aggs": {
    "path": {
      "terms": {
        "field": "path"} },
    "proto": {
      "terms": {
        "field": "proto"}}}}
```

Found

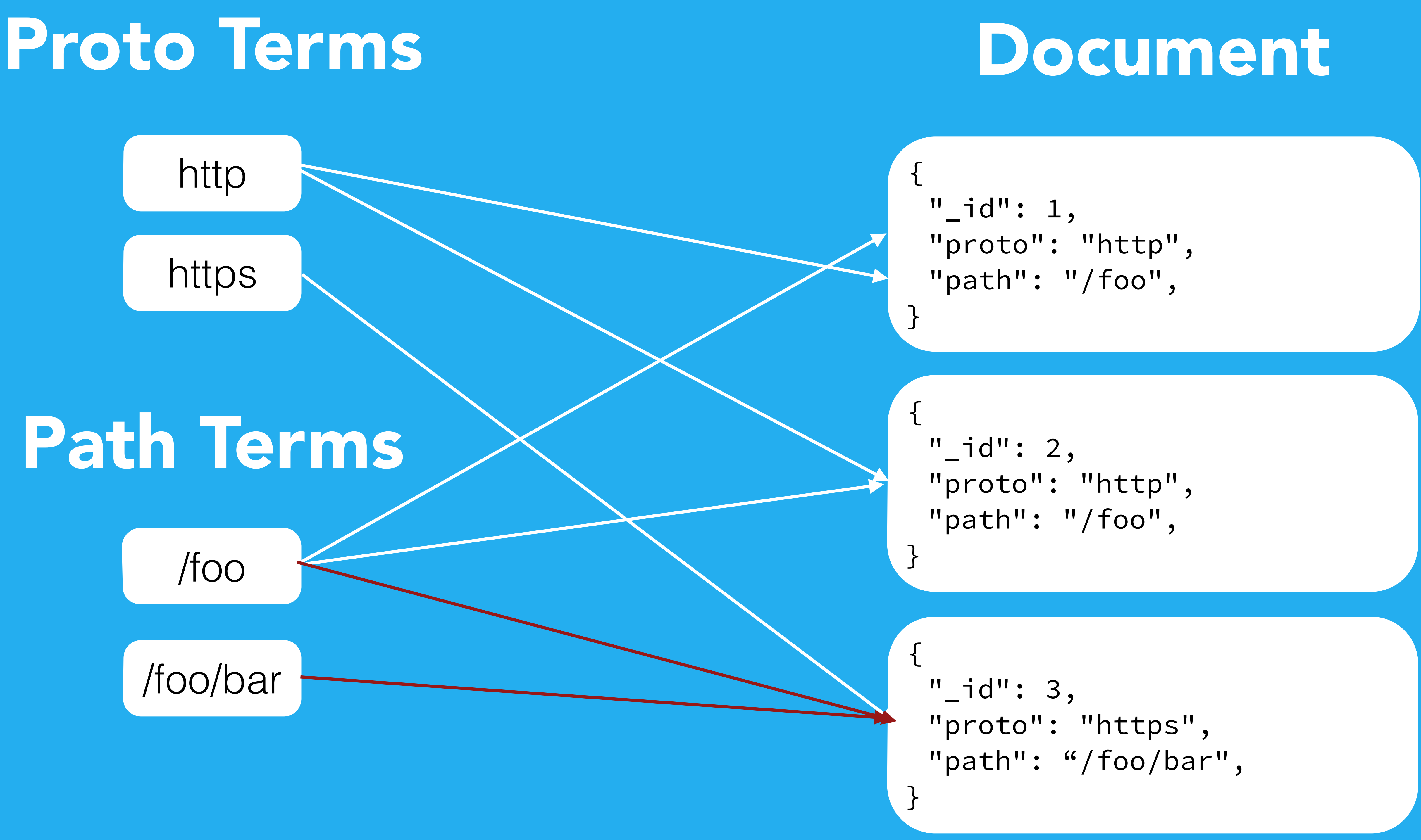
Question:
How many reqs did we get under each different path AND it's parents?

Found

Inverted Indexes for HTTP Logs



Inverted Indexes for HTTP Logs



How We Answer It

SQL

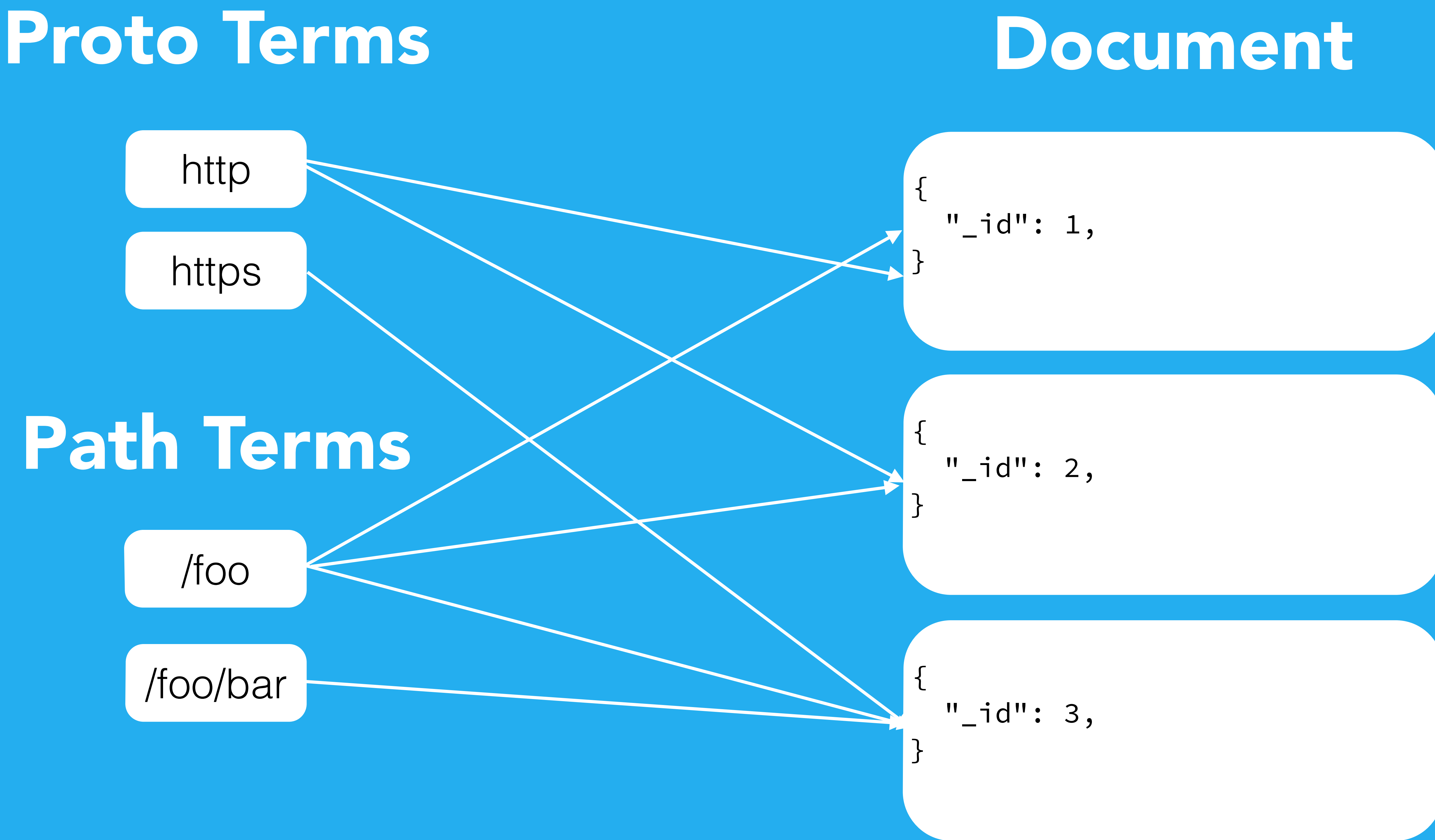
```
S   CT
sta    T(*)
FRO     gs
WH      at IN
(‘   to’,   h’)
  OUP B   at
```

ES

```
{
  "facets": {
    "path": {
      "terms": {
        "field": "path"} },
    "proto": {
      "terms": {
        "field": “proto"}}}}
```

Let’s Save some Space

Space Now Saved!



Reasons to Consider ES

1. Speed

Traditional databases
often are slower for full text search

2. Relevance

Search is all about relevance. A huge array of tools are provided by ES/Lucene to ensure results are relevant.

3. Aggregate Statistics

Elasticsearch can be faster than your RDBMS when it comes to aggregate stats!

4. Search Goodies

Users nowadays expect features like ultra-fast type-ahead search, “Did you mean?”, and “More Like this”

Logstash, an ES Success Story

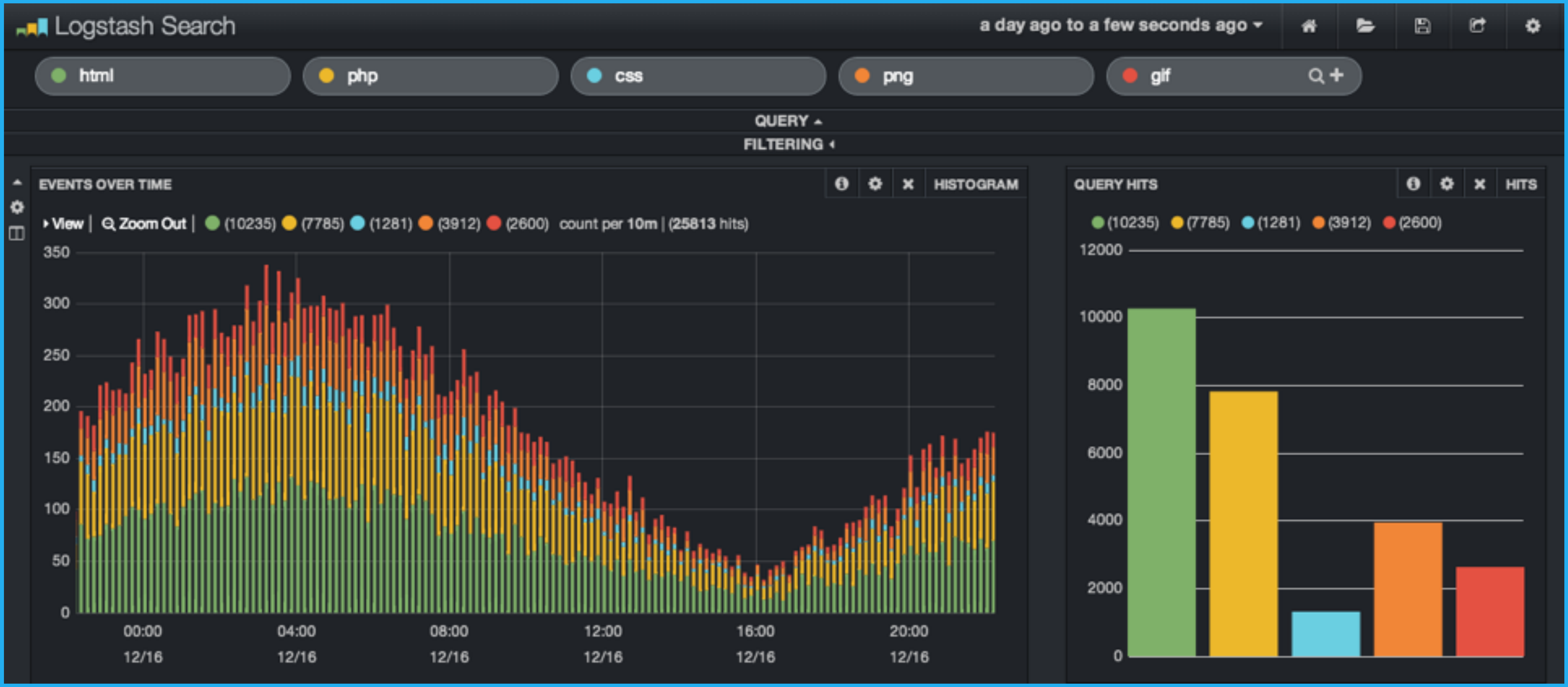
Indexes

- logs-2013-01
- logs-2013-02
- logs-2013-03
- logs-2013-04
- logs-2013-05
- logs-2013-06

Multi Index Query

```
curl http://es.srv/logs-2013-05,logs-2013-06/
_search -d '
"query": "…"
'
```

Kibana + Logstash



Generic Document Store

Document Store Properties

- Distributed
- Excellent read performance / scalability
- Mediocre delete/update performance
- Rich queries on top of document properties

Things ES is bad at

- **Extremely high write environments:** Lucene is not write optimized. You probably won't hit limits here however!
- **Large amounts of document churn:** Deleting and remerging segments can get expensive
- **Transactional Operations:** Lucene is no RDBMS. It is meant for fast, denormalized operations.
- **Primary Store:** Still too new

Thank You!
Check out our hosted ES solution @ http://found.no