

# Make Your Data FABulouS

**Philipp Krenn**

**@xeraa**



**Developer** 🥑

**What is the perfect  
datastore solution?**

# It depends...



# Pick your tradeoffs



# CAP Theorem

# Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

Seth Gilbert\*

Nancy Lynch\*

## Abstract

When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three. In this note, we prove this conjecture in the asynchronous network model, and then discuss solutions to this dilemma in the partially synchronous model.

# Consistent

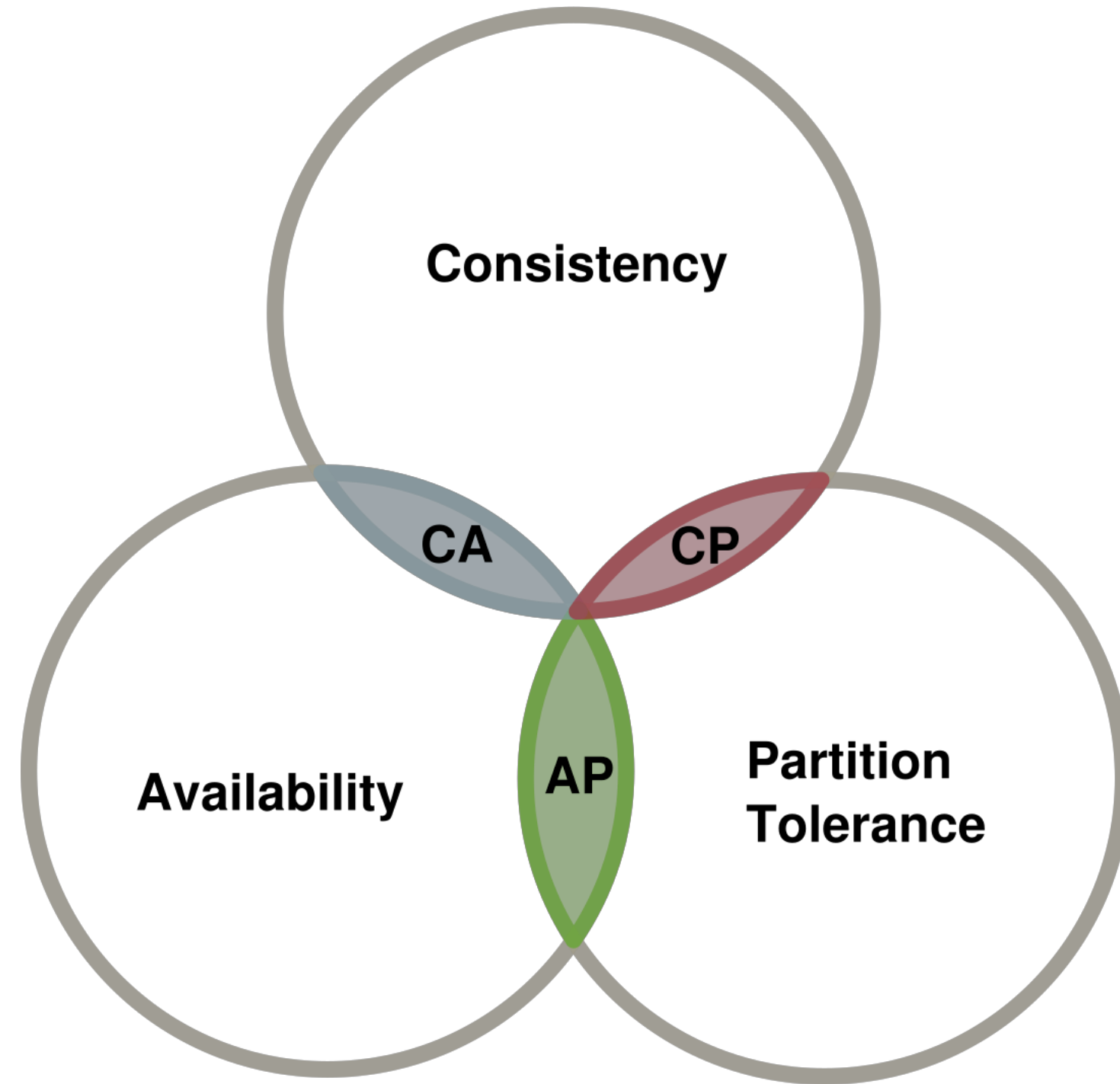
**"[...] a total order on all operations such that each operation looks as if it were completed at a single instant."**

# Available

**"[...] every request received by a non-failing node in the system must result in a response."**

# Partition Tolerant

**"[...] the network will be allowed to lose arbitrarily many messages sent from one node to another."**



[https://berb.github.io/diploma-thesis/original/061\\_challenge.html](https://berb.github.io/diploma-thesis/original/061_challenge.html)



# Misconceptions

**Partition Tolerance is not a choice in a distributed system**

# Misconceptions

**Consistency in ACID is a predicate**

**Consistency in CAP is a linear order**

# Robinson Crusoe



***/dev/null breaks CAP: effect of write are always consistent, it's always available, and all replicas are consistent even during partitions.***

— <https://twitter.com/ashic/status/591511683987701760>

# FAB Theory



# Mark Harwood



# Fast

**Near real-time** instead of batch processing



# Accurate

**Exact** instead of approximate results

# Big

**Parallelization** needed to handle the data

# Say Big Data



# one more time

**Fast** 

**Big** 

**Accurate** ?



elasticsearch

# Shard

## Unit of scale







**"The evil wizard Mondain had attempted to gain control over Sosaria by trapping its essence in a crystal. When the Stranger at the end of Ultima I defeated Mondain and shattered the crystal, the crystal shards each held a refracted copy of Sosaria.**

**<http://www.raphkoster.com/2009/01/08/database-sharding-came-from-uo/>**



# Terms Aggregation

## Word Count

Luke **64**

R2 **31**

Alderaan **20**

Kenobi **19**

Obi-Wan **18**

Droids **16**

Blast **15**

Imperial **15**

## Word Count

Droid **13**

3PO **13**

Princess **12**

Ben **11**

Vader **11**

Han **10**

Jedi **10**

Sandpeople **10**

```
PUT starwars
{
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 0
  }
}
```





Search... (e.g. status:200 AND extension:PHP)

Uses lucene query syntax



Add a filter +



starwars



Data Options



## Metrics

Tag Size

Count

## Buckets

Tags

Aggregation

Terms

Field

word.keyword

Order By

metric: Count

Order

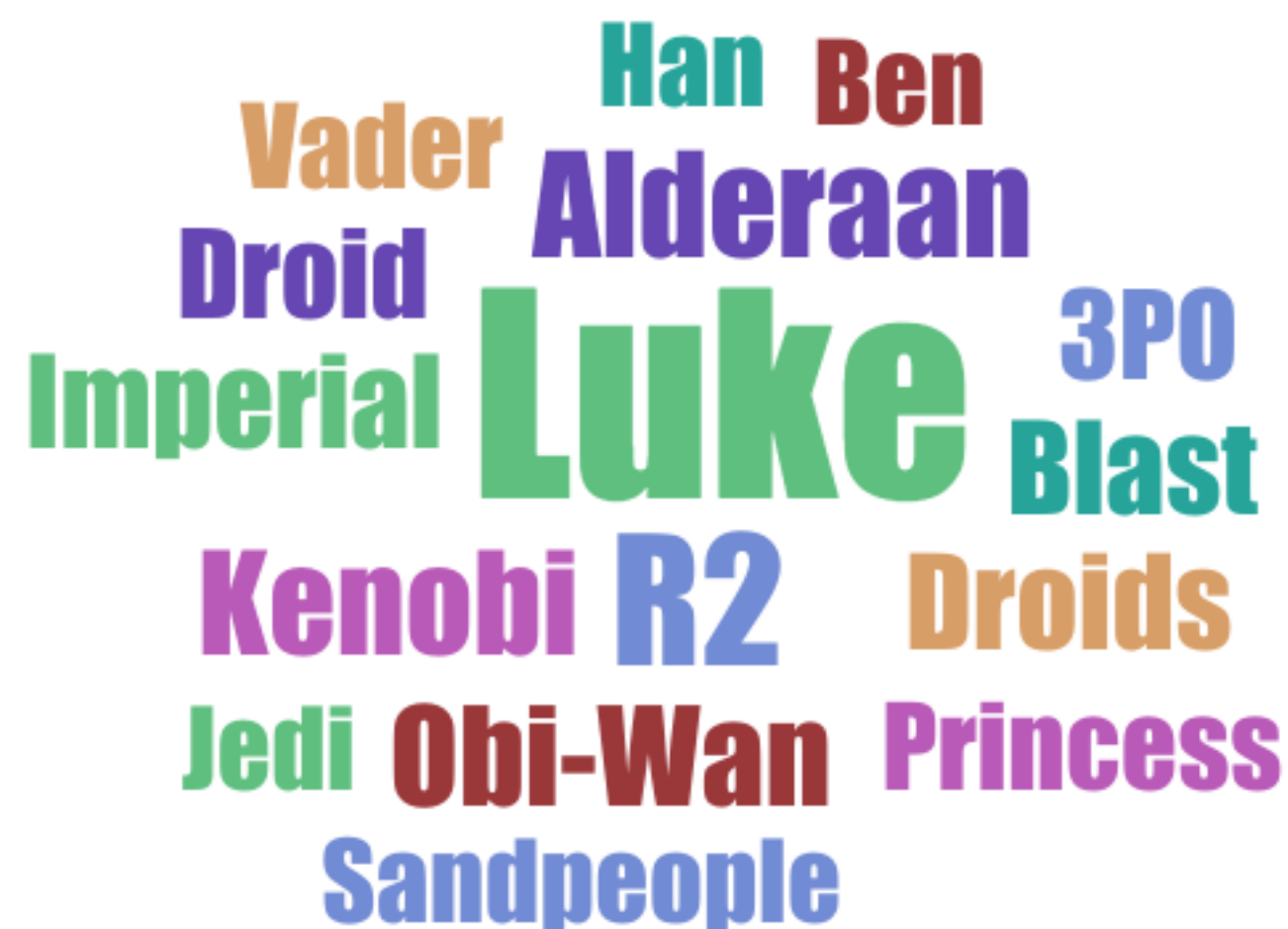
Size

Descending

25

☐ Group other values in separate bucket ☐ Show missing values 

Custom Label



```
GET starwars/_search
{
  "query": {
    "match": {
      "word": "Luke"
    }
  }
}
```



GET starwars/\_search

```
{
  "aggs": {
    "most_common": {
      "terms": {
        "field": "word.keyword",
        "size": 1
      }
    }
  },
  "size": 0
}
```









# Routing

`shard# = hash(_routing) % #primary_shards`

GET \_cat/shards?index=starwars&v

index	shard	prirep	state	docs	store	ip	node
starwars	3	p	STARTED	58	6.4kb	172.19.0.2	Q88C3v0
starwars	4	p	STARTED	26	5.2kb	172.19.0.2	Q88C3v0
starwars	2	p	STARTED	71	6.9kb	172.19.0.2	Q88C3v0
starwars	1	p	STARTED	63	6.6kb	172.19.0.2	Q88C3v0
starwars	0	p	STARTED	70	6.7kb	172.19.0.2	Q88C3v0

# (Sub) Results Per Shard

```
shard_size = (size * 1.5 + 10)
```

# How Many?

## Results per shard

## Results for aggregation

"doc\_count\_error\_upper\_bound": 10

"sum\_other\_doc\_count": 232



GET starwars/\_search

```
{
  "aggs": {
    "most_common": {
      "terms": {
        "field": "word.keyword",
        "size": 1,
        "show_term_doc_count_error": true
      }
    }
  },
  "size": 0
}
```



GET starwars/\_search

```
{
  "aggs": {
    "most_common": {
      "terms": {
        "field": "word.keyword",
        "size": 1,
        "shard_size": 20,
        "show_term_doc_count_error": true
      }
    }
  },
  "size": 0
}
```



# Cardinality Aggregation

# Naive Implementation: HashSet

```
HashSet noDuplicates = new HashSet();  
noDuplicates.add("Luke");  
noDuplicates.add("R2");  
noDuplicates.add("Luke");  
// ...  
noDuplicates.size();
```

# Simple Estimator: Even distribution 0 – 1

```
hash("Luke") -> 0.44  
hash("R2")    -> 0.71  
hash("Jedi")  -> 0.07  
hash("Luke")  -> 0.44
```

**Estimated cardinality:**  $\frac{1}{\text{minvalue}} = \frac{1}{0.07} = 14$

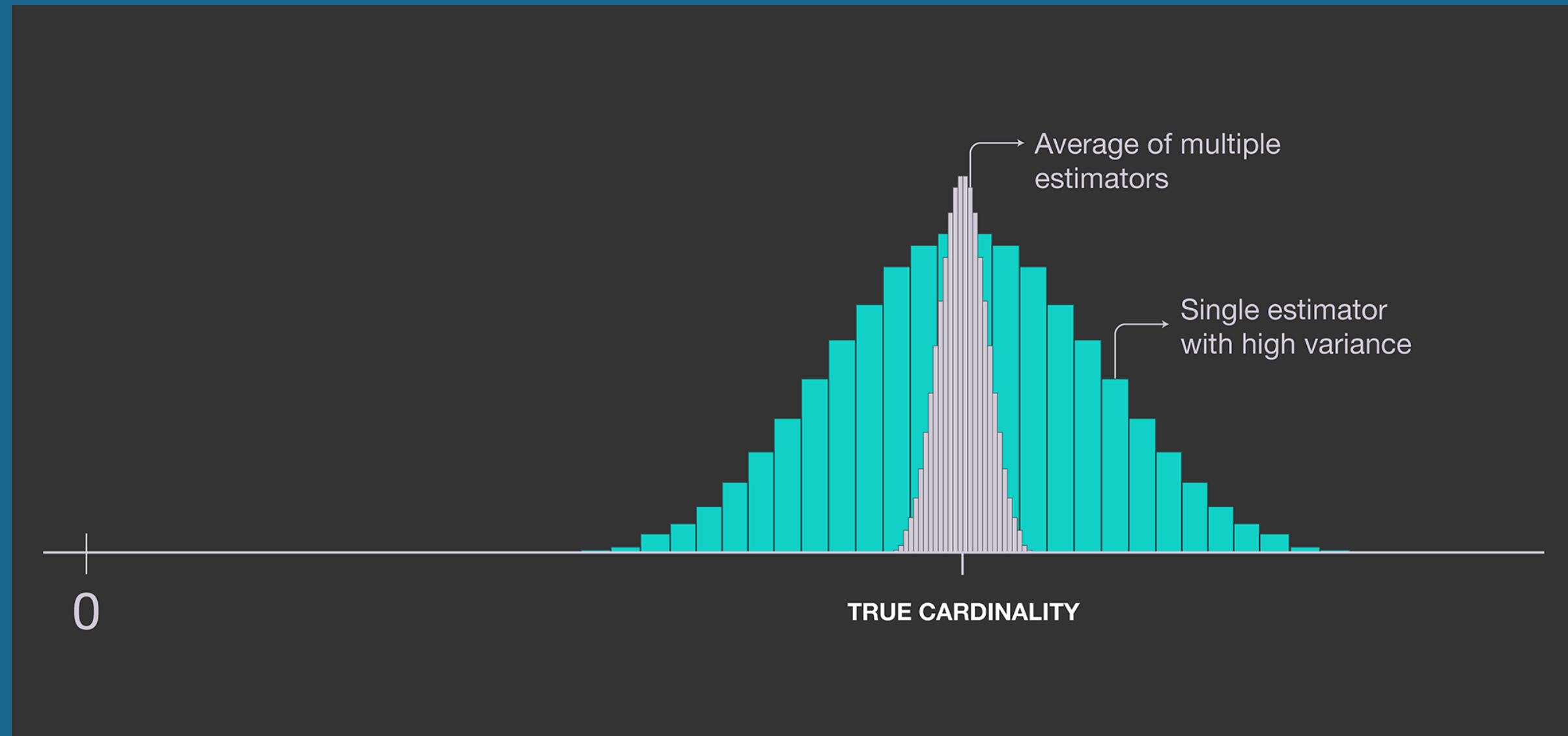
# Probabilistic Counting: Leading 0

hash(value) -> ... 0 0 0  
... 0 0 1  
... 0 1 0  
... 0 1 1  
... 1 0 0  
... 1 0 1  
... 1 1 0  
... 1 1 1

**Probability**  $\frac{1}{2^3} = \frac{1}{8}$  **or generally**  $\frac{1}{2^k}$



# LogLog: Probabilistic Averaging



# LOGLOG COUNTING OF LARGE CARDINALITIES

MARIANNE DURAND AND PHILIPPE FLAJOLET

ABSTRACT. Using an auxiliary memory smaller than the size of this abstract, the LOGLOG algorithm makes it possible to estimate in a single pass and within a few percents the number of different words in the whole of Shakespeare's works. In general the LOGLOG algorithm makes use of  $m$  "small bytes" of auxiliary memory in order to estimate in a single pass the number of distinct elements (the "*cardinality*") in a file, and it does so with an accuracy that is of the order of  $1/\sqrt{m}$ . The "small bytes" to be used in order to count cardinalities till  $N_{\max}$  comprise about  $\log \log N_{\max}$  bits, so that cardinalities well in the range of billions can be determined using one or two kilobytes of memory only. The basic version of the LOGLOG algorithm is validated by a complete analysis. An optimized version, super-LOGLOG, is also engineered and tested on real-life data. The algorithm parallelizes optimally.

## 1. INTRODUCTION

The problem addressed in this note is that of determining the number of *distinct* elements, also called the *cardinality*, of a large file. This problem arises in several areas of data-mining, database query optimization, and the analysis of traffic in

# LogLog: Bucketing for Averages

**4 bit bucket, rest for cardinality per bucket**

```
hash("Luke") -> 0100 101001000 -> [4]: 3  
hash("R2")   -> 1001 001010000 -> [9]: 4  
hash("Jedi") -> 0000 101110010 -> [0]: 1
```

Algorithm LOGLOG( $\mathfrak{M}$ : Multiset of hashed values;  $m \equiv 2^k$ )  
 Initialise  $M^{(1)}, \dots, M^{(m)}$  to 0;  
 let  $\rho(y)$  be the rank of first 1-bit from the left in  $y$ ;  
     for  $x = b_1 b_2 \dots \in \mathfrak{M}$  do  
         set  $j := \langle b_1 \dots b_k \rangle_2$  (value of first  $k$  bits in base 2)  
         set  $M^{(j)} := \max(M^{(j)}, \rho(b_{k+1} b_{k+2} \dots))$ ;  
 return  $E := \alpha_m m 2^{\frac{1}{m}} \sum_j M^{(j)}$  as cardinality estimate.

# HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm

Stefan Heule  
ETH Zurich and Google, Inc.  
stheule@ethz.ch

Marc Nunkesser  
Google, Inc.  
marcnunkesser  
@google.com

Alexander Hall  
Google, Inc.  
alexhall@google.com

## ABSTRACT

Cardinality estimation has a wide range of applications and is of particular importance in database systems. Various algorithms have been proposed in the past, and the HYPERLOGLOG algorithm is one of them. In this paper, we present a series of improvements to this algorithm that reduce its memory requirements and significantly increase its accuracy for an important range of cardinalities. We have implemented our proposed algorithm for a system at Google and evaluated it empirically, comparing it to the original HYPERLOGLOG algorithm. Like HYPERLOGLOG, our improved algorithm parallelizes perfectly and computes the cardinality estimate in a single pass.

estimate significantly for a range of important cardinalities. We evaluate all improvements empirically and compare with the HYPERLOGLOG algorithm from [7]. Our changes to the algorithm are generally applicable and not specific to our system. Like HYPERLOGLOG, our proposed improved algorithm parallelizes perfectly and computes the cardinality estimate in a single pass.

*Outline.* The remainder of this paper is organized as follows; we first justify our algorithm choice and summarize related work in Section 2. In Section 3 we give background information on our practical use cases at Google and list the requirements for a cardinality estimation algorithm in





GET starwars/\_search

```
{
  "aggs": {
    "type_count": {
      "cardinality": {
        "field": "word.keyword",
        "precision_threshold": 10
      }
    }
  },
  "size": 0
}
```





# precision\_threshold

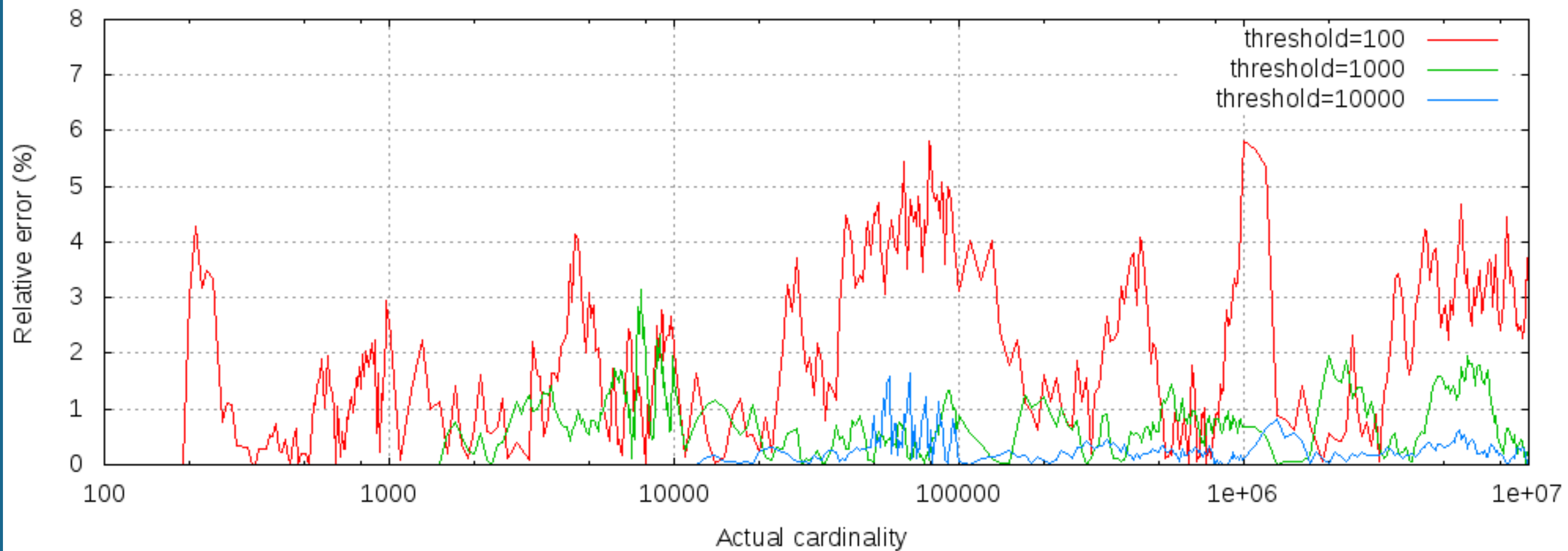
**Default 3,000**

**Maximum 40,000**

# Memory

`precision_threshold x 8 bytes`

Cardinality error



GET starwars/\_search

```
{
  "aggs": {
    "type_count": {
      "cardinality": {
        "field": "word.keyword",
        "precision_threshold": 12
      }
    }
  },
  "size": 0
}
```



# Precompute Hashes?

**Client or** `mapper-murmur3` **plugin**



# It Depends

👍 **large / high-cardinality fields**

👎 **low cardinality / numeric fields**

# Improvement: LogLog- $\beta$

[https://github.com/elastic/elasticsearch/  
pull/22323](https://github.com/elastic/elasticsearch/pull/22323)

# Improvement?

**"New cardinality estimation algorithms for  
HyperLogLog sketches"**

**<https://arxiv.org/abs/1702.01284>**

# Inverse Document Frequency

```
GET starwars/_search
{
  "query": {
    "match": {
      "word": "Luke"
    }
  }
}
```







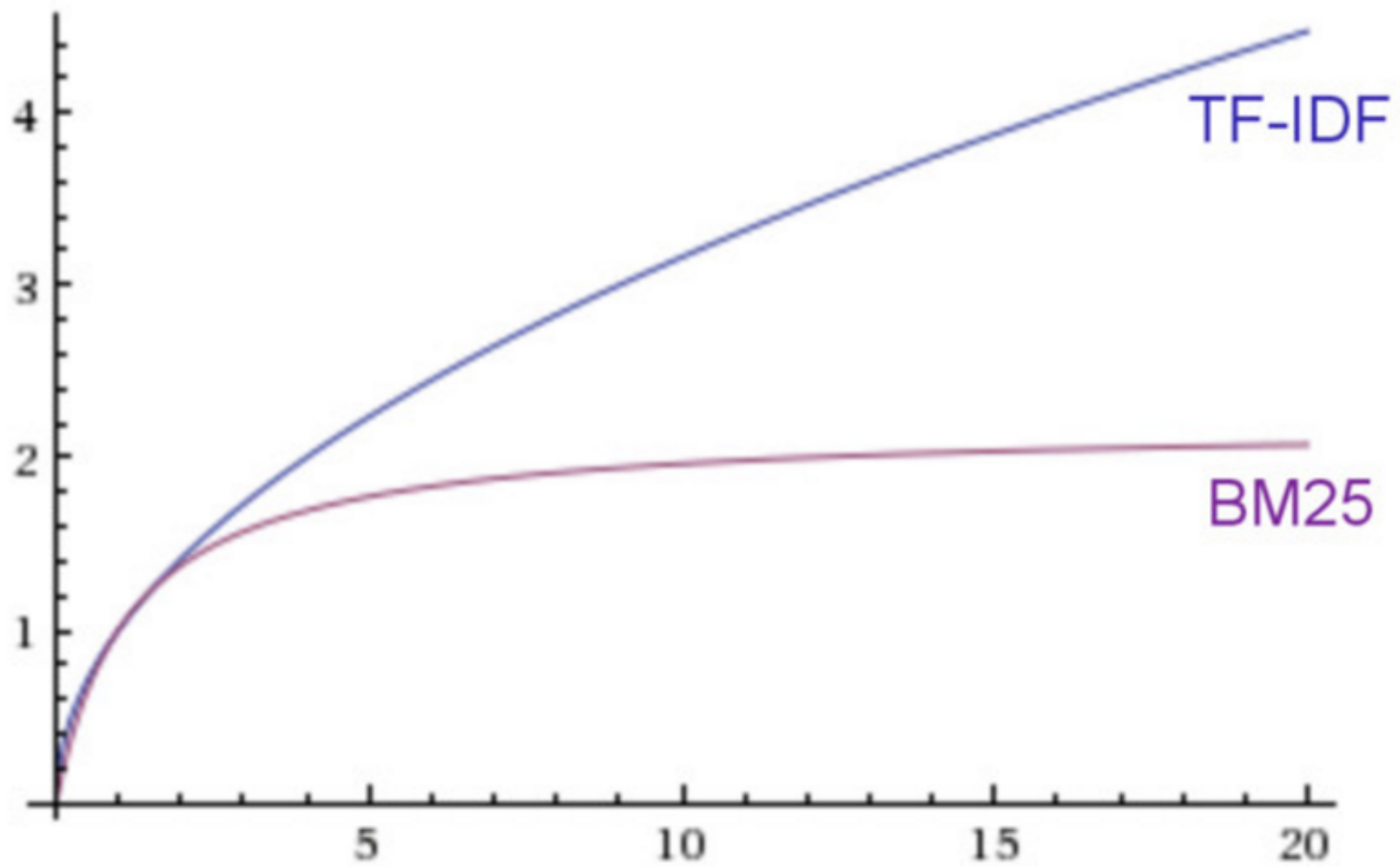
# Term Frequency / Inverse Document Frequency (TF/IDF)

# BM25

## Default in Elasticsearch 5.0

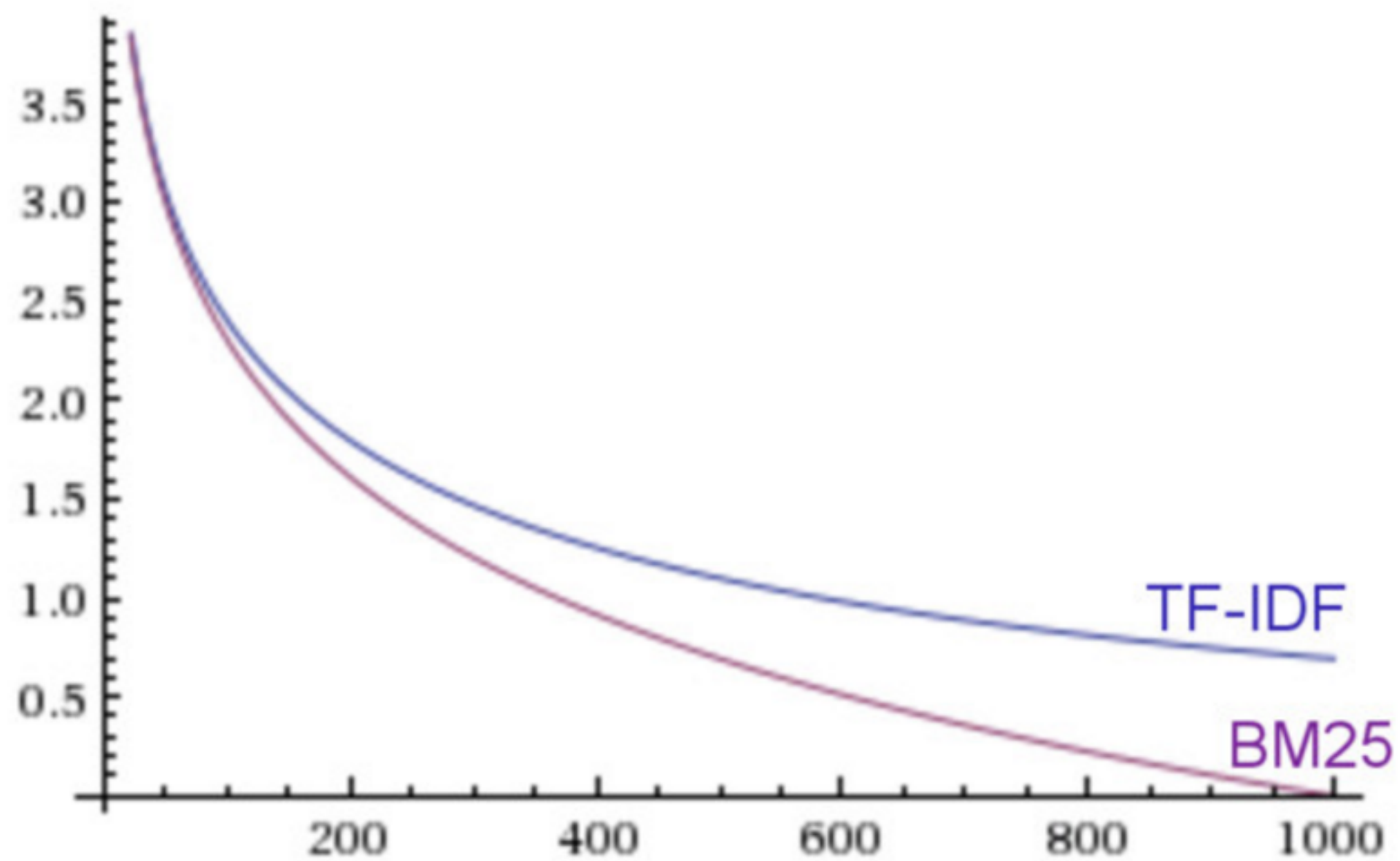
# Term Frequency

$$tf(t \text{ in } d) = \sqrt{\text{frequency}}$$



# Inverse Document Frequency

$$idf(t) = 1 + \log\left(\frac{numDocs}{docFreq + 1}\right)$$



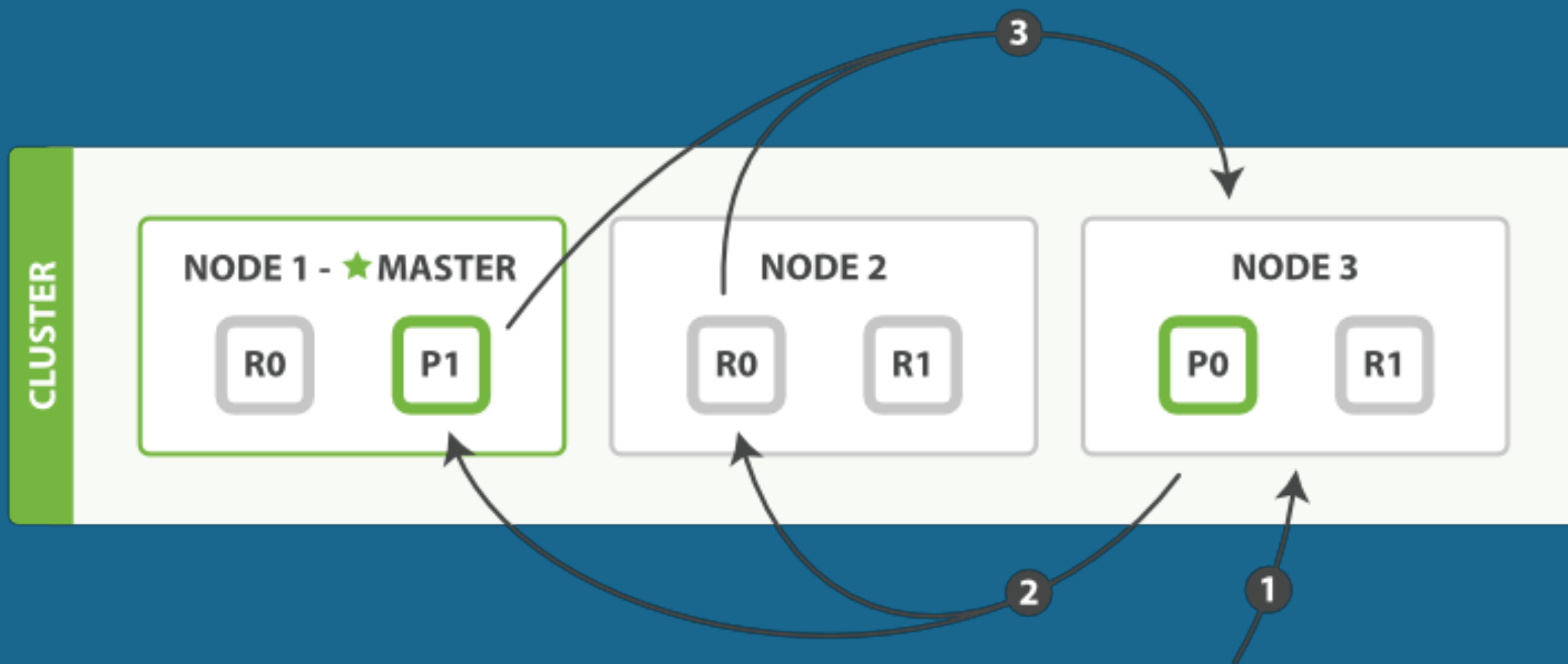
# Field-Length Norm

$$\textit{norm}(d) = \frac{1}{\sqrt{\textit{numTerms}}}$$

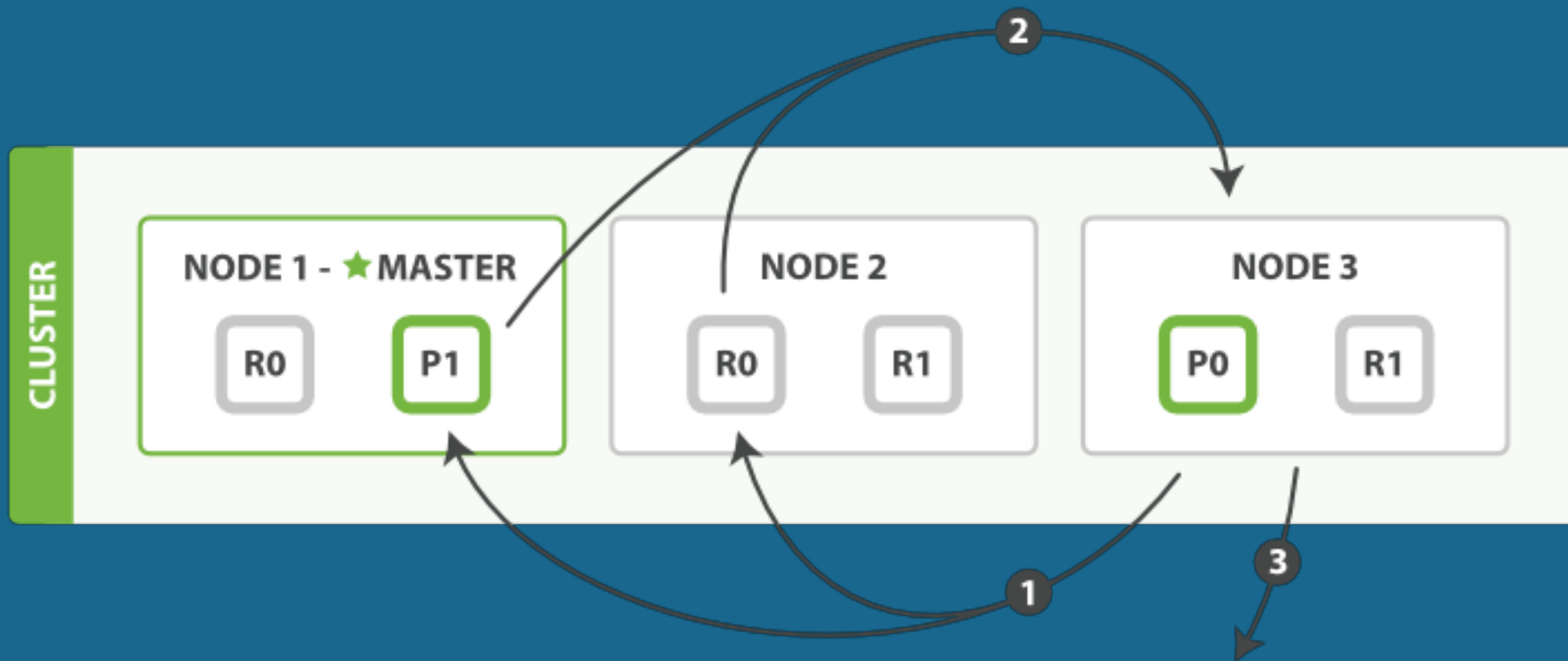


# Query Then Fetch

# Query



# Fetch



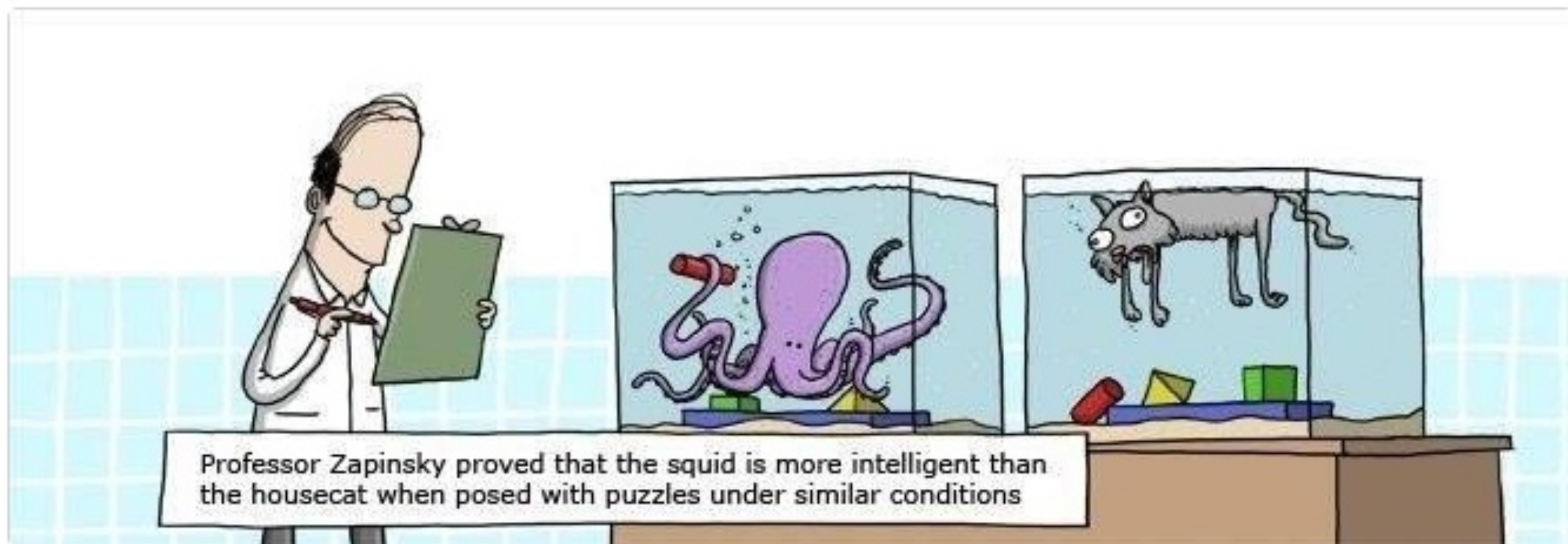
# DFS Query Then Fetch

**Distributed Frequency Search**

```
GET starwars/_search?search_type=dfs_query_then_fetch
{
  "query": {
    "match": {
      "word": "Luke"
    }
  }
}
```









***Don't use  
dfs\_query\_then\_fetch  
in production. It really  
isn't required.***

— <https://www.elastic.co/guide/en/elasticsearch/guide/current/relevance-is-broken.html>

# Single Shard

**Default in 7.0**

# Simon Says

**Use a single shard until  
it blows up**



```
PUT starwars/_settings
{
  "settings": {
    "index.blocks.write": true
  }
}
```

```
POST starwars/_shrink/starletwars?copy_settings=true
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0
  }
}
```

```
GET starletwars/_search
{
  "query": {
    "match": {
      "word": "Luke"
    }
  },
  "_source": false
}
```



```
GET starletwars/_search
{
  "aggs": {
    "most_common": {
      "terms": {
        "field": "word.keyword",
        "size": 1
      }
    }
  },
  "size": 0
}
```





# Change for the Cardinality Count?



# Conclusion

# Tradeoffs...

**Consistent**   **Available**  
**Partition Tolerant**  
**Fast**   **Accurate**   **Big**

# Questions?

**Philipp Krenn**

**@xeraa**

**PS: Stickers**