Building a distributed search engine with Tantivy

tantivy

Harrison Burt (aka @ChillFish8)

- Living in the United Kingdom
- Software developer @Quickwit building a super awesome distributed search engine
- Creator of lnx (more on the next slide)
- harrison@quickwit.io

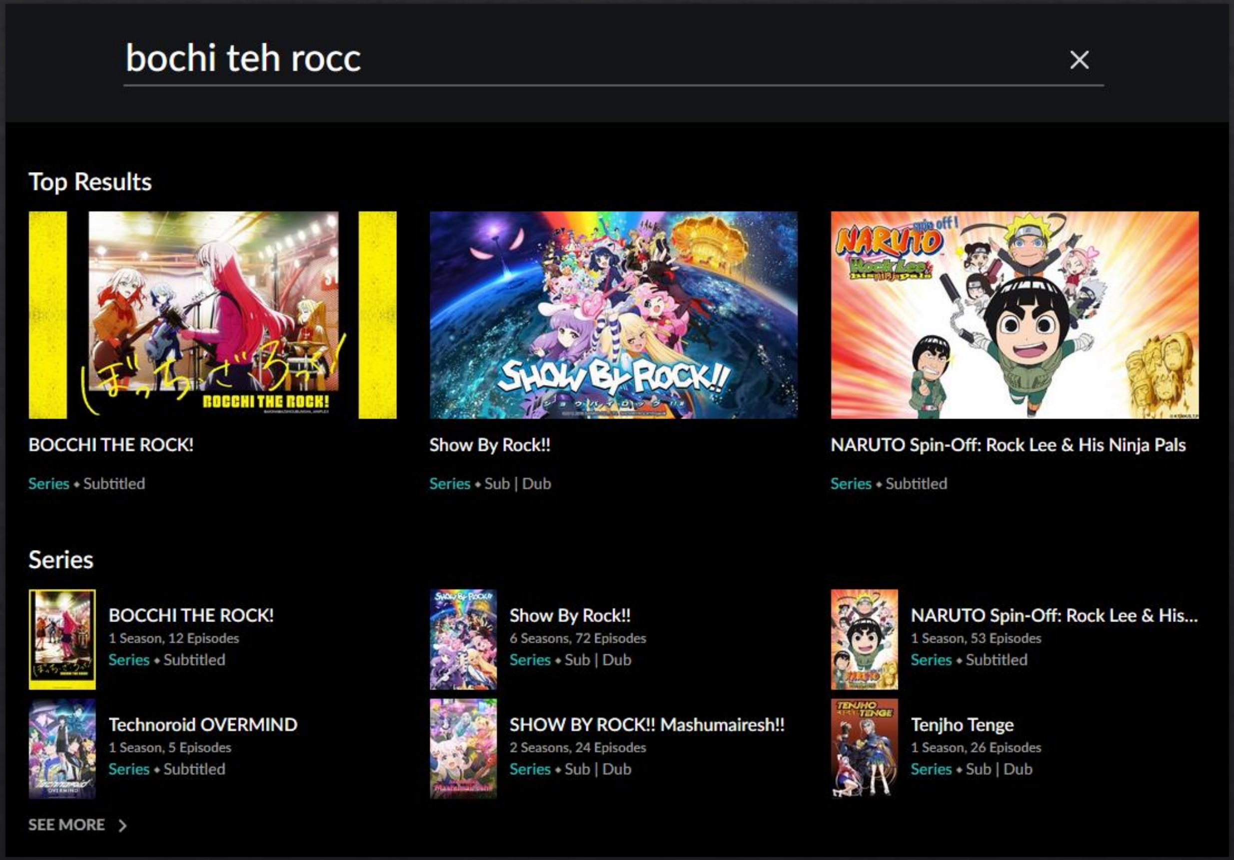
What is lnx?

- Search engine build on top of Tantivy, akin to Elasticsearch or Algolia
- Aimed towards user-facing search
- Typo-tolerance support
- Easily configurable
- Fast out of the box, with additional tuning parameters available.
- Indexing throughput @ 30-60 MB/s
- High-availability soon™

What is user facing search?

- Relevancy is a priority, even if a search contains a typo or two
- The latency of search matters in milliseconds
- Documents are often mutable
- Search as you type means a high throughput of searches

algolia



Crunchyroll's search which retrieves results as you type.

"Tantivy is a full-text search engine library inspired by Apache Lucene and written in Rust."

What is it?

- A full-text search engine
- BM25 Scoring (the same as Lucene)
- Incremental indexing
- Faceted search
- Range queries
- JSON Fields
- Aggregations
- Cheesy logo with a horse 🎉
- And more which doesn’t fit on this slide!

QUICKWIT

lnx

A basic implementation

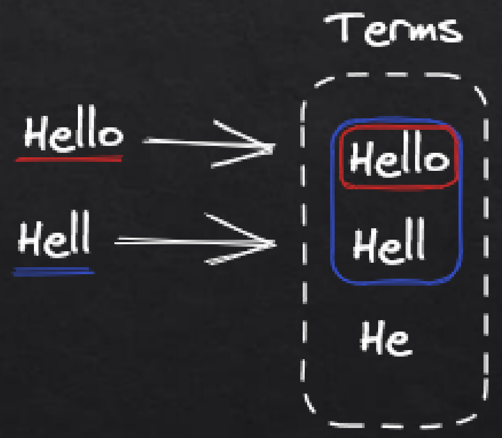
- We create a schema to define our fields and the properties they have
- We create an index using the schema we just made and store data a temporary directory
- We can add docs to the index by creating an indexer with a memory pool of a given size in bytes
- Calling commit will make our doc visible to the readers.
- Searchers allow us to execute queries and get the results collected by a collector(s)

```rust
use tantivy::{Document, Index};
use tantivy::collector::{Count, TopDocs};
use tantivy::query::QueryParser;
use tantivy::schema::{Schema, STORED, TEXT};

fn main() -> tantivy::Result<()> {
  // Define out schema
  let mut schema_builder = Schema::builder();
  let title_field = schema_builder.add_text_field("title", TEXT | STORED);
  let schema = schema_builder.build();

  // Indexing documents
  let index = Index::create_from_tempdir(schema.clone())?;
  let mut index_writer = index.writer(50_000_000)?;
  let mut my_document =  Document::default();
  my_document.add_text(title_field, "The Old Man and the Sea");
  index_writer.add_document(my_document)?;

  // Commit the changes
  index_writer.commit()?;

  // Searching the index
  let reader = index.reader()?;
  let searcher = reader.searcher();
  let parser = QueryParser::for_index(&index, vec![title_field]);
  let my_query = parser.parse_query("\"Old Man\"").unwrap();
  let my_collector = (Count, TopDocs::with_limit(10));
  let (count, results) = searcher.search(&my_query, &my_collector)?;
  println!("{count} Documents matched the query!");
  for (score, address) in results {
    let doc = searcher.doc(address)?;
    println!("Got doc with score {score}: {doc:?}");
  }

  Ok(())
}
```

Adding typo-tolerance

- Full-text search is great, but it’s not the best for user experience when searching as it doesn’t account for typos
- Tantivy provides us with this in the form of the `*FuzzyTermQuery*`
- Uses Levenshtein distance to work out what terms to match within a given edit distance
- Tantivy by default uses a FST (Finite state transducer) which allows for very fast Levenshtein distance matching on the index terms
- This comes at a cost in the form of more CPU time increasing query latency

```
// Match words within the edit distance or start with the term
FuzzyTermQuery::new_prefix(
    // The query term to compare against
    term,
    // The maximum edit distance that can be used to match a term
    max_edit_distance,
    // Should transposing a word (swapping) count as a distance of 1 or 2?
    transposition_cost_one,
);
```



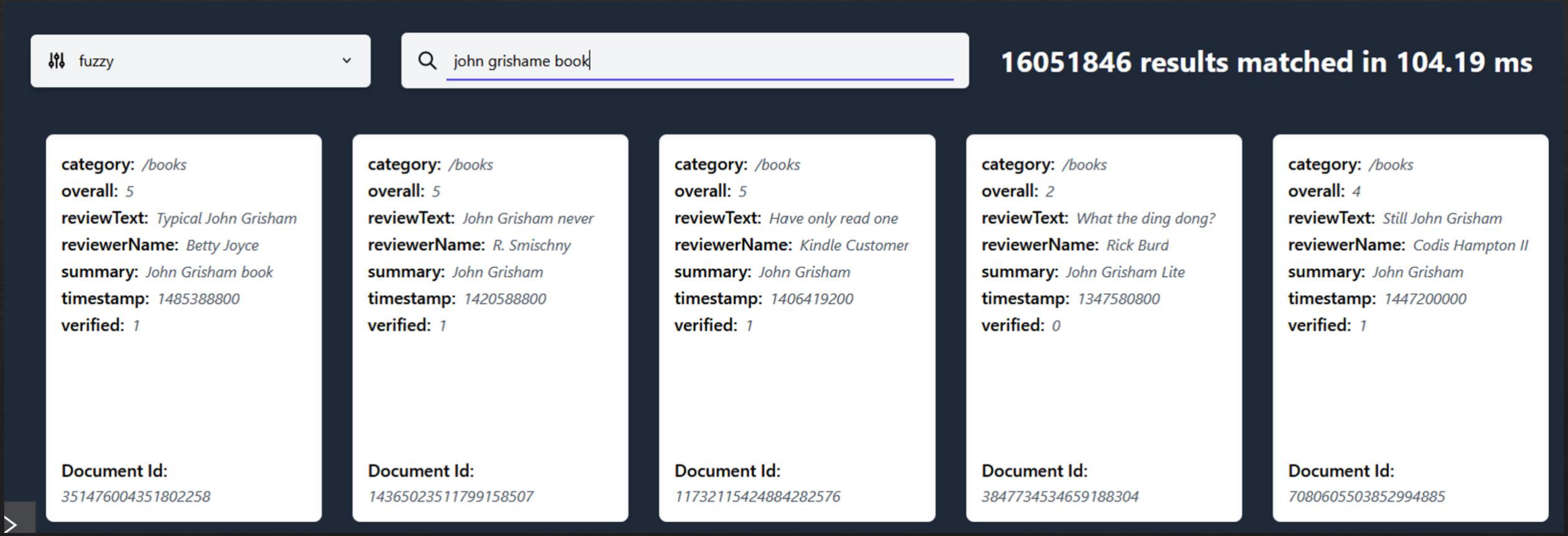
What we're left with on disk

- Tantivy serializes our index into various files making up a segment
- We also have some metadata files like `*meta.json*` and `*.managed.json*`

Name	Date modified	Type	Size
meta.json	09/01/2023 10:47	JSON File	3 KB
6a3ed54f605e4c1aa54087e6103e1379.fieldnorm	09/01/2023 10:47	FIELDNORM File	1 KB
2cb6d9f1553548469026e9a44a517b31.term	09/01/2023 10:47	TERM File	12,885 KB
2cb6d9f1553548469026e9a44a517b31.store	09/01/2023 10:47	STORE File	3,882 KB
2cb6d9f1553548469026e9a44a517b31.pos	09/01/2023 10:47	POS File	2,205 KB
2cb6d9f1553548469026e9a44a517b31.idx	09/01/2023 10:47	IDX File	8,267 KB
2cb6d9f1553548469026e9a44a517b31.fast	09/01/2023 10:47	FAST File	1 KB
.tantivy-writer.lock	09/01/2023 10:46	LOCK File	0 KB
.tantivy-meta.lock	09/01/2023 10:47	LOCK File	0 KB
.managed.json	09/01/2023 10:47	JSON File	5 KB

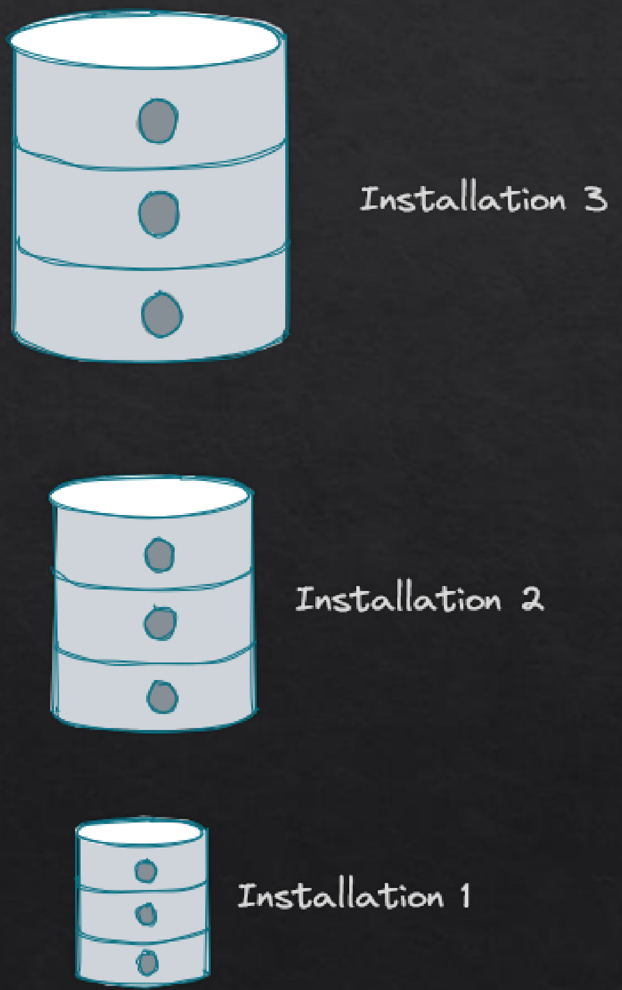
Now to wrap it in an API and ship it

This is how lnx works under the hood as you can see here:



Some issues but nothing major

- As search traffic increases, in order to scale we need to use bigger and bigger machines
- Modern cloud doesn't make this the end of the world

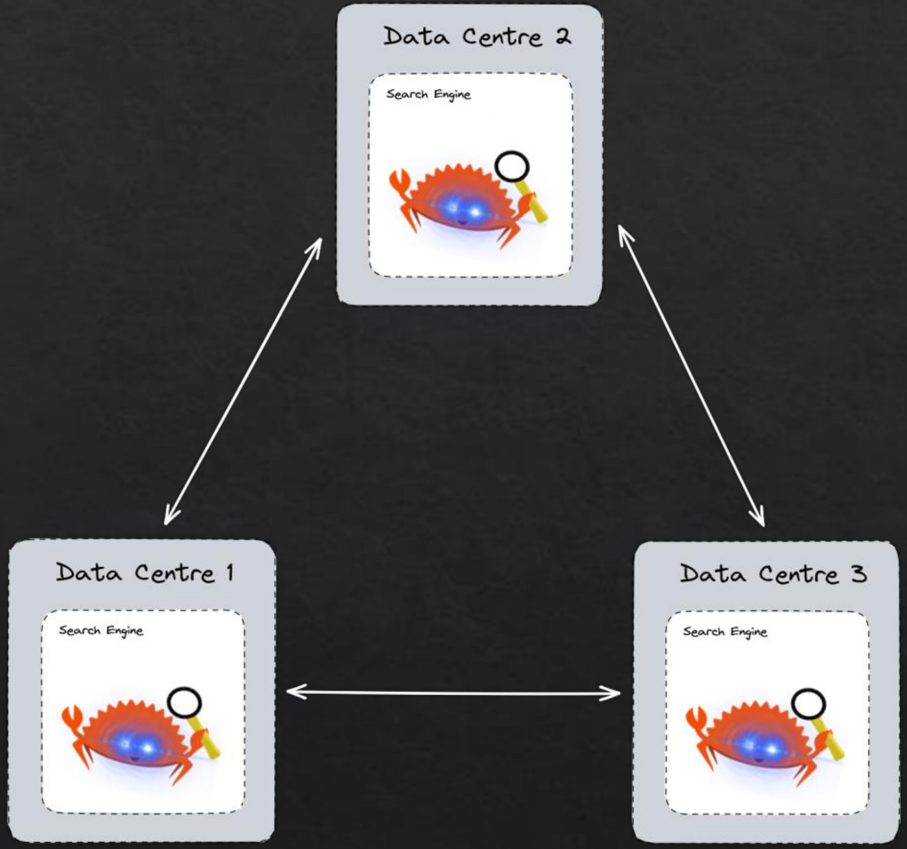


Disaster

- Server is on fire
- Site unable to return search queries
- Loosing money
- Angry management waiting for you to fix this mess

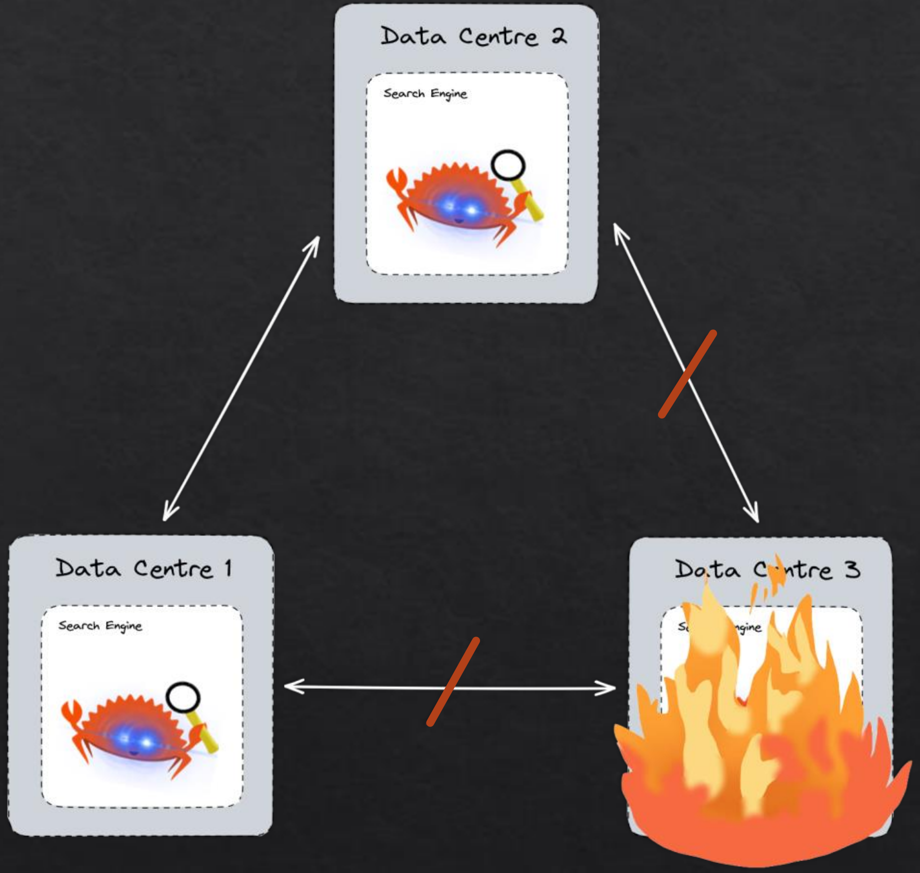
With replicas we can tolerate failure

- We deploy a cluster of 3 nodes, all replicating the same state
- Each node is in a different data centre / availability zone



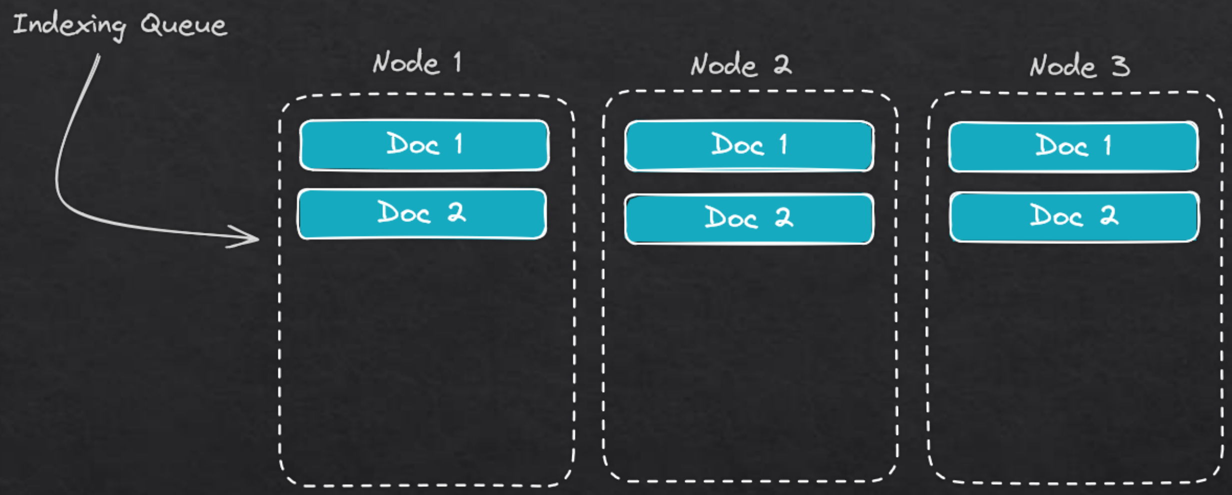
With replicas we can tolerate failure

- Data centre 3 fails, the other two replicas can continue to serve incoming operations.
- This also lets you do seamless upgrades / restarts of the system (sometimes)



Replicating our data across nodes

- We replicate documents not the index itself
- The processing of each document is applied by each node
- This makes our lives a bit simpler but comes at the cost of wasting our resources



How hard can it be…

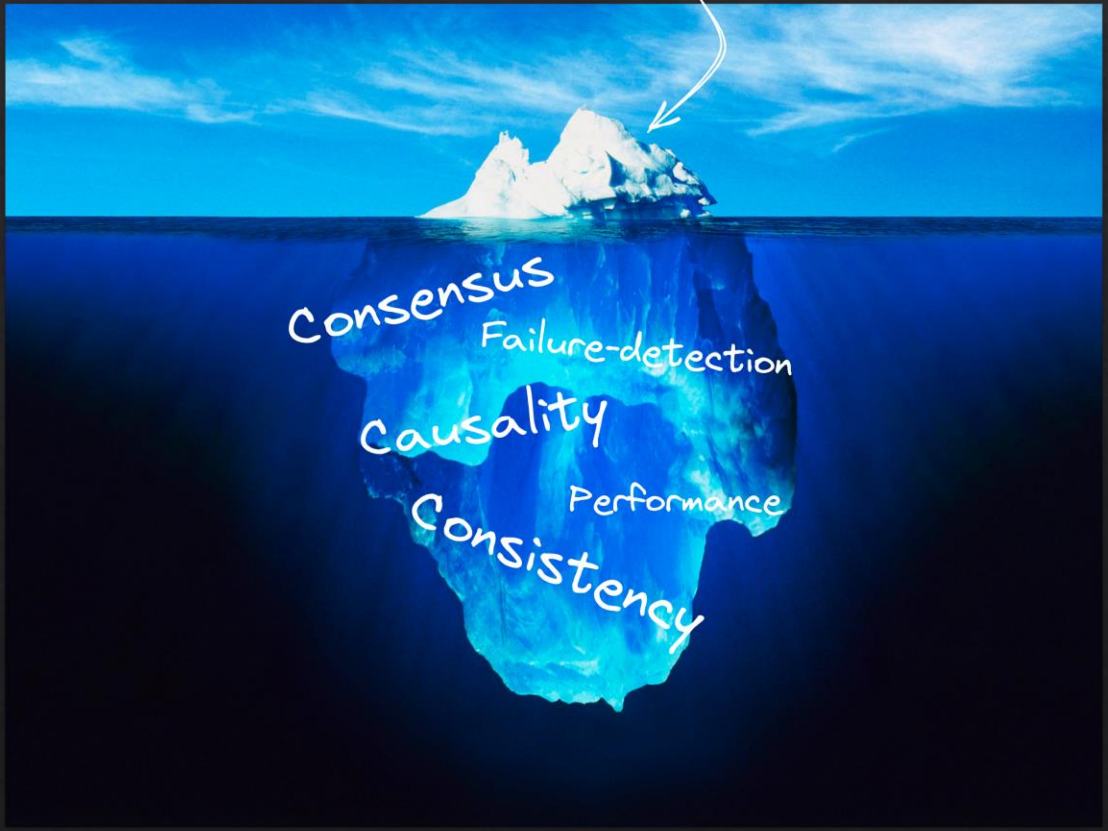
"Paxos???"

"Why is time such a pain?"

"What the hell is a Raft?"

"What do you mean networks aren't reliable?"

"Just spin up more nodes with RPC"



The wider world is scary…

- We need some way on converging state
- CAP theorem becomes a thing (Consistency, Availability, Partition-tolerance)
- We must handle networks failing

Evaluating our options

The Raft way:

- Leader-base system
- Produces a replicated log of operations
- Pre-made implementations of Raft in Rust
- Very strict set of rules in order to be correct

The eventually consistent way:

- A leader-less system
- Operations are idempotent
- Gives us more freedom to change our replication behaviour should we wish

Evaluating our options

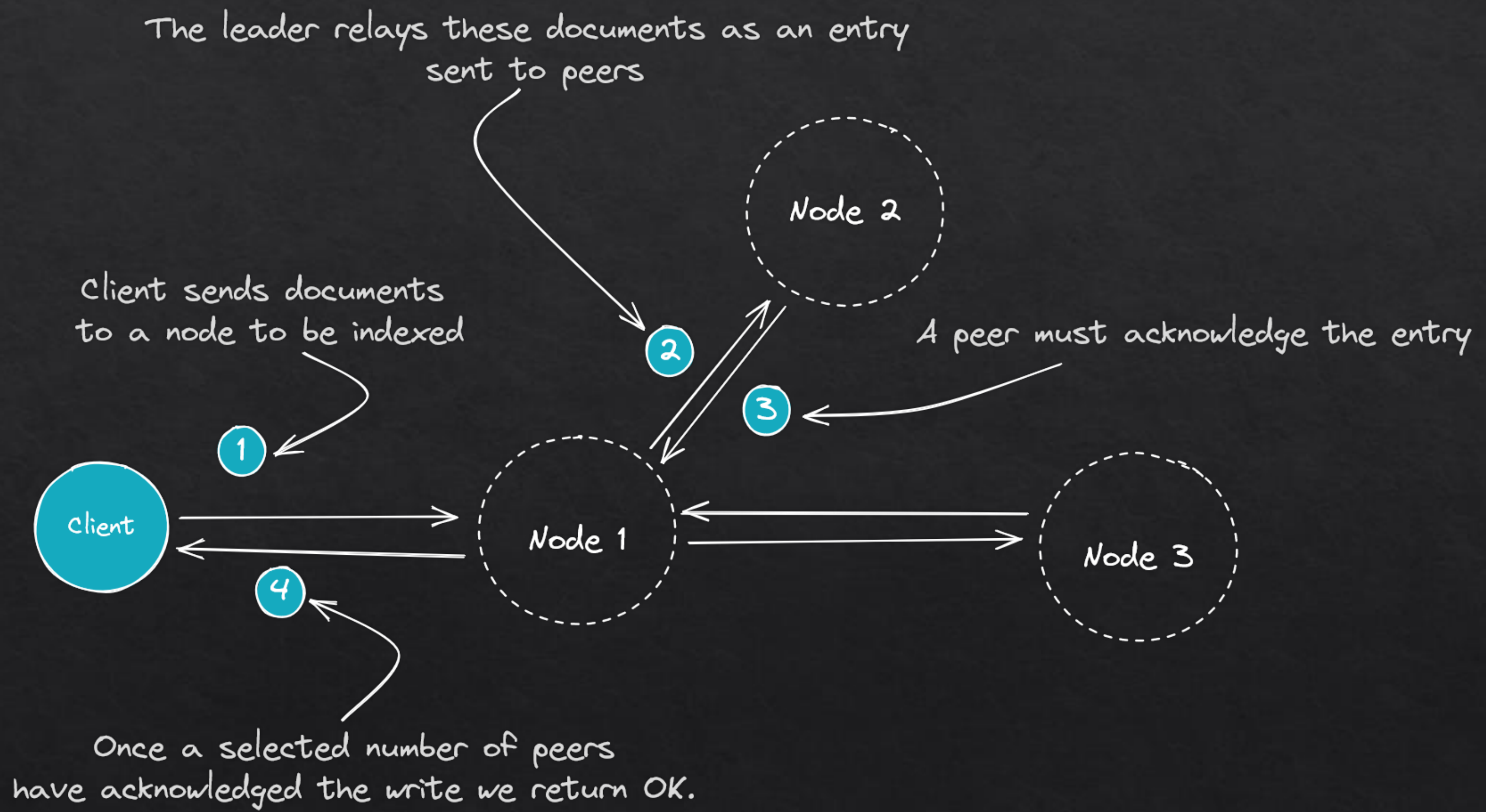
The Raft way:

- Leader-base system
- Produces a replicated log of operations
- Pre-made implementations of Raft in Rust
- Very strict set of rules in order to be correct

The eventually consistent way: ✓

- A leader-less system
- Operations are idempotent
- Gives us more freedom to change our replication behaviour should we wish

Using eventual consistency



Some issues

- How do we implement this?
- How do we make this easier to test?

Luckily the work has been done for us

Datacake provides all of the tooling for creating distributed systems:

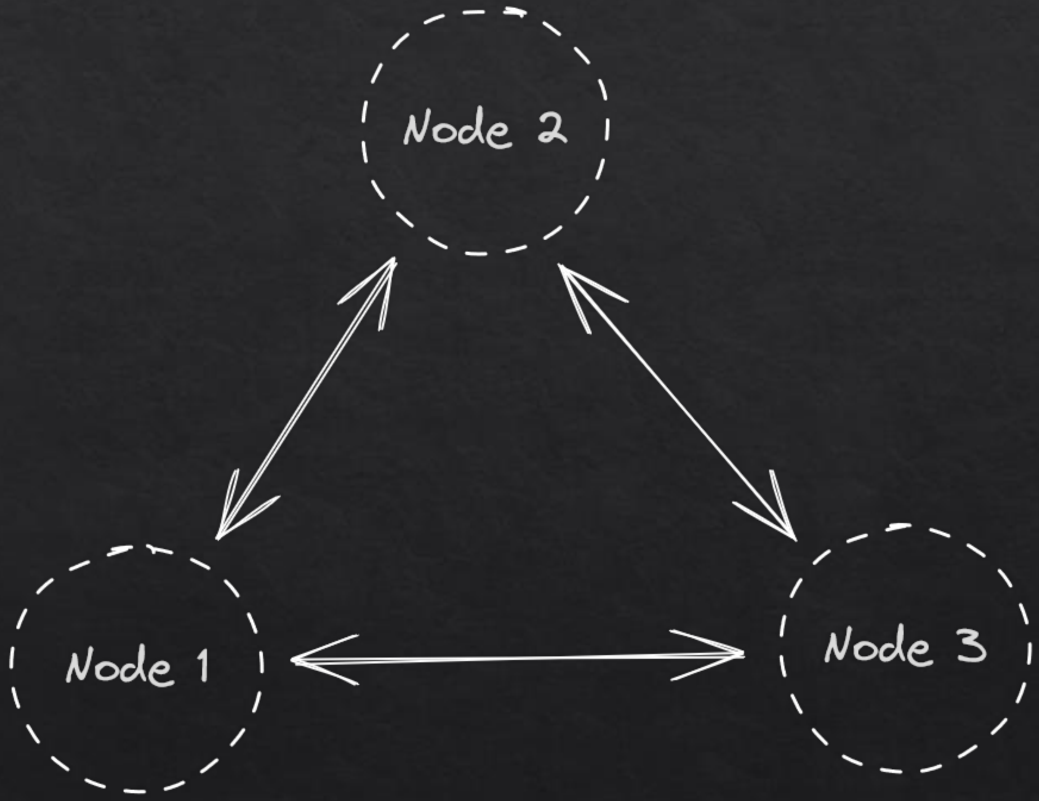
- Zero-copy RPC framework with simulation support
- Membership and failure detection wrapping `*chitchat*`
- Pre-built eventually consistent store for small-ish key-values
- Pre-built storage implementations
- CRDT implementations and hybrid logical clocks

https://github.com/lnx-search/datacake

```rust
let addr = "127.0.0.1:8080".parse::<SocketAddr>()?;
let connection_cfg = ConnectionConfig::new(
    addr,
    addr,
    Vec::<String>::new(),
);

let node = DatacakeNodeBuilder::<DCAwareSelector>::new(1, connection_cfg)
    .connect()
    .await?;

let store = node
    .add_extension(EventuallyConsistentStoreExtension::new(MemStore::default()))
    .await?;

let handle = store.handle();

handle
    .put(
        "my-keyspace",
        1,
        b"Hello, world! From keyspace 1.".to_vec(),
        Consistency::All,
    )
    .await?;
```

Creating the cluster

```
let node_1 = DatacakeNodeBuilder::<DCAwareSelector>::new(1, connection_cfg_1)
    .connect()
    .await?;

let node_2 = DatacakeNodeBuilder::<DCAwareSelector>::new(2, connection_cfg_2)
    .connect()
    .await?

let node_3 = DatacakeNodeBuilder::<DCAwareSelector>::new(3, connection_cfg_3)
    .connect()
    .await?;

node_1
    .wait_for_nodes([2, 3], Duration::from_secs(30))
    .await?;
node_2
    .wait_for_nodes([1, 3], Duration::from_secs(30))
    .await?;
node_3
    .wait_for_nodes([2, 1], Duration::from_secs(30))
    .await?;
```



Extensions

- Add new functionality to the already running cluster
- Can be dynamically added or removed
- Have access to all of the utility methods the cluster provides (Cluster Clock, RPC network, etc..)
- They can be a simple or as complex as needed

```rust
use datacake_node::{ClusterExtension, DatacakeNode};
use async_trait::async_trait;

pub struct MyExtension;

#[async_trait]
impl ClusterExtension for MyExtension {
    type Output = ();
    type Error = MyError;

    async fn init_extension(
        self,
        node: &DatacakeNode,
    ) -> Result<Self::Output, Self::Error> {
        // In here we can setup our system using the live node.
        // This gives us things like the cluster clock and RPC server:

        println!("Creating my extension!");

        let timestamp = node.clock().get_time().await;
        println!("My timestamp: {timestamp}");

        Ok(())
    }
}

pub struct MyError;
```

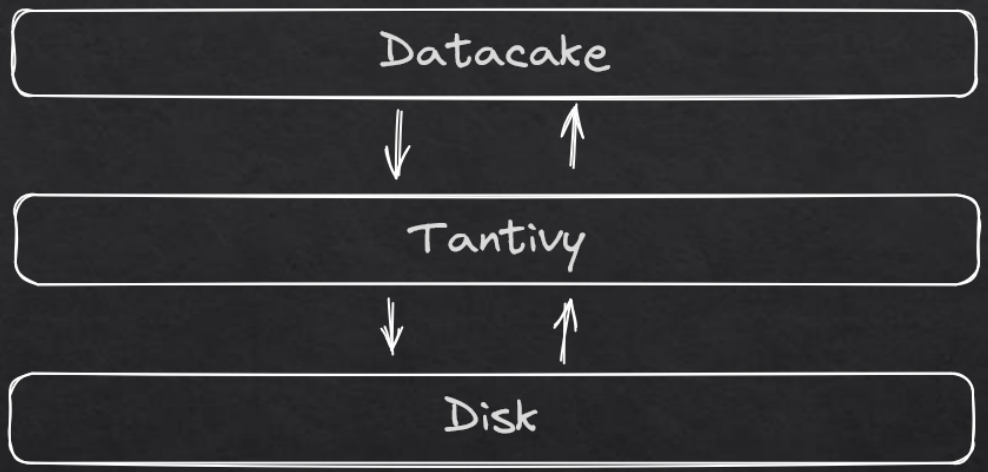
The eventually consistent store

- A pre-made extension adding an eventually consistency key-value store to the cluster
- Adjustable consistency levels
- Concept of keyspaces for organising documents
- Single storage trait for applying operations to a persistent store
- Not suitable for billion key scale databases

```
let store = node
    .add_extension(EventuallyConsistentStoreExtension::new(MemStore::default()))
    .await?;

let handle = store.handle();

handle
    .put(
        "my-keyspace",
        1,
        b"Hello, world! From my-keyspace.".to_vec(),
        Consistency::All,
    )
    .await?;
```

Combining it with tantivy

- We can combine the eventual consistency storage trait with tantivy acting as our persistent store
- Fetching, deleting and indexing documents as part of our operation using tantivy's in built doc store
- Simple demo available https://github.com/ChillFish8/tantivy-demo




```
INFO tantivy::indexer::index_writer: Prepared commit 19557
INFO tantivy::indexer::prepared_commit: committing 19557
INFO tantivy::indexer::segment_updater: save metas
INFO tantivy::indexer::segment_updater: Running garbage collection
INFO tantivy::directory::managed_directory: Garbage collect
INFO tantivy_demo: Indexing complete! elapsed=422.7787ms num_doc=19547
```

The end!

Questions!

- Harrison Burt (harrison@quickwit.io)
- Lnx
  https://github.com/lnx-search/lnx
- Quickwit
  https://quickwit.io/
- Datacake
  https://crates.io/crates/datacake
- Replicated Tantivy Demo
  https://github.com/ChillFish8/tantivy-demo