Solution

We can build on these ideas and group the text in clusters. No one will search for a whole paragraph; they will search for a few words at a time.

```
const NUMBER_OF_WORDS = 4;

const createIndex = (text: string) => {

    // regex matches any alphanumeric from any language and strips
spaces
    const finalArray: string[] = [];

    const wordArray = text
        .toLowerCase()
        .replace(/[^\p{L}\p{N}]+/gu, ' ')
        .replace(/ +/g, ' ')
        .trim()
        .split(' ');

    do {
        finalArray.push(
            wordArray.slice(0, NUMBER_OF_WORDS).join(' ')
        );
        wordArray.shift();

    } while (wordArray.length !== 0);

    return finalArray;
};
```

This will strip out all non-alphanumeric characters and make an array based on the words in groups of 4. For example, this text:

> Baseball is a game of strategy and skill, where every pitch, swing, and catch can change the course of the game in an instant.

We would have something like this:

```
["baseball is a game",
["is a game of",
["a game of strategy",
["game of strategy and",
["of strategy and skill",
...
```

This does not help us, as we need to be able to search as we type the word. So, again, we must go through every iteration of that iteration.

```
for (const phrase of index) {
    if (phrase) {
        let v = '';
        for (let i = 0; i < phrase.length; i++) {
            v = phrase.slice(0, i + 1).trim();
            // increment for relevance
            m[v] = m[v] ? m[v] + 1 : 1;
        }
    }
}
```

And we end up with something like this:

```
a game of strate: 1
a game of strateg: 1
a game of strategy: 1
an: 4
an i: 1
an in: 1
an ins: 1
an inst: 1
an insta: 1
an instan: 1
an instant: 1
and: 4
and c: 1
and ca: 1
and cat: 1
and cato: 1
```

Relevance

Instead of an array, we store it as a map. Notice the number by the word. Luckily, we don't have to store repeated phrases like "and." We can increment the value for each repetition of the words. This gives us relevance.

Searching

To search, we find where the search field is equal to `**searchField.term**`. Since we are ordering by this also, we don't need a `**where**` clause.

```
const data = await getDocs(
  query(
    collection(db, 'posts'),
    orderby(`searchField.${term}`),
    limit(5)
  )
);
```

Fuzzy Search

What we really want is not an exact search but a fuzzy search. We need some kind of typo tolerance.

Typo Tolerance with Soundex

The soundex algorithm has been used for years to simulate typo tolerance. It allows you to store text as sound patterns, not just as text.

```
// Take any string, and return the soundex
export function soundex(s: string): string {
    const a = s.toLowerCase().split("");
    const f = a.shift() as string;
    let r = "";
    const codes: Record<string, number | string> = {
        a: "",
        e: "",
        i: "",
        o: "",
        u: "",
        b: 1,
        f: 1,
        p: 1,
        v: 1,
        c: 2,
        g: 2,
        j: 2,
        k: 2,
        q: 2,
        s: 2,
        x: 2,
        z: 2,
        d: 3,
        t: 3,
        l: 4,
        m: 5,
        n: 5,
        r: 6,
    };
    r = f + a
        .map((v: string) => codes[v])
        .filter((v, i: number, b) =>
            i === 0 ? v != codes[f] : v != b[i - 1])
        .join("");
    return (r + "000").slice(0, 4).toUpperCase();
}
```

📝 There are different versions for different languages. Here is one for French for example.

Using this pattern, we can slightly modify our storage mechanism to translate for soundex first.

```
const temp = [];
// translate to soundex
for (const i of index) {
  temp.push(i.split(' ').map(
    (v: string) => soundex(v)
  ).join(' '));
}
index = temp;
// add each iteration from the createIndex
for (const phrase of index) {
  if (phrase) {
    let v = '';
    const t = phrase.split(' ');
    while (t.length > 0) {
      const r = t.shift();
      v += v ? ' ' + r : r;
      // increment for relevance
      m[v] = m[v] ? m[v] + 1 : 1;
    }
  }
}
```

The beauty of this is it greatly simplifies your storage.

```
P625 B400: 1
P625 B400 T000: 1
P625 B400 T000 S100: 1
R430: 1
S100: 1
S100 R430: 1
T000: 3
T000 B652: 1
T000 B652 A130: 1
T000 B652 A130 T652: 1
T000 P623: 1
T000 P623 O100: 1
T000 P623 O100 H652: 1
T000 S100: 1
T000 S100 R430: 1
T652: 1
```

You're only storing the sound itself, not the letters, and there are fewer iterations of sounds than words.

Remember to translate the text to soundex before you search as well!

```
// get the soundex terms
const searchText = text
  .trim()
  .split(' ')
  .map(v => soundex(v))
  .join(' ');

// search
const data = getDocs(
  query(
    collection(db, 'posts'),
    orderby(`search.${searchText}`),
    limit(50)
  )
);
```