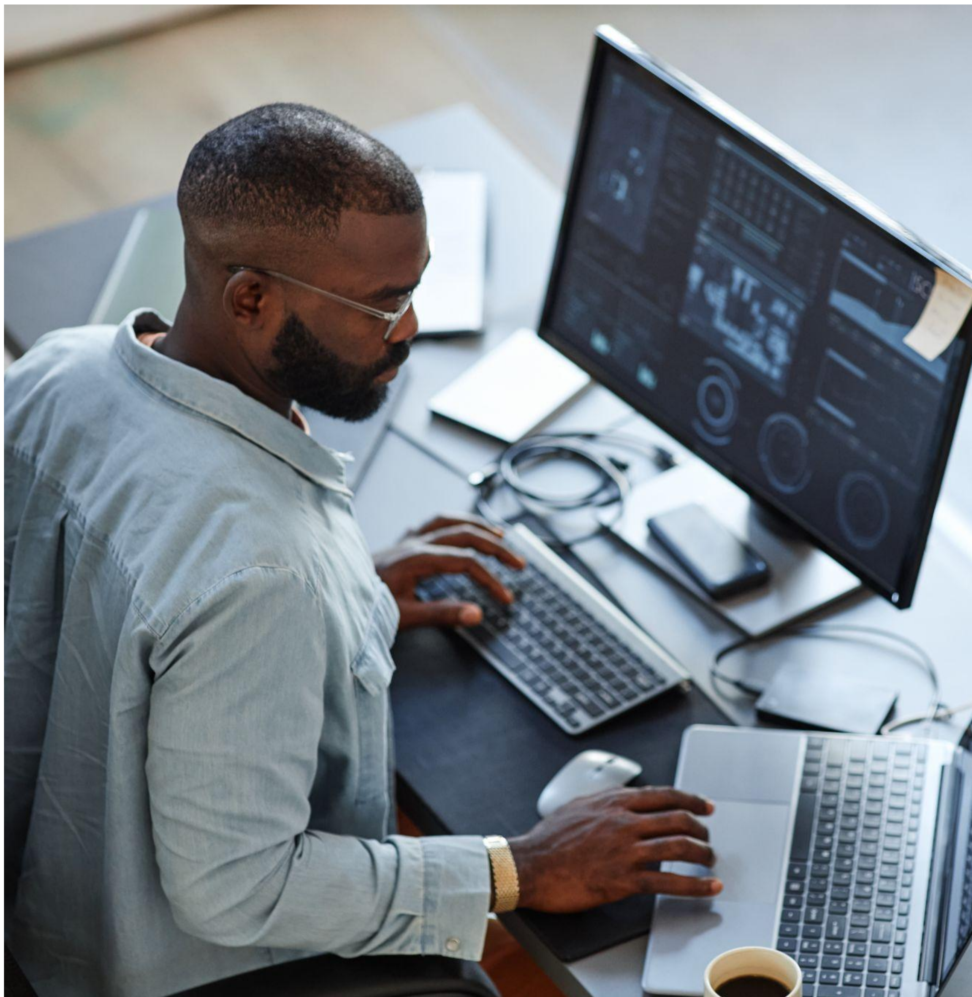
Google Cloud

Building applications with Spanner full-text search

Leverage Google's search expertise to power your data search architecture on Spanner.

August 1 2024



Authors

Alex Khesin
Principal Software Engineer, Spanner
Jeff Sosa
Senior Product Manager, Spanner
Karthi Thyagarajan
Senior Staff Solutions Architect, Google Cloud Databases

What is Spanner?

Spanner is Google's fully managed, globally distributed database service with virtually unlimited scale and industry-leading availability. As the database that powers Google's internal applications such as Gmail, YouTube, and Google Photos, Spanner is known for its reliability, scale, and strong consistency, making it the trusted choice for companies worldwide, including those in financial services, retail, gaming, technology, and media. With features such as automatic sharding, zero planned downtime, world class availability, seamless replication, and guaranteed strong consistency, Spanner handles the complexities of managing and operating a mission-critical database for both non-relational and relational workloads at scale.

Introduction to Spanner full-text search

Application users often need to perform searches on data that is not structured, such as the description of an item on an e-commerce site or to find relevant support documentation. Users also want to perform low latency text searches for items like names, titles of movies, addresses or transaction identifiers. The most common architectural approach for supporting searches of this type is to operate a separate search system optimized for full-text search along with pipelines for shipping data from the main OLTP database to this separate full-text search optimized solution. With Spanner full-text search (FTS), you can now easily search your online production data in the same database.

This whitepaper describes what is unique about Spanner FTS and how you can use Spanner FTS to build and operate applications with advanced search capabilities without having to copy and index data to and run dedicated search solutions. Spanner eliminates this complexity and delivers search via the proven Spanner platform that provides availability at virtually unlimited scale. This allows you to consolidate infrastructure and reduce complexity which ultimately lowers both operational and infrastructure costs for modern data architectures.

What is unique about Spanner full-text search?

Virtually unlimited scale

Users who are familiar with Spanner for their operational workloads know that Spanner offers virtually unlimited scale across both the storage and throughput dimensions. This remains true for full-text search workloads as well. Spanner automatically balances your data across shards as your storage usage increases. Besides providing scalability, this also makes managing your data infrastructure much easier in Spanner as opposed to traditional search systems. With proper design, your full-text search workloads can linearly scale out to handle virtually unlimited storage and/or throughput without compromising latency.

Ease of Use

Spanner FTS is built upon the search expertise developed by Google. This expertise has been leveraged to bring the ease of web search to Spanner database customers. FTS provides intelligent context-based search automatically when leveraging the enhance_query feature that incorporates machine learning-based processing to determine the intent of the query, and automatically incorporates knowledge of stop words, synonyms and spell correction. FTS also provides traditional fuzzy search capability through NGRAMS-based search that allows you to find matches based upon individual character variances in the text, as well as the ability to match similar-sounding names using the SOUNDEX feature. Tokenization and search can also be done in multiple languages with automatic language recognition.

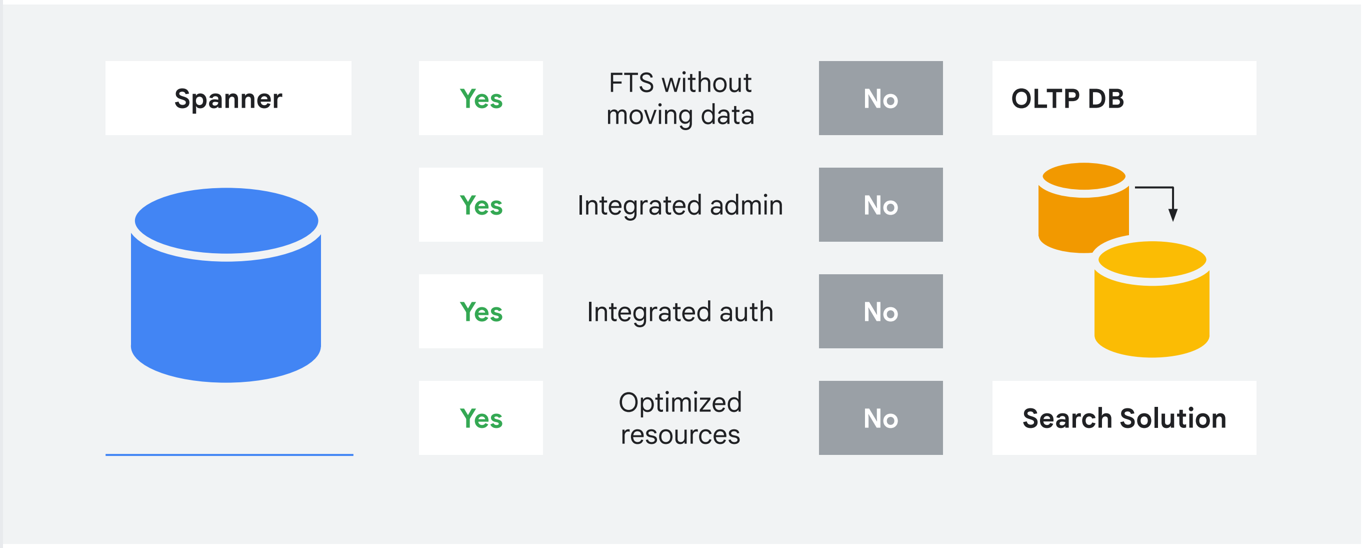
FTS can also process search criteria entered with logical operators similar to Google web search. An example is when a user types in “red OR green” as the search input, FTS will automatically search for “red” OR “green”.

Finally, Spanner users can use graph, vector and full-text search all from the same database using standard SQL.

All of these sophisticated search capabilities can be leveraged using standard SQL language, and without the complexity of leveraging a separate search platform.

Operational simplicity and cost efficiency

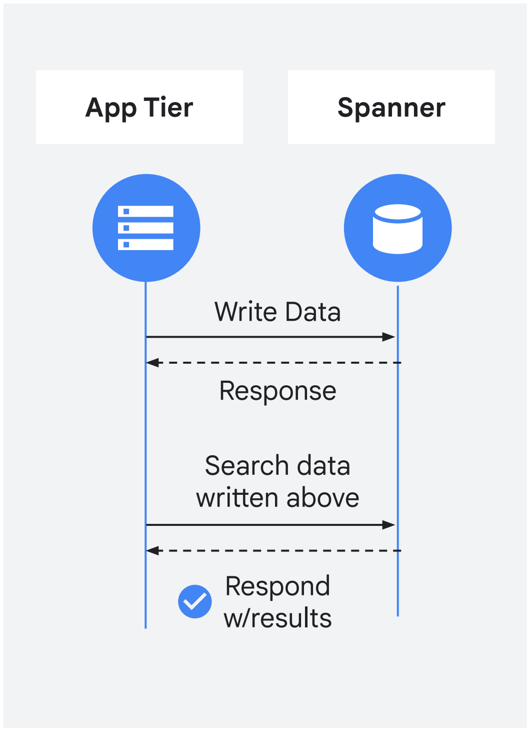
Spanner FTS delivers a rich set of search capabilities that always operate on the latest committed data, without having to copy data to other systems. Not having to spin up high throughput, scalable, resilient data movement pipelines between your OLTP database and the full-text search solution means a major reduction in operational toil. Furthermore, with Spanner FTS, resources are shared between operational and full-text search queries thus yielding better resource utilization and reduced costs compared to having dedicated systems. And lastly, documents stored in Spanner FTS benefit from all the enterprise features of Spanner including data protection, security, authentication/authorization, auto-scaling and compliance..



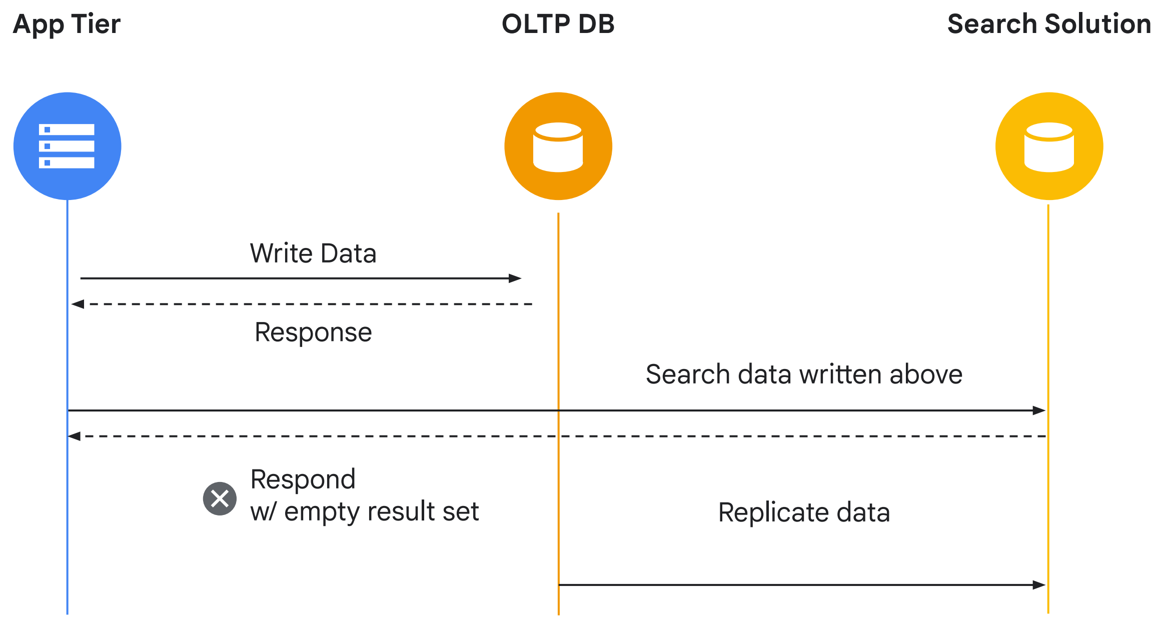
Real-time search results

Closely related to the operational simplicity point above is the benefit of transactionally-consistent search results offered by Spanner FTS. Spanner FTS search indexes are transactionally consistent and are updated in real time when data is committed, allowing for fresh real-time search results. This is in contrast to traditional architectures for supporting full-text search capabilities that are separate from the primary OLTP database.

This tightly integrated full-text search capability in Spanner allows for real-time queries against previously committed data. Compare this to the approach of shipping OLTP data to another search system, where it's possible for queries that fetch very recently committed data to return stale results, which in turn can mean frustrated and/or confused users. The following diagrams illustrate this in more detail. The first diagram shows the simplicity of leveraging Spanner FTS, where the searched data is transactionally consistent.



The diagram below conveys the potential for search queries against recently written data in traditional OLTP + Search architecture returning stale results due to the data copy lag associated with the ETL process. With Spanner FTS on the other hand, you always get real-time, transactionally-consistent search results.



How to use Spanner full-text search

At a very high level, there are 3 steps required to facilitate full-text searches:

01 Tokenize

The `TOKENIZE_FULLTEXT` tokenizer will generate the following list of tokens for "`"The quick brown fox jumps over the lazy dog"`: [`"The"`, `"quick"`, `"brown"`, `"fox"`, `"jumps"`, `"over"`, `"the"`, `"lazy"`, `"dog"`].

02 Index

You also have to create one or more search indexes using the `CREATE SEARCH INDEX` clause. This step indexes the tokens above and essentially maintains a map between the tokens and the original document. Secondary indexes don't allow you to efficiently search across multiple columns. FTS allows you to do this by creating a single search index over multiple tokenized columns.

03 Search

Perform search using the `SEARCH` function, which uses the index created in step 2 to ensure that lookups are efficient.

04 Score

Spanner supports computing a textual topicality score to gauge how well a query matches a row. The scores calculate relevance to queries based on term frequency and other customizable options. You can leverage SQL to influence the scoring.

For a more detailed description of tokenization, indexing and search, please see the [Spanner Full-Text search documentation.]

Let's set the stage with a schema that we'll use for the rest of this whitepaper. Our schema, backing a very basic e-commerce website, has three tables: Customers, Products and Reviews.

TABLE: Customers

```
CREATE SEQUENCE CustomerIdSeq OPTIONS (sequence_kind = 'bit_reversed_positive');
CREATE TABLE Customers (
 CustomerId INT64 DEFAULT (GET_NEXT_SEQUENCE_VALUE(SEQUENCE CustomerIdSeq)),
 FirstName STRING(MAX),
 LastName STRING(MAX),
 Phone STRING(25),
 Email STRING(255),
 Address1 STRING(255),
 Address2 STRING(255),
 City STRING(255),
 State STRING(255),
 Zip STRING(255),
 Country STRING(255),
) PRIMARY KEY(CustomerId);
```

Table: Products

```
CREATE SEQUENCE ProductIdSeq OPTIONS (sequence_kind =
'bit_reversed_positive');
CREATE TABLE Products (
 ProductId INT64 DEFAULT (GET_NEXT_SEQUENCE_VALUE(SEQUENCE ProductIdSeq)),
 Description STRING(MAX),
 CategoryId INT64 NOT NULL,
 Price NUMERIC,
) PRIMARY KEY(ProductId);
```

Table:Reviews

```
CREATE SEQUENCE ReviewIdSeq OPTIONS
(sequence_kind = 'bit_reversed_positive');
CREATE TABLE Reviews (
 ReviewId INT64 DEFAULT
(GET_NEXT_SEQUENCE_VALUE(SEQUENCE
ReviewIdSeq)),
 CustomerId INT64 NOT NULL,
 ProductId INT64 NOT NULL,
 Rating FLOAT64,
 ReviewText STRING(MAX),
) PRIMARY KEY(ReviewId);
```

Indexing

To start, it's illuminating to distinguish conventional secondary indexes from full-text search indexes. Unlike conventional secondary indexes, full-text search requires tokenization and subsequent indexing of this tokenized text. Tokenization is a fairly involved subject in its own right since multiple tokenization methods are available, including NGRAMS-based tokenization. However the basic premise of tokenization is straightforward. With this background in mind, let's see how the above mentioned schema for the Products table has to be adjusted to include tokenization:

```
Table: Products (with tokenization to support FTS)
```

```
CREATE TABLE Products (
 ... <columns removed for readability>
 Description STRING(MAX),
 Color STRING(MAX),
 Description_Tokens TOKENLIST AS (TOKENIZE_FULLTEXT(Description)) HIDDEN,
 Color_Tokens TOKENLIST AS (TOKENIZE_FULLTEXT(Color)) HIDDEN
) PRIMARY KEY(ProductId);
```

The first thing you might notice is the addition of a new hidden column named `Description_Tokens` to the table, which will hold tokenized data that can then be indexed. The column must be declared as `HIDDEN` so that it does not get returned in `SELECT`* results.

Note

`TOKENIZE_FULLTEXT` is just one of the tokenizers supported by Spanner FTS.

To see a list of other tokenizers you can use,
please visit the Spanner full-text search documentation.

Before you can query this data using full-text search, you have to create a search index:

```
CREATE SEARCH INDEX ProductDescriptionIndex ON Products(Description_Tokens)
```

Note that the search index is created on the hidden TOKENLIST column. While this example has one column in the index, if you want to include multiple columns in your search you would need to have all the columns you wish to search in the same index. Having the columns in the same index allows for efficient searches across columns. Here is an example of combining columns:

```
CREATE SEARCH INDEX ProductDescriptionIndex ON Products(Description_Tokens, Color_Tokens)
```

Querying

With the search index in place, you can now issue full-text search queries against the `Products` table. Let's start with a basic query to search for products whose description contains the word 'natural':

```
SELECT * FROM Products WHERE
SEARCH(Description_Tokens, "natural")
```

```
/*--------------+-----------------------------------------------------------*
 | ProductId    | Description                                               |
 +--------------+-----------------------------------------------------------+
 | 23           | All natural shampoo                                       |
 *--------------+-----------------------------------------------------------*/
```

The SEARCH function uses the previously indexed TOKENLIST column named Description_Tokens to fetch matching results.

Here is the syntax when searching multiple tokenized columns:

```
SELECT * FROM Products WHERE SEARCH(Description_Tokens, "Natural") AND
SEARCH(Color_Tokens,"GREEN")
```

Note

While SEARCH is a common way to retrieve search results, Spanner FTS offers other options for retrieving search results.

To see a list of retrieval options, please visit the Spanner full-text search documentation.

In addition to a purely token-based full-text search, Spanner supports a richer mode called enhance_query. When enabled, it extends the search query to include more token variants. These additions increase search recall.

To enable this option, you need to set the optional argument enhance_query=>true in the SEARCH function

```
SELECT * FROM Products WHERE SEARCH(Description_Tokens, "Natural",
enhance_query=>true)
```

```
/*--------------+-----------------------------------------------------------*
 | ProductId    | Description                                               |
 +--------------+-----------------------------------------------------------+
 | 23           | All natural shampoo                                       |
 | 103          | Naturally made from aloe vera                             |
 *--------------+-----------------------------------------------------------*/
```

Spanner also supports a SCORE function that allows you to rank the search results. You can then use it in the ORDER BY clause to display the search results in ranked order.

```
SELECT *, SCORE(Description_Tokens, "natural shampoo", enhance_query=>true) AS Score
FROM Products
WHERE SEARCH(Description_Tokens, "natural shampoo", enhance_query=>true)
ORDER BY Score DESC
```

```
/*-------------+-------------------------------------------------------------*
 | ProductId    | Description             | Score                            |
 +-------------+-------------------------------------------------------------+
 | 23           | All natural shampoo     | 0.4666191637516022               |
 | 58           | Naturally made shampoo  | 0.29731062054634094              |
 *-------------+-------------------------------------------------------------*/
```

With that very brief introduction to full-text search on Spanner, let's go over the full set of search capabilities supported by Spanner FTS.

Partitioning search indexes

Search indexes in Spanner can be partitioned or unpartitioned. To illustrate the differences, let's go back to the Reviews table from our e-commerce scenario.

```
Table: Reviews

CREATE TABLE Reviews (
 ReviewId INT64 DEFAULT (GET_NEXT_SEQUENCE_VALUE(SEQUENCE ReviewIdSeq)),
 CustomerId INT64 NOT NULL,
 ProductId INT64 NOT NULL,
 Rating FLOAT64,
 ReviewText STRING(MAX),
 Rating_Tokens TOKENLIST AS (TOKENIZE_NUMBER(Rating)) HIDDEN,
 ReviewText_Tokens TOKENLIST AS (TOKENIZE_FULLTEXT(ReviewText)) HIDDEN,
) PRIMARY KEY(ProductId);
```

And you will create two indexes

```
CREATE SEARCH INDEX UnpartitionedReviewTextIndex
ON Reviews(ReviewText_Tokens);
```

```
CREATE SEARCH INDEX PartitionedReviewTextIndex
ON Reviews(ReviewText_Tokens) PARTITION BY CustomerId;
```

Now let's say you want to search all reviews submitted by the customer with CustomerId '1' and containing the word 'child-proof' in the `ReviewText` column:

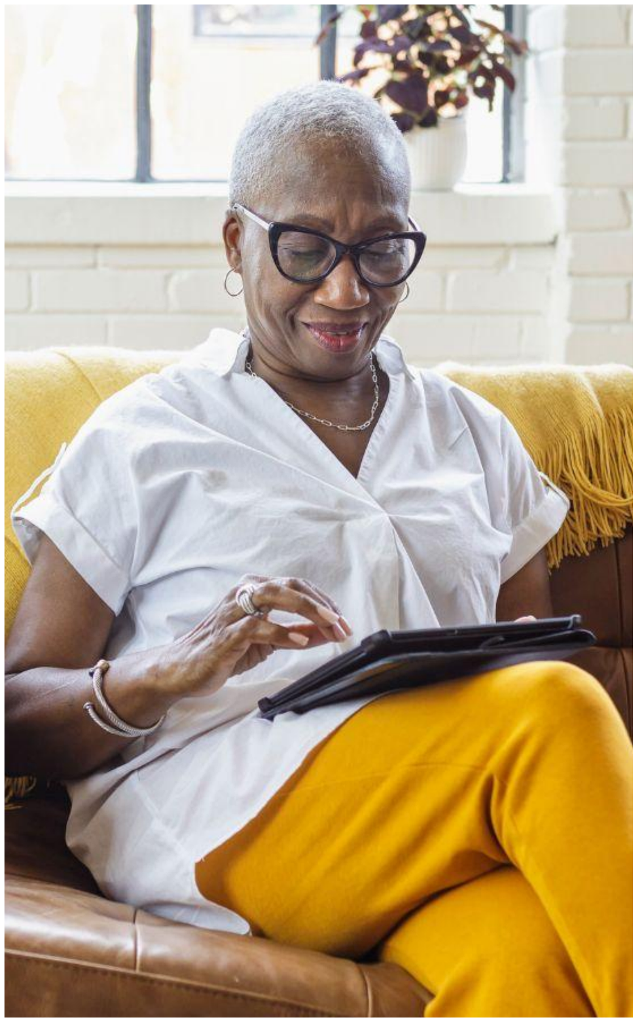
```
SELECT ReviewId
FROM Reviews
WHERE CustomerId = 1 AND SEARCH(ReviewText_tokens, 'child-proof')
```

Since the above query only searches reviews for a single customer, it'll use the second index `PartitionedReviewTextIndex` by default.

The partitioned index allows Spanner to only process rows with CustomerId=1, instead of searching through the entire data set. If on the other hand, you wanted to search reviews globally, you would issue the following query:

```
SELECT ReviewId FROM Reviews SEARCH(ReviewText_tokens, 'child-proof')
```

This query will use the first index `UnpartitionedReviewTextIndex` above by default. As you may have surmised, multiple search indexes might be necessary to serve your search needs. In general, we recommend that you use the finest granularity of partitioning that's appropriate for the use case at hand. So if you're building an admin page on your e-commerce website that allows internal employees to search reviews by CustomerId, then you should use the partitioned index `PartitionedReviewTextIndex` since it is more efficient. An unpartitioned index `UnpartitionedReviewTextIndex` is a better choice for a web page that allows customers to search all reviews. Spanner will choose the best index automatically. You just need to create the indexes.



Search index sort order

You can specify a sort order when creating search indexes. Let’s slightly adjust our Reviews table schema to include another column (`TimestampMicro`):

```
Table: Reviews
```

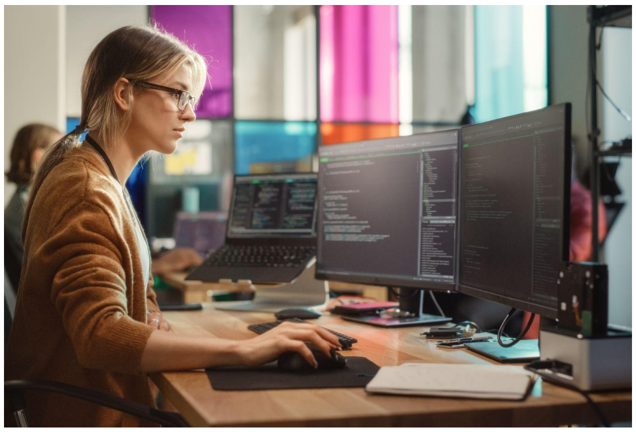
```
CREATE TABLE Reviews (
 ReviewId INT64 DEFAULT (GET_NEXT_SEQUENCE_VALUE(SEQUENCE ReviewIdSeq)),
 CustomerId INT64 NOT NULL,
 ProductId INT64 NOT NULL,
 Rating FLOAT64,
 ReviewText STRING(MAX),
 TimestampMicro INT64 NOT NULL,
 Rating_Tokens TOKENLIST AS (TOKENIZE_NUMBER(Rating)) HIDDEN,
 ReviewText_Tokens TOKENLIST AS (TOKENIZE_FULLTEXT(ReviewText)) HIDDEN,
) PRIMARY KEY(ProductId);
```

You could then define a search index with a specified sort order like so:

```
CREATE SEARCH INDEX SortedReviewTextIndex
ON Reviews(ReviewText_Tokens) ORDER BY TimestampMicro DESC;
```

With this index in place, we can issue an efficient search query to retrieve reviews matching a phrase in descending order of `TimestampMicro:`

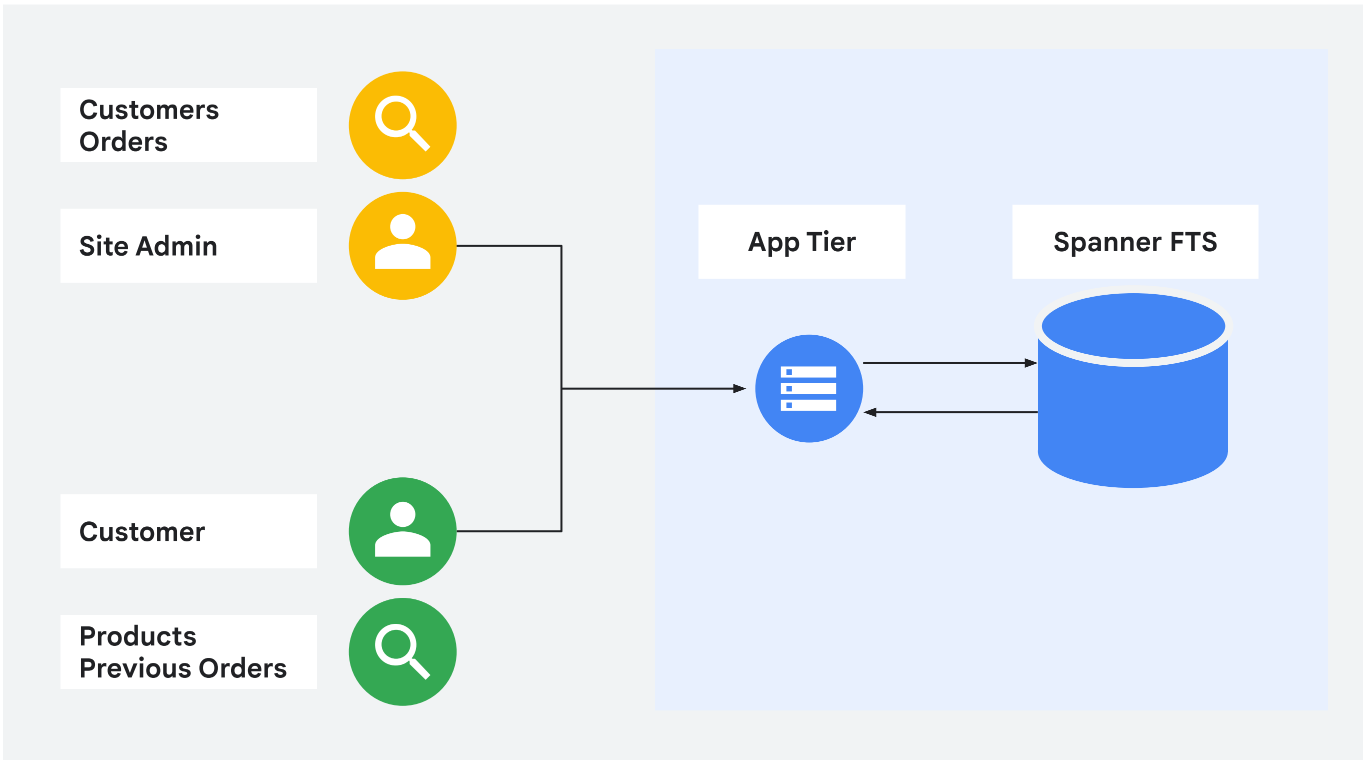
```
SELECT ReviewId, ReviewText, TimestampMicro FROM Reviews WHERE
SEARCH(ReviewText_Tokens, ‘child-proof’) ORDER BY TimestampMicro DESC;
```



Note that the built-in `TIMESTAMP` type for the `ORDER BY` clause was not used in the above index definition. This is because Spanner search indexes behave differently from Spanner secondary indexes when it comes to sort order. You can only use INT64 columns for the sort order of a search index. This restriction stems from the internal layout of search indexes; more details can be found here (Spanner full-text search documentation.) This is why the `TIMESTAMP` type was not used. `TIMESTAMP` uses nanosecond precision which doesn’t fit in a 64-bit integer.

Sample scenario: e-commerce site

We'll use an e-commerce website example to bring together some of the notable capabilities offered by Spanner FTS. The following diagram captures the key interactions originating from two personas, the site-administrator and the customer, that require different types of search capabilities that can be supported by Spanner FTS.



With the above context in mind, let's explore some key search queries supported by Spanner FTS.

Customer facing page to retrieve a ranked list of products

When a search results in multiple results, as is often the case, it is helpful to rank the results. You can use the `SCORE` function to perform ranking as shown below.

```
SELECT ProductId, Description,
      SCORE(Description_Tokens, "Giraf toy", enhance_query=>true) AS Score
FROM Products
WHERE SEARCH(Description_Tokens, "Giraf toy", enhance_query=>true)
ORDER BY Score DESC;
```

```
/*--------------+------------------------------------+---------------------*
 | ProductId    | Description                        | Score               |
 +--------------+------------------------------------+---------------------+
 | 5            | Giraf toys                         | 1.0278141498565674  |
 | 7            | Big giraffe toy                    | 0.29731559753417    |
 | 3            | Toys girafe                        | 0.2973131239414215  |
 *--------------+------------------------------------+---------------------*/
```

The SCORE function computes a score for each query term and then combines the scores. The per-term score is roughly based on term frequency–inverse document frequency (TF/IDF). The score is one component of the final ordering for a record. A query can combine it with other signals such as the freshness modulating the topicality score.

Using NGRAMS and SOUNDEX for spelling variants

One use of NGRAMS is that it allows you to search for modification in spelling for specific terms. A good example is when the search criteria is typed in by users and may contain spelling errors for things like a person's name, a product brand name, or have a digit off in a phone number being searched for. Note that enhance_query can also be used with misspelled words entered as the search criteria, but it is limited to matching normal dictionary words and not proper nouns like a person's name, as well as number strings that are also not dictionary words. Applications may need to use multiple FTS search techniques. There are some misspellings that can be corrected with ngrams, some with phonetic search and some with enhance_query. It is sometimes appropriate to combine them all together. This can be achieved with SQL using the UNION ALL clause. The specific query will vary for each scenario.

Here is an example of how to use NGRAMS to search for alternate spellings of names. Names of people can take different forms and also be misspelled very easily. Examples include alternate spellings like "Jeffrey" and "Jeffery", as well as different variants like "Jenn", "Jenny" and "Jennifer". A user may want to find all possible matches in these cases when searching a user database. Let's see how Spanner FTS can assist with this using a modified version of the Customer table.

```
Table: Customers
```

```
CREATE TABLE
 FirstName_Tokens TOKENLIST AS (TOKENIZE_NGRAMS(FirstName, ngram_size_max=>3)) HIDDEN,
  FirstName_Soundex STRING(MAX) AS (LOWER(SOUNDEX(FirstName))),
) PRIMARY KEY(CustomerId);
```

And you will create two indexes

```
CREATE SEARCH INDEX FirstNameSubstrIndex
ON Customers(FirstName_Tokens);
```

```
CREATE INDEX FirstNamePhoneticIndex
ON Customers(FirstName_Soundex);
```

```
SELECT * FROM Customers WHERE SEARCH_NGRAMS(FirstName_Tokens, "Jeffery")
```

```
/*--------------+----------------------------------------------------------*
 | CustomerId   | FirstName                                                |
 +--------------+----------------------------------------------------------+
 | 23           | Jeff                                                     |
 | 103          | Jeffrey                                                  |
 *--------------+----------------------------------------------------------*/
```

The next example shows how to sort the results by relevance using `SCORE.`

```
SELECT *
FROM Customers
WHERE SEARCH_NGRAMS(FirstName_Tokens, "Jennifer")
ORDER BY SCORE_NGRAMS(FirstName_Tokens, "Jennifer") DESC
```

```
/*--------------+----------------------------------------------------------*
 | CustomerId   | FirstName                                                |
 +--------------+----------------------------------------------------------+
 | 1            | Jennifer                                                 |
 +--------------+----------------------------------------------------------+
 | 2            | Jenn                                                     |
 +--------------+----------------------------------------------------------+
 | 3            | Jenny                                                    |
 *--------------+----------------------------------------------------------*/
```

SOUNDEX allows you to search for terms that are spelled differently but sound the same.

```
SELECT *
FROM Customers
WHERE FirstName_Soundex = LOWER(SOUNDEX("Stephen"))
```

```
/*-------------+----------------------------------------------------------*
 | CustomerId   | Description                                              |
 +-------------+----------------------------------------------------------+
 | 32           | Stephen                                                  |
 | 53           | steven                                                   |
 *-------------+----------------------------------------------------------*/
```

NGRAMS and enhance_query have some overlap in that they can both detect spelling differences between search text and matched text. The differences are outlined below.

	NGRAMS	enhance_query
Cost	Requires a more expensive substring tokenization based on ngrams	Requires a less expensive full-text tokenization
Search query types	Works well with short documents of about 1 or 2 words like a person's name or a city name	Works equally well with any size documents and any size search queries
Partial words search	A substring search that allows for misspellings	Only supported for full words search (SEARCH_SUBSTRING does not support the enhance_query argument)
Misspelled words	Supports misspelled words in either index or query	Only supports misspelled words in the query
Corrections	Finds any combination of spelling differences even if it's not a real word. It also works on proper nouns/names.	Corrects misspelling of common well known words and may not work for proper nouns/names.

List of Spanner full-text search capabilities

The basics of Spanner FTS were covered in a previous section and some other interesting search use cases supported by Spanner FTS covering a sample e-commerce application were discussed. However, Spanner FTS can do quite a bit more and this section provides a map of all the capabilities supported by Spanner FTS. For each of the capabilities, the table notes how an application would leverage the feature. The methods used can be standard SQL, enhance_query option, using NGRAMS, SCORE or SOUNDEX.

Full-Text Search Feaures	Spanner FTS method
Search Index	CREATE SEARCH INDEX
Keyword matching	Standard SQL
Phrase matching	SEARCH
Exact term matching	SEARCH
Case insensitive matching	SEARCH
NGRAMS tokenization	NGRAMS
Match similar-sounding words	SOUNDEX
Auto Language Detection	SEARCH
Proximity search	SEARCH
Substring and prefix search	Standard SQL
Wildcard search	Standard SQL
Boolean queries	Standard SQL
Filtering	Standard SQL
Contextual Number matching (“5” == “five”)	SEARCH with enhance_query
Boosting	Standard SQL
Results ranking	SCORE, SCORE_NGRAMS
Custom ranking	SCORE+SQL, SCORE_NGRAMS+SQL
Aggregations	Standard SQL
Snippets	SNIPPET
Governance	Standard SQL
Spell correction	NGRAMS, enhance_query
Contextual stop word handling	enhance_query
Contextual pluralization	enhance_query, NGRAMS (non-contextual)
Contextual Synonym Matching	enhance_query

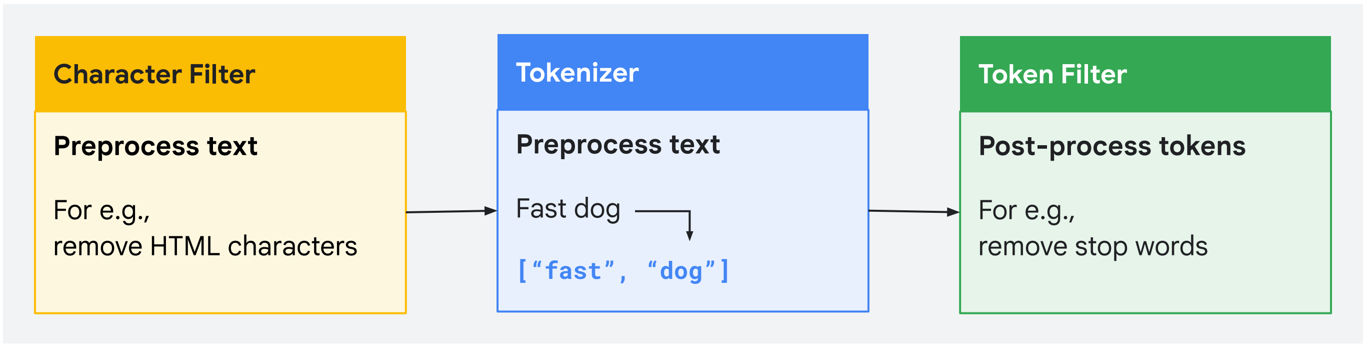
In addition to the above, there are some other capabilities that users coming from systems such as Elastic might expect. We describe how you can accomplish the broader goals supported by these capabilities in the Migration section below.

Migrating to Spanner FTS

In this section, we’ll look at functionality that you may be using in other full-text search systems and how you can perform them in Spanner. Spanner FTS ships with state of the art tokenizers and ranking algorithms to provide a simple out of the box search experience without the complex configurations associated with traditional FTS systems.

Analyzers during indexing

Search systems often support customer analyzers. To briefly summarize, an analyzer consists of three main components as illustrated in the diagram below.



Spanner FTS comes with out-of-the-box support for use cases that often require custom analyzers in other systems. We'll walk through a few examples to demonstrate.

Stemming

Stemming refers to a specific technique to analyze words that are a 'stem' of others, where a specific word is incorporated in another, such as "run", "runner" and running", We'll use an example to illustrate stemming - when searching for the word 'fishing' and returning 'fish''. While other systems might require you to set up and configure custom analyzers to facilitate retrieval of variations of a search term, Spanner FTS offers this capability via its enhanced query capability.

```
SELECT ProductId, Description FROM Products WHERE SEARCH(Description_Tokens, "fishing",
enhance_query=>true)
```

```
/*-------------+---------------------------------------------------------------*
 | ProductId   | Description                                                   |
 +-------------+---------------------------------------------------------------+
 | 15          | Fish love this food                                           |
 *-------------+---------------------------------------------------------------*/
```

Notice that in the examples above, you don't have to configure anything additional besides declaring your schema and search index as described in the section Basics of full-text search on Spanner. Simply by setting `enhance_query` to true when invoking the `SEARCH` function, FTS will automatically cover stemming use cases based upon the context of the search input.

NGRAMS-based tokenization

NGRAMS are outlined above and are a common type of analyzer employed by customers using dedicated search platforms. NGRAMS are important for a number of reasons. One example would be the spelling variations of individual words. While the FTS 'enhance_query' option will automatically deal with some spelling differences, NGRAMS is a better option particularly when dealing with proper nouns or other words that would not appear in a language dictionary, or standalone words that are not part of a phrase being searched for. Common examples of this are first and last names. Another common use case is dynamic filtering. See the prior section covering NGRAMS and SOUNDEX for more details and examples.

Custom dictionaries

Custom dictionaries are used in other full-text search systems. An example usage of this would be to handle synonyms. Spanner FTS handles some of these use cases without the need for custom dictionaries, thanks to `enhance_query`:

```
SELECT ProductId, Description FROM Products WHERE SEARCH(Description_Tokens, "car
windshield", enhance_query=>true)
```

```
/*-------------+--------------------------------------------------------*
 | ProductId    | Description                                           |
 +-------------+--------------------------------------------------------+
 | 120           | Vehicle Windshield                                   |
 | 155           | Car Windshield                                       |
 *-------------+--------------------------------------------------------*/
```

In the above example, phrases with car and vehicle are interchangeable in the search criteria without the need for custom dictionaries. This is an example of leveraging Google’s expertise in web search in Spanner, which allows you to not have to maintain custom dictionaries in most cases

Aggregations

In addition to full-text search, customers often use Elastic or OpenSearch to perform aggregations outside of the source OLTP database. Spanner natively supports many aggregate functions. The following is an aggregation example that counts the number of products of a certain type.

```
SELECT CategoryId, COUNT(ProductId) as NumProducts FROM Products WHERE
Search(Description_Tokens, “Builder glue”, enhance_query=>true) GROUP BY CategoryId;
```

```
/*-------------+--------------------------------------------------------*
 | CategoryId   | NumProducts                                           |
 +-------------+--------------------------------------------------------+
 | 1            | 4                                                     |
 | 2            | 2                                                     |
 *-------------+--------------------------------------------------------*/
```

Conclusion

Spanner full-text search enables you to perform efficient searches against business critical data while doing away with the costs and complexities associated with a separate, bolted-on solution. Spanner allows you to combine your online relational or key-value database and your search database into one, which provides operational simplicity and cost savings in addition to the other benefits of Spanner such as being fully managed, provides virtually unlimited scale, and is highly available. You can build new or extend existing applications with rich search capabilities that take advantage of the high performance and availability at virtually unlimited scale of Spanner while retaining transactional properties of your searched data that your business depends on.