TSearch2全文搜索

1.简介

1.1　全文搜索的概念和重要性

1.1.1 概念

全文搜索是指在文本内容中进行全面搜索的过程，包括对文本进行解析、分词、建立索引，并通过检索技术找到与检索条件匹配的文档或内容。

1.1.2 重要性

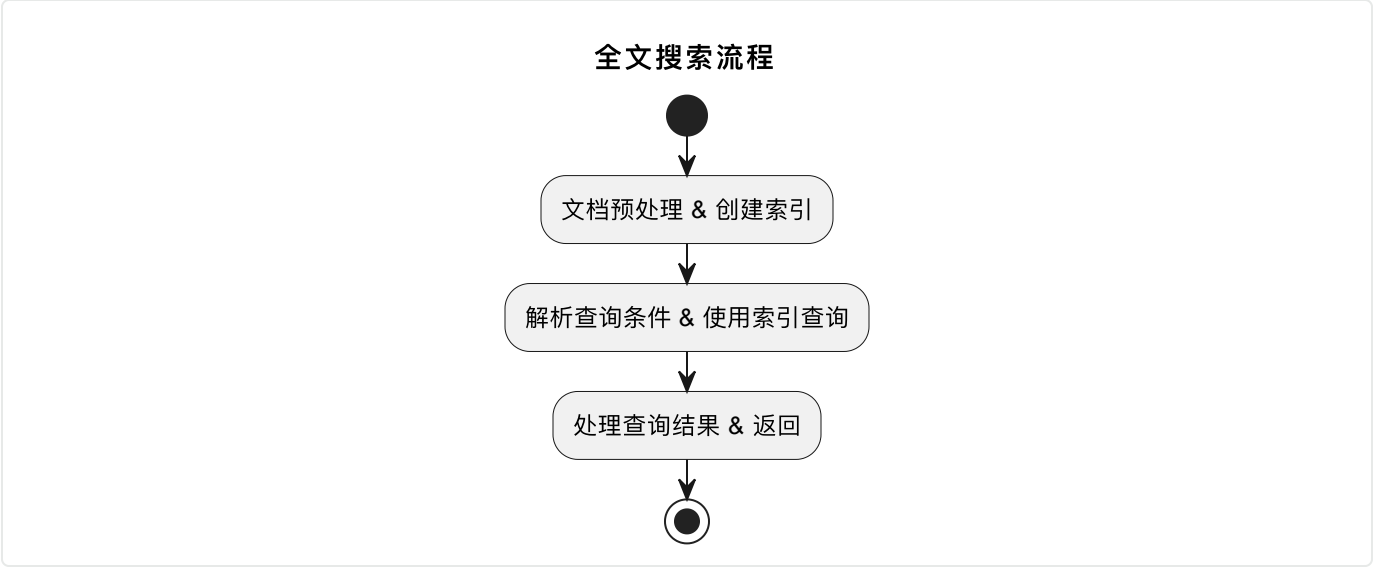
1. 提高检索效率：可以将文本内容进行索引的建立，从而加快检索的速度，提高搜索效率。
2. 提升用户体验：可以提供更准确、全面的搜索结果，满足用户对信息检索的需求，从而提升用户体验。
3. 支持语义检索：全文搜索不仅可以检索特定关键词，还可以根据语义关系进行搜索，提供更精准的搜索结果。
4. 促进信息发现：全文搜索技术可以帮助用户快速发现文本内容中隐含的信息，促进信息的发现和利用。
5. 支持大数据应用：在大数据环境下，全文搜索技术可以帮助用户快速检索和分析海量文本数据，发现有价值的信息。

1.2　全文搜索在信息检索和文档处理中的应用

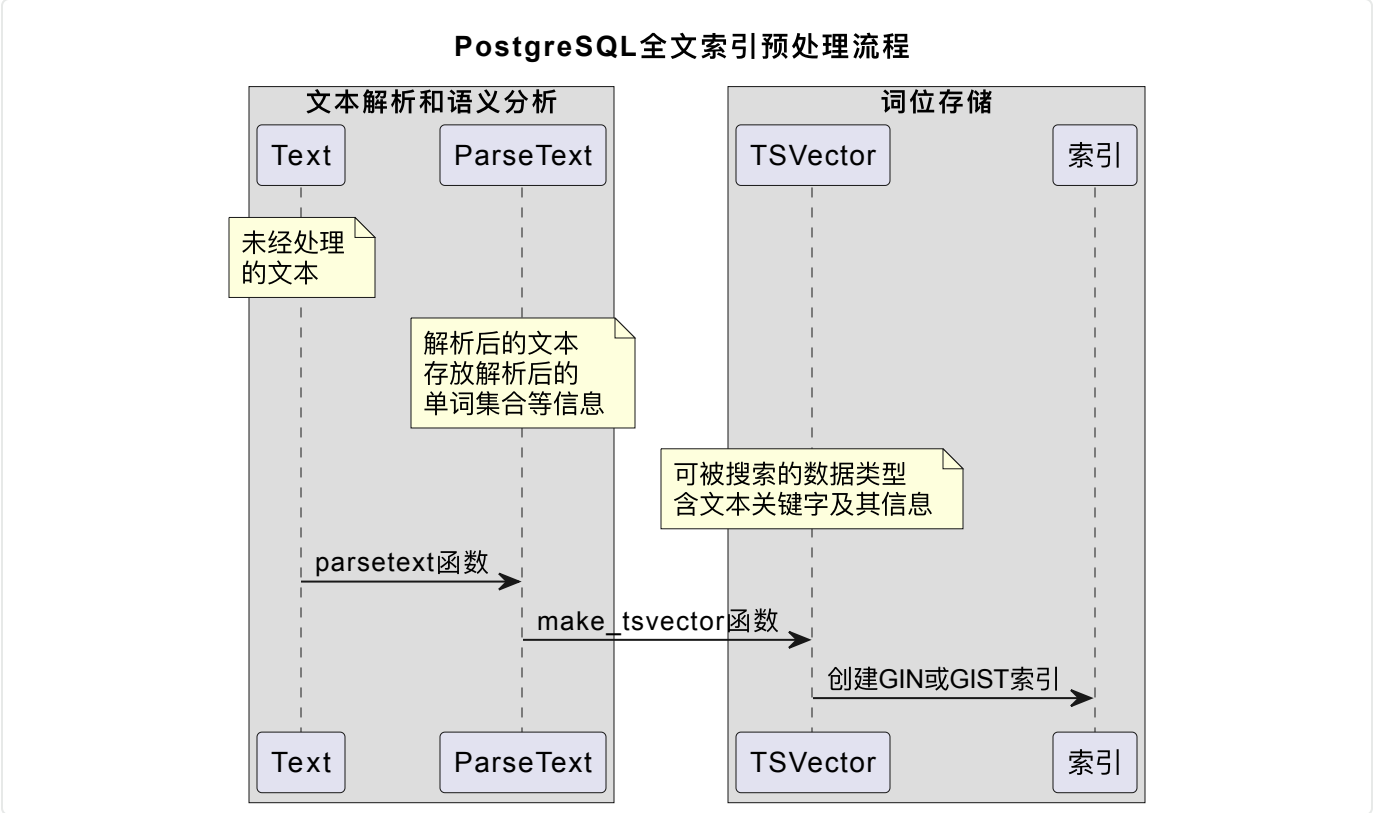
1. 信息检索：全文搜索可以**帮助用户快速准确地搜索到所需的信息**，提高检索效率和准确性。不仅可以根据关键词检索，还可以根据文档的内容和语义进行检索，提供更加精准的搜索结果。
2. 文档管理：全文搜索技术可以**帮助用户管理大量文档和内容**，通过建立全文索引，可以快速找到相关文档，减少查找时间，提高工作效率。
3. 数据分析：在大数据环境下，全文搜索可以帮助用户分析文本数据，**发现数据之间的关联关系和隐藏信息**，帮助用户做出更加准确的决策。
4. 信息发现：全文搜索可以帮助用户**发现文本内容中的有价值信息**，以便进一步研究和应用。通过全文搜索技术，用户可以更好地利用文档中的知识和信息。

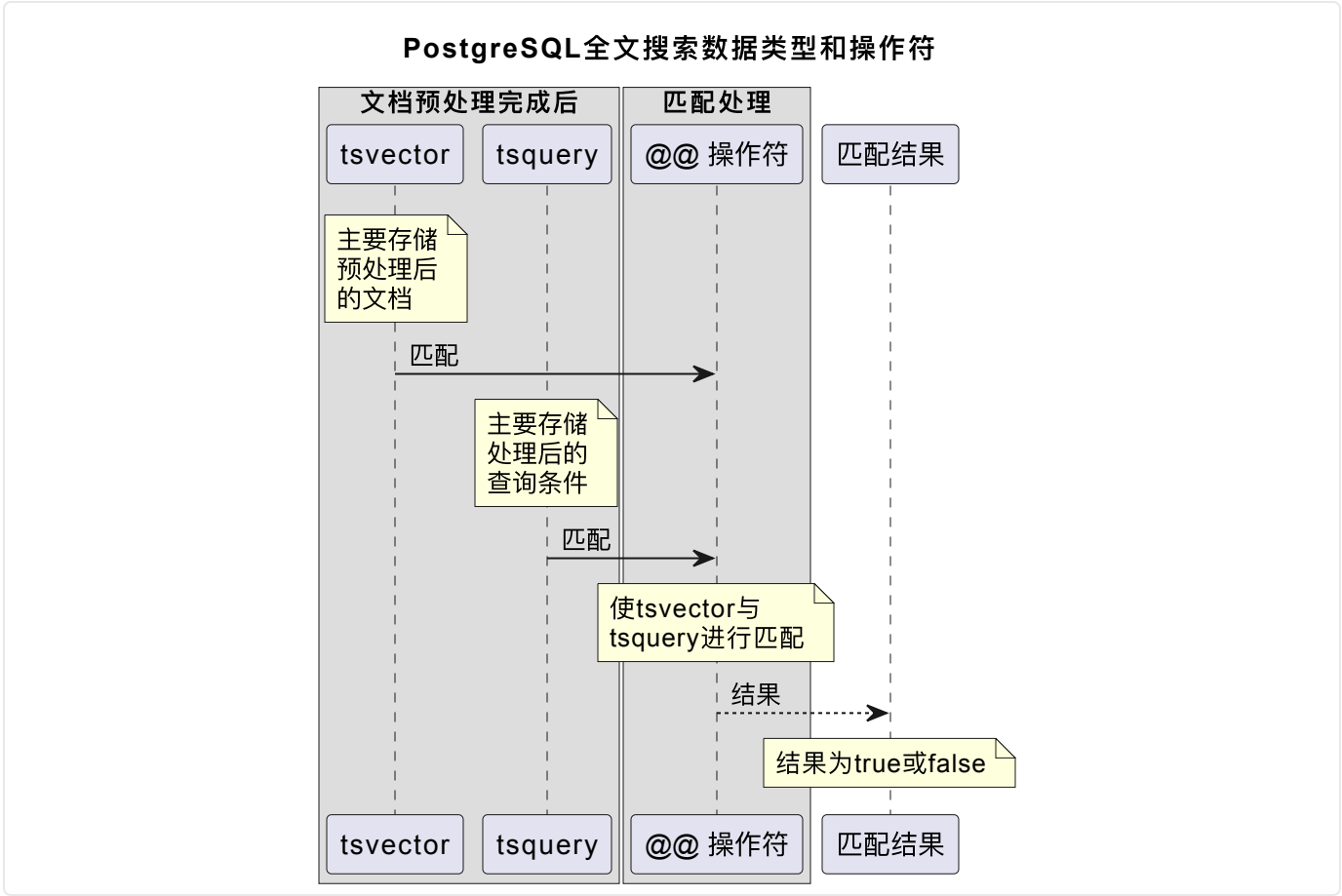
2.全文索引的过程

2.1 全文索引的流程



2.2 全文索引预处理流程





2.2.1 文本解析(基本预处理)

1. **字符过滤**：在解析之前，系统会去除文本中的标点符号、空白字符等非意义元素，确保后续处理集中在有效词汇上
2. **分词**：将文本切分为单词或术语（tokens）,PostgreSQL的全文检索支持多种语言的分词器，选择合适的分词器对于准确解析文本至关重要
3. **大小写统一**：为了提高匹配率，通常会将所有词汇转换为小写或大写，除非有特定的大小写敏感需求
4. **词干提取**：通过词干提取（stemming）或词形还原（lemmatization）减少词汇的形态变化，将词汇归一化为其基本形式，比如将“running”还原为“run”

2.2.2 语义分析（进一步预处理）

1. **停用词过滤**：移除常见但对搜索意义不大的词汇，如“the”，“is”，“in”等，以减小索引体积和提高搜索效率
2. **同义词处理**：识别并处理同义词，确保用户搜索时能够匹配到所有相关的词汇变体
3. **词性标注**（可选）：虽然不是所有全文检索系统都会做，但对某些高级应用来说，识别词汇的词性有助于提升搜索的精准度和语义理解

4. **专业术语识别**：对于特定领域，可能需要识别并特别处理专业术语，以提高检索的针对性

2.2.3 词位存储

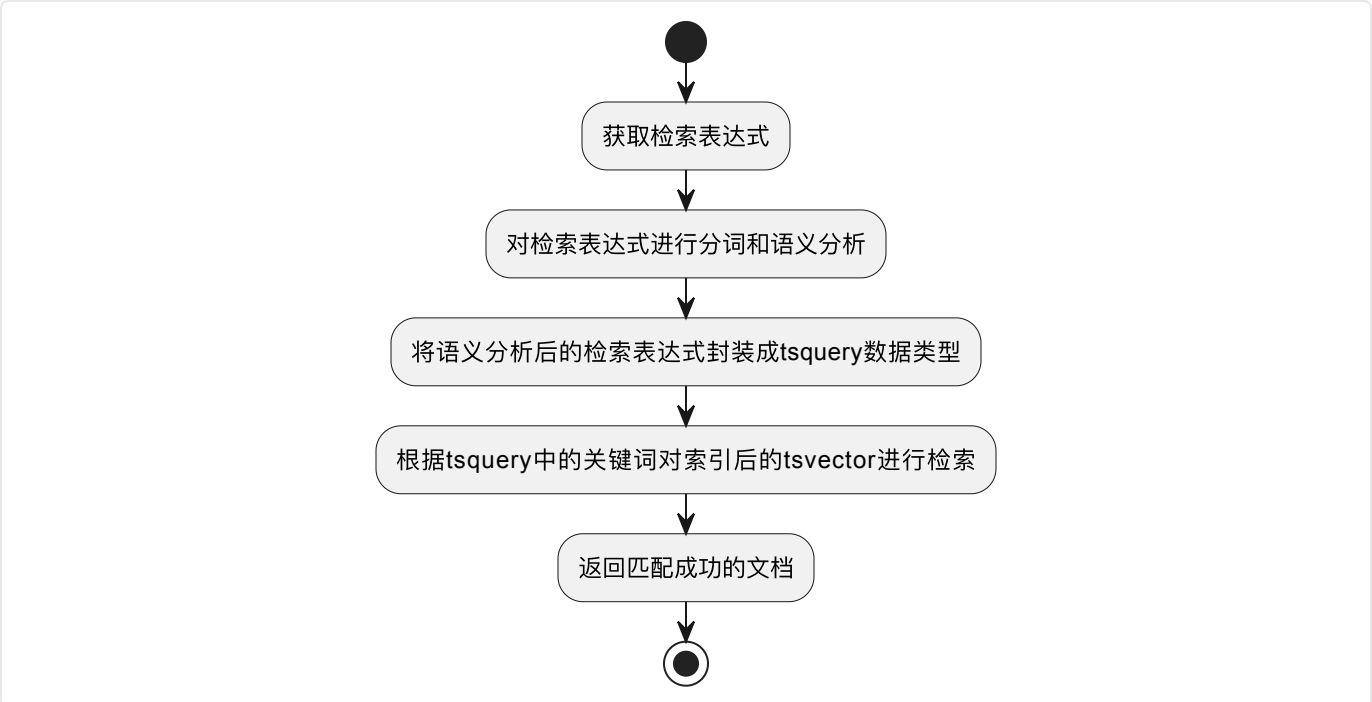
1. **位置信息提取：** 在文本解析阶段，系统需要提取每个词汇在文本中的位置信息，包括其在文本中的起始位置和结束位置。这一过程确保了系统能够准确地记录每个词汇在原始文本中的位置
2. **位置信息存储：** 提取的位置信息需要被存储起来，以便后续的搜索和匹配操作，通常采用数据结构来存储位置信息，如倒排索引。这一过程确保了系统能够快速地定位和匹配文本数据
3. **索引构建：** 存储位置信息的数据结构需要被构建成可用于搜索的索引，这一过程通常包括索引的建立、更新和维护。倒排索引是常用的索引技术之一，通过它可以实现高效的文本搜索

2.2.4 索引优化的常见策略

- **索引配置调整**：确保TSearch2的配置符合自己的数据特性和查询需求，比如选择合适的分词器、停用词列表和同义词典等。
- **索引更新策略**：根据数据更新频率设计合理的索引更新策略，例如使用定时任务或者触发器来更新索引，平衡查询性能与索引新鲜度。
- **索引分割**：对于大型数据集，考虑使用分区或分片技术来分散索引，减少查询延迟并提高系统的可扩展性。

3.全文索引的查询

3.0　查询流程图



3.1 查询语法

3.1.1 基本全文查询

最简单的全文查询使用@@操作符来比较一个文档字段和一个查询字符串：

```sql
SELECT * FROM your_table WHERE to_tsvector(your_column) @@ to_tsquery('查询字符串');
```

其中，to_tsquery函数将搜索词转换为TSearch2可以理解的查询格式，以便与全文搜索索引进行比较。

For example：

假设我们有一个表 products，其中有一列 description 存储了产品的描述信息。如果我们想要搜索包含关键词“手机”的产品，可以使用以下查询语句：

```sql
SELECT * FROM products WHERE to_tsvector(description) @@ to_tsquery('手机');
```

这条查询将返回在 description 列中包含关键词“手机”的产品记录。通过使用 @@ 操作符和 to_tsquery() 函数，我们可以进行基本的全文搜索查询，以便找到符合条件的产品。

3.1.2 查询构造

使用plainto_tsquery而不是to_tsquery可以创建一个更宽松的查询，它不会对搜索词进行词干提取或特殊字符处理。这意味着你可以更自由地指定搜索条件，而不必担心搜索词的精确形式。

```sql
SELECT * FROM your_table WHERE to_tsvector(your_column) @@ plainto_tsquery('查询字符串');
```

For example:

假设搜索"running shoes"

如果你使用 to_tsquery，它可能会匹配到包含 "run"、"running"、"runner"等形式的内容，因为它们都被转换成了“run”这个词根形式。这种情况下，搜索结果可能会包含更多种类的内容，但有些内容可能与搜索意图不太相关。

而如果你使用 plainto_tsquery，它只会匹配到确切包含“running shoes”这个词组的内容，因此搜索结果可能更加精确，符合你的搜索意图。

phraseto_tsquery可以用来搜索短语，保持词序不变。

短语 SQL

```sql
SELECT * FROM your_table WHERE to_tsvector(your_column) @@ phraseto_tsquery('查询字符串');

-- For example:
-- SELECT * FROM your_table WHERE your_column @@ phraseto_tsquery('running shoes');
-- 用phraseto_tsquery函数匹配整个短语
```

3.1.3 布尔操作符

在查询字符串中，你可以使用&（AND）、|（OR）、!（NOT）来进行布尔逻辑操作，例如：

SQL

```sql
SELECT * FROM your_table WHERE to_tsvector(your_column) @@ to_tsquery('good & bad');
```

3.1.4 模糊搜索

虽然TSearch2主要用于精确的全文搜索，但可以通过调整配置或使用模糊匹配的词干形式来实现一定程度的模糊搜索。

模糊搜索代码演示 SQL

```sql
-- 创建表
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    description TEXT
);

-- 随便插入一些数据
INSERT INTO products (description) VALUES
('These are running shoes'),
('This is a pair of sports shoes'),
('Comfortable shoes for walking'),
('Running shoes with extra cushioning'),
('Trail running shoes for outdoor adventures');

-- 安装并启用 TSearch2 扩展
CREATE EXTENSION tsearch2;

-- 进行模糊搜索来查找与关键词相匹配的记录，假设我想要搜索包含单词 “shoe” 的记录
SELECT * FROM products WHERE to_tsvector(description) @@ to_tsquery('shoe:*');

-- 如果有多个关键词，比如run和shoe
SELECT * FROM products WHERE to_tsvector(description) @@ to_tsquery('shoe:* & run:*');
```

简单来说，查询时，指定一个或多个关键词并在其后面使用 :* 就可以实现模糊搜索

3.1.5 权重与排序

TSearch2允许对搜索结果按照相关性排序，这通常是通过在索引时为文档的不同部分分配不同的权重来实现的。查询时可以使用ts_rank或ts_rank_cd函数来计算文档的相关度分数，并据此排序结果。

```sql
SELECT *, ts_rank(to_tsvector('english', content), to_tsquery('english', '搜索的关键词')) AS rank
FROM documents
WHERE to_tsvector('english', content) @@ to_tsquery('english', '搜索的关键词')
ORDER BY rank DESC;

--  english是指定的文本搜索配置。PostgreSQL支持多种语言的文本搜索，并且每种语言都有其特定的词汇规则、停用词列表等。
--  这里选用的是english配置
```

分析：

ts_rank(to_tsvector('english', content), to_tsquery('english', 'search')) AS rank：

这是计算每个文档与搜索关键词相关性的部分。这里做了两步操作：

- to_tsvector('english', content)：**将文档的content字段转换为一个文本搜索向量（tsvector）**。这个过程会将文本分解成单词，去除停用词（如“the”、“is”等常见但对搜索意义不大的词语），并为每个单词分配一个权重，基于它在文档中的位置（如标题比正文中的词更重）。
- to_tsquery('english', '搜索的关键词')：**将用户输入的搜索关键词转换为一个文本搜索查询（tsquery）**。这也会处理关键词的规范化，使其能有效地与tsvector进行匹配。
- ts_rank(...)：计算上述tsvector（文档内容转换后的表示）与tsquery（搜索关键词转换后的表示）之间的相关性排名。分数越高，表示文档与搜索关键词越相关。

WHERE to_tsvector('english', content) @@ to_tsquery('english', '搜索的关键词')：

这是筛选条件，用来过滤出那些内容中确实包含了与搜索关键词匹配的文档。符号@@用于比较tsvector和tsquery，如果文档内容与搜索关键词匹配，则返回true，否则返回false。

ORDER BY rank DESC：

根据之前计算的相关性排名（rank）降序排列查询结果。这样，最相关的文档会首先显示。

3.1.6 使用模板查询

TSearch2支持使用模板查询（template queries），允许插入变量到查询字符串中，这对于动态生成查询非常有用。

模板查询示例 SQL

```sql
-- 假设有一个名为articles的表，其中有一个名为content的列，你想要通过该列进行搜索。
-- 你希望创建一个模板查询，允许你动态指定搜索词。

-- 定义一个函数，接受搜索查询作为输入，并返回搜索结果
CREATE OR REPLACE FUNCTION search_articles(query_text TEXT)
RETURNS TABLE (
    article_id INT,
    title TEXT,
    content TEXT,
    rank DOUBLE PRECISION
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        article_id,
        title,
        content,
        ts_rank(to_tsvector('english', content), to_tsquery('english', query_text)) AS rank
    FROM
        articles
    WHERE
        to_tsvector('english', content) @@ to_tsquery('english', query_text)
    ORDER BY
        rank DESC;
END;
$$ LANGUAGE plpgsql;

-- 现在使用带有动态搜索查询的函数
-- 例如，搜索包含词语 'technology' 的文章：
SELECT * FROM search_articles('technology');
```

在这个示例中，我们创建了一个名为search_articles的函数，它接受一个搜索查询作为输入。在函数内部，我们使用这个查询来动态生成全文搜索，使用to_tsvector和to_tsquery，然后计算相关度分数使用ts_rank。最后，我们返回按照相关度分数排序的搜索结果。

3.1.7 特殊查询语法

你可以使用*作为通配符，但是要注意它是通过to_tsquery的特性来实现的，不是直接在查询字符串中使用。

使用:*来搜索词的开头，*:搜索词的结尾，或者*:middle:*搜索中间含有特定词的部分

```sql
-- 搜索以 'apple' 开头的词语
SELECT *
FROM documents
WHERE to_tsvector('english', content) @@ to_tsquery('english', 'apple:*');

-- 搜索以 'apple' 结尾的词语
SELECT *
FROM documents
WHERE to_tsvector('english', content) @@ to_tsquery('english', '*:apple');

-- 搜索包含 'middle' 的词语
SELECT *
FROM documents
WHERE to_tsvector('english', content) @@ to_tsquery('english', '*:middle:*');
```

3.1.8 配置和自定义

TSearch2的强大之处还在于其高度可配置性，你可以自定义词典、同义词表、停用词列表等，以适应特定领域的搜索需求。

代码示例 SQL

```sql
-- 创建一个自定义词典
CREATE TEXT SEARCH DICTIONARY custom_dict (
    TEMPLATE = pg_catalog.simple, -- 使用simple模板,通常意味着只进行基本的文本处理,
                                  -- 比如转换成小写等
    DictFile = pg_catalog.pg_bsd_hunspell, -- 指定词典文件的名称和路径,
                                           -- 这里是使用Hunspell兼容的词典文件格式
    Dictionary = 'english', -- 实际的词典内容文件名
    AffFile = 'english', -- 包含词缀信息的文件，用于支持词干提取和词形还原等功能
    StopWords = 'english' -- 停用词列表
);

-- 创建一个自定义同义词词典
CREATE TEXT SEARCH CONFIGURATION custom_config (
    COPY = pg_catalog.english
);

-- 将自定义同义词词典应用到自定义配置中
ALTER TEXT SEARCH CONFIGURATION custom_config
    ALTER MAPPING FOR word, asciiword WITH custom_dict, english_stem;

-- 创建一个停用词列表
CREATE TEXT SEARCH STOPWORD custom_stopwords (
    STOPWORDS = '{a, an, the, in, on, at, to, of}'
);

-- 将停用词列表应用到自定义配置中
ALTER TEXT SEARCH CONFIGURATION custom_config
    ALTER STOP WORD LIST TO custom_stopwords;

-- 创建一个全文索引
CREATE INDEX document_fulltext_idx ON documents USING gin(to_tsvector('custom_config', content));

-- 使用全文搜索进行查询
SELECT * FROM documents WHERE to_tsvector('custom_config', content) @@ to_tsquery('custom_config', 'search term');
```

示例中，我们创建了一个自定义的词典 custom_dict，一个自定义的同义词配置 custom_config，和一个停用词列表 custom_stopwords。然后我们将这些配置应用到了一个文档表的全文索引中，并使用 to_tsvector() 将文档内容转换为适合全文搜索的向量，使用 to_tsquery() 进行查询。

可以根据自己的需求来调整和扩展这些配置，以适应不同的场景和要求。

3.2 查询性能优化

1. 索引优化：在 TSearch2 中，可以使用 GiST 索引或 GIN 索引来提高全文搜索的性能。确保在需要搜索的列上创建适当的索引，以加快搜索速度
2. 配置调优：可以通过调整 TSearch2 的配置参数来优化查询性能，如调整搜索器的配置参数、配置对应的词典、停用词表等
3. 数据预处理：对于需要搜索的文本数据，可以进行预处理，如分词处理、去除停用词等，以减少搜索时的数据量和提高搜索效率
4. 查询语句优化：避免在 TSearch2 查询中频繁使用复杂的逻辑操作、通配符查询等，简化查询语句以提高效率
5. 定期维护：定期进行 TSearch2 索引的重建、优化，清理无用数据等维护工作，以保持全文搜索的高效性能

3.3 查询函数

PostgreSQL提供了一系列全文搜索函数，主要是to_tsvector、to_tsquery、ts_rank等。这些函数允许我们进行更复杂的文本搜索，使用更先进的搜索算法。

下面是介绍和简单演示这些函数的用法

首先，我们创建一个简单的表：

建表 SQL

```sql
CREATE TABLE documents (
  id SERIAL PRIMARY KEY,
  content TEXT
);

INSERT INTO documents (content) VALUES
('The quick brown fox jumps over the lazy dog.'),
('A quick brown dog jumps over the lazy cat.'),
('The lazy fox sleeps in the sun.');
```

1. to_tsvector: 这个函数将文本转换为一个tsvector（文本搜索向量），它是一个加权的词项列表。这个向量可以被用来执行全文搜索。它接受两个参数：一个文本搜索配置和一个文本字符串。它返回一个 tsvector，其中包含了文本的标记化和规范化信息。例如：

代码示例 SQL

```sql
SELECT content, to_tsvector('english', content) AS tsvector_content FROM documents;
```

这将返回一个包含文本中的单词及其位置的tsvector。

2. to_tsquery: 这个函数将一个查询字符串转换为一个tsquery（文本搜索查询），它是用来搜索tsvector的查询表达式。一个文本搜索配置和一个查询字符串。它返回一个 tsquery，其中包含了查询的标记化和规范化信息。例如：

代码示例 SQL

```sql
SELECT to_tsquery('english', 'quick & fox') AS tsquery;
```

这将返回一个表示查询“quick AND fox”的tsquery。

3. ts_rank: 这个函数计算文档与特定查询的相关性排名分数。它接受两个参数：一个文档的文本向量和一个查询的查询向量。它返回一个相关性排名分数，该分数反映了文档与查询的相关性。例如：

代码示例 SQL

```sql
SELECT content, ts_rank(to_tsvector('english', content), to_tsquery('english', 'quick & fox')) AS rank
FROM documents;
```

4. ts_headline: 用于生成包含匹配文本的摘要或标题，它接受两个参数：要处理的文本以及查询条件。

代码示例 SQL

```sql
SELECT ts_headline(content, to_tsquery('english', 'quick & fox')) AS headline
FROM documents;
```

5. ts_match_qv: 用于确定一个文本和查询向量是否匹配，它接受两个参数：一个文本向量和一个查询向量。

代码示例 SQL

```sql
SELECT content, ts_match_qv(to_tsvector('english', content), to_tsquery('english', 'quick & fox')) AS match
FROM documents;
```

6. ts_match_tt: 用于确定两个文本向量是否匹配，它接受两个参数：两个文本向量。

代码示例 SQL

```sql
SELECT content, ts_match_tt(to_tsvector('english', content), to_tsvector('english', 'quick brown')) AS match
FROM documents;
```

7. ts_parse: 用于解析查询文本并返回标准格式的查询向量，它接受两个参数：一个查询配置和一个查询文本。

代码示例 SQL

```sql
SELECT ts_parse('default', 'quick & brown') AS parsed_query;
```

8. ts_rank_cd: 用于计算文档的排名分数，结合文档的出现频率和与查询相关的文档频率。

代码示例 SQL

```sql
SELECT content, ts_rank_cd(to_tsvector('english', content), to_tsquery('english', 'quick & fox')) AS rank_cd
FROM documents;
```