gist索引

团队人员

- 叶超
- 叶海鹏
- 叶剑辉
- 陈振宇

索引是什么 21

gist是什么

概述

- Gist(Generalized Search Tree)，即通用搜索树。和btree一样，也是平衡的搜索树 1
- 与btree不同的是， btree索引常常用来进行例如大于、小于、等于即比较数字大小这些操作 1
- gist索引允许定义规则来将任意类型的数据分布到一个平衡的树中，并且允许定义一个方法使用此表示形式来让某些运算符访问 1

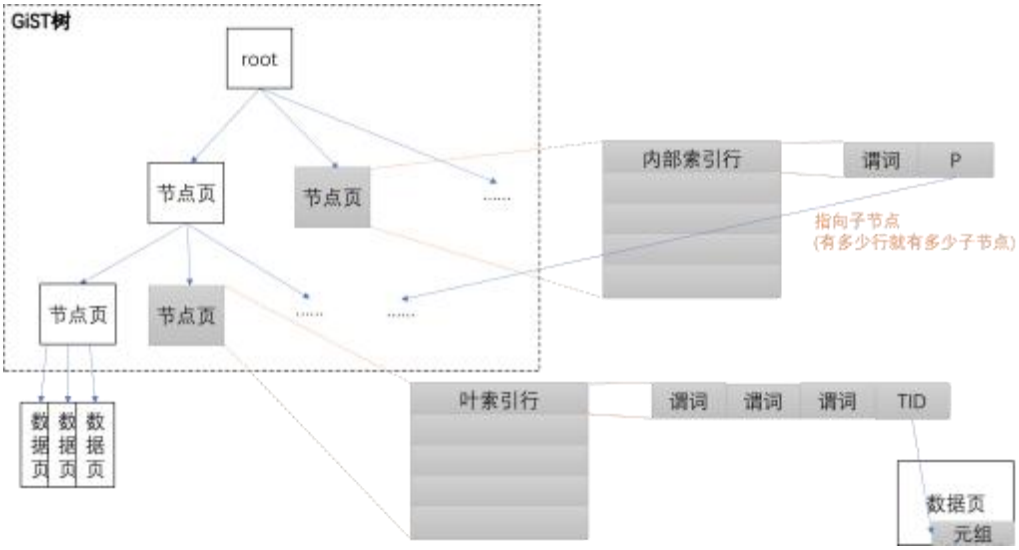
结构

GiST是一个高度平衡的树，由节点页组成。节点（非叶子节点）由索引行组成。

- 叶子节点的每一行（叶行）通常包含一些谓词（布尔表达式）和一个对表行的引用（TID）。索引的数据（键）必须满足此谓词。
  - 谓词：是一个布尔表达式，帮助GiST索引的一致性函数（consistent function）判断索引的节点是否与查询条件一致，从而决定搜索过程是否应该沿着该节点的子树继续

- TID：指向表中与索引键对应的数据行的物理位置。
  TID通常由表的ID、块的ID和元组在块中的位置组成（第几个HEAP PAGE，里面的第几条记录）

- 内部节点（非叶子节点）的每一行（内部行）包含一个谓词和对子节点的引用，子树的所有索引数据都必须满足该谓词。换句话说，内部行的谓词包含所有子行的谓词。



- gist索引的使用
  - 因为gist是一个通用的索引接口，所以可以使用GiST实现b-tree, r-tree等索引结构。
    不同的类型，支持的索引检索也各不一样
  - 几何类型，支持位置搜索（包含、相交、在上下左右等）

```
bill@bill=>create table t_gist (id int, pos point);
CREATE TABLE
bill@bill=>insert into t_gist select generate_series(1,100000), point(round((random()*1000)::r
INSERT 0 100000
bill@bill=>select * from t_gist limit 5;
 id |       pos
----+-----------------
  1 | (614.87,412.97)
  2 | (248.71,779.61)
  3 | (976.17,826.79)
  4 | (999.39,126.9)
  5 | (651.61,364.49)
(5 rows)
```

    - 在pos列上创建gist索引：
      create index idx_t_gist_1 on t_gist using gist (pos);
  - 标量类型

```
bill@bill=>create extension btree_gist ;
CREATE EXTENSION
bill@bill=>create table t_btree(id int,info text);
CREATE TABLE
bill@bill=>insert into t_btree select generate_series(1,100000),md5(random()::text);
INSERT 0 100000
```

- 创建gist索引：
  bill@bill=>create index idx_t_btree on t_btree using gist(id);

- PostgreSQL内置的几何, 标量等类型的GIST已经帮你做好了，
  要想做新增类型的索引，比如新增了一个存储人体结构的类型，怎么快速检索它们，那就要自己去实现新的GIST索引聚集算法了

- 源码解析：

```
/*
 * Strategy used to build the index. It can change between the
 * GIST_BUFFERING_* modes on the fly, but if the Sorted method is used,
 * that needs to be decided up-front and cannot be changed afterwards.
 */
typedef enum
{
        GIST_SORTED_BUILD,                      /* bottom-up build by sorting */
        GIST_BUFFERING_DISABLED,        /* in regular build mode and aren't going to
                                                                 * switch */
        GIST_BUFFERING_AUTO,            /* in regular build mode, but will switch to
                                                                 * buffering build mode if the index grows too
                                                                 * big */
        GIST_BUFFERING_STATS,           /* gathering statistics of index tuple size
                                                                 * before switching to the buffering build
                                                                 * mode */
        GIST_BUFFERING_ACTIVE           /* in buffering build mode */
} GistBuildMode;
```

这个**enum**（枚举）类型定义了一个名为**GistBuildMode**的集合，该集合包含了几种不同的**GiST**（**Generalized Search Tree**）索引构建模式。以下是每种模式的解释：

- **GIST_SORTED_BUILD (bottom-up build by sorting):**这种模式指的是通过排序堆关系（heap relation）中的元组（tuples）来构建GiST索引。这是一种自底向上的构建方法，首先将元组排序，然后根据排序的结果来构建索引树。
- **GIST_BUFFERING_DISABLED (in regular build mode and aren't going to switch):**在这种模式下，索引以常规方式构建，并且不会切换到缓冲构建模式。即使索引变得很大，它也不会改变构建策略。
- **GIST_BUFFERING_AUTO (in regular build mode, but will switch to buffering build mode if the index grows too big):**在索引构建开始时，它使用常规模式，但如果索引增长得太大，它将自动切换到缓冲构建模式。这种策略旨在平衡构建时间和内存使用。
- **GIST_BUFFERING_STATS (gathering statistics of index tuple size before switching to the buffering build mode):**在切换到缓冲构建模式之前，该模式首先收集关于索引元组大小的统计信息。这些信息可能用于确定何时以及是否应该切换到缓冲模式。
- **GIST_BUFFERING_ACTIVE (in buffering build mode)**:索引当前正在使用缓冲构建模式进行构建。在这种模式下，索引构建器可能使用额外的内存来缓存元组，并在合适的时机将它们批量插入到索引中，以减少磁盘I/O操作并提高构建性能。

这些构建模式的选择取决于多种因素，如可用内存、磁盘**I/O**性能、索引的预期大小以及构建索引所需的时间等。通过使用不同的构建模式，数据库系统可以在不同的工作负载和配置下实现更好的性能和效率。

- **Gist**索引创建过程

```c
/*
 *	gistbuildempty() -- build an empty gist index in the initialization fork
 */
void
gistbuildempty(Relation index)
{
	Buffer		buffer;

	/* Initialize the root page */
	buffer = ExtendBufferedRel(BMR_REL(index), INIT_FORKNUM, NULL,
							   EB_SKIP_EXTENSION_LOCK | EB_LOCK_FIRST);

	/* Initialize and xlog buffer */
	START_CRIT_SECTION();
	GISTInitBuffer(buffer, F_LEAF);
	MarkBufferDirty(buffer);
	log_newpage_buffer(buffer, true);
	END_CRIT_SECTION();

	/* Unlock and release the buffer */
	UnlockReleaseBuffer(buffer);
}
```

这段代码是**PostgreSQL**中用于初始化一个空的**GiST（Generalized Search Tree）**

① 函数定义：

void gistbuildempty(Relation index)

这是一个名为gistbuildempty的函数，它接受一个Relation类型的参数index，代表要初始化的索引。

② 变量定义：

Buffer buffer;

定义一个Buffer类型的变量buffer，它通常用于在内存中表示数据库中的一个磁盘块（例如，索引的一个页面）。

③ 初始化根页面：

buffer = ExtendBufferedRel(BMR_REL(index), INIT_FORKNUM, NULL,

EB_SKIP_EXTENSION_LOCK | EB_LOCK_FIRST);

* 使用`ExtendBufferedRel`函数来扩展或获取索引的根页面。

* `BMR_REL(index)`：从`Relation`对象中获取底层的关系描述符。

* `INIT_FORKNUM`：通常用于表示主索引分叉（fork）。

* `NULL`：这里没有提供特定的块号，所以函数会返回根页面或扩展索引。

* `EB_SKIP_EXTENSION_LOCK | EB_LOCK_FIRST`：这些标志控制如何锁定和扩展索引。具体来说，它们告诉系统跳过扩展锁（可能意味着索引已经是空的或正在被扩展）并锁定第一个块（即根页面）。

④ 初始化和记录日志：

START_CRIT_SECTION();

GISTInitBuffer(buffer, F_LEAF);

MarkBufferDirty(buffer);

log_newpage_buffer(buffer, true);

END_CRIT_SECTION();

* **`START_CRIT_SECTION();`** 和 **`END_CRIT_SECTION();`**：这两个宏可能用于定义一个关键部分，其中涉及对数据库结构的更改。这样的区域通常需要原子地执行，以确保数据的一致性。

* **`GISTInitBuffer(buffer, F_LEAF);`**：这个函数（或类似的函数）可能用于初始化`buffer`所指向的页面，将其标记为叶页面（尽管在根页面上下文中使用"叶"可能有些误导，因为GiST索引中的根页面通常不是叶页面）。但这里的`F_LEAF`可能是一个标志，用于指示页面的某种初始状态或属性。

* **`MarkBufferDirty(buffer);`**：将`buffer`标记为"脏"，意味着它已经被修改，并且需要在某个时间点被写回到磁盘。

* **`log_newpage_buffer(buffer, true);`**：这个函数（或类似的函数）可能用于在WAL（Write-Ahead Logging）日志中记录新页面的创建或修改。这是为了确保在系统崩溃后能够恢复数据的一致性。

⑤ 解锁和释放缓冲区：

UnlockReleaseBuffer(buffer);

使用UnlockReleaseBuffer函数来解锁并释放之前获取的buffer。这允许其他进程或事务访问或修改该页面。

总之，这段代码的主要目的是为给定的**Relation**（即索引）初始化一个空的根页面，并在需要时记录对索引的更改。

```
IndexBuildResult *
gistbuild(Relation heap, Relation index, IndexInfo *indexInfo)
{
        IndexBuildResult *result;
        double          reltuples;
        GISTBuildState buildstate;
        MemoryContext oldcxt = CurrentMemoryContext;
        int                     fillfactor;
        Oid                     SortSupportFnOids[INDEX_MAX_KEYS];
        GiSTOptions *options = (GiSTOptions *) index->rd_options;

        /*
         * We expect to be called exactly once for any index relation. If that's
         * not the case, big trouble's what we have.
         */
        if (RelationGetNumberOfBlocks(index) != 0)
                elog(ERROR, "index \"%s\" already contains data",
                         RelationGetRelationName(index));

        buildstate.indexrel = index;
        buildstate.heaprel = heap;
        buildstate.sortstate = NULL;
        buildstate.giststate = initGISTstate(index);

        /*
         * Create a temporary memory context that is reset once for each tuple
         * processed.  (Note: we don't bother to make this a child of the
         * giststate's scanCxt, so we have to delete it separately at the end.)
         */
        buildstate.giststate->tempCxt = createTempGistContext();
```

这个函数 **gistbuild** 是用来构建（或初始化）一个 **GiST**（**Generalized Search Tree**）索引的。**GiST** 是一种通用的搜索树结构，它允许用户定义自己的搜索方法和一致性检查函数，以支持各种复杂的数据类型和查询。下面是对这个函数的详细解释：

1) 参数:

heap: 关联的堆关系（表）。索引通常基于堆关系上的数据进行构建。

index: 要构建的 GiST 索引关系。

indexInfo: 索引构建时所需的其他信息，如填充因子等。

2) 初始化:

初始化了一个 GISTBuildState 结构体 buildstate，它包含了构建索引所需的状态信息。

获取了与索引关联的 GiSTOptions（如果有的话），这些选项可能包含关于索引构建的特殊指令。

确保索引是空的。如果它已经包含数据，函数将报错并终止。

3) 构建准备:

设置了 buildstate 的各个字段，包括索引关系、堆关系、排序状态（此处为 NULL）和 GiST 状态（通过 initGISTstate 初始化）。

创建了一个临时内存上下文 tempCxt，该上下文将在处理每个元组时重置。这是为了确保在处理大量数据时，内存使用不会持续增长。

4) 选择构建策略:

检查 GiSTOptions 中的 buffering_mode 选项。这个选项决定了索引构建的缓冲模式。如果设置为 GIST_OPTION_BUFFERING_ON，则使用统计缓冲模式（GIST_BUFFERING_STATS）。这种模式可能是为了测试或特定场景而设计的，允许索引构建时进行一些缓冲或优化。

如果设置为 GIST_OPTION_BUFFERING_OFF，则禁用缓冲（GIST_BUFFERING_DISABLED）。这意味着索引将按照传统的、无缓冲的方式进行构建。

如果 buffering_mode 没有明确设置（即“auto”），则函数可能会根据其他参数或默认设置选择适当的构建策略。

注意：代码在检查 buffering_mode 后被截断了，所以无法看到后续的构建逻辑。但通常，在选择了构建策略后，函数会遍历堆关系中的所有元组，将它们插入到 GiST 索引中，并根据所选的策略进行缓冲或优化。

最后，函数将返回一个 IndexBuildResult 指针，该指针可能包含有关索引构建结果的信息，如使用的空间量、插入的元组数等。然而，这个具体的返回值及其内容取决于实际的实现和调用者的需求。

- **Gist索引项排序**

```c
/*
 * Pairing heap comparison function for the GISTSearchItem queue
 */
static int
pairingheap_GISTSearchItem_cmp(const pairingheap_node *a, const pairingheap_node *b, void *arg)
{
	const GISTSearchItem *sa = (const GISTSearchItem *) a;
	const GISTSearchItem *sb = (const GISTSearchItem *) b;
	IndexScanDesc scan = (IndexScanDesc) arg;
	int			i;

	/* Order according to distance comparison */
	for (i = 0; i < scan->numberOfOrderBys; i++)
	{
		if (sa->distances[i].isnull)
		{
			if (!sb->distances[i].isnull)
				return -1;
		}
		else if (sb->distances[i].isnull)
		{
			return 1;
		}
		else
		{
			int			cmp = -float8_cmp_internal(sa->distances[i].value,
												   sb->distances[i].value);

			if (cmp != 0)
				return cmp;
		}
```

这个函数似乎是一个用于比较两个GISTSearchItem结构体的比较函数，它通常用于排序或作为某种搜索算法（如堆排序或优先队列）的一部分。下面是对这个函数的详细解释：

1. **参数：**

a 和 b：要比较的两个GISTSearchItem指针，被强制转换为const GISTSearchItem *类型。

arg：一个IndexScanDesc类型的指针，它可能包含了一些与排序或搜索相关的参数，但在这个函数中只使用了它的numberOfOrderBys字段。

2. **变量：**

i：用于循环的计数器。

scan：从arg转换而来的IndexScanDesc变量，它可能包含了一些与索引扫描相关的参数。

3. **排序逻辑：**

函数首先根据scan->numberOfOrderBys指定的数量来比较sa和sb中的distances数组。这个数组可能包含了一些距离值，用于某种基于距离的排序。

对于sa和sb中的每一个距离值，如果其中一个为NULL（或表示为空，这里使用了isnull字段），那么函数会返回-1或1，这取决于哪个为NULL。

如果两个距离值都不为NULL，那么使用float8_cmp_internal函数来比较它们的值，并取反结果。这意味着较小的距离值会排在前面（因为-float8_cmp_internal会将较小的值视为“较大”的排序键）。

如果比较结果为非零，则立即返回该结果。

4. **堆项与内部页的逻辑：**

在比较了所有距离值之后，函数接着检查sa和sb是否代表堆项（可能是索引中的叶子节点）。

如果sa是堆项而sb不是，则返回1，表示在排序中sa应该排在sb前面。

如果sa不是堆项而sb是，则返回-1，表示sb应该排在sa前面。

这可能是为了确保在搜索时首先访问堆项（即叶子节点），以实现深度优先搜索。

5. **默认返回值：**

如果上述所有比较都没有返回非零值，那么函数返回0，表示sa和sb在排序上是相等的。

这个函数可能用于一个需要排序GISTSearchItem的上下文中，例如在一个基于距离的搜索或查询优化中。

- Gist索引扫描

```
IndexScanDesc
gistbeginscan(Relation r, int nkeys, int norderbys)
{
        IndexScanDesc scan;
        GISTSTATE  *giststate;
        GISTScanOpaque so;
        MemoryContext oldCxt;

        scan = RelationGetIndexScan(r, nkeys, norderbys);

        /* First, set up a GISTSTATE with a scan-lifespan memory context */
        giststate = initGISTstate(scan->indexRelation);

        /*
         * Everything made below is in the scanCxt, or is a child of the scanCxt,
         * so it'll all go away automatically in gistendscan.
         */
        oldCxt = MemoryContextSwitchTo(giststate->scanCxt);

        /* initialize opaque data */
        so = (GISTScanOpaque) palloc0(sizeof(GISTScanOpaqueData));
        so->giststate = giststate;
        giststate->tempCxt = createTempGistContext();
        so->queue = NULL;
        so->queueCxt = giststate->scanCxt;      /* see gistrescan */

        /* workspaces with size dependent on numberOfOrderBys: */
        so->distances = palloc(sizeof(so->distances[0]) * scan->numberOfOrderBys);
        so->qual_ok = true;                     /* in case there are zero keys */
        if (scan->numberOfOrderBys > 0)
```

这个函数gistbeginscan是一个初始化GiST（Generalized Search Tree）索引扫描的函数。GiST是一种支持多种数据类型的平衡树索引结构，常用于空间数据等。下面是该函数的详细解释：

(1) 参数:

r: 表示要扫描的关系（表）的Relation对象。
nkeys: 索引扫描中使用的键的数量。
norderbys: 用于排序的字段数量（通常用于ORDER BY子句）。

(2) 局部变量:

scan: 用于存储初始化后的IndexScanDesc对象，它描述了索引扫描的状态。
giststate: 表示GiST索引状态的GISTSTATE对象。
so: 指向GISTScanOpaque对象的指针，它包含与GiST扫描相关的特定信息。
oldCxt: 保存当前内存上下文，以便稍后恢复。

(3) 主要逻辑:

a. 初始化IndexScanDesc:
* 使用`RelationGetIndexScan`函数根据关系、键的数量和排序字段的数量初始化一个`IndexScanDesc`对象。

b. 设置GISTSTATE:
* 调用`initGISTstate`函数来初始化一个与给定索引关系关联的`GISTSTATE`对象。这个对象存储了GiST索引扫描所需的状态信息。

c. 切换到扫描内存上下文:
* 使用`MemoryContextSwitchTo`函数切换到`giststate->scanCxt`内存上下文。这个上下文用于存储扫描期间分配的所有对象，确保在扫描结束时自动释放它们。

d. 初始化GISTScanOpaque:
* 分配并初始化一个`GISTScanOpaque`对象（这里简称为`so`），它包含与GiST扫描相关的特定信息。
* 分配并初始化`so->distances`数组，用于存储与排序字段相关的距离值。
* 初始化其他相关字段，如`so->qual_ok`、`so->killedItems`、`so->numKilled`、`so->curBlkno`和`so->curPageLSN`。

e. 设置IndexScanDesc的opaque字段:
* 将`so`赋值给`scan->opaque`，以便在扫描过程中可以访问这些特定于GiST的信息。

f. 恢复内存上下文:
* 使用`MemoryContextSwitchTo`函数恢复之前保存的内存上下文。

g. 返回:
* 返回初始化后的`IndexScanDesc`对象。

(4) 注意:

* 该函数仅初始化扫描状态，而不实际执行任何扫描操作。真正的扫描操作通常在后续的函数（如`gistrescan`）中执行。
* 索引扫描中使用的内存上下文确保了扫描过程中分配的所有资源在扫描结束时都会自动释放，从而避免了资源泄漏。

```
void
gistrescan(IndexScanDesc scan, ScanKey key, int nkeys,
                   ScanKey orderbys, int norderbys)
{
        /* nkeys and norderbys arguments are ignored */
        GISTScanOpaque so = (GISTScanOpaque) scan->opaque;
        bool            first_time;
        int                     i;
        MemoryContext oldCxt;

        /* rescan an existing indexscan --- reset state */

        /*
         * The first time through, we create the search queue in the scanCxt.
         * Subsequent times through, we create the queue in a separate queueCxt,
         * which is created on the second call and reset on later calls.  Thus, in
         * the common case where a scan is only rescan'd once, we just put the
         * queue in scanCxt and don't pay the overhead of making a second memory
         * context.  If we do rescan more than once, the first queue is just left
         * for dead until end of scan; this small wastage seems worth the savings
         * in the common case.
         */
        if (so->queue == NULL)
        {
                /* first time through */
                Assert(so->queueCxt == so->giststate->scanCxt);
                first_time = true;
        }
        else if (so->queueCxt == so->giststate->scanCxt)
        {
                /* second time through */
                so->queueCxt = AllocSetContextCreate(so->giststate->scanCxt,
                                                                                        "GiST queue context",
```

这个函数gistrescan是用于重新扫描一个已经存在的GiST索引扫描的。在数据库查询执行过程中，当查询的WHERE子句或ORDER BY子句发生变化时，索引扫描可能需要被重新配置以反映这些变化，而不是重新启动一个新的扫描。这就是gistrescan函数的作用。

下面是对函数gistrescan的详细解释：

一、参数:scan:

指向IndexScanDesc的指针，它包含了关于索引扫描的当前状态。

key: 指向ScanKey数组的指针，描述了索引扫描的搜索条件（但在这个函数中，这个参数被忽略了）。

nkeys: 搜索条件中键的数量（也被忽略了）。

orderbys: 指向ScanKey数组的指针，描述了用于排序的字段（同样，这个参数也被忽略了）。

norderbys: 用于排序的字段数量（也被忽略了）。

二、局部变量:

so: 指向GISTScanOpaque的指针，该结构体包含了与GiST扫描相关的特定信息。

first_time: 一个布尔变量，用于标识这是否是第一次重新扫描。

i: 一个整数变量，用于循环（但在给出的代码段中并未使用）。

oldCxt: 保存当前内存上下文，以便稍后恢复。

三、主要逻辑:

重新扫描现有索引扫描: 函数的主要目的是重置扫描的状态，以便它可以基于新的搜索条件或排序要求重新开始。

四、内存上下文处理:

如果so->queue（即搜索队列）是NULL，这意味着这是第一次重新扫描。在这种情况下，它使用扫描的默认内存上下文（so->giststate->scanCxt）来创建搜索队列。

如果so->queue不是NULL但so->queueCxt与so->giststate->scanCxt相同，这意味着这是第二次重新扫描。在这种情况下，它会在扫描的默认内存上下文之外创建一个新的内存上下文（GiST queue context），用于存储搜索队列。这是为了优化性能，因为在大多数情况下，索引扫描只会被重新扫描一次。如果扫描被重新扫描多次，那么除了第一次之外的搜索队列都会被保留在单独的内存上下文中，直到扫描结束。

解释代码段:在给出的代码段中，只展示了如何根据重新扫描的次数设置搜索队列的内存上下文。如果这是第一次重新扫描，它将使用默认的扫描内存上下文。如果是第二次或更多次重新扫描，它将创建一个新的内存上下文来存储搜索队列。

注意:函数中提到了多个Assert和palloc（或类似的内存分配函数）的调用，但在给出的代码段中并未显示。这些调用通常用于确保状态的有效性以及分配必要的内存。

实际的搜索逻辑和重新应用搜索条件或排序要求的代码在函数的其他部分中，而这些部分并未在给出的代码段中显示。

```
void
gistendscan(IndexScanDesc scan)
{
        GISTScanOpaque so = (GISTScanOpaque) scan->opaque;

        /*
         * freeGISTstate is enough to clean up everything made by gistbeginscan,
         * as well as the queueCxt if there is a separate context for it.
         */
        freeGISTstate(so->giststate);
}
```

这个函数 gistendscan 是用于结束一个 GiST 索引扫描的。当查询执行完毕，或者需要停止当前索引扫描并释放相关资源时，会调用这个函数。以下是该函数的详细解释：

A. 函数参数

scan: 指向 IndexScanDesc 的指针，它包含了关于索引扫描的当前状态。

B. 函数内部逻辑

获取 GiST 扫描的私有数据：GISTScanOpaque so = (GISTScanOpaque) scan->opaque;
通过 scan->opaque 字段，函数获取到与当前索引扫描相关的私有数据 GISTScanOpaque 结构体指针。这个结构体通常包含了执行 GiST 索引扫描所需的所有私有信息。

C. 释放 GiST 状态：

freeGISTstate(so->giststate); 调用 freeGISTstate 函数来释放与当前 GiST 索引扫描相关联的所有资源。这个函数会清理 gistbeginscan 函数中创建的所有东西，包括搜索队列（如果存在）、内存上下文（如果为搜索队列创建了单独的上下文），以及其他任何与 GiST 索引扫描相关的资源。
注意事项

D. 内存管理：

在数据库系统中，内存管理是非常重要的。确保及时释放不再使用的内存可以避免内存泄漏，这对于系统的长期稳定运行至关重要。

E. 错误处理：

虽然这个函数没有显示错误处理逻辑，但在实际系统中，错误处理通常是必要的。例如，如果 so 或 so->giststate 为 NULL，或者 freeGISTstate 函数调用失败，那么应该有一些错误处理机制来记录问题或采取适当的恢复措施。

F. 上下文切换：

由于这个函数可能会释放与当前扫描相关的内存上下文，因此在调用这个函数之后，任何尝试访问这些资源的代码都应该被避免，因为它们可能已经不再有效。
总结
gistendscan 函数用于结束一个 GiST 索引扫描并释放与之相关的所有资源。通过调用 freeGISTstate 函数，它确保了所有由 gistbeginscan 创建的资源都被正确地清理，从而避免了内存泄漏和其他潜在问题。