Intra-page Indexing in Generalized Search Trees of PostgreSQL

Andrey Borodin[1][0000−0002−1231−7959], Sergey Mirvoda[2][0000−0002−4615−7164], and Sergey Porshnev[2,3][0000−0001−6884−9033]

[1] Yandex, Khokhryakova str. 10, Yekaterinburg, Russia
amborodin@acm.org
[2] Ural Federal University, Mira str. 19,Yekaterinburg, Russia
{s.g.mirvoda,s.v.porshnev}@urfu.ru
[3] N.N. Krasovskii Institute of Mathematics and Mechanics of the Ural Branch of the Russian Academy of Sciences, S. Kovalevskaja str. 16, Yekaterinburg, Russia

Abstract. The Generalized Search Tree (GiST) is a framework for creating balanced tree access methods for data types, which can be provided as a database extension. This framework offers a big part of the access method's code but places some algorithmic limitations. One of these limitations is the concept that one tree node is a single page. In this paper, we propose changes to this limitation with additional intra-page indexing, based on the concept of skip tuples. This approach allows to increase of insert and update performance by the factor of 1.5 and opens new ways towards GiST API advancement. We implemented the proposed approach as a PostgreSQL core patch.

Keywords: Database · Indexing · GiST · Performance · PostgreSQL.

1 Introduction

PostgreSQL is one of the most advanced open source relational databases. And one of the prominent features of PostgreSQL is extensibility, the database is designed to be "hackable". PostgreSQL allows hackers to implement functions, describe data types, implement joins, hook internals and change functionality and features. Additionally, PostgreSQL has some levels of generalization to avoid writing boilerplate code. One of such parts is GiST.

Generalized index search tree (GiST) is an access method (AM) technique, which allows to abstract significant parts of data access methods structured as a balanced tree. Use of the GiST allows AM developer to concentrate on his own case-specific details of AM and skip common work on the tree structure implementation within the database engine, a query language integration, a query planner support, concurrency, recovery, etc.

The GiST was first proposed by J. Hellerstein in [12], further researches were undertaken by M. Kornacker [14, 13]. Later GiST was implemented in PostgreSQL with a large contribution by O. Bartunov and T. Sigaev [9]. Current

PostgreSQL GiST implementation accepts different trees as a datatype (and so-called operator class or opclass which identifies the operators to be used for the given index type). Opclass developer must specify 4 core operations to make a type GiST-indexable:

1. Split: a function to split a set of datatype instances into two parts.
2. Penalty calculation: a function to measure the penalty for the unification of two keys.
3. Collision check: a function that determines whether two keys may have overlap or are not intersecting.
4. Unification: a function to combine two keys into one so that combined key collides with both input keys.

Operations 1, 2 and 4 are responsible for the construction of the index tree, while operation 3 is used to query the constructed tree. In general case, tree construction can be seen as the serial insertion of a dataset into an index tree. Insertion is a recursive algorithm, executed for every node starting from the root of the tree. This algorithm searches within a node for an entry (also called downlink) with a minimal penalty of insertion of an item being inserted. For a chosen downlink the key is updated with operation 4, the algorithm is invoked recursively. If the algorithm is invoked on a leaf page it just places the item being inserted, if the node is overflown then the upward sequence of splits with operation 1 is started.

In terms of PostgreSQL operation 3 is called "consistency check" since it allows many different search strategies (intersection, inclusion, exclusion, adjacency, etc.). Also, PostgreSQL GiST requires one more operation – equality comparison called "same function". The data type can specify three optional operations: compress\decompress for compacting storage and distance for generalized kNN searches.

For example, if for operations 1–4 we pick rectilinear rectangles, we get regular R-tree [11], though many different indexing schemes are possible.

There are some differences between the original GiST and PostgreSQL implementation. For example, PostgreSQL implementation does not use stack recursion in algorithms and allows us to use in one index multiple different data types with different opclasses.

Currently, GiST is used by many geoinformation systems due to the PostGIS extension for PostgreSQL. PostGIS provides a versatile toolbox for map applications and other GIS functions. PostGIS uses GiST as the main indexing engine. The GiST is used in astronomic databases via pg_sphere extension. This extension is used to search for new stars by comparing observed signals using spatial join, implemented in GiST. Also, GiST is used for full-text search, for search in a set of ranges, for image similarity search and so on.

The PostgreSQL GiST implementation assumes one node of the generalized tree is a single page. It allows manipulating data that is larger than RAM: while some pages reside in RAM buffer, others reside in persistent memory. Our team is working on different improvements for spatial indexing [8]. Research of the GiST

implementation showed us that there is a room for insert\update performance improvements by rethinking "one node is one page" concept.

In this paper we will review current GiST implementation concepts, this review will cover two main GiST procedures: inserts and scans. We do not consider tuple deletion since it is irrelevant in current PostgreSQL MVCC implementation. After this review, we will propose algorithmic and structural improvements and describe necessary code changes. The 3rd section will be devoted to a technique called skip tuples, which allows faster inserts into an index and skipping of tuple groups during scans. Also, we will describe motivation and details of Advanced Generalized Search – a framework for a showcase of GiST advances which can be deployed by current PostgreSQL users in their databases and is production ready. Then we will cover proposed changes with relevant experiments and their analysis. In this section, we also choose parameters necessary to tune skip tuples technique. The next section will cover the limitation of current algorithms and implementations. Later we will discuss related work focused on tackling similar problems with different approaches by other teams.

2 Current Implementation

PostgreSQL typically uses 8 Kb pages. The concept "one node is one page" means that fan-out of index tree typically varies between 100 and 1000 (from 80 to 8 bytes per tuple) with practically observable fan-outs around 250. This rough estimate shows that the GiST tree is usually low and guarantees path from the root to leaf in few disk reads. But in practical usage scenarios, GiST performance is not constrained by block device throughput. But the CPU operations often are the bottleneck.

The PostgreSQL GiST has two major parts of the functionality, dependent on page layout: index construction and index scan. In turn, index construction consists of insertion into an index and buffered build for an existing table. This work is focused on the insertion part. Despite buffered insertion also benefits from described technique and code changes, it's performance rarely is a bottleneck. Due to PostgreSQL MVCC implementation GiST updates are also represented by GiST inserts. Index scan also contains two interleaving parts: generalized search (regular scan) and generalized kNN ordering. In this work, we focused on a regular index scan. GiST concurrency is based on page-level latches and is not affected by this work. Recovery in GiST is updated, but its changes are straightforward and will not be discussed in detail.

2.1 Index Insertion

The insertion algorithm for any given index tuple searches the suited leaf page by descending from root page to leaf. Each step on the internal page invokes penalty calculation for each tuple on the page to find the best fitting subtree.

The only case when the insertion algorithm does not need to deal with every tuple is zero-penalty tuples (for a given inserted tuple) in the middle of a search.

But the existence of many zero-penalty tuples means overlapping of keyspace and inefficient index from the search point of view.

Overflown pages are divided into many parts by the so-called split algorithm. Most of the split-algorithms have algorithmic complexities are at least $\Omega(n)$.

2.2 Index Scan

The GiST scan maintains a stack of pages that may contain tuples, satisfying query conditions. Initially, this stack contains only a root page. During a scan, each internal page on top of the stack is replaced by referenced child pages with index tuples relevant to the query search condition. If the top of the stack contains a leaf page – each tuple of this leaf page is examined on matching query search condition and, if passed, outputted as the scan result. The index scan is complete when the stack is empty.

This algorithm ensures the depth-first scanning order of the index in a regular scan. But GiST also supports the k-nearest neighbor (kNN) index scan. kNN type of scan ensures that tree paths are not searched when it is known that they cannot contain tuples with a distance less than that have already been found by the scan. In a regular scan and in a kNN scan the inner page is deconstructed into its child references by $O(n)$ algorithm: checking all contents against search conditions (consistency operation), where n is the fan-out number of the page.

2.3 Index Scan Thought Experiment

GiST itself has no usable performance prediction model. But there are performance prediction models for indexes implementable over GiST. For example, there is a quite accurate cost model for R-Tree [16]. But this model has some limitations. First, the output of this model is accurate for "optimum, not implemented yet method" to build an R-tree. But R-tree-over-GiST is far from optimum. Second, this model outputs the number of disk accesses as a function of data properties. But actual index performance is itself a function of a number of disk accesses. When the index fits into main memory, disk accesses are not a bottleneck.

We can apply some theoretical reasoning to estimate index scan performance in terms of key collision checks (consistency function calls). We observed that these calls dominate in the CPU profile during the execution of lasting GiST index scans. If we have an index with N leaf tuples and each GiST node has a fan-out factor f, the height h of the tree will be $h \approx log_f N$. If the scan will find exactly 1 tuple within a tree without overlapping subtree keyspace, number of calls to consistency function CN will be

$$CN \approx f, \quad h \approx f \log_f N. \tag{1}$$

Here the approximation of CN is minimal when $f = e$, where e is the basis of the natural logarithm. And e is far smaller than usual $f \approx 250$.

Of cause, this reasoning has limitations. First, we assumed that on each level GiST scan did not encounter keyspace overlap of subtrees. Access methods are designed with overlap minimization as a goal [2]. But it is not always achievable. Second, single cache-missed disk access will dominate thousands of collision check calls. Multiple cache-missed disk accesses will render CN optimization useless.

In-memory GiST insertion, according to our observations, is dominated by penalty function calls. In previous work [8] we were focusing on enhancing the penalty function of *cube* and PostGIS extensions to achieve better index properties during the spatial search. Reasoning about CN can be applied to a number of *penalty* function calls intact: lower fan-out should yield fewer *penalty* calls.

3 Proposed Changes

3.1 Initial Design Considerations

How to reduce tree node fan-out? Initially, we proposed multi-level intra-page tree at PostgreSQL hackers mailing list [5].

While we can't fill a page with just 3 tuples, we can install a small tree-like structure inside one page. General GiST index has a root page. But a page tree should have a "root" layer of tuples. Let's consider the concept of private tuples (or internal, intermediate, auxiliary, we have to distinguish them from initial internal\leaf dichotomy) without links to other pages. These private tuples could have only keys and a fixed-size array of underlying records offsets (with size f). Each layer is a linked-list. After the page has just been allocated there is only "ground" level of regular tuples. Eventually, record count reaches $f-1$ and we create a new root layer with two private tuples. Each new tuple references half of the preexisting records. Placement of new "ground" tuples on the page eventually will cause private tuple to split. If there is not enough space to split private tuple, we mark the page for the whole page-split during the next iteration of the insertion algorithms of owning the GiST tree. That is why tuple-split happens on $f-1$ tuples, not on f: if we have no space for splitting, we just adding a reference to the last slot. In this algorithm, page split will cause major page defragmentation: we take the root layer, halve it and place halves on different pages. When half of a data is gone to another page, restructuration should tend to place records in such a fashion that accessed together tuples are placed together. Let's look how page grows with fan-out factor $f = 5$.

When we added 3 ground tuples it's just a ground layer, here `RLS` is root layer start, `G` is ground tuple, $\texttt{I}_\texttt{x}$ is internal tuple of level x:

`RLS=0|G G G`, then we place one more tuples and layer splits:

`RLS=4|G G G G` $\texttt{I}_0$ $\texttt{I}_0$, each $\texttt{I}_0$ tuple now references two `G` tuples.

We keep placing `G` tuples:

`RLS=4|G G G G` $\texttt{I}_0$ $\texttt{I}_0$ `G G`, and then one of $\texttt{I}_0$ tuples is splitted:

`RLS=4|G G G G` $\texttt{I}_0$ $\texttt{I}_0$ `G G G` $\texttt{I}_0$, one more $\texttt{I}_0$ split causes new layer:

`RLS=12|G G G G` $\texttt{I}_0$ $\texttt{I}_0$ `G G G` $\texttt{I}_0$`G` $\texttt{I}_0$ $\texttt{I}_1$ $\texttt{I}_1$.

This structure could provide average tree fan-out as low as desired. An analysis of a similar approach is given in [10]. But this approach is too sophisticated

for industrial implementation within PostgreSQL codebase with a steep learning curve. This algorithm would require *pg_upgrade* with the previous version of GiST, intervene into the physical structure of index tuple. Recovery is affected by this structure too: there are new possible valid states in case of a crash, that has to be handled during recovery from WAL. After the discussion in pgsql-hackers list and few technical seminars, we decided to give up this structure in favor of a more simple and maintainable design.

3.2 Simplified Intra-page Indexing

To simplify intra-page indexing we decided to use a two-level indexing scheme instead of multilevel. Also, one of the design goals was to exclude the necessity of *pg_upgrade* of old GiST indexes to a newer version.

To achieve these goals, we decided to introduce the concept of skip tuples. The skip tuple is the tuple, which allows skipping the next few tuples if the key of skip tuple indicates that the following group of tuples is of no interest to a given algorithm.

This approach relies on tuple ordering on the page. The GiST in PostgreSQL 9.6 tends to shuffle records for the sake of code simplicity. We have fixed this [3, 7] in PostgreSQL 10, introducing routines for tuple overwrite. This advancement allowed to gain about 15% of insertion performance for the price of more complex code. But what was really important is that now GiST could sustain tuple ordering and in future versions of GiST, we could afford to rely on this order to implement a skip tuples approach.

Regular GiST tuples always have reference to a page. While tuples on internal pages have references to other pages of the same GiST index, tuple on leaf page has reference to pages in a heap. Skip tuples do not have reference to any other page, and we are using a reference part of the tuple structure to store count of tuples, united by a key of given skip tuple. We've found a spare bit in the tuple header structure and used it to indicate that given tuple is the skip tuple. Thus, we did not change any bit of code responsible for tuple accommodation on the page (as of PostgreSQL 11 development codebase).

3.3 Usage of Skip Tuples

When do the skip tuples appear? Initially, the GiST index is placed on one leaf page which is the root page. New tuples are simply appended at the end of the page. Obviously, at this moment intra-page indexing is not necessary: there is no room for a sophisticated algorithm to gain significant performance difference on 8 Kb of the data.

When the page is overflown, it is split and a new root is formed. GiST does not always split the page into 2 halves but can have up to 75 parts (arbitrary number chosen by GiST developers as a sane upper limit). For each new page GiST forms downlinks from the root page. These downlinks are placed into one skip group of the skip tuple. This defines a moment when the first skip tuple appears: when the root is first split by the codebase with support of skip tuples.

At this exact moment, we have to decide on one more tradeoff: if we place skip tuples on a leaf page, we favor faster scans, if we do not we favor faster inserts. From our experience, most of GiST use cases encounter an insufficient performance of GiST inserts and update. We decided to work on inserts-oriented algorithms. This is a subjective decision influenced mostly by GIS users. But proposed algorithms can be adjusted for scan-oriented without compatibility issues. Also, a multilevel structure can be added over two-level too.

Each time when the page is split new downlink must be formed (see Fig. 1). This downlink is possibly added to some skip group; thus, this group may overgrow some limit, we call this limit *skip group threshold* T. In case of this overflow, skip group is split with the regular split algorithm, which is already provided by data type for page split. This algorithm outputs some new skip groups, which replace the overflown skip group.

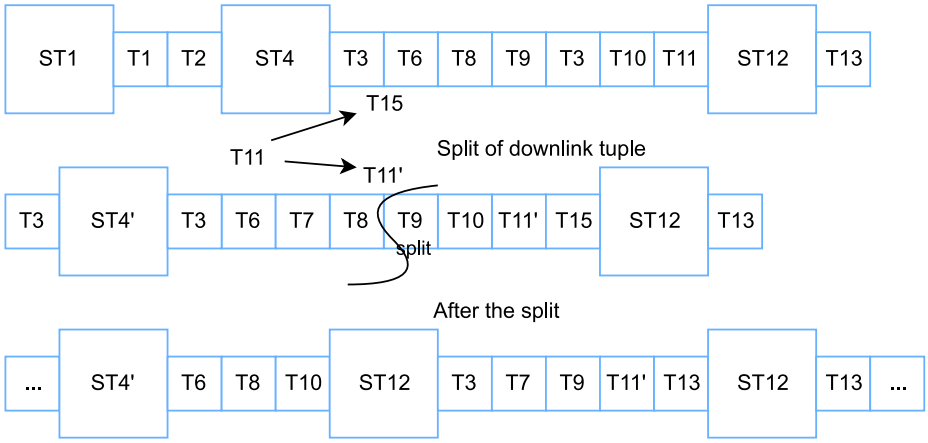

Fig. 1. Split of the skip group.

When the page with skip groups is overflown and has to be split, we pick vector of skip tuples from a page, split it, and distribute skip groups to new pages according to skip tuples split. On a rare occasion, this may produce skip group vectors, which do not fit a single page. Then we fall back to split of regular tuples from the same page.

When GiST is choosing subtree for insertion, it must pick downlink with minimal *penalty* value for inserting a given item (new index tuple). Since the penalty is the measure of "how much key space of subtree will be extended in case of insertion", we suppose that for given item *penalty* of skip tuple is always no greater than for any tuple inside its skip group. We checked that this assumption holds true for *penalty* function bundled by all extensions in *contrib* directory of PostgreSQL source code and *penalty* functions in PostGIS. But we have no strict proof that this assumption holds everywhere since GiST does not

apply enough restrictions on *penalty* function. For penalty function P, union function U, new entry e, items on page $i_1 \dots i_n$ we assume:

$$P(e, U(i_1 \dots i_n)) \leq P(e, i_x) \quad \forall x. \tag{2}$$

This inequality allows skipping the skip group during choose subtree algorithm if the penalty of its skip tuple is greater than already found. Most of the insertion performance improvement comes from this change. But that's not the only use of skip tuples.

During the execution of the scan, if the search conditions do not collide with a key of skip tuple, we can skip the whole skip group. Our theoretic analysis intentionally skips algorithms for deleting tuples from the index. PostgreSQL MVCC implementation prescribes that there are no routines to delete a single tuple from the access method, just scheduled vacuum – bulk deletion process, which is not affected by skip tuples directly.

4 Experimental Analisys

We have implemented proposed changes as a patch and published the patch on pgsql-hackers mailing list [6]. The patch is fully functional, passes all available regression and stress tests. We are going to work on inclusion it to mainstream PostgreSQL. Bug reports and experience feedback will be appreciated.

All conducted tests were single-threaded, conducted on a machine with Intel Core i7 (I7-4770HQ), 1600 MHz DDR3 SDRAM. PostgreSQL memory setting *shared_buffers* were configured to guaranty that all test data reside in RAM. The database cluster was completely wiped before each test. We had chosen a built-in data type *point* as the most basic and suitable for benchmarking. The *point* type is 16 bytes wide, represents a point on a plane. For indexing, it uses increasing of covered size by minimum bounding box as a penalty function and Korotkov split algorithm [15], which can be considered as one of the most advanced to date and is also used in PostGIS extension.

All test scripts are published on GitHub [4], along with bash scripts to run benchmarks for rapid results reproduction.

4.1 Tests with randomized data

We used following script to generate dataset and benchmark GiST insertion:

```
CREATE UNLOGGED TABLE x(c point);
CREATE INDEX ON x USING gist(c);
INSERT INTO x SELECT point(random(), random()) c
FROM generate_series(1,10000000) y;
VACUUM.
```

This script creates table textitx, which contains only one column textitc of a *point* type. Then the script creates a GiST index on this column. After that, the

table is filled with ten million points, both coordinates are uniformly distributed between 0 and 1. We measured the time of insertion by *psql* metacommand \timing, which obtains time from the database server. Time of the table and index creation and VACUUM does not affect the time of insertion. It is worth noting that during data insertion the heap is also populated and this affects insertion time, to minimize this influence we used UNLOGGED table, but results for WAL-logged tables do not differ significantly.

We used following script to benchmark index scan time:

```
SET enable_bitmapscan = off;
EXPLAIN ANALYZE
WITH pts AS (SELECT random() x, random() y
FROM generate_series(1, 100000) y),
QUERIES AS
(SELECT box(point(x,y), point(x+0.01, y+0.01)) b FROM pts)
SELECT (SELECT count(*) FROM x WHERE x.c <@ q.b) FROM queries q.
```

This script generates one hundred thousand of points with coordinates uniformly distributed between 0 and 1, creates boxes from these points with edge 0.1, and for each box counts a number of data points within the box. Each box is expected to contain slightly less than one thousand data points on average. EXPLAIN ANALYZE is appended to control the execution plan during benchmarks.

We found that on default cluster configuration with source code from the master branch (git branch for bleeding edge version) insertion takes on average 123 seconds, while with intra-page indexing with T = 16 same task takes 81 seconds. This constitutes a 34.5% performance improvement. At the same time on master selection task takes 35.13 seconds, while with patch the task takes on average 32.87 seconds, 6.5% improvement.

The task of the insertion of randomized data is further referenced as RI (random inserts), the task of scanning index for counts computation is referenced as RS (random selects).

4.2 Tests with Ordered Data

To build an efficient index, GiST relies on *penalty* and *split* functions. Given that assumption about a penalty of skip group holds, it is provable that the use of penalty function is unaffected by skip group mechanics, besides the fact, that skip tuples occupy space on a page and allow to store fewer tuples. But it is a known fact that penalty function may degrade if data is provided to GiST in sorted order [7]. There are some techniques to mitigate this problem, but it allows us to construct the "worst case" for skip tuple technique.

In this case data is generated and inserted by script:

```
CREATE UNLOGGED TABLE x(c point);
CREATE INDEX ON x USING gist(c);
INSERT INTO x SELECT point(x / 1000.0, y / 10000.0) c
FROM generate_series(1, 1000) y, generate_series(1, 10000) x.
```

This script uses the same table and index as RI but inserts Cartesian product of two evenly increasing series of numbers between 0 and 1. The task of this data insertion is further references as OI (ordered inserts). To test searches, in this case, we used the same script as in RS, but on OI data, this task is called OS (ordered-data selects).

We observe that on master OI task takes 127 seconds, while with patch and T = 16 it takes 88 seconds, which is 30% improvement. In turn, OS on master takes 22 seconds, while with the intra-page index it takes 31 seconds, which is 39% degraded. Obviously, this degrade can be mitigated by *penalty* function enhancement, but this constitutes that intra-page indexing cannot rely solely on *split* algorithm.

4.3 The Case of Big Pages

We observed opinion, that GiST performance is degrading on big pages. In this case, intra-page indexing could help prevent degradation. Currently, PostgreSQL allows to specify page size before compilation, the maximum page size is 32 Kb. This size is restricted mainly by tuple placement structure *ItemIdData* which leaves 15 bits to tuple offset on the page.

We have done a series of tests with different T's on 32 Kb pages.

Table 1. RI and RS tasks time on 32Kb pages (ms).

T	RI			RS		
	T_{master}	$T_{patched}$	$T_{patched}/T_{master}$	T_{master}	$T_{patched}$	$T_{patched}/T_{master}$
8	201635	84671	0.42	36307	38453	1.06
16	201635	80375	0.40	36307	36448	1.00
24	201635	80870	0.40	36307	34573	0.95
32	201635	80135	0.40	36307	40717	1.12

From these results, we can see that while performance gain is sufficient on RI task, there is neither gain in RS task nor visible dependency from the threshold T.

4.4 Effect of Different Thresholds

Following is the test result, obtained with the most common page size of 8 Kb.

From these results, we can conclude that on given physical characteristics of the database and data type, $T = 16$ is somewhat optimal, but have no significant influence on performance if T is picked from sane numbers.

5 Current Limitations and Future Work

The implementation of the new approach brings performance benefits for insertion tasks, fixing common bottlenecks.

Table 2. RI and RS tasks time on 8 Kb pages (ms).

T	RI			RS		
	T_{master}	$T_{patched}$	$T_{patched}/T_{master}$	T_{master}	$T_{patched}$	$T_{patched}/T_{master}$
8	123260	83111	0.67	35135	33556	0.96
16	123260	80930	0.66	35135	32873	0.94
24	123260	88469	0.72	35135	34662	0.99
32	123260	87234	0.71	35135	35400	1.01

But currently, the proposed approach has several unresolved questions. We hope to address these questions first in AGS and next in mainstream GiST.

5.1 Buffered GiST Build

GiST has buffered build, which is used to build an index structure for a preexisting table. Because buffered build initializes itself with building a small tree with regular inserts, it is installing skip tuples too. That is why buffered build has performance improvement from the described technique too. But it is certain that buffered build could benefit more if it consciously used skip tuples rather than benefiting from side effects of initialization with skip tuples.

5.2 kNN Search in GiST

Skip tuples are not integrated into the kNN search pairing heap. If the kNN has a search condition operator, this search will use skip tuples to improve its performance. But granularity of pairing heap is still a page and not a skip group. Changing the granularity of kNN pairing heap will incur CPU performance cost because pairing heap will be larger. But skip tuples are itself technique to improve CPU usage. Thus, at the present state, the implementation of the proposed approach does not use skip tuples during the kNN ordering scan.

6 Related Work

Currently PostgreSQL has space-partitioned GiST SP-GiST [1]. This index is also free from the concept one node is one page, but in another manner. SP-GiST is an unbalanced tree and each page can represent the subgraph of this tree. Unfortunately, SP-GiST inherits few limitations from this its nature of dividing space into subspaces without overlap: it cannot index overlapping keys. In terms of GIS this means that SP-GiST cannot store areas, it can be used only as point access method. Usually, for scans and inserts, SP-GiST uses less CPU but more IO of page buffers. Since it's unbalanced nature, the SP-GiST scan can be unpredictably deep.

7 Conclusion

The current implementation of intra-page indexing can make GiST inserts and updates 1.5x faster. It protects GiST from performance degradation if PostgreSQL is compiled with big pages. But the most important achievement of intra-page indexing is that it opens a way to advance GiST API towards better-generalized algorithms, less code for data type developers and more performant data access methods for these data types.

Acknowledgments

The work was partially supported by Act 211 Government of the Russian Federation, contract No. 02.A03.21.0006.

References

1. Aref, W.G., Ilyas, I.F.: Sp-gist: An extensible database index for supporting space partitioning trees. Journal of Intelligent Information Systems **17**(2-3), 215–240 (2001)
2. Beckmann, N., Seeger, B.: A revised r*-tree in comparison with related index structures. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. pp. 799–812. ACM (2009)
3. Borodin, A.: Gist inserts optimization with pageindextupleoverwrite, `https://commitfest.postgresql.org/10/661`, [Last accessed 30-June-2019]
4. Borodin, A.: Intra-page indexing benchmark, `https://gist.github.com/x4m/56f912ba9278a97f24dfa2b6db46fa7f`, [Last accessed 30-June-2019]
5. Borodin, A.: [proposal] improvement of gist page layout, `https://www.postgresql.org/message-id/flat/CAJEAwVE0rrr\%2B0BT-P0gDCtXbVDkBBG_WcXwCBK\%3DGHo4fewu3Yg\%40mail.gmail.com`, [Last accessed 30-June-2019]
6. Borodin, A.: [wip] gist intrapage indexing, `https://www.postgresql.org/message-id/7780A07B-4D04-41E2-B228-166B41D07EEE@yandex-team.ru`, [Last accessed 30-June-2019]
7. Borodin, A., Mirvoda, S., Kulikov, I., Porshnev, S.: Optimization of memory operations in generalized search trees of postgresql. In: International Conference: Beyond Databases, Architectures and Structures. pp. 224–232. Springer (2017)
8. Borodin, A., Mirvoda, S., Porshnev, S., Bakhterev, M.: Improving penalty function of r-tree over generalized index search tree possible way to advance performance of postgresql cube extension. In: 2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)(. pp. 130–133. IEEE (2017)
9. Chilingarian, I., Bartunov, O., Richter, J., Sigaev, T.: Postgresql: The suitable dbms solution for astronomy and astrophysics. In: Astronomical Data Analysis Software and Systems (ADASS) XIII. vol. 314, p. 225 (2004)
10. Christodoulakis, S., Manolopoulos, Y., Larson, P.Å.: Analysis of overflow handling for variable length records. Information Systems **14**(2), 151–162 (1989)
11. Guttman, A.: R-trees: a dynamic index structure for spatial searching, vol. 14. ACM (1984)
12. Hellerstein, J.M., Naughton, J.F., Pfeffer, A.: Generalized search trees for database systems. September (1995)

13. Kornacker, M.: Access methods for next-generation database systems. University of California, Berkeley (2000)
14. Kornacker, M., Mohan, C., Hellerstein, J.M.: Concurrency and recovery in generalized search trees. In: ACM SIGMOD Record. vol. 26, pp. 62–72. ACM (1997)
15. Korotkov, A.: A new double sorting-based node splitting algorithm for r-tree. Programming and Computer Software **38**(3), 109–118 (2012)
16. Theodoridis, Y., Sellis, T.: A model for the prediction of r-tree performance. In: PODS. vol. 96, pp. 161–171 (1996)