GIN

Report Index

A

abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 61
aerospace propulsion, 52
aerospace robotics, 68
aluminium, 17
amorphous state, 67
angular velocity measurement, 58
antenna phased arrays, 41, 46, 66
argon, 21
assembling, 22
atomic force microscopy, 13, 27, 35
atomic layer deposition, 15
attitude control, 60, 61
attitude measurement, 59, 61
automatic test equipment, 71
automatic testing, 24

B

backward wave oscillators, 45

compensation, 30, 68
compressive strength, 54
compressors, 29
computational fluid dynamics, 23, 29
computer games, 56
concurrent engineering, 14
contact resistance, 47, 66
convertors, 22
coplanar waveguide components, 40
Couette flow, 21
cree, 17
crystallisation, 64
current density, 13, 16

design for manufacture, 25
design for testability, 25
diamond, 3, 27, 43, 54, 67
dielectric losses, 31, 42
dielectric polarisation, 31
dielectric relaxation, 64
dielectric thin films, 16
differential amplifiers, 28
diffraction gratings, 58
discrete wavelet transforms, 72
displacement measurement, 11
display devices, 56
distributed feedback lasers, 38
GIN: on the way to 8.4

- Multicolumn GIN
- Fast GIN update
- Partial match
- Miscellaneous
Multicolumn GIN

Traditional approach:
- Index on ( a int[], b int[] ) - store pairs of (a[i],b[j])
- Fast search on a[] and (a[],b[]) , slow on b[]
- Extremely ineffective storage: $\sim Na*Nb$
Multicolumn GIN

Suggested approach:
- Index on ( a int[], b int[] ) - store each element separately along with its column number
- Fast search on any subset of columns
- Effective storage: ~ Na+Nb

CREATE INDEX  gin_idx ON TAB USING GIN(A,B);
Multicolumn GIN structure

Entry page, level 0 (leaf)
- 1:aaa
  - Pointer to posting tree: B-Tree over ItemPointer to heap
- 1:abc
  - Posting list: sorted array of ItemPointer to heap
- 2:abc
  -...

Entry page, level 0
- 2:baa
- 2:bar

Posting page, level N: ItemPointer
- 14:17
- 218:1
- 1021:6

Posting page, level 0 (leaf)
- 1:33
- 2:7
- 14:17

Posting page, level 0 (leaf)
- 123:1
- 158:18
- Right bound 218:1

Entry page, level N: keywords
- 1:abc
- 2:bar
- 2:foo
Multicolumn GIN: Tuple layout

Posting list (tuple size < TOAST_INDEX_TARGET):

- `itup->t_info & INDEX_SIZE_MASK`
  - size of whole tuple

- `ItemPointerGetBlockNumber(&itup->t_tid)`
  - size of original tuple (without posting list)

- `ItemPointerGetOffsetNumber(&itup->t_tid)`
  - number of elements in posting list

- VALUE(S)

- Posting list of ItemPointers to heap

- `IndexTupleSize()`
- `GinGetOrigSizePosting()`
- `GinSetOrigSizePosting()`
- `GinGetNPosting()`
- `GinSetNPosting()`
- `GinGetPosting()`
Multicolumn GIN: Tuple layout

Posting tree (tuple size >= TOAST_INDEX_TARGET):

- `itup->t_info & INDEX_SIZE_MASK`
  - size of whole tuple

- `ItemPointerGetBlockNumber(&itup->t_tid)`
  - block number of root of posting tree

- `ItemPointerGetOffsetNumber(&itup->t_tid)`
  - magick number GIN_TREE_POSTING

- `VALUE(S)`

- `IndexTupleSize()`
- `GinSetPostingTree()`
- `GinGetPostingTree()`
- `GinIsPostingTree()`
Multicolumn GIN: Tuple layout

Single-column (current):

Multi-column:
Multicolumn GIN: GinState

typedef struct GinState
{
    FmgrInfo    compareFn[INDEX_MAX_KEYS];
    FmgrInfo    extractValueFn[INDEX_MAX_KEYS];
    FmgrInfo    extractQueryFn[INDEX_MAX_KEYS];
    FmgrInfo    consistentFn[INDEX_MAX_KEYS];
    FmgrInfo    comparePartialFn[INDEX_MAX_KEYS];
    bool        canPartialMatch[INDEX_MAX_KEYS];
    bool        oneCol;
    TupleDesc   origTupdesc; /* index->rd_att */
    /* OffsetNumber, Datum[i] */
    TupleDesc   tupdesc[INDEX_MAX_KEYS];

    /* Instead of index_getattr():
    * OffsetNumber gintuple_get_attrnum(GinState*, IndexTuple) — returns colN
    * Datum gin_index_getattr(GinState*, IndexTuple) — returns value
    */
} GinState;
Multicolumn GIN: Example

100,000 int[500], cardinality 500,000

```
=# \d test
Table "public.test"
<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>integer[]</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>integer[]</td>
<td></td>
</tr>
</tbody>
</table>
Indexes:
   "gin_idx" gin (v1, v2)

=# \d tt
Table "public.tt"
<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td>integer[]</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td>integer[]</td>
<td></td>
</tr>
</tbody>
</table>
Indexes:
   "gidx_v1" gin (v1)
   "gidx_v2" gin (v2)
```

```
=# select pg_relation_size('gin_idx') as mc_idx_size,
    pg_relation_size('gidx_v2')+
    pg_relation_size('gidx_v1') as sum_idx;

 mc_idx_size | sum_idx
-------------+-----------
 539492352   | 538984448
```
Multicolumn GIN: Example

100,000 int[500], cardinality 500,000

=# explain analyze select count(*) from tt where v2 && '{1,3}' and v1 && '{31,56}';

Aggregate (cost=1338.86..1338.87 rows=1 width=0) (actual time=4.892..4.895 rows=1 loops=1)
   -> Bitmap Heap Scan on tt (cost=1330.88..1338.85 rows=2 width=0) (actual time=4.611..4.789 rows=36 loops=1)
      Recheck Cond: ((v2 && '{1,3}'::integer[]) AND (v1 && '{31,56}'::integer[]))
         -> BitmapAnd (cost=1330.88..1330.88 rows=2 width=0) (actual time=4.577 rows=0 loops=1)
            -> Bitmap Index Scan on gidx_v2 (cost=0.00..665.32 rows=500 width=0) (actual time=1.836..1.836 rows=1516 loops=1)
               Index Cond: (v2 && '{1,3}'::integer[])
            -> Bitmap Index Scan on gidx_v1 (cost=0.00..665.32 rows=500 width=0) (actual time=1.924..1.924 rows=1489 loops=1)
               Index Cond: (v1 && '{31,56}'::integer[])

Total runtime: 4.994 ms
Multicolumn GIN: Example

100,000 int[500], cardinality 500,000

=# explain analyze select count(*) from test where v2 && '{1,3}' and v1 && '{31,56}';

Aggregate (cost=22.95..22.96 rows=1 width=0) (actual time=1.740..1.742 rows=1 loops=1)
  ->  Index Scan using gin_idx on test  (cost=0.00..22.94 rows=3 width=0) (actual time=0.274..1.615 rows=36 loops=1)
      Index Cond: ((v1 && '{31,56}'::integer[]) AND (v2 && '{1,3}'::integer[]))
Total runtime: 1.855 ms

Multicolumn index vs. 2 single column indexes

<table>
<thead>
<tr>
<th></th>
<th>Multicolumn index</th>
<th>2 single column indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>539 Mb</td>
<td>538 Mb</td>
</tr>
<tr>
<td>Speed</td>
<td>1.885 ms</td>
<td>4.994 ms</td>
</tr>
<tr>
<td>Index</td>
<td>~ 340 s</td>
<td>~ 200 s</td>
</tr>
<tr>
<td>Insert</td>
<td>72 s/10000</td>
<td>~ 66 s/10000</td>
</tr>
</tbody>
</table>
Fast GIN update: The Problem

CREATE TABLE
INSERT 10,000 int[]
CREATE INDEX
3.1 s + 11 s
13.1 s

CREATE TABLE
CREATE INDEX
INSERT 10,000 int[]
~0 s + 100 s
100 s

BULK index insert ~ 10 times faster!
Fast GIN update: The Problem

1: 1
2: 2,3
3: 2
5: 1,3
6: 1
7: 1,2
12: 2
14: 3
25: 3

1, 5, 6, 7
12, 7, 3, 2
5, 2, 25, 14

1
2
3

Oleg Bartunov, Teodor Sigaev
PostgreSQL Conference, Ottawa, May 20-23, 2008
Fast GIN update: The Problem

1 new object -> 6 index updates!
Fast GIN update: Idea

• Delayed Insert
  – Accumulate new index rows on separate pending pages
  – Use Bulk Insert (as in CREATE INDEX) at vacuum time

• Search
  – Index scan on GIN + scan of pending pages
Fast GIN update: Page Layout

Root of GIN tree (BlockNumber=1)
- abc
- bar
- foo

Meta page (BlockNumber = 0)
- List's head
- List's tail, freeSpaceSize

Page 1 (head)
- (34:1),aaa
- (34:1),afg
- (34:1),foo

Page 2
- (34:1),xxx
- (18:4),azz
- (18:4),zzz

Page N (tail)
- (10,1),aaa
- (70:7),zzz
- freespace
Fast GIN update

• Requirement of locking protocol:
  – Any access should lock metapage first

• Properties of pending list:
  – Rows are unordered in the list (as inserted)
  – Keys of the same row are stored continuously in the list
Fast GIN update: small row

1. LockExclusive(metapage)

2. Size < FreeSpace(tail) - if Yes, go to 3; if No, go to Next slide.

3. LockExclusive(tail)

4. Insert into tail and update FreeSpace on metapage

5. UnlockAll
Fast GIN update: big row

1. LockExclusive(metapage)
2. Size < FreeSpace(tail)
3. Yes
   - Previous slide
4. No
   - Unlock(metapage)
   - Make sublist
   - LockExclusive(tail)
   - LockExclusive(metapage)
   - Append new sublist into tail of existing list and unlock all
Fast GIN update: vacuum

1. LockExclusive(metapage)
2. Unlock/Unlock metapage and CHECK_FOR_INTERRUPTS()
3. Read page from list
4. Delete readed pages from a head by separate scan of list [Step 2]
5. Insert collected data in GIN using bulk insert [Step 1]
6. Yes
7. Yes
8. Yes
9. No
10. No
11. No
12. Yes
13. No
14. No
15. Yes
16. No
Fast GIN update

- Row's data between [Step 1] and [Step 2] exists in both regular structure and in the pending list (preserve integrity)
- Search should start from the pending list and then go to regular search in GIN (preserve consistency of search)
Fast GIN update: search

LockShare(metapage) → Is list empty?

Yes

UnLock(metapage)

No

LockShare(head) → UnLock(metapage) → Go to regular search in GIN

Sequentially read list with checking of match by consistentFn for each row
# Fast GIN update: The Problem

<table>
<thead>
<tr>
<th>Operation</th>
<th>1st Attempt</th>
<th>2nd Attempt</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE TABLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT 10,000 int[]</td>
<td>~0 s + 100 s</td>
<td>100 s</td>
</tr>
<tr>
<td>VACUUM TABLE</td>
<td>~0 s + 18 s + 12 s</td>
<td>30 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BULK INSERT</td>
<td>10 s</td>
</tr>
<tr>
<td>OLD_GIN</td>
<td>100 s</td>
</tr>
<tr>
<td>NEW_GIN</td>
<td>30 s</td>
</tr>
</tbody>
</table>
## Fast GIN update: Tests

### Number of elements: 100, cardinality: 500, rows: 100,000

<table>
<thead>
<tr>
<th>Nrows</th>
<th>insert orig.</th>
<th>insert FU</th>
<th>i+v orig.</th>
<th>i+v FU</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>89</td>
<td>8</td>
<td>262</td>
<td>255</td>
</tr>
<tr>
<td>100</td>
<td>104</td>
<td>36</td>
<td>275</td>
<td>230</td>
</tr>
<tr>
<td>1000</td>
<td>904</td>
<td>324</td>
<td>1074</td>
<td>576</td>
</tr>
<tr>
<td>10000</td>
<td>13319</td>
<td>3363</td>
<td>13569</td>
<td>5719</td>
</tr>
</tbody>
</table>

### Number of elements: 1000, cardinality: 500, rows: 100,000

<table>
<thead>
<tr>
<th>Nrows</th>
<th>insert orig.</th>
<th>insert FU</th>
<th>i+v orig.</th>
<th>i+v FU</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>203</td>
<td>36</td>
<td>2647</td>
<td>1434</td>
</tr>
<tr>
<td>100</td>
<td>4126</td>
<td>318</td>
<td>6229</td>
<td>1932</td>
</tr>
<tr>
<td>1000</td>
<td>14527</td>
<td>9389</td>
<td>15777</td>
<td>11112</td>
</tr>
<tr>
<td>10000</td>
<td>92108</td>
<td>36517</td>
<td>93410</td>
<td>46919</td>
</tr>
</tbody>
</table>

### Number of elements: 100, cardinality: 500,000, rows: 100,000

<table>
<thead>
<tr>
<th>Nrows</th>
<th>insert orig.</th>
<th>insert FU</th>
<th>i+v orig.</th>
<th>i+v FU</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>111</td>
<td>6</td>
<td>466</td>
<td>799</td>
</tr>
<tr>
<td>100</td>
<td>3691</td>
<td>35</td>
<td>4009</td>
<td>6598</td>
</tr>
<tr>
<td>1000</td>
<td>190299</td>
<td>318</td>
<td>190487</td>
<td>18910</td>
</tr>
<tr>
<td>10000</td>
<td>–</td>
<td>17668</td>
<td>–</td>
<td>34225</td>
</tr>
</tbody>
</table>
Partial Match: The Problem

- Prefix search for a text search
- Improve performance LIKE '%foo%'
  - It's not a full text search
  - Btree index (text_pattern_ops) can improve
    - LIKE '%FOO'
    - LIKE 'FOO%'

Partial Match: Idea

- Index all permutations of string!

```sql
contrib_regression=# select permute('hello');
permute
------------------------
{hello$, ello$h, llo$he, lo$hel, o$hell}
```

- `$` is used for visualization, we use \0
- LIKE '%l%' => ~ 'l*'
- LIKE 'h%o' => ~ 'o$h*'
- LIKE '%o' => ~ 'o*$'
- LIKE 'h%' => ~ 'h*$'

- Add support of partial match to GIN – currently only exact comparison
Partial Match: API

Four (or five) interface functions (pseudocode):

- Datum* extractValue(Datum inputValue, uint32* nentries)
- int compareEntry(Datum a, Datum b)
- Datum* extractQuery(Datum query, uint32* nentries, StrategyNumber n, bool* pmatch[])
- bool consistent(bool check[], StrategyNumber n, Datum query, bool *needRecheck)
- int comparePartial(Datum query_key, Datum indexed_key, StrategyNumber n)
Datum* extractQuery(Datum query, uint32* nentries, StrategyNumber n, bool* pmatch[])  

Returns an array of Datum of keys of the query to be executed. n is the strategy number of the operation. Depending on n, query can be different type.

Each element of the pmatch[] should be set to TRUE if the corresponding key requires partial match, FALSE if not. If *pmatch is set to NULL then GIN assumes partial match is not required. ExtractQuery is responsible for allocation memory for pmatch.
Partial Match: API

int comparePartial(Datum query_key, Datum indexed_key, Strategynumber n)

Compare a partial-match query with an index key.
Returns:

- $<0$ - means the index key does not match the query, but the index scan should continue
- $=0$ - means that the index key does match the query
- $>0$ - stop index scan, since no more matches are possible
Partial Match

Query: LIKE 'h%' => 'h*$'

compareEntries('h', key')

Begin scan with partial match

compareToPartial( 'h*$', 'hello$') => 0 (match)

compareToPartial( 'h*$', 'horse$a') => -1 (scan)

compareToPartial( 'h*$', 'horror$') => 0 (match)

compareToPartial( 'h*$', 'rticle$a') => 1 (stop)
Partial Match: wildspeed

750,000 words, average length is 8 characters, time in ms

<table>
<thead>
<tr>
<th></th>
<th>h%</th>
<th>hel%</th>
<th>h%o</th>
<th>%l%</th>
<th>%lll%</th>
<th>%l</th>
<th>%lll</th>
<th>%ll%o</th>
</tr>
</thead>
<tbody>
<tr>
<td>h%</td>
<td>28.0</td>
<td>1.1</td>
<td>1.1</td>
<td>434</td>
<td>0.7</td>
<td>426</td>
<td>0.7</td>
<td>18</td>
</tr>
</tbody>
</table>

Btree/seqscan | 8.5 | 1.0 | 8.6 | 415 | 408 | 407 | 404.0 | 404 |

CREATE INDEX ... USING btree (w text_pattern_ops) : 3.175 seconds
CREATE INDEX ... USING gin (w2 wildcard_ops) : 1 hour 10 minutes

Limitation: during index scan with comparePartial() ItemPointers are collected in TIDBitmap which might become lossy. In that case GIN will emit error with suggestion to increase work_mem. TIDBitmap is used to OR-ed ItemPointer's lists.
Partial Match: prefixes in tsearch

The popular request for the text search

```sql
SELECT 'superstar on party'::tsvector @@ 'super:*' AS yes;
   yes
   -----
      t

SELECT 'supernovae:1A sky:2B'::tsvector @@ 'super:A*' AS yes;
   yes
   -----  
      t
```

API of dictionary supports prefix flag
Miscellaneous

- Removed @@ text search operator (index API changes)
- Full index scan support (if provided by opclass) - pmatch
- Fast GIN statistics – gin_stat(Index)
Comparison of exact (ts_stat) and approximated stats about 500,000 documents

```sql
=# select a.word, b.ndoc as exact, a.estimation as estimation, round ( (a.estimation*b.ndoc)*100.0/a.estimation,2) || '%' as error from (SELECT * FROM gin_stat('gin_x_idx') as t(word text, estimation int) order by estimation desc limit 5 ) as a, stat b where a.word = b.word;

<table>
<thead>
<tr>
<th>word</th>
<th>exact</th>
<th>estimation</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td>page</td>
<td>340430</td>
<td>340858</td>
<td>0.13%</td>
</tr>
<tr>
<td>figur</td>
<td>240104</td>
<td>240366</td>
<td>0.11%</td>
</tr>
<tr>
<td>use</td>
<td>147132</td>
<td>148022</td>
<td>0.60%</td>
</tr>
<tr>
<td>model</td>
<td>133444</td>
<td>134442</td>
<td>0.74%</td>
</tr>
<tr>
<td>result</td>
<td>128977</td>
<td>129010</td>
<td>0.03%</td>
</tr>
</tbody>
</table>
```

(5 rows)

Time: 550.562 ms
Patches

- http://www.sigaev.ru/misc/fast_insert_gin-0.2.gz
- http://www.sigaev.ru/misc/multicolumn_gin-0.2.gz
- http://www.sigaev.ru/misc/wildspeed-0.12.tgz
- Fast insert and multicolumn patches are mutually exclusive
Acknowledgements

- hackers
  - Creative discussions and reviews
- EnterpriseDB, jfg://networks
  - GIN Partial match
  - prefix search
  - Fast GIN update
  - multicolumn support