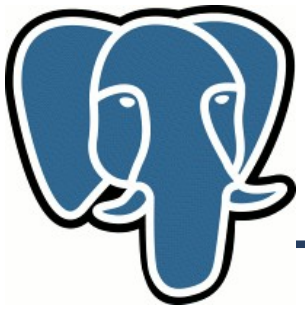


GIN

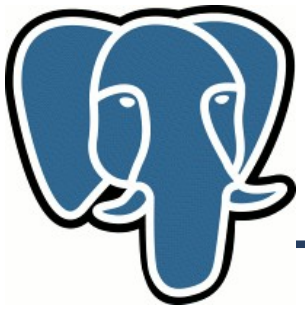
Report Index

A

B

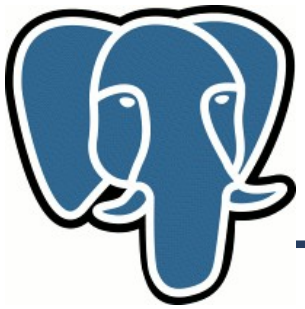
D

E



GIN: on the way to 8.4

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



Multicolumn GIN

Traditional approach:

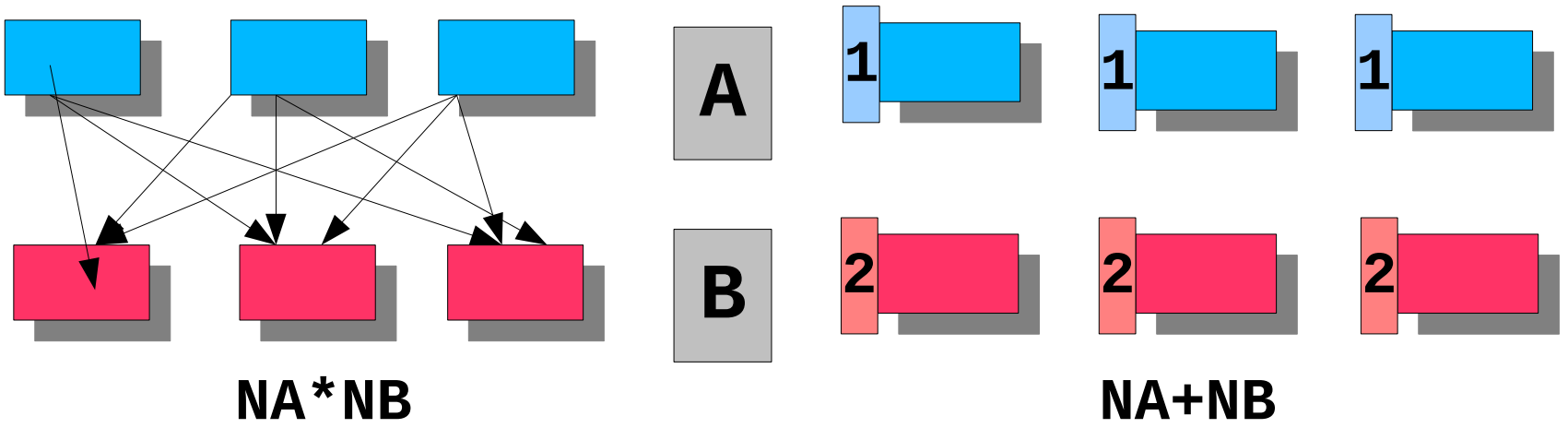
- Index on (a int[], b int[]) - store pairs of (a[i],b[j])
- Fast search on a[] and (a[],b[]) , slow on b[]
- Extremely ineffective storage: ~ Na*Nb

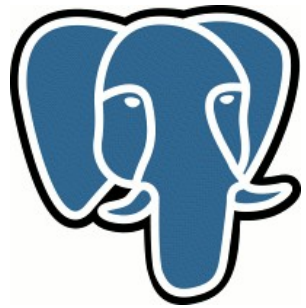
Multicolumn GIN

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 N: keywords

1:abc | 2:bar | 2:foo

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

Right link

Posting page, level N: ItemPointer

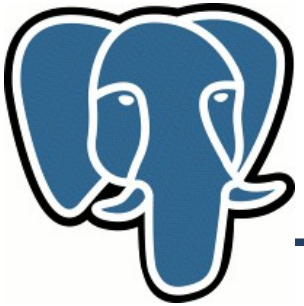
14:17 | 218:1 | 1021:6

Posting page, level 0 (leaf)

1:33 | 2:7 | 14:17 | Right bound 14:17

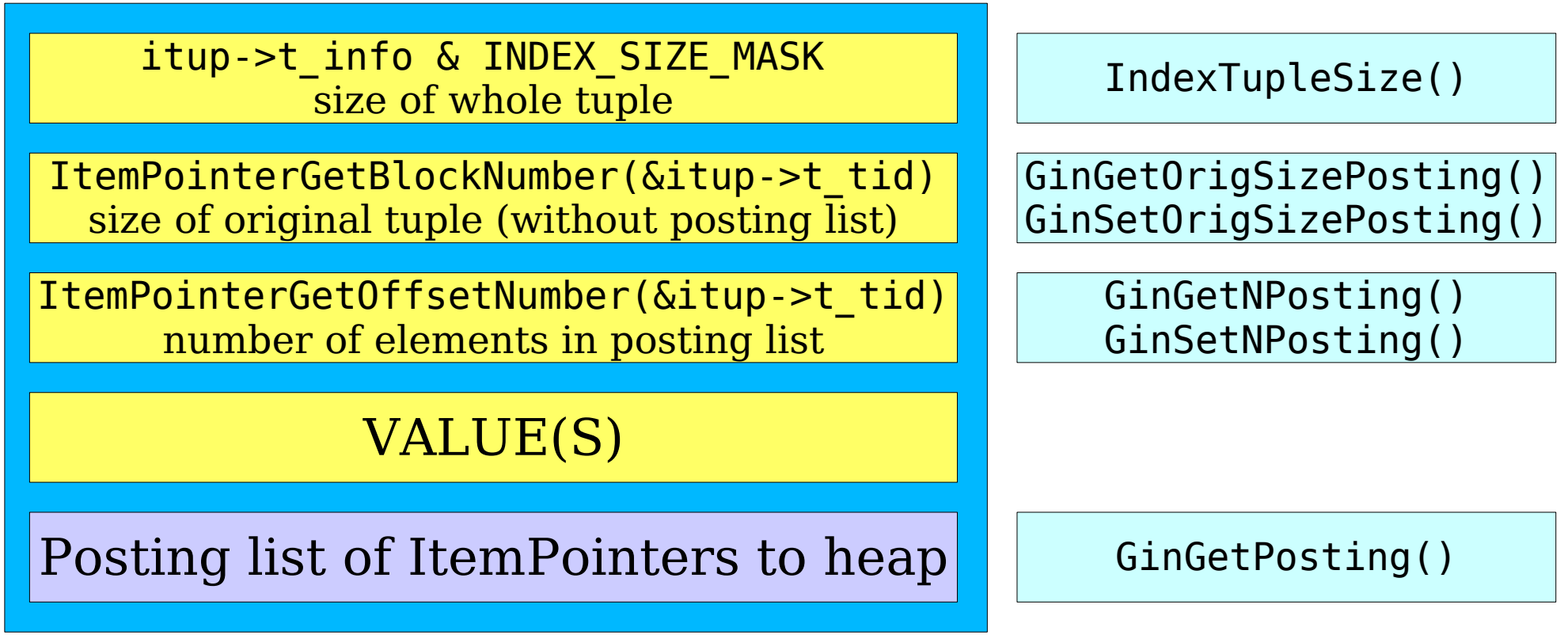
Posting page, level 0 (leaf)

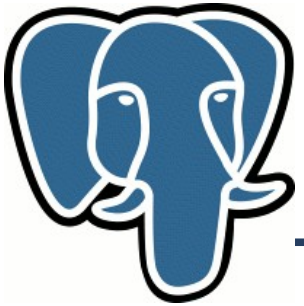
123:1 | 158:18 | Right bound 218:1



Multicolumn GIN: Tuple layout

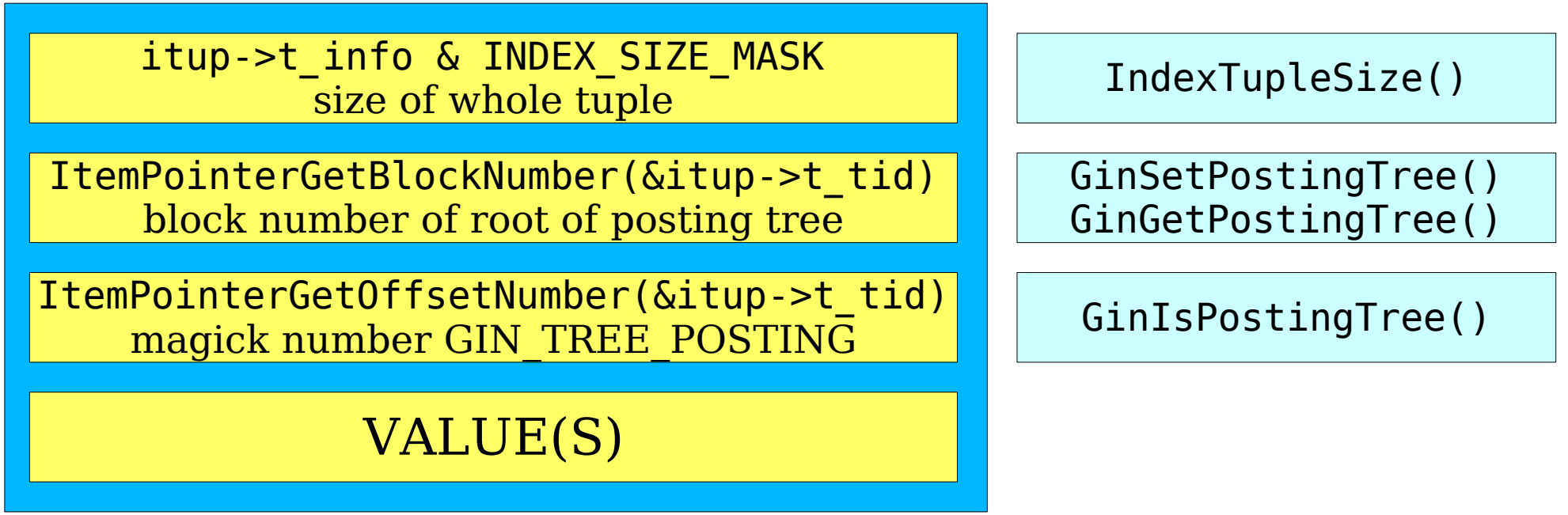
Posting list (tuple size < TOAST_INDEX_TARGET):





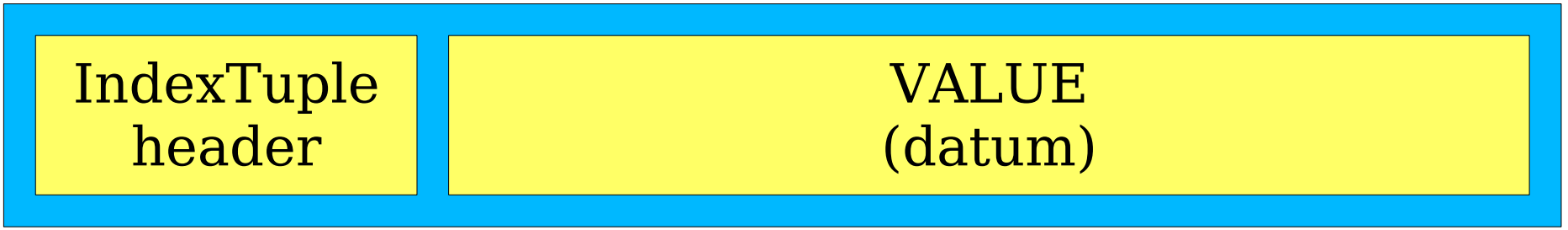
Multicolumn GIN: Tuple layout

Posting tree (tuple size >= TOAST_INDEX_TARGET):

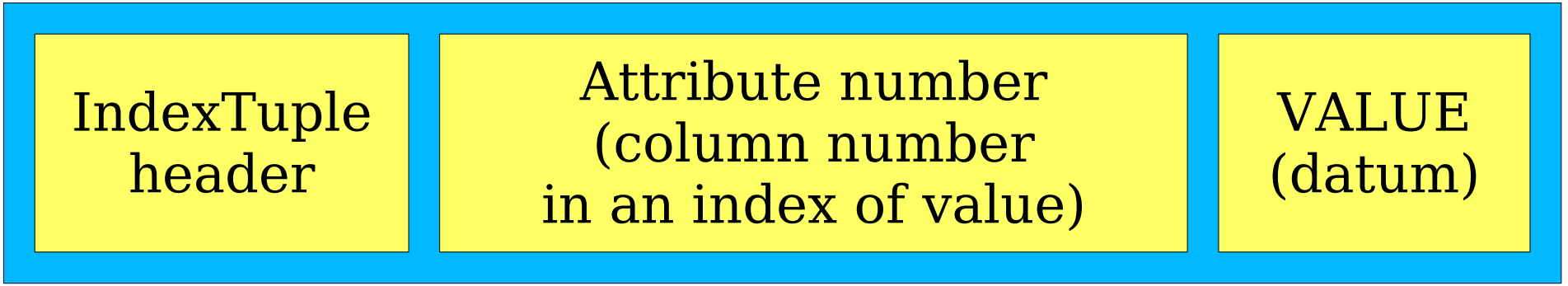


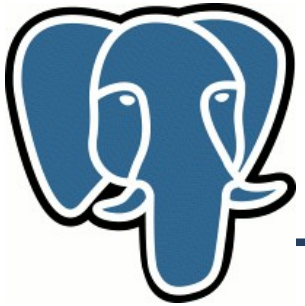
Multicolumn GIN: Tuple layout

Single-column (current):



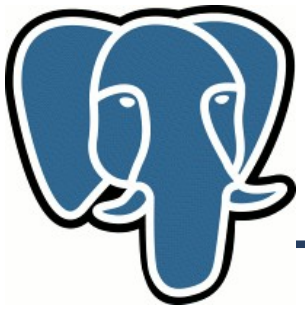
Multi-column :





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;
```



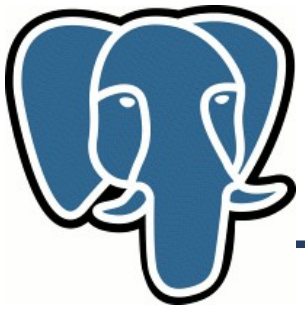
Multicolumn GIN: Example

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

```
=# \d test
      Table "public.test"
 Column |   Type    | Modifiers
--------+-----------+-----------
 id     | integer   |
 v1     | integer[] |
 v2     | integer[] |
Indexes:
    "gin_idx" gin (v1, v2)
```

```
=# \d tt
       Table "public.tt"
 Column |   Type    | Modifiers
--------+-----------+-----------
 id     | integer   |
 v1     | integer[] |
 v2     | integer[] |
Indexes:
    "gidx_v1" gin (v1)
    "gidx_v2" gin (v2)
```

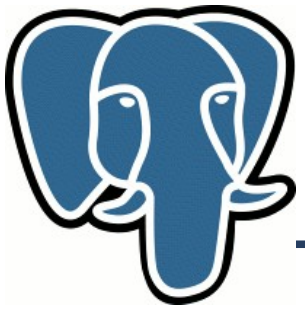
```
=# select pg_relation_size('gin_idx') as mc_idx_size,
          pg_relation_size('gidx_v2')+
          pg_relation_size('gidx_v1') as sum_idx;
 mc_idx_size |  sum_idx
-------------+-----------
   539492352 | 538984448
```



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 row
s=1 loops=1)
   ->  Bitmap Heap Scan on tt  (cost=1330.88..1338.85 rows=2 width=0) (actual ti
me=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=4.57
7..4.577 rows=0 loops=1)
               ->  Bitmap Index Scan on gidx_v2  (cost=0.00..665.32 rows=500 wid
th=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 wid
th=0) (actual time=1.924..1.924 rows=1489 loops=1)
                     Index Cond: (v1 && '{31,56}'::integer[])
 Total runtime: 4.994 ms
```



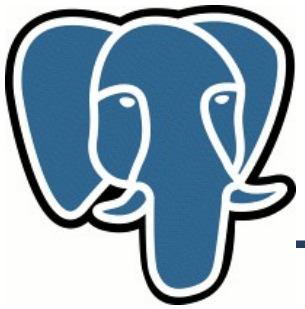
Multicolumn GIN: Example

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) (actu
al 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

Size:	539 Mb	538 Mb
Speed:	1.885 ms	4.994 ms
Index:	~ 340 s	~ 200 s
Insert:	72 s/10000	~ 66 s/10000

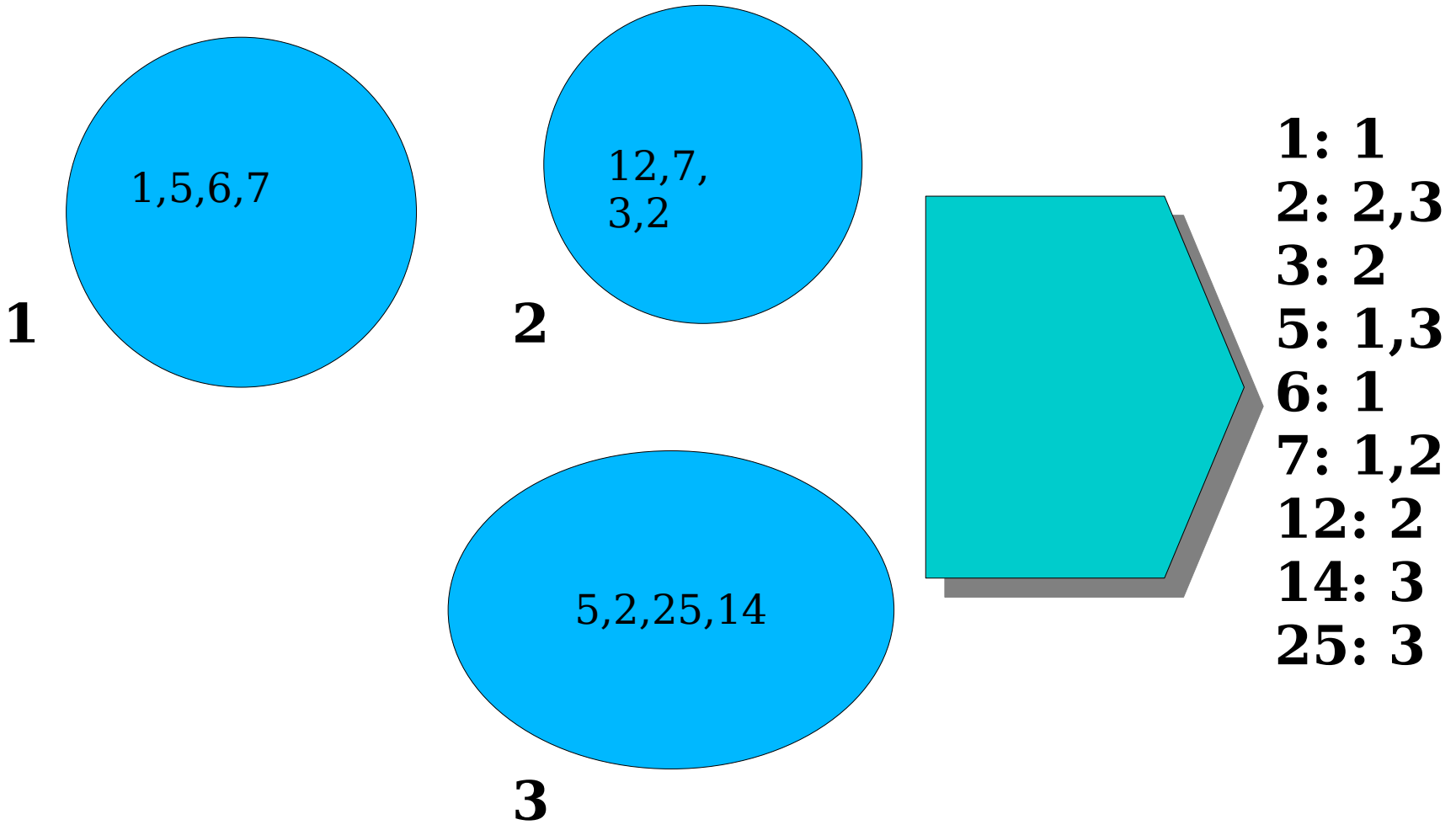


Fast GIN update: The Problem

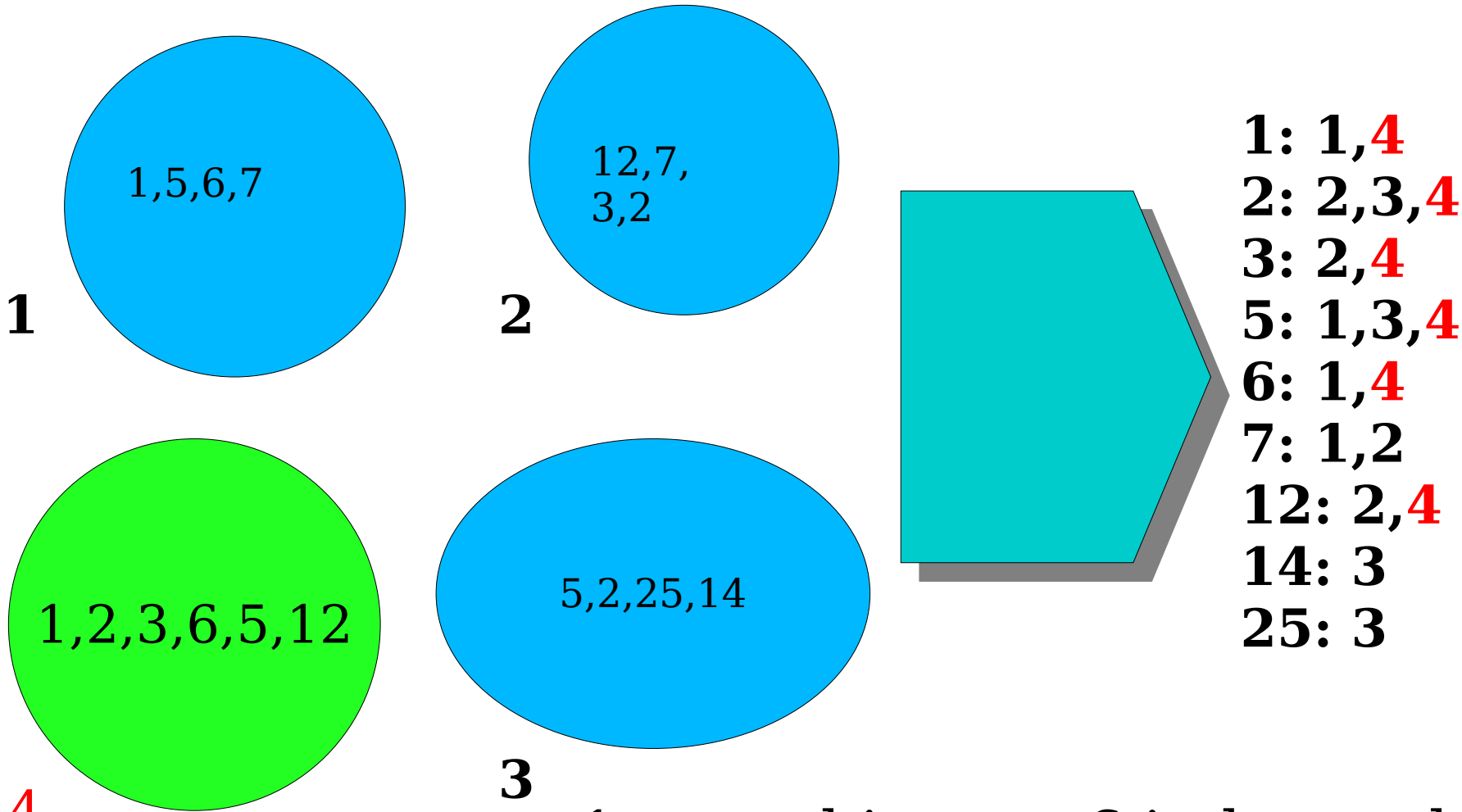
```
CREATE TABLE                CREATE TABLE
INSERT 10,000 int[]         CREATE INDEX
CREATE INDEX                INSERT 10,000 int[]

 3.1 s + 11 s               ~0 s + 100 s
    13.1 s                       100 s
```

BULK index insert ~ 10 times faster !

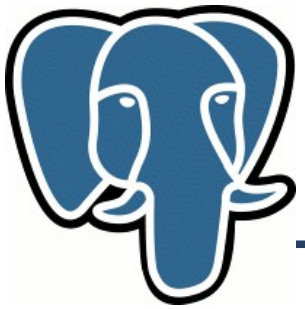
Fast GIN update: The Problem



Fast GIN update: The Problem



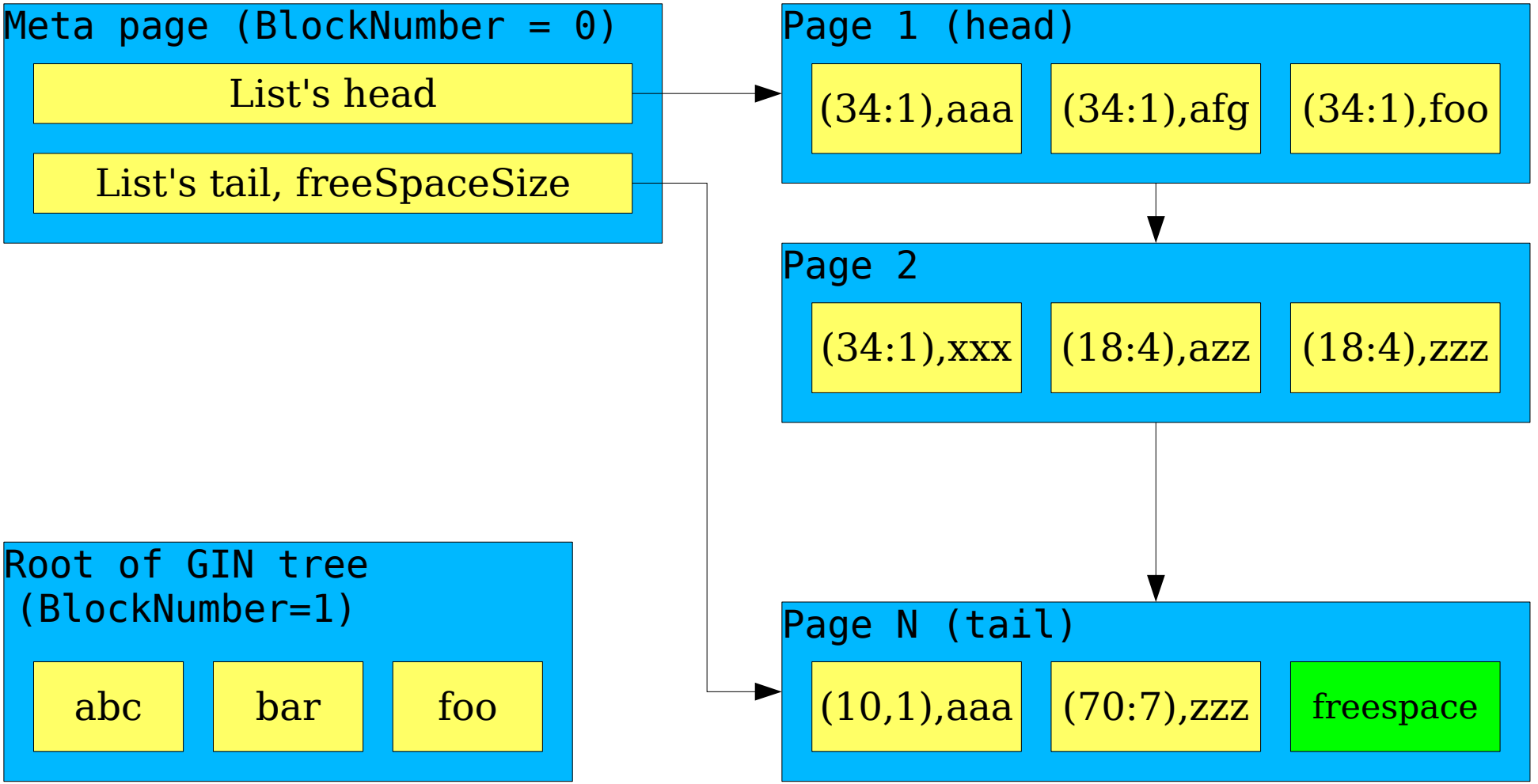
1 new object -> 6 index updates !

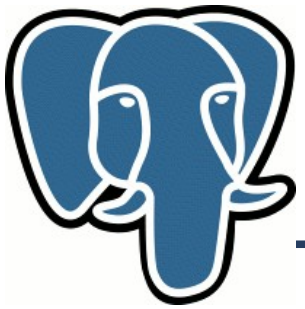


Fast GIN update: Idea

- 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

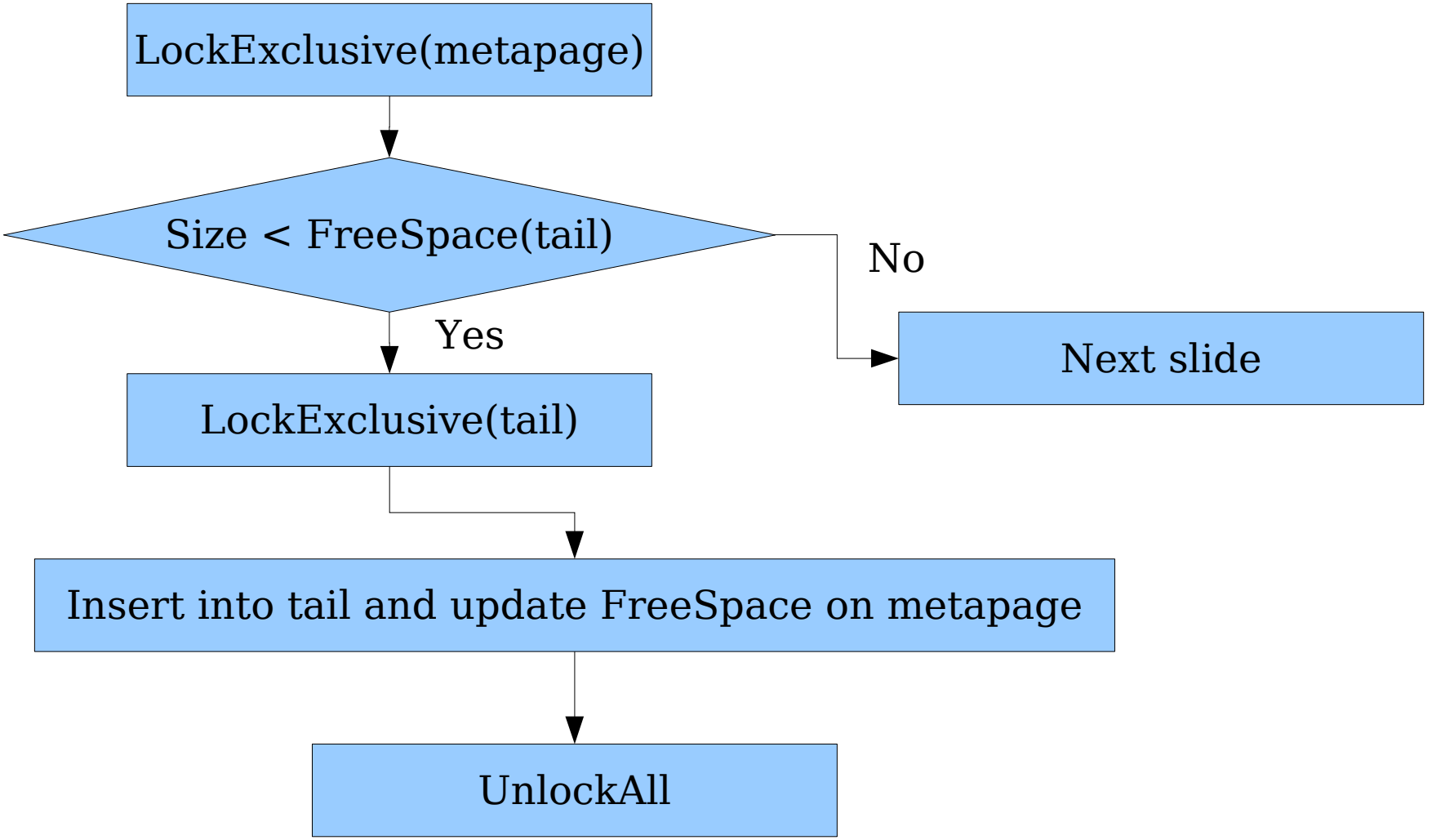




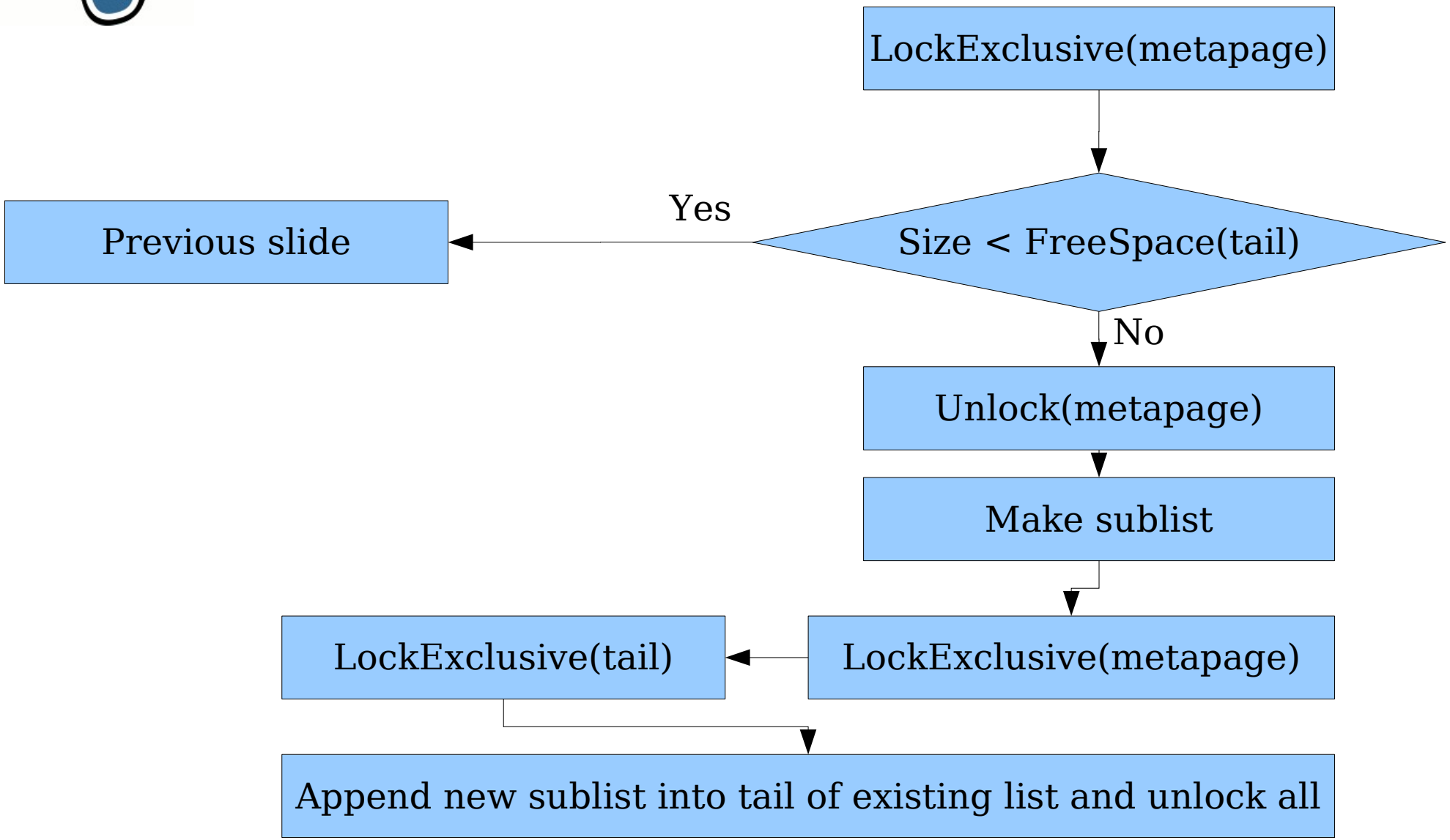
Fast GIN update

- 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

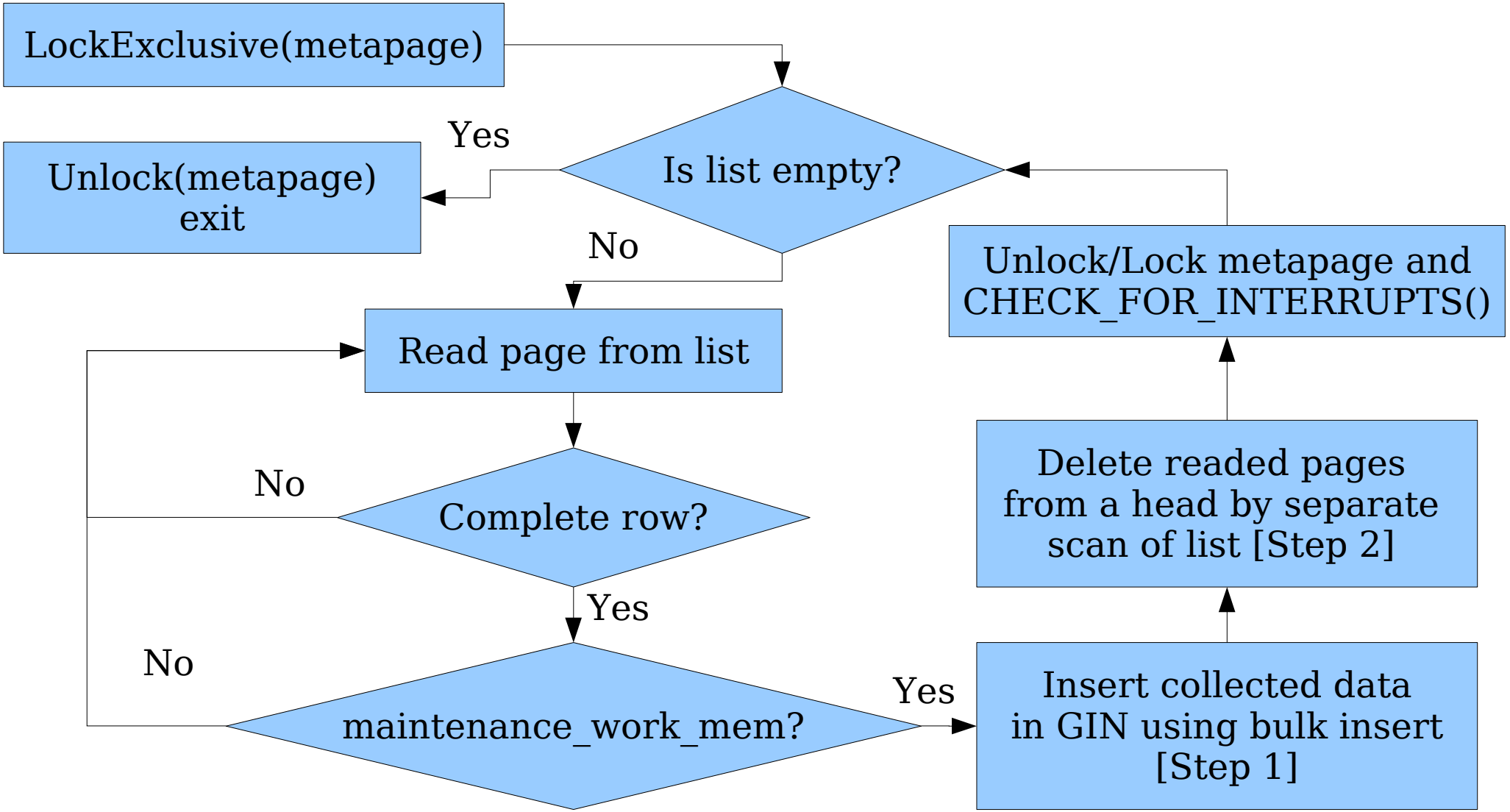
Fast GIN update: small row

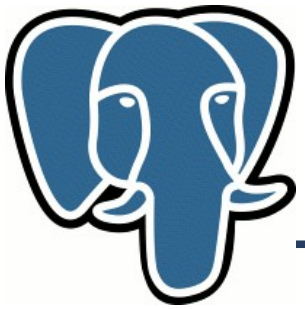


Fast GIN update: big row



Fast GIN update: vacuum

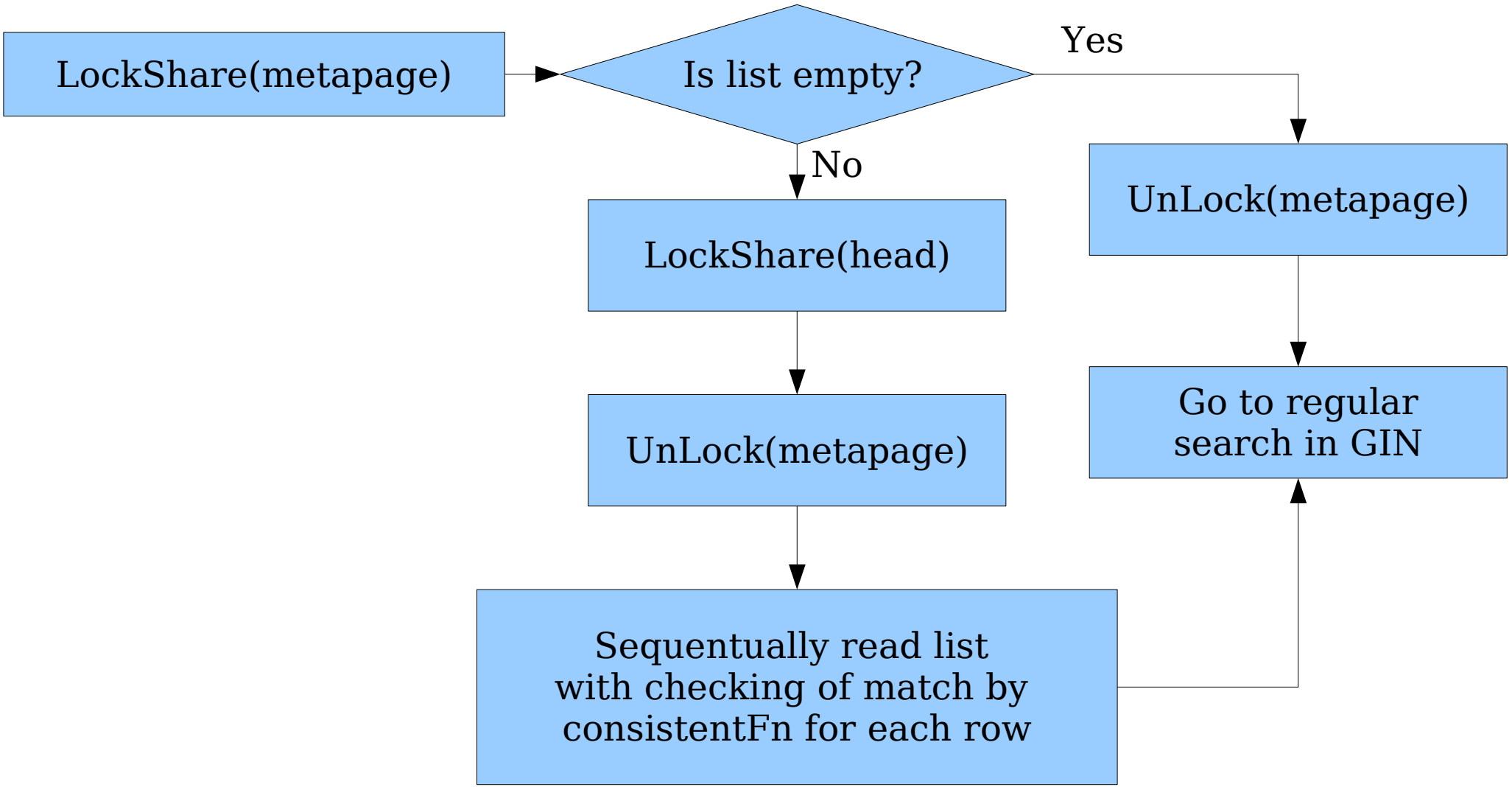


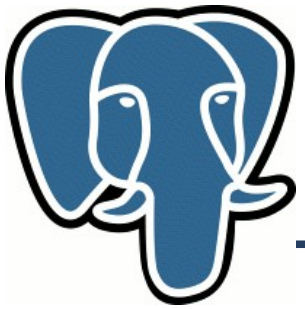


Fast GIN update

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

Fast GIN update: search



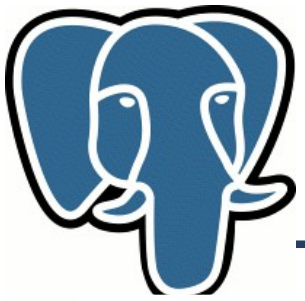


Fast GIN update: The Problem

```
CREATE TABLE
CREATE INDEX
INSERT 10,000 int[]

~0 s + 100 s
     100 s
```

```
CREATE TABLE
CREATE INDEX
INSERT 10,000 int[]
VACUUM TABLE

~0 s + 18 s + 12 s
     30 s
```

BULK INSERT	OLD_GIN	NEW_GIN
10 s	100 s	30 s



Fast GIN update: Tests

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

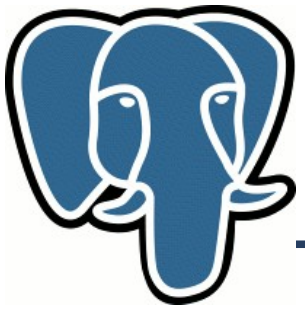
Nrows	insert orig.	insert FU	i+v orig.	i+v FU
10	89	8	262	255
100	104	36	275	230
1000	904	324	1074	576
10000	13319	3363	13569	5719

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

Nrows	insert orig.	insert FU	i+v orig.	i+v FU
10	203	36	2647	1434
100	4126	318	6229	1932
1000	14527	9389	15777	11112
10000	92108	36517	93410	46919

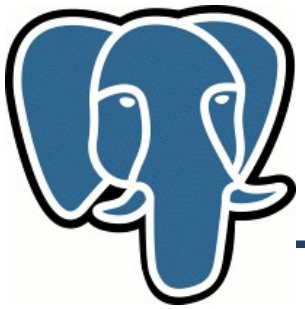
Number of elements: 100, cardinality: 500,000, rows: 100,000

Nrows	insert orig.	insert FU	i+v orig.	i+v FU
10	111	6	466	799
100	3691	35	4009	6598
1000	190299	318	190487	18910
10000	-	17668	-	34225



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%'

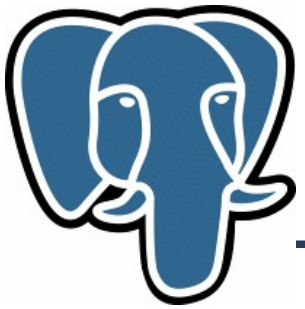


Partial Match: Idea

- Index all permutations of string !

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

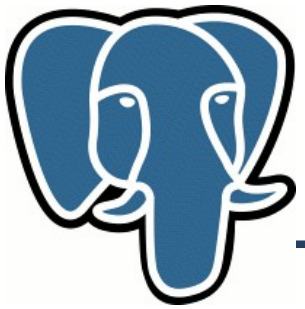
  - '$' is used for visualization, we use \0
  - LIKE '%l%' => ~ 'l*'
  - LIKE 'h%o' => ~ 'o$h*'
  - LIKE '%o'   => ~ 'o$*'
  - LIKE 'h%'   => ~ 'h*$
- Add support of partial match to GIN – currently only exact comparison



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)

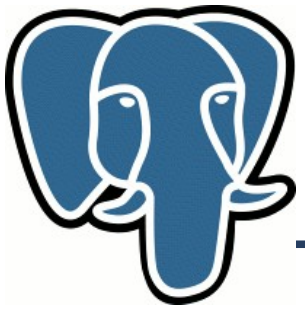


Partial Match: API

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

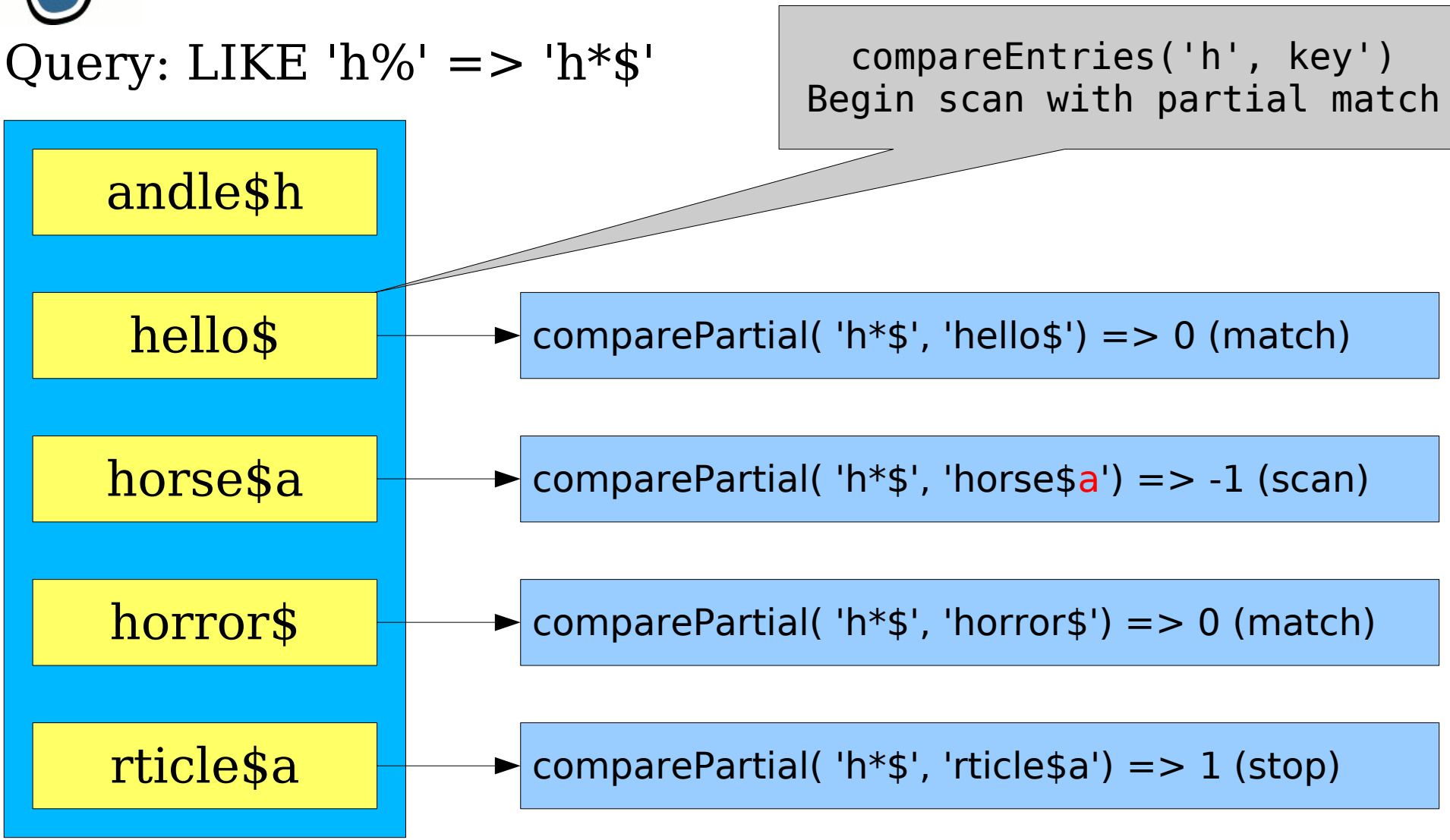
int comparePartial(Datum query_key,
Datum indexed_key, Strategynumber n)

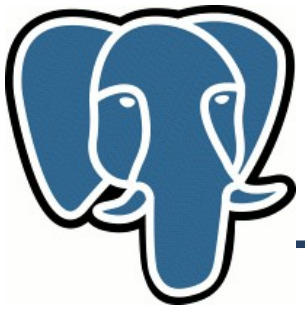
Compare a partial-match query with an index key.
Returns:

- <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*$'





Partial Match: wildspeed

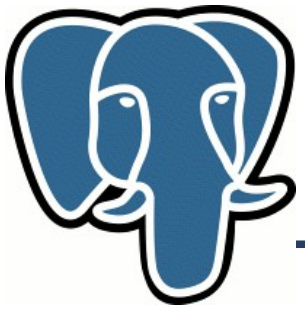
750,000 words, average length is 8 characters, time in ms

	h%	hel%	h%o	%l%	%lll%	%l	%lll	%ll%o
wildspeed	28.0	1.1	1.1	434	0.7	426	0.7	18
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.

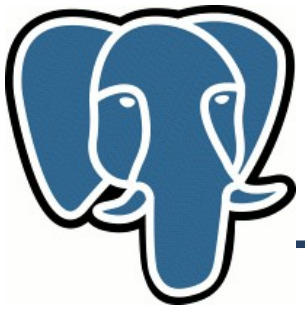


Partial Match: prefixes in tsearch

The popular request for the text search

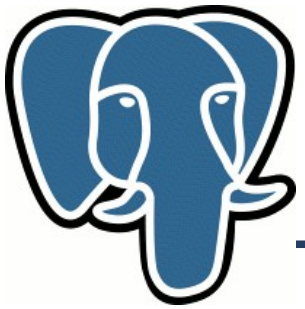
```
SELECT 'superstar on party'::tsvector @@ 'super:*' AS yes;
   yes
  -----
    t

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

API of dictionary supports prefix flag



Miscellaneous

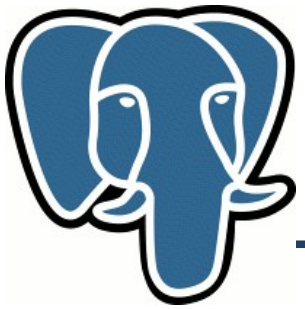
- Removed @@@ text search operator (index API changes)
- Full index scan support (if provided by opclass) - pmatch
- Fast GIN statistics – gin_stat(Index)



Miscellaneous: GIN stat

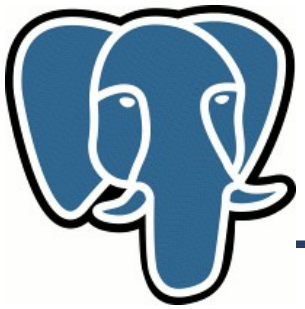
Comparison of exact (ts_stat) and approximated stats about 500,000 documents

```
=# select a.word, b.ndoc as exact, a.estimation as estimation, round
( (a.estimationb.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;
  word   | exact  | estimation | error
--------+--------+------------+-------
 page    | 340430 |     340858 | 0.13%
 figur   | 240104 |     240366 | 0.11%
 use     | 147132 |     148022 | 0.60%
 model   | 133444 |     134442 | 0.74%
 result  | 128977 |     129010 | 0.03%
(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 instert 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