How to reason about Indexing your Postgres database

pganalyze

@LukasFittl

What We'll Talk About Today

1. The Good (and Bad) of Indexes
2. How Postgres Chooses Which Index To Use
3. Four Ingredients for Making A Good Index
4. A methodology for finding "Good Enough" indexes
5. Automation with the new pganalyze Index Advisor
6. Q&A

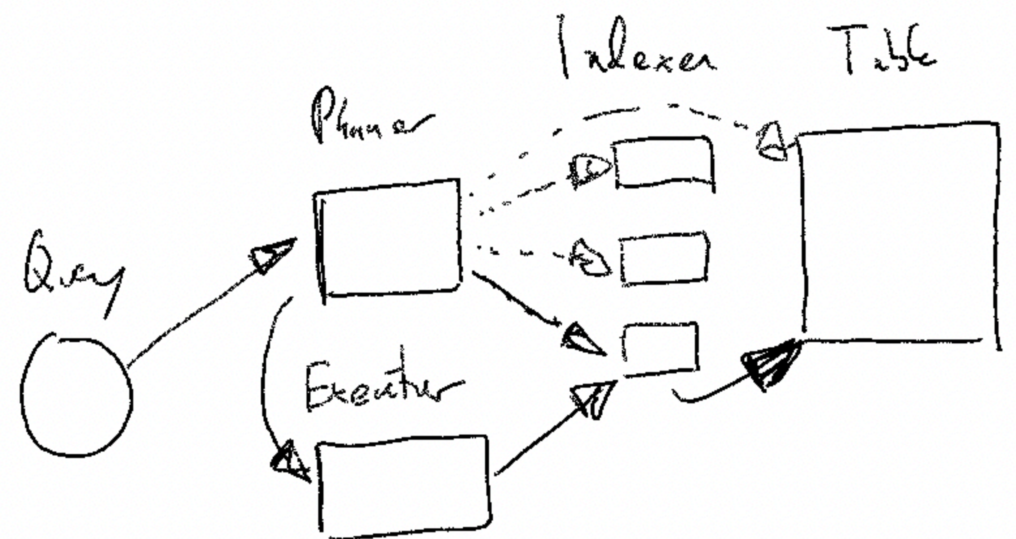
The Good (and Bad) of Indexes

Basic Facts

Indexes are a per-table caching data structure

Different index types have different structures and search algorithms

The Postgres planner decides which index to use based on its cost model



The Good

Postgres automatically updates Indexes for us

Indexes can make it faster to find relevant rows of a table

Sometimes indexes can return the data directly from the index (Index Only Scan)

The Bad

Indexes need to be updated as the data in the table is updated

Too many indexes, or overly complex indexes, will slow down your writes (and use disk space)

Adding indexes can prevent HOT (Heap-Only Tuple) updates, making writes even more expensive

How Postgres Chooses Which Index To Use

Which Query Can Use Which Existing Index?

	Query #1	Query #2	Query #3	Query #4
Existing Index A	XXX	(X)		
Existing Index B		XXX		
Existing Index C			XXX	

Legend: XXX = best index, (X) usable but not the best

Which Query Can Use Which Existing Index?

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
IDX 1	XXX	(X)									
IDX 2						XXX					
IDX 3					XXX			XXX	XXX		XXX
IDX 4		XXX							(X)	(X)	
IDX 5			XXX								

Legend: XXX = best index, (X) usable but not the best

Four Ways To Scan A Table (and/or an Index)

Sequential Scan: Read table sequentially, until all requested results are found (no index being used).

Bitmap Index/Heap Scan: Produce a bitmap of all matching table entries (using one or more indexes), and then read the table via the bitmap.

Index Scan: Find the matching data directly in one index, and go to the table to find extra columns being requested, and check MVCC visibility.

Index Only Scan: Get all data directly from the index (no data lookups from the table). In some cases use table to check MVCC visibility.

```
Bitmap Heap Scan on public.tbl  (cost=2691.86..441387.45 rows=227539 width=738) (actual …)
  Recheck Cond: ((tbl.col1 = '123'::bigint) AND (…))
  Heap Blocks: exact=414
  Buffers: shared hit=1743 read=12796 dirtied=223
  I/O Timings: read=1991.582
  →  Bitmap Index Scan on table_idx  (cost=0.00..2634.97 rows=227539 width=0) (…)
       Index Cond: (tbl.col1 = '123'::bigint)
       Buffers: shared hit=4 read=33
       I/O Timings: read=1.680
```

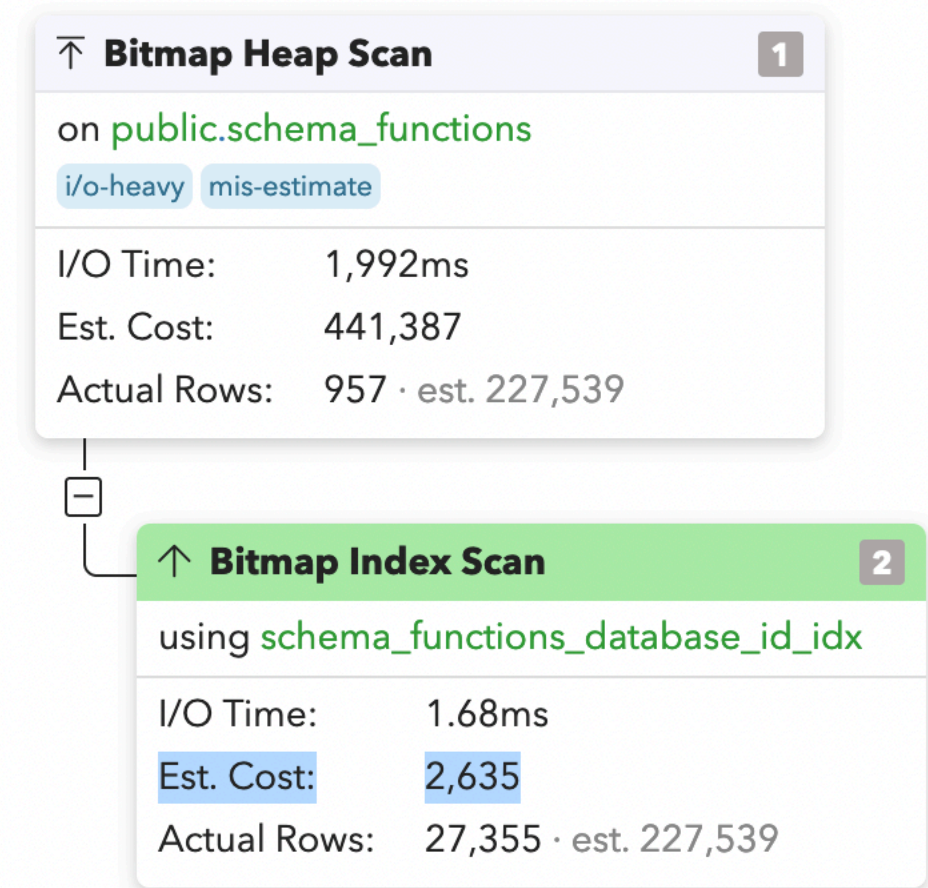
EXAMPLE

Cost

EXPLAIN plans show the cost of each plan node

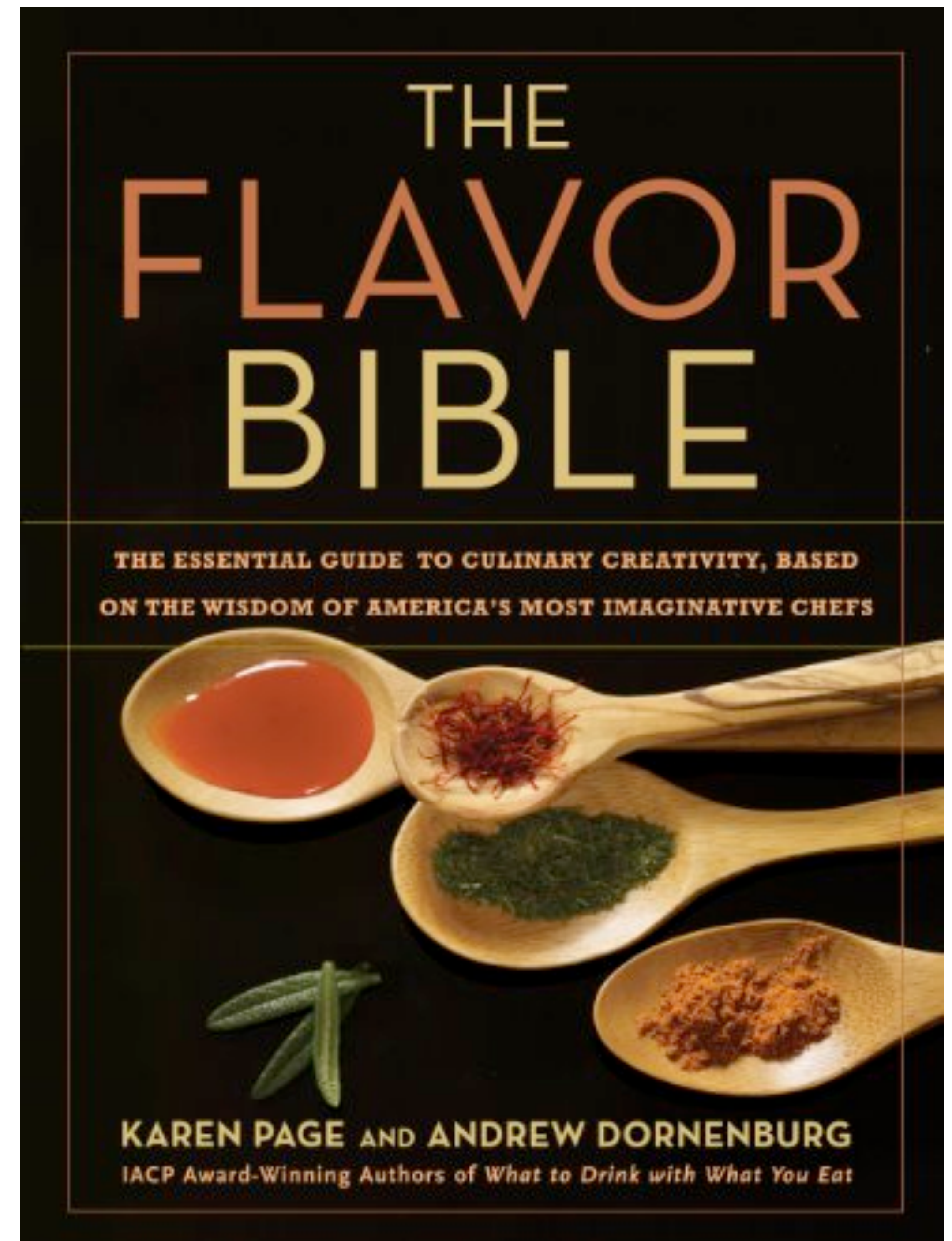
Costs are a heuristic calculated by the planner

Costs decide which index gets used (if any)



Four Ingredients for Making A Good Index

The Essential Guide
To **Indexing** Creativity



1. Breaking Down Queries Into Scans

```sql
SELECT inventory_items.id
  FROM warehouses
  JOIN inventory_items USING (warehouse_id)
 WHERE warehouse.location_state = 'CA'
       AND inventory_items.tags @> ARRAY['milk']
ORDER BY inventory_items.expires_at ASC
LIMIT 10
```

```sql
--- inventory_items scan
SELECT id
  FROM inventory_items
 WHERE warehouse_id = $n
       AND inventory_items.tags @> ARRAY['milk']
ORDER BY inventory_items.expires_at ASC
LIMIT 10;

--- warehouses scan
SELECT warehouse_id
  FROM warehouses
 WHERE warehouse_id = $n
       AND location_state = 'CA'
```

EXAMPLE

Are JOIN clauses indexable?

```
FROM warehouses
JOIN inventory_items USING (warehouse_id)
```

```
 FROM warehouses
WHERE warehouse_id = $n
```

```
 FROM inventory_items
WHERE warehouse_id = $n
```

It depends! (On the JOIN type)

Hash Join: Doesn’t need sorted output, and doesn’t use the JOIN clause to restrict the data when scanning the table

Merge Join: Needs sorted output from the scan node (thus can benefit from a sorted index like B-tree), but doesn't use the JOIN clause to restrict the data when scanning the table

Nested Loop Join: Doesn’t need sorted output from the scan node, but **for one of the two tables** uses the JOIN clause to restrict the data when scanning the table

2. Turning Scans Into Index Elements

Index Elements

```
CREATE INDEX ON users USING btree ( email ) WHERE deleted_at IS NULL
```

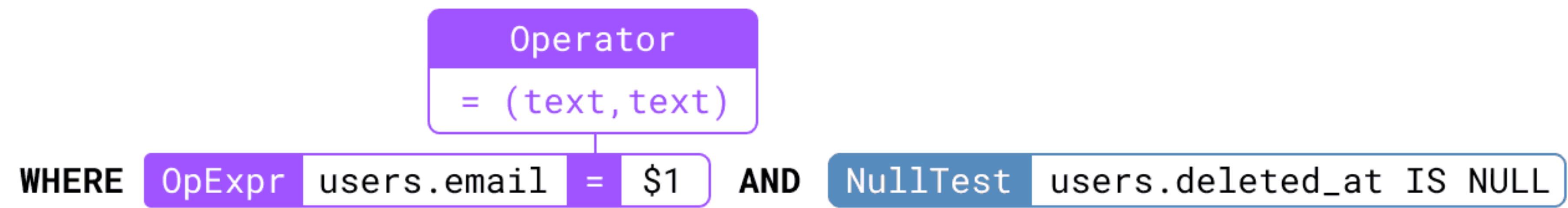
text_ops

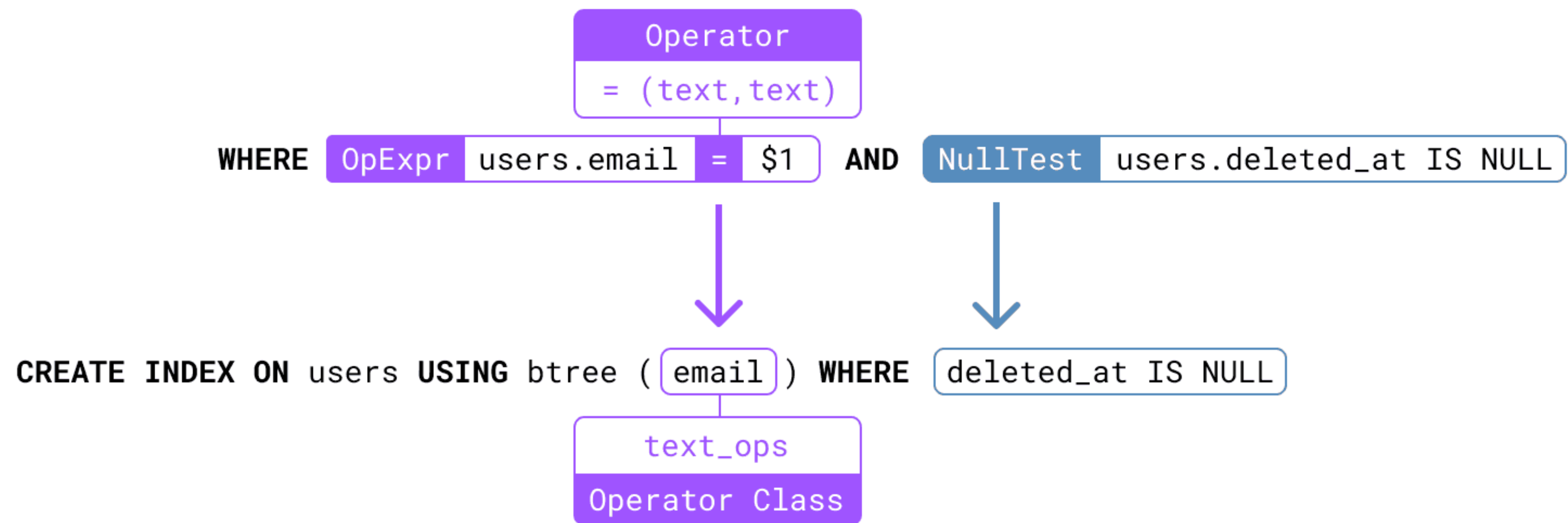
Operator Class

```c
/* Postgres source - src/include/nodes/parsenodes.h */

/*
 * IndexElem - index parameters (used in CREATE INDEX, and in ON CONFLICT)
 *
 * For a plain index attribute, 'name' is the name of the table column to
 * index, and 'expr' is NULL.  For an index expression, 'name' is NULL and
 * 'expr' is the expression tree.
 */
typedef struct IndexElem
{
    NodeTag     type;
    char       *name;           /* name of attribute to index, or NULL */
    Node       *expr;           /* expression to index, or NULL */
    char       *indexcolname;   /* name for index column; NULL = default */
    List       *collation;      /* name of collation; NIL = default */
    List       *opclass;        /* name of desired opclass; NIL = default */
    List       *opclassopts;    /* opclass-specific options, or NIL */
    SortByDir   ordering;       /* ASC/DESC/default */
    SortByNulls nulls_ordering; /* FIRST/LAST/default */
} IndexElem;
```

EXAMPLE


Operator
= (text,text)
WHERE
OpExpr
users.email
=
$1
AND
NullTest
users.deleted_at IS NULL


Operator
= (text,text)
WHERE
OpExpr
users.email
=
$1
AND
NullTest
users.deleted_at IS NULL
CREATE INDEX ON users USING btree (
email
) WHERE
deleted_at IS NULL
text_ops
Operator Class

```
// Equality operator

SELECT * FROM users WHERE email = $1;

// ⇒ B-Tree index element: email
//
// Single column index: CREATE INDEX ON users USING btree (email);


SELECT * FROM users WHERE email = $1 AND organization_id = $2;

// ⇒ B-Tree index element: email
// ⇒ B-Tree index element: organization_id
//
// Multi-column index: CREATE INDEX ON users USING btree (organization_id, email);



// Range overlap operator (&&)

SELECT * FROM timeranges WHERE seen_at_range && tstzrange($2, $3);

// ⇒ GIST index element: seen_at_range
//
// Single column index: CREATE INDEX ON timeranges USING gist (seen_at_range);
```

EXAMPLE

Indexing ORDER BY + LIMIT

B-Tree indexes are sorted

Order matters - trailing columns should match ORDER BY

Index lookup can keep the sort order

Avoids an explicit Sort step at runtime

No need to load all matching rows

When a LIMIT is present and the ORDER BY matches, it limits how much data is retrieved from the index

Additional considerations

Partial indexes

Expression indexes

INCLUDE index columns

3. Generating Multi-Column Index Combinations

„Multicolumn indexes should be used ***sparingly****. In most situations, an index on a single column is sufficient and saves space and time."*

– POSTGRES DOCUMENTATION ON MULTICOLUMN INDEXES

Multi-Column for B-Tree

Lead with index elements referenced in a clause through an equality operator (e.g. "col = $1"), followed by all other elements.

Within that set of elements, sort by selectivity (more selective parameters are ordered first).

Other Index Types

GIST: Sort elements by number of distinct values in the table (more distinct elements are ordered first)

GIN: Indexed columns have equal weight (but its often too expensive to index multiple columns)

BRIN: Indexed columns have equal weight

Hash: Multi-column indexes not supported

4. Modeling I/O Overhead

Index Size

For a given index, we can estimate its hypothetical size with heuristics

Estimation based on the data types used and the number of index entries

Similar to how bloat estimation queries show you the expected size of an existing index

```
/* ──────────────────────────────
 * quick estimating of index size.
 * Each B-tree index tuple contains:
 *
 * - sizeof(ItemIdData): 4 (after page header
 * - sizeof(IndexTupleData): 8 (index tuple h
 * - actual data size based on each key's ave
 *
 * For this estimation it is assumed all
 * values are not NULL.
 *
 * Additionally, the following data is
 * present once in each page:
 *
 * - sizeof(PageHeader) : 24
 * - sizeof(BTPageOpaqueData): 16
 *
 * for calculating fill of index pages this u
 *
 * fillfactor parameter, or default fillfacto
 * fixed additional bloat: 20%
 */
line_size = ind_avg_width +
MAXALIGN(sizeof(IndexTupleData)) +
sizeof(ItemIdData);

usable_page_size = BLCKSZ -
SizeOfPageHeaderData -
sizeof(BTPageOpaqueData);

bloat_factor = (200.0
 - (fillfactor == 0 ? BTREE_DEFAULT_FILLFACTO
 + additional_bloat) / 100;

entry->pages = (BlockNumber)
(entry->tuples * line_size *
bloat_factor / usable_page_size);
```

Table Writes

INSERT and UPDATE statements involving the table you're indexing will be slowed down

If its a very busy, write-heavy table, add new indexes carefully

```
SELECT relname, n_tup_ins + n_tup_upd
  FROM pg_stat_user_tables
 ORDER BY 2 DESC;

          relname          | ?column?
---------------------------+------------
 snapshot_benchmarks       | 2612081919
 snapshots                 | 2100837321
 queries                   |  575963537
 schema_indices            |  480640469
 schema_columns            |  470930261
 query_stats_35d_20220614  |  445560822
 query_stats_35d_20220609  |  443129148
 query_stats_35d_20220608  |  441617161
 query_stats_35d_20220601  |  441287771
 query_stats_35d_20220531  |  440735110
 query_stats_35d_20220607  |  440513863
 query_stats_35d_20220602  |  438787367
 query_stats_35d_20220525  |  436720065
 query_stats_35d_20220524  |  435440884
 query_stats_35d_20220613  |  435251212
 query_stats_35d_20220526  |  434236159
 query_stats_35d_20220606  |  434046609
 query_stats_35d_20220610  |  432917410
 query_stats_35d_20220527  |  431926732
 query_stats_35d_20220523  |  430703041
 query_stats_35d_20220603  |  430184241
 query_stats_35d_20220519  |  423875277
 query_stats_35d_20220518  |  421547720
 query_stats_35d_20220517  |  420719082
 query_stats_35d_20220512  |  419632624
 query_stats_35d_20220520  |  418601716
 query_stats_35d_20220530  |  418508256
 query_stats_35d_20220511  |  418450087
 query_stats_35d_20220513  |  413823246
 query_stats_35d_20220516  |  413656384
 query_stats_35d_20220615  |  410649512
 query_stats_35d_20220604  |  389391391
```

Index Write Overhead

Complex multi-column indexes are more expensive

Indexing text columns is more expensive than indexing integer columns

What if we could quantify the overhead on write activity, instead of just index size?

Indexes

Total Index Write Overhead	0.75
Definition	**Index Write Overhead**
`btree (id)`	0.03
`btree (database_id, severity) WHERE (state <> 2)`	0.00
`btree (reference_type, reference_id)`	0.10
`btree (server_id, "check", grouping_key) WHERE (state <> 2)`	0.02
`btree (server_id)`	0.06
`btree (server_id, "check")`	0.14
`btree (database_id, "check")`	0.12
`btree (organization_id, "check")`	0.14
`btree (server_id, severity) WHERE (state <> 2)`	0.01
`btree ("check")`	0.10
`btree (database_id)`	0.04

```
CREATE TABLE customers (
  id bigserial PRIMARY KEY,
  name text,
  organization_id bigint
);

CREATE INDEX customer_organization_id_idx ON customers (organization_id);


# Index write overhead = index_entry_size / table_row_size

table_row_size = \
  23 + # Heap tuple header
  4 + # Heap item ID
  4 + # size of bigserial
  50 + # avg_width of text
  4 # size of bigint
⇒ 85 # bytes

index_entry_size = \
  8 + # Index entry header
  4 # size of bigint
⇒ 12 # bytes


# Index write overhead = 0.14 (12.0 / 85.0)
#
# “for 1 byte of table data, we will write 0.14 bytes to the index”
```

EXAMPLE

A methodology for finding "Good Enough" indexes

What New Index Could Improve Our Query Performance?

	Scan #1	Scan #2	Scan #3	Scan #4
Existing Index A	XXX	(X)		
Existing Index B		XXX		
Existing Index C			XXX	
New Index	10x faster			50x faster

Legend: XXX = best index, (X) usable but not the best

“What If?” Analysis

We could actually create the index.

But that's expensive for many index ideas.

Are there any alternatives?

Using HypoPG for Hypothetical Indexes

Postgres extension that’s supported on most cloud platforms

HypoPG doesn’t actually create the index

Estimates its theoretical size and tricks the planner into providing an EXPLAIN

```
CREATE EXTENSION hypopg;

SELECT * FROM hypopg_create_index('CREATE INDEX ON hypo (id)') ;
 indexrelid |      indexname
------------+----------------------
      18284 | <18284>btree_hypo_id
(1 row)


EXPLAIN SELECT val FROM hypo WHERE id = 1;
                                    QUERY PLAN
-----------------------------------------------------------------------------------
 Index Scan using <18284>btree_hypo_id on hypo  (cost=0.04..8.06 rows=1 width=10)
   Index Cond: (id = 1)
(2 rows)


SELECT indexname, pg_size_pretty(hypopg_relation_size(indexrelid))
  FROM hypopg_list_indexes ;
      indexname       | pg_size_pretty
----------------------+----------------
 <18284>btree_hypo_id | 2544 kB
(1 row)
```

EXAMPLE

Indexing as a Set Covering Problem

Which Index Should We Choose?

	Scan #1 (col1=$1)	Scan #2 (col1=$1 AND col2=$2)	Scan #3 (col2=$1)	Scan #4 (col3=$1)
IDX1 (col1) IWO=0.1	15x faster	2x faster		
IDX2 (col1, col2) IWO=0.2	3x faster	10x faster	3x faster	
IDX3 (col2) IWO=0.1		9x faster	3x faster	
IDX4 (col3) IWO=0.1				50x faster

Greedy Set Covering Algorithm

First, take the set that covers the most elements

Then, take the set that covers the most remaining elements

Keep going until all elements are covered

	Q1	Q2	Q3	Q4
IDX 1	X	X		
IDX 2		X		
IDX 3			X	
NEW IDX	10x faster			50x faster

Hypothetical Index + Greedy SCP = problem solved?

	Scan #1 (col1=$1)	**Scan #2** (col1=$1 AND col2=$2)	**Scan #3** (col2=$1)	**Scan #4** (col3=$1)
IDX1 (col1) IWO=0.1	15x faster	2x faster		
IDX2 (col1, col2) IWO=0.2	3x faster	10x faster	3x faster	
IDX3 (col2) IWO=0.1		9x faster	3x faster	
IDX4 (col3) IWO=0.1				50x faster

Hypothetical Index + Greedy SCP = problem solved?

	Scan #1 (col1=$1)	**Scan #2** (col1=$1 AND col2=$2)	**Scan #3** (col2=$1)	**Scan #4** (col3=$1)
IDX1 (col1) IWO=0.1	15x faster	2x faster		
IDX2 (col1, col2) IWO=0.2	3x faster	10x faster	3x faster	
IDX3 (col2) IWO=0.1		9x faster	3x faster	
IDX4 (col3) IWO=0.1				50x faster

Did we miss out on a faster index? Yes, we did.

	Scan #1 (col1=$1)	Scan #2 (col1=$1 AND col2=$2)	Scan #3 (col2=$1)	Scan #4 (col3=$1)
IDX1 (col1) IWO=0.1	15x faster	2x faster		
IDX2 (col1, col2) IWO=0.2	3x faster	10x faster	3x faster	
IDX3 (col2) IWO=0.1		9x faster	3x faster	
IDX4 (col3) IWO=0.1				50x faster

Eliminate Bad Index Options First

If a potential index is **50%** worse (or more), eliminate that scan/index combination from consideration **before** running the Greedy SCP algorithm.

Avoids overly broad multi-column indexes.

The Refined Solution

	Scan #1 (col1=$1)	**Scan #2** (col1=$1 AND col2=$2)	**Scan #3** (col2=$1)	**Scan #4** (col3=$1)
IDX1 (col1) IWO=0.1	15x faster	~~2x faster~~		
IDX2 (col1, col2) IWO=0.2	~~3x faster~~	10x faster	3x faster	
IDX3 (col2) IWO=0.1		9x faster	3x faster	
IDX4 (col3) IWO=0.1				50x faster

The Refined Solution

	Scan #1 (col1=$1)	Scan #2 (col1=$1 AND col2=$2)	Scan #3 (col2=$1)	Scan #4 (col3=$1)
IDX1 (col1) IWO=0.1	15x faster	~~2x faster~~		
IDX2 (col1, col2) IWO=0.2	~~3x faster~~	10x faster	3x faster	
IDX3 (col2) IWO=0.1		9x faster	3x faster	
IDX4 (col3) IWO=0.1				50x faster

Both Indexes Cover 2 Scans - Which To Pick?

Index Write Overhead as a Tie Breaker

If two index options cover the same number of queries, optimize for **lowest total Index Write Overhead.**

The Refined Solution

	Scan #1 (col1=$1)	Scan #2 (col1=$1 AND col2=$2)	Scan #3 (col2=$1)	Scan #4 (col3=$1)
IDX1 (col1) IWO=0.1	15x faster	~~2x faster~~		
IDX2 (col1, col2) IWO=0.2	~~3x faster~~	10x faster	3x faster	
IDX3 (col2) IWO=0.1		9x faster	3x faster	
IDX4 (col3) IWO=0.1				50x faster

The Refined Solution

	Scan #1 (col1=$1)	Scan #2 (col1=$1 AND col2=$2)	Scan #3 (col2=$1)	Scan #4 (col3=$1)
IDX1 (col1) IWO=0.1	15x faster	~~2x faster~~		
IDX2 (col1, col2) IWO=0.2	~~3x faster~~	10x faster	3x faster	
IDX3 (col2) IWO=0.1		9x faster	3x faster	
IDX4 (col3) IWO=0.1				50x faster

The Refined Solution

	Scan #1 (col1=$1)	Scan #2 (col1=$1 AND col2=$2)	Scan #3 (col2=$1)	Scan #4 (col3=$1)
IDX1 (col1) IWO=0.1	15x faster	~~2x faster~~		
IDX2 (col1, col2) IWO=0.2	~~3x faster~~	10x faster	3x faster	
IDX3 (col2) IWO=0.1		9x faster	3x faster	
IDX4 (col3) IWO=0.1				50x faster

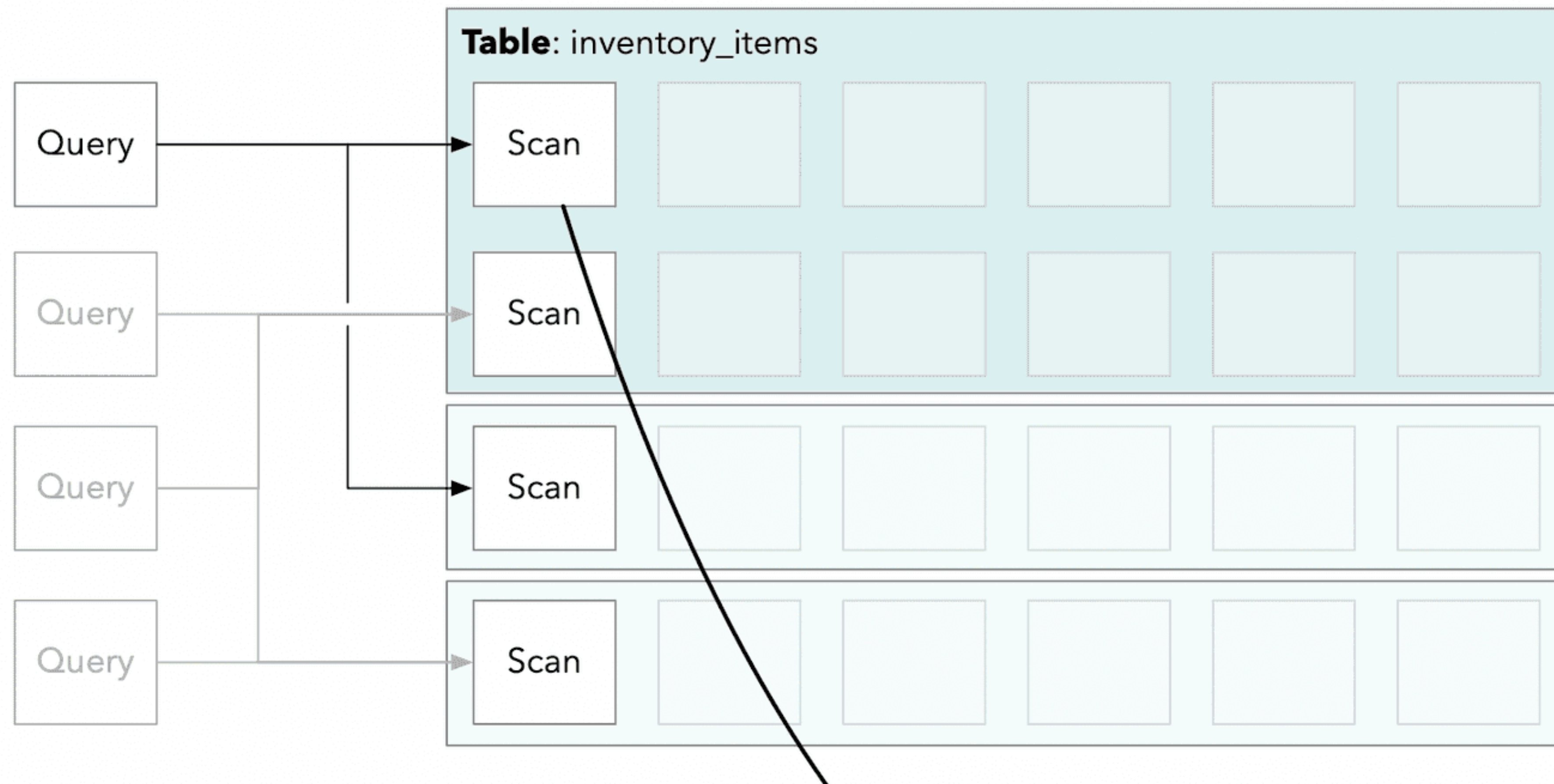
This can get very complex, very quickly.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
IDX1	1x					2x								
IDX2		10x	3x											
IDX3		9x	3x											
IDX4				50x										
IDX5					1x									
IDX6					2x									
IDX7								1x						
IDX8							4x	3x						
IDX9							5x							
IDX10						5x								
IDX11									90x					
IDX12										50x	100x	3x		
IDX13										60x	110x	2x		
IDX14													12x	36x

Automation with the new pganalyze Index Advisor

Phase 1: Query Analysis

Phase 1: Query Analysis



pganalyze

```
SELECT query, calls, mean_exec_time FROM pg_stat_statements LIMIT 1;

-[ RECORD 1 ]--
+-----------------------------------------------------------------------------------------------

query          | SELECT "replication_stats".* FROM "replication_stats" WHERE "replication_stats"."server_id" = $1
AND "replication_stats"."collected_at" BETWEEN $2 AND $3
calls          | 11
mean_exec_time | 493.72526418181815
```

```
{
    "Relation Name": "replication_stats",
    "Namespace Name": "public",
    "Scans": [
        {
            "Scan ID": "00000000-0000-0000-0000-000000000001",
            "Restriction Clauses": [
                "(collected_at \u003e= $2)",
                "(collected_at \u003c= $3)",
                "(server_id = $1)"
            ],
            "Join Clauses": [],
            "Estimated Scans Per Minute": 0.0031746031742514906
        }
    ]
}
```

EXAMPLE

SCREENSHOT

pganalyze

ORGANIZATION
pganalyze

- Dashboard
- Query Performance
- Index Advisor
- EXPLAIN Plans
- Schema Statistics
- Log Insights
- Connections
- VACUUM Activity
- Config Tuning
- System
- Alerts & Check-Up
- Settings

Server
prod-db-main

Database
pgaweb

Last 24 hours

SELECT ... FROM servers s JOIN organizations o USING (organization_id) WHERE s.id = $1

Query #43844977 · Fingerprint: 0a3dd78dc59bae3a · Role: pgaweb_workers

Avg Time
0.02ms

Calls Per Minute
10,928.46 / min

Compare to 7 days ago

Overview | NEW Index Advisor | Query Samples 0 | EXPLAIN Plans 0 | Query Tags 0 | Log Entries 0

SQL Statement

```
SELECT s.deleted_at AT TIME ZONE $2, s.disabled_at AT TIME ZONE $3, s.integrated_log_insights, s.integrated_explain, o.organization_id, o.sl
ug, o.name, o.subscription_plan_id, o.subscription_status, o.billing_addons
  FROM servers s JOIN organizations o USING (organization_id)
 WHERE s.id = $1
```

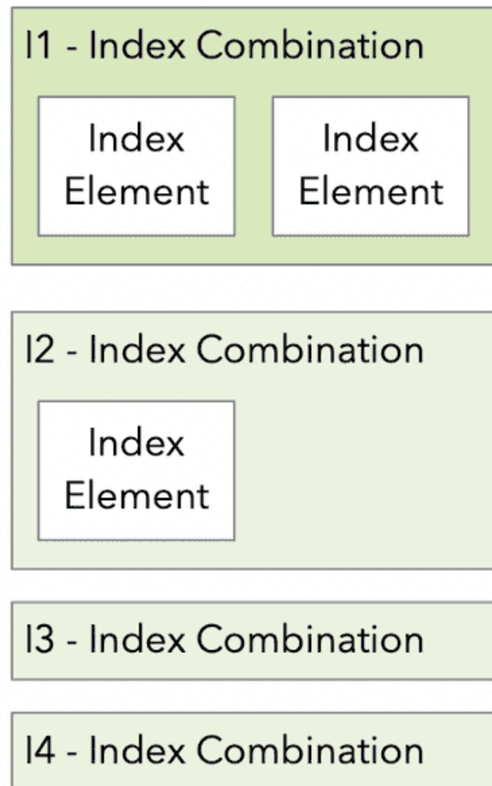
Hide full query text

Scans

TABLE	SCAN EXPRESSION		SCAN METHOD	COST	TABLE SIZE
public.servers	(id = $n) AND (organization_id = $n)		Index Scan using serve...	8.00	115.4 MB
	WHERE clause	(id = $n)			
	JOIN clause	(organization_id = $n)			
public.organizations	(organization_id = $n)		Bitmap Heap Scan usin...	4.00	3.5 MB
	WHERE clause	-			
	JOIN clause	(organization_id = $n)			

Phase 2: Index Selection

Phase 2: Index Selection



"What If?" Analysis

"Good Enough" Index Selection

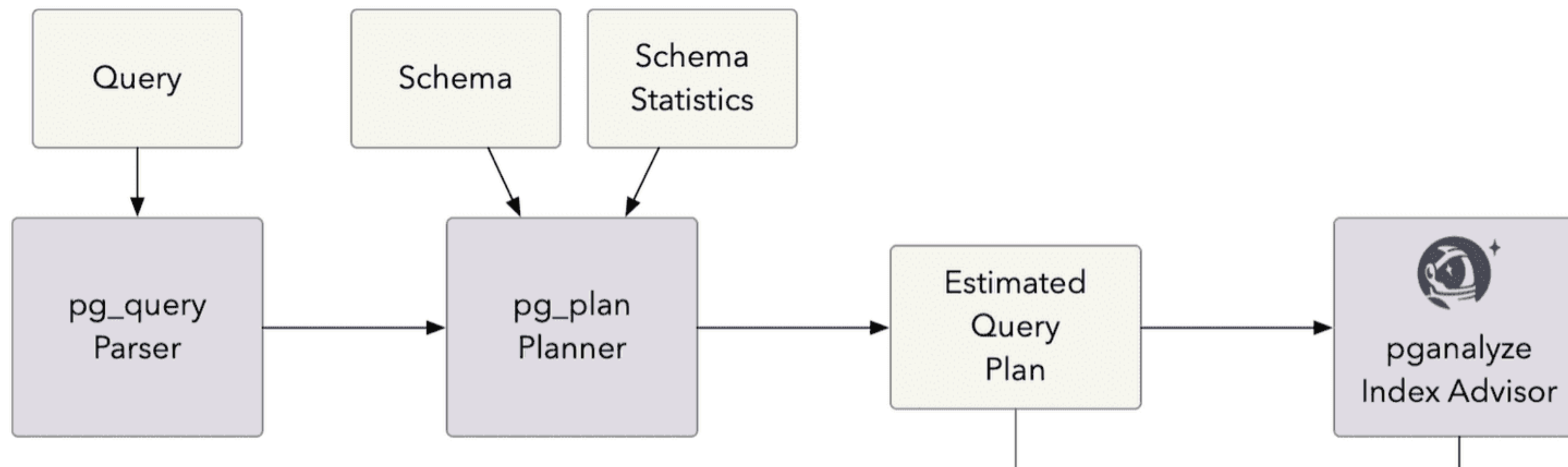
	S1	**S2**	S3	S4
I1	42.0			
I2	**30.0**	**5.0**		
I3		2.5		
I4			3.0	8.2

Missing Index **I2** covers **S1** and **S2**

```
CREATE INDEX ON inventory_items(warehouse_id);
```

pganalyze

Out-of-band "What If?" analysis



"How we deconstructed the Postgres planner to find indexing opportunities"

https://pganalyze.com/blog/deconstructing-the-postgres-planner

pganalyze

```
{
    "Relation Name": "replication_stats",
    "Namespace Name": "public",
    "Scans": [
        {
            "Scan ID": "00000000-0000-0000-0000-000000000001",
            "Restriction Clauses": [
                "(collected_at \u003e= $2)",
                "(collected_at \u003c= $3)",
                "(server_id = $1)"
            ],
            "Join Clauses": [],
            "Estimated Scans Per Minute": 0.0031746031742514906
        }
    ]
}
```

↓

```
{
  "Missing Indexes": [
    {
      "Index": {
        "Type": "btree",
        "Hypothetical": true,
        "Size Bytes": 1602707456,
        "Definition": "CREATE INDEX CONCURRENTLY ON public.replication_stats USING btree (server_id, collected_at)"
      },
      "Index Write Overhead": 0.18,
      "Scans": [
        {
          "Scan ID": "00000000-0000-0000-0000-000000000001",
          "Old Cost": 2205286.4000,
          "New Cost": 5051.2556
        }
      ]
    }
  ],
  "Failed Scans": []
}
```

EXAMPLE

Scans

```sql
S1 = SELECT * FROM public.replication_stats WHERE (collected_at ≥ $2) AND (collected_at ≤ $3) AND (server_id = $1)
```

Existing indexes

```sql
X1 = CREATE UNIQUE INDEX replication_stats_pkey ON public.replication_stats USING btree (id)
```

Possible new index options

```sql
I1 = CREATE INDEX ideal_index_1 ON public.replication_stats USING btree (collected_at)
I2 = CREATE INDEX ideal_index_2 ON public.replication_stats USING btree (server_id)
I3 = CREATE INDEX ideal_index_3 ON public.replication_stats USING btree (server_id, collected_at)
```

Scan coverage for possible new indexes

```
 I1  skipped because it is never considered a "good enough" index (within 50% of best index)
 I2  skipped because it is never considered a "good enough" index (within 50% of best index)
 I3  covers 1 scans (S1, ) with IWO 0.18
```

Running greedy set cover algorithm to choose best indexes

1. I3 (best choice for 1 remaining scans to be covered)

Summary

	S1
X1	
I1 (IWO = 0.10)	2.3
I2 (IWO = 0.14)	2.3
I3 (IWO = 0.18)	436.6

EXAMPLE

SCREENSHOT

pganalyze

ORGANIZATION
pganalyze

- Dashboard
- Query Performance
- Index Advisor
- EXPLAIN Plans
- Schema Statistics
- Log Insights
- Connections
- VACUUM Activity
- Config Tuning
- System
- Alerts & Check-Up
- Settings

Server
prod-db-main

Database
pgaweb

Index Advisor

Overview | Missing Indexes | Unused Indexes

Total Data Size	Total Index Size	Table Writes	Avg. Index Write Overhe...
5.8 TB	3.7 TB	814,306	0.55
▾ 1 TB		per minute	index bytes per table byte

Opportunities for Database (34)

Search...

IMPACT	TABLE	INDEX	QUERIES	TABLE SIZE	TABLE WRITES / MIN	INDEX WRITE OVERHEAD
	public.alert_policy_settings	btree (alert_policy_id, check_name)	2	824 kB	0.011	+0.59
	public.databases	btree (datname)	1	39.6 MB	0.578	+0.18
	public.integration_configs	btree (server_id)	5	208 kB	0.000	+0.07
	public.integration_configs	btree (organization_integration_id, or…	5	208 kB	0.000	+0.11
	public.issues	btree (database_id, created_at)	1	4.1 GB	22.667	+0.07
	public.issues	btree (organization_id, "check", updat…	1	4.1 GB	22.667	+0.17
	public.issues	btree (database_id, server_id, organiz…	8	4.1 GB	22.667	+0.13
	public.issues	btree (organization_id, server_id)	6	4.1 GB	22.667	+0.11
	public.issues	btree (organization_id, "check", state)	2	4.1 GB	22.667	+0.16
	public.issues	btree ("check", server_id, updated_at)	8	4.1 GB	22.667	+0.17
	public.organization_memb…	btree (alert_policy_id)	1	2.7 MB	0.033	+0.14
	public.organization_memb…	btree (invited_by_id, accepted)	1	2.7 MB	0.033	+0.09
	public.organization_memb…	btree (organization_id)	23	2.7 MB	0.033	[illegible]

SCREENSHOT

pganalyze

ORGANIZATION
pganalyze

- Dashboard
- Query Performance
- Index Advisor
- EXPLAIN Plans
- Schema Statistics
- Log Insights
- Connections
- VACUUM Activity
- Config Tuning
- System
- Alerts & Check-Up
- Settings

Server
prod-db-main

Database
pgaweb

Issue #11494: Missing Index

Overview

Severity
Info

Check Frequency
Daily

Last Updated
2022-06-14 07:04:55pm PDT

State
Triggered

Acknowledge

Description
Missing index on public.alert_policy_settings **btree (alert_policy_id, check_name)**

Guidance

```
CREATE INDEX CONCURRENTLY ON public.alert_policy_settings USING btree (alert_poli
cy_id, check_name);
```

Copy CREATE INDEX command

We recommend testing opportunities in pre-production or staging environments first before deploying changes to production. If possible, it is advisable to use a copy of the production database for your tests, otherwise you may not see a representative performance improvement or query plan change.

› How to test the opportunity

Cost Improvement

Weighted Improvement	26×
Index Write Overhead	+0.59

Scans (2)

Search...

SCAN EXPRESSION	COST	EST. SCANS / MIN	COST IMPROVEME...
› (check_name = $n) AND (alert_policy_id = $n)	220.44	0.5669	27×
› (alert_policy_id = $n)	200.86	0.0080	11×

Affected Queries (2)

Search...

QUERY ID	QUERY	AVG TIME (MS)	CALLS / MIN	% ALL RUNTIME
#4383...	SELECT alert_policy_settings.* FROM alert_policy_settings WHERE ... LIMI...	0.59ms	0.5669	0.00%

SCREENSHOT

138 lines (127 sloc) | 15.3 KB

Raw Blame

Running greedy set cover algorithm to choose best indexes

1. I12 (best choice for 8 remaining scans to be covered)
2. I21 (best choice for 3 remaining scans to be covered)
3. I9 (best choice for 1 remaining scans to be covered)
4. I3 (best choice for 1 remaining scans to be covered)

Summary

	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15	S16	S17
X1			XXX				XXX						XXX			XXX	XXX
X2																	
X3				XXX					XXX								
X4			(x)	(x)	XXX		(x)	(x)		(x)	XXX	XXX		XXX	(x)	(x)	(x)
X5																	
I1 (IWO = 0.07)																	
I2 (IWO = 0.11)	6.1																
I3 (IWO = 0.14)								68.1									
I4 (IWO = 0.15)								68.1									
I5 (IWO = 0.10)	7.2																
I6 (IWO = 0.23)																	
I7 (IWO = 0.31)																	
I8 (IWO = 0.08)	33.6																

thanks!

Learn more about the pganalyze Index Advisor

PGANALYZE.COM/POSTGRES-INDEX-ADVISOR

Get a free trial of pganalyze

PGANALYZE.COM

Get free pganalyze eBooks and Postgres blog posts

PGANALYZE.COM/RESOURCES **PGANALYZE.COM/BLOG**

We will send you an email with a recording of this webinar tomorrow!

Feel free to get in touch with us at pganalyze.com/contact