Automating Index Selection Using Constraint Programming

PGDay Chicago 2024

Lukas Fittl

Agenda

1. Background on Index Selection
2. A Constraint Programming Model for Index Selection
3. Utilizing the Index Selection Model in Practice
4. Advanced Use Cases
   (Per-Scan Rules, HOT Updates, Consolidation)

pganalyze

Background on Index Selection

The Index Selection Problem

We want to select which indexes to create on a table, so that:

- Queries are fast
- Write overhead is kept low

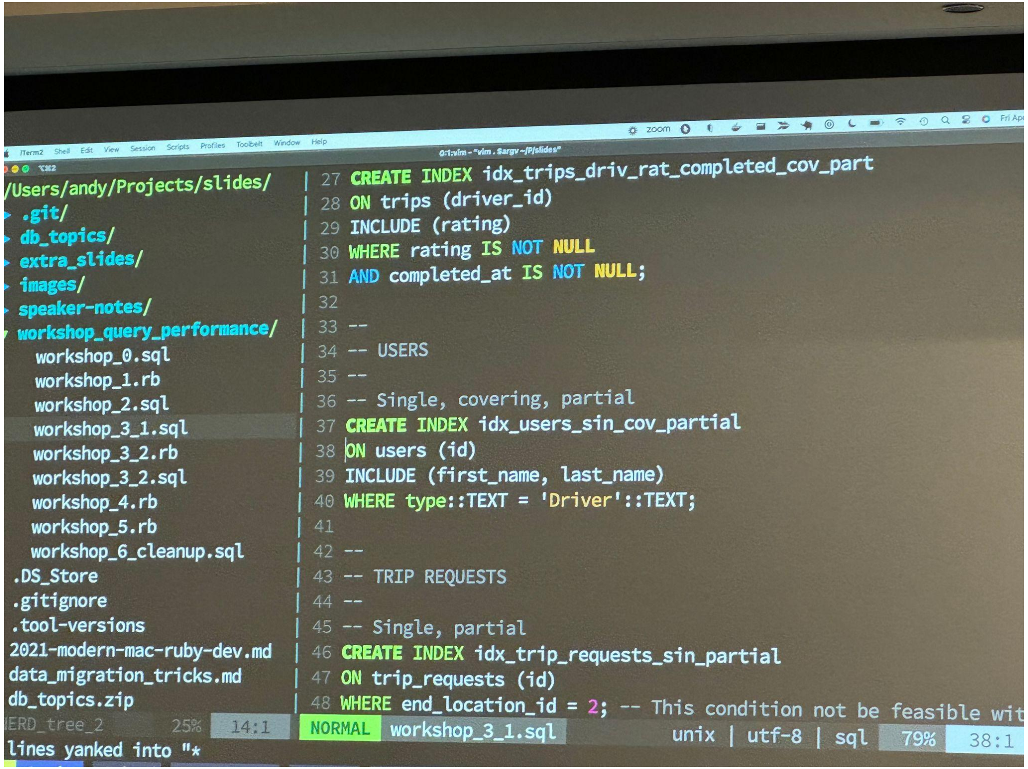
Which indexes should we select?

Research Background

- An Optimization Problem on the Selection of Secondary Keys (Lum & Ling, 1971)
- Index Selection in Relational Databases (Whang, 1987)
- CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads (Dash et al., 2011)
- Dexter -- The Automatic Indexer for Postgres (Kane, 2017)
- An Experimental Evaluation of Index Selection Algorithms (Kossmann et al., 2020)

“Let’s create different indexes & try them out”

1. Run EXPLAIN (ANALYZE, BUFFERS)
2. Read the EXPLAIN output and come up with ideas
3. CREATE INDEX



“Let’s pick some indexes that seem right”

\di index_issues*

```
 public | index_issues_on_check
 public | index_issues_on_database_id
 public | index_issues_on_database_id_and_check
 public | index_issues_on_database_id_and_severity
 public | index_issues_on_organization_id_and_check
 public | index_issues_on_reference_type_and_reference_id
 public | index_issues_on_server_id
 public | index_issues_on_server_id_and_check
 public | index_issues_on_server_id_and_check_and_grouping_key
```

Hypothetical Indexes & HypoPG

The **HypoPG extension** lets us ask “What would be the estimated cost of this query, if this index existed?”, without having to create that index.

In the simplest approach to solving index selection, we could:

- Find all columns a query filters by
- Come up with possible indexes based on the columns
- Run each possible index through HypoPG
- Select the index with the lowest cost

Hypothetical Indexes & HypoPG

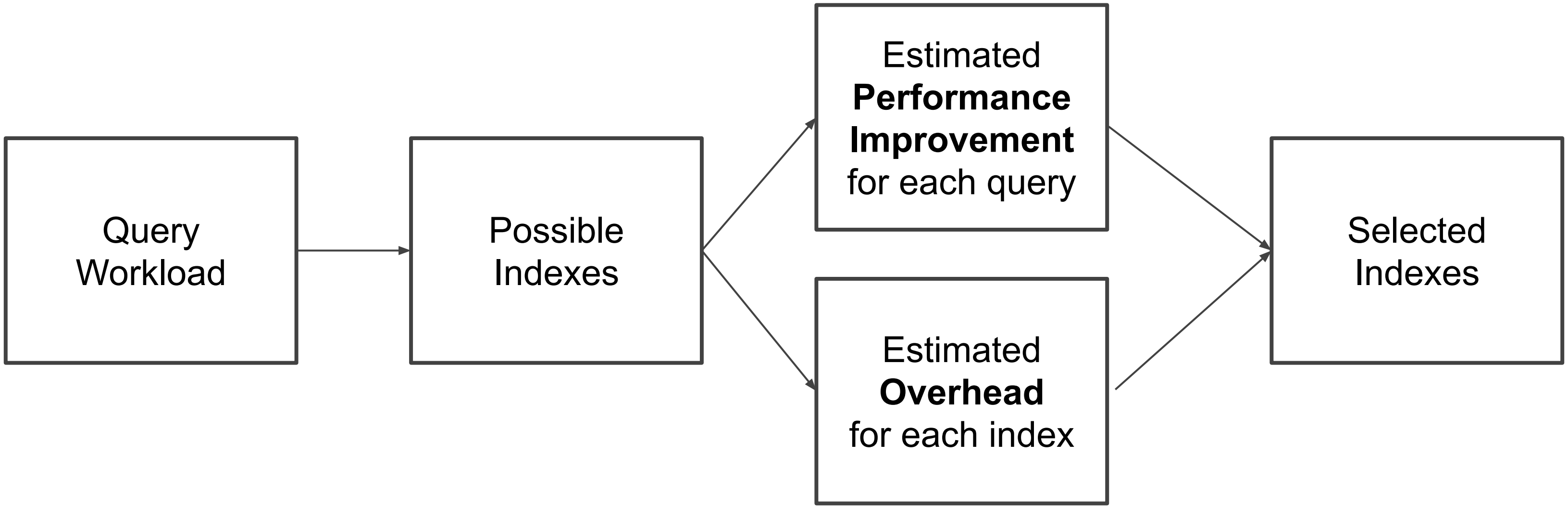
But…

How to create indexes for a **whole workload**, not just a single query?

Which multi-column indexes make sense to cover multiple queries?

How can we avoid badly slowing down writes with too many indexes?

The Index Selection Problem



Queries, Scans and Costs

- Make it easier to reason about complex queries, split them up into scans by table (**scan = Index Scan using idx** on table tbl)
- For each table, and each scan:
  - Get sequential scan cost (tiny tables don’t need indexes!)
  - Get existing index scan costs
  - Get possible index scan costs

Queries, Scans and Costs

- Use Postgres planner **costs** to estimate performance improvement (they are cheap to calculate for hypothetical indexes using HypoPG)

- “**Costs are arbitrary units.** They do not represent milliseconds or any other unit of time. Instead, **they are anchored to a single read of a sequential page, which costs 1.0 unit.**”
  - Tadeáš Peták, paraphrasing the Postgres documentation

Splitting up queries into scans

```
WITH slow_queries AS (
 SELECT qs.database_id, qs.fingerprint, qs.postgres_role_id, SUM(qs.total_time) / SUM(qs.calls) AS avg_time,
SUM(qs.shared_blks_read) / SUM(qs.calls) AS avg_blks_loaded, SUM(qs.calls) AS total_calls
   FROM query_stats_3dd qs
  WHERE qs.database_id IN (
   SELECT id FROM databases
    WHERE server_id = $4 AND NOT hidden
 ) AND qs.collected_at >= $5
  GROUP BY 1, 2, 3 HAVING SUM(qs.calls) > $6 AND SUM(qs.total_time) / SUM(qs.calls) > $7
)
SELECT q.id, (
 SELECT MAX(runtime_ms) FROM query_samples_7d qs
  WHERE qs.database_id = qfp.database_id AND qs.query_fingerprint = qfp.fingerprint AND qs.postgres_role_id =
qfp.postgres_role_id AND qs.occurred_at >= $1
) AS max_time
 FROM slow_queries JOIN query_fingerprints qfp USING (database_id, fingerprint, postgres_role_id) JOIN queries q
ON (qfp.query_id = q.id)
WHERE q.statement_types && ARRAY[$2,$3]
```

Splitting up queries into scans

pganalyze

public.databases	(NOT hidden) AND (server_id = $n) AND (id = $n) WHERE clause: (NOT hidden) AND (server_id = $n) JOIN clause: (id = $n)	Bitmap Heap Scan
public.queries	((statement_types && (ARRAY[$n])::text[]) OR (statement_types && … WHERE clause: ((statement_types && (ARRAY[$n])::text[]) OR (statement_types && (ARRAY[$n])::text[])) JOIN clause: (id = $n)	Bitmap Heap Scan
public.query_samples_7d	(occurred_at >= $n) AND (database_id = $n) AND (query_fingerprint… WHERE clause: (occurred_at >= $n) AND (database_id = $n) AND (query_fingerprint = $n) AND (postgres_role_id = $n) JOIN clause: -	Append
public.query_stats_35d	(collected_at >= $n) AND (database_id = $n) WHERE clause: (collected_at >= $n) JOIN clause: (database_id = $n)	Seq Scan

Estimated Overhead for each index

How to we measure the fact that each index has a cost?

Historically, approaches have used estimated **storage size** of a given index (e.g. as calculated by HypoPG in the case of Postgres).

However, in practice, and especially in the cloud, **I/Os** are often more expensive and problematic, than storage space.

Our Approach - Index Write Overhead (IWO)

Index Write Overhead = the estimated size of an index write (in bytes), based on the index definition, divided by the size of the average table row.

table

- col1 text, avg_width = 30 bytes
- col2 bigint, avg_width = 8 bytes
- col3 uuid, avg_width = 16 bytes

avg row size = 54 bytes

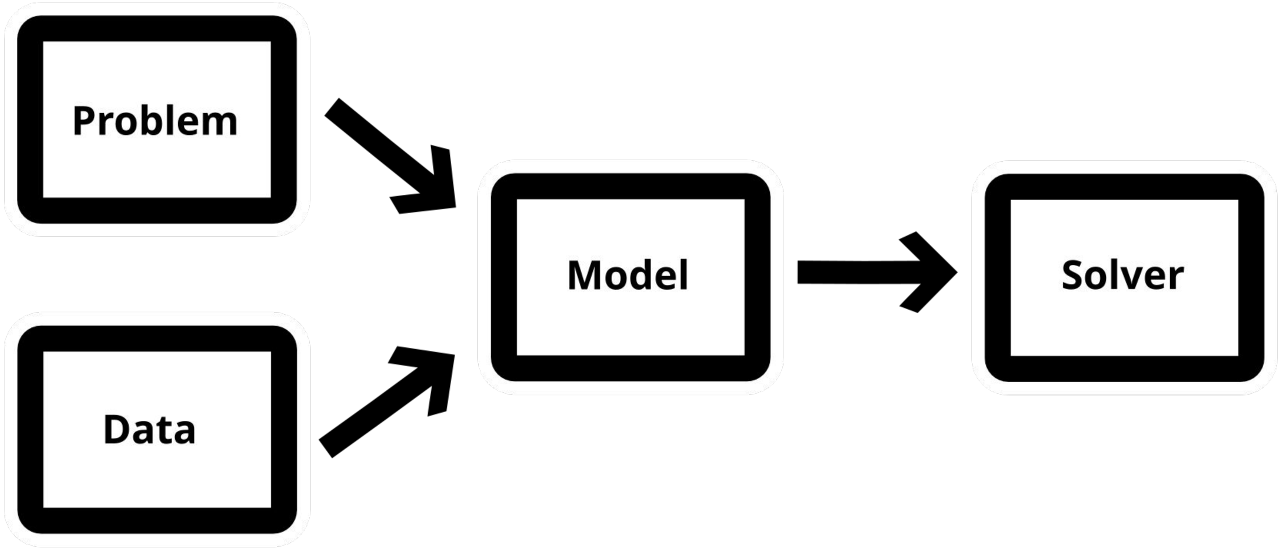
	IWO
idx1 (col2)	8/54 = 0.14
idx2 (col2, col1)	38/54 = 0.70
idx3 (col3)	16/54 = 0.29

A Constraint Programming Model For Index Selection

Optimization

- Find a good solution to a problem
- How?
  - Heuristics
  - Exact methods (MIP, CP, etc.)

Optimization



Optimization

Make the index write overhead small

↓

$$\min \sum_{i \in \mathcal{I}} x_i w_i^p + \sum_{k \in \mathcal{E}} y_k w_k^e$$

↓

```
model.Add(model.objective ==
          cp_model.LinearExpr.WeightedSum(model.x, model.pind_iwo) +
          cp_model.LinearExpr.WeightedSum(model.y, model.eind_iwo))
model.Minimize(model.objective)
```

Full details in our Technical White Paper

— — — Link

A Constraint Programming Approach for Index Selection in Postgres

Lukas Fittl, Marko Mijalkovic, and Philippe Olivier

2024-04-25

Abstract

We introduce a new method to automatically determine a set of indexes to create for a given Postgres query workload. This approach uses constraint programming to formulate a model representing the objectives and the constraints defined by the user. The input data is acquired by processing the Postgres query workload statistics derived from `pg_stat_statements`, coming up with multiple hypothetical indexes, and costing them using HypoPG. The model combined with the data, when solved, yields a result satisfying the intent of the user. This work was presented at the PGCon 2023, JOPT 2023, and PGDay Chicago 2024 conferences.

1 Introduction

As a database grows in size, reading from it becomes slower. Database indexes mitigate this problem by trading some write speed in exchange for faster reads. However, choosing which combination of indexes to create in order to optimize this trade-off is a complex task that generally requires vast domain knowledge.

This is not a new problem, and research on index selection dates back at least to the 1970s [1]. A recent survey compares several index selection methods, from early tentatives in the 1980s up to today with more sophisticated approaches [2]. There are also projects in the Postgres community for automatic indexing, such as Dexter [3].

The method presented in this document takes multiple concepts into consideration, and allows the careful balancing of trade-offs between different measures

4 Objectives and Constraints

In this section we define the objectives and the constraints that can be added to the base model described previously. It should be noted that, optionally, it is possible to assign arbitrary tags to the scans. This tagging system allows most objectives and constraints to *only consider* (as well as to *specifically ignore*) various subsets of scans.

4.1 Objectives

We describe how the objectives are expressed in the model, as well as their associated constraints.

4.1.1 Minimize Index Write Overhead

The *Minimize Index Write Overhead* objective (19) aims to minimize the total IWO of the selected indexes

$$\min \sum_{i\in\mathcal{E}} w_i^{\mathrm{e}} x_i + \sum_{j\in\mathcal{P}} w_j^{\mathrm{p}} y_j. \tag{19}$$

Let X be the value found by this objective. The associated constraint (20) ensures that any future solution does not total more than X IWO (adjusted for tolerance)

$$\sum_{i\in\mathcal{E}} w_i^{\mathrm{e}} x_i + \sum_{j\in\mathcal{P}} w_j^{\mathrm{p}} y_j \le \lfloor X(1+t) \rfloor . \tag{20}$$

4.1.2 Minimize Number of Indexes

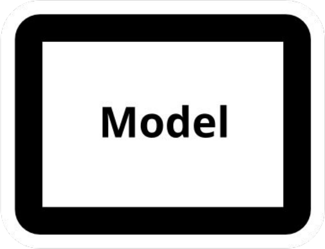
The *Minimize Number of Indexes* objective (21) aims to minimize the number of indexes selected in the solution

$$\min \sum_{i\in\mathcal{E}} x_i + \sum_{j\in\mathcal{P}} y_j. \tag{21}$$

Let X be the value found by this objective. The associated constraint (22) ensures that any future solution does not contain more than X indexes (adjusted for tolerance)

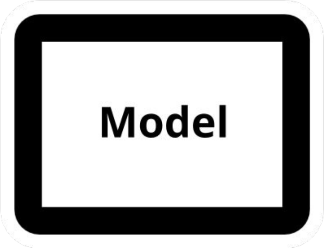
$$\sum_{i\in\mathcal{E}} x_i + \sum_{j\in\mathcal{P}} y_j \le \lfloor X(1+t) \rfloor . \tag{22}$$

Declarative Model



- Variables: <u>What</u> we want to find
- Constraints: <u>Rules</u> we must follow
- Objectives: <u>Goals</u> we want to achieve

Declarative Model



- Variables: What we want to find ← **solution**
- Constraints: Rules we must follow
- Objectives: Goals we want to achieve

The solver finds a solution (the best?) to the model.

Index Selection Model

- Variables: Which indexes to select
- Constraints: User-defined rules
- Objectives: User-defined goals

The index selection model will find a suitable selection of indexes.

Example: *“Select the indexes that minimize the costs and the IWO.”*

Single and Multiple Goals

Single goal:

- Minimize the costs: Easy! Use more indexes
- Minimize the IWO: Easy! Use fewer indexes

Single and Multiple Goals

Single goal:

- Minimize the costs: Easy! Use <u>more</u> indexes
- Minimize the IWO: Easy! Use <u>fewer</u> indexes

conflict

Multiple goals:

- Minimize the costs <u>and</u> the IWO: ???

Conflicting Goals

Multi-objective methods:

- Weighted sum method
- ϵ-constraint method
- Lexicographic method
- **Hierarchical optimization method**

Conflicting Goals

Sort the goals by preference:

1. First goal: Minimize the costs
2. New rule: The costs cannot be higher than X
3. Second goal: Minimize the IWO

Conflicting Goals

Sort the goals by preference:

tolerance

1. First goal: Minimize the costs
2. New rule: The costs cannot be higher ~~than X~~ <u>than X + 10%</u>
3. Second goal: Minimize the IWO

Conflicting Goals

Sort the goals by preference:

tolerance

1. First goal: Minimize the costs
2. New rule: The costs cannot be higher ~~than X~~ than X + 10%
3. Second goal: Minimize the IWO

"Costs can be 10% worse than whatever the lowest possible costs are. Which selection of indexes gives me that for as little IWO as possible?"

Example

–

1. Minimize costs (10% tolerance)

2. Minimize IWO

IWO ↓		S_1	S_2	S_3
3	I_1	4	3	
3	I_2		3	4
1	I_3	8		5
1	I_4	7	2	8

Example

–

1. **Minimize costs (10% tolerance)**

   Indexes: I_1, I_2, I_4

   Costs: 4 + 2 + 4 = 10

   IWO: 3 + 3 + 1 = 7

2. **Minimize IWO**

IWO ↓		S_1	S_2	S_3
3	I_1	4	3	
3	I_2		3	4
1	I_3	8		5
1	I_4	7	2	8

pganalyze

Example

1. **Minimize costs (10% tolerance)**

   Indexes: I_1, I_2, I_4

   Costs: 4 + 2 + 4 = 10

   IWO: 3 + 3 + 1 = 7

2. **Minimize IWO + rule: costs ≤ 11**

(11 is 10% worse than 10)

IWO	Index	S_1	S_2	S_3
3	I_1	4	3	
3	I_2		3	4
1	I_3	8		5
1	I_4	7	2	8

pganalyze

Example

–

1. Minimize costs (10% tolerance)

Indexes: I_1, I_2, I_4

Costs: 4 + 2 + 4 = 10

IWO: 3 + 3 + 1 = 7

10% worse than

2. Minimize IWO + rule: costs ≤ 11

Indexes: I_1, I_3, I_4

Costs: 4 + 2 + 5 = 11

IWO: 3 + 1 + 1 = 5

IWO ↓		S_1	S_2	S_3
3	I_1	4	3	
3	I_2		3	4
1	I_3	8		5
1	I_4	7	2	8

pganalyze

Utilizing The Index Selection Model In Practice

```
"Method": "CP",
"Options": {
  "Goals": [ { "Name": "Minimize Total Cost", "Tolerance": 0 },
             { "Name": "Minimize Index Write Overhead", "Tolerance": 0 } ],
}
```

pganalyze

— — —

Summary

Total runtime: 5.05 s, model runtime: 0.08 s

Goals

1. **Minimize Total Cost** achieved minimum 1,245,000, no tolerance
2. **Minimize Index Write Overhead** achieved minimum 2.03, no tolerance

Result

Indexes Used:	23 (12 existing, 11 to be added)	**Scan Coverage:**	38 covered (38 by existing, 34 by to be added), 1 uncovered
Index Write Overhead:	2.03 (0.98 existing, 1.05 to be added)	**Scan Cost:**	1,245,000 total, 887,600 maximum per-scan
Update Overhead:	Up to 101.12 non-HOT updates / min	**Scan Impact:**	33,660,000,000 total, 33,620,000,000 maximum per-scan

Missing Indexes to Add

```
I16 CREATE INDEX CONCURRENTLY ON public.issues USING btree ("check", server_id, state, updated_at);
I20 CREATE INDEX CONCURRENTLY ON public.issues USING btree ("check", updated_at);
I29 CREATE INDEX CONCURRENTLY ON public.issues USING btree (database_id, created_at);
I32 CREATE INDEX CONCURRENTLY ON public.issues USING btree (database_id, server_id);
I41 CREATE INDEX CONCURRENTLY ON public.issues USING btree (database_id, updated_at);
I58 CREATE INDEX CONCURRENTLY ON public.issues USING btree (grouping_key, server_id);
I81 CREATE INDEX CONCURRENTLY ON public.issues USING btree (organization_id, "check", state, updated_at);
I82 CREATE INDEX CONCURRENTLY ON public.issues USING btree (organization_id, "check", updated_at);
I83 CREATE INDEX CONCURRENTLY ON public.issues USING btree (organization_id, server_id);
I100 CREATE INDEX CONCURRENTLY ON public.issues USING btree (server_id, severity);
I104 CREATE INDEX CONCURRENTLY ON public.issues USING btree (server_id, updated_at);
```

Copy CREATE INDEX commands

"Method": "CP",
"Options": {
  "Goals": [{ "Name": "Minimize Total Cost", "Tolerance": **0.10** },
             { "Name": "Minimize Index Write Overhead", "Tolerance": 0 }],
}

pganalyze

Summary

Total runtime: 5.01 s, model runtime: 0.11 s

Goals

1. **Minimize Total Cost** achieved minimum 1,245,000, tolerance 0.1 allows up to 1,369,500
2. **Minimize Index Write Overhead** achieved minimum 0.98, no tolerance

Result

Indexes Used:	12 (12 existing, 0 to be added)	**Scan Coverage:**	38 covered (38 by existing, 0 by to be added), 1 uncovered
Index Write Overhead:	0.98 (0.98 existing, 0.00 to be added)	**Scan Cost:**	1,329,000 total, 887,600 maximum per-scan
Update Overhead:	Up to 101.12 non-HOT updates / min	**Scan Impact:**	33,720,000,000 total, 33,620,000,000 maximum per-scan

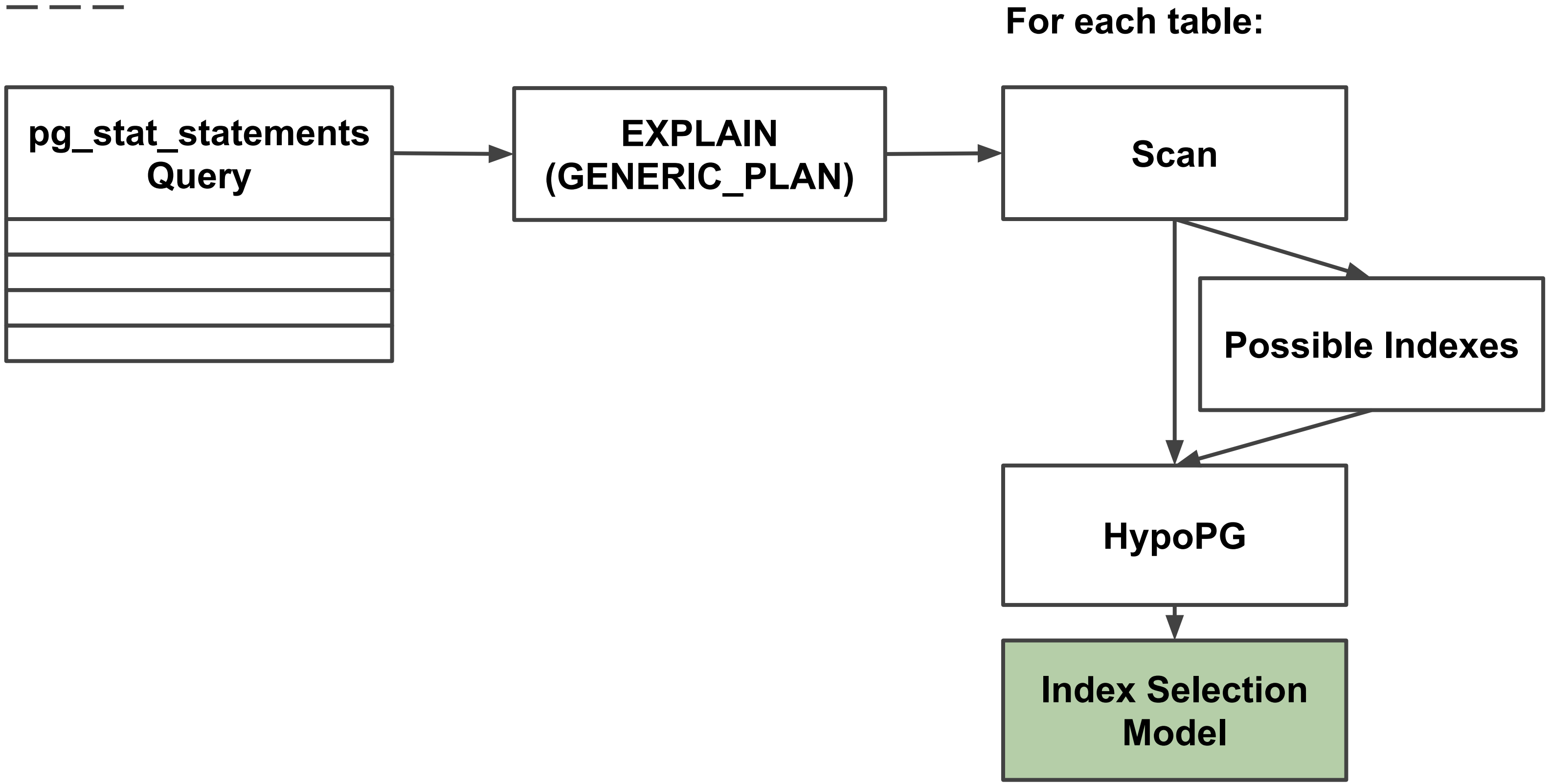
No changes recommended

With the current configuration, the Indexing Engine did not find any missing indexes to recommend. Try changing the index selection configuration settings for different trade-offs.
Learn more in documentation

Total Cost: 1,245,000 vs 1,329,000

Demo

How does this work?



index-selection.yml gives developer control

```
CREATE INDEX …
CREATE INDEX …
CREATE INDEX …
CREATE INDEX …
```

→

index-selection.yml

```
Goals:

 -  Name: Minimize Total Cost

    Tolerance: 0.10

 -  Name: Minimal Number of
    Indexes
```

Advanced Use Cases

Per-Scan Optimization

I29 btree (database_id, created_at) — +0.05 Index Write Overhead

SCAN EXPRESSION	EST. COST	EST. NEW COST	EST. SCANS/MIN
S32 (created_at >= $n) AND (created_at <= $n) AND (database_id = $n)	1,776.06	12.33	0.06

But what if we care about individual scans?

Total Cost: 1,245,000 vs 1,329,000

"Method": "CP",
"Options": {
  "Goals": [{ "Name": "Minimize Total Cost", "Tolerance": 0.10 },
             { "Name": "Minimize Index Write Overhead", "Tolerance": 0 }],
  "Rules": { **"Maximum Per-Scan Cost (Normal)": 100** }
}

pganalyze

— — —

Summary

Total runtime: 5.22 s, model runtime: 0.08 s

Goals

1. **Minimize Total Cost** achieved minimum 1,245,000, tolerance 0.1 allows up to 1,369,500
2. **Minimize Index Write Overhead** achieved minimum 1.29, no tolerance

Result

Indexes Used:	16 (12 existing, 4 to be added)	**Scan Coverage:**	38 covered (38 by existing, 33 by to be added), 1 uncovered
Index Write Overhead:	1.29 (0.98 existing, 0.31 to be added)	**Scan Cost:**	1,303,000 total, 887,600 maximum per-scan
Update Overhead:	Up to 101.12 non-HOT updates / min	**Scan Impact:**	33,680,000,000 total, 33,580,000,000 maximum per-scan

Missing Indexes to Add

```
I14 CREATE INDEX CONCURRENTLY ON public.issues USING btree ("check", server_id);
I29 CREATE INDEX CONCURRENTLY ON public.issues USING btree (database_id, created_at);
I32 CREATE INDEX CONCURRENTLY ON public.issues USING btree (database_id, server_id);
I83 CREATE INDEX CONCURRENTLY ON public.issues USING btree (organization_id, server_id);
```

Copy CREATE INDEX commands

Now our scan is indexed better.

We recommend testing insights 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.

HOT Updates

- HOT = Heap Only Tuples
- Reduces individual UPDATE overhead by not updating index entries
- Reduces future autovacuum effort by enabling on-access HOT pruning (which can happen on a per-page basis, only for Heap Only Tuples)
- **If you index a previously unindexed column***,
  any UPDATE statement involving that column could be impacted**

  * Postgres 16+ allows BRIN indexes to not interfere with HOT updates
  ** Updates that don’t actually change the column value may still use HOT

HOT Updates

pganalyze

Columns

Name	Type	Null %	Avg. Size	Est. Updates / Min	HOT?	Modifiers
id	integer	0.00%	4	n/a	No	NOT NULL DEFAULT nextval('public.issues_id_seq'::regclass)
database_id	bigint	2.93%	8	n/a	No	
check	character varying(255)	0.00%	29	n/a	No	NOT NULL
description_template	text	0.00%	81	21.95	Yes	
reference_id	text	96.70%	11	n/a	No	
reference_type	character varying(255)	96.70%	8	n/a	No	
created_at	timestamp without time zone	0.00%	8	n/a	Yes	NOT NULL
updated_at	timestamp without time zone	0.00%	8	93.63	No	NOT NULL
severity	integer	0.00%	4	0.08	No	NOT NULL
state	integer	0.00%	4	9.00	No	NOT NULL
server_id	uuid	0.00%	16	n/a	No	NOT NULL
organization_id	uuid	0.00%	16	n/a	No	NOT NULL
details	jsonb	0.00%	91	63.27	Yes	NOT NULL DEFAULT '{}'::jsonb
grouping_key	jsonb	0.00%	74	n/a	No	NOT NULL DEFAULT '{}'::jsonb
grouping_key_labels	jsonb	0.00%	49	<0.01	Yes	NOT NULL DEFAULT '{}'::jsonb
query_text	text	55.10%	92	n/a	Yes	

"Method": "CP",
"Options": {
"Goals": [{ "Name": "Minimize Total Cost", "Tolerance": 0.10 },
{ "Name": "Minimize Index Write Overhead", "Tolerance": 0 }],
"Rules": { **"Maximum Per-Scan Cost (Normal)": 100** }
}

pganalyze

Summary

Total runtime: 1.74 s, model runtime: 0.07 s

Goals

1. **Minimize Total Cost** achieved minimum 59,280, tolerance 0.1 allows up to 65,208
2. **Minimize Index Write Overhead** achieved minimum 0.30, no tolerance

Result

Indexes Used:	4 (3 existing, 1 to be added)	**Scan Coverage:**	28 covered (27 by existing, 1 by to be added), 4 uncovered
Index Write Overhead:	0.30 (0.26 existing, 0.04 to be added)	**Scan Cost:**	61,700 total, 15,450 maximum per-scan
Update Overhead:	Up to 283.16 non-HOT updates / min	**Scan Impact:**	1,100,000 total, 329,300 maximum per-scan

Missing Indexes to Add

```
I15 CREATE INDEX CONCURRENTLY ON public.servers USING btree (last_snapshot_id);
```

Copy CREATE INDEX commands

We recommend testing insights 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.

"Method": "CP",
"Options": {
"Goals": [**{ "Name": "Minimize Update Overhead", "Tolerance": 0 },**
{ "Name": "Minimize Total Cost", "Tolerance": 0.10 }],
{ "Name": "Minimize Index Write Overhead", "Tolerance": 0 }]
}

pganalyze

— — —

Summary

Total runtime: 1.36 s, model runtime: 0.04 s

Goals

1. **Minimize Update Overhead** achieved minimum 0.00, no tolerance
2. **Minimize Total Cost** achieved minimum 77,220, tolerance 0.1 allows up to 84,942
3. **Minimize Index Write Overhead** achieved minimum 0.26, no tolerance

Result

Indexes Used:	3 (3 existing, 0 to be added)	**Scan Coverage:**	27 covered (27 by existing, 0 by to be added), 5 uncovered
Index Write Overhead:	0.26 (0.26 existing, 0.00 to be added)	**Scan Cost:**	77,230 total, 15,540 maximum per-scan
Update Overhead:	Up to 0.00 non-HOT updates / min	**Scan Impact:**	1,100,000 total, 328,900 maximum per-scan

No changes recommended

With the current configuration, the Indexing Engine did not find any missing indexes to recommend. Try changing the index selection configuration settings for different trade-offs.
Learn more in documentation

Consolidating Indexes

- What if we've over indexed in the past, and want help reducing indexes?
- **We can use an index selection model to determine index to remove**
- This may cause new indexes to be created to consolidate existing ones
- Depending on the situation, removing indexes may cause significant slowdowns. **Caution and testing on database clones is advised!**

We have 12 indexes. **Can we go to <= 5?**

Indexes

Total Index Size	2.9 GB	Index Write Overhead	0.59

Name	Definition	Constraint	Valid?	First Seen	Index Size	Index Write Overhead
index_issues_on_check	btree ("check")		VALID	Over 30 days ago	164.8 MB	0.08
index_issues_on_database_id	btree (database_id)		VALID	Over 30 days ago	158 MB	0.03
index_issues_on_database_id_and_check	btree (database_id, "check")		VALID	Over 30 days ago	169.6 MB	0.10
index_issues_on_database_id_and_severity	btree (database_id, severity) WHERE (state <> 2)		VALID	Over 30 days ago	12.8 MB	0.00
index_issues_on_organization_id_and_check	btree (organization_id, "check")		VALID	Over 30 days ago	173.9 MB	0.11
index_issues_on_reference_type_and_reference_id	btree (reference_type, reference_id)		VALID	Over 30 days ago	590.8 MB	0.03
index_issues_on_server_id	btree (server_id)		VALID	Over 30 days ago	166 MB	0.05
index_issues_on_server_id_and_check	btree (server_id, "check")		VALID	Over 30 days ago	174.5 MB	0.11
index_issues_on_server_id_and_check_and_grouping_key	btree (server_id, "check", grouping_key) WHERE (state <> 2)		VALID	Over 30 days ago	168.1 MB	0.03
index_issues_on_server_id_and_severity	btree (server_id, severity) WHERE (state <> 2)		VALID	Over 30 days ago	12.2 MB	0.01
index_issues_on_server_id_and_updated_at	btree (server_id, updated_at) WHERE (state = 2)		VALID	Over 30 days ago	685.8 MB	0.01
issues_pkey	btree (id)	PRIMARY KEY (id)	VALID	Over 30 days ago	484.5 MB	0.03

```
"Method": "CP",
"Options": {
  "Goals": [ { "Name": "Minimize Total Cost", "Tolerance": 0 },
             { "Name": "Minimize Index Write Overhead", "Tolerance": 0 } ],
  "Rules": { "Maximum Number of Indexes": 5 }
}
```

[x] Allow consolidation/removal of indexes

pganalyze

Summary

Total runtime: 5.44 s, model runtime: 0.45 s

Goals

1. **Minimize Total Cost** achieved minimum 1,427,000, no tolerance
2. **Minimize Index Write Overhead** achieved minimum 0.59, no tolerance

Result

Indexes Used:	5 (2 existing, 3 to be added)	**Scan Coverage:**	38 covered (6 by existing, 35 by to be added), 1 uncovered
Index Write Overhead:	0.59 (0.27 existing, 0.32 to be added)	**Scan Cost:**	1,427,000 total, 887,600 maximum per-scan
Update Overhead:	Up to 11.61 non-HOT updates / min	**Scan Impact:**	33,470,000,000 total, 33,370,000,000 maximum per-scan

Missing Indexes to Add

```
I33 CREATE INDEX CONCURRENTLY ON public.issues USING btree (database_id, server_id, "check");
I91 CREATE INDEX CONCURRENTLY ON public.issues USING btree (reference_id, reference_type);
I98 CREATE INDEX CONCURRENTLY ON public.issues USING btree (server_id, organization_id, "check");
```

Copy CREATE INDEX commands

Existing Indexes to Remove

```
I2 DROP INDEX CONCURRENTLY index_issues_on_check; --- btree ("check")
I3 DROP INDEX CONCURRENTLY index_issues_on_database_id; --- btree (database_id)
I4 DROP INDEX CONCURRENTLY index_issues_on_database_id_and_check; --- btree (database_id, "check")
I5 DROP INDEX CONCURRENTLY index_issues_on_database_id_and_severity; --- btree (database_id, severity) WHERE (state <> 2)
I6 DROP INDEX CONCURRENTLY index_issues_on_organization_id_and_check; --- btree (organization_id, "check")
I7 DROP INDEX CONCURRENTLY index_issues_on_reference_type_and_reference_id; --- btree (reference_type, reference_id)
I8 DROP INDEX CONCURRENTLY index_issues_on_server_id; --- btree (server_id)
```

```
"Method": "CP",
"Options": {
  "Goals": [ { "Name": "Minimize Total Cost", "Tolerance": 0 },
             { "Name": "Minimize Index Write Overhead", "Tolerance": 0 } ],
  "Rules": { "Maximum Number of Indexes": 5 }
}
```

[x] Allow consolidation/removal of indexes

pganalyze

Summary

Total runtime: 5.44 s, model runtime: 0.45 s

Goals

1. **Minimize Total Cost** achieved minimum 1,427,000, no tolerance
2. **Minimize Index Write Overhead** achieved minimum 0.59, no tolerance

Result

Indexes Used:	5 (2 existing, 3 to be added)	**Scan Coverage:**	38 covered (6 by existing, 35 by to be added), 1 uncovered
Index Write Overhead:	0.59 (0.27 existing, 0.32 to be added)	**Scan Cost:**	1,427,000 total, 887,600 maximum per-scan
Update Overhead:	Up to 11.61 non-HOT updates / min	**Scan Impact:**	33,470,000,000 total, 33,370,000,000 maximum per-scan

Result

Indexes Used:	16 (12 existing, 4 to be added)	**Scan Coverage:**	38 covered (38 by existing, 33 by to be added), 1 uncovered
Index Write Overhead:	1.29 (0.98 existing, 0.31 to be added)	**Scan Cost:**	1,303,000 total, 887,600 maximum per-scan
Update Overhead:	Up to 101.12 non-HOT updates / min	**Scan Impact:**	33,680,000,000 total, 33,580,000,000 maximum per-scan

In Summary

- The goal is to (semi-)automate index selection based on **application developer & data team** intent
- Provide explanations why a particular index was chosen, and **make it easy to introspect/override the logic**
- Offer a configurable system that supports choosing multiple, conflicting objectives (e.g. make queries fast, but keep overhead low)
- Initial focus is on **checking for missing indexes** (e.g. to catch a change early that adds new queries but forgets the index)

thanks!

Email me to talk more about this:

lukas@pganalyze.com

Try out the code:

github.com/pganalyze/**pgday-chicago-2024**

github.com/pganalyze/**lint**

pganalyze