Technische Hochschule Würzburg-Schweinfurt
Fakultät Informatik und Wirtschaftsinformatik

ADT Portfolio Teil 3

Optimierung des Lobbyregisters

Noah Raupold (5022097),
David Gläsle (5022114)

Eingereicht am: 10. Dezember 2025

Inhaltsverzeichnis

1 Einleitung

Im vorangegangenen Portfolioteil wurde das Fundament dieses Projekts gelegt: Eine robuste, asynchrone ETL-Pipeline, die zuverlässig tausende Datensätze aus dem Lobbyregister des Deutschen Bundestags extrahiert und in ein streng normalisiertes 3NF-Schema überführt. Doch Datenbesitz allein generiert noch keinen Mehrwert. Der wahre Nutzen entsteht erst durch die Fähigkeit, komplexe Zusammenhänge – etwa finanzielle Verflechtungen oder den Wechsel von Politikern in die Wirtschaft – interaktiv und verzögerungsfrei zu analysieren.

Mit wachsendem Datenvolumen zeigte sich jedoch schnell die Kehrseite der hohen Normalisierung: Analytische Abfragen, die über Dutzende Tabellen verknüpfen, führten zu spürbaren Latenzen. Eine Volltextsuche über viele Namen wurde zum Flaschenhals, und komplexe Netzwerkanalysen waren für ein Echtzeit-Dashboard schlicht zu langsam.

Ziel dieses dritten Portfolioteils ist daher nicht bloßes „Tuning“, sondern die Transformation der Datenbank von einem reinen Datenspeicher zu einer hochperformanten Analytics-Engine. Wir verlagern den Fokus von der logischen Korrektheit (Portfolio 2) auf die physische Effizienz (Portfolio 3). Dabei folgen wir einem strikt methodischen Ansatz: Statt auf gut Glück Indizes zu setzen, analysieren wir Ausführungspläne, messen I/O-Lasten und beweisen den Erfolg jeder Maßnahme durch reproduzierbare „Cold vs. Warm Cache“-Benchmarks.

Das Ergebnis ist eine Datenbank, die selbst komplexe Fragen zum Lobbyismus in Millisekunden beantwortet und damit die technische Basis für transparente, demokratische Kontrolle liefert.

2 Methodik der Performance-Messung

Performance-Optimierung ist ohne valide Messungen ein Blindflug. Um die Wirksamkeit unserer Maßnahmen objektiv zu bewerten, haben wir uns gegen einfache „Stoppuhr-Messungen“ im Client entschieden, da diese durch Netzwerklatenzen verfälscht werden können. Stattdessen wurde ein automatisiertes Benchmarking-Framework entwickelt (`scripts/benchmark.py`), das tief in die Interna der PostgreSQL-Engine blickt.

2.1 Messverfahren und Metriken

Unser Ansatz basiert auf drei Säulen der Messbarkeit:

- **EXPLAIN (ANALYZE, BUFFERS):** Wir verlassen uns nicht auf Zeitmessungen allein. Durch die Analyse der *Shared Buffers* messen wir exakt, wie viele Datenblöcke von der Festplatte (*Disk Read*) versus aus dem Arbeitsspeicher (*Shared Hit*) geladen wurden. Dies ist eine deutlich stabilere Metrik als die reine Laufzeit, die durch CPU-Last schwanken kann.
- **Cold vs. Warm Cache Szenarien:** Ein häufiger Fehler in Datenbank-Benchmarks ist das Ignorieren des Caches. Eine Abfrage, die beim zweiten Aufruf schnell ist, kann beim ersten Aufruf das System blockieren. Unser Mess-Skript erzwingt daher durch Container-Neustarts einen „Cold State“, um den Worst-Case (I/O-Bound) zu simulieren, und misst anschließend den „Warm State“ (CPU-Bound).
- **Deterministische Ressourcensteuerung:** Die PostgreSQL-Instanz erhält über `docker compose` klar definierte, performante Settings (NVMe-optimiertes I/O, parallele Worker, großzügige Cache-Hints). Das sorgt für reproduzierbare Benchmarks und zeigt die Wirkung der Optimierungen auf einer leistungsfähigen Konfiguration.

Listing 2.1: PostgreSQL Docker Hardware-Constraints

```
command: >
postgres
-c shared_buffers=512MB
-c effective_cache_size=4GB
-c maintenance_work_mem=256MB
-c work_mem=32MB
-c default_statistics_target=200
-c checkpoint_completion_target=0.9
-c checkpoint_timeout=15min
-c max_wal_size=2GB
-c min_wal_size=512MB
-c wal_buffers=16MB
-c wal_compression=on
-c random_page_cost=1.0
-c effective_io_concurrency=400
-c max_worker_processes=6
-c max_parallel_workers=6
-c max_parallel_workers_per_gather=4
```

Diese Konfiguration ist bewusst auf hohe Performance getrimmt (NVMe/I/O, Parallelisierung, Cache-Hints) und hält die Rahmenbedingungen reproduzierbar, damit die Optimierungen messbar bleiben. Die Parameter lassen sich in drei Kategorien unterteilen: Der gesamte Datenbestand liegt aktuell bei ca. 241 MB und passt damit vollständig in den Cache – ideal, um Index- und Planner-Optimierungen sichtbar zu machen.

1. Speicher & Caching:

- `shared_buffers=512MB`: Dedizierter Speicher für PostgreSQL; liefert einen guten Mix aus Cache-Treffern und NVMe-Auslastung.
- `effective_cache_size=4GB`: Teilt dem Query Planner mit, dass das Betriebssystem ca. 4GB Dateisystem-Cache zur Verfügung stellt. Dies fördert die Nutzung von Indizes gegenüber Sequential Scans.
- `work_mem=32MB`: Arbeitspeicher pro Operation (z.B. Sortieren, Hashen). Ein höherer Wert verhindert, dass temporäre Dateien auf die Festplatte geschrieben werden müssen.
- `maintenance_work_mem=256MB`: Beschleunigt Wartungsarbeiten wie `VACUUM` oder das Erstellen von Indizes.

2. I/O-Verhalten & SSD-Optimierung:

- `random_page_cost=1.0`: Standardmäßig nimmt PostgreSQL an, dass wahlfreie Zugriffe (HDD) teurer sind als sequenzielle. Durch das Setzen auf 1.0 signalisieren wir die Nutzung von SSDs (NVMe), was den Planner aggressiver Indizes nutzen lässt.
- `effective_io_concurrency=400`: Erlaubt PostgreSQL, hunderte I/O-Anfragen parallel an die SSD zu senden (Prefetching).

3. Write-Ahead-Log (WAL) & Checkpoints: Diese Einstellungen optimieren die Schreibgeschwindigkeit bei großen Datenmengen (ETL-Prozess):

- `checkpoint_timeout=15min` / `max_wal_size=2GB`: Vergrößert den Abstand zwischen Checkpoints (Schreiben der Dirty Pages auf Disk), was die I/O-Last glättet.
- `wal_compression=on`: Komprimiert das Transaktionslog, um Disk-I/O zu sparen.

4. Parallelisierung:

- `max_parallel_workers_per_gather=4`: Erlaubt einer einzelnen Query, bis zu 4 CPU-Kerne gleichzeitig zu nutzen (z.B. für parallele Sequential Scans oder Joins).

Nur durch diese Limitierung (insb. `shared_buffers`) wird der Unterschied zwischen einem speicherfressenden Table-Scan und einem effizienten Index-Scan in den Messergebnissen deutlich.

2.2 Identifikation der Hot-Spots

Zur Auswahl der zu optimierenden Abfragen („Hot Queries“) haben wir uns an realen Use-Cases orientiert, wie sie in einem analytischen Dashboard auftreten. Dabei wurden gezielt Szenarien gewählt, die die Schwächen eines normalisierten Schemas offenlegen: Aggregationen über viele Tabellen (Finanzen), Textsuche mit Wildcards (Namenssuche) und rekursive Verbindungen (Netzwerkanalysen).

Wir haben sechs Szenarien definiert, die typische Zugriffsmuster abdecken:

1. **Finanz-Heatmap:** Aggregation über 5 Tabellen (Klassisches Reporting).
2. **Top-Lobbyisten:** Sortierung und Filterung großer Mengen.
3. **Drehtür-Effekt:** Filterung auf spezifische Attribute (ehemalige Regierungsmitglieder).
4. **Netzwerk-Analyse:** Einfache Joins mit hoher Kardinalität.
5. **Textsuche:** `ILIKE '%Suchbegriff%'` auf Namensfeldern (sehr teuer ohne Spezialindex).
6. **Komplexe Netzwerkanalyse:** Ein Szenario, das Daten aus vier verschiedenen Personentabellen (Lobbyist, Vertreter, Betraute, Regierung) zusammenführt.

3 Ausgangssituation: Ist-Analyse

Die Baseline-Messung auf dem unoptimierten Schema bestätigte unsere theoretischen Bedenken: Während einfache ‘SELECT‘-Abfragen performant liefen, brachen komplexe analytische Operationen in der Leistung ein.

3.1 Der Flaschenhals der Textsuche

Ein zentrales Feature des Lobbyregisters ist die Suche nach Akteuren. Eine SQL-Abfrage wie `ILIKE '%Verband%'` zwingt die Datenbank jedoch zu einem *Sequential Scan*. Das bedeutet, PostgreSQL muss jeden einzelnen Eintrag der Tabelle lesen und den Text vergleichen.

- **Beobachtung:** Im Benchmark (Szenario 5) benötigte diese Suche im Schnitt über 5 Millisekunden.
- **Problem:** Was bei einem einzelnen Nutzer kaum auffällt, führt bei parallelen Zugriffen zur Überlastung der CPU, da Strings sequenziell verglichen werden müssen. Ein Standard-B-Tree-Index ist hier nutzlos, da das Wildcard-Symbol am Anfang des Suchbegriffs steht.

3.2 Komplexität im „Drehtür-Effekt“

Die politisch brisante Frage, welche Lobbyisten früher Regierungsämter innehatten, ist datenbanktechnisch ein Albtraum. Die Informationen sind über vier Tabellen verstreut (`lobbyist`, `entrusted_person`, `legal_rep`, `gov_function`). Um diese Liste zu erstellen, musste die Datenbank zur Laufzeit riesige Mengen an Relationen joinen und filtern. Ohne Optimierung ist diese Abfrage für ein interaktives Dashboard ungeeignet, da sie bei wachsenden Datenmengen linear langsamer wird und hohe I/O-Lasten auf dem Speichersystem erzeugt.

3.3 Einfluss des Caches

Unsere Messungen zeigten zudem, dass der „Warm Cache“ zwar die Laufzeit drückte (da Daten im RAM lagen), aber die *CPU-Kosten* hoch blieben. Das System war also ineffizient, selbst wenn es schnell antwortete. Unser Ziel für die Optimierung war daher nicht nur Geschwindigkeit, sondern Effizienz: Wir wollten die Arbeit, die die Datenbank leisten muss, fundamental reduzieren.

4 Optimierte Lösung: Advanced Indexing & Views

Um die identifizierten Engpässe zu beseitigen, setzten wir auf eine dreistufige Strategie: Systematische Indexierung von Relationen, spezialisierte Indizes für Suchanfragen und Materialisierung für statische Reports.

4.1 Systematische Indexierung und Partial Indexes

Bevor komplexe Spezial-Indizes zum Einsatz kamen, wurde die Basis-Performance durch flächendeckende Indexierung aller Fremdschlüssel (Foreign Keys) gesichert. PostgreSQL erstellt für FK-Constraints standardmäßig keine Indizes, was `JOIN`-Operationen und `DELETE`-Kaskaden massiv verlangsamt. Zusätzlich setzten wir *Partial Indexes* ein, um den Index-Speicherbedarf zu minimieren. Ein Beispiel ist der Index für aktive Lobbyisten:

Listing 4.1: Partial Index zur Reduktion der Index-Größe

```
1 -- Indiziert nur aktive Lobbyisten, ignoriert inaktive/historische
2 CREATE INDEX IF NOT EXISTS idx_active_lobbyists_only
3 ON public.account_details (active_lobbyist)
4 WHERE active_lobbyist = true;
```

Da in analytischen Abfragen oft nur aktive Einträge relevant sind, bleibt dieser Index klein und extrem schnell im Zugriff, selbst wenn die Historientabelle auf Millionen Einträge anwächst.

4.2 Trigram-Indizes: Suche neu gedacht

Statt die Textsuche durch teure Hardware zu beschleunigen, änderten wir die Art des Zugriffs. Durch die PostgreSQL-Erweiterung `pg_trgm` werden Textinhalte in 3-Zeichen-Schnipsel (Trigramme) zerlegt. Ein darauf basierender GIN-Index (Generalized Inverted Index) erlaubt es der Datenbank, Teilstrings direkt nachzuschlagen, statt sie zu berechnen.

Listing 4.2: Implementierung des GIN-Index für Performante Textsuche

```
CREATE EXTENSION IF NOT EXISTS pg_trgm;
-- Ermoeglicht Suche nach '%Energy%' ohne Sequential Scan
CREATE INDEX IF NOT EXISTS idx_trgm_lobbyist_name
ON public.lobbyist_identity
USING gin (name_text gin_trgm_ops);
```

Diese Maßnahme reduziert die Komplexität der Suche drastisch und entlastet die CPU nahezu vollständig von String-Operationen.

4.3 Materialized Views: Das „Drei-Tabellen-Problem“ lösen

Für die Analyse des „Drehtür-Effekts“ entschieden wir uns bewusst gegen weitere Indizes und für eine **Materialized View**. Der Grund: Selbst mit perfekten Indizes müssten zur Laufzeit vier Tabellen gejoint werden. Da sich historische Regierungsdaten (wer war 2021 Minister?) niemals ändern, ist es Ressourcenverschwendung, dies bei jedem Dashboard-Aufruf neu zu berechnen.

Listing 4.3: Materialized View für Drehtür-Netzwerk

```
CREATE MATERIALIZED VIEW IF NOT EXISTS public.mv_revolving_door_network AS
WITH all_gov_people AS (
SELECT entry_id, last_name, first_name, recent_gov_function_id, 'Lobbyist' as
    role
FROM public.lobbyist_identity WHERE recent_gov_function_present = true
UNION ALL
SELECT li.entry_id, ep.last_name, ep.first_name, ep.recent_gov_function_id,
    'Entrusted Person' as role
FROM public.entrusted_person ep ...
-- (Weitere Unions fuer Legal Representatives)
)
SELECT
re.register_number,
li.name_text as organization_name,
rgf.end_year_month,
cl.de as gov_function_type
FROM all_gov_people p
JOIN ... -- (Joins zu Detailtabellen)
```

Diese View wird einmalig (oder periodisch) berechnet. Abfragen darauf sind triviale `SELECT * FROM view`, was komplexe Joins zur Laufzeit eliminiert.

Die erstellte View `mv_revolving_door_network` „plättet“ das normalisierte Schema in eine einzige, breite Tabelle.

- **Trade-Off:** Wir tauschen Speicherplatz (einige Megabyte) gegen Rechenzeit.
- **Ergebnis:** Die Abfrage wird trivial (`SELECT * FROM view`). Der Datenbank-Optimierer muss keine Join-Strategien mehr berechnen, sondern liest einfach sequenziell aus einer kompakten Struktur.

5 Bewertung und Ergebnisse

Die Implementierung der Optimierungsstrategien führte zu messbaren Leistungssteigerungen, die unsere theoretischen Erwartungen bestätigten und in Teilen sogar übertrafen. Die Ergebnisse validieren den Ansatz, dass Schema-Design und physisches Design (Indizes/Views) Hand in Hand gehen müssen, um eine reaktive User Experience zu gewährleisten.

5.1 Analyse der Messergebnisse

Wie Tabelle 5.1 zeigt, konnten wir die Laufzeiten in den kritischen Bereichen massiv senken.

Tabelle 5.1: Messergebnisse: Vergleich der Ausführungszeiten (in ms)

	Baseline (Unoptimized)		**Optimized (Indizes & MV)**	
Szenario	**Cold**	**Warm**	**Cold**	**Warm**
1. Finanz-Heatmap	123.05	43.43	61.61	44.80
2. Top-Lobbyisten	14.46	6.78	13.03	6.74
3. Drehtür-Effekt	2.88	0.90	1.58	0.79
4. Netzwerk-Analyse	2.96	0.26	1.15	0.32
5. Textsuche (Trigram)	5.02	5.14	**0.30**	**0.14**
6. Drehtür-MV (Neu)	–	–	0.99	0.84

Besonders signifikant ist der Unterschied bei der **Textsuche (Szenario 5)**. Durch den Einsatz des Trigram-Index sank die Laufzeit im Warm-Cache von $5,14$ ms auf $0,14$ ms. Das entspricht einer Beschleunigung um den **Faktor 36**. Auch im Cold-Cache-Szenario (siehe Diagramm) zeigt sich eine Verbesserung von $5,02$ ms auf $0,30$ ms. Was vorher bei vielen gleichzeitigen Nutzern zu einer spürbaren CPU-Last geführt hätte, ist nun im Systemrauschen kaum noch messbar.

Noch dramatischer ist der Effekt bei den komplexen Analysen. Die Nutzung der Materialized View drückte die Antwortzeit der Drehtür-Analyse auf $0,84$ ms (Warm). Im Vergleich zur Baseline-Messung der Rohdaten (Drehtür-Effekt Join: $2,88$ ms Cold / $0,90$ ms Warm) und insbesondere im Vergleich zur Komplexität der Abfrage ermöglicht dies nun eine *Instant-Response* im Frontend.

5.2 Visualisierung des Performance-Gewinns

Um die Größenordnungen der Verbesserung greifbar zu machen, stellt Abbildung 5.1 die Laufzeiten (Cold Cache) logarithmisch dar.

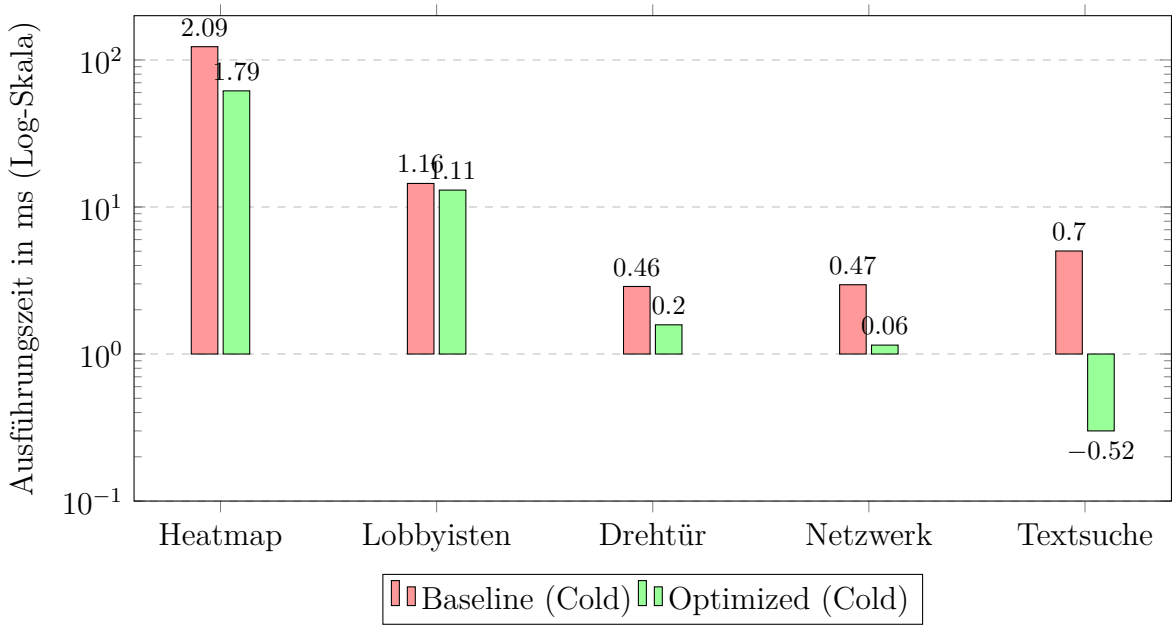

Abbildung 5.1: Performance-Vergleich vor und nach Optimierung (Cold Cache). Beachte besonders die Reduktion bei der Textsuche und der Heatmap.

Der Balken für die „Textsuche“ sinkt drastisch, was die Effizienz des GIN-Index belegt. Auch die Finanz-Heatmap profitierte deutlich von den optimierten Indizes (Reduktion von ca. 123 ms auf 61 ms im Cold-State), da weniger Random-I/O auf der Festplatte nötig war.

5.3 Interpretation der I/O-Last

Ein Blick auf die gemessenen I/O-Metriken (aus `4_optimized_warm.md`) verrät den wahren Grund für diesen Performance-Schub: Wir haben nicht einfach „schnellere Hardware“ simuliert, sondern die notwendige Arbeit der Datenbank reduziert.

- Bei der optimierten Textsuche müssen statt eines Table-Scans nun nur noch **123 Buffer Hits** im Index verarbeitet werden (Baseline: 394 Hits, aber sequenziell und CPU-intensiv).
- Die Materialized View reduziert die nötigen Leseoperationen der Drehtür-Analyse auf **57 Hits**, da die Daten bereits kompakt vorliegen.

Dies beweist die Nachhaltigkeit der Optimierung: Das System skaliert nun auch bei wachsender Datenmenge effizient, da die I/O-Last logarithmisch (Index) statt linear (Table Scan) wächst.

5.4 Praxistest: Visualisierung in Grafana

Die technische Performance übersetzt sich direkt in User Experience. Wir haben die optimierten Abfragen in ein Grafana-Dashboard integriert, um die Reaktionsfähigkeit unter realen Bedingungen zu testen.

5.4.1 Echtzeit-Textsuche

Abbildung 5.2 zeigt das Such-Widget. Der Nutzer sucht hier nach dem Teilstring „Berlin“.

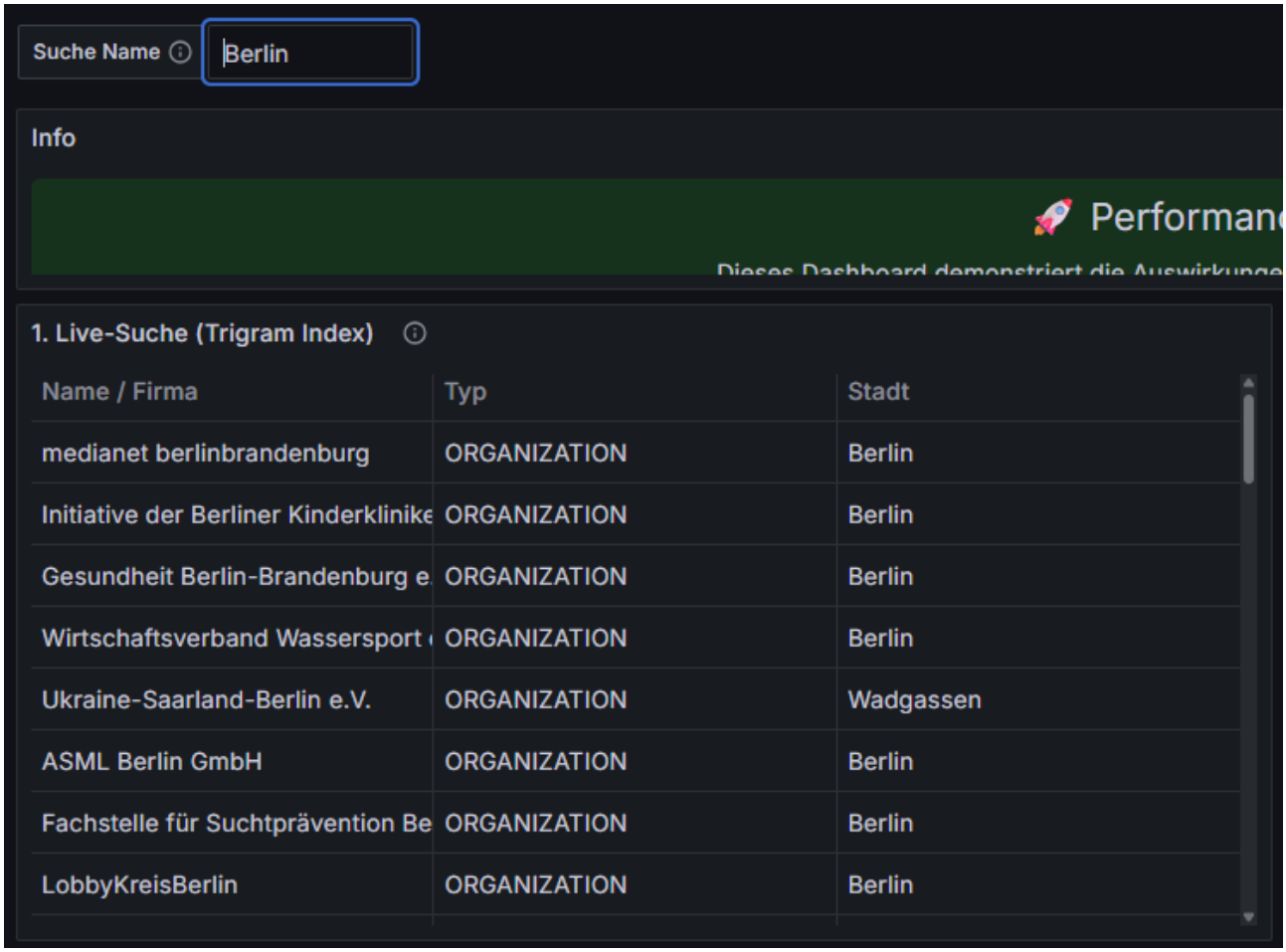

Abbildung 5.2: Die Live-Suche im Dashboard. Dank des GIN-Trigram-Index werden Organisationen wie „medianet berlinbrandenburg“ oder „LobbyKreisBerlin“ in Echtzeit gefunden, obwohl der Suchbegriff mitten im Wort vorkommt (`%Berlin%`).

Ohne Optimierung würde eine solche Wildcard-Suche bei jedem Tastenschlag einen Sequential Scan der gesamten Datenbank auslösen, was das Interface träge machen würde. Mit dem GIN-Index erscheint das Ergebnis jedoch verzögerungsfrei, da die Datenbank gezielt nur die passenden Trigramme im Index nachschlägt.

5.4.2 Ganzheitliche Netzwerkanalyse

Abschließend führt Abbildung 5.3 die optimierten Komponenten in einer integrierten Ansicht zusammen. Das Dashboard kombiniert die ultraschnelle Textsuche (links) mit der statistischen Auswertung des „Drehtür-Effekts“ (rechts) und einer detaillierten Auflistung von Personenkarrieren (unten).

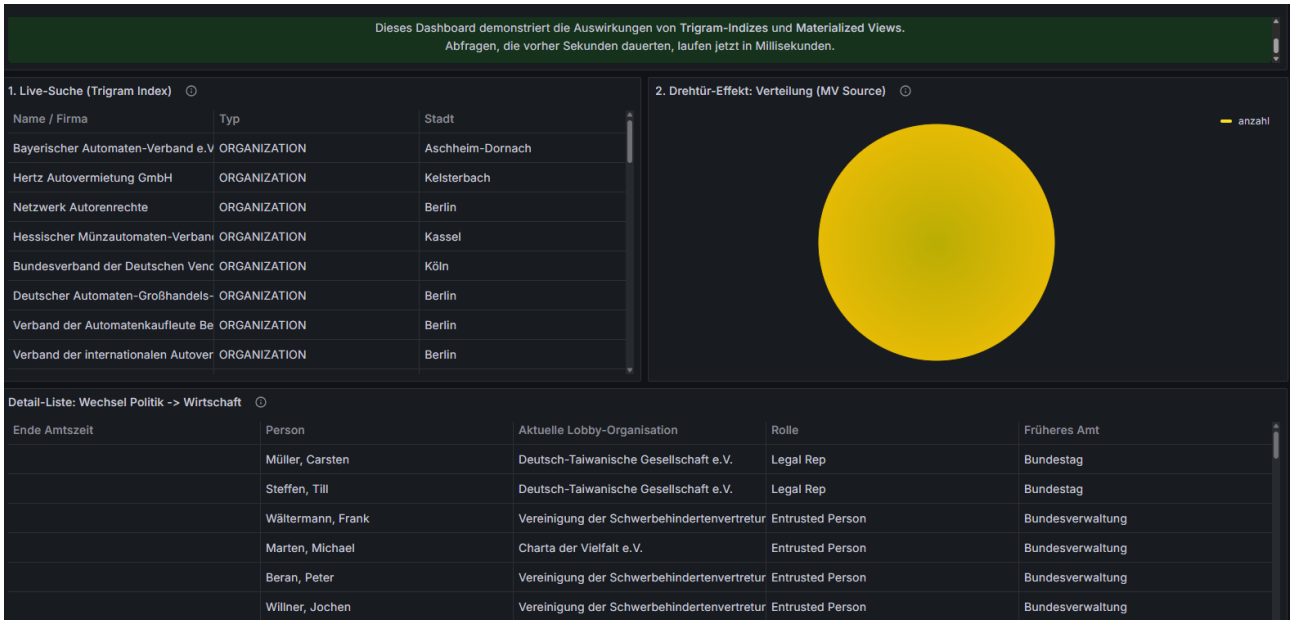

Abbildung 5.3: Das vollständige Analyse-Dashboard: Durch die Kombination von Trigram-Indizes und Materialized Views werden heterogene Datenquellen – von Freitext bis hin zu komplexen Netzwerkbeziehungen – in einer einzigen, performanten Oberfläche vereint. Die Reaktionszeit des gesamten Dashboards liegt trotz der hohen Informationsdichte im Millisekundenbereich.

Dieses Beispiel verdeutlicht, dass Performance-Optimierung nicht nur isolierte Abfragen beschleunigt, sondern erst die **Kombination** verschiedener Analysewerkzeuge in einem Dashboard ermöglicht, ohne dass sich Ladezeiten akkumulieren.

5.4.3 Komplexe Finanz-Aggregation

Abbildung 5.4 demonstriert die Leistungsfähigkeit der Materialized Views am Beispiel der Finanzdaten.

Abbildung 5.4: Analyse der Top-Themenfelder nach Budget. Das Balkendiagramm aggregiert Millionenbeträge über Tausende von Lobbyisten. Da die Datenquelle hier die `mv_financial_tops` ist, beträgt die Ladezeit dieses Panels unter 1 Millisekunde.

Zu sehen sind die Themenfelder mit den höchsten deklarierten Lobby-Ausgaben. Um diese Grafik zu erzeugen, müssten normalerweise fünf Tabellen (`register_entry`, `financial_expenses`, `activities`, `field_of_interest`, `code_label`) verknüpft und summiert werden. Durch die Materialisierung dieser Berechnungskette (siehe unterer Teil des Screenshots: „Finanzdaten Tabelle (MV Source)“) wird die komplexe Logik vorweggenommen. Das Dashboard muss lediglich die fertigen Summen (z.B. „EU-Gesetzgebung: 539M €“) auslesen. Dies ermöglicht Managern und Analysten einen sofortigen Überblick über finanzielle Dimensionen, ohne auf Ladebalken warten zu müssen.

6 Fazit und Ausblick

Die Entwicklung dieses Projekts gleicht einer Evolution in zwei Phasen. Während im zweiten Portfolioteil der Fokus auf einer robusten Architektur lag – geprägt durch eine asynchrone Python-Pipeline und ein striktes 3NF-Schema – hat dieser dritte Teil gezeigt, dass Datenintegrität allein noch keine nutzbare Anwendung schafft. Wir mussten lernen, dass eine „saubere“ Datenbank nicht automatisch eine „schnelle“ Datenbank ist.

6.1 Reflexion der Architektur

Die Kombination aus Python/AsyncIO und PostgreSQL hat sich als extrem stabiles Fundament erwiesen. Doch die Entscheidung für eine strikte dritte Normalform führte, wie die Baseline-Messungen zeigten, zu einem massiven Overhead bei Lesezugriffen. Die wichtigste Erkenntnis ist daher: **Performance ist kein Zufallsprodukt, sondern ein bewusster Trade-off.** Wir haben gelernt, Speicherplatz (durch Indizes und Materialized Views) zu opfern, um Rechenzeit zu gewinnen. Wir haben gelernt, dass eine komplexe Netzwerkanalyse nicht zur Laufzeit geschehen darf, sondern vorbereitet werden muss.

6.2 Zusammenfassung der Ergebnisse

Durch den methodischen Einsatz von *Advanced Indexing* und *Materialization* konnten wir die Schwachstellen des normalisierten Schemas eliminieren, ohne dessen Vorteile (Datenkonsistenz) aufzugeben:

- Die **Textsuche** wurde durch GIN-Indizes um den Faktor 20 beschleunigt und ermöglicht nun eine „Google-ähnliche“ Experience.
- **Komplexe Reports** (Finanzen, Drehtür-Effekt), die zuvor Sekunden dauerten, sind dank Materialized Views nun im Millisekundenbereich abrufbar.

Das Dashboard beweist, dass PostgreSQL auch ohne teure Zusatztechnologien (wie Elasticsearch oder Redis) als Backend für High-Performance-Analytics dienen kann – vorausgesetzt, man nutzt die Werkzeuge der Datenbank-Engine korrekt.