

Bericht Datenbankprojekt

Datenbanksysteme, FS24

JobScout

Team XX



13.06.2024

1. Zugangsdaten.....	4
1.1. PostgreSQL.....	4
1.2. MongoDB.....	4
1.3. Grafana.....	4
2. Einführung & Ausgangslage.....	5
3. Projektidee & Use Cases.....	6
3.1. Entscheidungsunterstützung für Wertgenerierung.....	6
3.2. Persona - Daniel, engagierter Software-Engineer.....	7
3.3. Erforderliche Datenquellen für Analyse.....	8
3.4. Methode zur Berechnung der Kennzahlen für die Entscheidungshilfe.....	8
3.5. Datenbanktechnologien.....	9
4. Datenmodell & Datenbankschema.....	9
4.1. Datenmodell, Abstrakt.....	9
4.2. Datenbankschema SQL.....	11
4.3. Datenbankschema NoSQL.....	12
5. Daten laden und transformieren.....	13
5.1. Überblick Systemarchitektur.....	13
5.2. PostgreSQL.....	13
5.2.1. Daten Laden.....	14
5.2.2. Daten Parsen.....	15
5.2.3. Daten Normalisieren und Transformieren.....	16
5.3. MongoDB.....	17
5.3.1. Daten Laden.....	18
5.3.2. Daten Transformieren.....	18
5.3.3. Daten Cleanup.....	22
6. Daten analysieren & auswerten.....	23
7. Abfragen.....	26
7.1. SQL Abfragen.....	26
7.1.1. Abfrage mit «job_location» und einem «unique_job_skill».....	26
7.1.2. Abweichung von Bewertungskriterien im Punkt Komplexität.....	28
7.2. NoSQL Abfragen.....	29
8. Effizienz & Performance.....	31
8.1. Materialized Views.....	31
8.2. SQL Performance.....	31
8.2.1. EXPLAIN und ANALYZE.....	31
8.2.2. Visualisieren des Ausführungsplans.....	34
8.2.3. Optimierungen der SQL Datenbank.....	35
8.3. NOSQL Performance.....	40
8.3.1. Basismessung.....	40
8.3.2. Verbesserungspotenzial.....	41
9. Datenschutz & Datenbanksicherheit.....	43
9.1. Analyse Schwachpunkte.....	43
9.2. Sicherheit SQL.....	44

9.2.1. Credentials & Rechte.....	44
9.2.2. Datenverlust, Backup und Wiederherstellung.....	46
9.3. Sicherheit NoSQL.....	46
9.3.1. DB-Zugriff.....	46
9.3.2. Credentials & Rechte.....	47
9.3.3. Datenverlust.....	48
9.4. Grafana.....	48
10. Visualisierung & Entscheidungsunterstützung.....	49
10.1. Dashboard Grafana.....	49
10.2. Konfigurieren Panels.....	50
10.2.1. Panel 1 - Detailed View.....	50
10.2.3. Panel 2 - Map.....	51
10.3. Variablen.....	52
10.4. SQL-Panel Spezifisches.....	53
10.4.1. Transformation für GeoMap.....	53
10.4.2. SQL Query.....	55
10.5. NOSQL-Panel Spezifisches.....	56
10.5.1. Re-Transformation Dokumente NoSQL.....	56
10.5.2. NoSQL Query.....	57
10.6. Entscheidungsempfehlung Personas.....	58
11. Fazit & Lessons Learned.....	60
12. Quellen und Abbildungsverzeichnis.....	61
12.1. Quellenverzeichnis.....	61
12.2. Abbildungsverzeichnis.....	62

1. Zugangsdaten

Disclaimer:

Die Infrastruktur des "████████"-Servers wird von einem Mitstudenten bereitgestellt, da aufgrund des Infrastruktur-Setups kein Remote-Zugriff auf den PostgreSQL-Server auf der Lab Services-VM möglich ist. (Requests auf den Port █████ werden wahrscheinlich durch NAT vorgefiltert und somit ausgesteuert)

1.1. PostgreSQL

IP	Port	Username	Passwort
████████	████	postgres	Wird nicht angegeben
████████	████	full_access_user	Wird nicht angegeben
████████	████	readonly	Wird nicht angegeben
████████	████	read_only_user	████████

1.2. MongoDB

IP	Port	Username	Passwort
████████	████	root	Wird nicht angegeben
████████	████	Backupper	Wird nicht angegeben
████████	████	Developer	Wird nicht angegeben
████████	████	Grafana	Wird nicht angegeben
████████	████	readonly	████████

ConnectionString für MongoDB Compass:

1.3. Grafana

IP	Rolle	Username	Passwort
████████	Admin	████████	Wird nicht angegeben
████████	Readonly	readonly	████████

2. Einführung & Ausgangslage

Die stetige Digitalisierung macht auch vor dem HR-Sektor und dem Jobmarkt nicht halt. Das digitale Publizieren von Job-Inseraten ist in vielen Business-Sektoren bereits zum Standard geworden - Firmen posten ihre Inserate meist in mehrfacher Ausführung auf verschiedenen Plattformen gleichzeitig. Die dadurch entstandene Flut an Inseraten kann bereits die Suche nach passenden Jobs zu einer langwierigen Angelegenheit gestalten, geschweige denn das effiziente Herausfiltern der geforderten Fähigkeiten und deren Abgleich mit dem eigenen Profil.

Mit unserem Projekt möchten wir genau jene Herausforderung angehen. Durch die strukturierte Analyse von (LinkedIn-)Job-Inseraten werden wir eine innovative Datenbankanwendung entwickeln, welche es den Usern erlaubt, mit wenigen Klicks passende Job-Inserate zu finden. Dabei werden die individuellen Fähigkeiten des Users und die gewünschte Arbeitsregion bestmöglich in die Vorschläge miteinbezogen.

Mit einem Fokus auf ein einfaches und stimmiges Benutzererlebnis wird diese Anwendung zu einem essentiellen Tool für alle Jobsuchenden, die ihre wertvolle Zeit nicht mit endlosen Scrollen und Ansehen von irrelevanten Inseraten vergeuden möchten.

Die Mitglieder des Projektteams waren auch schon in der Situation, dass die Suche nach einem passenden Job-Inserat zur Geduldsprobe wurde und wären auch über ein solches Tool froh gewesen. Deshalb sind wir motiviert, mit dem Projekt *JobScout*, der mühsamen und zeitaufwendigen Job-Suche Abhilfe zu schaffen.

3. Projektidee & Use Cases

Unsere Projektidee zielt darauf ab, Personen zu unterstützen, die auf der Jobsuche sind oder sich im Arbeitsmarkt neu orientieren möchten und somit einen Mehrwert bieten. Anhand von Datenanalysen und Visualisierungen wollen wir verschiedene konkrete Use-Cases erarbeiten, deren Umsetzung letztlich das gesteckte Ziel erreicht. Die erarbeiteten Use-Cases werden im folgenden Unterkapitel genauer beleuchtet.

3.1. Entscheidungsunterstützung für Wertgenerierung

In der heutigen Zeit hält die Digitalisierung auf dem Jobmarkt immer mehr Einzug. Eine direkte Folge dessen ist, dass ein Grossteil der Publikationen in digitaler Form auf verschiedenen Plattformen (LinkedIn, Xing, Jobs.ch etc.) verläuft.

Nach einer detaillierten Analyse und verschiedenen Aufbereitungen konnten wir verschiedene Use Cases identifizieren. Im Rahmen von unserem Projekt liegt der Fokus auf folgendem Anwendungsfall:

1. Personalisierte Jobempfehlungen anhand von Skills und Region

Basierend auf den Skills und dem gewünschten Arbeitsort/Region werden dem User die am besten passenden Job Inseraten empfohlen. Hierbei werden die Jobs mit der höchsten Übereinstimmung am besten bewertet und somit dem User zuerst angezeigt. Damit erlaubt man dem Enduser eine einfache, übersichtliche und schnelle Art, die passenden Job Inserate zu finden.

Zusätzlich zu unserem primären Use Case haben wir noch weitere potentielle Anwendungsfälle gefunden, welche jedoch im Rahmen von diesem Projekt nicht realisiert werden.

2. Analyse der gefragtesten Skills für spezifische Job-Kategorien

Job-Inserate könnten nach Job-Kategorie gruppiert und die gefragten Skills extrahiert werden. Somit könnte man Recruiting-Agenturen oder auch Personen, welche sich neu orientieren möchten, einen besseren Einblick geben, welche Skills gerade beispielsweise in der IT- oder im Verkaufssektor gefragt sind.

3. Identifizieren von Skill-Gaps für spezifische Positionen

Ähnlich wie beim vorhergehenden Use-Case, könnten die gesammelten Daten auch genutzt werden, um zu identifizieren, welche Skills am gefragtesten für spezifische Positionen sind. Hiermit sieht man auch automatisch von welchen Skills der Markt noch nicht gesättigt ist und Weiterbildungs- & Kurs-Anbieter könnten diese Daten nutzen, um ihr Ausbildungspotfolio laufend dem Bedarf anzupassen.

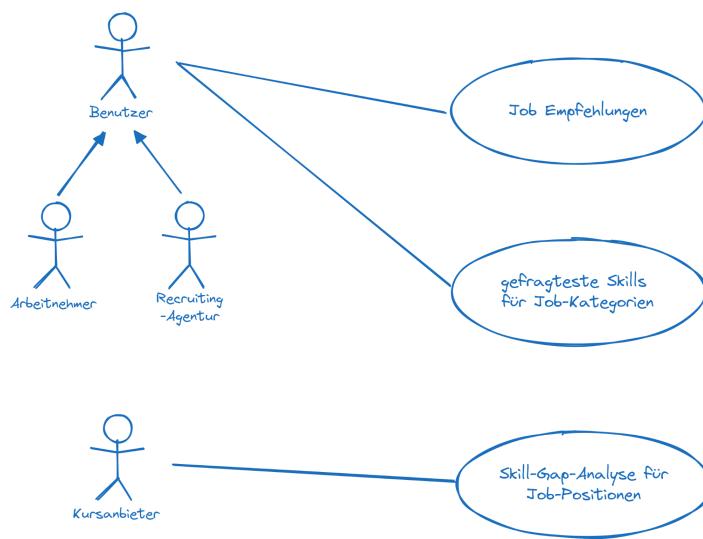


Bild 1 - Use Case Diagramm

3.2. Persona - Daniel, engagierter Software-Engineer



Bild 2 - Persona Daniel (Bild von [Dzmitry Dudov](#) auf [Unsplash](#))

Daniel ist 28 Jahre alt. Er ist gebürtiger US-Amerikaner und ist für sein Studium in die Schweiz gezogen. Konkret hat er Informatik an der ETH Zürich studiert und vor fünf Jahren seinen Abschluss gemacht. Seitdem arbeitet er als Softwareentwickler in verschiedenen Unternehmen und ist Senior Software Engineer mit einem Fokus auf hardwarenahe Systeme mit zeitkritischen Sensordaten. Daniel verbringt gerne Zeit mit Freunden, entdeckt neue Restaurants und Cafés. Im Sommer ist er gelegentlich auch mal bei einem Festival oder Konzert zu finden. Er schätzt seinen aktiven Lifestyle mit einem ausgewogenen Verhältnis zwischen Arbeit und Freizeit und ist daher stets daran interessiert, in seiner Karriere voranzukommen, ohne dabei sein persönliches Leben zu vernachlässigen. Als berufstätige Person hat Daniel nicht wirklich viel Zeit, um sich intensiv mit der Jobsuche zu beschäftigen. In seinem letzten Familienbesuch in den USA hat er seine derzeitige Freundin kennengelernt. Seitdem besteht sein Wunsch, wieder zurück in die USA und mit seiner Freundin zusammenzuziehen. Er sucht nach einer effizienten Möglichkeit, relevante Stellenangebote zu finden, die seinen Fähigkeiten und Präferenzen entsprechen, ohne dabei Stunden mit dem Durchsuchen von Jobportalen zu verbringen.

Unser Projekt zielt darauf ab, Personen wie Daniel bei der effizienten (internationalen) Jobsuche zu unterstützen. Somit können sie ihre wertvolle Zeit sinnvoll nutzen, anstatt zu versuchen, die Nadel in einem schier endlos grossen "Heuhaufen" von Jobinseraten zu finden.

3.3. Erforderliche Datenquellen für Analyse

Für den gewählten Use-Case benötigen wir Daten über Jobinserate. Spezifisch sollen jene auch die benötigten Skills und den Ort des Arbeitgebers beinhalten. Zusätzlich wäre es von Vorteil, wenn die Daten auch die Plattform und eine Form von eindeutigen Identifier beinhalten würden, damit man später den User bei Interesse direkt zum Inserat weiterleiten kann.

Im Projekt arbeiten wir mit LinkedIn Jobinseraten - eine der weltweit grössten Plattformen für den Arbeitsmarkt. Die hierzu verwendeten Daten stammen aus folgendem Datenset, welches vom Autor ASANICZKA über Kaggle zur Verfügung gestellt wird:

<https://www.kaggle.com/datasets/asaniczka/1-3m-linkedin-jobs-and-skills-2024>
(zuletzt aufgerufen am 10.06.2024)

Das 6.19GB grosse Dataset besteht aus den folgenden drei CSV-Dateien:

- job_skills.csv
Kommagetrennte List mit allen geforderten Skills, Jobinserat kann via Link aufgelöst werden
- job_summary.csv
Prosatexte der Inserate selbst, Jobinserat kann via Link aufgelöst werden
- linkedin_job_postings.csv
Alle Daten, welche nebst Skills und Zusammenfassung über ein Inserat gesammelt wurden (z.B. Location, Jobtitel etc.)

Insgesamt sind 1'348'454 Jobinserate in diesem Dataset enthalten. Hierbei repräsentieren diese Inserate nur die vier englischsprachigen Länder: Kanada, Grossbritannien, Australien und die USA. Wovon die USA den grössten Anteil mit rund 85% der Job Inserate besitzt. In dieser Hinsicht limitiert uns das Dataset, dass primär nur der Jobmarkt der USA gut abgedeckt werden kann.

In Relation zum gesamten Dataset sind die Zusammenfassungen der Inserate mit rund 5.1GB Prosatext riesig. Dies könnte zu einem späteren Zeitpunkt zu einer Herausforderung für die Performance und oder die DB-Server-Infrastruktur generell werden. Zusätzlich sind dies unstrukturierte Daten (Text), aus welchen wahrscheinlich nur schwer einen Nutzen für einen Use-Case im Rahmen der Möglichkeiten einer DB gezogen werden kann.

Im Kapitel 6 wird das Dataset noch detaillierter analysiert für die Anwendung in Verbindung mit dem gewählten Use-Case.

3.4. Methode zur Berechnung der Kennzahlen für die Entscheidungshilfe

Nach der Analyse unserer Daten sind wir auf folgende Herausforderungen gestossen:

- Fachlich identische Skills der Inserate haben teils Tippfehler und oder sind anders geschrieben
- Inserate können sich in verschiedenen Eigenschaften unterscheiden

Die Prinzipien der *Levenshtein-Distanz* und *Hamming-Distanz* ermöglichen es uns, ähnliche Skills zu finden, basierend auf der Benutzereingabe und zusätzlich auch potenzielle Tippfehler innerhalb der Daten auszugleichen.

Mit dem Prinzip der *euklidischen Distanz* ermöglicht es uns, eine Ähnlichkeit zwischen Inseraten zu berechnen und basierend darauf Empfehlungen abzugeben.

3.5. Datenbanktechnologien

Für die Umsetzung dieses Projekts sollten zwei konkrete Datenbanken ausgewählt werden - eine dokumentenorientierte (NoSQL) und eine relationale Datenbank.

Die Wahl für die NoSQL-Datenbank fiel schnell auf den Marktführer im Bereich der dokumentenorientierten Datenbanken - MongoDB. Da die Teammitglieder in diesem Bereich noch limitierte Erfahrungen haben, passt dies auch gut, da MongoDB ebenfalls Teil des DBS-Curriculums ist und wir so die vermittelten Inhalte im Gleichen praktisch vertiefen konnten.

Bei der Entscheidung der relationalen Datenbank waren primär MySQL und PostgreSQL im Rennen. Beides sind mitunter die populärsten open source relationalen DBs auf dem Markt. Im Vergleich bietet MySQL ein einfacheres Setup und im Allgemeinen eine etwas simplere Handhabung. PostgreSQL brilliert auf der anderen Seite mit mehr Datentypen, Materialized Views, mehr Index-Typen und grösserer Auswahl für Sprachen zum Schreiben von Stored Procedures. Da die Teammitglieder bereits Erfahrungen mit PostgreSQL sammeln konnten und jene DB mehr Flexibilität für komplexe Applikationen resp. Datenbankabfragen bietet, fiel der Entscheid letztlich auf PostgreSQL.

Das Backend der Datenbankapplikation bildet die MongoDB- bzw. PostgreSQL-Datenbank, welche jeweils auf einem zentralen Server läuft. Darin werden die Daten gespeichert, aufbereitet und strukturiert zur Verfügung gestellt. Das Frontend übernimmt das Business-Intelligence-Tool (BI-Tool), welches sich auf die beiden Datenbanken verbindet und via Abfragen bzw. Views die nötigen Daten für die Visualisierung anziehen kann. Weitere Details zum BI-Tool siehe Kapitel 9.

4. Datenmodell & Datenbankschema

4.1. Datenmodell, Abstrakt

Um die zu verwendenden Daten zu verstehen, haben wir ein Datenmodell entwickelt. So haben wir einen Überblick über die Beziehung zwischen den Daten und die Struktur der Daten. Das Datenmodell besteht aus drei Entitäten: «*job_summary*», «*job_posting*» und «*job_skills*».

Als Haupt-Entität dient uns das «*job_posting*». Diese Entität repräsentiert einen LinkedIn Job-Post. Also ein Jobinserat für eine offene Stelle. Aus «*job_posting*» laufen zwei Beziehungen zu «*job_summary*» und «*job_skills*». Die Entität «*job_summary*» enthält weitere Informationen zum Jobinserat. Die «*job_skills*» beschreiben die benötigten Fähigkeiten, die für den Job gefragt sind.

Dieses Datenmodell erlaubt es uns, die verschiedenen Entitäten und ihre Beziehungen miteinander zu überblicken. So können wir effizient Datenbank-Abfragen tätigen. Auch ermöglicht dieses Datenmodell uns eine geeignete Struktur für beide Datenbanken zu finden, so dass die Abfragen selbst effizient und sinnvoll gegliedert werden können.

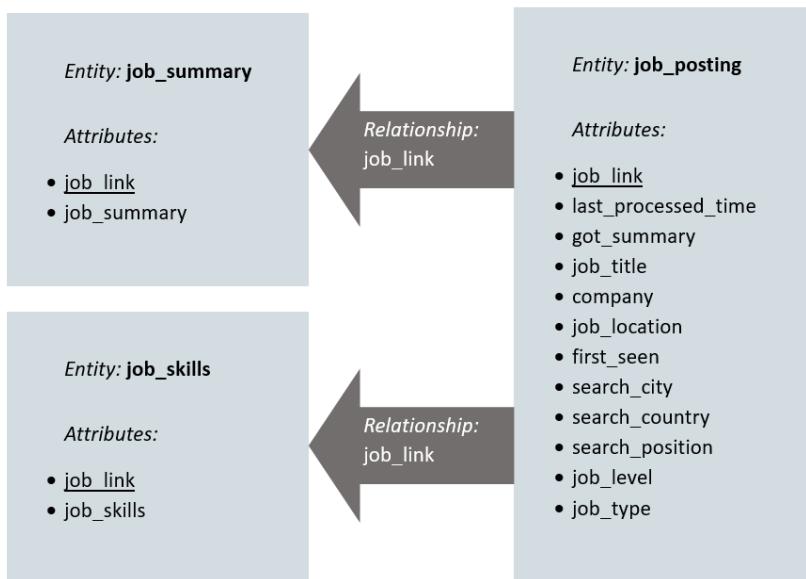


Bild 3 - Konzeptionelles Datenmodell



Bild 4 - Konzeptionelles Datenmodell nach einfacher Chen Notation

4.2. Datenbankschema SQL

Das Datenbankschema der SQL-Datenbank ähnelt sehr stark dem konzeptionellen Datenmodell mit der Ausnahme, dass die n:m Beziehung mit einer Zwischentabelle gelöst ist. Die Attribute sind noch dieselben, mit Ausnahme der Foreign-Keys und Primary-Keys.

Jede Entität besitzt im SQL-Schema einen Primary Key mit dem Typen UUID (Universal Unique Identifier). Auf die Datenstruktur und die Transformation der Daten wird im Kapitel *Daten laden und transformieren* noch genauer eingegangen.

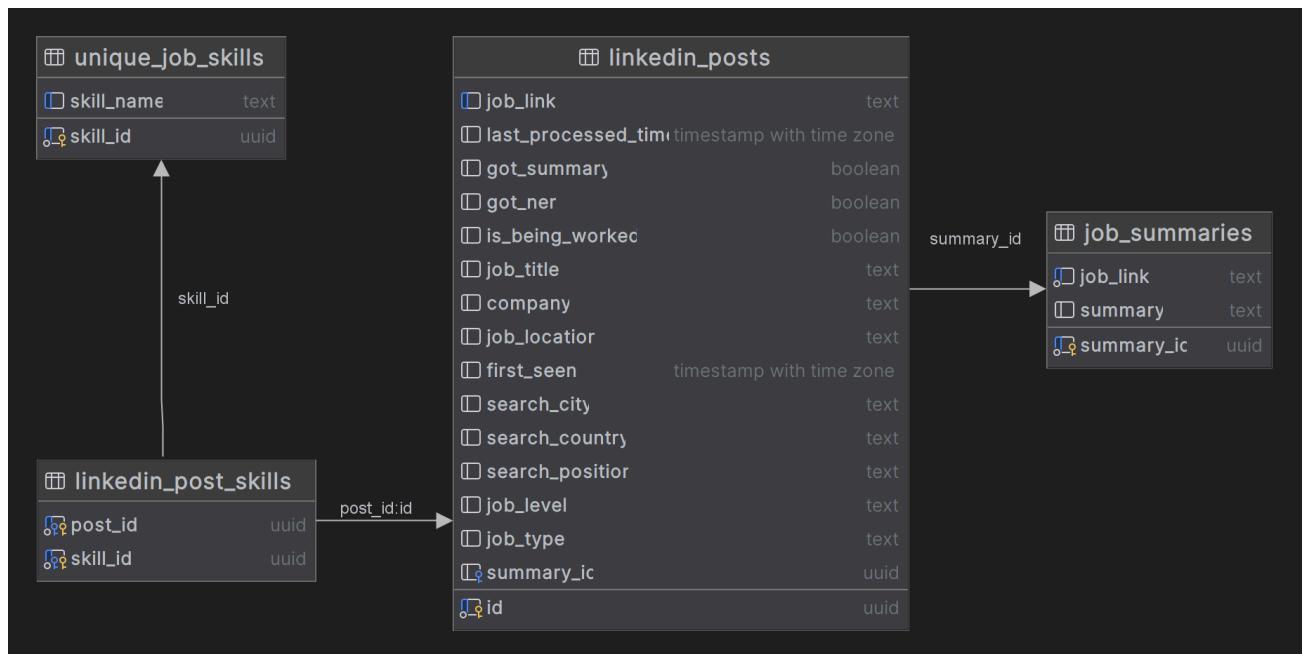


Bild 5 - SQL Datenbankschema

Attribute der Inserats-Entität (*linkedin_posts*) wie z.B. *job_location* würden normalerweise im Normalisierungsprozess in eine dedizierte Tabelle ausgelagert werden. Basierend auf dem Projektauftrag und dem gewählten Use-Case war absehbar, dass es sehr wahrscheinlich nur eine Query geben wird, welche die Inserate "en block" laden wird. Wären die Daten sauber normalisiert (3. Normalform) in verschiedenen Tabellen verteilt, müssten sie durch kostspielige Joins geladen werden. Deshalb würden jene Daten im Kapitel *Effizienz & Performance* wieder denormalisiert, um die Performance der Query zu steigern und besser auf den Use-Case zu optimieren. Deshalb hat man sich bewusst gegen eine Normalisierung entschieden und diese Attribute in einem denormalisierten Stand belassen.

Würde in der Zukunft die Datenbank erweitert oder falls die Anzahl Records der Entität *linkedin_posts* so drastisch wachsen würde, dass die Tabelle zu gross wird, müsste man den Sachverhalt nochmals analysieren und eine Restrukturierung/Normalisierung der Daten erneut in Betracht ziehen.

4.3. Datenbankschema NoSQL

Das Entity-Relationship-Document-Diagramm (ERDD), welches für die MongoDB erstellt wurde, ist simpler als das originale ERD. Dies liegt daran, dass die «*job_skills*» bei der Transformation direkt als Array von Strings in die «*job_summary*» aufgenommen wurden. Es wurde bewusst darauf verzichtet, die «*job_skills*» als ganze Objekte ins Array hinzuzufügen, da dies Komplexität und Datengröße unnötigerweise steigern würde.

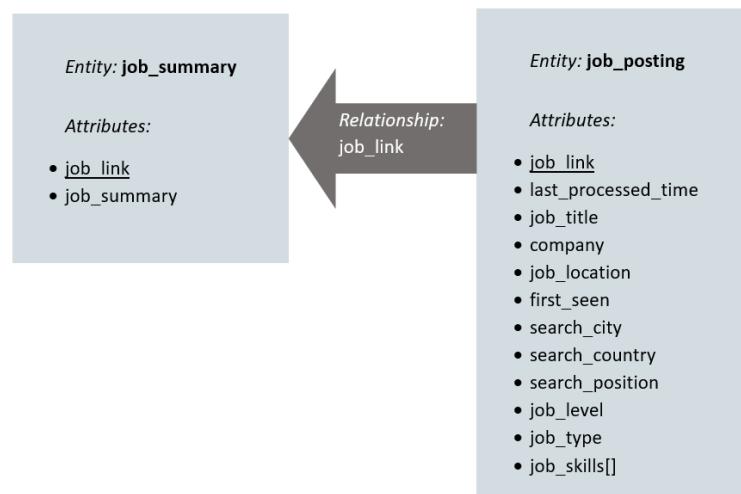


Bild X: ERDD MongoDB

Das ERDD umgesetzt als JSON Struktur sieht dann in der MongoDB folgendermassen aus. Es gibt die Dokumente «*job_posting*» und «*job_summary*».

«*job_posting*» enthält die meisten und wichtigsten Informationen für unsere Abfragen. Die «*job_summary*» Dokumente wurden aufgrund ihrer Grösse und beschränkten Relevanz für gewisse Abfragen ausgelagert.

```
1  {
2      "job_summary": {
3          "job_link": "",
4          "job_summary": ""
5      }
6 }
```

Bild 6 - NoSQL Schema job_summary

```
1  {
2      "job_posting": {
3          "job_link": "",
4          "last_processed_time": "2001-03-19 00:00:00.000000+00",
5          "job_title": "",
6          "company": "",
7          "job_location": "",
8          "first_seen": { "$date": "2001-03-19T00:00:00.000Z" },
9          "search_city": "",
10         "search_country": "",
11         "search_position": "",
12         "job_level": "",
13         "job_type": "",
14         "job_skills": ["", ...]
15     }
16 }
```

Bild 7 - NoSQL Schema job_posting

5. Daten laden und transformieren

5.1. Überblick Systemarchitektur

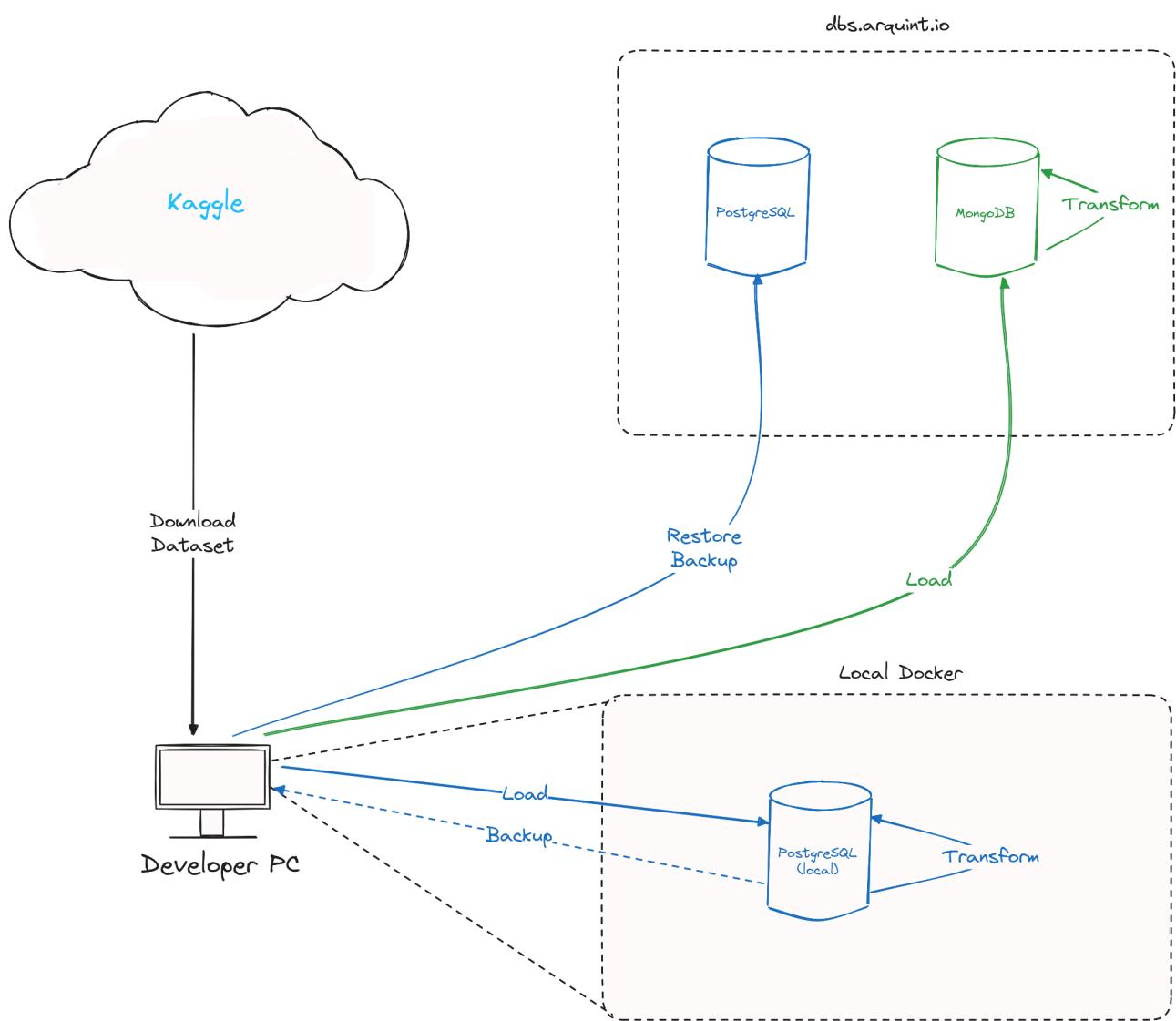


Bild 8 - Überblick Datenflüsse für Laden & Transformieren

Das ausgewählte Dataset wurde von Kaggle auf eine lokale Maschine heruntergeladen und entpackt. Anschliessend wurden die Daten jeweils in die PostgreSQL- und MongoDB-Datenbank eingelesen (*LOAD*) und entsprechend verarbeitet (*TRANSFORM*). Die einzelnen Schritte werden individuell in den folgenden Kapiteln im Detail aufgegriffen. Der wesentliche Unterschied in der Herangehensweise war, dass die Daten für die PostgreSQL Datenbank zuerst in einer lokalen Docker-Instanz verarbeitet und anschliessend als Backup auf dem effektiven System restored wurden. Dies hatte den Hintergrund, dass anfänglich die Server-Instanz von PostgreSQL einige Probleme bereitete und man keine Zeit hiermit verlieren wollte. Alle Credentials zu den jeweiligen Umgebungen sind im Kapitel *Zugangsdaten* zu finden.

5.2. PostgreSQL

Das Importieren von Daten in die PostgreSQL Datenbank wurde mittels der Import-Tools von DataGrip gelöst. Dies musste in mehreren Schritten gelöst werden, da die Daten weder die richtigen Werte für die richtigen Typen enthalten noch voll normalisiert sind. Der Datenimport erfolgt nach dem *Extract-Load-Transform*-Verfahren (ELT).

5.2.1. Daten Laden

Um die Daten in die PostgreSQL Datenbank zu laden, wurden die .csv Dateien direkt über DataGrip in die entsprechenden Tabellen importiert. Diese Import-Tabellen haben wir bei allen Spalten mit dem Typen **varchar** (also als Text) erstellt, damit wir uns selbst um das Parsing kümmern können und so volle Kontrolle darüber haben.

Bild 9 - Schema von Import-Tabelle für LinkedIn Job Posts

job_link	last_processed_time	got_summary	got_ner	is_being_worked	job_title	company
https://www.linkedin.com/jobs/1234567890	2024-01-19 16:57:42.281900	t	t	f	Long Term (Travel) M...	TravelNurse...
https://www.linkedin.com/jobs/1234567891	2024-01-19 16:57:42.290400	t	t	f	OPERATIONS ASSISTANT...	Dollar Tree...
https://www.linkedin.com/jobs/1234567892	2024-01-19 16:57:42.415500	t	t	f	Dynamic Stretch Spec...	Life Time I...
https://www.linkedin.com/jobs/1234567893	2024-01-19 16:57:42.451500	t	t	f	Registered Nurse Cli...	Palmetto In...
https://www.linkedin.com/jobs/1234567894	2024-01-19 16:57:42.925500	t	t	f	Jerry's Foods Edina ...	Jerry's Ent...
https://www.linkedin.com/jobs/1234567895	2024-01-19 16:57:43.336400	t	t	f	Procurement Buyer II	Westrock Co...
https://www.linkedin.com/jobs/1234567896	2024-01-19 16:57:44.195900	t	t	f	Tunnel Inspection En...	Stantec...
https://uk.linkedin.com/jobs/1234567897	2024-01-19 16:57:45.711600	t	t	f	Head of School of Law	Omni...
https://www.linkedin.com/jobs/1234567898	2024-01-19 16:57:47.230300	t	t	f	Tropicalist Brand Am...	Cerveceria ...
https://www.linkedin.com/jobs/1234567899	2024-01-19 09:45:09.215800	f	f	f	Sr. Systems Engineer...	ClearanceJo...
https://www.linkedin.com/jobs/1234567900	2024-01-21 00:49:49.218200	t	t	f	Travel RN Endoscopy/...	TravelNurse...
https://www.linkedin.com/jobs/1234567901	2024-01-21 00:49:49.506200	t	t	f	Senior Linux Platfor...	Bose Corpor...
https://ca.linkedin.com/jobs/1234567902	2024-01-21 07:14:26.590000	t	t	f	Analyste-Développeur...	TEHORA inc.
https://uk.linkedin.com/jobs/1234567903	2024-01-21 07:40:09.596200	t	t	f	Senior Ecologist	Penguin Rec...
https://www.linkedin.com/jobs/1234567904	2024-01-21 07:40:10.492800	t	t	f	Manager of Telemetry	A-Line Staf...

Bild 10 - Import-Tabelle für LinkedIn Job Posts mit geladenen Daten

Bild 9 - Schema von Import-Tabelle für LinkedIn Job Posts

Wie man hier sieht, sind nicht alle Daten bereits im richtigen Format für die PostgreSQL Datenbank. Die booleschen Werte zum Beispiel sind als **t** für **true** und **f** für **false** gesetzt.

Gross-Klein Schreibung

Nun gibt es noch das Problem, dass im DataSet leider gewisse Daten doppelt vorkommen, nur leicht anders geschrieben werden.

Ability to operate vehicles and equipment
Ability to Operate Vehicles and Equipment

Bild 11 - Doppelte Elemente in Job Skills Tabelle (Gross-Kleinschreibung)

Wir sehen hier, dass dies genau der gleiche Skill wäre, wenn die Gross-Kleinschreibung auch gleich wäre. Dieses Problem kann man gut lösen, da wir dafür einfach die ganzen Texte entweder **toLowerCase** oder mit **toUpperCase** vergleichen können. Etwas, das aber problematisch ist, ist, dass wir nun auch die ganzen Einträge in der **n:m Tabelle (linkedin_post_skills)** re-mappen müssen auf den jeweiligen Datensatz, der nach dem Löschen der Duplikate bestehen bleibt. Dies ist Ressourcen- und Zeitintensiv, könnte aber auch mithilfe eines SQL-Skripts gemacht werden.

In einem ersten Versuch mit einem SQL-Skript konnte man auch nach 6 Stunden Skript-Laufzeit nicht alle Duplikate löschen und re-mappen. Somit mussten wir uns einen neuen Weg suchen, dieses Problem zu lösen.

Nach mehreren Anläufen, die Duplikate durch SQL-Skripte zu entfernen, die nicht zum gewünschten Ergebnis geführt haben, haben wir beschlossen, unsere Strategie in dieser Angelegenheit zu überdenken.

Da wir einige der Daten nicht mittels Skripts endgültig normalisieren können, werden wir bei den Abfragen mittels Algorithmen arbeiten, um ähnliche Einträge in den Skills auch in Betracht zu ziehen. Dies kann man mittels z.B. der Levenshtein-Distanz, der Hamming-Distanz, des Jaccard-Index, Cosinus-Ähnlichkeit oder der Jaro-Winkler-Distanz erkennen. Dazu mehr im dazugehörigen Kapitel 6 *Daten analysieren und auswerten*.

5.2.2. Daten Parsen

Die rohen Daten wurden zunächst in Import-Tabellen geladen und anschliessend mittels SQL-Skripten in neue Tabellen überführt, wobei die Daten in die korrekten Typen konvertiert wurden.

```
INSERT INTO linkedin_posts (
    job_link,
    ...,
    job_type
)
SELECT
    job_link,
    last_processed_time::TIMESTAMPTZ,
    (got_summary = 't')::BOOLEAN,
    (got_ner = 't')::BOOLEAN,
    (is_being_worked = 't')::BOOLEAN,
    ...,
    first_seen::TIMESTAMPTZ,
    ...,
FROM
    import_linkedin_posts;
```

5.2.3. Daten Normalisieren und Transformieren

Wir haben in unserem DataSet von Kaggle drei Tabellen. Zwei davon, die für die LinkedIn Posts und die für die Jobbeschreibungen, sind bereits normalisiert. Die dritte Tabelle, die für die Skills zuständig ist, weist noch keine normalisierte Struktur auf. Nach dem Import der Daten befinden sich in derselben Spalte noch mehrere Skills, durch Kommas getrennt, auf.

Auch hier wieder gehen wir mittels eines SQL-Skripts vor:

```
DO $$  
DECLARE  
    rec RECORD;  
    _skill_id UUID;  
BEGIN  
    CREATE INDEX ON import_job_skills (job_link);  
    CREATE INDEX ON linkedin_posts (job_link);  
  
    -- Create a new table for unique skills  
    CREATE TABLE unique_job_skills (  
        skill_id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),  
        skill_name TEXT UNIQUE  
    );  
  
    -- Create an associative table to map skills to linkedin_posts  
    CREATE TABLE linkedin_post_skills (  
        post_id UUID,  
        skill_id UUID,  
        PRIMARY KEY (post_id, skill_id),  
        FOREIGN KEY (post_id) REFERENCES linkedin_posts (id) ON DELETE CASCADE,  
        FOREIGN KEY (skill_id) REFERENCES unique_job_skills (skill_id) ON DELETE CASCADE  
    );  
  
    -- Populate the unique_skills table with unique skills  
    FOR rec IN SELECT DISTINCT unnest(string_to_array(job_skills, ',')) AS skill, job_link  
        FROM import_job_skills  
    LOOP  
        rec.skill := TRIM(rec.skill);  
  
        -- Attempt to insert the skill if it doesn't exist  
        INSERT INTO unique_job_skills (skill_name)  
        VALUES (rec.skill)  
        ON CONFLICT (skill_name) DO NOTHING;  
        SELECT skill_id INTO _skill_id FROM unique_job_skills WHERE skill_name = rec.skill;  
  
        -- Insert the association into the junction table, with the correct post_id from  
        linkedin_posts  
        INSERT INTO linkedin_post_skills (post_id, skill_id)  
        SELECT id, _skill_id  
        FROM linkedin_posts  
        WHERE job_link = rec.job_link  
        ON CONFLICT DO NOTHING;  
    END LOOP;  
END;  
$$;
```

Auch wurde bei der Transformation der Daten der Primärschlüssel der Tabellen geändert. Anstelle des Job-Links, also der URL als Text, gibt es nun für jedes LinkedIn Posting, sowie jede Jobbeschreibung und jeden Skill eine **UUID (Universally Unique Identifier)**. Auch die **n:m** Tabelle zwischen Skills und den

LinkedIn Posts ist dabei neu dazugekommen. Deren **PK** (Primary Key) besteht aus den beiden **FK's** (Foreign Keys).

Später wurde dann noch klar, dass auf der Tabelle *linkedin_posts* die Spalten *got_ner*, *got_summary* und *is_being_worked* keinerlei Nutzen für uns beinhalten und sie somit gelöscht werden können. Jene sind höchstwahrscheinlich Relikte vom Scraper, welcher zur Datensammlung vom Dataset Autor verwendet wurde. Die Spalten wurden mit folgendem Snippet aus dem Schema gelöscht:

```
ALTER TABLE linkedin_posts DROP COLUMN got_ner;
ALTER TABLE linkedin_posts DROP COLUMN got_summary;
ALTER TABLE linkedin_posts DROP COLUMN is_being_worked;
```

Somit sieht die Datenbank nach der Transformation wie folgt aus:

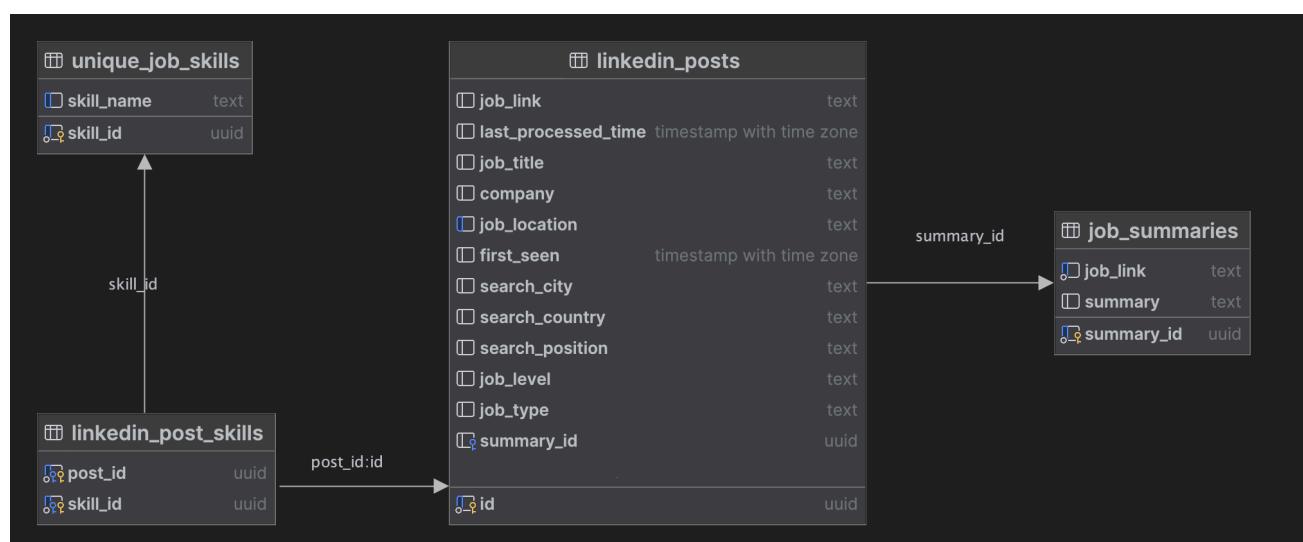


Bild 12 - Datenbankdiagramm nach Datenimport und Transformation

5.3. MongoDB

Die Daten wurden mittels dem Programm MongoDB Compass eingelesen, welches bestimmte Hindernisse des Einlesens/Konvertieren schon selbst beseitigen kann. Der Datenimport erfolgt nach dem *Extract-Load-Transform*-Verfahren (ELT).

5.3.1. Daten Laden

Um die Daten in die MongoDB zu laden, wurde für jedes CSV im Kaggle Datensatz eine "Collection" erstellt. Mithilfe der "Add-Data" Funktion des MongoDB-Compass Clients wurden dann die CSV in die jeweiligen Collections eingelesen. Die "Add-Data" Funktion scannt bei diesem Vorgang die Daten und schlägt den passenden Datentyp vor.

Somit löste sich das Problem von t & f für booleans automatisch, da Compass dies automatisch als bool erkannte und umwandelte.

```
_id: ObjectId('65fefc803b3bf0f5a693508a')
job_link : "https://www.linkedin.com/jobs/view/account-executive-dispensing-norcal..."
last_processed_time : "2024-01-21 07:12:29.00256+00"
got_summary : true
got_ner : true
is_being_worked : false
job_title : "Account Executive - Dispensing (NorCal/Northern Nevada) - Becton Dicki..."
company : "BD"
job_location : "San Diego, CA"
first_seen : 2024-01-15T00:00:00.000+00:00
search_city : "Coronado"
search_country : "United States"
search_position : "Color Maker"
job_level : "Mid senior"
job_type : "Onsite"
```

Bild 13 - Importiertes Dokument aus dem job_posting CSV

5.3.2. Daten Transformieren

Das Transformieren wurde in mehreren Schritten mit Pipelines/Aggregates direkt auf dem DB Server gelöst.

- «**job_skills**» zu array
 - In der «**job_skills**» collection die Daten aus einem kommagetrennten String in einen Array umwandeln
- «**job_skills**» vereinen mit «**job_postings**»
 - separate «**job_skills**» in die Dokumente der «**job_postings**» collection vereinen.
- «**job_summary**» vereinen mit «**job_postings**»
 - separate «**job_summary**» in die Dokumente der «**job_postings**» collection vereinen.
- **Dokument in root Objekt**
 - Die Attribute sollten nicht im Root-Object sein

«job_skills» zu array

Bei dieser Collection hat man als Ausgangslage Dokumente mit Attributen «job_link» und «job_skills». Ein Beispiel von einem «job_skills» Attribut von einem Dokument:

“Building Custodial Services, Cleaning, Janitorial Services, Materials Handling, Housekeeping, ...”

Dies ist für die Suche und das Filtern nicht effizient, daher muss es transformiert werden:

Mit einer Pipeline wurde das Attribut bei den Kommas getrennt und die Werte als Array im Attribut wieder hinzugefügt. Nun bestehen die Objekte in der «job_skills» collection aus «job_link» und einem array «job_skills».

Code	Pipeline Schritt
<pre>\$set: { job_skills: { \$split: ["\$job_skills", ", ", ""], }, }</pre>	Mit set wird ein neues Attribut names «job_skills» gesetzt (In diesem Fall überschrieben). Der Wert wird definiert durch split, welches den alten Wert nimmt und anhand „,“ aufsplittet.
<pre>\$out: { db: "DBS", coll: "job_skills", }</pre>	Mit out werden die Daten in die angegebene Collection in der DB gespeichert.

«job_skills» vereinen mit «job_postings»

Um die Abfragen zu vereinfachen und verschnellern, werden die Skills für einen Job aus der Collection «job_skills» als Array zu den Dokumenten in «job_postings» als attribut hinzugefügt.

Schwierigkeiten:

Bei diesem Schritt wurde klar, dass beim Generieren der Indexe standardmäßig **regular indexes** generiert werden und nicht **unique indexes**. Bei den knapp 1.3 Millionen Datensätzen führte das zuerst dazu, dass die Pipeline auch nach mehreren Minuten ins Timeout gelaufen ist. Nach dem Indexieren mit Unique Indexen konnte die Pipeline nach wenigen Sekunden den Task erfolgreich beenden.

Code	Pipeline Schritt
<pre>"\$lookup": { "from": "job_skills", "localField": "job_link", "foreignField": "job_link", "as": "skills" }</pre>	Mit lookup werden die Dokumente aus «job_skills» und «job_postings» (Auf dieser Collection wird die Pipeline ausgeführt) über job_link gelinkt.
<pre>"\$unwind": "\$skills"</pre>	Mit unwind wird das zuvor erstellte array skills in Dokumente gewandelt
<pre>"\$addFields": { "job_skills": "\$skills.job_skills" }</pre>	Mit addFields wird den Dokumenten in «job_postings» das Attribut job_skills mit dem vorher vorbereiteten Inhalt hinzugefügt.
<pre>"\$project": { "skills": 0 }</pre>	Mit project wird das nicht mehr benötigte Skills entfernt
<pre>"\$out": "DBS.job_postings"</pre>	Mit out werden die Daten in die angegebene Collection in der DB gespeichert.

«job_summary» vereinen mit «job_postings»

Ursprünglich war hier dasselbe geplant wie bei «job_skills». Jedoch stellt die grosse Datenmenge hier ein Problem dar. Das Summary CSV ist 5GB gross, im Vergleich sind die anderen beiden CSVs klein (nur 400 MB und 600 MB gross).

Beim Versuch, die Summaries auch als Attribut in die «job_postings» hinzuzufügen, scheiterten alle Versuche durch Timeouts, Exceptions und weiteres. Folgende zwei Optionen funktionieren bei geringeren Datenmengen:

```
1  [
2  {
3    $lookup: {
4      from: "job_summary",
5      localField: "job_link",
6      foreignField: "job_link",
7      as: "summary_docs",
8    },
9  },
10 {
11   $unwind: {
12     path: "$summary_docs",
13   },
14 },
15 {
16   $addFields: {
17     summary: "$summary_docs.job_summary",
18   },
19 },
20 {
21   $project: {
22     summary_docs: 0,
23   },
24 },
25 {
26   $out: {
27     db: "DBS",
28     coll: "job_postings",
29   },
30 },
31 ];
```

Bild 14 - Versuch pipeline für «job_summary»

```
1  db.job_postings.find({ summary: { $exists: false } })
2  .forEach(
3    function(posting)
4    {
5      var summary = db.job_summary.findOne(
6        { job_link: posting.job_link },
7        { job_summary: 1 });
8      if (summary) {
9        db.job_postings.updateOne(
10          { _id: posting._id },
11          { $set: { summary: summary.job_summary } },
12          { upsert: true }
13        );
14      }
15    }
16  );
```

Bild 15 - Versuch script für «job_summary»

Dokument in root Objekt

Wie im Unterricht erwähnt wurde, sollten die Attribute der Objekte jeweils nicht im root eines Dokumentes sein, sondern enkapsuliert in einem sammelnden Objekt. Die folgenden Screenshots zeigen die Transformation für die Collection «*job_postings*» und wurden für die Collection «*job_summary*» analog umgesetzt.

```

▼ {
  ▶ "_id": { },
  ▶ "job_link": "https://www.linkedin.com/jobs/view/account-executive-color-maker-in-san-diego-ca-junior-level-on-site-req-123456789",
  ▶ "last_processed_time": "2024-01-21 07:12:29.00256+00",
  ▶ "got_summary": true,
  ▶ "got_ner": true,
  ▶ "is_being_worked": false,
  ▶ "job_title": "Account Executive - Dispensing (NorCal, CA)",
  ▶ "company": "BDI",
  ▶ "job_location": "San Diego, CA",
  ▶ "first_seen": { },
  ▶ "search_city": "Coronado",
  ▶ "search_country": "United States",
  ▶ "search_position": "Color Maker",
  ▶ "job_level": "Mid senior",
  ▶ "job_type": "Onsite",
  ▶ "job_skills": [ ]
}

```

Bild 16 - Dokumentstruktur vor Transformation

```

▼ {
  ▶ "_id": { },
  ▶ "job_posting": {
    ▶ "job_link": "https://www.linkedin.com/jobs/view/account-executive-color-maker-in-san-diego-ca-junior-level-on-site-req-123456789",
    ▶ "last_processed_time": "2024-01-21 07:12:29.00256+00",
    ▶ "got_summary": true,
    ▶ "got_ner": true,
    ▶ "is_being_worked": false,
    ▶ "job_title": "Account Executive - Dispensing (NorCal, CA)",
    ▶ "company": "BDI",
    ▶ "job_location": "San Diego, CA",
    ▶ "first_seen": { },
    ▶ "search_city": "Coronado",
    ▶ "search_country": "United States",
    ▶ "search_position": "Color Maker",
    ▶ "job_level": "Mid senior",
    ▶ "job_type": "Onsite",
    ▶ "job_skills": [ ]
  }
}

```

Bild 17 - Dokumentstruktur nach Transformation

Dies wurde mit der folgenden Pipeline erreicht:

Code	Pipeline Schritt
<pre>\$replaceRoot: { newRoot: { _id: "\$_id", job_posting: "\$\$ROOT", }, }</pre>	Mit replaceRoot wird ein neues Root Object erstellt und das Alte ersetzt. Die Id des alten Objektes wird übernommen.
<pre>\$project: { "job_posting._id": 0, }</pre>	Mit project wird das alte <i>id</i> -Attribut im inneren Objekt weggelassen.
<pre>\$out: "test"</pre>	Mit out werden die Daten in die angegebene Collection in der DB gespeichert.

5.3.3. Daten Cleanup

Da im Kapitel [Daten Clean-Up](#) des SQL-Teils entschieden wurde, die Daten nicht zu säubern, wurde dies auch im NoSQL-Teil unterlassen. So sind die Daten konsistent zwischen den beiden Ansätzen.

6. Daten analysieren & auswerten

Der Use Case 1 beschreibt:

"Personalisierte Jobempfehlungen anhand von Skills und Region"

Basierend auf den Skills und dem gewünschten Arbeitsort/Region werden dem User die am besten passenden Job Inseraten empfohlen. Hierbei werden die Jobs mit der höchsten Übereinstimmung am besten bewertet und somit dem User zuerst angezeigt. "Damit erlaubt man dem Enduser eine einfache, übersichtliche und schnelle Art, die passenden Job Inserate zu finden."

Bei diesem Use Case soll es dem User letztendlich erlaubt werden, einige seiner Skills einzugeben, basierend auf welchen Jobs mit der höchsten Relevanz (Anzahl übereinstimmender Skills) gesucht werden. In einer "perfekten" Welt würde dies in einer relativ simplen Abfrage resultieren - die Skills würden ohne Synonyme, in einer uniformen Schreibweise (Trennzeichen etc.) und ohne Schreibfehler sowohl von Firmen in Jobinseraten gelistet als auch von Usern in die Applikation eingegeben und Jobs könnten somit mit einer einfach Gleichheits-Überprüfung der eingegebenen Skills gefunden werden.

Leider leben wir nicht in einer perfekten Welt und es geschehen allerlei Fehler bzw. Differenzen, die sich auch in unseren Daten widerspiegeln. Die folgenden Abschnitte analysieren diese und zeigen auf, wie diese Herausforderungen konzeptionell überwunden werden können. Die konkreten technischen Umsetzungen werden in den entsprechenden DB-spezifischen Unterkapiteln aufgezeigt.

Gross-/Kleinschreibung

Je nachdem, ob der Autor einen Skill als vollständigen Satz geschrieben hat oder nicht, unterscheidet sich ggf. die Kapitalisierung der Anfangsbuchstaben verschiedener Wörter.

Diese Herausforderung lässt sich relativ trivial lösen, indem man den User-Input und die Skills in der DB in einen uniformen Schreibweise (entweder gross oder klein) für die Abfrage transformiert. Theoretisch verliert man Informationen bei dieser Operation, jedoch sollte dies nur (branchen-)spezifische Akronyme sein, welche auf eine spezifische Kapitalisierung der Buchstaben setzen. Die Anzahl jener Sonderfälle sollte jedoch so gering sein, dass sie im grossen Ganzen das Verwerfen vom Gewinn der unabhängigen Gross-/Kleinschreibung nicht rechtfertigen würde.

Wortauftrennungen

Gibt ein User seinen Skill "VueJS" ein, so meint er inhaltlich dasselbe, wie ein Recruiter, welcher in einem Jobinserat "Vue JS" als benötigten Skill aufgelistet hat.

Hierbei steht man vor der Herausforderung, welche sich relativ trivial durch ein Auftrennen des User Inputs in einzelne Wörter/Token lösen lässt. Grundsätzlich lassen sich auch die Daten auftrennen, jedoch würde dies zu einer erheblich grösseren Datenmenge und einem wiederkehrenden Aufwand führen (für jeden Datenimport müsste dies als Preprocessing-Operation durchgeführt werden). Zusätzlich hilft die Auftrennung der User Inputs für die Skills in der einfacheren technischen Implementation - hat man ein Teilwort, welches der User sucht, so kann man dessen Existenz in einem Job Posting Skill einfacher prüfen, als das Inverse ob ein potentielles Wort von einem Skill teil von einem zusammengesetzten Wort im User input ist.

Synonyme

Je nachdem, in welcher Branche sich der ausgeschriebene Job befindet, gibt es eine andere Fachsprache mit verschiedenen Synonymen für eine Begrifflichkeit. Oder auch generell kann es solche Synonyme auch durch verschiedene sprachliche Schreibweisen geben. In einer altdeutschen Schreibweise würde beispielsweise folgender Skill formuliert: "Software Entwicklung in ...". Derselbe Skill würde hingegen in einer neudeutschen Formulierung (Verwendung von englischen Begriffen in die Sprache) wie folgt lauten: "Software Development in ..".

Diese Herausforderung ist nicht leicht zu meistern, da es keine allgemeine technische Lösung gibt. Synonyme oder auch anderssprachige Begriffe (mit derselben Bedeutung) müssten durch spezifisches Branchen Know-How ermittelt und die spezifischen Auflösungen explizit erfasst werden. Eine andere Variante wäre der Einsatz eines dafür trainierten Machine-Learning-Models, welches die Synonyme versteht und auflösen kann. Beide Ansätze würden einen erheblichen Mehraufwand darstellen, welchen

man betreiben könnte, falls man diese Applikation auf den Markt bringen möchte. Im Kontext von diesem Projekt erachten wir dies als Out-of-Scope und werden diese Herausforderung somit nicht beachten. Es wird vorausgesetzt, dass ein User die konkreten Begriffe und etwaige Synonyme von selbst in der Abfrage eingibt.

Schreib- & Tippfehler

Ob beim User während der Eingabe seiner Skills oder beim Recruiter während des Schreibens des Inserats - Tippfehler sind schnell passiert. Zum Beispiel ist der Skill "Entwicklung von ..." nicht korrekt, da ein c fehlt, dennoch ist "Entwicklung von ..." gemeint.

Um diese Kategorie von Problemen zu lösen, bedarf es einer Abschätzung, was gemeint sein könnte. In der Informatik sind diese Techniken unter dem Begriff *Fuzzy Matching* bekannt. Allgemein wird hierbei eine Ähnlichkeit zweier Wörter erstellt - je grösser die Ähnlichkeit, desto grösser die Wahrscheinlichkeit, dass dieses Wort gemeint sein könnte. Folgende Algorithmen werden oft als Basis für Fuzzy String Matching eingesetzt und wurden als Lösung für diese Problemstellung evaluiert:

- Hamming Distanz
Beschreibt die Anzahl Zeichen, in welchen sich zwei Wörter unterscheiden. In anderen Worten zählt diese Metrik die notwendigen Substitutionen, welche nötig sind, um ein Wort zum anderen zu transformieren. Da jedoch nur gleich lange Wörter miteinander verglichen werden können, ist dieser Algorithmus nicht wirklich anwendbar für diesen Use-Case.
- Levenshtein Distanz
Ist eine Metrik, welche beschreibt, wie viele Operation notwendig sind, um einen String zu einem anderen zu konvertieren. Die möglichen Operationen sind: Einfügen, Entfernen und Ersetzen. Wäre gut grundsätzlich gut für unseren Use-Case anwendbar, jedoch könnten hiermit nur effektive Vergleiche von einzelnen Wörtern gemacht werden. Die Skills der Job-Inserate sind jedoch vielfach Teil- oder ganze Sätze.
- Damerau-Levenshtein Distanz
Stellt eine Erweiterung von der Levenshtein Distanz dar. Falls zwei nebeneinander stehende Zeichen vertauscht sind, so zählt dieser Algorithmus die Umstellung als eine Operation im Vergleich zu zwei Operationen, welche von der Standard-Implementation gezählt würden. Dieser Algorithmus wäre für unseren Use-Case etwas besser geeignet als die Standard Levenshtein Distanz, da Tippfehler resp. Zeichendreher besser abgedeckt werden. Verfügt jedoch über dieselben Unzulänglichkeiten wie die Basis-Variante.
- Longest Common Subsequence
Dieser Algorithmus sucht eine möglichst lange Wortfolge, welche in beiden Strings enthalten ist. Konkret wird mit Einfüge- & Entfernungs-Operations jene eruiert. Diese Metrik würde sich ideal für unseren Use-Case eignen, da hiermit der User-Input (ein Wort bis Teilsatz) sich direkt mit den Skills in den Job-Inseraten vergleichen lassen. Je länger die gefundene Sequenz, desto besser würde ein Skill zu der Suchanfrage eines Users passen.

Wirft man einen Blick auf die bereits vorhandene Unterstützung, ist dieser relativ ernüchternd. Seitens PostgreSQL gibt es zwar die Extension [fuzzystrmatch](#) welche unter anderem bereits eine Implementation für die Levenshtein Distanz stellt. Jedoch bietet eine On-Premise Installation von MongoDB keinerlei Funktionalitäten für String Matching von Haus aus. Lediglich in ihrem Atlas Cloud-Service gäbe es eine Fulltext-Search-Funktionalität, welche jedoch auch nicht wirklich unseren Use-Case befriedigen könnte. Alle obigen Algorithmen sind Programmierprobleme, welche sich nicht so trivial in SQL, resp. einer MongoDB Aggregation Pipeline abbilden lassen. Die von uns präferierten Damerau-Levenshtein-Distanz & Longest Common Subsequence sind sogar Probleme, welche nur mithilfe von Dynamic Programming (DP) effizient gelöst werden können. Im Kontext von PostgreSQL könnten jene wahrscheinlich mit etwas Aufwand in PLSQL in einer Stored Procedure implementiert werden. In der Welt von MongoDB gibt es keine Stored Procedures im klassischen Sinne, lediglich *Atlas Functions* welche mit unserem Setup nicht nutzbar sind. Rein mit MongoDB müsste man die gegebenen Aggregationsfunktionen verwenden, um in

einer Pipeline das gewünschte Verhalten zu implementieren. Eine solche funktionale Implementation eines DP-Problems würde sich jedoch recht komplex gestalten, falls überhaupt möglich.

Zusätzlich gäbe es gerade seitens PostgreSQL noch Bedenken hinsichtlich der Performance von einer eigenen Stored Procedure. Die Verwendung solcher sind vom Query Planner nur schwer einsetzbar und ein Einsatz von *Functional Indices* wäre in unserem Fall auch nicht realisierbar, da ein Teil der Argumente dynamisch (User-Input) wäre.

Nach einer detaillierten Analyse des Problemraums für die Korrektur von Tipp- und Schreibfehler und unter Anbetracht aller obigen Problematiken und potentiellen Trade-Offs, haben wir uns dafür entschieden, dieses Teilproblem von unserem Use-Case nicht zu lösen. Wir setzen voraus, dass der User seine Eingaben korrekt vornimmt und diese übereinstimmen mit den Skills der Jobinserate, resp. nur alle übereinstimmenden Resultate letztlich angezeigt werden. Möchte man dieses Problem lösen, so wäre der Einsatz eines Software-Layers oder einer dedizierten Datenbank wie z.B. Elasticsearch notwendig.

7. Abfragen

7.1. SQL Abfragen

7.1.1. Abfrage mit «job_location» und einem «unique_job_skill»

Diese SQL-Abfrage ist entworfen, um LinkedIn-Posts zu finden, die auf spezifische Job-Skills und einen Arbeitsort/Region abgestimmt sind.

```
SELECT lp.*  
FROM linkedin_posts AS lp  
JOIN linkedin_post_skills AS lps ON lp.id = lps.post_id  
JOIN unique_job_skills AS ujs ON lps.skill_id = ujs.skill_id  
WHERE lp.job_location = :city  
AND ujs.skill_name LIKE :skill;
```

Diese Abfrage hat jedoch noch die Limitation, dass nur ein Skill für die Suche verwendet werden kann und dieser Skill auch nicht erkannt wird, wenn z.B. die Gross- Kleinschreibung nicht stimmt. Dies wird in den verbesserten Abfragen gehandhabt.

Um also die Jobanzeigen zu finden, bei der alle Skills die der Benutzer angibt vorhanden sind, müssen wir das Skript etwas verändern. Grob beschrieben sucht das folgende SQL-Skript nach Jobanzeigen innerhalb der Tabelle «linkedin_posts», die folgende Kriterien erfüllen:

1. Die Jobanzeige befindet sich in einem Ort, der den Namen «New York» enthält.
2. Jede Fähigkeit (skill), nach der gesucht wird (in diesem Fall «development»), muss mindestens einmal in der zugeordneten Liste der Fähigkeiten für jede Jobanzeige auftauchen.

Hier werden auch der Ort sowie die Skills mit dem LIKE Operator anstelle dem «=>» verglichen. Dies, da die Daten teilweise inkonsistent und recht lang sind und man so größer nach Stichworten suchen kann. So entsteht für jeden Eintrag ein Score, welcher verwendet wird, um dem Benutzer dann die wichtigsten Job-Inserate zuerst anzuzeigen. Je höher der Score, desto mehr gesuchte Skills sind im Jobinserat gelistet.

Mit diesen Anpassungen sieht dann das Skript wie folgt aus:

```
SELECT lp.id,
       lp.job_link,
       lp.job_title,
       lp.job_location,
       (SELECT COUNT(DISTINCT skill_pattern)
        FROM (SELECT skill_pattern
              FROM linkedin_post_skills lps
              JOIN unique_job_skills ujs ON lps.skill_id = ujs.skill_id,
              UNNEST(ARRAY [%development%, '%management%']) AS skill_pattern
             WHERE lps.post_id = lp.id
               AND LOWER(ujs.skill_name) LIKE LOWER(skill_pattern)) AS matched_skills) AS
skill_count
FROM linkedin_posts lp
WHERE LOWER(lp.job_location) LIKE LOWER('new york%')
ORDER BY skill_count DESC;
```

Technisch gesehen ist das Skript wie folgt aufgebaut (Grobbeschrieb):

- **Auswahl der Job-Postings**

Das Skript wählt alle Job-Postings aus der Tabelle «linkedin_posts», deren Job-Standort «New York» enthält. Die Suche erfolgt hierbei ohne Berücksichtigung der Gross- und Kleinschreibung (`WHERE LOWER(lp.job_location) LIKE 'new york%'`).

- **Ermittlung der Fähigkeiten**

Für jedes Job-Posting wird eine Unterabfrage (`SELECT COUNT(DISTINCT skill_pattern)`) durchgeführt, um die Anzahl der einzigartigen Fähigkeiten zu ermitteln, die im Job-Posting vorhanden sind. Die Fähigkeiten werden aus der Tabelle «linkedin_post_skills» («lps») bezogen, welche mit der Tabelle «unique_job_skills» («ujs») über «skill_id» verknüpft ist.

- **Musterabgleich**

In der Unterabfrage wird die Funktion «UNNEST» verwendet, um eine Liste von Musterfähigkeiten ('%development%', '%management%') zu erstellen. Jede dieser Musterfähigkeiten wird mit den Einträgen in «unique_job_skills» («ujs.skill_name») abgeglichen. Nur Einträge, bei denen der «skill_name» die gesuchten Skills (das Muster) enthält (`LOWER(ujs.skill_name) LIKE LOWER(skill_pattern)`), werden berücksichtigt.

- **Zählen der Übereinstimmungen**

Die Anzahl der eindeutigen Musterfähigkeiten, die im Job-Posting gefunden wurden, wird als «skill_count» zurückgegeben.

- **Sortierung der Ergebnisse**

Die ausgewählten Job-Postings werden nach der Anzahl der gefundenen Fähigkeiten («skill_count») in absteigender Reihenfolge sortiert (`ORDER BY skill_count DESC`).

7.1.2. Abweichung von Bewertungskriterien im Punkt Komplexität

Gewünscht sind in den Projektkriterien folgende Komplexitätsmerkmale für die SQL-Abfrage:

- **Aggregation**

Die Aggregation erfolgt durch die Verwendung der Funktion `COUNT(DISTINCT skill_pattern)`. Dies zählt die eindeutigen «skill_pattern»-Werte, die in der Unterabfrage gefunden werden.

- **Gruppierung**

Dieses Kriterium erfüllt unsere Abfrage **nicht**. Jedoch haben wir weitere Komplexitäten als Kompensation, die wir weiter unten erklären.

- **Join**

Ein Join wird verwendet, um Daten aus zwei Tabellen («linkedin_post_skills» und «unique_job_skills») zu kombinieren. `FROM linkedin_post_skills lps`
`JOIN unique_job_skills ujs ON lps.skill_id = ujs.skill_id`.

- **Selektion**

Selektion erfolgt durch die WHERE-Klausel, die die Bedingung angibt, die erfüllt sein müssen, damit eine Zeile in das Ergebnis aufgenommen wird: z.B. `WHERE LOWER(lp.job_location) LIKE LOWER('new york%')`.

- **Projektion**

Projektion ist das Auswählen spezifischer Spalten aus den Tabellen, was durch die Angabe der Spalten im Select-Teil der Abfrage erfolgt: `SELECT lp.id, ...`.

Da die Gruppierung in unserer Abfrage keinen Sinn ergäbe, haben wir uns entschieden die SQL-Abfrage dafür auf andere Weisen komplexer zu machen:

- **Unterabfrage**

Eine Unterabfrage (Subquery) ist eine SQL-Abfrage, die innerhalb einer anderen Abfrage eingebettet ist. In unserer Abfrage erfolgt die Unterabfrage innerhalb der SELECT-Klausel, um die Anzahl der eindeutigen «skill_pattern»-Werte zu zählen. Diese Unterabfrage erhöht die Komplexität, da sie eine zusätzliche Abfrage innerhalb der Hauptabfrage ausführt und die Ergebnisse zur Berechnung eines Wertes verwendet.

- **Array-Vergleich**

Der Array-Vergleich erfolgt durch die Verwendung der UNNEST-Funktion, die ein Array von Mustern `[%development%, '%management%']` entpackt und diese Muster zur Überprüfung der Übereinstimmung in der LIKE-Klausel verwendet: `UNNEST(ARRAY [%development%, '%management%']) AS skill_pattern`. Diese Technik erhöht die Komplexität der Abfrage, da sie erweiterte SQL-Funktionalitäten nutzt, um mehrere Muster innerhalb einer einzelnen Abfrage zu vergleichen und zu überprüfen.

Dies erhöht die Komplexität weitaus mehr, als es eine Gruppierung der Ergebnisse getan hätte.

7.2. NoSQL Abfragen

Der Use Case 1 beschreibt:

"Personalisierte Jobempfehlungen anhand von Skills und Region"

Basierend auf den Skills und dem gewünschten Arbeitsort/Region werden dem User die am besten passenden Job Inseraten empfohlen. Hierbei werden die Jobs mit der höchsten Übereinstimmung am besten bewertet und somit dem User zuerst angezeigt. "Damit erlaubt man dem Enduser eine einfache, übersichtliche und schnelle Art, die passenden Job Inserate zu finden."

Die nachfolgende Aggregation filtert zunächst alle Job Postings basierend auf der Stadt, in der die Stelle angesiedelt sein soll. Dies reduziert die Anzahl der Dokumente, auf denen die aufwändigeren Schritte der Pipeline ausgeführt werden müssen. Anschliessend wird ein Attribut erstellt, das angibt, wie viele übereinstimmende Fähigkeiten zwischen dem Job Posting und den vom Benutzer angegebenen Fähigkeiten vorhanden sind. Dieses neue Attribut wird genutzt, um die Postings so zu sortieren, dass das am besten passende Stellenangebot zuerst angezeigt wird. Um möglichst viel Überschneidung zwischen den Suchwörtern und den Jobskills zu erlangen, werden beim Überprüfen dieser Übereinstimmungen die Skills alle zu Lowercase gemappt.

Limitationen der Abfrage:

Bei dieser Abfrage erfolgt ein direkter Vergleich der einzelnen Fähigkeiten. Dies bedeutet, dass Abweichungen in Leerzeichen oder Rechtschreibung bei den Fähigkeiten eines Stellenangebots dazu führen können, dass diese nicht als Übereinstimmung erkannt werden. Nur auf die Gross-/Kleinschreibung kommt es nicht an.

Disclaimer Requirements:

Laut den Anforderungen muss die Query die Operatoren "\$lookup", "\$sum" (als Akkumulator), "\$group", "\$match" und "\$project" beinhalten. In der aktuellen Query wird jedoch bewusst auf den Einsatz von "\$lookup" verzichtet, da die Daten aus Performancegründen bereits in ein einziges Dokument transformiert wurden (wobei in dieser Transformation ein \$lookup verwendet wurde).

Anstelle eines herkömmlichen Akkumulators wird der numerische Wert des Vergleichs zwischen dem "Such Array" und dem "Skills Array" verwendet. Der Operator "\$group" ist in diesem Anwendungsfall nicht sinnvoll und wird daher weggelassen. "\$match" wird verwendet, um nach der Stadt zu filtern. Der Operator "\$project" kann ebenfalls weggelassen werden, da die benötigten Werte später in Grafana selektiert werden.

Code	Pipeline Schritt
<pre>\$match: { "job_posting.job_location": { \$regex: ".*XXXX.*", \$options: "i", }, },</pre>	<p>Zuerst werden die Job Postings gefiltert nach der gewünschten Ortschaft</p>
<pre>\$addFields: { matching_skills_count: { \$cond: { if: { \$isArray: "\$job_posting.job_skills", }, then: { \$size: { \$setIntersection: [{ \$map: { input: "\$job_posting.job_skills", as: "skill", in: { \$toLower: "\$skill" } } }, ["nurse practice act", "title 22 compliance", "skill3"]] }, else: 0 }, }, } },</pre>	<p>Hier wird zuerst überprüft ob das Job Posting überhaupt ein Skills Array hat:</p> <p><i>JA:</i> Die Einträge des Skills Array werden zuerst auf Lowercase gemappt und dann verglichen mit einem Array von den User angegebenen Skills. Die Anzahl übereinstimmenden skills werden als Attribut <i>matching_skills_count</i> zum Job Posting hinzugefügt</p> <p><i>Nein:</i> Da das Job Posting keine Skills angegeben hat, wird das Attribut <i>matching_skills_count mit Wert 0</i> hinzugefügt</p>
<pre>\$sort: { matching_skills_count: -1, }</pre>	<p>Die Dokumente werden anhand dem soeben erstellten Attribut sortiert, sodass die Job Postings mit den meisten übereinstimmenden Skills zuoberst stehen</p>
<pre>{ \$limit: XXXX, }</pre>	<p>Nur die obersten <i>N</i> Postings werden angezeigt</p>

8. Effizienz & Performance

8.1. Materialized Views

In unserem Datenbankprojekt haben wir uns gegen die Verwendung von Materialized Views entschieden. Durch die User Inputs sind unsere Abfragen an die Datenbank so dynamisch, dass Materialized Views schlicht unmöglich sind. Materialized Views eignen sich besser für Abfragen, die häufig mit denselben Kriterien durchgeführt werden, da sie so den Abfrageaufwand reduzieren können, indem sie vorab berechnete Ergebnisse speichern.

Bei unserer Applikation kann der User anhand angegebener "Job Location" und "Skills" die gewünschten Job Postings finden. Da der User beliebig viele Kombinationen von Skills und Locations angeben kann, wäre der Problembereich, welcher durch Materialized Views abgedeckt werden muss, unendlich gross. Deshalb verwenden wir normale dynamische Abfragen.

8.2. SQL Performance

Um die Performance unserer SQL-Abfragen zu steigern, müssen wir die aktuelle Performance zuerst anschauen und den Execution-Plan analysieren. Danach müssen wir entweder die Abfragen oder in bestimmten Fällen auch die Datensätze in der Datenbank anpassen und die neuen Abfragen nochmals messen, um diese dann mit den alten zu vergleichen. Nur so wissen wir am Schluss, ob wir die Abfragen optimieren konnten oder nicht.

8.2.1. EXPLAIN und ANALYZE

Der Befehl **EXPLAIN ANALYZE** wird in PostgreSQL verwendet, um den Ausführungsplan einer SQL-Abfrage zu untersuchen. Dieses Tool ist entscheidend für die Performance-Analyse und dann später für die Optimierung von SQL-Datenbankabfragen. **EXPLAIN** alleine zeigt den Plan, den der Query-Planner erstellt, um die angeforderten Daten zu holen. Der Zusatz **ANALYZE** führt die Abfrage tatsächlich aus und liefert Statistiken über die Ausführung, die dabei helfen, den Prozess besser zu verstehen und z.B. auch gleich die tatsächliche Ausführungszeit zusätzlich noch anzeigt in jedem Schritt. So kann man den Prozess besser verstehen und optimieren.

Durch die Verwendung von EXPLAIN ANALYZE erhalten Entwickler und DB-Admins detaillierte Einblicke in die Funktionsweise einer Abfrage, dies umfasst Informationen über:

- Die Kosten und den voraussichtlichen Aufwand zur Ausführung der Abfrage (angegeben in «cost»).
- Die Anzahl der verarbeiteten Zeilen («rows») und die Breite der Zeilen («width»).
- Detaillierte Schritte wie Sequential Scans, Index Scans, Sortierungen und Joins.
- Zeitstatistiken für jede Operation (z.B. «actual time»).

```

Sort  (cost=88843.47..88843.80 rows=135 width=178) (actual time=3983.338..3987.001 rows=17505
loops=1)
  Sort Key: ((SubPlan 1)) DESC
  Sort Method: quicksort  Memory: 3938kB
    -> Gather  (cost=1000.00..88838.69 rows=135 width=178) (actual time=2.125..3954.516
rows=17505 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        -> Parallel Seq Scan on linkedin_posts lp  (cost=0.00..62506.95 rows=56 width=170)
(actual time=0.268..632.333 rows=5835 loops=3)
          Filter: (lower(job_location) ~% 'new york%'::text)
          Rows Removed by Filter: 443650
        SubPlan 1
          -> Aggregate  (cost=187.53..187.54 rows=1 width=8) (actual time=0.224..0.224 rows=1
loops=17505)
              -> Sort  (cost=187.53..187.53 rows=1 width=32) (actual time=0.222..0.222
rows=2 loops=17505)
                  Sort Key: skill_pattern.skill_pattern
                  Sort Method: quicksort  Memory: 25kB
                  -> Nested Loop  (cost=0.99..187.52 rows=1 width=32) (actual
time=0.184..0.219 rows=2 loops=17505)
                      Join Filter: (lower(ujs.skill_name) ~% skill_pattern.skill_pattern)
                      Rows Removed by Join Filter: 53
                      -> Function Scan on unnest skill_pattern  (cost=0.00..0.03 rows=3
width=32) (actual time=0.000..0.000 rows=3 loops=17505)
                          -> Materialize  (cost=0.99..186.43 rows=21 width=27) (actual
time=0.009..0.062 rows=18 loops=52515)
                              -> Nested Loop  (cost=0.99..186.33 rows=21 width=27) (actual
time=0.025..0.179 rows=18 loops=17505)
                                  -> Index Only Scan using linkedin_post_skills_pkey on
linkedin_post_skills lps  (cost=0.56..8.93 rows=21 width=16) (actual time=0.016..0.024 rows=18
loops=17505)
                                      Index Cond: (post_id = lp.id)
                                      Heap Fetches: 0
                                      -> Index Scan using unique_job_skills_pkey on
unique_job_skills ujs  (cost=0.43..8.45 rows=1 width=43) (actual time=0.008..0.008 rows=1
loops=320401)
                                          Index Cond: (skill_id = lps.skill_id)
Planning Time: 0.462 ms
Execution Time: 3988.105 ms

```

Die Ausgabe von **EXPLAIN ANALYZE** zeigt uns hier verschiedene interessante Aspekte dieser Abfrage. Jedoch ist zu beachten, dass in der Abfrage mit genau drei «unique_job_skills» gearbeitet wurde. Mit einer anderen Anzahl Job skills könnte die tatsächliche Ausführungszeit und die anderen Zahlen auch variieren.

Hier eine detaillierte Analyse dieser Ausgabe:

- **Gesamtausführungszeit und Sortierung**

Die endgültige Sortierung der Ergebnisse wurde mittels «Quicksort» Methode durchgeführt, wobei 3938 kB Speicher genutzt wurden. Dies geschah nach der Berechnung der «skill_count» für jedes Job-Posting.

Die Ausführungszeit betrug etwa 3988 ms (ca. 4 Sekunden).

- **Hauptabfrage**

Der «Gather» Knoten zeigt, dass die Abfrage parallel ausgeführt wurde. Zwei parallele Worker wurden geplant und gestartet. Die Gesamtkosten («cost») für diesen Schritt werden von 1000 bis 88838 geschätzt, mit 17505 Zeilen, die tatsächlich verarbeitet wurden.

Der «Parallel Seq Scan» auf der Tabelle «linkedin_posts» filtert die Job-Postings, deren Standort «New York» enthält. Dies ist effizient, da die parallele Verarbeitung ermöglicht, dass grosse Datenmengen schneller verarbeitet werden.

- **Filterung**

Der Filter `lower(job_location) ~~ '%new york%' ::text` entfernte 443'650 Zeilen, sodass 17'505 Zeilen übrig blieben.

- **SubPlan (Unterabfrage)**

Der SubPlan 1 berechnet die Anzahl der Fähigkeiten, die mit den Mustern «'%development%', '%management%', %engineering%» übereinstimmen. Dieser Plan wird für jedes der 17'505 Job-Postings ausgeführt.

Die Aggregation (zählen der Übereinstimmungen) erfolgte für jede Zeile in etwa 0.224 ms. Das Quicksort Verfahren wurde angewendet, um die Musterfähigkeiten zu sortieren, was 25 kB Speicher benötigte.

- **Joins und Scans**

Verschachtelte Schleifen-Joins (Nested Loops) wurden verwendet, um die Fähigkeiten aus «linkedin_post_skills» und «unique_job_skills_pkey» zu verwenden, um die Join-Bedingungen effizient zu erfüllen. Dies reduzierte die benötigte Zeit für die Joins.

Der Materialize Knoten speicherte Zwischenergebnisse, um die Joins zu beschleunigen.

- **Planung und Ausführungszeit**

Die Planungszeit für die Abfrage betrug nur 0.462 ms, was sehr schnell ist. Die gesamte Ausführungszeit war 3988.105 ms, was auf eine relativ hohe Komplexität der Abfrage hinweist, insbesondere aufgrund der vielen Zeilen und der parallelen Verarbeitung.

Zusammengefasst kann man folgendes daraus schliessen:

- Die Abfrage nutzt **Parallelisierung** effektiv, was die Verarbeitung grosser Datenmengen beschleunigt.
- **Indizes** werden gut genutzt, um die Joins effizient durchzuführen.
- Die **Sortierung** und **Aggregation** der Fähigkeiten könnten optimiert werden, möglicherweise durch Verwendung von Hash-Aggregationen oder effizienten Join-Strategien.

8.2.2. Visualisieren des Ausführungsplans

Zusätzlich zu den bereits existierenden Tools **EXPLAIN** und **ANALYZE** haben wir ein Visualisierungs-Plugin im DataGrip verwendet: **Postgres Explain Visualizer** (<https://plugins.jetbrains.com/plugin/18804-postgres-explain-visualizer>, Plugin ID: com.tjhelmut.postgres-explain-visualizer) um den Execution-Plan zu visualisieren und wir so direkt verschiedene Daten auf einen Blick analysieren können.

Die Visualisierung des Ausführungsplans sieht wie folgt aus:

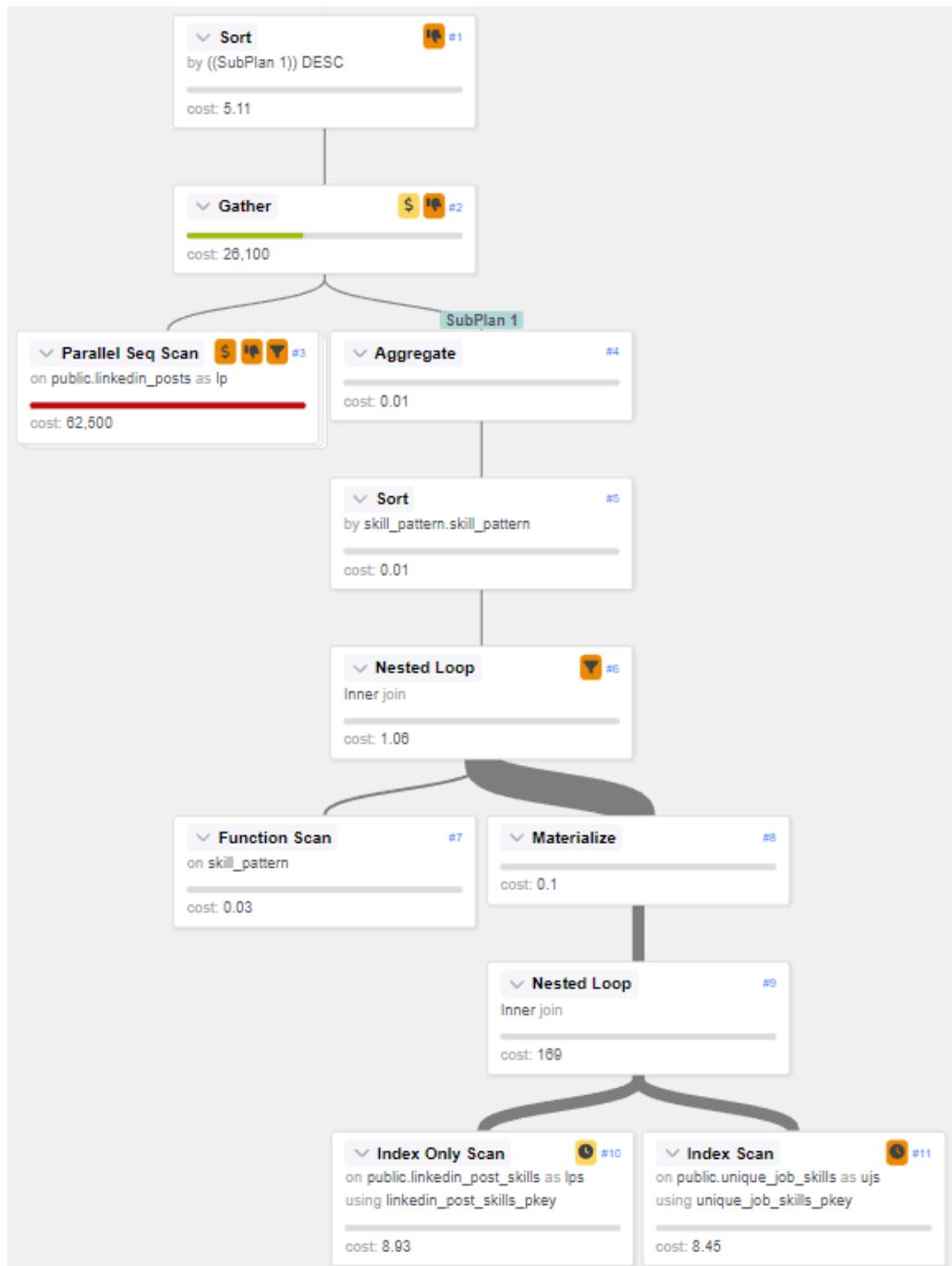


Bild 18 - Visualisierung des Ausführungsplans vor Optimierung

Struktur und Aufbau

- Knoten: Jeder Knoten im Graph repräsentiert eine Operation oder einen Schritt in der Abfrage, wie beispielsweise Scans, Joins oder Sortierungen. Knoten auf höherer Ebene repräsentieren komplexe Operationen, die aus mehreren untergeordneten Schritten bestehen können.
- Kanten: Die Kanten zeigen die Richtung des Datenflusses. Daten fliessen von den untergeordneten zu den übergeordneten Knoten, wobei die Endknoten die finale Datenmenge für das Abfrageergebnis liefern.
- Verzweigungen: Der Plan kann sich an mehreren Stellen verzweigen, um Parallelität oder die Kombination von Daten aus verschiedenen Quellen zu ermöglichen. Eine Verzweigung tritt auf, wenn ein Schritt im Plan Daten von zwei oder mehr unabhängigen Operationen benötigt.

8.2.3. Optimierungen der SQL Datenbank

Indizierung der relevanten Spalten

Eine einfache Option, die Datenbankabfrage zu optimieren, ist es, die relevanten Spalten zu indexieren.

1. «linkedin_posts.job_location»

Da wir die Ergebnisse basierend auf der Spalte «job_location» filtern, wird eine Indizierung dieser Spalte die Suchvorgänge beschleunigen. Obwohl der LIKE Operator verwendet wird, reicht in den meisten Fällen ein B-Baum-Index aus. Wenn die Suchmuster jedoch komplexer sind, könnte ein Textmusterindex (wie GIN oder GiST mit der «pg_trgm» Erweiterung) effektiver sein.

2. «linkedin_post_skills.post_id»

Diese Spalte wird verwendet, um die Tabelle «linkedin_posts» mit «linkedin_post_skills» zu verknüpfen. Ein Index auf «post_id» beschleunigt diesen Join-Vorgang. Dieser Index existierte bereits, da diese Spalte ein Primärschlüssel ist.

3. «unique_job_skills.skill_id»

Diese Spalte wird verwendet, um «linkedin_post_skills» mit «unique_job_skills» zu verknüpfen. Ein Index auf «skill_id» verbessert die Effizienz dieses Joins. Auch dieser Index existierte bereits, da diese Spalte ein Primärschlüssel ist.

4. «unique_job_skills.skill_name»

Diese Spalte wird in der LIKE Klausel verwendet, um Fähigkeitenmuster abzugleichen. Ähnlich wie bei der «job_location» Spalte könnte ein B-Baum-Index hilfreich sein, aber ein GIN oder GiST Index mit «pg_trgm» könnte für Musterabgleiche effizienter sein.

Die SQL-Befehle für die Indizierung

Die SQL Befehle für die Primärschlüssel müssen nicht ausgeführt werden, sofern diese schon existieren.

```
-- Index auf job_location für linkedin_posts
CREATE INDEX idx_linkedin_posts_job_location ON linkedin_posts(job_location);

-- Index auf post_id für linkedin_post_skills (existiert bereits, PK)
CREATE INDEX idx_linkedin_post_skills_post_id ON linkedin_post_skills(post_id);

-- Index auf skill_id für unique_job_skills (existiert bereits, PK)
CREATE INDEX idx_unique_job_skills_skill_id ON unique_job_skills(skill_id);

-- Index auf skill_name für unique_job_skills (für Musterabgleiche)
CREATE INDEX idx_unique_job_skills_skill_name ON unique_job_skills(skill_name);
```

Hinweis: Wenn noch nicht installiert, dann muss die «pg_trgm» Erweiterung noch installiert werden mit dem Befehl: `CREATE EXTENSION pg_trgm;`

Performance-Analyse nach der Indexierung der Spalten

In unserem Fall konnten die Indizes alleine keinen wirklichen Performance-Increase generieren. Wir sehen hier einige Testdurchläufe mit Zeitmessungen ohne vs. mit Indizes:

#	Ohne Indizes	Mit Indizes
1	4390.137 ms (Planning Time: 0.642 ms)	3398.321 ms (Planning Time: 0.472 ms)
2	4121.296 ms (Planning Time: 0.504 ms)	3641.058 ms (Planning Time: 0.543 ms)
3	4203.280 ms (Planning Time: 0.532 ms)	3633.554 ms (Planning Time: 0.488 ms)
4	4307.673 ms (Planning Time: 1.284 ms)	3410.756 ms (Planning Time: 0.522 ms)
5	3814.058 ms (Planning Time: 0.474 ms)	3514.513 ms (Planning Time: 0.480 ms)
6	3773.938 ms (Planning Time: 0.495 ms)	3508.460 ms (Planning Time: 0.512 ms)
7	3941.406 ms (Planning Time: 0.546 ms)	3414.813 ms (Planning Time: 0.468 ms)
8	3739.779 ms (Planning Time: 0.506 ms)	3377.617 ms (Planning Time: 0.467 ms)
9	3978.346 ms (Planning Time: 0.518 ms)	3580.887 ms (Planning Time: 0.570 ms)
10	4100.212 ms (Planning Time: 0.526 ms)	3555.782 ms (Planning Time: 0.466 ms)
AVG	~3997.7764 ms (PT: ~0.6027 ms)	~3503.5761 ms (PT: ~0.4988 ms)

Wir haben im Durchschnitt eine Verbesserung von fast ~500 ms, also einer halben Sekunde bei denselben Argumenten. Somit sehen wir hier, dass die Indizes eine positive Auswirkung auf die Performance der Abfrage haben.

Sargable Queries

Sargable Queries sind SQL-Abfragen, die so gestaltet sind, dass sie die Indexe in einer Datenbank effektiv nutzen können. Das Wort "sargable" leitet sich von "Search ARGument ABLE" ab und bedeutet, dass eine Abfrage so formuliert ist, dass die Datenbank-Engine die Indexe nutzen kann, um die Abfrage schneller auszuführen.

Mit zum Beispiel einer Spalte wie "job_location", die indexiert wird, kann man folgend eine Query schreiben die diesen Index effizient nutzen kann:

```
SELECT * FROM linkedin_posts WHERE job_location = 'New York';
```

Hier kann die Datenbank den Index verwenden, um schnell Jobanzeigen in New York zu finden. Eine nicht-sargable Query wäre:

```
SELECT * FROM linkedin_posts WHERE LOWER(job_location) = 'New York';
```

Da hier die Funktion LOWER auf die Spalte "job_location" angewendet wird, kann der Index nicht genutzt werden, und die Datenbank muss jeden Eintrag prüfen.

Auch sind Vergleiche mit dem LIKE-Operator nicht-sargable, wenn sie dem Muster "%etwas hier%" folgen. Aber Abfragen, die dem Muster "'etwas hier'" folgen, sind sargable.

Die Query hat im Bezug auf den Location-Vergleich Potential für eine Optimierung hinsichtlich "Sargability". Konkret kann man die Where-Klausel für den Check der *job_location* wie folgt abändern:

```
... WHERE lp.job_location ILIKE '%new york%' ORDER BY ...
```

Der *ILIKE* Operator ist case-insensitive, womit die Anforderung für ein beidseitiges *LOWER()* entfällt. Jedoch besteht weiterhin der beidseitige %-Operator, welcher die (Sub-)Query letztlich doch nicht sargable macht.

LOWER auf Tabellen direkt anwenden

Beim Vergleich der Skills wird ebenfalls ein case-insensitive Check gemacht:

```
... AND LOWER(ujs.skill_name) LIKE LOWER(skill_pattern)) AS matched_skills ...
```

Hier bestand die Hypothese, dass die "on-the-fly" *LOWER()* Operation für das *skill_pattern* die Query verlangsamt. Deshalb wurden testhalber die Tabelle *unique_job_skills* geklont und dort die folgende Transformation auf alle Records ausgeführt:

```
-- clone skills-table & -records
CREATE TABLE unique_job_skills_lower (LIKE unique_job_skills INCLUDING ALL);
INSERT INTO unique_job_skills_lower
SELECT * FROM unique_job_skills;

-- drop unique key/index before transformation to avoid conflicts
ALTER TABLE unique_job_skills_lower DROP CONSTRAINT
unique_job_skills_lower_skill_name_key;

-- transform to lower
UPDATE unique_job_skills_lower
SET skill_name = LOWER(skill_name);
```

Wie bereits im Kapitel *Daten laden und transformieren* festgestellt, gibt es viele Skills, welche sich nur durch ihre Schreibweise unterscheiden. Mit der Transformation zu Lowercase gibt es demnach einige Duplikate - rund 855'000, um genau zu sein. Jene wurden mit folgender Query ermittelt:

```
-- count duplicate skills
SELECT SUM(duplicate_count) FROM (
    SELECT COUNT(skill_name) AS duplicate_count
    FROM unique_job_skills_lower
    GROUP BY skill_name
    HAVING count(skill_name) > 1
);
```

Um eine akkurate und vergleichbare Messung zu t  igen, muss jedoch der Unique Constraint/Index wiederhergestellt werden. Dies ist jedoch im aktuellen Zustand aufgrund der vielen Duplicates nicht m  glich. Um die Duplikate erfolgreich zu bereinigen, muss je ein bleibender Record bestimmt und alle anderen gel  scht werden. Die einzelnen Skills werden jedoch   ber die Link Tabelle *linkedin_post_skills* mit den entsprechenden Inseraten verkn  pt. Um dies losgel  st testen zu k  nnen muss diese Tabelle ebenfalls dupliziert werden (da es sich hier um rund 23 Millionen Records handelt wurde ein andere, performantere Variante gewählt):

```
-- clone link-table
CREATE TABLE linkedin_post_skills_lower AS
SELECT * FROM linkedin_post_skills;
```

Um die Duplikate erfolgreich zu bereinigen und auch in der Link-Tabelle die Mappings entsprechend sauber anzupassen wurde folgende "Multi-Stage"-Query verwendet:

1. Alle doppelten Skills werden geladen und in der Variable *skill_duplicates* zwischengespeichert. Zusätzlich wird f  r jede Duplikatsgruppe jeweils die UUID des ersten Records mitgespeichert (*keeper_id*). Der Datensatz mit jener ID ist derjenige, welcher letztlich nicht gel  scht wird
2. F  r alle Duplikate welche gel  scht werden sollen (*id != keeper_id*) werden in der neuen Link Tabelle *linkedin_post_skills_lower* die Mappings dupliziert, welche auf den zu behaltenden Skill-Record zeigen
3. Alle duplizierten Skills, ausser der jeweils erste, werden gel  scht
4. Alle Inserat-Skill-Mappings, welche auf nicht mehr existierende Skills (gel  scht in Schritt 3) zeigen, werden ebenfalls noch gel  scht

```

-- 1) find all duplicate records & identify the one to keep (first one)
WITH skill_duplicates AS (
    SELECT
        skill_id,
        skill_name,
        FIRST_VALUE(skill_id) -- use since MIN() doesn't work for UUID's
            OVER(PARTITION BY skill_name) as keeper_id
    FROM unique_job_skills_lower
    WHERE skill_name IN (
        SELECT skill_name
        FROM unique_job_skills_lower
        GROUP BY skill_name
        HAVING COUNT(skill_name) > 1
    )), remapped_skills AS (
-- 2) remap every to-be-deleted skill to the kept one
    INSERT INTO linkedin_post_skills_lower
    SELECT lps.post_id as post_id,
        skill_duplicates.keeper_id as skill_id
    FROM linkedin_post_skills_lower lps
        JOIN skill_duplicates on lps.skill_id = skill_duplicates.skill_id
    WHERE lps.skill_id = skill_duplicates.skill_id
        AND lps.skill_id != skill_duplicates.keeper_id
-- 3) delete all duplicate skills, except kept one
), deleted_duplicates AS (
    DELETE FROM unique_job_skills_lower
    WHERE skill_id IN (SELECT skill_duplicates.skill_id
        FROM skill_duplicates
        WHERE skill_duplicates.skill_id != skill_duplicates.keeper_id)
)
-- 4) delete all obsolete mappings targeting deleted skills
DELETE FROM linkedin_post_skills_lower
WHERE skill_id IN (SELECT skill_duplicates.skill_id
    FROM skill_duplicates
    WHERE skill_duplicates.skill_id != skill_duplicates.keeper_id);

```

Zu guter Letzt kann der Unique Constraint / Index mit folgendem Script ohne Probleme wiederhergestellt werden. Somit ist jetzt nach einem etwas aufwändigen Setup alles bereitgestellt für das Testing.

```

-- re-establish unique key/index on skill_name
ALTER TABLE unique_job_skills_lower
ADD CONSTRAINT unique_job_skills_lower_skill_name_key
    UNIQUE (skill_name);

```

Während dem Testen beider Approaches (on-the-fly LOWER oder LOWER Records) fiel auf, dass die Variante wo alle Skills bereits in Lowercase in der DB gespeichert sind gleich schnell resp. teilweise sogar marginal langsamer (potentiell aufgrund Hardware Fluktuationen) war als die on-the-fly Transformation. Somit konnte diese Hypothese hinterlegt werden und die beiden Tabellen, welche zu Testzwecken angelegt wurden, wurden wieder gelöscht. Der Vorteil hierbei ist, dass die Locations mit korrekter Gross- und Kleinschreibung in der Benutzeroberfläche angezeigt werden können.

Alles in allem ist dies ein spannender Unterschied im Vergleich zu MongoDB, wo die Lowercase-Transformation auf der DB einen spürbaren Vorteil gebracht hat (siehe folgendes Unterkapitel).

8.3. NOSQL Performance

8.3.1. Basismessung

Um Performance Verbesserungen zu erstellen, muss zuerst die aktuelle Query analysiert werden. In MongoDB kann man dies durch das Anhängen von `".explain("executionStats");"` bei einem Shell Command erreichen, oder in MongoDB Compass im "explain" Tab nachschauen.

Die Query wurde mit folgenden Parametern gestartet:

- Location Filter auf `"a"`, sodass möglichst viele Dokumente zurückgegeben werden, jedoch immer noch nach etwas gefiltert wird
- Die gewünschten Skills auf `["nurse practice act", "title 22 compliance", "skill3"]` verglichen

Somit braucht die Aggregation insgesamt 22'394 Millisekunden und gibt 834'419 Dokumente zurück:

```
j,
"executionStats": {
  "executionSuccess": true,
  "nReturned": 834419,
  "executionTimeMillis": 31674,
```

Bild 19 - Execution Details Aggregation Basic

Mit `explain` wird zusätzlich auch noch ein Estimate zurückgegeben, wie viel Zeit (summiert) nach jeder Stage vergangen ist und wie viele Dokumente zurückgegeben wurden:

Total ms: 31'674 ms			
Stage	nReturned	time estimate	(increase)
Filter	834'419	433	433
AddFields	834'419	17'693	17'260
Sort	834'419	30'213	12'520

Für die **Filter**-Stage braucht die Aggregation geschätzt 532 millisekunden und gibt 834'419 Dokumente zurück (von den 1.3 Mio):

`"nReturned": 834419,`
`"executionTimeMillisEstimate": 532`

Für die **addFields**-Stage, also dort wo die Arrays verglichen werden, braucht die Aggregation geschätzt 8829 - 532 = 8297 millisekunden und gibt 834'419 Dokumente zurück.

`"nReturned": 834419,`
`"executionTimeMillisEstimate": 8829`

Für die letzte **sort**-Stage, also dort, wo die Dokumente anhand der Übereinstimmungen sortiert werden, braucht die Aggregation geschätzt 2428 - 2419 = 9 Millisekunden und gibt 7099 Dokumente zurück.

`"nReturned": 7099,`
`"executionTimeMillisEstimate": 2428`

8.3.2. Verbesserungspotenzial

In der Aggregation Pipeline wird schlussendlich über alle Skills von den Jobpostings in einer Stadt iteriert. Bei jedem Skill wird dann zusätzlich noch ein Mapping gemacht, um sicherzustellen, dass alles kleingeschrieben ist.

Dieses Umformen bei jeder Abfrage ist kostenintensiv und könnte einfach gespart werden, wenn man die Skills direkt kleingeschrieben abspeichert. Es wird bewusst dagegen entschieden, den klein geschriebenen Skills Array als weiteres Attribut zu den jeweiligen Dokumenten hinzuzufügen, um die originale Schreibweise beizubehalten. Dies würde zu extrem redundanten Daten führen und wenn man die Skills später visualisieren möchte, spielt es keine so grosse Rolle, ob die Skills mit einem Gross- oder Kleinbuchstaben anfangen.

Das Transformieren kann mit folgender Aggregation-Pipeline erreicht werden:

```
[  
  {  
    "$addFields": {  
      "job_posting.job_skills": {  
        "$map": {  
          "input": "$job_posting.job_skills",  
          "as": "skill",  
          "in": { "$toLower": "$$skill" }  
        }  
      }  
    },  
    {  
      "$out": "job_postings_lowercase"  
    }  
]
```

Die Aggregation Pipeline erstellt ein Mapping, welches jeden Eintrag nimmt, "to lowercase" transformiert und wieder in das Array hinzufügt.

Nach Beenden der Pipeline hat sich die Struktur des Dokumentes nicht verändert, nur die Skills sind jetzt alle klein geschrieben.

Nun kann man bei der Abfrage das in Kleinschreibung umwandeln weglassen, was Rechenarbeit und somit Zeit spart. Die Neue Abfrage lautet Folgendermassen:

```
[  
  {  
    $match: {  
      "job_posting.job_location": {  
        $regex: ".*XXX.*",  
        $options: "i",  
      },  
    },  
  },
```

Die Pipeline filtert immer noch zuerst nach der gewünschten Stadt. Somit müssen wir nur noch die Jobpostings iterieren, die überhaupt in Frage kommen.

<pre>{ \$addFields: { matching_skills_count: { \$cond: { if: { \$isArray: "\$job_posting.job_skills", }, then: { \$size: { \$setIntersection: ["\$job_posting.job_skills", ["XXX", "XXX", "XXX",],], }, }, else: 0, }, }, }, }</pre>	<p>Das Prinzip für die “AddFields” Stage ist auch immer noch dieselbe. Wir fügen ein Attribut “matching_skills_count” hinzu, welches angibt, wie viele Skills übereinstimmen.</p> <p>Das Berechnen dieses Wertes ist nun jedoch viel einfacher, da wir jetzt “setIntersection” auf die beiden Arrays anwenden können.</p>
<pre>{ \$sort: { matching_skills_count: -1, }, }</pre>	<p>Das “Sort” bleibt unverändert.</p>

Führt man nun diese Pipeline aus und misst die Zeit, gibt es folgenden Angaben:

Total ms: 22'748 ms			
Stage	nReturned	time estimate	(increase)
Filter	834'419	368	368
AddFields	834'419	9299	8'931
Sort	834'419	21'455	12'156

Die unten stehende Tabelle zeigt schön auf, dass die von MongoDB geschätzte Zeit für die “**AddFields**” Stage um etwa 8 Sekunden verschnellert wurde. Die anderen beiden Stages haben auch eine Verbesserung, jedoch nur wenige hundert ms, was darauf zurückzuführen ist, dass das ja nur ungefähre Schätzungen von MongoDB sind und nicht exakte Messungen.

Stage	Original estimate ms increase	Optimized estimate ms increase	Optimisation ms
Filter	433	368	~100
AddFields	17'260	8'931	~8'300
Sort	12'520	12'156	~400

9. Datenschutz & Datenbanksicherheit

9.1. Analyse Schwachpunkte

In einem System mit Datenbanken und mehreren Benutzern gibt es einige Schwachstellen, die wir bei uns beachten müssen:

- **S1: Nicht volle Kontrolle über das System**

Da unsere DB-Server und Grafana auf dem Server eines Mitstudenten laufen, haben wir nicht die volle Kontrolle über das System. Das bedeutet, dass wir uns auf die Sicherheit und Verwaltung durch den Serveradministrator verlassen müssen (in einigen Punkten). Dies bedeutet aber auch, dass es für uns umso wichtiger ist, Backups der Datenbank zu machen und sicherzustellen, dass wir die Datenbank jederzeit auf ein neues System replizieren können.

- **S2: Geteilte Benutzer mit vollen Rechten**

In unserem Projekt gibt es Benutzer, die volle Zugriffsrechte haben und diese Nutzer werden von den Teammitgliedern geteilt. Wir haben also nicht für jedes Projektmitglied einzelne Benutzer auf Grafana oder den DB-Servern erstellt. Das kann problematisch sein, weil es schwer nachvollziehbar ist, wer welche Änderungen vorgenommen hat. Außerdem erhöht es das Risiko, dass jemand aus Versehen wichtige Daten löscht oder verändert. Auch ist ein Passwort schneller mal geleakt oder wird weitergegeben, wenn mehrere Personen es kennen.

Auf diese Schwachstellen müssen wir besonders achten. Natürlich gibt es auch die üblichen Schwachstellen, die wir nicht vergessen sollten: Sicherheit der Verbindung, Datenintegrität und Backups, Sichere Passwörter, usw.

Hier ist eine Risikoanalyse unserer zwei Haupt-Schwachstellen vor und nach Mitigation dieser.

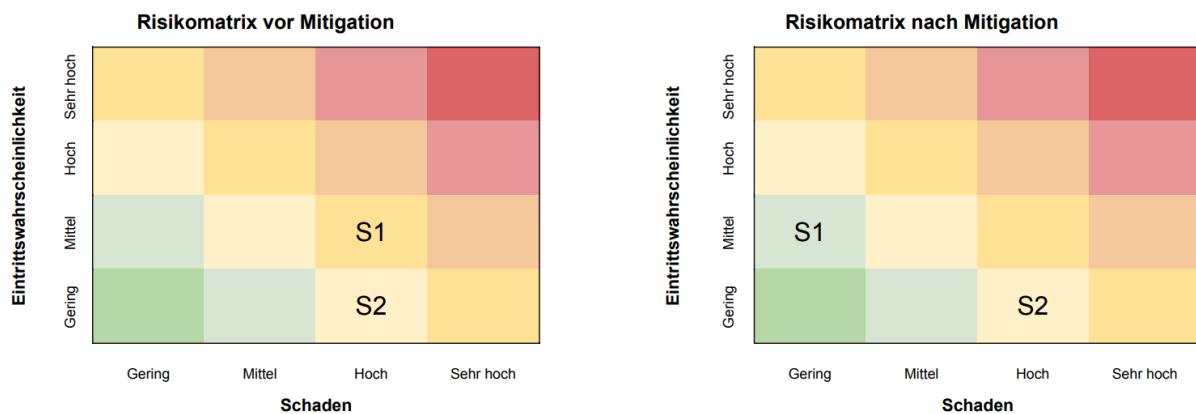


Bild 20 - Risikoanalyse Schwachstellen SQL-Server

Folgendes hat zu diesen Risikomatrizen geführt:

- **Nicht volle Kontrolle über das System (S1), vor Mitigation**

Die Eintrittswahrscheinlichkeit, dass in diesen wenigen Wochen etwas gravierendes passiert, ist nicht sehr hoch. Sie ist aber sehr wohl präsent. Nicht bezahlte Server-Rechnung des Administrators, Ausfall beim Hosting, usw. Dies könnte alles zu einem Verlust unserer Daten und des Projektfortschritts führen. Der Schaden wäre hoch, da in dieses Projekt mehrere Wochen und viele Arbeitsstunden investiert wurden.

- **Nicht volle Kontrolle über das System (S1), nach Mitigation**

Durch Backups und das Arbeiten mit Skripts für die Benutzer-Erstellung, etc. können wir die Datenbank resp. Grafana recht schnell wieder auf einem neuen System zu replizieren. Da die Daten alle sowieso schon öffentlich zugänglich sind, gibt es auch keine Bedenken bez. Datenschutz. Die Eintrittswahrscheinlichkeit bleibt also gleich.

- **Geteilte Benutzer mit vollen Rechten (S2), keine Mitigation**

Da die Eintrittswahrscheinlichkeit recht gering ist, dass in diesen wenigen Wochen jemand der Projektmitglieder ausversehen das Passwort an eine Person weitergibt oder sich jemand unbefugt Zugang verschafft, haben wir uns entschieden, dies nicht zu behandeln. Der Schaden könnte jedoch hoch sein, wenn ein drittes Team absichtlich unsere Datenbank resp. Grafana so sabotiert, dass dies einen Einfluss auf die Bewertung hätte.

9.2. Sicherheit SQL

Die SQL Datenbank wird auf einem PostgreSQL-Server gehostet und stellt gesharte Datenbanken für verschiedene Teams zur Verfügung. Der Server ist so konfiguriert, dass jedes Team über eigene Zugangsdaten und Berechtigungen verfügt. Der Hauptadministrator hat umfassende Rechte, während andere Benutzergruppen spezifische Lese- und Schreibrechte gemäss ihrem Projekt erhalten. Der Zugriff auf den Server erfolgt über sichere Verbindungen (SSL/TLS), um die Sicherheit der Datenübertragung zu gewährleisten.

9.2.1. Credentials & Rechte

Folgende Benutzer existieren auf dem PostgreSQL Server:

Username	Passwort	Rolle / Rechte
postgres	Wird nicht angegeben	Super-Admin / Alle Rechte auf DB Server Gruppe 2
full_access_user	Wird nicht angegeben	User / Read Write Execute
readonly	Wird nicht angegeben	User / Read <i>Kommentar: User wurde von Administrator erstellt für Verbindung zu Grafana.</i>
read_only_user	[REDACTED]	User / Read

Wir haben die Passwörter von Benutzern mit mehr als Read-Rechten hier aus Sicherheitsgründen bewusst nicht angegeben.

Erstellung PostgreSQL Benutzer

Man kann User entweder grafisch oder per SQL erstellen. Um dies einfach replizieren zu können, entschieden wir uns für die Variante per SQL. Dieses Skript wie folgt aus:

```
-- Create full access user
CREATE USER full_access_user WITH PASSWORD '██████████';

-- Grant privileges for full access user
GRANT CONNECT ON DATABASE dbs TO full_access_user;
GRANT USAGE ON SCHEMA public TO full_access_user;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO full_access_user;
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO full_access_user;
GRANT ALL PRIVILEGES ON ALL FUNCTIONS IN SCHEMA public TO full_access_user;
GRANT CREATE ON SCHEMA public TO full_access_user;

ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL PRIVILEGES ON TABLES TO
full_access_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL PRIVILEGES ON SEQUENCES TO
full_access_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL PRIVILEGES ON FUNCTIONS TO
full_access_user;

-- Create read-only user
CREATE USER read_only_user WITH PASSWORD 'ReadOnlyUsr.2024$g02';

-- Grant read-only privileges for read-only user
GRANT CONNECT ON DATABASE dbs TO read_only_user;
GRANT USAGE ON SCHEMA public TO read_only_user;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO read_only_user;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO read_only_user;

-- Grant read-only privileges to in future added tables
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO read_only_user;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON SEQUENCES TO read_only_user;
```

Hier eine kurze Zusammenfassung dieses SQL-Skripts:

- Ein Benutzer **full_access_user** wird erstellt, der volle Rechte auf Tabellen, Sequenzen und Funktionen im Schema public hat und Objekte im Schema erstellen darf. Diese Rechte gelten auch für zukünftige Objekte.
- Ein Benutzer **read_only_user** wird erstellt, der nur Lesezugriff auf Tabellen und Sequenzen im Schema public hat. Diese Leserechte gelten ebenfalls für zukünftige Objekte.

9.2.2. Datenverlust, Backup und Wiederherstellung

Bei der Entwicklung und Verwaltung von Datenbanksystemen ist es wichtig, die Daten immer wieder zu sichern. In diesem Kapitel geht es um die Sicherung der PostgreSQL Datenbank und um den Transfer dieser Datenbank auf einen zentralen Server, wo sie von allen Teammitgliedern genutzt werden kann.

Nachdem die Daten lokal korrekt transformiert wurden, wurde eine Datensicherung in Form eines Backups vorgenommen. Dazu wurde pg_dump verwendet. Dies ist eine PostgreSQL Utility, die mit der Installation von PostgreSQL mitgeliefert wird. Die Sicherung wurde im Custom-Format erstellt, das mehrere Vorteile bietet, darunter die Komprimierung und eine schnellere Wiederherstellung im Vergleich zu Textdumps.

Für das Erstellen des Backups wurde folgender Befehl verwendet:

```
pg_dump -U dbsstudent -Fc dbs > backup.dump
```

Nach der Erstellung der Sicherung ist es wichtig, die Integrität der Sicherungsdatei zu überprüfen. Dies konnte gleich durch den Transfer gemacht werden, indem man die Datenbank auf einem neuen System wiederherstellt.

Zur Sicherheit wurde das Backup auch auf ein Google Drive gelegt, auf das alle Teammitglieder Zugriff haben. Auch wenn wir also die Daten irgendwie beschädigen oder etwas nicht umkehrbares testen wollen, können wir immer wieder auf dieses Backup zugreifen und die komplette Datenstruktur und die Daten selbst wiederherstellen.

Die Wiederherstellung auf dem zentralen Server ist mittels **pg_restore** erfolgt. Auch dies ist wieder ein PostgreSQL Utility, welches bei der Installation von PostgreSQL mitgeliefert wird. Dies kann entweder über DataGrip gelöst werden, indem man das Restore der Datenbank über das GUI macht (verwendet im Hintergrund ebenfalls **pg_restore**). Dabei muss man jedoch beachten, dass man die Parameter etwas anpasst.

Das Backup selbst beinhaltet die Berechtigungen des Backup Source Systems. Auf diesem neuen Datenbankserver (auf dem wir das Backup wiederherstellen wollen) existieren nicht alle Rollen und User, die es auf dem Source System gab. Also muss man beim Restore den Parameter **-O (no-owner)** verwenden. Dieser überspringt die Zuweisung der Eigentümerschaft für alle wiederhergestellten Objekte. Dies ist eben dann nützlich, wenn auf der Zielumgebung die Benutzer und Rollen des Dumps nicht existieren.

Wenn man dies auf dem Server von seinem Host aus ausführen möchte, kann man weitere Argumente verwenden um dies zu tun:

```
pg_restore --dbname=dbs --single-transaction -O --schema=public --username=postgres  
--host=domainhere --port=porthere C:\path\to\backup.dump
```

9.3. Sicherheit NoSQL

Für die Mongo Database werden verschiedene Mechanismen zum Schutz der Daten verwendet

9.3.1. DB-Zugriff

Wenn das Projekt wie geplant auf Atlas aufgesetzt wird, kann man in den Einstellungen definieren, von welcher Adresse überhaupt auf den DB Server zugegriffen werden darf. Somit kann man zum Beispiel seinen eigenen Server whitelisten oder auch zum Beispiel nur Firmeninternen Access zusätzlich erlauben. Da ein eigener Server verwendet wird, ist diese Option nicht anwendbar für das Projekt. Der Server selbst könnte auch so konfiguriert werden, was für das Projekt aber weggelassen wird, da es out of scope ist.

9.3.2. Credentials & Rechte

Natürlich ist es auch wichtig, verschiedene User zu verwenden, mit sicheren Passwörtern und nur den benötigten Rechten (Principle of Least Privilege). Fast noch wichtiger ist es, keine standard Passwörter zu verwenden für den Root Account.

Die untenstehenden Nutzer wurden mit folgendem Script erstellt. Wörter in Capslock sind Placeholder für die jeweiligen Werte der Tabelle (ausser DBS - Name der DB):

```
use DBS
db.createUser({
  user: "USERNAME",
  pwd: "PASSWORD",
  roles: [{ role: "ROLE", db: "DBS" }]
})
```

Bild 21 - Script User Erstellung

Username	Passwort	Rechte
Grafana	Wird nicht angegeben	read
Developer	Wird nicht angegeben	dbAdmin
root	Wird nicht angegeben	root
readonly	[REDACTED]	readonly

Der User **Developer** wird von allen Entwicklern verwendet und besitzt die dbAdmin Rechte, um mit der DB zu arbeiten und mögliche Wartungsarbeiten zu machen. "readWrite" wäre zu wenig, da man mit dieser Berechtigung keine Indizes verwalten kann.

Der User **Grafana** wurde, wie der Name schon sagt, für Grafana erstellt und muss nur die Daten lesen können.

Natürlich gibt es auch den **root** Benutzer, welcher alle Rechte hat, wie zum Beispiel andere Accounts verwalten.

Der User **Backupper** wird für das Backup erstellt. Da er auf der admin DB backup Rechte braucht, wird er so erstellt:

Bild gelöscht, da sensible Information

Bild 22 - Script User Erstellung

9.3.3. Datenverlust

Natürlich besteht immer die Gefahr, dass ein Entwickler die Daten aus Versehen unbrauchbar rendert oder der Server an sich kaputt geht oder im schlimmsten Fall kompromittiert wird.

In Atlas wäre das Backup erstellen automatisch eingebaut, jedoch nicht für die kostenlose Sandbox Version aus dem Unterricht. Unter Additional Settings, könnte man das Backup aktivieren.

The screenshot shows the 'Additional Settings' section of the MongoDB Atlas cluster configuration. It includes two main toggle switches: one for 'Turn on Backup (M2 and up)' which is off, and another for 'Termination Protection' which is also off. Below each switch is a brief description and a 'Learn more' link.

Bild 23 - Atlas Backup aktivierung

Da die in diesem Dokument beschriebene Lösung auf einem privaten Server läuft, ist diese Option sowieso nicht einsetzbar.

Jedoch kann man mittels MongoDB Database Tools, mit den Befehl *mongodump*, auch ein Backup erzeugen ohne auf Atlas angewiesen zu sein:

```
> mongodump --uri="mongodb://Backupper:.....@domainhere:portnr" --out=./Backup_06.05.2024
```

Diesen Command könnte man nun auch mit einem simplen shell script automatisieren und täglich ein Backup erstellen lassen.

9.4. Grafana

Zusätzlich zu den direkten Datenbankzugriffen für NoSQL und PostgreSQL, nutzen wir die Grafana UI-Schnittstelle zur Visualisierung und Analyse der Daten. Grafana befindet sich auf demselben Server wie die beiden DB-Server Instanzen und ermöglicht eine benutzerfreundliche Darstellung der Daten. Die Verbindung zwischen Grafana und der Datenbanken ist ebenfalls SSL/TLS-geschützt, um die Sicherheit der Daten zu gewährleisten.

In Grafana haben wir verschiedene Benutzerrollen definiert: Benutzer mit Änderungsrechten können Dashboards erstellen und bearbeiten, während andere Benutzer nur Leserechte haben und die Dashboards lediglich ansehen können. Diese Trennung der Rechte hilft, die Integrität der Dashboards zu wahren und unbefugte Änderungen zu verhindern.

Username	Passwort	Rechte
[REDACTED]	Wird nicht angegeben	Admin / Alle Rechte auf Gruppe 2
readonly	[REDACTED]	Member / Nur Leserechte

10. Visualisierung & Entscheidungsunterstützung

Im Unterricht wurde Metabase für die Visualisierung empfohlen. Nach einiger Zeit wurde jedoch klar, dass Metabase nicht über die benötigten Funktionalitäten verfügt oder diese nicht ausreichend ausgeprägt sind, insbesondere beim Handling der Variablen. Metabase fügt Variablen mit Anführungszeichen in die

Query/Aggregation ein, was sich nicht abschalten lässt und die Abfrage an den Stellen, wo die Variablen in unseren Queries verwendet werden, ungültig macht.

Aus diesem Grund wurde Grafana als Alternative gewählt. Da sich niemand im Team weder mit Metabase noch mit Grafana auskennt, wäre die Einarbeitungszeit für beide Technologien gleich. Es war jedoch von anderen Teams bekannt, dass Grafana für unsere Anwendungsfälle besser geeignet ist.

10.1. Dashboard Grafana

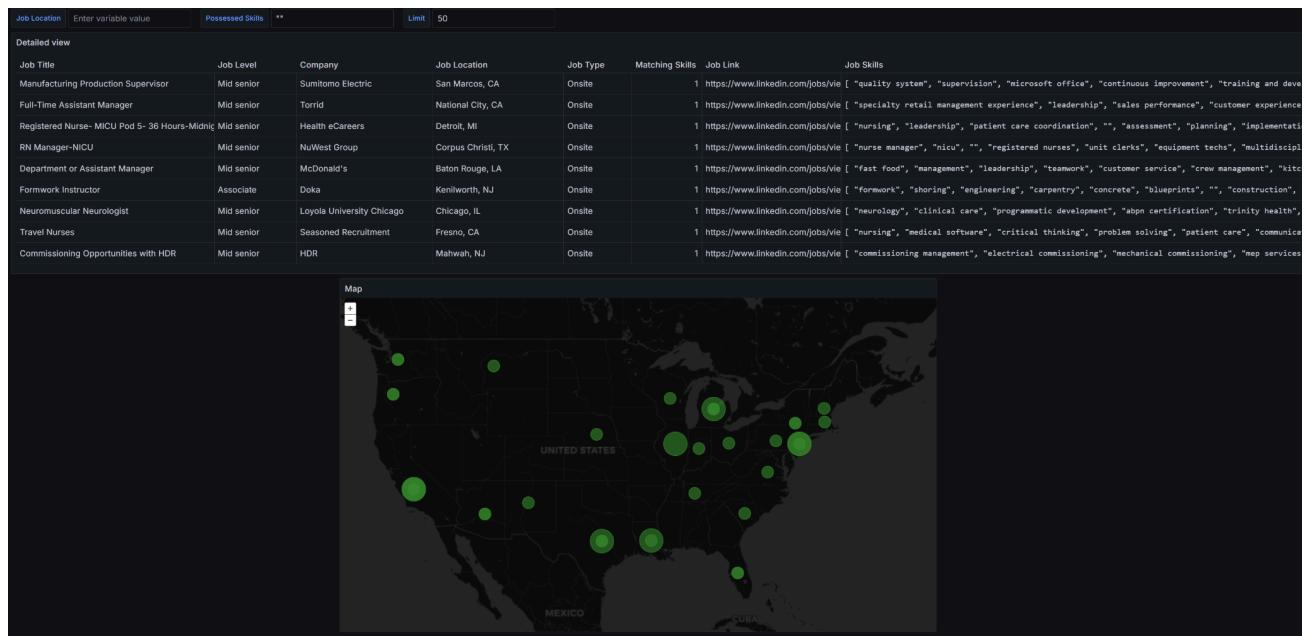


Bild 24 - Grobe Übersicht des Grafana Dashboards

Die beiden Dashboards für SQL und NOSQL sind beide in 3 Teile aufgebaut:

- Panel 1 - Detailed View:
 - Hier werden die wichtigsten Infos der gefundenen Job postings in Tabellenform angezeigt
- Panel 2 - Map
 - Hier werden die gefundenen Job Postings zu ihrem US-Bundesstaat zugeordnet um eine Übersicht zu bekommen, wo welche jobs sind
- Variablen
 - Ganz oben am Dashboard gibt es 3 Variablen die man ausfüllen kann
 - Job Location - In welcher Stadt sollen die jobs sein
 - Possessed Skills - Eine Auflistung von Fähigkeiten, welche ich besitze und in den Jobs verlangt werden soll. Die obersten Resultate haben am meisten übereinstimmende Elemente
 - Limit - Wieviele Job postings sollen angezeigt werden

10.2. Konfigurieren Panels

Das Konfigurieren der Panels ist im Grundsatz gleich für SQL und NoSQL.

Als erstes wählt man aus, welchen Typ das Panel haben soll - Table, Map, Chart, Pie und vieles mehr.



Bild 25 - Optionen Panel Types

Als nächstes wählt man die Data source aus - also den DB Server - und fügt die Query hinzu, die alle benötigten Daten aus dieser Datasource abfragt. Dieser Schritt ist nun von den jeweiligen Technologien abhängig und wird in späteren Kapiteln jeweils genauer beschrieben.

10.2.1. Panel 1 - Detailed View

10.2.2.

City	MD	Possessed Skills	"nurse practice act","headto	Result #	50	Table view
Detailed view						
Job Title	Job Level	Company	Job Location	Job Type	Matching Skills	Job Link
RN Med Surg/Tele at Maxim Healthcare Staffing	Mid senior	Health eCareers	Easton, MD	Onsite	1	https://www.linkedin.com/jobs/view/rn-med-surg-tele-at-maxim-healthcare-staffing/
School Nurse RN - Baltimore MD at Maxim Health	Mid senior	Health eCareers	Baltimore, MD	Onsite	1	https://www.linkedin.com/jobs/view/school-nurse-rn-baltimore-md-at-maxim-health/
Clinical Education Specialist RN	Mid senior	Futurecare Associates, Inc.	Baltimore, MD	Onsite	1	https://www.linkedin.com/jobs/view/c clinical-education-specialist-rn-futurecare-associates-inc-baltimore-md/

Bild 26 - Panel 1 - Detailed View

Als erste Visualisierung soll eine einfache Tabelle angezeigt werden, mit den Infos all der Job-Postings, die am besten passen. Für das wurde der Visualisationstyp "Table" gewählt.

Bei diesem Panel-Typ werden einfach alle Spalten angezeigt, welche die Query zurückgibt.

Um das Resultat noch ein wenig zu verschönern, kann man sogenannte Overrides erstellen. Bei den Overrides kann man Sachen wie andere Displaynamen oder Spaltenbreite definieren.

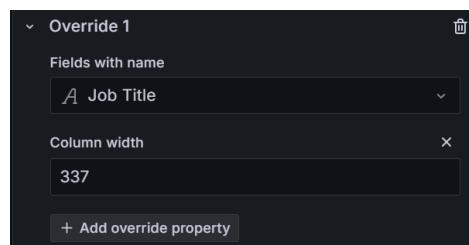


Bild 27 - Override job_location

Damit die entsprechenden Links anklickbar sind und den User direkt zum Inserat führen können, wurde für das Feld *Job Link* folgenden Override erstellt. Für sowohl Titel als URL des Links wurde die Grafana Variable \${__value.raw} verwendet, welches den Inhalt des Felds widerspiegelt.

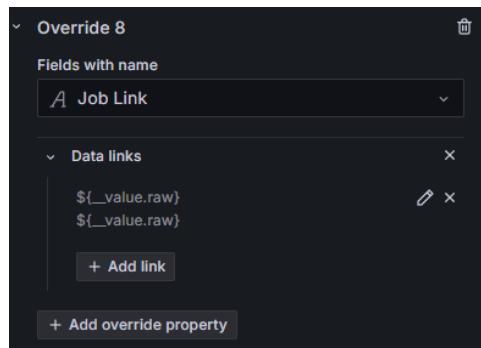


Bild 28 - Override für Links

10.2.3. Panel 2 - Map

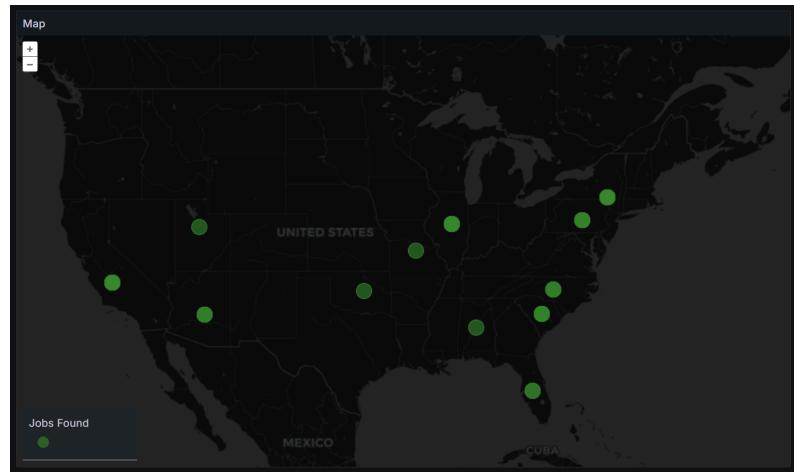


Bild 29 - NoSQL Visualisation - GeoMap

In dieser Visualisation soll eine Übersicht gegeben werden, aus welchen Staaten die Jobinserate kommen. Um dies zu erreichen, wird der Visualisierungstyp "GeoMap" gewählt. Anders als beim "Panel 1 - Detailed View" muss man für das GeoMap-Panel noch ein paar weitere Einstellungen vornehmen, damit die Daten richtig angezeigt werden.

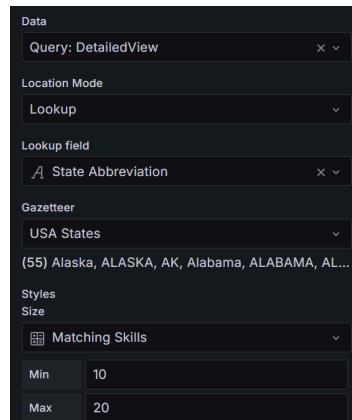


Bild 30 - Visualisierungssetings Settings - GeoMap

“Data” bestimmt, aus welcher Query die Daten bezogen werden - von der sprachspezifischen Query.

Für “Location Mode” wurde Lookup gewählt, was bedeutet, dass man Namen/Kürzel zu geografischen Orten zu ordnet. In diesem Beispiel wird das Attribut “State Abbreviation” genommen und den US Staaten zugeordnet. Ist ein Job-Posting nicht aus den Staaten, kann es leider nicht auf der Karte angezeigt werden.

Unten bei Styles wird definiert, dass die Grösse der Kreise anhand des “Matching Skills” Attribut auf der Karte berechnet werden. Somit: Je mehr Skills übereinstimmen, desto grösser ist der Kreis des Job Postings auf der Karte.

10.3. Variablen

Damit der User seine Skills und die gewünschte Stadt eingeben kann und Grafana das in die Abfrage einarbeiten kann, braucht es sogenannte Variablen.

Im untenstehenden Bild kann man repräsentativ die Einstellungen der Variable *inp_location* sehen.

- type
 - Es wurde “Text box” gewählt, so dass der Benutzer freitext eingeben kann.
- Name
 - Prefix *inp* um in den Queries klar zu signalisieren, dass dies ein dynamischer Input ist.
- Label Name
 - Dieser Wert wird dem User angezeigt. Im untenstehenden Screenshot wurde er auf *Job Location* gesetzt, damit es für den User verständlicher ist.
- Default Value
 - Wurde leer gelassen, da standardmäßig jobs aus allen Städten angezeigt werden sollen.

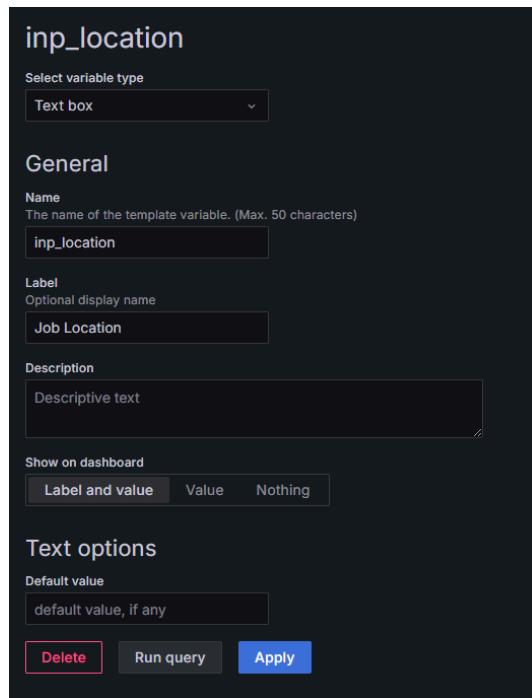


Bild 31 - Job Location Variable in Grafana

Folgende Tabelle repräsentiert eine Übersicht aller erstellter Variablen für dieses Dashboard:

Name	Type	Label	Default Value
inp_skills	Text box	Possessed Skills	"" oder ''
inp_limit	Text box	Limit	50
inp_location	Text box	Job Location	

Disclaimer:

Bei Skills ist der Default value für NoSQL "" und für SQL '', da dies später in der Query der Inhalt eines Arrays sein wird und somit mindestens ein Element (wenn auch leer) enthalten muss.

10.4. SQL-Panel Spezifisches

10.4.1. Transformation für GeoMap

Damit auf der Grafana Geo-Map (Visualisierung 2) die einzelnen Locations in den USA korrekt angezeigt werden können, wird hierfür das Kürzel vom entsprechenden US-Bundesstaat benötigt. Aktuell ist jenes nur im Feld *job_location* zusammen mit dem Namen der Stadt vorhanden: <Stadt>, <Kürzel>. Dieses Kürzel soll nun extrahiert und in der neuen Spalte *state_abbreviation* gespeichert werden.

Zuerst haben wir versucht, auf klassischem Wege eine neue *nullable* Spalte mit "ALTER TABLE xx ADD COLUMN.." hinzuzufügen. Jedoch nach über 1.5h Laufzeit haben wir diesen Ansatz gestoppt. Anstatt dessen haben wir die Tabelle *linkedin_posts* strukturell geklont und die neue Spalte auf der leeren Tabelle hinzugefügt:

```
CREATE TABLE linkedin_posts_new (LIKE linkedin_posts INCLUDING ALL);

ALTER TABLE linkedin_posts_new
    ADD COLUMN state_abbreviation text NULL;
```

Bevor wir die Daten kopieren konnten, haben wir den Index auf der Spalte *job_location* auf der neuen Tabelle mit folgendem Code temporär gelöscht, damit während dem Kopieren der Index nicht mit jedem INSERT neu von der DB berechnet werden muss.

```
DROP INDEX idx_linkedin_posts_new_job_location;
```

Nun können wir die Daten mit dem Code-Snippet unterhalb direkt von der "alten" Tabelle in die neue kopieren und zugleich die US-Bundesstaaten aus der Spalte *job_location* extrahieren (mittels *string_to_array(..)*). Die Query wurde innerhalb von rund 14 Sekunden erfolgreich ausgeführt und hat 1'348'454 Records in die neue Tabelle *linkedin_posts_new* kopiert.

```
INSERT INTO linkedin_posts_new(id, job_link, last_processed_time,
                                job_title, company, job_location, first_seen,
                                search_city, search_country, search_position,
                                job_level, job_type, summary_id,
                                state_abbreviation)
SELECT id, job_link, last_processed_time, job_title, company, job_location,
       first_seen, search_city, search_country, search_position, job_level,
       job_type, summary_id,
       (string_to_array(job_location, ', '))[2] -- extract state
FROM linkedin_posts;
```

Anschliessend konnten die Tabellen umbenannt werden, so dass die erstellte Tabelle nun *linkedin_posts* heisst und die alte implizit ersetzt:

```
ALTER TABLE linkedin_posts RENAME TO linkedin_posts_old;
ALTER TABLE linkedin_posts_new RENAME TO linkedin_posts;
```

Da jedoch die alte Tabelle über Foreign Keys verfügt, welche auf andere Tabellen resp. von anderen Tabellen auf sich selbst zeigt, müssen jene ebenfalls neu erstellt bzw. angepasst werden. Letztlich soll die alte Tabelle *linkedin_posts_old* nirgends mehr durch Foreign Keys referenziert werden. Dies geschieht mit folgendem Script:

```
-- add fk for linkedin_posts / drop fk from linkedin_posts_old
ALTER TABLE linkedin_posts
    ADD CONSTRAINT linkedin_posts_job_summaries_summary_id_fk
        FOREIGN KEY (summary_id) REFERENCES job_summaries (summary_id);
ALTER TABLE linkedin_posts_old DROP CONSTRAINT fk_summary_id;

-- drop old fk, add new one on linkedin_post_skills
ALTER TABLE linkedin_post_skills DROP CONSTRAINT linkedin_post_skills_post_id_fkey;
ALTER TABLE linkedin_post_skills
    ADD FOREIGN KEY (post_id) REFERENCES linkedin_posts
        ON DELETE CASCADE;
```

Initial wurde der Index auf dem Feld *job_location* temporär gelöscht, dieser muss nun wiederhergestellt werden:

```
CREATE INDEX idx_linkedin_posts_job_location ON linkedin_posts(job_location);
```

Zu guter letzt kann die alte Tabelle gelöscht werden, da sie nun vollständig von der Neuen abgelöst wurde:

```
DROP TABLE linkedin_posts_old;
```

10.4.2. SQL Query

```
SELECT lp.job_title          AS "Job Title",
       lp.job_level           AS "Job Level",
       lp.company              AS "Company",
       lp.job_location         AS "Job Location",
       lp.state_abbreviation AS "State Abbreviation",
       lp.job_type              AS "Job Type",
       (SELECT COUNT(DISTINCT skill_pattern)
        FROM (SELECT skill_pattern
               FROM linkedin_post_skills lps
               JOIN unique_job_skills ujs ON lps.skill_id = ujs.skill_id,
                  UNNEST(ARRAY [${inp_skills:csv}]) AS skill_pattern
              WHERE lps.post_id = lp.id
                AND ujs.skill_name ILIKE skill_pattern) AS matched_skills)
      AS "Matching Skills",
       lp.job_link             AS "Job Link"
  FROM linkedin_posts lp
 WHERE lp.job_location ILIKE '${inp_location}%'
 ORDER BY "Matching Skills" DESC
 LIMIT ${inp_limit};
```

Dies ist nun die SQL Query, welche von Grafana an den SQL Server geschickt wird, um die Daten für die Panels abzufragen. Um dies zu ermöglichen wurden noch folgende anpassungen gemacht:

- **SELECT:** Wir haben hier die Projektion angepasst. Und zwar selektieren wir nur gebrauchte Felder und geben diesen auch einen neuen Alias, damit für den Benutzer klarer wird, um was es sich dabei handelt.
- **Nutzung der Dashboard-Variablen:** Wir nutzen in diesem SQL-Skript auch die Variablen, die wir im Grafana Dashboard definiert haben. Mit diesen kann der Benutzer die Suchargumente einfach anpassen. Jene Variablen haben die Notation \${var_name} und werden zur Laufzeit von Grafana mit den entsprechenden Eingabewerten vom User befüllt.
- **Limit:** Zusätzlich in diesem Skript wollen wir nicht alle Ergebnisse anzeigen, wie es bis dahin akzeptabel war, sondern wollen dem Benutzer die Möglichkeit geben selbst zu entscheiden, wie viele Vorschläge er erhalten möchte. Oft sind nur die ersten Vorschläge wirklich interessant für den Benutzer.

Das Feld *State Abbreviation* ist per se nur für die Darstellung der GeoMap (Visualisierung 2) relevant. Deshalb wurde auf dem Tabellen-Panel jene Spalte explizit in Grafana mittels eines Overrides ausgeblendet:

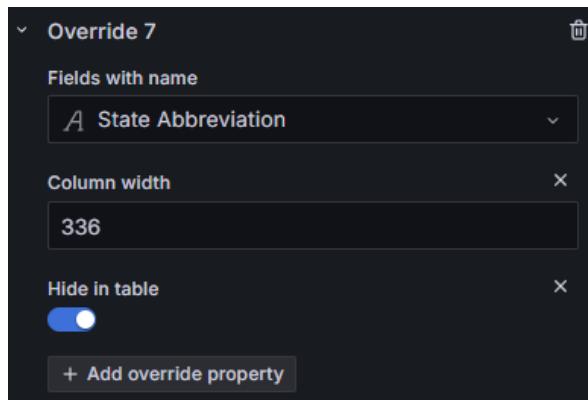


Bild 32 - Ausblendung State Abbreviation in Grafana Tabelle

10.5. NOSQL-Panel Spezifisches

In diesem Kapitel wird angesprochen, was spezifisch für NoSQL unternommen / konfiguriert wurde für die Panels.

10.5.1. Re-Transformation Dokumente NoSQL

Um die Visualisierung in Grafana zu vereinfachen, wurden noch 2 Transformationen unternommen: Einerseits wurden die Attribute von dem *root->job_posting* Attribut direkt in das Root Objekt verschoben. Obwohl im Unterricht erwähnt wurde, dass die erste Variante die anzustrebende Variante ist, ist es nach der Transformation massiv einfacher zu handhaben. Die Transformation wurde mit der linken Query erreicht.

Bei der zweiten Transformation wird der US Staat aus der "job_location" geparsed und als Attribut "state_abbreviation" hinzugefügt.

```

1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16
17 17
18 18
19 19
20 20
21 21
22 22
23 23
24 24
25 25
26 26
27 27
28 28
29 29

```

Bild 33 - Transformation zurück ins Root Element

```

1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
9 9
10 10
11 11
12 12
13 13
14 14
15 15
16 16

```

Bild 34 - Parsen des US Staates

10.5.2. NoSQL Query

Die Aggregation, welche in den Kapiteln zuvor verwendet wurde, muss nun natürlich angepasst werden. Einerseits müssen die Variablen des Dashboards eingebettet werden und andererseits sollten die Attributnamen lesbarer umbenannt werden.

Dies ist nun die finale Aggregation, welche an den NoSQL-DB-Server geschickt wird, um die Daten für die Panels abfragen:

```
1 [  
2 {  
3   "$match": {  
4     "job_location": {  
5       "$regex": ".${{inp_location}}.*",  
6       "$options": "i"  
7     }  
8   },  
9 },  
10 {  
11   "$addFields": {  
12     "matching_skills_count": {  
13       "$cond": {  
14         "if": { "$isArray": "$job_skills" },  
15         "then": { "$size": { "$setIntersection": ["$job_skills", [${{inp_skills}}] ] } },  
16         "else": 0  
17       }  
18     }  
19   },  
20 },  
21 {  
22   "$sort": { "matching_skills_count": -1 }  
23 },  
24 {  
25   "$limit": ${{inp_limit}}  
26 },  
27 {  
28   "$project": {  
29     "Job Link": "$job_link",  
30     "Job Title": "$job_title",  
31     "Company": "$company",  
32     "Job Location": "$job_location",  
33     "Job Level": "$job_level",  
34     "Job Type": "$job_type",  
35     "Job Skills": "$job_skills",  
36     "State Abbreviation": "$state_abbreviation",  
37     "Matching Skills": "$matching_skills_count"  
38   }  
39 },  
40 ]
```

Bild 35 - Aggregation für Grafana

Gelb markiert sieht man, wie die Variablen des Dashboards in die Aggregation eingebettet wurden. Für die Lesbarkeit wurde eine weitere Stage "\$project" hinzugefügt, so dass die Titel der Attribute im UI gut lesbar sind. Außerdem werden so auch nur die Attribute angezeigt, die man wirklich braucht.

Database.Collection	DBS	. . .	job_postings
QueryType	①	Table	
Infer Schema	②	<input checked="" type="checkbox"/>	
Value Fields	③		
Job Link	:	string	-
Job Title	:	string	-
Job Level	:	string	-
Company	:	string	-
Job Location	:	string	-
Matching Skills	:	int32	-
Job Skills	:	*json.RawMessage	-
State Abbreviation	:	*string	-

Bild 36 - NoSQL Visualisations settings 1 - GeoMap

Anders als bei den SQL-Panels muss man für NoSQL auch noch die Datentypen der einzelnen Attribute definieren, da es kein striktes Schema gibt, das den Datentyp für Grafana definiert. Alle Typen, welche einen Asterisk als Präfix haben, sind nullable.

Die meisten Attribute sind simple Strings, bis auf 3 Ausnahmen:

- Matching skills
 - Berechneter Integer, wie viele Skills übereinstimmen mit dem User input
- State Abbreviation
 - Berechneter Substring von Job Location. Der Asterix macht den Wert nullable, da bei gewissen Job Inseraten der US Staat nicht geparsed werden kann.
- Job Skills
 - Ist ein JSON Array in den MongoDB Dokumenten und wird demnach als json.RawMessage deklariert. Er ist auch nullable, da gewisse job postings keine skills angegeben haben

10.6. Entscheidungsempfehlung Personas

Unsere Persona Daniel möchte nun seinen Plan vom Auswandern in die USA in die Tat umsetzen. Seine Freundin lebt etwas ausserhalb von Lexington Park, einer grösseren Stadt im US-Bundesstaat Maryland. Bevor er jedoch seinen aktuellen Job in der Schweiz an den Nagel hängt, möchte er eine nahtlose Anschlussstelle in den USA haben.

Als berufstätiger junger Erwachsener und mit den zusätzlichen administrativen Tätigkeiten, welche die Auswanderung mit sich bringt, ist seine Zeit aktuell ziemlich limitiert. Auf der Suche nach einer Plattform, die auf ihn zugeschnittene Jobinserate ohne stundenlanges Scrollen liefert, ist er auf *JobScout* gestossen und probiert es gleich aus.

Er möchte einen relativ kurzen Arbeitsweg und gibt als *City* den Suchbegriff "Lexington Park" ein. Für seine Fähigkeiten gibt er folgende Auswahl aus seiner aktuellen Jobbeschreibung in das Feld *Possessed Skills* ein:

- "c++"
- "software integration"
- "python 3"
- "threading"
- "process communication"
- "timeseries analysis"

Mit diesen Kriterien gibt das Dashboard *g02-mongodb-dashboard* (MongoDB) folgende Daten für seine [Suchanfrage](#) zurück:

City	Lexington Park	Possessed Skills	*c++, *software integration*	Result #	50	
Detailed view						
Job Title	Job Level	Company	Job Location	Job Type	Matching Skills	
Senior Software Developer with Security Clearance Mid senior	ClearanceJobs	Lexington Park, MD	Onsite	6	https://www.linkedin.com/jobs/view/senior-software-developer-with-security-clearance-at-clearancejobs-3805761229	
Senior Software Developer	Mid senior	MAG Aerospace	Lexington Park, MD	Onsite	5	https://www.linkedin.com/jobs/view/cyber-interface-vulnerability-researcher-securit-mid-senior/
Cyber Interface Vulnerability Researcher (Securit Mid senior)	ClearanceJobs	Lexington Park, MD	Onsite	1	https://www.linkedin.com/jobs/view/cyber-interface-vulnerability-researcher-securit-mid-senior/	
Sr. Systems Engineer (ENKG2076)	Mid senior	Air Combat Effectiveness Consult	Lexington Park, MD	Onsite	1	https://www.linkedin.com/jobs/view/tawhaak-weapon-system-tws/
Cyber Vulnerability Researcher (Security Enginee Mid senior)	The MIL Corporation	Lexington Park, MD	Onsite	1	https://www.linkedin.com/jobs/view/cyber-vulnerability-researcher-binary-reverse-engineering/	
Cyber Vulnerability Researcher (Security Enginee Mid senior)	ClearanceJobs	Lexington Park, MD	Onsite	1	https://www.linkedin.com/jobs/view/cyber-vulnerability-researcher-binary-reverse-engineering/	
Software Engineer	Associate	Sabre Systems Inc.	Lexington Park, MD	Onsite	1	https://www.linkedin.com/jobs/view/software-engineer-computer-science-system-engineering-statistical-analysis-data-management/
Cyber Interface Vulnerability Researcher (Securit Mid senior)	The MIL Corporation	Lexington Park, MD	Onsite	1	https://www.linkedin.com/jobs/view/cyber-interface-vulnerability-researcher-rf-interfaces-gps-acars/	
Computer Forensic and Intrusion Detection Analy	Mid senior	The MIL Corporation	Lexington Park, MD	Onsite	1	https://www.linkedin.com/jobs/view/top-secret-clearance-ba-bs-degree-us-citizenship-network-operations-intrusion-detection-analy/

Bild 37 - Resultate für Persona

Das beste Resultat ist folgende Stelle als *Senior Software Developer with Security Clearance*. Daniel freut sich darüber, dass diese Stelle in der gewünschten Stadt liegt und alle seine angegebenen Skills beinhaltet. Er klickt auf den Link vom Inserat und beginnt umgehend mit seiner Bewerbung auf die Stelle via LinkedIn.

Jobs Found	
Job Link	https://www.linkedin.com/jobs/view/senior-software-developer-with-security-clearance-at-clearancejobs-3805761229
Job Title	Senior Software Developer with Security Clearance
Job Level	Mid senior
Company	ClearanceJobs
Job Location	Lexington Park, MD
Matching Skills	6
Job Skills	c++, python 3, html, parallel processing, threading, process communication, web development, timeseries analysis, scripting, database, programming, software development, software integration, customer training, demonstrations, special access program (sap), dod secret security clearance, covid19 vaccination, equal opportunity/affirmative action employer, diversity and inclusion, americans with disabilities act (ada), section 503 of the rehabilitation act of 1973, vietnamera veterans' readjustment assistance act of 1974
State Abbreviation	MD

Bild 38 - Passentster Job für Persona

Auf dem anderen Dashboard, welches die PostgreSQL Datenbank als Datenquelle verwendet (*g02-postgres-dashboard*), werden dieselben Resultate für eine [identische Suchanfrage](#) zurückgegeben und es wird das gleiche Inserat zuoberst gelistet resp. Daniel empfohlen.

Mittels weniger Klicks wurde es Daniel ermöglicht, konkrete und auf ihn zugeschnittene Job-Inserate zu finden. Die relevantesten Resultate wurden ihm automatisch zuoberst angezeigt, so dass er seine Zeit nutzen kann, sich für diese Stellen zu bewerben, anstatt jene Zeit mit Scrollen und Ansehen von unpassenden Inseraten zu vergeuden. Alles in allem konnte unsere Datenbank-Applikation ideal auf die Bedürfnisse von Daniel eingehen und mit ihm einen zufriedenen User gewinnen.

11. Fazit & Lessons Learned

In dem Projekt konnte das Team in den verschiedensten Bereichen etwas unerwartetes Lernen. Bei den Daten selbst wurde uns während dem Projekt klar, dass bei eigentlich simplen Aktionen schnell Probleme auftreten können, wenn man mit 1.3 Millionen Datensätzen arbeitet. So war das Transformieren oder Normieren von Daten schwieriger als gedacht - vermeintlich "triviale" Operationen führten schon zu (Performance-)Problemen, welche gelöst werden mussten. Bei der Wahl der Daten selbst - dem Kaggle - hatten wir grosse Probleme, etwas zu finden, was genügend stimmige und nutzbare Daten enthielt.

Selbst bei den ausgewählten Job-Posts auf Kaggle, die von LinkedIn stammen, mussten wir feststellen, dass die Daten nicht konsistent und ordentlich formatiert waren. Statt klar strukturierter Begriffe wie "git" oder "development" fanden wir eine Vielzahl von Einträgen, die oft unbrauchbare oder inkonsistente Werte enthielten. Diese Daten waren wild durchmischt und wiesen zahlreiche Fachbegriffe auf, die teils in Gross-, teils in Kleinschreibung und oft ohne klar erkennbare Struktur zusammengefügt waren. Diese Unordnung entspricht jedoch vermutlich eher der Realität, wenn man mit echten Daten arbeitet, und zeigt die Herausforderungen, die bei der Datenbereinigung und -standardisierung auftreten können.

Zusätzlich wurden wir in diesem Aspekt wahrscheinlich auch etwas von unserem Informatik-“Weltbild” in die Irre geführt - nicht in jedem Beruf lässt sich eine Kompetenz in wenigen Begriffen definieren.

Die Arbeit mit Grafana war für uns alle neu, sodass wir dort viel lernen konnten. Rückblickend wäre ein Anwendungsfall, der auf Statistiken oder numerischen Werten basiert, besser geeignet gewesen, da wir mehr Möglichkeiten gehabt hätten, die Daten in Grafana zu visualisieren.

Die Zusammenarbeit im Team funktionierte gut. Wir konnten die Aufgaben fair verteilen, sodass jeder einen Beitrag zu jedem Kapitel leisten konnte. Dies hatte beim Dokumentieren den Vorteil, dass jeder in der Lage war, jedes Kapitel zu schreiben und somit problemlos das nächste Kapitel übernehmen konnte.

12. Quellen und Abbildungsverzeichnis

12.1. Quellenverzeichnis

- [Bild] (2023) *Mann in schwarzem Crewneck-T-Shirt mit schwarzer Brille*, Unsplash. Verfügbar unter: <https://unsplash.com/photos/man-in-black-crew-neck-t-shirt-wearing-black-framed-eyeglasses-bku-eac45BQ> (Zugriff am: 13. Juni 2024).
- PostgreSQL (o.J.) *Fuzzy String Match*, PostgreSQL-Dokumentation. Verfügbar unter: <https://www.postgresql.org/docs/current/fuzzystrmatch.html> (Zugriff am: 13. Juni 2024).
- MongoDB (o.J.) *Atlas App Services*, MongoDB. Verfügbar unter: <https://www.mongodb.com/products/platform/atlas-app-services> (Zugriff am: 13. Juni 2024).
- Wikipedia (2023) *Edit distance*, Wikipedia. Verfügbar unter: https://en.wikipedia.org/wiki/Edit_distance (Zugriff am: 13. Juni 2024).
- Wikipedia (2023) *Longest common subsequence*, Wikipedia. Verfügbar unter: https://en.wikipedia.org/wiki/Longest_common_subsequence (Zugriff am: 13. Juni 2024).
- Wikipedia (2023) *Hamming distance*, Wikipedia. Verfügbar unter: https://en.wikipedia.org/wiki/Hamming_distance (Zugriff am: 13. Juni 2024).
- Wikipedia (2023) *Damerau-Levenshtein distance*, Wikipedia. Verfügbar unter: https://en.wikipedia.org/wiki/Damerau%20%93Levenshtein_distance (Zugriff am: 13. Juni 2024).
- Wikipedia (2023) *Levenshtein distance*, Wikipedia. Verfügbar unter: https://en.wikipedia.org/wiki/Levenshtein_distance (Zugriff am: 13. Juni 2024).
- Wikipedia (2023) *Materialized view*, Wikipedia. Verfügbar unter: https://en.wikipedia.org/wiki/Materialized_view (Zugriff am: 13. Juni 2024).
- Wikipedia (2023) *Sargable*, Wikipedia. Verfügbar unter: <https://en.wikipedia.org/wiki/Sargable> (Zugriff am: 13. Juni 2024).

12.2. Abbildungsverzeichnis

Bild 1 - Use Case Diagramm.....	6
Bild 2 - Persona Daniel (Bild von Dzmitry Dudov auf Unsplash).....	7
Bild 3 - Konzeptionelles Datenmodell.....	10
Bild 4 - Konzeptionelles Datenmodell nach einfacher Chen Notation.....	10
Bild 5 - SQL Datenbankschema.....	11
Bild 6 - NoSQL Schema job_summary.....	12
Bild 7 - NoSQL Schema job_posting.....	12
Bild 8 - Überblick Datenflüsse für Laden & Transformieren.....	13
Bild 9 - Schema von Import-Tabelle für LinkedIn Job Posts.....	14
Bild 10 - Import-Tabelle für LinkedIn Job Posts mit geladenen Daten.....	14
Bild 11 - Doppelte Elemente in Job Skills Tabelle (Gross-Kleinschreibung).....	14
Bild 12 - Datenbankdiagramm nach Datenimport und Transformation.....	17
Bild 13 - Importiertes Dokument aus dem job_posting CSV.....	18
Bild 14 - Versuch pipeline für «job_summary».....	21
Bild 15 - Versuch script für «job_summary».....	21
Bild 16 - Dokumentstruktur vor Transformation.....	22
Bild 17 - Dokumentstruktur nach Transformation.....	22
Bild 18 - Visualisierung des Ausführungsplans vor Optimierung.....	34
Bild 19 - Execution Details Aggregation Basic.....	40
Bild 20 - Risikoanalyse Schwachstellen SQL-Server.....	43
Bild 21 - Script User Erstellung.....	47
Bild 22 - Script User Erstellung.....	47
Bild 23 - Atlas Backup aktivierung.....	48
Bild 24 - Grobe Übersicht des Grafana Dashboards.....	49
Bild 25 - Optionen Panel Types.....	50
Bild 26 - Panel 1 - Detailed View.....	50
Bild 27 - Override job_location.....	50
Bild 28 - Override für Links.....	51
Bild 29 - NoSQL Visualisation - GeoMap.....	51
Bild 30 - Visualisierungssettings Settings - GeoMap.....	51
Bild 31 - Job Location Variable in Grafana.....	52
Bild 32 - Ausblendung State Abbreviation in Grafana Tabelle.....	56
Bild 33 - Transformation zurück ins Root Element.....	56
Bild 34 - Parsen des US Staates.....	56
Bild 35 - Aggregation für Grafana.....	57
Bild 36 - NoSQL Visualisations settings 1 - GeoMap.....	58
Bild 37 - Resultate für Persona.....	59
Bild 38 - Passentster Job für Persona.....	59